目录

vol1 onekeydevdesk选型与实践

vol2:最小编程学习集与语言开发实训

Appendices:

https://github.com/minlearn/minlearnprogramming/：minlearn的一云多端云OS/统一编程栈方案。

这是一套我2016-2020博客的汇编集和实践库,定位与主题为一云多端云OS/统一编程栈方案，分为minlearnprogramming技术文档和onekeydevdesk演示库

《minlearnprogramming》提出了一云多端云OS/统一开发栈的中心思想，及描述了onekeydevdesk的架构原理和实现

《onekeydevdesk》是一个一云多端云OS/统一开发栈实现，以配合我在《minlearnprogramming》一云多端云OS/统一开发栈的想法。

> 什么是一云多端及统一开发栈？
>
> 多场景一云多端OS有多种实现，除了苹果统一芯片华为googlefuchsia统一OS，还有统一boot和libos等虚拟方案，而onekeydevdesk/onekeydevdeskos仅是多场景一云多端OS多种实现之一，采用的是boot和容器融合方案，a boot and container based multiple scene implentmented os and programming stack in one，基本上它是一个基于debian,整合了pve和electron开发栈的devops as desk系统。
>
> onekeydevdesk还同时考虑了一个编程栈。（1）onekeydevdesk是一套企图将统一OS统一应用容器组成的一云多端OS平台做入boot的方案，这种虚拟机，app容器合一的架构特性保证了虚拟到各os的app共享同样级别的virutal appliance基础，（2）同样集成于boot的Electron web栈则保证了桌面/分布式同型，问题域集成和appmodel，再加上full support的开发/可视调试/发布合一， 保证了shortest debug route。（3）在app层，把所有electron stack的APP整理成一个OS的应用库形成单栈应用OS，把所有个人可能遇到的开发用基础云APP弄成ide devable和pve backended，降低了最终开发学习难度。（4）以上成全我们最终的统一开发的onekeydevdesk最简编程实训栈。

架构图：

docs

- minlearnprogramming intro
- minlearnprogramming toc:
- offline pdf edition: minlearnprogramming
- onekeydevdesk intro
- onekeydevdesk docs

blogs

- blogs

本项目长期保存,联系作者协助定制onekeydevdesk/onekeydevdeskos包括不限于机型适配，应用集成等。

DISKBIOS：一个统一的实机装机和云主机装机的虚拟机管理器方案设想

本文关键字：用**hypervisor**装机，用虚拟机管理程序代替**winpe**，带**VNC**的装机环境**,**单机虚拟化方案

在前面xaas系列文章中我们涉及到多种不同的虚拟机/虚拟化技术和装机技术,比如在《发布virtiope》时，我们谈到云主机装机，那是在kvm的guest os里装机。比如在《发布colinux代替os subsystem》中我们谈到host/guest os技术,《发布tinycolinux代替docker》中我们比较了tinycolinux与boot2docker中的iso linux，甚至在《免sandstorm的davros》时我们谈到sandstorm其实也使用了一部分xaas虚拟技术。

其实虚拟机/虚拟化技术/虚拟机管理有多种，最常见的就是virtualbox,vmware这种工具级和应用级的虚拟方案（多用于开发机环境），它属于纯粹的面向单机内的虚拟方案。甚至有WIN10 WSL之类的东西其实就是二个OSSUBSYSTEM同时并列被实现在WINDOWS体内,当然colinux也属这类，它能在一个host os中装多个guest os,只是多用于服务器环境或NAS类环境 ---- 当然，如果严格说起来，它们可以组虚拟机集群只是不被提倡这样做。

集群下的虚拟机技术有kvm,vps这种，KVM是一种hypervisor，即真正的大型iaas用的硬件支持虚拟下的虚拟机,它介于物理机和os之间。有全虚拟化和半虚拟化如kvm,hyperV,xen等等，搭配openstack这种软件可以做iaas。用于运营和云计算环境组建分布式集群。一般非常巨大。vs kvm, openvz这种虚拟OS较轻量，它走的是常规的OS级的虚拟方案----面向在单机OS内部虚拟出多个OS，这符合我们对虚拟机的设想，且它的这种能力也使之非常灵活，可用于多种架构也可同时用于分布式和单机环境下的虚拟。

虚拟化技术要么从使一台机器或一个分布式架构增加虚拟化隔离的能力分裂出子OS或从OS出发，要么从有限的OS子组件虚拟分裂其计算能力。因此也有像docker这样的方案，它有来自内核调用的支持，它基本倾向于用户层虚拟化。它可以做到纯粹的文件系统虚拟化，像一些沙盒环境和绿色软件做到的那样，这是虚拟机中轻量的一种：虚拟容器。

在以上由管理器虚拟出OS的虚拟机技术中，被管理的os有的叫guestos，有的叫virtualos，有的叫容器OS，（你可以完全不用纠结这类说法，只要明白它们所处的明显不同的阶层就好了），而管理器往往以元OS的身份存在和被实现，它发挥的是一种管理硬件资源的能力-----相当于bios管理程序，只不过它面向装各种虚拟OS，且它本身往往采用的是与虚拟OS同样的OS技术:liveos，所以它又像WINPE这样的东西，这给了我们一个新思路：它可用于装机，如果这种管理程序用在给单机实现装机的话，可以说是除了BIOS之外的OS的OS，我们将它称为DISKBIOS。

比如OPENVZ，由于它足够轻量且可以被工具化。那么当它用于普通个人装机，不妨可以一试，这样它就兼具运营目的和实用目的了:比如，个人不但可以装小鸡出售，甚至可以仅用它实现一台机器同时运行多OS，多种不同的OS，且实现远程WEB VNC管理自己家里的那台机器。

来讨论一下可行性吧：

提出一个**live os as winpe**

谈到支持虚拟化的内核，一般就是patch过的kernel，而guest kernel可以复用同样的内核技术，且各种具体虚拟OS层的OS模板可复用同样的linux打包发行技术，OS上套OS是虚拟化技术中最省事的。也有OSv这种重新发明了内核的(据说它每个用户一个VM)。

传统上OS分开系统空间和应用空间，先OS再各种应用，现在xaas时代，OS kernel只不过是我们组建复杂虚拟OS集的最小单元，至于各种应用。。当然处在虚拟OS中，作为元的hypervisor OS中只运行虚拟机管理程序：负责虚拟OS的创建，暂停，停止，资源分配，甚至更多事情，比如远程可视监视VNC，ETC。。

anyway，我们要谈到的实机装机也完全可以是通用最简单的OS套OS的思路：提出一个元liveos，在其中装一个虚拟机管理器用于分配和引导虚拟OS。hypervisor os和各种虚拟OS都采用tinycolinux，linux内核保证能编译支持多种最新MAS设备的能力使之能极好地代替WINPE作这种live装机环境，开机时可以选择让这些虚拟OS同时运行（服务器环境时），或开机装仅运行一个（实机装机时）。。

采用tinycolinux是为了保证元live os的轻量，毕竟装机环境不能太大，要注意host liveos 和guest virtual os是一直在线运行的，这个live os是一直在线管理的，只有guest os是可以选择性关闭，重启的。这是与winpe等装机环境离线运行装好的OS不同的地方。不过这也是它的优点：元管理OS常驻可以实现在线装机。

在**liveos**中配合虚拟机管理程序用于实机在线装机/维护

虚拟机管理器则用openvz来做向linux patch虚拟化支持。再加上vzctl命令行工具或WEB管理器openvz panel等工具（如上所说机器一直开着元管理常驻就可以远程管理且装机/像ghost一样恢复etc..），各种IP池可用端口转发加虚拟网卡驱动来实现。

甚至我们可以增强它，使之还包含paas的东西，，比如像sandstorm一样虚拟出app,app-grains,etc..将资源管理更细粒化，真正极度服务于运营目的。

当然，如果是面向实机，实际上一个虚拟OS独立所有可用资源的情况下，不必这样做，直接装好了即可。

下一文可能就是用在《tinycolinux编译openvz内核》，然后制种一个live tinycolinux，，，再制种一个tinycolinux guest os的文章了。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

DISKBIOS：一个统一的混合OS容器和应用容器实现的方案设想（2）

本文关键字：将**ovz**用于应用级容器设想和**dbcolinux fs**用于**os template**设想,**boot into chroot at system startup,**将**initrd**做成自带**livefs,ovz as chroot**管理系统，**livefs as metafs template to make linux an container os**，为一个**app**配一个**OS**

在《DISKBIOS设想1：一个统一的实机装机和云主机装机的虚拟机管理器方案设想》中我们讲到利用ovz维护单机和云服务器环境统一其装机方案的设想，主要设想就是我们简单地制造出一个装好了带ovz的tinycolinux as pe环境（与硬盘上另一套tinycolinux共存），这样对于单机，我们可以用这个ovz作为pelinux维护后者，通过web方式重装/恢复后者的操作系统，对于服务器，我们可以在资源允许范围内虚拟出多个这样的硬盘系统并用于运营，同样以web/单机的形式管理后者，注意这里pelinux和硬盘linux，虚拟linux都设想为用的同一份tinycolinux rootfs。

想象是美好的，但我们并没有触及到如何使用ovz来实现《设想1》里的东西，在《发布一统tinycolinux，带openvz，带pelinux,带分离目录定制》1,2,3系列文章中我们讲到在tinycolinux上编译ovz和定制/system /usr分离式rootfs的过程在《发布dbcolinux上的cozylight》一文中我们把它称为dbcolinux，也并没有串联起ovz和tinycolinux rootfs。

那么这篇就是增强这些设想的详细内容且再进一步的过程了，来深入讨论一下，那么，将ovz tinycolinux kernel与dbcolinux rootfs结合起来，这样有什么好处呢，最终地，我们希望ovz在dbcolinux里究竟要达到什么样的效果呢？

对于问题1，因为pelinux是常驻的，包含了ovz的pelinux也是普通linux，我样可以以此为模板不断制造新的硬盘操作系统(the system0-systemx)，这样的好处是我们能自定rootfs到/systemx下并boot into chrooted systemx的方式进入它，由此我们做到了OS级的虚拟容器且更轻便。。对于问题2，云服务器的本质就是各种容器和容器化，包括OS级容器和APP级容器，因为OVZ本身就是OS级别的容器所以通常认为它不能用来替docker这样的东西，但想一想docker那种用了分层文件系统的容器它只是将文件隔离在了各层，我们是否可以利用ovz本身的方式将OS虚拟视为应用虚拟，打造一个应用级的容器呢（共享内核，共享rootfs，仅应用容器自身的内容被放在这个容器）？这里的技术可以是：从system0开始，每一个新增的systemx都共享了整个机器的内核和rootfs，仅将应用的数据或程序放在这里，这样资源占用形式其实跟docker分层差不多。这样，一个systemx可以是OS容器，也可以是APP容器，看你怎么看待了，反正ovz使之oneapp oneos的理念做到了极致。

> 那么为什么一定需要这种应用级容器呢？为什么ovz和docker这样方式的ovz要共存呢？
>
> 举个例子，曾存在一种讨论，PC上的多桌面是不是必要的，一帮人认为多窗口多任务有了，多桌面实际上只是在同一个桌面开多个窗口，在窗口间切换即可。但实际上有了多桌面/虚拟桌面来归类这些窗口，在窗口粒度层切换其实也是很方便的，尤其对于一些需要保持多个任务在线，且满屏应用的PC使用者来说，，，，这有点像多显示器。多显示器更进一步，它是虚拟桌面/多桌面不需要切换的一种，扫视即可切换。
>
> 所以，正如多桌面多任务可以共存增益的道理一样，其实ovz这种OS级的容器和docker这种APP级的容器都是需要的（一个共存OS相当于上面讨论的情景中的多桌面，一个共存多容器相当于多窗口）都是需要的。诚如《带pelinux,带分离目录定制（3）》文尾讲的，我们需要使ovz成为应用级容器.

这里的技术细节会是哪些呢？正如上面不断提到的，所有的技术归宗于chroot+定制rootfs template来解释。我们可以把ovz想象成chroot管理器，必要条件是我们依然要在单机或ECS上装是linuxpe带ovz，并以linuxpe自带的rootfs为基础准备一套rootfs作为模板,通过mount源模板rootfs虚拟路径到systemx的实际路径一一对应建立所需具体rootfs的方式。pelinux livefs中的rootfs只是映射到了/hd/systemx，不破坏它作为管理系统和元系统的livemode。然后：我们设想kernel是bzimage,rootfs是initrd，它们二者由grub的linux指令和initrd启动,于是按照linux的启动技术：

我们可以使系统开始启动的时候就进入一个chroot(使用busybox的switch_root，pelinux所在/被清空，将/过继给了硬盘中第一个启动的rootfs — the system0),只是如果这样，那么往后的系统便无法按ovz的方式再开，当然通过其它途径再开是没有问题的不过这样就失去了ovz的意义仅是一个单机dual booting方案,这类似于单机下多OS dualbooting的设想。dualing boot multios but anytime one os running)，然而这不是我们需要的。

我们需要的是：不清空pelinux的所在的liveos，在它里面chroot分化容器：再使用chroot（exec /usr/sbin/chroot /systemx /sbin/init）或ovz的vz enter切换进入容器的方式。这样的话，我们需要为每个systemx下的busybox init进入的路径指定一个x，就如同为kernel提供一个chroot参数一样，这次因为我们是从已运行的meta pelinux中chroot，而我们定制的rootfs只从/system下启动，所以，我们要1，改从kernel到busybox级支持chroot，2，支持/system路径的参数化。

实现了这二者后，那么实际上diskbios可以叫anybios/containerbios了，单机下为bios for disk，容器或ecs下叫anybios/containerbios。

这样，在ecs上开多个OS，和利用chroot在旧安卓手机上安装linux这样的课题可以统一了。我发现老毛子在OS方面造诣很深，win10精简,reactos,还有这个linux-deploy，linux deploy也是那种同时运行二个linux的chroot，可以用usbwifi连网使PC与mobile同处一个局域网，且mobile作为移动nas代替我《一个设想：什么是真正的云，及利用树莓派和cloudwall打造你的真正云中心》提到的树莓派。

还有，我们在前面的文章中谈到couchdb使sql结果也得到持久，，实质为同步。既是一个dbserver，也是一个appserver，重点不是这个，我们还急待使cloudwall与elmlang联合，使cloudwall中inliner/ddoc的IDE环境中可以动态可视调视。。

---

github - minlearn下载,如找不到本文可尝试百度搜索本公号名字！！

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一个统一的**parallel bootloader efi**设想:免**PE**，同时引导多个系统

本文关键字：**Multi loader vs parallel booter**

一直以来，我们都是使用各种bootloader来处理与不同OS的安装和启动任务的，如简单的单OS单loader（比如ntldr这种）,多OS的multi loader(比如grub这种)，这些loader往往与OS和磁盘模式(mbr?gpt?fat32?ext4?)相对。

刚开始的bootloader作为BIOS到OS的bootstraper，而且仅负责磁盘部分。它们工作在实模式，在《在阿里云上安装黑苹果（2）：虚拟机方案研究和可行性参考》中我们谈到，由于BIOS是16位的，与当前32/64的硬件管理程序不搭，所以出现了新的固件规范EFI。后来在新的EFI规范中，EFI自带磁盘BOOT，还负责其它功能比如还带驱动作为OS为硬件认证。管理CMOS,etc…..

> PS：其实EFI指的是固件本身，这些东西是硬件的，它们不可更改，而clover这些东西不是EFI，是EFI的软件部分。EFI整个删掉并不会影响主板是不是EFI属性，不会删掉内置在各硬件中的firmware。这些软件部分的EFI可以驱动硬件（它们另有意义，如做硬件检测），但并不是OS驱动层的驱动意义（实际驱动硬件），实际上EFI中的驱动运行在DEX中不运行在CPU中，而且EFI中的驱动跟OS中的驱动没有承接关系，EFI规范只是被设计用来启动和引导期的特殊任务，并不影响OS的地位。。

无论如何，作为复杂的预处理系统。此时的loader是一个关于EFI的全部生态。完成更多的任务。实际上复杂的EFI也带工具(efi shell,gui,etc..)。甚至可以浏览网页……俨然是一个小PE了。

而我们谈到的PE其实更偏向指群晖webassist,winpe，苹果 recovery这些。其实它们跟正常OS一样，也包括完整由内核组成的系统，也是由上述各种loader启动的。

parallel boot设想:同时引导多个系统

那么既然有更复杂的EFI，而且存在可能将其发展得越来越多高级，那么可以在loader中直接发展Preinstall PE，或当recovery(post install环境)吗，不搭配内核和工具不组建一个OS,不走普通PE的路子，单loader本身可以复杂到如此吗？—— 甚至，能在其中集成虚拟机管理系统吗，这样我们就可以parallel boot同时启动多个OS了。那么，还有没有虚拟机和实体通用的这种loader呢。

这些设想从何而来呢，我们知道，一台OS是独占整个机器的。在机器完成引导化之后。这个OS就独占了机器的全部资源，安装在硬盘上的多系统引导实际上只是multi bootloader，而并非parallel bootloader，如果EFI可以从一套机器硬件组合中按配额来划分它们组成2个/或多个子机器表示。那么，这样的parallel bootloader将不难于实现。因为我们可以在每一个子机器表示下安装不同的OS，实现多个系统的同时启动。

而这些做带来的意义是很巨大的，我们知道，虚拟化从来都集成在系统引导之后，exsi等裸金属虚拟化方案，是在HOST系统里搭虚拟机管理软件hypervisor。它是涉及到OS的。一些工具级的虚拟化软件如virtualbox其实也本质上是这么回事。在实机上，我们从来都是单个时刻只运行一个OS。再在这个OS里各种分裂化。不能以硬件本身作虚拟化，去掉HOST。

最基本的意义。上述方案的成功，可以使得在一个PC上安装多个OS，按常规/而非虚拟化的方式，就能同时使它们运行变得可能。—— 而且不需要涉及到集成一个与OS同质化的PE或RECOVERY。使之变成通用计算机的标配EFI。

市面上有几种特殊的接近这种多样化用途的**loader**

在xhyve中有user space的grub2,在vmlite中有能在实机引导vhd的loader，在《在阿里云上安装黑苹果（2）：虚拟机方案研究和可行性参考》中我们谈到模拟层的OVMF。CLOVER本身也是模拟层的。

这些都可以成为parallel booter inside efi的实现参考。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

兼容多OS or 融合多OS？打造实用的基于osxsystembase的融合OS管理器

本文关键字：兼容**OS**。

相信兼容多os一直是人们的一个梦想，因为在一台机器上跑多个OS是很客观的需求，不光开发，有时一般办公生产都会涉及到在单机上开多个OS的需求。这种东西不光要能用，而且要求要“好用”。我们在前面多次谈到这些。如《reactos》,《colinux,去虚拟化一种文件系统共享的多OS设想》，《dbcolinux利用虚拟机管理器装机》，etc。。

在兼容多系统的发展道路上，有colinux这样的东西，也有wsl这样的东西，有龙井linux这样的东西，还有fydeos这样的东西。也有exsi这样的东西，还有虚拟机，docker，群晖vmm这样的东西，更有虚拟机中的osx parallesdesk这样的特殊品种：”融合os“，当然还有很多。。。

兼容多OS的分类

一般地，附在这些实体，虚拟架构上的多OS，有时有二个，互成host/guest，这是最典型的情况,有时有多个。但基本都有一个管理性的OS，或虚拟机管理程序，或hypervisor,为方便讨论，我们将其区别称为（管理性OS和用户性OS）因为如果视管理虚拟机本身所在的OS也往往是一种独立OS，是“OS的OS”，用户OS主要是子系统的话，那么其中运行的子OS是平等的。用户主要使用的，就是运行的各种子OS。------ 如果管理性OS和使用者OS都是主体，就是前一种，如果使用者OS是主体（仅对用户可见），就是后一种。 ——— 这样讨论就方便多了，所以首先，“兼容多OS”基本可以按这二类归类：

1，从平台和架构上来，有的是面向实机，裸金属装机，如exsi，独占机器，有的是面向计算意义的。除exsi之外的都算。他们只占据该计算架构中的某块资源。如云主机架构中的各种OS。有的是硬件虚拟化，（有的是hypervisor，受硬件支持,有的是OS级的虚拟化。只要OS提供了分裂子OS的功能，就可以在一台机器上跑多个OS)有的是工具层面的，如各种虚拟机程序vmware,virtualbox,etc..。

2，从相互归属性上来讲，colinux,虚拟机oss,fydeos+3 oss,docker subos都有鲜明的host/guest特性，因为我们平时不但要管理这些子OS，也在主OS中工作，同时接确这二者，这种主要是双OS，而exsi，vmm,云openstack是一类，我们基本，或很少，不能、或无须接确到上层。我们把后者称为平等多OS。

来深入继续归类：

从性质和技术来分，有的是经典的内核增强技术。如龙井,wsl,colinux，都是从严肃的内核改造/再造开始，而vmm,docker,fydeos这样的东西，虽然有内核定制，但都是轻量定制了的内核加多样化了的虚拟管理程序，和不同的rootfs为主。而虚拟机则纯是一种软件层级的再造方案+（可选的硬件直通能力）。

从生态上来分，hypervisor类的多OS往往有多种不同的OS。而OS级的虚拟化，往往都是一种OS的变种，互通性容易些。

最最重要的分类问题来了：

在日常工作中。那些我们平时需要频繁用到多OS的情况中，哪些分类是起决定作用的。即“好不好用”这个最终问题，才是决定用户选择的方案分类问题，这个分类问题就是性能和最终体验问题：

但事实是，我们目前很难找到一种保持原生OS体验，性能不打大折扣的方案：从性能上来讲，虚拟机是最经不过考究的，即使加了硬件直通性能也大打折扣，wsl，colinux这种基本CPU能力都能直通host的次之。从体验上来讲，也许只有parallesdesk最好用（也许融合，不打破才是最省事最好用的目前方案，虽然其它的方案也在进行中。。但目前唯有PD最实用），wsl次之，其它也都是半成品/实验品，可是PD它只存在于osx，而且收费。

那么，可不可以将PD做成类exsi的东西呢？

设想：打造基于osx base system的实用windows/osx融合OS管理器

这种方案就像在winpe上集成virtualbox或vmware一样，然而选择osx是因为PD在osx上的体验最好。我们可以定制osx base system，将PD装在osxpe上，然后开机启动。在其中安装并列的osx和windows。这二者的互通和融合通过主OSXpe中集成的finder和工具栏docker来进行，必须保证osx base system的管理者OS身份。因为它足够小。

或许你可以按《OS X Recovery Partition: Customizing With Different Apps》进行得到这样的一个东西。如果还能将它硬件化到nano itx小主机就爽了，做成黑苹果更通用。

————

从《发布一统tinycolinux，带openvz，带pelinux,带分离目录定制》系列到《利用hashicorp packer把dbcolinux导出为虚拟机和docker格式》，再到《打造一个Applevel虚拟化，内置plan9的rootfs:goblin》，我们贯彻的选型依据之一，就是：哪个兼容多OS方案是最好用的。最后由于实现的需要。我们不可能采用PD这种，最终屈服于从linux加OS虚拟化级去定制，而且我们需要更偏向开发相关，而不纯粹是装机的那些问题，所以就有了goblin。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一种虚拟boot作通用bootloader及一种通用qemu os的设想

本文关键字：自带**bios**的**boot**，自带虚拟**BOOT**的**BIOS**,**packer**下以**tc+gcc**方式编译**Cloverefibooter**

物联和云越来越流行了，OS的融合潮流变得不可逆转，在多场景，一云多端OS方面，比起鸿蒙，fuchsia，pvelinux，libos这些从OS层面努力的方向：虚拟化方向（其中鸿蒙fuschia是重新发明OS而pvelinux这些不是），m1芯片下的macos/ipados/ios融合，更像是从硬件开始作为起点的方向。而从boot虚拟化方向的OS工作，如coreboot，更像是整合方向的融合：不重造OS甚至拉高替代OS的一部分。

在前面《Boot界的"开源os"：coreboot，及再谈云OS和本地OS统一装机的融合》我们讲到，开源coreboot是固件界的"OS"（libreboot努力使之完全没有闭源成份），linux本身是一个开源，但真正伟大的"非玩具"超规模级现代OS。背后可以没有一个公司，却连接起所有公司和组织，个人真正为它维护，想想西方难于组织起全国民级搞防疫，在现在来看linux的成功也依然是难想象的，还有gnu tools。---- 本文提到的coreboot也是这样一种类似linuxkernel复杂度，由一群来自社会各界的人联合，以松散方式完成但实实在在不可思议发生了的作品。

在《云主机装黑果实践(9)：继续处理云主机上黑果clover preboot问题》中，我们讲到在OSX 10.11.6+xcode7.3.1下从低版本编译clover至r5068，在那文我们讲到，clover的编译很复杂，clover在edk2的框架下被编译，edk2/edksetup.sh被设置为souce到clover的ebuild中起作用。edk2中有一个basetools集，里面建立了一个类似cmake之类的复杂体系，当你调用clover/ebuild.sh时，会分解为类似build --skip-autogen -D USE_LOW_EBDA -p Clover/Clover.dsc -a X64 -b RELEASE -t GCC49 -n 3 的具体命令，然后调用edk2来编译，调用edk2 basetools中的生成框架逻辑在build/中生成toolchain命名的框架文件夹，如RELEASEXCODE5，然后调用basetools中来编译。

> edk2开源方式代替intel的uefi实现tianocore，coreboot可接入它以payload，，clover使用edk，主要使用edk中的duetpkg和omvf而来的(DUET is designed to boot UEFI over a legacy BIOS.),而omvf是为支持qemu而存在的。
>
> 事实上到这里为止，该如何理解clover? 整个edk2你可以理解为可以刷进固件的bootloader，也可以在clover中，配合rEFIt_UEFI被当成booter+firmware的组合工作，这就是一种"自带bios的boot，自带虚拟BOOT的BIOS"，它可以统一云主机实机装机，实现通用虚拟云主机BOOT(虚拟机就应当用虚拟boot)，也可以实现在虚拟boot中集成各种工具，写进实体机固件或仅放在硬盘配合boot在启动时发挥作用，实现诸如《一个设想，在统一bios/uefi firmware，及内存中的firmware中为pebuilder.sh建立不死booter》作不死booter之类的作用，当然，也可以集成轻量虚拟机，实现类似headfirmware之类的作用，还可以集成oskernel，实现linuxboot之类的作用。

注意：不要用在白苹果上使用clover会破坏固件。其它机器不会。

在linux gcc下编译clover r5068

在《云主机装黑果实践(9)》中我们是用xcode5编译clover的，xcode是适配具体osx版本的，因此现在,我们探索在gcc和tinycorelinux下编译clover的方法：

我们首先讨论在osx上cross gcc编译clover的方法，因为r5068在osx上除了xcode还有cross的gcc8,gcc4.8.gcc4.9可用(在/basetools/conf/tools_def.template中定义，它还有一种GCC5支持但是clover的脚本中却没有相关buildgcc5.sh)，，r5068是https://sourceforge.net/p/cloverefiboot/code/5068/在2019-09-04之后的tag，我们配合使用的edk2是edk2-code-stable201908-trunk。由于gcc49适合01.02.2016 it is possible to compile full Clover package only with gcc-4.9，所以对于r5068是过时了，实际上xcode5也没有能完全编译r5068到成功结束我们只是取得了clover就了事。在OSX 10.11.6+xcode7.3.1下cross compile gcc49实际上也不成功。gcc4.8肯定也没得用了，

那么gcc8呢？实测在osx10.15.7上，配合xcode12 command tool cross compile gcc8然后编译r5068是成功的，而且一次成功（中间会有一些python error但不影响总体）到最后，不只是止于生成到clover.efi。

我的方法是先编译出cctools-895.tar.gz，nasm-2.14.02.tar.xz这二个到~/src/opt/local(cctools 895,921。前者在11.06下可通过，后者通不过。在osx10.15上你当然可以尝试921)，然后buildgcc8.sh，成功，最后TOOLCHAIN_DIR=~/src/opt/local ./ebuild.sh -gcc49一定执行cp -R Patches_for_EDK2/* ../进行patch一次(因为,ebuild.sh中的patch代码被注释掉了。)，之后用全新的目录来执行（不能用上一次xcdoe5或patch过后的结果覆盖执行），对的，虽然是gcc8但指定了toolchain_dir环境变量依然可用-gcc49 或-t GCC49参数。如果碰到permission denied，把export TOOLCHAIN_DIR=~/src/opt/local放到ebuild.sh中然后sudo ./ebuild.sh -gcc49。

对于在packer下以tc+gcc方式编译Cloverefibooter这样的实践，下面是我的一个初步尝试。

基于《一个fully retryable的rootbuild packer脚本,从0打造matecloudos(1):实现compiletc tools》中的packer模板和buildtools脚本的开头部分，结合buildgcc8作修改成buildrom： (clover的buildgcc8.sh是我在packer序列脚本中追求的，"all slient compiling"和"判断连续命令最后一条是否出错"的典范。而且，3,脚本与大文件分开，源码包在外部下载，以及"4.fully retryable"。注意到它还使用了bash的eval。):

```
1，开头部分处理：
TC_TGT:删掉
TC=/mnt/sda1/tmp/tc，由/mnt/sda1/tmp/tc改为/mnt/sda1/tmp
PATH处理：PATH=/usr/local/bin。。。。。（这个路径下有tc compiletc中的bins，及busybox wget,如果不让path /usr/local/bin 优先/usr/bin,会让wget.tcz与busybox wget混淆（后者不支持ssl））
mkdir -p /usr/local/etc/ssl/certs/ (这个ssl fix可以整合进packer模板或iso镜像，wget,curl需要)
su - tc -c 'tce-load -iw -s compiletc wget curl patch rsync python3.6'

然后是从buildgcc8中提出来的部分（这个toolchain用于处理basetools中的C能正常编译从而编译整个clover），作处理：
（注意export与免export定义出的变量的不同，前者是可以进入环境变量的）
```

```
export DIR_MAIN=${DIR_MAIN:-$TC/build}
export PREFIX=${PREFIX:-$TC/tools}     (这样就组成了/mnt/sda1/tmp/src,build,tools的合理结构)

export MAKEFLAGS="-j 2" （这个一定要保留，可以缩短至少十几分钟的时间，我们packer模板是设置了双核支持的）
export DOWNLOADPREFIX=${DOWNLOADPREFIX:-http://www.shalol.com/d/src/buildmatecloudos}  （里面会是包没有文件夹层次）

2，然后是主体部分：
download source,update clover的patches到edk2，basetools很娇弱。如果你碰到py multiprocess connect broken pipeline就是fresh的源码没有做好，不够干净。
然后修改Clover/ebuild.sh加入：
SYSNAME=Linux
TOOLCHAIN_DIR=/usr/local

3，最后
在linux下进入Clover直接sudo ./ebuild.sh编译clover，需要用-gcc53或-t GCC53而不是49，因为在tools_def中,49只用于osx或linux中通过cross compile命名为x86_64-clover-linux-gnu的新toolchain。
你可能会遇到all warnings treated as error,修改tools-def中的gcc53相关参数即可
buildTime=$((stopBuildEpoch - startBuildEpoch))改为这个去掉expr。获得执行时间，你可以决定startBuildEpoch的放置位置，以自由决定包含在其中的compile xxx || exit 1统计的函数执行量，以及时间跨度。

4，补充

如果你想重新考虑进buildgcc自己编译toolchain，那么还要加上以下：
.......
export BINUTILS_VERSION=${BINUTILS_VERSION:-binutils-2.32}    (bintuils不改了还是用buildgcc8的吧)
export GCC_VERSION=${GCC_VERSION:-9.2.0}  （8.3.0,9.2.0在osx下都可crosscompile编译成功，在tc11下仅920成功）(gcc依赖库buildtools和buildgcc8都一样) (linux下不需要cctools912仅需要nasm2.14.02)
......
export BUILD=""
export HOST="x86_64"  (接下来有介绍为什么这样设置)
export TARGET="x86_64-pc-linux-gnu"    （这个就代替了上面的TC_TGT，注意这里与原buildgcc8 cross compile用的x86_64-clover-linux-gnu的不同,如果你指定target=x86_64-clover-linux-gnu，只要target名字与build/host不同，虽然都是x64下for linux 64，但gcc编译脚本都将其视为某种cross compile，host = target，但是build不同，叫做crossed native；build = target，但是host不同，叫做crossback。三者都不同的话，叫做Canadian cross。那么32到64或64到 32不算cross compile？我们在前面文章利用hashicorp packer把dbcolinux导出为虚拟机和docker格式(2)其实实践过。）
......

ExtractTarball改为DownloadandExtractTarball ()与前者整合，这样可以下载一个编译一个，不用全部预下载。在local package=${tarball%%.tar*}和tarball="${DIR_DOWNLOADS}/$tarball"之间把原来属于download的逻辑写进来即可，注意到由curl改为了wget:
if [[ ! -f ${DIR_DOWNLOADS}/${tarball} ]]; then
        echo "Status: ${tarball} not found."
        wget --no-check-certificate -qO- ${DOWNLOADPREFIX}/${tarball} > ${DIR_DOWNLOADS}/${tarball} || exit 1
fi
把所有涉及到extract的地方改为新函数名。把#mountRamDisk都注释。放入CompileNasm 2.14.02的函数。放入buildgcc8的CompileLibs函数。

不需要CompileBinutils。在tc11 gcc930上native编译gcc。完全可以像编译普通程序那样单个pass命令native编译的gcc，直接使用native上的binutils，不需要涉及到even native compile binutils。你也可以在编译出新gcc之前或之后native compile新的bintuils，这样系统上有二个bintuils或一个到二个gcc，对于待编译出的新gcc，buildgcc8中的nativecompilebintuils(被注释)是设想先用新编出的gcc再来编译出这个bintuils的。期间需要pathmunge $PREFIX/bin一次 （利用路径下新出的gcc）。所以实际上，nativecompilebinutils为新出的gcc所用（你可能需要pathmunge让gcc用上新binutils），是不必要的。

重点部分GCC_native () ：
local cmd="LD=/usr/local/bin/ld AR=/usr/local/bin/ar RANLIB=/usr/local/bin/ranlib ${GCC_DIR}/configure --build=${BUILD} --host=${HOST} --target=${TARGET} --program-prefix='' --program-suffix='' --prefix='$PREFIX' ......
这里build为空，host=x86_64,不加这个，gcc configure不通过，提示whether we are cross compiling error: cannot run C compiled分析原因应该是它决策不出host参数，不知这是一次native compile还是cross compile，因而不知去哪里找bintuils中的ld，虽然/usr/local/bin就在PATH,ld就在PATH中，虽然gcc/config.guess也会自动检测你的native tripe组参数，但是直接用build=host=target=x86_64-pc-linux-gnu也会提示上述出错信息。
但是直接把bintuils喂给它，可以bypass上面所有这些判断（这是gcc编译脚本的bug?），故host用x86_64,并直接给出命令变量LD=/usr/local/bin/ld AR=/usr/local/bin/ar RANLIB=/usr/local/bin/ranlib，以及target。
GCC_native()中TOOLCHAIN_SDK_DIR相关的逻辑可以完全删除。只是这条ln -f "$PREFIX"/bin/gcc "$PREFIX"/bin/cc最好保留。
--program-prefix='' --program-suffix=''是因为上面ln -f和edk2 basetools的tools_def都指定toolchain为gcc
```

对集进**pebuilder**的设想,及实现通用虚拟云主机实现装机**BOOT**，和利用它打造不死**boot**

我们可以把pebuilder.sh现在的镜像(黑果黑群黑win)都设置成一个单独的rom区，在peubild.sh中的那个preseed中就分区，把这个通用boot放进去。然后所有的OS镜像都放在vda2，打造通用qemu os，把常用OS都qemu云guest os化。这样的一个ROM可以在实机上弄成ISO版本或EXE安装版本或U盘版本供实机使用。

最后，还是那句，注意不要在白果上使用clover这类虚拟固件，因为它会破坏实体白果的固件。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

Boot界的”开源os“：coreboot，及再谈云OS和本地OS统一装机的融合

本文关键字:**firmware as service, linux as boot**，，**boot as infrastructure**，**real cloud os and app**，云操作系统选型，实机装机/裸金属通用的云**OS**。**hypervisor/vm**通用的云**OS**。

Firmware编程是计算机（软件）编程的最初一种形式，也是OS的雏形。没错，它是针对硬件的”软件“编程。这个层面最初形成了我们看到的OS。最明显的层面就是OS中的HAL层面。

历史上。bios是这个规范，如今，UEFI，各种各样复杂的loader也出现了。甚至还有linux as loader，即把linux作为开机出厂程序。用户装的OS作为第二级OS。

那么这种情况合理吗？其实所谓UEFI只是机器引导之后的一个特殊执行体而已，这个执行体在没有进入OS之前，可以是其它任何软件。甚至一个极简的linux rootfs。所以，loader as uefi跟loader as linux没有任何区别-它们共享一个相同的最基础部分:就是那个机器引导，之后同质之前同源。

这样有什么好处吗？我们知道UEFI中有很多驱动。UEFI的作用就在firmware中提出一种新的地址模式和能够使用现代语言的编程方式来处理硬件的改变。可是，这不只越来越复杂的UEFI能办到。在一种现代OS中这样的东西中也能办到，而且天然就是现成的，不用重造轮子。比如linux，它类UEFI本身就是高级语言的地址空间产物，它的宏内核设计使得它的驱动天生就是内含的。其中的很多drivers也不用在UEFI中重写呢。而且它够小。它作为一级主板预置程序可以发挥像WINPE，LINUXPE,syno web assist,osx recovery一样的作用，而且，除此之外，os还能兼能提供现成的工具，如busybox等。这样，linux loader既是recovery，也是firmware.等等，它还有其它作用（稍后就说到）

为什么会有这种需要和改变呢？这种需要的出现是因为cloud computering出现了。它要求对现有的PC，裸金属机器。进行新的一轮upwards streaming抽象。使OS的概念重新分层分级，包含单机，实现，云主机，各种寄宿有OS的地方 —— 形成一种多用的云OS，既然这样，何妨让Cloud Computing也配一个抽象Firmware不就好了么。。

所有会有我们今天谈到的firmware as serice，coreboot就是相关的产品。

Coreboot：抽象firmware as serice,云host os

它的前身是linuxbios，不过后来变成了现在的coreboot,libreboot是它的一个极力去除闭源驱动的开源版本。

> coreboot performs a little bit of hardware initialization and then executes additional boot logic, called a payload.
>
> With this separation of hardware initialization and later boot logic, coreboot can scale from specialized applications run directly from firmware, operating systems in flash, and custom bootloaders to implementations of firmware standards like PCBIOS and EFI without having to carry features not necessary in the target application, reducing the amount of code and flash space required.

记得当时刷chromebook用的就是这个coreboot。它对各种设备，包括云主机都有payloads.

而linux as payload更有趣。

> Two aspects emphasized by proponents of Linux-as-a-payload are the availability of well-tested, battle-hardened drivers (as compared to firmware project drivers that often reinvent the wheel) and the ability to define boot policy with familiar tools, no matter if those are shell scripts or compiled userland programs written in C, Go or other programming languages.

所以，接上面“稍后就说到”：它不光是advanced firmware, recovery，还可以做虚拟机管理器，这样买来的预置了linuxbios的机器永远不怕坏了。因为可以在其中安装多OS。而且这些OS是linuxbios的二级OS，它们是用户OS。只是一些类似虚拟机guest os的主机（所有的guest都是平等的，而且可以parallel booting）。Splashtop产品就是这样的一种设备。Avatt:all virtual all the time就是基于coreboot，在linux里提供了ovz, kvm虚拟机管理器，可以允许客户安装自己免坏的OS，类似exsi，但是可以作为日常用机而不是服务器。———— 当然，这种firmware是需要刷机的。

而正是这个虚拟机管理器功能，就带来了一个更为巨大的意义：——— 我们一直知道，虚拟机，虚拟OS，虚拟化是云计算的主要概念和手段，虚拟机的运用放在今天实在太重要了，因为它不但是装机/运维问题，开发/devops问题，virtual appliance/appcontainer问题，也是云计算和云开发的基础课题。云主机一般是虚拟机。但实机，自从coreboot它使得本地也可以更方便地云化，从此也可以通过虚拟机装机。而不必仅限于传统的host内使用guestOS的方式和结构。—— 作为一种从最开始处：从类传统PC的BOOT处，对云OS装配firmware，的”OS”，它已经天然是个“实机云HOST OS”了。

unikernel和云guest OS

那么guest os呢？还要有unikernel+guest os的设计。

什么是guest os，就是我们通过云在其中运行应用的那类OS，宠统来讲，我们最常接确到的“云OS”而不是什么hyperior，其实我们之前文章一直在探讨，什么是云OS，在不同的层次上，有很多OS，甚至APP都称自己为云OS。

比如，对于装机和开发，在谈群晖的系列文章中时，我们谈到docker as devops，qnap的qvpc，群晖是一种云OS，对于云APP，lamp是一种云OS(with appstacks)，cloudwall是一种云OS (with db and sync only)，Tumblr 它也是一种云OS，因为它提供了一个聚合功能。

所以我们最后结结实实地总结得过一句:云OS从来没有自己的专用OS，云APP也没有本质上专用的云APP。都是现在的本地OS和本地APP的分布式的叠加。所有这一切，都造成了抽象的过度堆彻。

那么现是在思考解决这样问题的时候了。拿webos来说，Webapp,webappstack虽然是打洞主义，然而它的碎片化特征却是独一无二的（超链本身就是一种碎片化,webapp是page,是applet，非常碎片化，通过超链的表现相连）。现在，OS的分布化和碎片化也要发展起来了。

基本的思路就在这里，基本的解决问题的产品就是unikernel。将OS也像APP一样碎片化。就有了我们即将谈到的云OS（关于云APP。也有碎片化的云APP）。而unikernel。它对传统OS也进行了裁剪和重抽象，它也通过碎片化解决了抽象的过度堆彻。

Unikernels are single address space library operating systems. An application compiled into a unikernel only has the required functionality of the kernel and nothing else. Such a stripped-down kernel makes unikernels extremely lightweight, both in terms of image size and memory footprint, and also can lead to security benefits due to a reduced attack surface. There are many such lightweight unikernel implementations, e.g, LING, IncludeOS, and MirageOS. LING’s website takes 25 MB of memory because it runs on top of the LING unikernel. IncludeOS’s base VM starts at 1MB and a DNS server running on MirageOS compiles into a 449 KB image.

（others : 容器OSCore os, smart os,vOS, etc..）

Unikernel往往去掉了传统OS kernel中的某些hal部分。因为它是hyperior的和hostos的。而且它形成了一个操作系统配一个进程的构架。

而其实，一个操作系统一个进程，内核和应用都在一个地址空间内。一个APP配一个OS，是云计算中理想的Guest OS。这有点像user mode linux和colinux,这样的一套OS+APP就形成一个Virtual Appliance，谈到appliance，这是虚拟化自己的APPMODEL，类似web的web app,native的native app,mobile的mobileapp，virtual appliance最初的形式是vmlite的是appliance，Microsoft vpc的xp融合模式，就是不透出guest os，OS作为守护，仅把融合在host os的app透出来。—— 所以，它是对virtual appliance的增强。

Unikernel也是对容器和容器OS的强化。也是真正的applvl的虚拟化。见《打造一个Applevel虚拟化，内置plan9的rootfs:goblin（1）》。因为它不是用容器管理器的方案来处理host/guest关系的，而是实在的把OS包含其中。

也是对appmodel的增强，一个APP配一个OS，就没有了server, 它是对servless app的增加。

可以说，配有coreboot的云主机+配有unikernel的guest os，是包含了对云OS装机， devops,applicance 容器。的4 in 1所有路径上的解决方案了。

——

无论如何，以后统一用虚拟机装实机/虚拟机，云主机，开发，作容器，的梦想可以融合了，可以实现了。这是我们前面诸多文章的目标。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

Linuxbootlinux as UEFI,linux over UEFI

本文关键字：**linux as UEFI**，**linux over UEFI**，**qube-os**

在前面《将虚拟机集成在BIOS和EFI层，硬件融合的新起点：虚拟firmware，avatt的编译》及前面一些关于coreboot的文章中，我们提到coreboot是一种开源的boot+firmware，它是从硬件加载到firmware（能够存在于flash的其它非hardware initial必要部分）全包的，coreboot聪明地分开了这二者，将firmware部分视为payload。我们还提到coreboot最初的payload就是linux，除此之外，它还支持UEFI，任何可以在硬件加载完后可执行的东西。

这就是说，coreboot的硬件加载部分依然是需要自实现的。面对一块新主板新平台，开发上，除了软件部分（可以用QEMU调试），硬件上它要求刷机有时甚至要求烧录和焊接技能。—— 这对大部分工作来说都是很麻烦的，写好了也不便于安装。

而UEFI是一种开放的firmware标准（虽然大部分UEFI是闭源的），而且适用于现大部分INTEL平台，为了在平台上开发使用UEFI，照样需要完成硬件加载和firmware衔接工作，和重点的fimware编写工作（DXE），TianoCore是一种开源的UEFI，coreboot的硬件加载部分可以结合tinacore使用。视TianoCore为payload。—— 虽然还是有很多工作，不过，在这种方案下，coreboot的硬件加载可以直接采用该平台开放的部分，然后接入自己的UEFI逻辑，而intel vendor UEFI firmware平台随处可见，是用户可得的。

那么对于我们追求的Linux as firmware，有没有一种更干净的连接起硬件加载部分和firmware的机制呢？并在UEFI上使用 LINUX的技术呢？

这就是linux as UEFI或linux over UEFI。Linuxboot就是这类技术实现的代表

LINUX BOOT

The LinuxBoot project https://github.com/linuxboot/linuxboot (formerly NERF) is a collaboration between Google, Facebook, Horizon Computing Solutions, and Two Sigma that aims to build an open, customizable, and slightly more secure firmware for server machines based on Linux. It supports different runtimes, like the Heads firmware or Google's NERF.

Unlike coreboot, LinuxBoot doesn't attempt to replace the chipset initialization code with opensource. Instead it retains the vendor PEI (Pre-EFI environment) code as well as the signed ACM (authenticated code modules) that Intel provides for establishing the TXT (trusted execution environment). The LinuxBoot firmware replaces the DXE (Driver Execution Environment) portion of UEFI with a few open source wrappers, the Linux Kernel and a flexible initrd based runtime.

1,它首先是一种UEFI实现，跟TianoCore一样，用来代替闭源驱动，相当于coreboot+uefi payload。不过它的工作比coreboot+uefi payload更少。 2,由于它也是全包的解决方案,所以也是一种coreboot代替品，而且它面向使用linux这点与coreboot也不同，coreboot中并没有使用linux当uefi的部分。 3,linuxboot+runtime可以采用cb的硬件加载，也可以使用自己的硬件加载，其linux kernel部分是不变的。 4,它可以实现在硬盘的UEFI分区上直接置换firmware。这样至少对于安装是很方便的。 5,以上其实最终实际上也符合我们在《将虚拟机集成在BIOS和EFI层，硬件融合的新起点：虚拟firmware，avatt的编译》文提到的理论基础，即：linux本身与UEFI技术中的驱动本来就存在重复工作，所以wrapper一下得到的结果也能很好融入UEFI。而linux作为开机预加载环境，可以实现kexec直接加载linux（kernel inside bootloader，不需要专门的bootloader），如果在其中加入虚拟机管理器，可以实现前述我们提到的并启和多OS运行。甚至，虚拟化firmware，可以让黑苹果，黑ios这样的东西变得现实 ——— 这些我们前面都有例子。

Linux boot唯一需要你定制和附加的，就是initrd制作部分，也就是它要接入的runtime，vs Coreboot and Coreboot payload，linuxboot将定制firmware的工作压缩到了仅需要定制rootfs的程度。

Linux boot runtime

Linux boot支持Coreboot runtime（采用其硬件加载部分），linuxboot其前身NERF也是一种runtime，不过像coreboot一样，linuxboot发展到了支持多种其它runtime。如除了coreboot，还有Heads firmware，https://github.com/osresearch/heads

还有这里提到的，https://github.com/u-root/u-root

u-root is an embeddable root file system intended to be placed in a flash device as part of the firmware image, along with a Linux kernel. Unlike most embedded root file systems, which consist of large binaries, u-root only has five: an init program and four Go compiler binaries.

基本上,u-root非常接近我们要得到的goblinux:go,busybox,linux as hyperisor inside boot, linux as user rootfs 。一份及早封装到上游的抽象打包linux for init programmers。

———————

在linuxboot的runtime中集成虚拟机管理器，就得到了我们前面提到的并启虚拟化支持。Linuxboot它主要面向服务器主板作虚拟化。Qubes os就是另外一种主要面向PC主板虚拟化的。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

为你的硬件自动化统一构建root和firmware

本文关键字:**Buildroot**，**coreboot as Buildrom**

前面《基于虚拟机的devops套件及把dbcolinux导出为虚拟机和docker格式》系列文章中，参照LFS,我们曾手动自建过交叉编译环境和linux，完成了包括gcc as 开源toolchain,linux as os kernel，busybox as开源os rootfs,各种libs的所有手工化构建步骤，不过仅限于在PC上，和x86之间进行，都没有考虑进我们现在除PC外的其它使用linux的硬件，—— 这就是嵌入式平台及特化PC，它们甚至是我们目前硬件平台中的绝大部分，如路由器和手机(nas,laptop也算)，—— 虽然说linux对嵌入式都有支持，构建的流程大同小异。它们跟PC平台相比，对构建的需求还是有点不太一样：

首先，但嵌入式硬件多样，有着各种各样的非x86 CPU和cross compile需求，采用的一些版本也是PC上的特化版本，如libc。libc是程序与内核交流的媒介与用户编程接口，，嵌入式上往往使用ulibc.

而且，嵌入式中，firmware跟os,甚至app的界限非常不明显，往往高度耦合，前面我们在《Boot界的"开源os" : coreboot，及再谈云OS和本地OS统一装机的融合》中说到，firmware编程是OS前面的那块软件编程，嵌入式平台的firmware与os往往高度偶合，往往firmware直接就是boot loader，使用uboot这种方案。程序要求也不一样，嵌入式往往就一到几个守护程序持续运行，简单的自启脚本即可，除了像带需要带安装APP支持的智能路由器需要像openwrt这种拥有类PC包管理的之外。

而且他们使用构建结果，安装系统的方式，PC跟嵌入式也不一样，它们需要rootfs尺寸要裁剪非常小，它往往是一个静态的firmware或rootfs镜像，直接刷到嵌入式硬件上的一块静态的固定存储量的ROM上用于承载OS。

因此针对嵌入式的构建，这些支持往往分别分散在各个厂商的手中。

—— 所以，如果有各种嵌入平台，还要考虑进coreboot as 开源firmware，，为什么不把for pc,for embededs，for rootfs,for firmware编译和安装统一考虑进这个综合构建呢？

—— 只是，因为这里面的要处理的工作和复杂度实在太高太大量（这已经是制作一个linux发行版，甚至硬件的工程量了），难于用一个框架来述说。你看LFS就需要一本书。这套构建工作最好是自动化的。要用一套脚本来处理的话就最好了。

—— 幸好，build firmware和build root其实是一个很相近的过程，都是从交叉编译工具的构建和驱动开始， (其实，区分这二者也很简单。一个是硬盘上的OS，一个是没进入OS之前的那个硬件初始化（firmware）中的OS)。这二者统一，自然而然。

市面上终于也出现有这类工具，Cross-ng和Build root，OpenEmbedded,openwrt，就是这样的工具/环镜。如cross-ng只处理交叉构建，buildroot就是整合toolchain（它也可以使用external toolchain built from cross-ng）,kernel,bb,rootfs,libs的工具,只限于直到生成打包的文件系统。不往发行版本一步，coreboot,firmware的OS，buildroot有点像coreboot as buildrom，。openembeded可以直接用来构建发行版。openwrt也跟openembeded一样生成发行版不过它大都针对路由器。

这里面最大的好处是，1，当这类工具统一在同一套脚本上，它可以实现all in one,self contained，所有的部件保持同一风味，比如linuxkernel,bb是同flavor的，它们使用kconfig，且将编译过程中所有涉及到的大件都menuconfig化，如果libc也可以，coreboot也可以，不是更好么。在build root内部就是这样做的，它增加了glibc-menuconfig,linux-menuconfig这样的东西。使得脚本更统一，更in flavor。2，both for embed and pc,buildroot本来就是for嵌入式的，像build root同时支持了ublic和glibc库和各种mainboards,cpu。这样就给了我们融合多套硬件制作同一套软件镜像上的统一构建用的codebase,，而coreboot也是for 嵌入式的，我们看到其soc下有很多欠入式,广大嵌入式的构建工作更应应用coreboot的统一设计而不是分散厂商的。— 这些可以用同一套codebase生成融合硬件使用的镜像不好么？。3，可以统一使用，其写好的firmware可以在虚拟机中，如qemu中测试（qemu因此也是firmware和build root眼中的一种抽象主板）。Qemu可以单独喂给一个firmware,也可以同时结合喂给一个firmware+OS的组合。4,可以统一让融合硬件使用同一个codebase，谈到硬件融合，我们前面有很多例子，虚拟appliance+一个有融合作用的HOST as shell也可以解决硬件融合的很多事情，比如PD的融合模式，而现在，像一些激进的硬进如purism,librem(一个可以发展装有linux的手机。类ubuntu touch一体手机电脑融合OS的硬件产品)都ship了Coreboot

带着这些观点，我们来研究build root和coreboot的脚本

其中的所有原理，都是我们在《基于虚拟机的devops套件及把dbcolinux导出为虚拟机和docker格式》曾手动自建交叉环境和编译之后kernel,bb，libs,所涉及到的那些，Coreboot和Buildroot的编译都能在一台unix派生系的host上完成，要求host上一个toolchain加少量必要的库就差不多了。因为它们都面向构建基础套件。

Build root

Buildroot可以在本机上直接构建，官网提到GCC要4.4以上，HOST要用常见unix派生系os。。当然最新发布的buildroot已经用上了vagrant虚拟机，这样就省了好多工作(如果要在host手动安装编译需要的东西，参见那个vagrantfile中要安装的包即可，)。

我们仅讲解自动的。在vagrant中，我们发现最新的buildroot都使用到gcc7来编译了。buildroot(2019.8)大约有10000多个package(实现一个发行版直接可以拿来作为仓库),每个package下面都有.mk和.config.in，package源码都是一次编译即时下载到src/dl的。你也可以make source把它引用的packages源码下好，里面有board/qumu,生成的output中，output/中良好排序着交叉三元组：build, target, host中的临时文件，还有我们要得到的最终的东西，images，可以对比《基于虚拟机的devops套件及把dbcolinux导出为虚拟机和docker格式》来理解和使用。

这里需要搞清build root涉及到的devops工具的关系和区别：

vagrant和packer都是hashicorp的Devops七件套之一。packer是提供一个初始镜像，控制虚拟机如VB，PD（我们讲过PD和VMWARE方面的例子），自动化输入。inplace生成新镜像。vagrant是命令行版本的虚拟机管理器。它可以管理VB，PD等（provoders），如果有可用镜像box，可在vagrant配置文件中写明，然后用vagrant up起来给任何一个provider使用，up后可在虚拟机内干任何事情，关机后不导出镜像

它们的关系才是它们的重点，1，用packer也可以制作vagrant使用的镜像。2，Vagrant技术原理它跟packer差不多，里面也有provisioner等等。也是控制虚拟机，也有自动输入和配置文件，只是表现形式产出结果不一样。3，它们共享很多其它机制，如都可以在host/guest间用nat，Nat方式也可以在本地搭建一个也有http sharing。只是配置文件内的写法大同小异。 下面来谈在vagrant guest os中使用host的VPN。比如我的VPN在HOST的127.0.0.1:1087端口,虚拟机里面的是不可能获得本地vpn的，所以我们需要在vagrant files中。加一句。export http_proxy=http://10.0.2.2:1087;export https_proxy=http://10.0.2.2:1087;因为用的是nat 网络。

> 加在这个位置: config.vm.provision 'shell', privileged: true, inline:" export http_proxy=http://10.0.2.2:1087;export https_proxy=http://10.0.2.2:1087 sed -i 's|deb http://us.archive.ubuntu.com/ubuntu/|deb mirror://mirrors.ubuntu.com/mirrors.txt|g' >>/etc/apt/sources.list ... config.vm.provision 'shell', privileged: false

如果不在vagrant虚拟机中开启VPN支持。可能会卡在Downloading and extracting buildroot 2019.08不动。这其实也是从前文《packer中学到的》，，可在打开的虚拟机窗口中测试box os用户名密码都是vagrant，ifconfig看到，在本机是访问不到nat内部的不能ssh vagrant@10.0.2.15。

Build Coreboot

Coreboot最新版是coreboot-4.10,使用linux kernel式的kconfig，和bb式的可视裁剪配置。所以它跟build root是flavor的，也对qemu有board支持。。跟buildroot的职责一样，完全对引导期，进入OS前的firmware OS的构建。

Coreboot并没有使用vagrant，你得在一台HOST上配置再编译。也是配置toolchain(因为稍后Qemu x86，所以用any toolchain)，mainboard那些，不同的有payload，对于mainboard，如果是测试，可以完全选择使用Qemu 模拟x86.

之后make一路通过。

我们的改变

如何集成到我们的cohdevbox srctree中。及你们的工程中。

上面提到vagrant和packer同基础，工作方式和配置都大同小异，在我们现在的实践文章中，都是用的packer，那么为什么不能把vagrant当packer用呢。Vagrant还可以避免出现packer那种一旦失败，全部从0重来的尴尬。即，在Vagrant退出按ctrlc,再重新执行up，它会重用上次grant provision的结果(因为它是虚拟机嘛)。这跟packer不一样，packer的构建不会重用上次provision的结果，会重新构建。虽然它不产生镜像（实际上那个镜像就是你的provider在你硬盘上的镜像，关机复制得到和packer一样的结果）。

我们会在srctree中提供一份编好的tinycorelinux硬盘镜像作为vagrant用的box。然后整个源码会deprecated packer。只用vagrant.然后在其中通过配置文件手动编译avatt，结合修改，代替现在的cohdevbox srctree。

最后，我们可能全程用qemu作虚拟机，因为qemu支持命令行提供firmware，它可以统一测试firmware和rootfs，完全可以用qemu来替换VirtualBox，qemu是命令行虚拟机工具。与其他的虚拟化程序如 VirtualBox 和 VMware 不同, qemu可以模拟CPU(全虚拟)，甚至喂给firmware。但vb,pd都不可以。还有一点，QEMU不提供管理虚拟机的GUI（运行虚拟机时出现的窗口除外），也不提供创建具有已保存设置的持久虚拟机的方法。除非您已创建自定义脚本以启动虚拟机，否则必须在每次启动时在命令行上指定运行虚拟机的所有参数。

我们选用的host平台是osx,不能用到kvm/qemu,因为kvm是linux的。在osx上我们不打算用到任何加速方案（半虚拟）。因为是测试。直接使用qemu默认的console窗口而不使用任何GUI管理器，vagrant可以通过libvirt用上qemu。

注意到coreboot编译GCC toolchain等输出信息很精简。而我们在《packer》文中不能控制选择过程warning等信息。所以决定研究后采用。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

将虚拟机集成在BIOS和EFI层，vavvt的编译(1)

本文关键字:**corebootv3低版本编译,让dbcolinux用上buildroot,在tinycorelinux上编译coreboot,kvm-coreboot,ovz-coreboot**

在前面《为你的硬件自动化统一构建root和firmware》文中，我们讨论了为平台和硬件自动构建linux dist的buildroot和coreboot(buildrom)方案。在那里，我们研究的是最新版的源码。文尾我们还谈到要把这一切port到cohdevbox srctree中且使用osx vagrant+plain qemu thru somewhat “libvirt plugin” under linux(毕竟qemu既是个模拟器也是虚拟机)，却发现遇到了几个困难：1)市面上却并没vagrant-libvirt to run qemu commands directly，vagrant与packer的插件都是面向kvm/qemu的。2）且osx下，与libvirt/kvm/qemu等同的方案却是xhyve thru vagrant-xhyve，3）第三，cohdevbox/dbcolinux的packer构建中我们是基于tinycorelinux384 32bit来进行对dbcolinux深度目录定制的，里面的kernel都是2.6.x的。且4）第四，要考虑集合进openvz/kvm as coreboot payload，即把buildroot和coreboot一起使用，比如coreboot会使用buildroot出来的结果。

—— 所以，源码的版本选择可能需要更好处理，困难4也可能会难于解决，不过，幸好之前我们也进行过在dbcolinux中集成ovz的工作研究。另外，我们还找到了avatt : all virtual all the time，avatt is a buildrom-based buildsystem that can be used to build a virtualization-aware BIOS based on coreboot using KVM or OpenVZ (latter is currently work in progress)，是20090705的一个项目 https://repo.or.cz/avatt.git/commit/1ee14af6661f1ebb0f18b9d1241ef1a67278c5ab。它整合了buildrom(coreboot早期的自动化构建脚本，此功能最新版coreboot已整合)和buildroot并加了自己的一些default configs和patchs，avatt/buildrom可参见 https://repo.or.cz/buildrom.git/commit/4f38d5c4d9e8cdb99fcb2bd5aac859605d46e4ef，其大约是coreboot V3一期的东西，至于avatt/buildroot,也是mirror的buildroot同时期的东西，值得一提的是其主要选择了ublic和i586作为target。

这成为我们本文将buildroot融合到cohdevbox srctree中的最好尝试。鉴于困难1，2我们依然选择使用packer而不是vagrant，这里使用virtualbox而不是PD（主要是PD business版商业化太麻烦，另外，还记得在linux上我们使用过packer+vmware?）。

准备工作

定制一下packer的启动。在start.sh中。

> export PACKER_CACHE_DIR=~/Packer/cache，加了这个可以将cache移出当前目录。比如你可以将它放到与>> output_directory（~/Packer/output）同根下一起。 packer build -force -on-error=ask dbcolinuxbases.virtualbox，加了force可以在每次clean后不用手动去删临时文件。

新的dbcolinuxbases.virtualbox文件里会是这样：

```
>> "type": "virtualbox-iso",
>> "guest_os_type": "linux",
>> "hard_drive_interface": "ide",之前在PD上，现在VB上这二个都是默认的others,ide，但一定要加。以免不创建硬盘。空>> 间是默认大小的。Guest os一定要是某种linux，否则无法启动某些TC grub条目。
>> “cpus”: 2,
>> "memory": 4096，其默认"guest_os_type": “other”,只有512m，不足于编译GCC时内存用量，加这个就可以在不用提及>> guest type的情况下配置内存用量。
>> 我们调整了srctree下目录，将各种src和tc.iso,tc下各种3.x/pkgs全移到res下。srctree/中只保留res和scripts(里面是整个>> avatt源码)
>> "http_directory": "res/",
>> 我们固定了port，而不用脚本中使用HTTP_PORT参数。虽然硬编不够灵活，但可以免除一些麻烦
>> "http_port_min": 8000,
>> "http_port_max": 8000,
>> 在上文《利用hashicorp packer把dbcolinux导出为虚拟机和docker格式（3》的基础上。我们在"boot_command”中新>> 增这二个包。
>> "tce-load -iw bash.tcz<return>",
>> "tce-load -iw texinfo.tcz<return>”,
>> "tce-load -iw tar.tcz<return>”,
>> "tce-load -iw quilt.tcz<return>"
>> 除此之外，"sudo sh -c 'echo http://10.0.2.2:{{ .HTTPPort }}/ > /opt/tcemirror’”,因为我们换了网络地址。
>> "provisioners”中，除调用Bootstrap.sh保持不变。传src，也变成传scripts，和一条调用make.sh
>> {"type": "file","source": "scripts","destination": "/mnt/hda1/tmp"},
>> {"type": "shell","pause_before":"1s","execute_command": "echo '' | sudo -S sh -c '{{ .Vars }} {{ .Path }}'","scripts":>> ["./scripts/make.sh"]}
```

make.sh的内容：

```
>> cd /mnt/hda1/tmp/scripts/buildroot/target/generic/
>> tar zxf sket.tar.gz
>> 如果不加以上，上面的传scripts会不成功。所以需要处理源码，除了generic中的，删掉scripts其它所有>> target_busybox_skeleton，target_skeleton，把generic这二个文件夹打包成sket.tar.gz放在generic下。
>> cd /mnt/hda1/tmp/scripts/
>> sudo make defconfig
>> sudo make
>> buildroot的整体逻辑与流程。我们看到defconfig实际上就是buildrom-defconfig加buildroot-defconfig，先后顺序很重>> 要。都是把.config先复制出来，然后sudo make
```

在packer中测试编译Buildroot

上面准备工作完成之后，启动start.sh,进入条件测试，如果没有bash.tcz，可能会出现相关错误 Retry，会出现一些常见下载错误。首先是wget命令。这是因为tc384中的wget版本过低。 ==> virtualbox-iso: echo "2009.08-git" >/mnt/hda1/tmp/scripts/buildroot/project_build_i586/avatt/root/etc/br-version ==> virtualbox-iso: wget: invalid option -- 'n' Buildroot.config里面，去掉BR2_WGET="wget --passive-ftp -nd”中的-nd，retry

接下来的这些下载错误，属于文件没找到错误，基本在buildroot.config或packages/xxx/xxx.mk中可以找到修改地址的地方。将其找到，一一修改为正确的调用packer http sharing的逻辑：

> 找不到linux-2.6.26.8.tar.bz2，BR2_KERNEL_MIRROR="http://www.kernel.org/pub/" 按路径改这个>> http://10.0.2.2:8000/src/dl/ Retry,找不到uClib-HEAD.tar.bz2,去uClib.mk，UCLIBC_SITE:=http://10.0.2.2:8000/src/dl/uClibc/snapshot。snapshot>> 等这种命名是ulibc的源码发布方式。这里的uClib源码要找0.9.30，在本地解压重打包成uClib-HEAD.tar.bz2，>> 记得里面的顶层文件夹uLibc-0.9.30改成uLibc-HEAD，如果直接用官方最新的，1)可能编译过程会找不到配置需要你手动>> 配置。也会出现error: pthread.h: No such file or directory对应不到头文件。2)高版本gcc编译低版本uclib及kernel会出>> 现unifdef.c 中的 getline 提示重复，将extra/scripts/unifdef该文件中getline全改成get_line，有三个地方。 Retry,找不到mpfr-2.4.1.tar.bz2，mpfr.mk中，改MPFR_SITE:=http://10.0.2.2:8000/src/dl/mpfr-$(MPFR_VERSION)，>> patches文件是mpfr-2.4.1.patch，同样放到mpfr-2.4.1/下，会在虚拟机自动被下载并命名成mpfr-2.4.1.patch。 Retry,找不到binutils-2.18.50.0.9.tar.bz2，改BR2_GNU_MIRROR="http://10.0.2.2:8000/src/dl/gnu”，binutils包放在>> src/dl/linux/devel/binutils下。 retry,如果出现binutils/objdump] Error 2，则是因为makeinfo没有安装，如果出现unknown command `cygnus’，临时>> 解决办法是将shell texinfo降级到4.13，通过下载编译源码编译安装比较老一点的版本。幸好tc384的makeinfo足够低。 Retry,找不到gcc-4.3.3.tar.bz2,gmp-4.2.4.tar.bz2,前面改过BR2_GNU_MIRROR,这里仅需将gcc包放到>> res/src/dl/gnu/gcc/gcc-4.3.3,gmp包放到src/dl/gnu/gmp Retry,如果出现bug出错字样,可能内存不够导致编译gcc失败。可以retry一次或者在配置文件中多加一点内存，预留的2G足>> 够。 Retry,找不到ccache,改ccache.mkCCACHE_SITE:=http://10.0.2.2:8000/src/dl/ccache，包放好 Retry,找不到module-init-tools,改module-init-tools.mk，包放到src/dl/linux/utils
>
> Retry,找不到linux-2.6.24.tar.bz2,之前那个2.6.26.8应该只是取得头文件用，这里才是编译真正的kernel用，改>> buildroot.config里面BR2_KERNEL_SITE="http://10.0.2.2:8000/src/dl/ovz,，注意前面的kernel是改>> BR2_KERNEL_MIRROR这里是KERNEL_SITE，有所不同。这里要放的包是ovz的linux kernel linux-2.6.24.tar.bz2，跟>> uClib一样要处理一下得到。直接下载legacy openvz的2.6.24-ovz008 branch 最新src tarball即可。重打包时保证包内根目>> 录为linux-2.6.24 因为这里我们要绕过BR2_CUSTOM_LINUX26_PATCH="patch-ovz008.1-combined.gz"的下载及使用，因为这个2.26.24>> 是本来就经过了patched的ovz src，也发现二个BR2_LINUX26_CUSTOM=y和>> BR2_KERNEL_LINUX_ADVANCED=y要作何处理呢，这二不要动否则不会生成vmlinux。patch source的修改处为>> target/linux/Makefile.in.advanced： 1)$(call DOWNLOAD,$(LINUX26_PATCH_SITE),$(LINUX26_PATCH_SOURCE)) 2)toolchain/patch-kernel.sh $(LINUX26_DIR) $(DL_DIR) $(LINUX26_PATCH_SOURCE) 一个是下载，一个是apply patch，统统#禁用即可,其实也可在buildroot.config里找到开关：>> BR2_KERNEL_PATCH="$(BR2_CUSTOM_LINUX26_PATCH)”，置空””即可，我选择是前一种方法。
>
> retry，找不到busybox-1.13.4.tar.bz2,改BUSYBOX_SITE:=http://10.0.2.2:8000/src/dl/busybox Retry,找不到e2fsprogs-1.41.3.tar.gz，改e2fsprogs.mk中>> E2FSPROGS_SITE=$(BR2_SOURCEFORGE_MIRROR)/e2fsprogs和>> BR2_SOURCEFORGE_MIRROR="http://10.0.2.2:8000/src/dl”。 Retry，找不到pciutils-3.0.1.tar.gz，改pciutils.mk中PCIUTILS_SITE:=http://10.0.2.2:8000/src/dl/linux/pci和>> PCIIDS_SITE:=http://10.0.2.2:8000/src/dl/linux/pci/ Retry,找不到qemu-0.10.5，改qemu.mk中QEMU_SITE:=http://10.0.2.2:8000/src/dl/qemu/ Retry，如果没有tar.tcz，可能会出现tar解压错误unrecognized option '--strip-components=1'。 Retry，相当于在qemu srctree中make，结果出现 ==> virtualbox-iso: Makefile:3: config-host.mak: No such file or directory 于是，在buildroot.config禁用BR2_PACKAGE_QEMU=y，BR2_PACKAGE_VZCTL=y，>> BR2_PACKAGE_VZQUOTA=y（Qemu应该是往kernel中加入kvm，后二者是加入Ovz?日后解决） Retry,找不到fakeroot_1.9.5.tar.gz，改fakeroot.mk中FAKEROOT_SITE:=http://10.0.2.2:8000/src/dl/fakeroot/ retry，找不到genext2fs-1.4.tar.gz，改ext2root.mk中GENEXT2_SITE:=$(BR2_SOURCEFORGE_MIRROR)/genext2fs,>> 至于BR2_SOURCEFORGE_MIRROR在前面改过了。

终于到这里了，脚本将利用target_skeleton打包成，至于# BR2_TARGET_ROOTFS_EXT2 is not set，表示它并不会被使用，ext2的rootfs在嵌入式上很流行。但脚本默认使用了INITRAMFS,即BR2_TARGET_ROOTFS_INITRAMFS=y，它利用生成的rootfs.i586.initramfs_list文件。然后写入到echo "CONFIG_INITRAMFS_SOURCE=\"$(INITRAMFS_TARGET)\"" >> $(LINUX26_DIR)/.config产生作用，这个linux26_dir是project_build_i586/avatt/linux-avatt-openvz，会在构建vmlinux时自动将rootfs整合到vmlinux内部。

整个此buildroot全程构建时间约<20分，若成功会在buildroot/binaries下会生成vmlinux(约4.4M大)，供后续buildrom使用

在packer中测试编译Buildrom

virtualbox-iso: /mnt/hda1/tmp/scripts/buildrom/buildrom-devel/bin/fetchsvn.sh: line 9: svn: not found 不需要tce-load -iw subversion,我们只需要按与buildroot那些下载逻辑一样。在buildrom的sources下载到它需要的Mkelfimage-svn-3473.tar.gz即可，Mkelfimage-svn-3473.tar.gz怎么来的？在github.com/coreboot/coreboot搜索带mkelfimage的commits，https://github.com/coreboot/coreboot/search?q=mkelfimage&type=Commits，发现

https://github.com/coreboot/coreboot/commit/ebc92186cc9144aaacd37ca1ae94fcff60ec577a，这条刚好是以前通过svn导入的3473：git-svn-id: svn://svn.coreboot.org/coreboot/trunk@3473 2b7e53f0-3cfb-0310-b3e9-8179ed1497e1。下载其包文件解压源码，将其移到一个svn文件夹下，然后打包成mkelfimage-svn-3473.tar.gz，放到src/sources中。

再做以下必要修改：在mkelfimage.mk，改MKELFIMAGE_URL=http://10.0.2.2:8000/src/sources/mkelfImage，然后把下面一句由： @ $(BIN_DIR)/fetchsvn.sh $(MKELFIMAGE_URL) $(SOURCE_DIR)/mkelfimage $(MKELFIMAGE_TAG) $@ > $(MKELFIMAGE_FETCH_LOG) 2>&1 改为：@ wget $(WGET_Q) -P $(SOURCE_DIR) $(MKELFIMAGE_URL)/mkelfimage-svn-$(MKELFIMAGE_TAG).tar.gz ，这句是参照接下来unifdef.mk中取来的

retry,找不到unifdef,unifdef.mk中改UNIFDEF_URL=http://10.0.2.2:8000/src/sources/unifdef Retry,这里没有安装quilt.tcz会出现quilt:command not found retry,找不到lzma443,lzma.mk中改LZMA_URL=http://10.0.2.2:8000/src/sources/lzma

脚本在将结束时将寻找前一步产生的rootfs和vmlinux。生成elf coreboot payload bios.然后启动coreboot的编译。

Retry,提示找不到coreboot-svn-3772.tar.gz，buildrom是将coreboot作为它的一个package的，在https://github.com/coreboot/coreboot/commits/master?before=91eb2816faa4b2689f30ca47ff2585cf79ac53f3+28735找到目标：https://github.com/coreboot/coreboot/commit/544dca4195a6fb77286299bf42d4db5813dc28ac，按上述类似的方法处理。

未完待续。

我们要做的，会慢慢精简它到固定版本的buildroot脚本,然后把定制目录的dbcolinux移进去。至于使用这个rom，我们可以将其刷入bios中（当然，涉及到很多适配工作）也可以放在efi分区中。用处呢？我们可以利用其虚拟机管理功能建立devops，或用于类exsi裸金属式装机，在这个rom中加入moeclub的脚本作netboot系统安装运维达到osx network recovery的效果且更灵活,比如，使用局域网wifi usb disk存储镜像在线安装系统。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

硬件融合的新起点：虚拟**firmware，avatt**的编译**(2)**

本文关键字：硬件融合的新起点**,**融合**PC**和手机，操作系统并启/网装法**,**真正的**virtual app**

在前面《在阿里云上装黑苹果》时我们谈到clover,它是一种能虚拟硬件的EFI而不仅仅是一个loader，其改写硬件逻辑的作用，是重启后依然存在的。（所以，它有可能对硬件造成损伤，切记不要在白苹果上使用clover，mbp使用clover丢firmware和坏屏的例子不少）。从虚拟EFI，到后来，我们又谈到parallel os bootloader和将hypersior集成到loader的设想，并最终发现了coreboot，而它居然可以集成linux和虚拟机，。。。。。这种loader层和firmware层面的虚拟化层面，为PC和云裸金属提供了一个地道的虚拟化层面。即firmware as infrature技术,它使cb进化到了具有融合硬件的作用，试想一下，现在的PC和移动端越来越强大，注重融合体验，可是都做得不够好，而如果走让不同的融合设备共享同样的infrasture，即统一硬件的路子，，就可以解决很多以前不好解决现今统一上层之后变得很好解决的大部分问题。

其实这个我们在《为你的硬件自动化统一构建root和firmware》也隐约提到过如，虚拟机virtual applicance 这些（vs webapp,nativeapp,etc..）都不好用，purism一体linux phone,laptop。现在放大来讲一下：现在做的桌面移动统一，都是从生态上整合，因为技术上内核从一开始就不一样。上流决定下流，这样会产生很多问题，兼容，适配，开发，separated codebase for different hardware。故，桌面移动统一，需要从硬件生态提出一个firmware层。之后所有的终端，不管你是谁，一律使用用户态OS即可，这种使用从firmware层（技术层）的融合可以解决绝绝大部分问题。比如一开始我们追求的OS并启，OS网装法，还比如类似上面的purism：使笔记本和手机共享无缝使用同一种linux，所有这类问题它都可以解决，以后就不用再使用chroot这样的技术给手机换OS了。还可以带来新的创新视点：还比如，基于融合OS上的融合语言层和融合开发层，如 as Devops tools（这其实是cb的一个辅助作用属工具性质）。还比如使用hyperkit统一虚拟化和容器技术，。使得虚拟机直接成为app runtime，提出真正的virtual app。就像GO语言的整合粒度那样。

关于硬件融合设想

一个人其实基本上会用到各种硬件几件套，手机和PC，还有NAS，,办公PAD，编程pad，游戏pad。作为一个编程者需要接确到的OS，1，PC：装客户OS和NAS OS，同时运行，2，手机端装programming os和nas 客户端OS,试想下这些都能实现在现有硬件上。

因为你不是硬件厂商，我们只能从换firmware开始做融合硬件，有一些coreboot for phone的方案也在慢慢出现。其实coreboot也有for andriod后端，你完全可以通过这种技术给andriod手机写linux OS。那么laptop呢。最好的测试平台是买一台chromebook,测试cb的最好平台也是cb，google的pixel book和pixel手机总是第一时间被coreboot支持，至少早期的硬件性能不够就不要考虑了，如ubuntu touch刷早期 meizu之类的方案，当然对于programmingpad，我们做不了纯粹的融合硬件设想，比如像带实体键盘的programmingpad(实体键盘与触摸主要是一个能不能盲打的区别，这跟键子的边沿设计有关可以用键盘膜)，也不好直接从黑莓,titan上做，就不考虑。

总之，虚拟firmware似乎是融合硬件的终极大法，甚至还有人做成了融合ios:

> Corellium此类方案其实也是从firmware层着手，如同clover for Mac，Phones using coreboot
>
> Corellium iOS Kernel Research Virtual iPhone Hardware. Corellium virtualizes any mobile device hardware through software platform and allows you to test on various different devices without actually having physically had those devices. This is not the same as simulation as we get with Xcode and with other platforms where you would just simulate the iOS on a different iPhone through the Mac. this is actually a virtualized hardware it’s going to be basically a one-to-one clone of the device as if the device was a physical devices.
>
> 要使用这个刷机系统和得到想知道的私人信息，你访问他们官网发邮件，回复要100万美元..

后来我们还编译了avatt，研究了coreboot的自动化脚本技术,这里继续：

编译**avaat(2)**

在前面我们buildroot讲到会生成到vmlinux，vmlinux集成了ramfs的rootfs，在buildrom里会生成到work/coreboot/svn/targets/emulation/qemu-x86/coreboot.rom，最终cp到deploy，所以这是一种从Vmlinux->到payload->到rom的过程，那么我们如何在packer中自动获取这个deploy到host呢。

我本来企图用virtualbox-ose-additions-modules-2.6.33.3-tinycore.tcz在packer中装guest iso和设置packer的shared folder来实现。无奈几次都没有成功。所以，为了省事，最后找到这个，写在dbcolinuxbases.virtualbox中：

dbcolinuxbases.virtualbox中的改动

```
…
{"type": "file","source": "/mnt/hda1/tmp/scripts/buildrom/buildrom-devel/deploy/*","destination": "/Users/admin/Packer/output/
","direction": "download"}
```

此除之外加上这二句，让packer变得更像vagrant一点:

```
"skip_export":true,
"keep_registered": true
```

还得加一个tce-load -iw python.tcz在适当位置

make.sh中的改动：

最后，由于packer是按上传给他的sh来检测改动，并在一次retry中生效的，形成一次debug，为避免debug周期过长，我们需要把make.sh多做几个脚本放开调用，这样可以避免修改一个sh后retry的等待时间过长。

上次讲到在coreboot编译时，这里继续

Package/coreboot-v2/coreboot.inc中把svn逻辑改为wget，由@ $(BIN_DIR)/fetchsvn.sh $(CBV2_URL) $(SOURCE_DIR)/coreboot $(CBV2_TAG) $(SOURCE_DIR)/$(CBV2_TARBALL) > $(CBV2_FETCH_LOG) 2>&1 改为：@ wget $(WGET_Q) -P $(SOURCE_DIR) $(CBV2_URL)/$(CBV2_TARBALL)

如果之前没有安装python.tcz，那么这里还会有：virtualbox-iso: ./buildtarget: line 49: python: not found

好了下面来开放那三个在《编译avatt(1)》中禁用的三个虚拟机相关项，qemu和ovzctl,ovzquote，我们只开放后二者，因为kvm在avatt中是不能用的。

vzctl-3.0.23 Vzctl.mk中，我们看到整个br的逻辑其实是一个wrapper.从这里可以看到vzctl采用的是package的自动，属于vzctrl.mk中尾端的那个调用。里面AUTOTARGETS是来自package/Makefile.autotools.in，里面define AUTOTARGETS $(call AUTOTARGETS_INNER,$(2),$(call UPPERCASE,$(2)),$(1)) endef，参照vzctl，找出qemu patchs的位置是package/buildroot-libtool.patch，

我们看到执行vzctl.mk会下载到正确的vzctl-3.0.23，但是编译的时候出错：

```
==> virtualbox-iso: mkdir: can't create directory '/mnt/hda1/tmp/scripts/buildroot/project_build_i586/avatt/root/etc/vz/dists'
: File exists
```

将vzctl.mk中的VZCTL_INSTALL_TARGET:=YES改为NO。因为这条重复了。

相反，对比vzquota-3.0.12 vzquote.mk，我们看到它主要是mk中的逻辑。没有使用到auto wrapper，其实vzctl本来就是用libtool和autotools编译的，所以会用到此auto wrapper。而vzquota不用。

一路通过。vzctl和vzquote会生成在ram rootfs的/usr/sbin,buildrom-devel/config/payloads 中运行这个rootfs的是COMMAND_LINE=console=tty0 console=ttyS0,115200 rdinit=/linuxrc

好了。最后。让我们本地用qemu测试运行导出的rom。进入本地output文件夹：

qemu-system-x86_64 -bios build/emulation-qemu-x86.rom -serial stdio

不加serial，会发现运行命令的那个console窗口和打开的graphic窗口都不显（这里显示payload,加-nographic可以让这个graphic不出现）。-serial stdio是让一切显示在console,可是我们发现在console窗口中在出现rootfs登录提示符前卡住了，上面的cmdline又是正确的，问题只能出在sket.tar.gz中，修正target_busybox_skeleton和target_skeleton中的/etc/inittab：

```
# Put a getty on the serial port
# ttyS0::respawn:/sbin/getty -L ttyS0 115200 vt100 # GENERIC_SERIAL
将ttyS0前面的#去掉。
```

终于出现avatt命令行了，你可以在这里root登录。你还可以在运行qemu-system-x86_64后加-hda xxx.image -m 1024 -net nic加硬盘，内存和网卡，当然这个rootfs似乎并不好用(target中只开放有限工具，甚至fdisk都没)，都甚至不能完整分配硬件资源测试ovz。需要修正。ovz也是32位的只认有限内存。

未完待续。

———

通用系统网装法，现在是网络时代，只要保证bios不死，如果还停留在u盘装机时代就OUT了。我们现在是用qemu运行，那么如何把linux不刷机弄为虚拟efi运行呢。比如放在efi文件夹就如同clover那样，不过这样就不能打造不死bios了。以后尝试。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

mineportal新硬件选型，威联通or群晖？

本文关键字：威联通vs群晖,公网IP盒子,群晖personal photostation

在《使用群晖作mineportalbox（1）》中，我们谈到组织单一同步文件夹的省事方法，即，在那里，我们基于将所有用户使用NAS过程中产生的资料按逻辑置入一个统一可同步cloudstation文件夹的思想（这样主要是为了统一cloudstation上传下载+cloudsync云备份，且支持各种除cloudstation之外的xxxstation直接存取里面的数据，而避免一定要用到群晖默认的那套xxxstation->顶层xxx文件夹结构和多共享文件夹分别备份的需求），发现群晖是不能将photo,video（media），www,codesnippters(softs),_current syncing docs置入一个单一共享cloudstation进行备份的-比如将其置入home下或者一个单独可cloudstation同步共享文件夹，其中最大的问题就是photostation和webstation。前者通过ds photo自动备份上来的照片_current pics像册文件夹和photostation调用的图片必须要在顶层photos下，后者webstation不直接利用home中的文件夹建立虚拟机。

本文是那文的增强，依然只谈三个问题，硬件选型，公网转发，和省事单一可同步文件夹组织。

威联通

事情是我得到了一台qnap，发现较群晖其硬件和系统层面各有特色，虚拟方面，x86架构都支持虚拟docker和虚拟机管理，有了虚拟机的情况下，mineportal作为身边的云主机这个概念才算是完善的-因为在资源范围内可以按需开不同的机器，qnap威联通性价比都说比群晖要高，但实际上qnap太吃内存了，同样存在开虚拟机需求的情况下，qts 16G内存不见得能开三个虚拟机，而dsm已经可以建立起好几个虚拟机了，好吧，个人感受当不得真。

qnap虚拟机的亮点在于其可当PC可当NAS可当虚拟机建立开发机可当服务器可当安卓盒子可当机顶盒播放器的特点，但其实只有docker出来的和linuxstation出来的pc是运行在裸机层次的，运行在虚拟机内的要通过qvpc技术透出到hdmi，实际上也是vnc网络信号，其体验由于虚拟机是软件指令下的机器，性能在本地vnc也是柔柔的操作体验-这是性能问题不是网络问题。

而，实际上，这种双系统主机是非常有意义的，比如nas系统+安卓系统，可以远程连接安卓播放nas中的东西，或在安卓中下载一个百度云客户端进行云同步。连结远程windows，可以调试本地局域网下的linux开发机，在多虚拟机环境下，你可以虚拟出一个软路由系统（假设它有一个公网IP，稍后会讲到），将其它虚拟系统中的服务通过软路由网关服务透出来。

其它方面,qnap的videostation支持rmvb streaming,但是它的andriod photo支持怪异的备份方式，它的notestation可以导出pdf，然而默认标题留空的话不会自动取内容第一句填上，这对用它作记事的用户是十分不友好的。当然还有更多可以比较的方面。

公网转发和IP盒子

另外一个使mineportal靠近身边的云主机这个概念的就是公网IP盒子了。frp加云服务器的方式毕竟太不稳定， 盒子的原理就是一个高速转发器，网上可买到。可直接也可旁接。

在直接法下，盒子可以通过直接法连需要获得公网IP的设备（直接法下，盒子LAN口接待获公IP设备）。也可以侧接法，旁接法下，LAN口闲置。 这二种接法下，盒子的WAN口始终要接路由器某LAN口拉出来的线，因为无论如何盒子必须要联公网。

盒子设置页面是盒子访问路由器获得公网信息（比如IP）的手段。盒子只能通过一条网线和电脑直连访问到盒子设置界面（盒子LAN口），里面有二个页面:盒子设置和公网IP设置，有几点要注意：

> 1)公网IP设置页面中的那个内网地址是要转发到的内网地址，可以是148.100（直接法，由盒子本身作网关），也可以（旁接法）下，由路由器作网页， 此时这里可以填路由器下可能分配的地址192.168.1.xx。此时，盒子只是一个路由器局域网内转发器（所以不用任何转发规则和端口定义， 就可以透出NAS对应端口的服务。因为远程IP并没有限制你对端口的使用） 2)即使上面那个IP设置不断变更，通常也不需要担心地址错乱而reset盒子，因为其网关和访问地址永远是192.168.148.1 3）如果一旦reset,盒子重置后需要更新，要断电20秒。否则会提示一直初始化中。 4) 将盒子界面的内网IP设成路由器 后。转发规则会失效。 所以这种情况下，不适合一种方式，就是达到ISP给路由器分配出公网IP的效果，然后你企图在路由器上作DDNS，转发。

在直接法下，盒子本身也是一个网关，，侧接法下，属反向代理，程序上，外网不能通过程序手段获取得到你的设备IP。

最方便的做法还是侧接，因为盒子跟路由器放一起，上网设备只需连接路由器（可以无线），这种方式下也相当于给路由器配了个IP（注意这只是相当跟ISP给你路由器分配IP不同，下面会解释到） 所以，侧接法这名字和接法这个名字比直接法更符合IP盒子名字和直接隐喻。 是推荐的方法还可省去一条网线。

省事单一可同步文件夹：dsm支持personal photostation

威联通的qts支持链接，可以将一个顶层共享文件夹的内部目录透出来成为一个新顶层共享文件夹，这样就可以将所有资料（无论是按用户逻辑归类的还是NAS应用产生的默认文件夹）归类到一个可同步文件夹下。

后来由于种种原因我还是弃用了qts转用了x86的群晖，因为虚拟机是我必须的，hdmi透出桌面不是我必须的。我必须的是dsm xxxstation给用户培养的使用习惯，如notestation用于记事不用写默认标题，ds photo支持自然方式同步备份手机照片。还有cloud sync支持百度云而qts仅支持部分。

其中最重要的，还是我发现dsm支持personal photostation，photostation设置中开启home photostation，且右上角用户头像中开启当前用户home的photo目录，在其中建文件夹当像册，建立到dsm桌面的个人photostation快捷方式，ds photo中选择那个向下箭头可以以personal photostation方式登录。这一切就跟使用根下的photo归类一样。你可以将archived photos归档和_current syncing photos同时做到home/photo下。

所以我现在新的统一备份和归类方法是：

> 即home/cloudstation:_current,archived.docs home/notes:_current,archived home/photo:_current,archived （photostation个人） home/video:xxx （videostation没有个人） home/www:xxx (webstation个人)

以上archived都是经_current整理后打包的档。各个_current你可以理解为_2018lifedocs,_2018huaweimobilepics,_2018homepc等等， cloud现在单独跟其它station并列并不作最顶层，同样支持整个文件夹一起cloudsync,hyper backup备份，使用cloudstation下只需要上传备份某_currents即可。

关于利用类群晖或qnap的硬件作mineportal，atom等intel baytrail等架构功率和速率上完全可以类比移动平台arm。一些插座式主机，Marvell插座式电脑还可作手机伴侣，群晖也有vesa到显示器背面的产品。这些都是通常这二种NAS的变体。但mineportal一般用于服务环境，建议四核x86环境毕竟开虚拟机性能上至少四核内存越大越好。

至于客户端，你可以用vnc，云终端，或chromebook同样是低功率架构（群晖+chromebook是天然的搞创作，办公生产的绝配，这二种都原生web based，用vnc+一套键盘鼠标可以达到透出虚拟机桌面的效果，产生《利用chromebook打造无线同屏器效果》的话题）。当然这样的搭配就不要玩用于游戏了。玩游戏可以用专门的安卓手机,psp,win gbc等。甚至，由于客户端可以无穷定制和专门化，也可以有《利用七寸umpc打造programming pad》这样的话题。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

ubuntu touch: deepin pc os和deepin mobile os的天然融合

本文关键字：**ubuntu touch as deepin mate os,second pc os**

在《一个matepc,mateos,mateapp的goblinux融合体系设计》我们一直寻求第二PC的硬件选型，它可以是一个小主机配个电脑通过typec相连供电，一台一体机配个小主机钉显示器后面，一台双系统机箱内的双主机，或者裸机架架起的主机群，还可以是你能想到的任何组合方式。二台主机可以同局域网（通过路由器），数据线直接交互，甚至异地（通过互联网）。。这种双主机需求是很常见和急迫的。

这些主机间用某个主机上的OS管理器管理，呈一样的外观，就好像他们在同一台主机同一个OS下的表现一样，这就是融合os，在《兼容多OS or 融合多OS？打造基于osxpe的融合OS管理器》《一种含云主机集群，云OS和云APP的架构全融合设计》中我们都谈到这种技术的基础和理念，由来，类parallesdesk方案：它尽量抹去了不同操作系统间的沟壑，而不用真的试图去填补这些OS间的异同。

谈到融合，有更多的例子，比如锤子tnt，三星dex将PC和mobile模式合而为一的显示方案，变形本，这些只是硬件上的例子，是处理现在既成事实的条件下，在多样化，不同质的产品方案间求得统一方案的权宜之计。还比如上面提到的mate os ------ 它本质也是一种融合os管理器技术。只不过我们要更进一步。

我们将从OS层面去融合，如果融合可以从选型开始加少量的融合工作本身，依然可以不用折腾太多。那么，何妨从软件的底层去融合呢？比如用同尽可能同一份OS同时用于pc,matepc,作mate os。这样，可以将相同的OS间共享同样的机制，subsystem，比如同样的os可以将同步做在os级别，matepc可以直接与mainpc互为可同步的mate，增加一个新的节点，只是增加一个同质的os，同步照样可用，运维也方便。。比如将mateableos作二份发布，一侧fs托管在别处。则另一侧必为其管理性系统，比如提取一个阿里云access key就可以在本地mirror它。这样就做到了在OS->filesystem层面的同步。

1,把deepin和skynas作为一对mateos?

最近我用上了deepin linux(说实话，很早以前，大约2015年第一次尝试它也是各种不顺手，也不是因为小bug，而是根本不习惯bsd派生系用在桌面的风格和习惯，ubt之前也用过一直没能习惯，故放弃，后来折腾了半年的osx之后，有了过渡，所以这次2019年9月再次折腾v15的第11版，虽然时间过去这么久deepin已由ubt based变成了debian based,也由qml切换到了qt+go后端,虽然这次少量bug依旧存在，但最终通过试用它几天后我总算还是成功继承了自己使用在桌面使用osx的感觉)，加上发现它里面的应用已经足于应付我日常工作和开发了，而且也实现了它的承诺：美观轻量的linux桌面环境，所以最终决定就把它作为自己的装机OS,mainpc os了。

deepin还缺少icloud，timemachine这样的互联网，局域网备份装机支持，这也是我要为deepin找一个deepin mate的原因。我选择的是阿里云ecs+skynas群晖：虽然配备了大容量存储和本地式黑群非常好用，但配有公网IP和异地备份的远程云更合理化。现在ADSL也是越来越快了，如果不是用来存储小丽姐，其实最大100G的云服务器是够用的，而且这个成本一年也是个人用户能够承担的。

基于上面的同os的matepc设计，阿里云ecs上应装deepin，webdeepin，the headless deepin mate os for deepin，这样的第一步，是把deepin的kernel提取出来，作成一个syno的webasisst之类的东西 ，支持rootfs的安装和升级。至于mateos间的文件系统及文件系统同步设计（可直接使用brtfs的snap？oss远程文件系统？），又或者可用couchdb实现的数据库分布式文件系统。二个系统在开机后就自动同步了，不用在mainpc上像群晖一样打开一个守护程序。又或者它是一个git repo的东西，手动同步的，支持客服同步APP同逻辑（只不过remote,local分布不同）。

无论如何，为deepin增加云存储功能。且保证好用稳定的同步,互为mateable，这些，一定要做到OS层。

当然，未来我们的mateos，是Os级整个的同步，包括api,kernel，不只支持装机和用户数据cloud sync，因为它要是能够支持bcxszy的matestubos and bpi programming设想的，这是后话。

2,如果matepc还是一台装用mainpc os的手机

可是它要是能用于三端mateable，手机和云端和本mainpc，这就是一个更为复杂的选型和融合了。

最近我还发现了ubuntu touch这个项目，其实不过这个项目在2018年就被官方deprecated给了另一个团队了，然而，它最大的特点是可以利用常见的一些手机作为matepc，甚至把它们当成开源手机硬件平台使用。这不是chroot技术，也不是linux on deploy技术，而是实实在在的将ubuntu全新安装在这些设备中。

ubuntu touch与deepin有着极为相似的生态，甚至可以将前者发展为deepin mobile.

在这台第三PC上，要安装mate os for pc,可以做成deepin mobile，----- 一般来说，pc和mobile这两个是个人最常用的mateable的标配，路由器或云主机都不是，路由器在外面就离线了没有公网，云主机同步过来的文件并不是立等可取，只有mobile随身带，当它能用于100G个人理想数据存储量。------ 这样所有的APP可PC可MOBILE，可ECS，mateable entity之间可以相互之间融合app了。

话说，ubuntu touch的目的之一就是降低多端APP融合的难度。这样多端一统的OS设计，可以同步用户数据，解决装机/全盘备份问题，甚至可以统一web，手机app，消灭web/webapp本身。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

聪明的**Mac osx**本地云：同一生态的云硬件，云装机，云应用，云开发的完美集

本文关键字：**native cloud,uniform cloud os and client os**，利用**PC**打造云平台。**caching as cloud, sync as cloud**

在前面我们讲到多种云平台和云OS的选型，像群晖，dsm之类都是为yunos专门建立起一种硬件平台和软件系统的典型，它有点像极路由，其appmodel和各种扩展都是基于web选型，它们的区别也许只是一个：一个面向大容量存储服务，一个面向连接服务。它们是设想与生产环境和工作环境一起协调工作的，但这三者明显有功能和职责相交的地方，但如果，存在有一种硬件软件选型，既可以负责起日常工作需要的那些任务，还可以同时作中心存储等任务而不需要另外添置新的硬件，不是更好么？这既节省了一台硬件，又节省了一个OS的开销。比如，就把PC同时当NAS当工作环境当路由服务，不好么？

在前面在《把群晖+DOCKER当WEB云OS》一文中，我们讨论了云OS从硬件到一系列其它方面的选型，包括开发。那么新的集成方案可以是同类的东西么（yet another alternatives）,这需要涉及到一系列问题:

> 首先，生产环境的客户端和服务端能否做在一起，nas和软路由强调自身作为服务性环境存在，如果将其移到客服同体，比如在pc上开一虚拟机或docker或其它什么方式虚拟黑群晖，这会造成资源抢占效率低下，这台pc作为客户端也需要配备大功率的显卡，显得不伦不类，但这也不是什么大问题----对于个人使用是可以，我们在前面也涉及到多种host/guest共体的选型，host其实也可以作为管理OS。这其实我们第一种方案就是这样的《利用colinux和mailinabox在本地打造mineportal2》 其次，客户app和服务性app是否真的可以共存一PC中，比如使用容器技术分隔资源配额，互不侵犯。这样还可以统一传统的包服务和容器服务，像沙盒技术可以同时用在客户端和服务端APP打包技术中，只不过服务性app或容器需要透出服务，而且通常采用的web UI模型，其实普通的GUI就够了，因为也有remote appmodel这种程序，如果非要涉及到web，可以仅把web当ui,不要把它当一种全栈的appmodel，比如利用静态网站生成器的原理，treate web as only ui ------ 其实app提供一种ui就能在其上打造出一个appmodel而不论你在其中采用的UI方案是什么。 第三，关于同步，在前面我们看到群晖是基于客户端与服务端sync的，是工具性质的，更前面我们看到cloudwall这种是基于pouchdb这种数据库级的自动化sync的。----- 其实，一个appmodel，只要它是远程分布的，只要它提供客服sync方案的，这基本都是云。而云，这个概念广得很，其实云不一定要跟OS有关，跟采用何种内容同步有关，不一定跟开发有关，它可以是各种加速，比如，云加速下载就是把热门资源用自己的服务器缓存起来，像百度云和迅雷一样，也可以是云计算，也可以是区块链。 第四，各种虚拟化和devops支持-devops其实就是云开发的代名词。这些我们慢慢道来，在《TERRA++》文中我们提到可编程可以成就devops和CI。--- 其实可碎片化就能可持续集成，像文档化的xml，可编程化构建的容器语法都是，这些在上篇《terra++》中有述。

综上，它们在技术上他们也是可以统一的只是选型不一，但都可以是云。所以是可行的。

Osx即是这样做的，还做得特别聪明，为什么是osx呢不是win呢？win没有集成太多的云服务，且除了pc只有mac是最接近可用的生产力工具了，而后者刚好也做得很不错很具典型。

osx的native cloud模式:remote gui appmodel+caching as cloud

Osx生态没有专门的server版，是直接在osx上建立的，要说有，也是一个叫mac osx server的普通app，但osx被当作云还有其更多方面。

首先，其icloud与finder是整合的，数据存储在苹果的服务器，document文件夹(osx叫它文稿)通过finder作同步客户端，这是第一步，第二步，配合osx server的描述文件管理器，icloud数据还可以caching在另外一台装有osx server的机器上作本地同步,vs sync as content distribution,streaming还没有发展起来，caching其实也是一种聪明的方法。

> 使用osx yun，你不必自建那么多服务，这些服务商的寿命比你的寿命还长，还有，这种中转性的服务足够了。比如我github一下，也可以混用客户端与服务端。本地只需作个中转不需要自建服务。而且osx的各种icloud app，比群晖的那些更好用。比如备忘录，图片上传多端可见，

其次，osx是直接把任何服务性的app都当成云APP的，这就是我们在前面一直谈到的，vs web as cloudappmodel，其实remote guy appmodel也是云app的一种，内容分发方面，如上所述，现如今5g和streaming技术不太发达，这种appmodel还远远没有发展起来导致的。

第三，osx server负责管理硬件，管理APP缓存，数据的分发（内容分发），甚至其还管理装机数据的caching,即time machine(time machine这不就是群晖等的active backup吗)，而且，osx server的做法更聪明，timemachine这类app是客服共体的，就像finder作为同步客户端一样。而且mac它的生态更统一，涵盖整个PC还有IOS这种可移动端。

下面来讨论一下具体做法，并把其各种方案和与群晖类似的其它方面作一对比

类群晖,把双盘**mini**打造成**mbp**的**matebox:**装机服务器数据恢复器，访问点接入器，应用服务器，和**devops**开发器

OSX的同步和appmodel讨论过了，然后是其它。

解决其装机维护的问题：

我们的例子中，用了一台双盘的Mac mini（一般装osx server一盘作time machine backup）,一台macbook和一台iphone来发明，在MINI在这主机上绑定破解的server,我还配合使用了oray三件套（蒲公英，花生壳，向日葵控控a2），因为mini是属于无屏环境，只能通过网络显示器，如ipkvm的向日葵控控a2这种来解决装机和维护问题(Mini的设计要是加个小led屏幕和有限键盘，可以启动免去外插键盘和hdmi显示屏就好了,其实群晖也可以这样，只是它有一个不坏的rom，里面有bootloader，不用像普通OS一样装机)，Osx server可以提供多硬件管理控制和细微的企业各级人士权限控制。十分强大。利用上述工具装完机后，可以利用a2远控远程装机，oraybox组vpn,花生壳穿透建站等，后期还可以time machine恢复，icloud恢复数据，etc..在移动端也可以这样做。这种生态集成程序是空前的。

解决其多开虚拟机的问题:

唯一与群晖系列不好比较的是osx 没有集成vmm这种东西，还有docker，不过这也可以通过下载appstore或第三方的同类服务来代替。我下载的是破解版的paraelle desk（集成docker/vagrant可以当devops使用），当然其它方案组合也是可以的，，客服共体的优势不但是GUI是共享的，而且APP生态也是共享的。

原来的bcxszy，在序言中不光是写了一体化语言，还写了一体化uniform web/native选型，中途我们改成了JS，现在我们回来了，现在的bcxszy2,加了一体化平台选型，开发转用terralang，在整个语言选型中，我们追求的都是一种平台与远程开发兼容一体的语言。，从下一篇开始，我们写《利用terra++模拟cpp》,用terra++，是因为只有自己发明的语言，你才能用好,就像我们一直在发明自己的dbtinycolinux一样。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一个设想:基于colinux,the user mode osxaas for both realhwlangsys

关键字：**umwinlinux,**从文件夹中启动的**linux,user mode linux windows,iaas,baas,paas**穿插开发运行环镜，是原生装机系统，还是语言系统后端虚拟机，实机/虚拟机/**os**内部 统一操作系统。真正的应用程序级统一的**user mode OS**，用户态操作系统。用户态操作系统内核。

自古以来，像python,js,php这类动态脚本语言系统都严重依赖于后端虚拟机实现，毕竟，可移殖性是soft vm的重大作用之一，这使得基于其上的开发和发布可以做到伪“跨平台”（实际上是各大虚拟机在其上都实现了一遍），更有甚者，.net和java这些虚拟机更是提出了统一后端，使得常见的多语言系统有了共同的后端规范，基本上可以将包括上面这些语言在内的各大各自为政的语言整合到all in one和极致，比如ironpy,ironjs,ironphp based on clr。 —— 所有这些，不过是把不同OS本地上的各异性封装了一次，用软件再造一层抽象，有了统一的接口再在其中建自己的东西，这里的抽象与封装过程作为基本技术，在软件技术/艺术的各个层次频频可见。

但是遗憾的是，类似技术并没有上提到OS层，，OS作为规范硬件各异性并提供native dev and run的统一层面，与上面提到的python,js,php等langsys backend soft vm有异曲同工之妙，然而正如它们没有进一步发展为.net上的免binding ironpy,ironjs,ironphp一样，各种OS上面实现的APP规范和系统调用（各种os subsystems,etc ..）实际上是各自为政的，不可移殖的，unix posix与windows win32/64实际并不兼容，所以才会有各种CUI级别的cygwin,wine和各种用户层虚拟机方案（接下来会讲到）的出现，它们的工作正是为了统一这个层面。前者主要是为了运行程序，后者主要是为了开发/iaas化加速。

有几种特殊的OS：

现在的云各种虚拟OS的加速方案kvm,virtio,openvz等，还有各种虚拟机管理器virtualbox技术，还有vagrant开发虚拟机，特别是vagrant,随着开发的复杂化，建立paas,baas,iaas的穿插环境，在vagrant中建立起各种虚拟机环境，这种需求都开始变得很明显和频繁。

这些技术的出现，都可以称作是一种user mode os的层次的东西。而jvm,clr这样的规范和实现，一开始也都是工作在用户层的。有相同的架构层次和整合基础。

separated user mode os from kernel mode os

虽然OS这个层面并不需要直接考虑任何langsys，但是不妨这样想，在OS的实现层，如果有一套类langsys的soft“虚拟机”，用来代替os的subsystem（传统意义上的os subsystem api兼容机制并不足以提供太多的东西。），那么可以实际上有二种 OS:

一种OS是那些kernel的东西就足够，并不需要包括cui层次，只包括driver实现层次，是仅仅管理内核层次的OS。

而另一种OS不直接附在硬件上而是作为一个vm存在，专门用来负责除硬件虚拟化之外的其它任何应用兼容和开发层任务，就像jvm,clr，安卓内部的java虚拟机一样。

而第二种OS实际上可以完全采用第一种OS的kernel实现技术，只不过它全程运行在user mode下。以第一 种OS为meta os，后面第二种OS的实例可以在资源限制范围内无限开。 ——- 这完全类似于文章开头就谈到的：在langsys层提出clr,jvm，用它来建立起isolated langsyses的统一后端，达成最大兼容和可移殖。

业界类似方案有colinux，也类似openvz这类方案，都有二套OS内核。二者兼备才能跨内核和跨用户层都能做到高度统一，比如，通过colinux等usermodeos,实机OS可仅作metaos，而user os可以作各种虚拟层.

为什么是colinux？

以上这些技术在colinux中全被包含,它符合浅封装原生OS和不带来太多损耗的原则。 colinux实际上是user mode linux的一种，不过它是建立在以windows/linux为host上的只是不能以windows为guest。并且，它支持从某个实机盘和实机驱动。

为什么不是虚拟机管理软件和reactos这样的方案？

虚拟机太重。 reactos太注重重复windows已经做过的工作，忽略了新时代user mode os的需求。 拿龙井Longene来说，龙井也放弃了驱动兼容（它在最新一份解说中，提到类linux的andriod系已变成主流，windows才是需要兼容linux的，所以不必兼容）。毕竟多OS共存才是合理现象。本来就不必从那个层次兼容。弄错了兼容层次。它现在专心做应用兼容了。 他们都用了wine,希望ros能改正过来，向龙井靠拢，将roadmap改成先发展和完善user mode的东西，再有余力去发展驱动兼容级的实现装机方向。

这样的二套OS可以装在实机上，当用在实机上，用户可以在任意架构的机器上同时（注意这个同时）安装运行多种操作系统并不需要安装额外的驱动，性能并不会有太大的损耗，在装机上可彻底去除UEFI这样的东西，机器出厂商仅需要集成第一层OS及驱动支持即可 — 向用户透露更好的直接装应用的层次。

user mode os不但可用于装机，还用于语言系统和开发运行

再来说第二层OS用于作为langsys backend，如果第一层OS可以支持任何类型的第二层虚拟机，那么几乎历史上所有的APP开发/运行层兼容问题都不复存在了。比如，一台手机的实机OS集成了三个第二层OS虚拟机，那么它就可以同时运行winphone app,ios app,andriod app，还可以适用于云主机虚拟机，还可以用于任何CUI层的vagrant开虚拟机过程。

看来，提出一个先行于realhw os和考虑整合langsys后端的用户态OS，看来是潮流啊。。

xaas:大一统的user mode os

当第二层OS可以以vagrant方式被管理和使用时，它实际上变成了xaas，因为它可以为langsys baas服务了。

在我以前的文章《发布engitor》《发布enginx》中，实际上enginx +engitor是当paas和langsys baas用的，尤其是engitor还有engitor as visual editor service的意思，可以看作eaae service吧。

如果这些都可以做进我的msyscuione->xaas文件夹与engitor,enginx放一起，（比如msyscuione中可直接安装不同第二层user mode OS，比如colinux,再在colinux中装msyscuione langsys），而不用到像vagrant之类太多虚拟技术和云技术。统一用于实机，云端装机/开发。那么这个user mode os就是不折不扣的跨iaas,paas,baas等的综合xaas user mode os了。

或许还要加上我《一个设想：基于colinux，去厚重虚拟化，共盘直接文件系统安装运行的windows,linux》就更完美了。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

去windows去PC，打造for程序员的碎片化programming pad硬件选型

本文关键字：混合，多合一**OS**，封闭应用，有限应用机器。程序员的硬件选型，程序员的7寸**umpc programming pad**，移动编程机器

在前面我们谈到nas与chromebook的绝配，一个是webos，一个是web client os，群晖中的docker不但用于开发，运行，和devops，甚至可以用于”新式装机“，比如群晖中有lxqt,qnap中有多种guest os,他们大都基于docker技术或相似技术，如果说webos总是出现在host/guest os的组合中使server OSs能做到不断融合和generic，，那么作为客户端OS的chromeOS+chromebook，也有类似这样的演化方向么。

chromebook with fydeos PC选型:去windows,去PC

这个是有的，fydeos即是这样一种host chromeos与guest andriod,linux,windows混合的定制版，fydeos也是用类docker技术实现的三种OS混合，它的基础是google的Crostini，当然，可以用，和用得舒服自然，这二者的差别可以是一个天，一个地，fydeos能运行andriod和linux的能力是很彼此相关的，因为他们都是linux系，可是运行windows程序需要借助docker wine化的codewears公司的crossover层，实际效果只能是勉强带起，很不令人满意,这里面需要太多工作需要填充业界还没极尽完善 ------ 但不论如何，第一次，当这些生产环境能第一次结合，且真正第一次能证明能协调工作时，这是一个伟大的进程。

> 为什么这么说呢？从《除了linux，我们有第二种开源OS吗》一文开始，我们就在探求一种能实际用于生产环境的多OS实现选型，就像我们在选型多语言系统一样 ------ 我们的实际需求有可能是这样:我们刻意去windows，去PC，理想将一直只工作在某linux下，但经常或偶尔地，我们需要windows下面某个zip程序，办公需要wps生产力环境，至于游戏嘛。。就算了，我们只是先想得一个这样的办公生产工具，以上有限几个程序最好都暂时能工作在这个linux中，而又不想总是依赖一个windows pc设备-----未来，PC，windows和这些有限的windows app都在我们要淘汰的设备和app之列。
>
> 然而，在windows下融合linux较容易（colinux,win wsl）反之这个过程却难多了。业界努力多年，但这二种APP一直不能以自然的方式共处，并得于实用。----- 一般地，做win+linux混合os方案或混合应用的，首先想到的就是从kernel层着手，像龙井系列，也有从subsystem着手的，像win10 wsl，colinux，也有从应用层着手的，像virutalbox,也有从api层着手模拟的像cygwin,wine，这些都带来了超多的工作量，都是因为OS从入口处就分别过大，而APP构筑于OS kernel的生态之下，仅想模拟app绕过kernel是不现实的，或者说是非常困难的，------ 设想下从kernel层着手，只能使用内核调用层的翻译技术来使二个系统生态相处，打破了windows kernel内核本来就具有生态封闭的特点(虽然有开源reactos，但它本身发展未演示进beta)，强行揉合和翻译内核调用会带来超多的工作量。如龙井就是这样。windows本来就kernel和app不易剥离，想要集成更是难上更难。除非真采用为fake一点的方案。

这样的退而求其次的方法就是wine,加上容器技术，它可以用相对不原生的方法从APP层开始，这个时候性能考虑已经不重要了。通过使用crossover，fydeos终于可以第一次勉强把这些放到一台机器中。

程序员的新硬件选型:移动端a programming pad

在前面的文章中我们谈到nas与chromeos绝配时，视chromebook,pc,smartphone为nas的集散型结构的三端，并有利用七村小本加vim打造程序员的programmingpad想法，这一切的考虑都是为了向程序员-这个特殊的群体倾倒，毕竟，除了一个nas as mineportal，程序员依赖以上终端的日常实际上很典型:追求高效输入，碎片化文字创作或代码调试的人群，这cover了好多人使用终端的场景。

硬件选型上，当一个现代程序员的装备越来越碎片化，越来越自动化，那么其需配备的终端的演化方向又将如何呢，

access box:

在选型群晖或qnap时，我们总是要谈到ip盒子，frp这样的东西，因为一个mineportal必须首先是个mineaccess serverbox，access server的作用在于，它可以为应用准备一个access point和私网路由接入逻辑，使mineportal变真正的内网盒子。内网云。ip盒子的微道维系的口号是私人云接入商。而oray的蒲公英更接近这个概念。蒲公英就是一个带oray官方vpn线路和组网逻辑的路由器，排除这个他就是普通路由器，围绕mineportal的大部分应用都是内网应用，大都基于局域网ip ，如群晖的摄像头，如控制开机，如监控，或许我们还需要远程访问，但并不需要一个公网对外服务。这个VPN就可以。蒲公英X3附送的免费组网，一台路由器下可挂3个成员可以是路由器也可以是普通客户端，Oray的产品刚好弥补了一个类阿里云ecs云主机除了服务器本身之外所需要的远控，域名，外网服务，但实际上有了蒲公英，我们并不需要花生壳，向日葵，局域网控制也可用纯软件实现。。

除此之外，程序员还需要一个特殊的代码终端：

programming pad:

普通的PC太大，现在的手机全是触屏，实际上有时输入并不自然。程序员在地铁上这些碎片化场合无论用PC还是mobile，平板，都是极为不便的，以前有人试过有pocketchip，gemini pda，gpdwin,一号本，onebook我也试过，但是甚至gemini pad,gpdwin也是不好的方案，除了7寸键盘本身有点大之外，gemini的键盘设计得很好，然而实际上跟gpdwin一样它设置成桌面式放置并不是长时间手持防累的，这是smartphone和PDA的领域。而后二者要么没有实体键盘，要么尺寸不够6寸----对，我觉得能快速输入用的手机最大只能是6寸的而并非7寸的。

我最终选定的是黑莓的keyone,keyone2,priv实体键盘安卓系列。

实际上这样一个programming pad对硬件有要求，对软件也有假设和要求。

devops下的programmingpad ide

如果选定在一台6寸物理键盘的手机上，那么打造programming pad的最终的想法。除了与我在前面讲到的一系列其它软件上的选型相关，它应该还应有以下几个考虑：

programmingpad终端的OS不要求是windows，不要求是chromebook，可以是单andriod。它只要装上普通的APP端与远端的服务端设施配合工作即可。这个服务端必须是一个devops，它负责反聩程序员高速输入和调试源码的远端反应，谈到devops，我们在《docker as engitor》讲过，在《群晖docker上安装gitlab as mydockerbox,mysnippterhosting》时我们谈到安装了gitlab的群晖dockerbox也是devbox,因为它支持devops。其实本地也有devops,像cpp的sandstorm用的那套ekam，https://github.com/capnproto/ekam，甚至在jupyter中也有，如基于jupyterhub多人版的jupyter实现的mybinder.org中，利用了docker打造了一个ide和类似gitlab webide的环境。

这一切都是因为devops的基础，都是engitor需要讲到，需要以此为基础构建的东西，所以该有的，除了他们各自内部的不同，其它他们该有的，都会有。

然而就是上面谈到的那个APP的要求了，它显而易见应是个liveide，然而与普通的liveide相比，它应该也是特制版的，比如它要受屏幕限制，因为工作面积小，碎片化时间短，它还要更“智能”，比如像个对话机器人式反馈结果，当然通常情况下，它只是一个配备了vim的更好用的代码编辑器，或一个更形象可观的可视化的IDE，更或者，他们的组合 --- 比如像ellie一样的就好，见我以前文章。

设想一下，如果是jupyter devops下的ide，它应该是mybinder.org一样，有源码归类的树形目录，有远程docker连接调试，如果是gitlab下的，就是那个web ide。还可以像ellie一样分成三屏，在手机上，或6寸umpc上，滑动卷屏，随时查看源码，html显示，程序运行demo和debug。。。

全linux的组合可以作为生产工具（一个nas,几个终端）。对于封装的个人云应用，我们交付的服务端，以及是终端。什么是封装的私人应用呢？那就是有一个mineaccessbox的，以mineportalbox为运行环境，可以programmingpad终端上运行其APP且调试的组合。

关注我，关注"shaolonglee公号"

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

利用colinux制作tinycolinx，在ecs上打造server farm和vps iaas环境代替docker

本文关键字:将tinycorelinux装在硬盘上，custom tinycore linux kernel,tcl3安装使用方法,tcl安装到硬盘,自定义linux rootfs,利用colinux代替docker组建容器。单机端口反代重用技术，内网转发复用端口

在《阿里云上利用virtiope+colinux实现linux系统盘动态无损多分区》中我们用colinux实现了一个uniform pe装机环境的东西，在《利用colinux,免租用云主机将mineportal2做成nas》一文中，我们利用colinux结合mineportal打造了一个多用实用的nas，在《一个设想：基于colinux，是metaos for realhw and langsys，也是一体化user mode xaas环境》中我们提到colinux还可用于类vagrant开发虚拟机，软件兼容层等等之类的东西，xaas环境，而其实，只要上升到host/guest层面，作为guest的colinux还可以完成更多的东西，比如这里接下来要讲到的：利用colinux切分服务器资源，打造你的专属serverfarm --- 其实，它也属于《一个设想：基于colinux .xaas..》提到的xaas的一个应用：它可以模拟docker容器，将ECS分为多个子colinux系统，比如，对于一个1C核1G内存的ECS，我们可以根据128M为粒度大小，利用colinux能在conf中设置mem的大小的能力，将ECS资源划分为512/128=4个colinux容器。除了给内存配额在tinycorelinux wiki中还有定义governers限制CPU的能力和讲解。这一切都不需要利用到docker这种过度虚拟化的方案带来的虚拟地狱缺点：比如docker利用分层联合文件系统，用户难于维护。

为了实现可用性，我们还必须作一系列改进，比如，由于128m内存限制太小，我们必须利用专门的colinux os发行版而不是巨大的ubuntu etc..，比如采用tinycore linux os，它可以做到以至少10M的服务器核心而存在，这跟windows上的boot2docker使用定制版的core linux os是一样的道理。选用tcl的另一个考虑是它有独特的发行包机制，它的发行包比较精简，专用的软件包发行限制可以避免VM用户装乱七八糟的大软件，第三，tcl的处处即挂载的cloud run+可持久机制(整个根可挂载到内存livecd下或其它介质，home目录可挂载,tcz loop mounts必挂载)使得即使给了用户root也不会轻易破坏系统适合VM使用。

对于一个真实可用的vm container环境，还会有其它高级课题，比如，一台ECS只有一个80端口，多个内网colinux VM环境需要重用80端口出网。无论如何，下面先来搞定将tinycore与colinux结合的问题：为colinux制作一个精简的tinycorelinux发行。

制造一个精简server发行版

这一切需要在另外一台linux上完成，比如直接利用我发布的colinux14.04版本:

一开始我参照网上《将ubuntu8.04 iso安装到colinux的方法》先试验下载了最新的各种iso，通过conf文件中直接挂载/dev/cobd1=xx.iso,root=/dev/cobd1,initrd=tinycore.gz的方式企图进入，发现最后都不能进入，有colinux initrd的问题，有tinycore.gz文件系统的问题，有colinux内核的问题。光盘方式看来是不行了。

不会要重编内核，或者重封装rootfs吧？

但其实硬盘方式加+低版本microcore3.8.4.iso是可行的，我重新做了一个1G的硬盘，在colinux中mkfs.ext3格式它，拷贝提取iso中的microcore.cpio直接释放到硬盘，tinycorelinux就开始运行了直到出现登录用户提示符，即cpio -idmv < microcore.cpio到根目录，然后用colinux引导新的硬盘系统（原colinux vmlinux可以无修改，initrd.gz也可以利用上），这样天然地就是tinycorelinux的硬盘模式了。（initrd.gz注入后，再重启，提示scatter harddisk installation mode后进入login提示，用tc用户无密码登录），我把它称为tinycolinux。

tinycolinux的conf中还可以支持tinycorelinux中提到的各种bootcodes，比如root=/dev/cobd0,home=/dev/cobd0,opt=/dev/cobd0,tce=/dev/cobd0这些,由于tinycorelinux会搜索分区上的/tce目录为应用下载目录，所以我也在\新建了一个，否则默认tce-load -wi xxx出来的options会出现在/tmp/tce中，(硬盘模式下home,opt,root都出现在当前所在的硬盘根下，在/下建一个/tce目录，有跟conf中设置tce=/dev/cobd0同样的效果)。

由于一切都是在硬盘完成的，整个文件系统都是可持久的，除了应用安装不用处理其它持久化问题了：

TinyCoreLinux持久化问题-用户数据和应用扩展保留

tinycorelinux一开始定位live iso和cloud模式，体现在它能在liveiso完全无持久,和有持久介质的多种场景下运行，其root文件系统核心集和应用扩展settings在每一次重启后都是fresh的重启就丢了一切会话和应用扩展和其设置数据，因为一切都是挂载到外部持久的条目或加载到ram的tcz扩展镜像，对于前者，它实际上就是挂载到持久介质的入口而已 --- 正由于这些都是挂载hook点，所以可以集中卸载，，更改依然在外，对于后者，ram下天然不能持久，二者可以维护一个干净的重启后环境，

而对于必须带入下一次重启，或整个文件系统的持久化，除非你指定保存逻辑和定义保存条目 --- 注意这句话，稍后就会谈到。

有三个可配置挂载的目录，/home,/opt和/tce，你可以挂载全部三个到可持久外部介质或选一二，设想tcl在完全无持久介质的liveiso情况下启动，它根本就没有持久能力，但若指定了至少一个可持久目录到外部介质后启动，它就可以在外部介质上得到更改保存，但这些改变不会被带入到除了这三部分之外的任何根文件系统的其它部分。

有一种情况比较特殊，当tincorelinux整个文件系统被置于入硬盘并从硬盘启动时，实际上硬盘整个就是可读写的（norestore bootcode天然启用，/home,/opt,/tce都是现成被默认定义了的可持久目录）。

而至于对于要带入下一次重启和根目录文件系统的那部分持久和更改，你可以指定restore：比如bootcodes定义了restore，和/home到硬盘和新增了.filetool.lst中的新条目，它就会产生mydata.tgz备份/home和/opt，到这个持久上---你可以定制包括/home,/opt的可持久路径和bootcode中指定restore所在的路径，进行一次filetool -b，并在下一次由系统恢复。

对于应用，同样的方式(如上bootcode指定)，可以有定义了一个保存在可持久介质上的/tce目录，比如它在硬盘上，/tce下的options和onboot.lst的更改就能持久化。就能将*.tcz动态挂载进来（tcz是一些只读文件系统包，挂载进来的时候是挂到内存中）。要注意在这里，应用加载逻辑是持久过的，但应用依然留在内存运行。且应用的设置部分还没有经过显式持久化。

所以进一步地，如果tcz要带入系统作更改，你依然可以结合.filetool工具和机制，将具体tcz安装后需要持久的部分持久化到可持久中或者ln -s部分目录到硬盘，还可以在在/opt/bootlocal.sh中定义开机利用这些目录持久的逻辑。

将tinycolinux打造成完全的硬盘系统

到此为止，应用依然是靠一次性加载到内存来运行的。针对已经安装好的tinycolinux，我们需要纯粹在硬盘上安装运行的应用扩展：

（虽然对于live系统和3个目录组成的定点集中维护来看它是最佳的，但实际上，除非对应用本身进行定制，否则应用安装过程实际上可能会对整个系统文件系统产生更改，而这也是其它linux distro软件包的默认行为），，况且，我发现tiny core linux有几种包不一样，像nginx和mysql，前者会loop mounts，后者不会产生loop mounts，mysql安装tce-load -wi安装后会留在/usr/local目标中持久，而nginx在-wi后重启系统仅留下一些软链接，不能统一处理，这反而给安装造成了困扰，为了追求更自然的类现在包管理机制的方案和统一省事的安装方法，我想出的办法最初是直接下载tcz包释放到目标，因为tcz包是简单的文件系统打包大都释放到usr/local目录也比较容易手动安装：

1）下载应用时只tce-load -w下载到options

绕过tce-load -wi会创造loop mounts的过程。改用tce-load -w而不安装。

2）从这个地址下载http://mirrors.163.com/tinycorelinux/8.x/x86/tcz/squashfs-tools.tcz，提取二个可执行文件放到把它放在根目录/bin中

利用它来进行手动解压。

3）然后视tcz内容在windows上用7z打开查看看它是要释放到哪的，

其下载地址往往是/opt/tcemirror中条目后加/3.x/tcz/包名.tcz得到的，手动解压，unsquashfs -f -d / /tce/optional/nginx.tcz （这里以nginx为例，它释放到/下）

处理好各个deps的tcz,对于nginx是openssl和pcre。如果所有的deps都安装了还是发现不了so文件，重启一次必发现。sudo nginx会发现不了libprce.so.0

然后在opt/bootlocal.sh中加入随系统自动启动条目，对于openssh是/usr/local/etc/init.d/openssh start，对于nginx是nginx -s start吧。

完工！nginx在重启后自动运行！要卸载时只须处理/usr/local目录。

其实tcl这种live机制也可用于装机代替virtiope，这是后话了。 除了这些，当这些colinux vm用于建站时,还需要nginx反代多个VM重用一个80的技术，其实说到内网转发复用端口，《基于colinux打造nas》一文中也可以用它来出网。制造发行版的二大基本组件，os本身已经解决了，还有toolchain的问题。根据我的《host2guest guest2host nativelangsys及cross compile system》一文，虽然tinycore linux有gcc应用包，不过我倾向于把它放在windows hosts，来编译制造tinycore linux可用的目标。好了。这些都不讲了。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

enginx:基于openresty,一个前后端统一,生态共享的webstack实现

本文关键字：**openresty,enginx**

webstack的前世今生就是一个重复造轮的过程，它的目标是将本地程序栈弄成分布式b/s web，其实这在语言端可以做(比如语言模块的http unit，然后是一层层我们从桌面时代开发最基本的socketapp开始，http封装之后也许是一个aysn网络io库，最终到达语言库级的webframework直到专门的独立程序支持，也许这个时候人们发现那个网络io库可以独立出来作为一个server，再比如第三方容器在这种需求下很容易出现，流控安全等需要也会泛滥)，于是终于发展到用独立的服务器OS组件来实现这些强化，形成专门的产品来做，体现在开发上首先是webserver+CGI处理。web作为b/s在架构上假设有服务端程序存在，而cgi就是开发web程序的语言同webserver交互的扩展，动态语言将运行结果转成web page app的手段。像mod_swgi，mod_php就直接将phpcgi做到了语言。如webstack.语言则屈居之下。—— 这完全是语言，独立件，一方做大了包裹另一方的关系但二者始终是一体的。

以前webstack都是语言加各个产品小件的堆叠，像wamp,lemp,wapp,lnmp,mean…等，除去语言和OS服务这些大件，显然存在前后端，首先是page server，然后是存储后端。mysql,rawfilesystem,document stor（文档xml强调的是文档语义，能存文件和文档是其第二层含义）等。还有各种为了其它功能新加进来的增强件比如种新的本地/分布数据库memcached,本地件imagicx。。

PS：其实这些都是模拟桌面时代的appstack,人类其实在各层次复用同样的方法，解决方案和产品，形成各种类似appstack,webstack的其中明显二大件之前端部分模拟的是desktop app时代的page,后端模拟的是filesystem等等，毕竟是分布式中已经定义好b/s规范，连协议http都规范好了，有这二个产品，就足于构成一个webapp stack必须的界面和存储了，只要有app domain logic，就可以产生一个app了 —— 同道理的是mobile app那些，这里省略一W字…… 以上备注栏里你看到了应用程序形成和选型历史的大架构，好了打住，继续：

存在几种流行的webserver(有的有容器比如tomcat支持，有的对语言支持好比如apache rewriter强大,etc..,有的功能单一，比如nginx静态页面友好在流量控制方面功能强大)，人们有时还混合使用它们，这使得webstack前后端分区呈现更复杂的前前后,前中后，前中后后等结构。

这里要来说的是大头的nginx+apache

我们不需要apache

其实作为一个webserver,它的本质是流量引擎+webserver,当它与前端语言cgi沟通时，它就是webserver,当它负责与后端+前端一起上时，它就是流控引擎。很显然地，nginx最初的意义是分布式流量的“enginx”，在这种意义下，nginx能管好流控这是它最大的责任和优势，而apache显然做得有点过了：

apache并不仅是webserve其实它还提负容器的责任，这种在用户态重造一遍的事应该由语言层（比如msyscuione中的包管理或engitor as paas中的语言层来完成）。它跟nginx一样存在流控与各种语言对接的后端，然而它走的路子是，由语言来完成对数据库后端的对接，apache负责与语言沟通，这其实是webstack通用的设计和应用路径，只是这种情形下，webstack变得有点碎片。各种语言和存储组合起来就是一个webstack，，这样看似自由，实际只能二块二块地用，十分固定，于是你看到的不是wamp就是lemp，而meam这样的东西是不好用的。

总结：，我们没有一个一体的webstack生态圈产品，各个产品都是语言绑定的。具体前后端紧联系的。甚至系统绑定的。原因是，webstack中webserver的功能定义不清。

nginx=engin x: 强大，专一的流控单一，也是前后端开弓的中间单元

而对nginx的合理应用可以完成做到用nginx+其它产品打造一个干净的前后端：

nginx它没有容器不像apache，因为这是语言层要干的事。

在这点上，apache于webstack必须有的地方没有，nginx却全有。我们也不需要nginx+apache的架构。我们完成可以去掉apache,语言直接作为容器，nginx直接作为语言容器的前端，以direct servering langsys的路子进行。

而且nginx的流控实在是太强大了。它可以做到一体化流控（负载，访问点控制，QOS，反代）与安全服务器。

而且，lamp,wamp一条龙的webstack设计者们肯定没想到，nginx它不但可以server前端，它实际上也可以server后端，比如nginx直接与mysql构成不需要通过php的phpmyadmin版本。

这有什么意义呢？

它使WEB件，从语言件彻底分开，实现了诸如，提倡nginx前端直接与后端mysql等通讯，仅要求语言提供php-cgi之类的东西而不要求它们提供php-mysqli之类的东西，构架上清希化出了一个“整合的独立的，webstack”，而不再是分散的wamp,wemp,mean等等。从此不同的语言导致的开发，发布的，架构上的区别都不存在。都是一样的从nginx为入口的体系，它掩盖后端那些子件的复杂性和开发维护必要。更重要的意义：打造了一个干净的多语言webstack

大一统的webstack

openresty的架构有趣就在于此，它是个足够创意新奇，大一统的产品，它对前后端都进行绑定。

nginx之于db,storbackend(openresty一志整合)，就像jupyter整合语言件backends。我们从此仅需要一套 “xny” webstack — x平台的y语言，及中间的nginx，就可以做到统一开发和发布，最最深远的意义还在于：有望借助enginx，所有web程序共享同样的产品生态，统一发布，无沟运维。

下载（本地下载）：

以下提供下载的enginx，在openresty基础上做了适应msyscuione->各个langsys的适应。与engitor(paas)天然互补结合:一个提供语言与容器，一个提供安全和最终的paas服务。。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

服务器的“服务器”,enginx之于分布式部署的创新意义：使任何服务器程序秒变集群

本文关键字：**nginx,**元服务器，单机集群,分布式集群，集群引擎

虽然webstack往往被做成集群来进行，但即使存在那样的必要也很难为随便找到的一组服务器搭个集群，enginx就是用来做这事的，继《发布enginx:基于openresty,一个前后端统一,生态共享的webstack实现》，进一步地，enginx之于服务器部署的创新意义还在于：它可以使任何分散，逻辑上相关的一组服务器建立起前后端的拓扑，可能这些srvs并不是webstack使用的那些。

engin x – 从webstack到common svrstack:通用分布式服务器引擎，集群引擎

进言之，它可以使任何服务器程序纳入一个统一的生态，就像openrestry之于web stack srvs一样，因为openrestry是不止webstack srv的srv，而且还是通用服务器子件的集群引擎。所以其中有dns srv也是合理的。。。比如你完全可以用enginx做vpn服务器，搭配游戏逻辑服务器做游戏服务器，等等。

这可以让任何分布式环境，变得自定义。可定制，高伸缩。而且，注意这个而且，仅仅在配置层面就可以完成。不需要涉及到开发。因为基于openresty的越来越多的插件是用来解决这个问题的。

enginx就像是集群定制器，服务器的服务器。通用分布式服务器/集群引擎，甚至可以集成到一台单机。

从分布式到单服多站互转

而且，你可以用enginx做单机集群，这对集中运维，单机多服节约成本方案都有帮助。拿游戏服务器集群和游戏开服来说，对于一个游戏服务器。比如有数据服务器，逻辑服务器地图服务器，登录服务器，消息流转网关 等游戏产品。web也可类比：一个大型用于生产环境的web服务器，拿aliyun的产品方案来说，有rds：就是多个数据库服务器。 ecs：放业务逻辑的服务器。 ocs:放存储逻辑的服务器，比如文件服务器。 至于slb，，其实就是把以上东东连起来的网关。（使分散的成为一体，且完成一些资源匀衡的任务） 就是游戏服务器那一套。或者普通TCP服务器的那一套。以上这些可能分散在网上不同的环境下。

比如：集群环境中的每一个分布式子件都可以是一个进程，一台机器，甚至一个远程进程中的应用，可以是软件，有些是逻辑处理的，有些是流量控制的，有些是负责安全的，这在编程上体现为复杂服务器环境设计，即svr程序的开发 。需要涉及到线程池，异步IO，文档协议库等等，而 在运营和运维上来，随着业务的扩大，体现为增加新的进程，新机器的过程。比如对于，游戏运营来说，要开更多的服，可以在一台服务器，但是无限扩展并不总是好的方法，更高IO更高并发可伸缩总是以不断增加升级配置的成本来解决。也提高了运维成本。 但是，完全可以通过engin x,在开发上，配合enginx，仅需要配置一台逻辑服务器。在部署上，分散的服务子件都可以用软件模拟。按软件模拟微缩到原先的复杂服务器程序设计的层面到一台单机，配合enginx重新设计，可达到更高灵活度，可大大降低费用，且可极致利用完服务器资源，

还是来说阿里云上做站的那些产品和方案,不一定要用业界那一套。感觉上ocs,rds,ecs,slb这些鬼，可以对站的服务器环境进行再抽象，使之变成一套自然，前后协调的天然单元。比如用游戏服务器的网关模拟webstack中的负载。只要能保持单机高可用，在限制资源配额内，就能做到真正的那种单服全集成的方案。（VS多服真正的负载均衡）

而这，其实不过是借enginx,在抽象上的一种做法在部署上的一种重组方案而已。

以后的文章，会大量例举这方面的例子。比如nginx支持反代和负载，可以用于集群。再讨论其综合运用方面模拟现有各种wamp,lamp等的技术细节。来讨论用enginx 模拟通用的游戏服务器环境和强化/整合/替换各种现有web环境栈的做法。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

engitor:基于jupyter,一个一体化的语言,IDE及通用分布式架构环境

关键字:工具层**devops**

很难为jupyter这样的一个东西定性，它最初只是一个增强的python repl环境，后来变成了CS架构并支持了多语言，S为语言kernel，C为notebook,console,qtconsole这样的东西，可以分开部署使用。

> IPython 3.x was the last monolithic release of IPython, containing the notebook server, qtconsole, etc. As of IPython 4.0, the language-agnostic parts of the project: the notebook format, message protocol, qtconsole, notebook web application, etc. have moved to new projects under the nameJupyter. IPython itself is focused on interactive Python, part of which is providing a Python kernel for Jupyter. 如果想快速尝新，下载windows下Anaconda的py发行版，第一个用ipy4是Anaconda2系列的Anaconda2 2.4.0版本. 我们当然关注的是jupyter system与传统CS程序相比的那些不同点：

首先，它不是应用，而是侧重语言系统。要说它是应用，它也只是“编程教育利器”，“一个多语言在线IDE”，是语言系统方面的应用（so,也是CS应用）

其次，它至少有以下特点,先来说表层的，那些直观可见的东西：

jupyter是一个分布式IDE

1，以语言为后端，客户端接受服务端的执行结果，直接输出执行结果。以页面上的cell为单位。 2，CS二端组成了一个分布式的DEMO SHOW系统。

总之就是IPython,他的一个很大优点就是可以把代码写码过程、运行结果展示合在一起，并持久保存在一个notebook中，并由jupyter支撑这个过完成程。

再来说点深刻一点的：

jupyter可能是一个自带开发发布的分布式devops计算环境

它增强了语言IDE，它是分布式交互开发环境（做成了CS和WEB嘛，大凡与WEB沾边的，应用架构上已属分布式）。

它改变了开发协作方式，人们发布ipynb,就可以共享源文件和执行结果，而不需要下载到自己的机器上利用本地语言系统运行一次。如果这个结果可以直接形成应用(分cell的code block块可以像语言源文件和语言内模块一样组成软件)，这足于给编程界带来一股强劲的创新了。发挥直男不由分说的特点来说简单就2点不用怨我：

第一，它改变了软件协作的方式，使ugc,ugc=user generated content，这里c就是coding或codes,它使W人组件开发做到了线上并直接存管结果。

PS：这什么意思呢？

如果github是人们递交静态源码仓库的地方，开发者是以offline的方式参与开发。那么如果有jupyter hub，那么它就是组合正在运行的软件组成更大软件的地方。这句话中隐含了组件这个词，组件是现代语言都有的大头，实际上简单来说就是，demo就是组件,可放置工作的dropin的复用件，能将运行中的程序部件作直接聚合积木搭建的东西，都是“组件”。如果这些组件可在网上直接整合，运行结果也托管。那么它立马可以产生一个“动态github”。如果你的app够小，一个ipynb就够。 这样，用户可在线上直接编程搭APP。因为开发用的语言系统和运行用的环境都在线上，结果也只需要呈现在网上。用户只需要复用ipynb贡献codes这些，作为ugc中的c即可。这对需要用户贡献用代码完成逻辑的社区应用系统或游戏应用大用，它使厂商直接接上第三方扩展者。可以极大快速丰富一个应用生态。

第二，它的可调试特性，使W人组件开发的无门槛性降得最低。因为它是个DEMO effect instant show system.

综合起来，它只是将IDE发展分布式，且其架构和产品定位上也可以作成“动态github”之类的东西而已，能理解到这层已经很不错了。

附下载地址了事(软件取名engitor有engitor=”engine tool editor“的意思因为受jupyter支持的语言系统应该到了toolkit直接搭应用的程度了，是编辑方式生成程序的内容生成工具和演示系统，软件已整合对msyscuione/langsys/qtcling的支持，下载后解压到D盘msyscuione下)：

下载地址见原文

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

比**WEB**更自然，**jupyter**用于通用软件开发的创新意义：使任何传统程序秒变**WEB**

本文关键字：**online language,**在线语言系统**,jupyter,ipython jupyter,**在线编译器，在线解释语言**,engitor**

在《engitor:基于jupyter,一个一体化的语言,IDE及通用分布式架构环境》一文中我们提到，jupyter不止是一个分布式IDE它还是个分布式架构，更准确地说，它使任何使用engitor开发的程序变成WEB架构下的程序，而这个“WEB”，是jupyter webiz之后的web,然而它比web更自然，更强大。

jupyter作为一个极具创新的普通的产品不可忽视性，在于它是一个能改变现有软件开发，发布，甚至教育，用户体验端的东西，使之秒变WEB，做成WEB架构下的该程序版本，正如jupyter主页上提到的interacting computing，利用这二字官方强调的重点还主要在这：可用于编程=可用于开发布署=可用于定义一个appmodel，也就是利用jupyter可以达成一个计算。这个计算就是WEB化。

因为jupyter system实际上是作为一个一体化的多语言开发测试部件live code,debug+demolet show engine和ide,meta container环境。而存在的。有点拗口？

第一，它增强了分级容器的范畴。使程序接上**WEB**式的组件开发环境，以及**UGC**支持。

普通的容器就是PAAS，RUNTIME AAS，像gae这样的东西，lamp这样的东西，在ipynb面前只是基础建设。前者没有细化到具体应用级，是从云服务到语言服务到语言框架服务的分级细化过程，提供服务的厂商只能算是ISP。而后者可以做到应用生态内部。为具体应用定义一个由.ipynb 组成的demolet定义的容器环境。

jupyter将应用接上了一个任意式的demolet容器，如果说传统分布式方案是从云服务到语言服务到语言框架服务的分级细化过程，那么jupyter就是直接具体应用分布式。

jupyter是langsys as distributed srv,langsrv,这跟容器/applicationsrv —- 语言库api as services 不同，它更适用当前者完成了之后，要将容器运用到具体应用层之后的情形，

PS：语言即库，库即API，API即组件。始终要记得可复用件在编程史上的演化（特别是其与平台，语言系统结合，脱离语言系统后变成分布式可开发复用件之后那些形态），可复用件必定存在一个容器或hosting backend.接下来谈到的分布式API也是如此。

第二，它增强的是**WEB**等分布式**API**，从程序的开发发布方式变成类**WEB**的分布式架构。

分布式架构要纳入被开发，首先要API。程序的最上层无非开发发布，最原始的开发件叫API，最原始的部署件叫库，传统WEB和桌面分布下，都有自己的方案。从本地组件到到分布式组件，到接口到服务性API，到脚本（甚至开发件和部署件在脚本环境下最终做到了统一，这就是组件,demo即api，能在语言环境下动态运行services方式被识别为api的，都是组件），都要跨网络和socket这层。WEB是直接整合HTTP。更通用的WEB是退一步将HTTP换成了websocket，可是于任何方案将API做成分布式，都免不了要解决网络这一层。

jupyter的架构最上层CS二端用消息通讯，消息是文本化的协议表示。更接近语言处理支持需要和持久化定义需要。比如json,xml-rpc,soap就是用对象化的方式来持久表示协议的方案。

jupyter基于消息，可用于BS，CS，它免HTTP是一种比HTTP WEB还要通用的通用分布式CS协议。是另一种websocket之类的等价品。它使纯粹一体化的CS式WEB变得可能，即jupyter WEB（jupyter console,qtconsole as common jupyter app based appbroswer）。甚至不用到websocket和jupyter notebook时免传统浏览器。

第三，它增强了**WEB**等综合通用分布式**APP**的范畴，使任意程序变成**page based applets**。

这里的APP用来指终端程序（VS库框架等），用jupyter来构建WEB，代码和运行可维持only in a page unit…发行单元以一体化的ipynb page(含界面+逻辑一体化不再是分散的html+服务端处理脚本)为单元。是富文本界面（js+.ipynb needed)和渲染过程的传统WEB界面的等价品。而且，天然CS，no bs式的web可分离部署是个巨大优点（有时候省事做站要求我们将逻辑后端和界面大量静态资源的前端分开部署）。

综合一，二，三，api的发行直接源于语言系统并如一所讲提供了由任何.ipynb demolet组成的容器环境，能采用WEB开发架构且脱离浏览器的分布式程序，这已经是个“通用分布式”的WEB环境了，于此，jupyter做到并增强了WEB。能将任务jupyter支持下的语言系统的任何程序变成WEB架构。

一切的一切，是对于分布式架构开发部署来说，jupyter的断层和抽象切入面是：从语言层就集成cs和WEB支持就开始将自己化身为langsys based srv that micmicing a websrv，，所以免bs.免分布式API..软件界伟大抽象的二大特点，一是选择正确的粒度选择，另一个就是jupyter聪明的断层，从精巧的切入面去正确的断层抽象。当然这可能是jupyter设计者没想到的而我只是个事后诸葛而已。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

demoasengine,postasapp–paasone于软件开发运营的创新：所有人共享一app code和demobase

本文关键字：利用**nginx**和**jupyter**打造开发发布运营教育一体的多语言**paas,**内容创作工具**CCT**，多人协作平台**UGC**，**demo as engine,post as app,**云语言系统，云开发社区

在《发布engitor》中我们说到，基于jupyter的engitor使语言系统云化，通俗来说,它是一个带IDE的（这个特点不可略）语言系统和ternoda等web服务件紧密结合一体的东西，其面向语言后端kernel自带notebook客户端as client的cs/bs特点可使jupyter应用直接从架构上WEB化,与此同时，其作为云语言系统IDE可按受用户输入并instantly show demo effect的CCT特点，使该应用开发上的东西接上了用户UGC，本身可作为demolet engine可用于UGC运营。

在《发布enginx》和《发布gbc》二文我们又谈到，基于openresty的enginx有强大的IO流控和定制胶合能力，不但可用于web集群组件的聚合搭webstack也可用于普通服务器搭任何自定义srvstack，且搭配一门脚本，enginx可以将其发展成这门语言为后端的领域逻辑服务器且纯粹脚本组件化的方式进行，再结合openresty中lua可去专门服务器化的能力，enginx可促成srvless的直接语言后端的干净srvstack系统，同时跨game/web appmodel and mixed problem domain生态一体化。

好了，恶补到此，打住！ 以上这二个产品中，对于语言系统作为后端和其它服务器作为前端的处理特点是同样鲜明的，这就开始有了技术整合这二者的基础和形成一套baas->paas容器的意味，那么，整合这二者 - 可能其一种技术实现方向就如同“走enginx的jupyter engitor”之类的东西吧（比如，让enginx代替其中的ternodo，让enginx中的语言系统engitor kernel化，加一些容器方面用nginx实现域名流转的事）会有什么特效呢？我可以告诉你 ------ 它除了有二者固有的前述优点，甚至还可以使任意应用和架构达成“一栈”结构，因为基本上engitor可以做到无客户端，而enginx可以做到无服务端，无不是真正没有，而是可免或可整合的意思，所以如果说bs/cs是二栈，那么去二者的过程结合可以促成一个一栈的东西，这个“一栈”架构我将其总结为paasone架构，这个架构简直是万金油：

免网络协议，免客户端开发，免重造轮子因为我可以复用mods，免API因为自带编辑器，免装调试器编码器因为带云语言后端，免UGC编辑器，免容器

而驱动这一切的，就是engitor+enginx支持的demo as engine,post as app特色这几个字，下面试从开发、部署，运营，实践教育工程角度来说一下这个集成化的优点，先说engitor+enginx之于开发部署方面：

app lvl一栈体系：让任何程序变身可开发引擎demo as engine,让任何程序自带CCT工具post as app

为什么说这二者的结合是BS、CS双栈变一栈，因为开发上传统需要面向二端独立开发现在真正变成了一端（没有了服务端或没有了客户端开发需求，paasone集成了它们为一套从langsys到applvl的demobase和codebase,以前的开发第一次真正意义上地进入了具体应用as engine的境界，发贴即是开发应用）：

详细来说，客户端方面，engitor的ipynb小程序直接识别为网页或qtconsole结果，不需要开发网页或专门的客户端UI，一种语言通吃服务端客户端，这个跟JS用于服务端与客户端开发不同，虽然是一种语言那至少还存在二个端全栈，paasone中只有一个端多种语言一栈（为jupyter集成多种语言kernel可以带来多种开发），而服务端方面，paasone中直接有了appstack as srv，和nginx,免容器，脚本组件服务器系统。开发上复用即可。搭配beanstalk和pbc这样的东西可以定制协议和实现交互语义化。以脚本语言解释器为worker的paas,baas，这一切，比传统多语言paas需要自己定制成bs/cs,而paasone天然转化其为uni appmodel成applvl 级可用的engine，比docker更紧密langsys，更合理，更轻量。始终要记住的是(这一切都是发生在applvl和demolvl层级的不要涉及到语言，容器那些基础PAAS)：

故， 1,在开发方面,paasone允许用户：不必为一种appmodel发明特定的客户端，engitor中的qtconsole使得一切变成app broswer 2,在部署方面的集成化优点：不必为一种appmodel发明特定的独立服务器，这是因为enginx是uni appmodel container,直接免专门服务器，且可以混用web,game同生态结构.. 这种统一化的微cs/bs架构跨domain的paasone选型架构，可以做到免容器，比docker之类的更轻更清希化语言和直接化应用的概念。 再总结一次：切记！这二个特点，使得任何程序本身就是“微APP引擎”和CCT工具。

免运营接入，比微信小程序乱入更合理:让任何程序直接变身UGC平台

开发上的东西，一旦它有UGC，，就接上了运营啊。。平常我们都是线下开发，而engitor允许线上直接coding，comitting，在线即运营。这就涉及到了用户和用户再开发扩展。微信小程序这样的方案是以用户和社区为基础，建成开发，facebook这样的也是如此，它们都没有明朗化一个“demolet engine”的概念，应用容器可以是OS级的服务，也可以是开发级的应用服务器，也可以是demolet engine。

而engitor+enginx显式化了demolet engine,微信小程序明显没有以demolet engine为基础的PAAS，显得更合理。

用于工程，和实践和教育

postasapp的微架构为运营准备了条件，对于实践和教育也是极佳的，这就是jupyter最初作为教育品出现的最初作用，paasone发展了它，在面向新手的亲善度方面：

1，无须客户端开发，其有UGC，天然自带作为engitor tools用。2，丰富的服务端开发mods，其可作为post as app 3，uniform appmodel and domain 4，只需面对一套语言和纯粹脚本组件可复用环境。

一句话，微APP架构，更容易促成实践和用户贡献。

所有编程要涉及到的角色的一体化选型：共享同样的**1ddemobase**选型

整合enginx+engitor成paasone，我们最终要达到的目的是联结起一个应用所有生态的参与者，实现人员，开发人员，应用开发人员，管理者，用户UGC扩展人员，最终用户，这些当中所有的人员，表现为，进入一个应用后：我调出客户端，就可以云开发去试用这款软件。

(这个paasone，未来可能将engitor+enginx结合体具现化并用它命名），这应该是1dddemobase基础建设方面最大的boss product了。对应用minlearnprogramming,micropractise,这应该是microapplet demobase了。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

paasone的创新(2)：separated langsysdemo ecosystem及demo driven debug

本文关键字：**visual demo driven debugging**，**engitor**编程，更好用的**qtquick**

接《demo as engine,post as app – passone:engitor+enginx之于通用软件开发/部署/运营/实践教育的创新意义》文，我们说到engitor可视作是enginx针对于语言系统分布化的一个特例，可以按enginx针对通用服务器程序分布化的方式打造成“分布式语言系统”（比如分离出ternoda和language kernel服务端，作成enginx下的子服务组件使之更合理化），我们还说，engitor是个天然的IDE编辑器(除了提供的qtconsole as language shell only还可以开发更多的编辑器服务as enginx组件，甚至强大到类似photoshop或游戏世界编辑器visual editor的那种 -not offline,but ingame editor)。编辑器也就是engitor一词”itor”主要的所指，其对于

1，运营级UGC的代码内容制作 2，语言/演示级隔离（可以将开发生态从偏重语言为中心的语言库/包级下放到以应用为中心的插件/扩展/api级）3，编辑器带来的另外一点优势：那就是可视化调试。 对于这三者都是重要的，这一切都是不断整合的的结果(engitor整合了编辑器,enginx整合了服务器子组件IO)，然而是合理的整合：综合上正如前文所讲，”基于engitor的paas化”使开发下放，形成一栈结构，可以使所有的APP工作者维护一套套CODEBASE和工作在一套套DEMOBASE上。

前文主要讲的是1，本文重点说说2，3：

engitor之于规避脚本语言越来越大被当成系统语言的怪圈：分开语言生态与应用生态

脚本语言中，讲使用范围最广的，除了java就是py了。。JS也在跨三界兴起(native,web,mobile)，然而它们都走了一条病态的生态之路。那就是应用引擎和语言引擎不分。拿js来说,,对于开发者来说，每一个 package 就是一个 “micro service”，是最小重用单元。大部分的 package 只有几百行代码，甚至有些只有几行代码。这样的重用粒度是在其他社区难以想象的。 ————————- 然而JS将这一切做到了包管理内和社区repos里作为语言库，但其实，类似py，JS npm这种什么问题域的东西都做成库的做法其实也不好。拿js cms开发来说，不需要存在demolvl的wordpress的等价物，因为它一切3rd plugins都作为mircoserive被贡献到了npm，作为语言生态而非demo生态存在。与语言包结合更为紧密些。。

而我们以前说过，问题域扩展会越来越多，一个应用栈系统支持，语言支持层应保持短小，demolvl engine应越来越大。而不是反过来，而脚本只应被宿主偶尔调用，不应形成自己宠大的生态。因为它不宜作为通用面向语言解决通用问题（因此需要配备这么大的包），而应欠入到具体问题，作为demolvl小包或热更层或engitor用户支持modable层。

脚本只宜被欠入被一个宿主拥有，围绕一个通用复杂度应用的混合语言选型中往往有二种语言，一系统语言一门脚本语言，不应出现二门宠大系统语言。脚本只应被宿主偶尔调用，不应形成自己宠大的生态。否则就是系统编程语言了，因为它不宜作为通用面向语言解决通用问题（因此需要配备这么大的包），而应欠入到具体问题，作为demolvl小包或热更层或engitor用户支持modable层。

在这方面，php比py,js的生态要好。php本身很小。没有一个包含XXX，XXX库的大全发行包，它的扩展也基本来自thin dll wrapped native dlls，lua看来也是标准意义上的脚本语言啊。。因为它不通用。也不须通用。也不宜通用。 作为一个初学者，一门好的工程语言，其实他的唯一门槛是学完了语言就可以开始编程（编码）—或许还要加一个调试支持（设计能力和抽象问题的能力只要不是太复杂大家都会有），语言的类库绝不是你学习一门语言必备的，你不必经过学类库(甚至包括标准库)完成编程学习库只是语言的addons。始终要记住：一门好的工程语言，它应假设初学者和非初学者在面对问题时全是只会语言语法和手头只有api可用的“api kits”。

engitor就提供了一个隔离层，它使任何语言的库分离在这个隔离层之下，向用户明确表示，上面的语言层很thin，而问题层的扩展可以无限fat，qtcling中，只有一种主语言那就是qtcpp，cling出来的部分是附属于具体应用的。qtcling可以任意免binding作系统语言和脚本语言切换的可能（因为它只有一种qtcpp语法），使一切逻辑不致于聚集到脚本层，仅热更和动态部分需要下放。

可视化editor能带来visual 调试

只要有调试，我就能编程，根本无须太依赖语法与问题，调试在编程中的作用大约除了编码就是调试，大约在这里要对应前面那句再加一句:一门好的工程语言，它应假设初学者和非初学者在面对问题时会迅速找到调试工具和调试支持，并假设语言语法和调试是二个最基本的充要门槛。

可是显式的调试支持往往没有工程上的支持，以前是CPP之于汇编，不可视，就是黑乎乎的汇编，现在是js之于WEB PAGE，完成可视化类photoshop，直接整合进语言所属生态层的。非IDE工具。在敏捷编程中，test cases就是为了作test(yet another form of “debugging” in programming engeering domain)作的测试设局 — 它用的是编程的方法，而且都没有一个debug case之说。

engitor使dev进入demolvl时代，engitor整合任意应用为编辑器，由demolvl驱动，就使该应用debugging过程进入了一个真实的demo环境，在其中作debug完全是demo驱动的，就像gui之于visual debug一样，也像chrome的F12调试语义CSS一样，一句话，engitor编程，自带debug cases且无须编程。这才是engitor编程的最大意义。

1dddemobase and 1dddebugbase > 1ddcodebase：一个更强大的微实践选型

我们在前文中说到,paasone为开发建立了一个视具体应用为开发发布生态的“1dddemobase”，但我们真的不应该为一门生来用作DSL的脚本建立一个类似js npm repos宠大的中间层“1ddcodebase”.增加了语言生态无谓的复杂度，这个1ddcodebase只宜是demolvl的codebase.

在前面的选型实践中，我总想维护一个“1ddcodebase”，就像QT那样，包含对语言改造支持，问题库，IDE，本地系统编程，脚本扩展整个生态的支持。（这个生态是我认为拿来支持编程教学和自学较为完备的。为什么必须要加一个native langsys？虽然web,mobile开发已完全不native相关，但因为我们需要涉及到平台相关部分。学习上这二代也有着紧密的承前启后关系不可割裂。），尤其是QTquick采用JS+利用web方案解决通用问题DEBUG无门槛的方式是极好的选型和教学范本（web编程和JS是调试设局最好的实践环境和语言学习环境，微服务和微实践—— 这一切都对应enginx和engitor做的工作）。然而其中终究依赖了二门语言qt=qpp+js和生态，这就造成了割裂：离开了QT封装的那些：qtquick那些优势就不存在。

基于qtcling的langone可以用来解决这个问题。它就是更强大的qtquick，受enginx和engitor支持，langone结构和一栈web化结合将促成新的微实践大局：

1，它将更加显式化微服务和微实践。2，它更强化1dddemobase而不是codebase，和langsys/app之间的开发生态分区：engitor paasone和langone支持下，语言选型谈化了，app ecosystem demo选型也将同样重要。3，qtcling是一门可以将丰富的现有脚本语言binding进来作统一qtcpp编程的语言，可以复用已有成果。作混合编程。

综上：qtcling+enginx+engitor的组合将会是更好的实践教学选型for beginners。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

cloudwall：一种真正的mixed nativeapp与webapp的统一appstack

本文关键字：在数据库中安装程序。以数据库直接为后端托管程序，文档数据库管理器直接为云文件存储程序。无**backend webapp**，在**web**中开发**webapp**

大约在很久以前，我开始放弃追求统一化分布式应用程序和本地程序为同一个appstack的努力，这二者之间似乎天然存在鸿沟，像是应用的使用方式决定的，这种人为的界限并不是用来跨越的，拿web来说，它作为一种分布式架构和分布式appstack架构，不能做到像本地GUI程序或硬件加速程序一样灵活，比如web强调将一切放在broswer端渲染导致需要采用html5,webgl,js+css html这样的东西来增强它，这样它才能稍微像本地程序，一个例子就是用WEB实现的WEBGAME - 这种效率跟本地硬件加速实现下的game完全不是一种路子，WEBGAME的体验跟传统PC游戏的体验也相似并不相通，因为始终无法在远程上实现硬件加速还能stream到本地。web在服务端采用http而不是原生tcpip，导致需要websocket才能做到像主动推送这种原生TCPIP轻松办到的事。当然还有很多不同。

这不是WEB的错，WEB最初就是那样被定义的：它本来就是一种高级的native tcpip程序构成的生态。它的界面是PAGEUI，而PAGEUI是一种应用层的渲染，在服务器端，WEB程序大都由LAMP，LNMP这样的东西作backend，这类程序本身，其实是普通的TCPIP程序，并不是某个WEBOS的基础组件，就像原生程序之于传统OS实现中的任务机制界面机制一样，这也就是说，所有的WEBAPP都是有backend的，就是那个lamp中的amp等东西。它们用服务器的方式组建了一个分布式appstack，定义了一种appmodel，因此历史上，像WEBAPP + WEBOS这类东西并没有纯的，- WEBAPP是原生界面中采用有限技术打出来的一个点再在这个点构建出的一整个stack生态，因此，WEBOS也是OS上的高级OS而已 -- 本身并没有WEBOS存在。

在《在tinycolinux上安装chrome》上我们曾谈到相似的概念:chromeos脱离不了它其实就是原生界面（X11，GDI）加一个浏览器的技术本质，其实并不能与真正严肃的OS工程类比 。一个像群晖那样的APP管理界面就能称为webos。还有像owncloud,standstorm这种：sandstorm比oc多了xaas的部分。

web作为云计算负责定义APPSTACK的成份意义比较大，云计算下的程序无非就是WEB程序，因此云这种东西，除了虚拟化那一层，在APP生态上，它其实依然没有属于自己的东西。依然是高级原生分布式程序的BS化。

那么，这一切会不会有突破呢？有朝一天，WEB也有自己完全不依赖传统BS架构的东西呢？变得像一种真正独立的，由新的东西构成的应用生态呢？而cloudwall也许是另外一种“webapp”:cloudwall的确提出了很多新的耳目一新的东西，它虽然还是面向WEBAPP，不过它其中的一些部分可以作为与传统WEB迥然不同的部分来产生新的审视，比如它的nobackend设计，它的宣传语也一针见血：cloudwall，an Operating system for noBackend webapps.如它所言，它甚至提出了一种新的webapp和webappstack,webos雏形---改变了传统webapp中的大部分。

cloudwall中的couchdb:the only backend as webos部分

首先，它使用了apache couchdb，这是一种直接与WEB接轨的文档化数据库，如果我们把我们接下来要谈的APPSTACK称为某WEBOS的appstack的话，那么couchdb定义了这种appstack的唯一的backend部分，这免去了需要lnmp作backend的需要：这是它独有的特点支撑了它与lnmp这些东西的某组件明显存有不同的所在：这种DB是文档型的，且它nobackend。

couchdb支持直接hosting app并运行，称为couchdb-hosted webapp，它加一个类似数据库管理器的东西天然就是一个类OC的云存储程序，支持各种cluchdb插件的开发，这就是webapp整个cloudwall就是这样一个couchdb管理器。

> CloudWall is a kind of offline-ready toy in-browser OS, for authoring, storing, and sharing docs and CouchDB-hosted webapps. CloudWall installs via replication and needs only CouchDB and modern HTML5-compliant browser to run. Also CloudWall can run on static hosting, as a set of files.
>
> All CloudWall components run in a browser tab, no server or even internet connection required after system started. Any local DB can eventually sync with external CouchDB instances over http(s). One CouchDB can have several users connected, thus providing shared workspace, docs and applications set.

在我的《appstack series》《app series》系列文章中，我一直在寻求云存储程序的选型，我们换过mongdb,postgres,这种程序选型其实说大了就是WEBOS，我们在这些文章中都提出过这样的企图和设想。

来看一个这类OS的设计：是否一个app必备一个stack?将它的栈放大到受WEBOS直接支持，那么这种云程序背后的OS技术就会明朗化：

实际上，当考虑到一个app要配一个appstack东西的时候，它依赖原生程序appstack定义了自己新的appstack的局限就永远都避免不了，因为这里的mongdb,postgres永远被当成了appstack的dbbackend部分，，，而webapp应该是没有明显无backend的：像nativeapp stack一样，它们应该被集成在某一webos内部被提前解决掉。

而couchdb就是整个用数据库管理系统来作OS直接管理和存储WEBAPP的东西（当然它也能天然像其它文档数据库一样直接管理静态文件作云存储），如果将couchdb像cloudwall一样作为整个webos，那么传统的webapp开发就被定义在这个webos中，cloudwall的四个appstack组件，它们被集成在称为cloudwall os的webos理念当中。

> GDI:Renders HTML and receives user interactions
>
> App runtime:Manages bindings between data and UI controls
>
> CloudWall:Prepares, runs and closes apps, manages app switch

> Storage:Stores apps and documents, optionally syncs with external CouchDB instances

而这种开发，已经使webapp开发变得像本地一样了（无须处理appstack的部分只须关注app内的事情），我们一直希望得到的效果：webapp像本地一样以文件存储为后端符合像本地应用的习惯，这个目的也达到了。

cloudwall中的inapp editor：语言和开发部分

在《bcxszy series》在所有的努力中，我想得到这样一种程序和开发方式：不改变原生程序与webapp的大面，使WEB程序变得像本地程序一样简单，这样可以共用本地程序/webapp开发的概念，在模糊appstack方面，这就是cloudwall中的couchdb中谈到的，已经被解决。这里要谈到的是与语言开发有关的部分：

可以说，在《bcxszy series》在所有的努力中，我还想促成这样一种程序和开发方式：源码即文件，随处打包再走，直接per app an ide开发,这无论对实用和开发，编程自学都是极为便利的。

所幸WEBAPP src 文本化，支持轻量带走inplace editor是所有WEB程序它的天然优势，而且虽然一开始WEB程序与本地程序有很多不同，但像WEB标准化，HTML标准化，JS语言标准化这样的东西，它们其实在走一种联合化的努力方向，使WEB生态接近本地生态，比如，JS的努力方向也有一种是nativejs：reactos

其实couchdb与web结合紧的另一方面就是js，js是一种能够真正带来naitveapp与webapp合一的增强剂，这使得cloudwall支持极度便利化的inappeditor，这样cloudwall支持下插件的开发就是cloudwall webos下的webapp开发了，它支持用couchdb直接存储和保存编辑app开发过程中的文件。

这样有了以上这二点，personalcloud应是web os的论据就充份了。在开发体验上跟本地开发一样，甚至更简单。等cloudwall以上二大概念不再局限bs开发时，那么它就可以是新一代的WEBOS。

这篇文章可以用来丰富《编程实践选型》web的极大化未完的部分，整个文章的思路可以用来作为《bcxszy》part 2实践部分。如果使用cloudwall的理念，以后《appstack》，《apps》可以整合为《apps》

关注我

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一个设想：什么是真正的云，及利用树莓派和**cloudwall**打造你的真正云中心

本文关键字：**datasyner,syncbox,datahub,mypcmate,myphonemate,p2pcloud**

谈到云，可以从好多方面去表征它，比如它是计算机资源的虚拟化，它是软件的服务化，它是APP和APP DEVSTACK的一种仿本地化和去远程化（参见我们一直追求的uniform native/web appstack），可是我们结结实实忽略了最重要的一点，它还应是新的应用方式和用户习惯养成方式，仅仅是WEB所定义的那样吗？不，那只是云综合应用形式中极为简单简陋的一支，现在的web已级慢慢实现了无缝让用户感到它像本地一样打开，发布，被应用且慢慢涵盖了一些native appstack的那些领域，比如W3C的一系列标准和webgame将渲染数据化为stream创新性地解决云渲染形成电影游戏这样的尝试 --- 所有这些弥足珍贵，可是，明显地，我们的云，除了这些还应有别的东西：

比如，真正的云，那种同步要是默认的，不能是手动的，这应该是操作系统里面的机制而不是应用层一个“暂停/开始 同步”的操作，这种机制应让用户无从感知，不必为这个操作干开发发布上任何多余的事情。所以要为云准备一个操作系统级的东西，重新定义这个云把它与基本传统BS/CS云中的常见OS分开，就如同OS要促成每个程序都要运行在它的内存空间而不会因为串位而把整个系统弄崩一样属于普通OS的默认顶层设计策略，，那么等同地讲，同步就应该是这种云OS的默认黑盒级策略。

还比如，我们的云，要支持多设备P2P，在设备节点间就能达成同步（是不是有点像数据库的主从replicate,难怪db就是天然的云OS呢。。）。将同步视为搭载了云OS的多设备间能达到开箱即用的机制和默认策略就打开的东西。这有什么好处呢，因为这样做才能像“云”，本地的设备和应用可以充当云的角色不再局限于星型云结构，而是网状云。前者明显有一中心多终端的星型特点，任何新增一个云内设备，都需要我们遵循从终端到中心的同步路径，设备间不具有同等的相互同步支持。我们需要的云，不必局限于向哪个方向同步，只要存在云，管它是一个浏览器还是一个云中心，都可以作为云，它们都可以协同同步。这就必定涉及到为任意设备建立对等同步的能力，且使数据在各个节点间以p2p方式传播。

当然，云还有可能是更多更多的东西。。我们想不到的东西。

那么，为什么必须一定要这样呢？

为什么我们需要一个云网络而不是一个云中心

在《免租用云主机将mineportal2做成nas，是个人件也可服务于网站系统是聚合工具也是独立pod的宿舍家用神器》中我谈到利用colinux与windows能共处一平台的特性，将PC做成宿舍版的nas server而不影响它同时作为一台普通PC使用的过程，它的优点是：免VPS，使用PC本身的资源特别是大容量硬盘，可做NAS也可做群晖那种能装APP的WEBOS，本身也可保有PC作客户端与NAS交互的特性，不过客服共机会有一定隐患 。它的缺点是：离开了宿舍，你就获取不到VPS有网即可访问的特性，除非你一天24小时开机，而且PC太大，即使是笔记本你也不能整天在家里和公司间来回以此为中心同步你的数据，你还得求助于VPS为中心的NAS机制。

在新的真正的云需求明朗化之前，其实，这些优缺点，都不易觉察，可现在我们知道了：他们都不是真正的云，以上VPS和PC为中心的NAS AS云OS的方案中，都有同步，然而它们的同步不会自动化，没有大局的OS支持使之成为“真正的云设备”负责同步策略，只是用户的操作而已，其二，它不光能通过同步器（终端）与它交互还能让新增的任何设备参与同步互为路径上的二点，，，所以，本地（作为一台PC实现或者其它什么东西）只能先做一台服务器，浏览器只能做终端，各种APP只负责上传，忠实地维护远程那个数据中心，一旦断网，这个云将无从为云。

那么，我们该怎么做呢？

利用**couchdb+**树莓打造对等云网络

这样的东西，基本上couchdb+树莓派可以满足：couchdb的同步协议使得实现了这个同步协议的软件，或者是一个浏览器，或者一整个设备节点，都可以以p2p方式参与云。传统以web为中心的云中，只存在云中心和终端，在couchdb的同步协议下，云中心形成一个网络，同步是默认的事情，终端也能迅速化为云上的普通对等节点脱离BS从属。

较PC或VPS作为这种云，树莓派作为云，可以建立起最经济合算的规划，它本身比较便宜，整体性能，和8G的存储也足于承载一个人日常的资料，且方便作为像手机充电宝这样的手机伴侣型产品在家里和公司间转移，它与PC上的浏览器可同步的特点，使得只要不在重装PC系统，破坏浏览器缓存的情况下，PC和树莓派其中任何一者损坏，都不会破坏中心数据，且具备像本地应用一样，随时offline操作，接入至少二个云之后迅速相互同步不丢失数据的能力，基于couchapp的可扩展机制，使得这种云可以功能更强大。

下篇就做一个这样的云吧。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在**tinycolinux**上安装和使用**cloudwall**

本文关键字：在**tinycolinux**上安装和使用**cloudwall,**同步器**as webos**，**uniform native web appstack**

在《cloudwall:一种统一nativeapp和webapp的appstack》中我们讲到，cloudwall是一种构建在counchdb+counchdbapp之上的管理层APP可直接作为personal cloud hosting 文档和支持cloudwall plugin开发，然而它产生的奇妙效用在于它能作为webos，提供webappstack的效用，类似我们一直追求的engitor：介乎os和app之间的层面，封装domainapp开发栈和开发工具的层面。然而它更强大：它提供本地远程一致的webapp开发和发布方式（以无差streamed to bs和anyinstance + inapp editor的方式）。

其实这一切都基本都是counchdb的效果，它集成HTTP，本身是个DB带存储，与浏览器和JS结合紧密，支持hosting couchdbapp，文档即APP，这个APP仅由HTMLCSSJS构成，这种机制为远程web提供了至少三个栈，满足在其中搭建APP的基本条件：1,它的http部分免去了协议开发,web页面又是易于streamable的，2,它的DB属性免去了存储逻辑开发需要。 它stream到本地和每个counchdb instance（replicate）的结果是一样的，保证了浏览器与服务器之间的数据可以做到本地和远程不断联（in-browser os ），本地和远程，最难跨越的就是这个无缝stream。3，它采用的counchdb使用全栈语言JS，托管在其中的每个cloudwall app本身既是服务端的程序也是客户端程序(nobackend webapp)。然而就像tiddywiki一样：实际上在服务端JS只是静态文档stream到客户端执行，服务端只视一切为文档只是同步器。而tiddywiki这样的东西少了数据库托管。

可以说，正是JS和couchdb的完美结合促成了cloudwall，一个lang一个hostingtime,runtime在B端，这种意义下的“WEBAPP”不分本地还是远程，都是通过数据库stream的一个端，这就是文章标题说的：uniform native web appstack.

下面，我们讲解在tinycolinux上搭建cloudwall，和讲解在使用它的过程中，那些可以作为personalcloud使用的方方面面。

之前几篇文章，我们提出了跨本地/远程的DISKBIOS XAAS系统，并完善了一个/system rootfs，如前面所说，这是一个system与user libary分开的rootfs系统，/下只有三个文件夹/boot,/system,/usr,在/usr下有/local,/tce,/opt,/home,/vz对应于tinycolinux的那些mountable文件夹。那么从本篇开始，我们将管这个新的tinycolinux为dbcolinux，且以后的发布类文章都搬到其上来实践,如下cloudwall即是一例。 在《cloudwall:一种统一nativeapp和webapp的appstack》中我们讲到，cloudwall是一种构建在counchdb+counchdbapp之上的管理层APP可直接作为personal cloud hosting 文档和支持cloudwall plugin开发，然而它产生的奇妙效用在于它能作为webos，提供webappstack的效用，类似我们一直追求的engitor：介乎os和app之间的层面，封装domainapp开发栈和开发工具的层面。然而它更强大：它提供本地远程一致的webapp开发和发布方式（以无差streamed to bs和anyinstance + inapp editor的方式）— 这一切正是我们自bcxszy以来就追求的。 cloudwall何以如此强大？其实这一切都不难理解，因为排除cloduwall,这基本都是counchdb的效果，它明显集成了HTTP，本身是个DB带存储，与浏览器和JS结合紧密，支持hosting couchdbapp，文档即APP，这个APP仅由HTMLCSSJS构成，这种机制为远程web提供了至少三个栈，满足在其中搭建APP的基本条件：1,它的http部分免去了协议开发,web页面又是易于streamable的，2,它的DB属性免去了存储逻辑开发需要。 它stream到本地和每个counchdb instance（replicate）的结果是一样的，保证了浏览器与服务器之间的数据可以做到本地和远程不断联（in-browser os ），本地和远程，最难跨越的就是这个无缝stream（既然WEBOS可以类比为一个云存储based带nativedev likehood appstacks的东西，其必定要有DB一层，所以为何不以DB的replicate直接为网盘同步呢和app sync呢？其实当初W3C的WEB标准也准备是为WEB在BS端准备无差的web storage,web sql,webgl,etc..以提供类nativedev的appstack效果，不过好多实践被逐渐抛弃了）。3，它采用的counchdb使用全栈语言JS，托管在其中的每个cloudwall app本身既是服务端的程序也是客户端程序(nobackend webapp)。然而就像tiddywiki一样：实际上在服务端JS只是静态文档stream到客户端执行，服务端只视一切为文档只是同步器(服务器不保存程序逻辑仅数据又像极了微端。在微端眼中，与B端浏览器搭配最好的服务端的标准设施应该就是DB了而不是logicserver。)。而tiddywiki这样的东西少了数据库托管。 可以说，正是JS和couchdb的完美结合促成了cloudwall，一个lang一个hostingtime,runtime在B端，这种意义下的“WEBAPP”不分本地还是远程，都是通过数据库stream的一个端（而其实couchdb也支持传统的serverside applogic vs synced applogic），这就是文章标题说的：uniform native web appstack. 下面，我们讲解在dbcolinux上搭建cloudwall，我使用的是gcc443 32bit，下的是otp_src_20.3.tar.gz(erlang),js185-1.0.0.tar.gz,apache-couchdb-2.1.1.tar.gz 除此之外，还需要准备3.x的zip-unzip.tcz,icu.tcz,icu-dev.tcz，好了，开始吧

编译**erlang,mozjs**

由于dbcolinux的rootfs还处在初级阶段，有一些程序编译和运行还需要原来的/下的目录布局，如make meunconfig指令时引用到的/usr/lib一定要存在否则即使安装了ncurses.tcz和ncurses-dev.tcz,还会一直提示undefined reference，所以在这，为了顺利完成以下的编译，我们暂且恢复它，这些空目录只是编译时的权宜，dbcolinux运行不需要，所以编译完后可删除。 要恢复的目录与文件有：

```
/tmp
/bin指向到/system/bin
/lib指向到/system/lib
etc/resolver弄回来,nameserver 8.8.8.8
/usr/bin指向/system/bin
/usr/include指向/system/include
/usr/lib指向/system/lib
```

解压otp_src_20.3.tar.gz，它不支持shadowbuild和configure，直接cd src,sudo ./configure –prefix=/usr/local/cloudwall/ –with-ssl=/system,如果不加ssl，稍后会出现Uncaught error in rebar_core，然后make,make install

现在来编译mozjs,会使用到python,python要编译进ssl才能安装pip，然后被用于接下来的mozjs,改下Python build目录下的Modules/Setup中的SSL段内容为：

```
SSL=/system
_ssl _ssl.c \
-DUSE_SSL -I$(SSL)/include -I$(SSL)/include/openssl \
-L$(SSL)/lib -lssl -lcrypto
```

再安装pip，这里就需要用到/etc/resolver这个原先的文件来解析下载地址。安装zip-unzip.tcz,直接cd js-1.85/js/src,./configure –prefix=/usr/local/cloudwall,make,make install,一路成功，注：这里千W不要下到mozjs-45.0.2.tar.bz2这样的源码包，couchdb2只要求185的spidermonkey js，编译mozjs 45要麻烦得多，它要求gcc47,glibc2.12，且接下来与couchdb连接不了，比如：会出现，编译/src/couch_js/*c下文件，c包含C++头文件发生error: unknown type name xxx 的情况，涉及到修改src/couch/rebar.config.script，但是最终不能成功。

接下来编译couchdb,cd src,./configure –disable-docs,不能执行rebar,会发现它引用了/usr/bin，按开头说的先把这目录恢复回来，又发现解压出来的apachecounch权限是乱的，全部弄为root,make时rebar会用到erlang,设export PATH=$PATH:/usr/local/cloudwall/bin,再make release，提示不能发现jsapi.h，修改src/couch/rebar.config.script：

```
{“linux”, CouchJSPath, CouchJSSrc, [{env, [{“CFLAGS”, JS_CFLAGS ++ ” -DXP_UNIX -I/usr/local/cloudwall/include/js”},
{“LDFLAGS”, JS_LDFLAGS ++ ” -L/usr/local/cloudwall/lib -lm”}]}]},
```

成功，提示你复制生成的rel到目标文件夹。

安装cloudwall

把生成的rel复制到cloudwall:cp -R rel/* /usr/local/cloudwall，并安装icu.tcz，现在，将js libs 也复制到/usr/local/lib下，然后

改下/etc/default.ini中二个127.0.0.1为0.0.0.0,

cd /usr/local/cloudwall/rel/couchdb,./bin/couchdb，成功。访问,xxx:5984/_utils/#verifyinstall，进fauxton，在左下user处增加默认的管理用户，用户名一定要admin，然后添加一个数据库mineportal，然后在这个数据库的design处创建一个文档出现文档编辑区，下载

https://cloudwall.me/cloudwall-2.2.json，用noteplus打开，复制，粘贴到文档编辑区，保存，提示成功后，访问如下页面：

xxx:5984/mineportal/_design/cw22/index.html

进去，输入admin和密码，inliner是创建文章的地方，code是创建codesippter的地方，inliner file,gallery等都像是ocwp mineportal，一个网盘设施所提供的功能那样齐全。它的强大之处就是可以inplace editor生成app.

为了方便启动，你也可以在网上找到etc/init.d之类的开机启动逻辑

---

cloudwall预定于mineporta3,cloudwall能轻易与elm-lang组合，这二者都有强烈的与JS直接绑定的特点。走的是亲JS的同路子，且一个使用erlang一个使用haskell的非主流路线的风格相像。结合使用应该会有奇效。比如，打造一个能在线调试的inapp visual editor for cloudwall，下文就暂定为《另一种ipy：在dbcolinux上安装elmlang》吧

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一个隔开**hal**与**langsys**，用户态专用**OS**的设想

本文关键字：**efi based os**，**native hosting oriented OS**与 **APP hosting oriented OS,**将**OS**编程与硬件编程独立，将用户**OS**变为真正的**APP**空间。新**dbcolinux**和**goblinux**设计。

在《一个设想基于colinux,the user mode osxaas for both realhw and langsys》中，我们开始提到了一种用特定host/guest OS组合同时作装机和作app hosting for langsys的思路，主OS用来装机，guest os作用户空间的通用程序托管runtime/移殖层子系统/app容器/特定语言系统面向的开发运行langsys backend baas，—— 这种思路实际上是最初级的用双系统方案来解决分离传统OS和APP容器OS的思路。因为传统的docker式独霸业界的方式我始终无法完全接受，我想找到一个能用于本地实地远程装机运维，能用于开发和集成devops的虚拟化层面，这种层面要能用于legacy native方式，要以科学的集成方式在它应处在的层次，而不是现在docker和裸金属这样，高阶，仅用于云。

鉴于此，我一直在找这方面的案例和整合方法，一些努力在我们的后来的文章中被频繁提到，如在《兼容多OS or 融合多OS？打造实用的基于osxsystembase的融合OS管理器》《一种追求高度融合，包容软硬方案的云主机集群，云OS和云APP的架构全设计》《去windows去PC，打造for程序员的碎片化programming pad硬件选型》中我们提到了其用于不同硬件平台融合和移殖子系统的用途部分，在《群晖+DOCKER，一个更好的DEVOPS+WEBOS云平台及综合云OS选型》《hyperkit:一个full codeable,full dev support的devops及cloud appmodel》中我们谈到了其用于appcontainer,devops和云架构，appmodel的部分。在《打造一个Applevel虚拟化，内置plan9的rootfs:goblin（1）》中，我们将其用到了具体APP（将plan9 rootfs集进app，treats os logic and app logic together，把file作为APP的逻辑）层。—— 以上其实都是结合使用二种one host/one guest或多种one host/multiple guests的架构，使之职责分离成硬件OS（传统OS）和APPOS的职责的具体过程，算是开头提到的那个初级思路的延伸。

最后，我在《DISKBIOS：统一实机云主机装机的虚拟机管理器方案设想》,《DISKBIOS：一个统一的混合OS容器和应用容器实现的方案设想（2）》希望将它整合成一个PE层。于是用了diskbios这个用词：它是从bios开始做起的。是通用高级BIOS。最后，我们在《一种追求高度融合，包容软硬方案的云主机集群，云OS和云APP的架构全设计》《一个matepc,mateos,mateapp的goblinux融合体系设计》，《ubuntu touch: deepin pc os和deepin mobile os的天然融合》中提到了它用于第二PC的例子，—— 这都是选型研案，在整个《XAAS:the final all in one os》实践部分，我们在一步一步实现这样的一个系统：dbcolinux+goblinux。

新**DBcolinux**方向：**dbcolinux based on efi**

而如今，这种整合思路有了新的方向。

因为在《一个统一的bootloader efi设想：免PE，同时引导多个系统》中我们谈到这项工作可以做成EFI。那篇文章中我们还谈到EFI实际上可以发展为pe的替代，比如它可以发展内存管理相当于一个OS代替PE OS，更复杂化还可以集成hypervisor，这样做的好处是可以在一台PC上并行boot主OS和host os。同时作主PC和第二PC也可以自由分离。且可以用于云。同时不再需要另外的recovery。

其实在这个架构中，已经没有了HOST OS，所有的OS都是平等的。而且它还可以有另外一个天然的附带作用。—— 如果hypervisor足够简小。它可以以足够轻量化的方式本身成为一个可用于OS也可用于APP的架构。这样结合我们上面谈到的《hyperkit:一个full codeable,full dev support的devops及cloud appmodel》《打造一个Applevel虚拟化，内置plan9的rootfs:goblin（1）》。我们完全可以集进诸如hyperkit所用的虚拟机xhyve。这样，移殖层，融合，容器，applangsys backend baas,都可以以更简单和自然的方式达到了。

我们当然也可以组建传统的host/guest,我们也可以发展一个主OS。在这个主OS里组建复杂的OS生态系统。如vpnkit,datakit那样。做复杂的OS或容器集群。

新作用**:goblin based on efi**

还有一个更重要的特色，由于所有的OS都是用户态OS了。它分离了传统OS的职责。hal，mm等层面可以完全从用户态OS中分离出去，保留在那个efi os中。用户态的OS都是单一职责的app hosting os，这样的OS就是直接面向APP的了，可以叫APP OS。这样就降低了APP开发者的学习难度。也降低了通用编程学习新手的入阶难度— 因为它们从此不用再学习任何系统知识，处理任何系统编程。在APP os层面可以专心与一门语言绑定，或专心融入一种问题，或抽象方法，如plan9 all is file的理念，变成纯粹面向domain的专用OS。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一个**matepc,mateos,mateapp**的**goblinux**融合体系设计

本文关键字：将桌面环境，**toolchain**设计为**subsystem,rootfs as Xaas,rootfs**层次的虚拟化,非**Virtual OS Infrastructure,**第二**PC**，模块化机箱，第二**PC**，存储，计算分开机箱，**nas**另置主机，**mirror os,mateos,**自建**icloud,**本地远程通用的云**os,**云**app**

在《bcxszy:applvl programming》整个选型中，虽然我们经常强调xaas,langsys,appstack,app四栈一体的开发，但xaas跟其它部分在本书中一直是分开的并割裂成二部分，我们一直将眼光聚焦在applvl开发，却视xaas平台为devops工具,装机运维和虚拟化容器层面的东西，这些只是高于开发并不影响开发的东西，它本身并没有任何开发元素的整合：比如一种语言，一种运行时，一种库，一种appmodel，一种问题,etc..，因此它本质上跟applvl的开发并没有得到整合，本书目录制定上，第一部分的xaas部分《OS/TOOLCHAIN LANGSYS/XAAS》主要解决接下来为《APP LANGSYS/MIDDLEWARE STACKS/DEVOPS TOOLS》定制xaas服务的过程，xaas只是一直为applvl开发作铺垫，在《bcxszy》的序言中我们的选型也一直是语言系统和开发开始的，没有平台选型相关的部分。

这是因为传统XAAS系统编程领域与applvl编程领域是分开的，它可以不是直接的app hosting影响开发 —— 事实上，大部分虚拟机语言和开发体系都是这样处理的:将app托管在另外的hosting，脱离和超越native平台的编程和软件开发才是主流。大部分专用os也都是这样处理的如lnmp这样的东西它强调的是一种串序的四栈结构，它将linux作为xaas,php作为langys,nm appstack作为appstack，把webapp作为appmodel。

而xaas可以是一种新型的平台，因为慢慢地，在整个第一部分，我们发现，我们也可以将一些applvl开发的部分及早地上升到os层次。如GO作为分布式语言，它更进了一步，它将lang vm整合进入了app。如plan9和usermode plan9，它将分布式协议实现在kernel或rootfs，进而影响这种OS xaas下的app开发。——— 也就是说：在开发相关的四栈整合处理中，它可以以整合后三者的乱序方式进行，甚至它能居于langsys和appstack之上或寄宿其中。整合《bcxszy》的第一，二部分为真正的四栈一体的一部分。并制作出一种新的专用OS - goblinx。

Xaas的选型：高于开发的部分，和融合开发的部分

来归纳一下整书我们对xaas的选型路径：

在《发布一统tinycolinux，带openvz，带pelinux,带分离目录定制》系列中，我们讲到了一种host与guest分离，支持实机虚拟化，并良好组织在/system下的linux设计 — the pe as mainly装机系统，那时我们集中于装机的想法，我们想得到一个类似esxi 的东西，相信兼容os一直是人们的一个梦想，而我们一直都没有到达过，这种兼容OS的思想在《兼容多OS or 融合多OS？打造基于osxpe的融合OS管理器》一文中再次被提到。

在《利用hashicorp packer把dbcolinux导出为虚拟机和docker格式》系列中，我们加入了devops思想，并去掉了为实机/裸金属/云主机作装机虚拟化hypior的首要需求，弃ovz选用这次的lxc主要是为了devops。lxc可以作为app容器，也可以作为系统容器，我们还是将其实现为一个pe。

这一代的dbcolinux是一个融合os(比如用osx base当parallesdesk的os，windows和osx作为其subos,这样osx base就是pd的专用os了)的不断增强，接下来的二个增强就强烈涉及到了开发：

在《打造一个Applevel虚拟化，内置plan9的rootfs:goblin（1）》中，我们提到了整合go,plan9到这个linux的思想：并提到了这样做的底层依据：整合busybox或plan9这样的live demo服务到每一个app作rootfs虚拟化，将虚拟集成到APP级，可以促成无架构APP，最重要的，将虚拟化做到app级，可以将devops也做到app级,比如，工具级的IDE就能生成app包做到包管级。利用一个工具Provisioner了，这二者可以达成最简单最实用的编程典范。plan9这种协议，是天生的本地远程同步/通讯协议，可以免协议开发(因为它是demo级的，甚至不是一个lib)在本地和远程都能运行的APP,且无修改地工作在一个OS下，就如同git p2p协议一样。由于它也是一种同步协议，甚至可以为每一个 app建立如pouchdb一样的内容同步机制。这样更接近天然云化了。

甚至hyperkit,Lektor:用开发本地tcpip程序的思路开发webapp,Shell式编程：会用就能编程，把问题域整合进系统域。这种思路，我们都提到了。

最终在《一种含云主机集群，云OS和云APP的架构全融合设计》中，我们提到分布式架构的本质问题是OS和APP的选型和实现问题，这就是说，在整创出新xaas,langsys,appstack,app四栈处理中，OS是一个可以将APP和APPDEV上升先行的路径，比如在OS级自带分布式支持的系统肯定出来的

本文即是上文《一种含云主机集群，云OS和云APP的架构全融合设计》关于goblinux的特化版。

Goblin基于上述融合设计的强化

在上文《一种含云主机集群，云OS和云APP的架构全融合设计》中，我们从这种OS适用的硬件，集群开发，所用的OS到所开发的APP都讲到了,那么现在，我们继承从tinycolinux->dbcolinux->goblinux系列的成果，一点一点，将前2者附加到整合了后者的版本上去：goblinux的实现实用版本。

在硬件上，goblinux，就不追求多PC集群了，为实用起见，它可以是二台PC，大体上，这二台PC是计算和存储分开的机箱设计，你可以想象它为一台类似群晖mini itx机箱nas上面又加了一块主板，市面上这种mini itx模块化机箱蛮多的，但双主板和不多见，在这种设计中，主存储的主板可以用nano itx或更小一点的pico都可以，主计算和日常使用的那块主板可以接近一台普通mini itx。如果可以，你也可以三主机，一台路由器主机，但实际上，路由器更适合与前二者分体，这二台pc，一台装gui生产/开发环境。一台装file server/mirror cloud环境。都是goblinux子系统。

在OS上，也可以是一个管理器下的二个融合OS，主pe的goblinux就是像群晖那样的rom boot = grub+kernel+srs driver libs+bootpe，它很小。这个主PE goblinux内含三个系统，一个guy desktop,一个file server， 一个connect server, 对应上面的三主机。前二个OS是标配，要做成融合结构，代表分布式的本地/远程端，一个GUI，一个mirror tolocal远程osstub,它里面没有app，不需要维护，是个黑盒,没有界面，只有数据。但是也不给维护接口。所有的操作在local mirrored gui操作系统，Mirror os可用于装在上述提到的本地另外一台nas pc上，也可以装在纯软件融合os管理器下的file server子系统上。file server也可放广域网追求异地文件备份效果只是交互性能大减。可脱网使用，重新联网时，将同步远程。注意这并不是VOI,VDI那套。

开发支持上，如上所述，这是一种天然的分布式APP和云化APP开发内置的软硬结构。它有以下几个特点：

如上所述，用go，且利用9p实现无须协议交互的本地远程p2p，写git之类的东西，因为plan9有os级的实现也有linux的userspace实现品，甚至有9p lib。demo态，可以in the kernel, or embedded in a subos,rootfs,or app，运行态与开发态的区别是，demo级rootfs的9p是运行级的不需被编程，而开发级的是预编程。9p usrspace 9p是demo level级的。为现阶段简单起见，plan9p只集成在usrspace级，且用demolvl的。

————

本文过后，我们会一步一步整合实现接近上述理想化的goblinux，无论如何，我们这是在将平台选型慢慢地实践弄成一个现实品的过程，和强化goblinux的最终过程。

Xaas内容过多，为清晰化起见，我们依然选择不整合第一二部分目录。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一种**matecloudos**的设想及一种单机复杂度的云**mateapp**及云开发设想

本文关键字：可编程的**os/os kernel/os rootfs,os as service,os as service,mateos**。**cloudsubos**，，客服同体，**api/runtime**共体，将**os api**化，**headless os core for cloud api**，融合云**app**

matecloudos,matecloudapp:真正的分布式

在前面我们谈到《enginx,engitor》系列，还谈到《Plan9:一个从0开始考虑分布式，分布appmodel的os设计》，这些文章共同点都是对已有分布式app（一类跨OS/OS进程的APP）的思考和未来创新设想，可以拿来从各个方向类比：前者是传统os+lamp,lnmp方案，以OS上安装的分布式软件作为OS服务子组件（多见于服务器OS环境）透出它们的服务，在这上面打造出一个web appstack(比如apache之于page gui,mysql之于持久db...)，在这里，可利用的服务，和API存在于这些组件抽象给语言的后端（非OS固有部分），而后者plan9的本质不同之处则在于提出了一个devable os，将内核中的对象和os本身作为可编程对象，透出这些API服务，直接在已有OS组件上作分布式不提出LAMP之类的多余结构，在这上面打造分布式APP。

后者显然更原生，也更好用，（如上，传统的os只是api runtime/c dlls，api服务也是离线的另起一层，且并不面向分布式，并不是与开发一起被设计。是先把os用起来的原则建起来的，os与api分离。以前，为了达成跨语言跨OS的API，是靠给API打stub完成的。这种方式粗鄙无用已被淘汰不单是rpc技术改良就可以完成的）。而新时代分布式程序和分布式开发的固有特点是：分布式开发要求开发接口与程序本身共体不分离，也就是脚本语言和web开发中,“srcfile即程序组件”那套，------ 当然，现在的web分布式是从lamp上搭建的，现在的OS kernel也都基本不是plan9这类。但我们可以在现存传统的os结果下进行改造。比如，我们可以另外铺设API。将OS作为一种headless api被调用环境。类似传统方式铺lamp服务那样的方式进行：只不过这次向上提升到针对os的组件进行。也即，我们利用plan9思路和方式，从传统OS本身开始做而不下放到lamp，OS服务要作为基础被分布式服务化，发展基于OS分布化之下的分布APP。而且不必涉及到统一使用者的OS跨OS。，如果成功。本地开发和分布式分开完全可以统一。------ 引申开来，即：os/os kernel as serivce，api和api runtime天然一体。组件即demo即api。本地组件即远程服务(自动API化，无须再次面向不同语言作显式API化)。

新**mateable**分布式用于最小实践教育路径和降低实践门槛考虑

这样做具体有什么意义呢。

matecloudos和matecloudapp，由于OS都是自带文件系统，GUI，和持久的一类综合体，这些服务不必再搭建，作为客户机和作为服务器环境的作用可以统一彼此互为mate，这些理念对于APP开发的统一作用和难度降低作用不言而喻：

首先，（1）第一条，如上所述，本地开发和分布式开发将共享同样的技术基础，变成单机复杂度，我们现在的分布式分开几乎都是统一后端和webapp这二种理念：由于webapp是现在唯一的真正跨端的模型。。我们现在大部分移动端和一些native端应用都是通过分离前后端为统一后端+不同的客户端APP技术完成的，将所有分布式开发视为前后端分离，后端统一API化，都是web形式的调用，比如某个网站后端，或一个天然的cloud function，前端则利用统一的ajax,reactive,pwa技术构建。即所谓的MVC模型。这些都在《一种设想：在网盘里coding,debuging，运行linux rootfs作全面devops及一种基于分离服务为api的融合appstack新分布式开发设想》说过，如果os本身被api和服务化了，那么完全可以用native app逻辑。不必另造一个webapp出来。来达成分布式APP，简化后端。

（2）甚至，我们完全可以在类plan9的理念下统一，使分布式APP和本地native APP开发理念同源，再度降低APPDEV的难度，我们的开发一般都是某种app层次的dev，需要呈现为一个APP形式，比如web在是webapp，移动端是mobileapp，桌面就是nativeapp，软件和编程就二种逻辑和抽象：一个业务逻辑，一个关于APP本身的appstack逻辑。appdev的本质是三种stack（一般是gui,持久，网络）的形式在所处不同形式下的演化。比如文档数据库。就是视图模式下的“文档”，区别于存在于磁盘中的文档。appstack占了一个app逻辑的大部分，可以极大得到简化。

一个例子是类似p2p性质的gitcore客户端程序即是服务端程序，可以以同步的方式代替协议交互。也可以同步的方式来处理数据交互。比如，直接把原生GUI透出为分布式API服务，客户端也可以是远程桌面remote app这样的东西，如果客户APP也是服务端一样的OS，则GUI无须渲染（只需像那些远程桌面协议一样传送位移量）提出一种新的类web的remoteappmodel。天然分布式。

（3）还有，如果是plan9这种kernel，由于整个os内核都是构建于基于对象上。所以可以以编程的方式来调用和共享，以OS抽象和语言统一的方式（这个作用不得了，见《一种shell programming编程设想》，统一问题域和抽象域，甚至语言方案域）同步二个OS间的对象。这样在OS的构建过程中，就穿插预埋了对未来这种OS上的APP编程的设计，甚至统一了APPDEV和这种OS上的系统APP开发。

（4）更多，在线IDE就随处可放。调试也不再基于堆栈跟踪这些反工程层面。。。。。。

我们的设想是基于deepin做一个这样deepincloudos，再在这上面发展一些好用的matecloudapp(集成云开发IDE和云LAMP，首要的就是在线IDE和mineportal apps:如file explorer内置同步，如果能在jupyter上写程序，那么这是一个天然debug backend app，已有这样的产品，不过这是工具层面的，我们就是要将其做到与OS级天然集成。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一种追求高度融合，包容软硬方案的云主机集群，云OS和云APP的架构全设计

本文关键字：兼容多主机硬件设计，兼容多**os**，兼容**native/cloud**程序模型**,**兼容本地程序/分布式程序。网络操作系统，不是**x11,**不是远程桌面，不是**web nas,**不是**pouch**存储同步。不是远程投屏。

云在人们的观念中就是远端，它承诺将计算发展成水电煤一样的可被直接利用的资源，与内容和我们本地的客户端或终端接入（所以有了云存储，云GPU等各种传统资源的云化，以及一些或细分或复用的云资源，如云验证，云游戏,etc..），虽然云技术的背后是一层一层的虚拟化，可是它并没有将这种工作进行到我们日常使用APP方面，作为用户，我们还是停在手机上装APP使用云资源的历史情景下 —— 试想一下，我们老早以前就有网络操作系统，TCP/IP这样的C/S程序。可是一直到现在，我们依然这样子使用云。——— 云程序没有过任何改变。它只是将服务端越做越肥兼带更强大的content streaming的改变(比如即将到来的5G)。

稍微涉及到复杂一点的自建云/云系统管理情形下，比如群晖，ECS，裸金属虚拟化，虽然云装机的方式更灵活多样，但我们发现我们最终还是在使用传统的OS来建立和使用云。还有远程桌面，vnc，远程桌面，局域网投屏,wifi display,ip kvm，x11,web界面，这些永远是我们解决操作远程OS的方式。

云APP进化到现在最高级的形式，也就是webapp，它尝试解决一部分问题，可是它似乎依然不够完善，比如，它如同一些本质是”伪remoteapp“一样，永远基于某种远程界面，一部分逻辑在远端。无权直接访问本地资源，且延时较高。

> PS：以上三个问题,表层来看，是因为云APP总有个OS，它与本地分属二个OS二个独立迥异的运行实体，这种现实割裂了我们原来使用本地操作系统操作APP来操作远程APP的习惯。—— 人们一般把兼容多OS称为云OS（如fydeos,openstack,linux,windows,etc..），把分布式APP称为云APP(web算，走socket的也算。。p2p去中心的算，暗网的也算。。)。这些OS从来都是相互的复合，早就被设计成单机OS+tcpip，从来没变过，只是被分布到了网络。所以它不可能有真正的分布式。——— 小到content streaming,vnc，大到开发，甚至一切，都只是权宜之道。。以至我们没有过真正的分布式OS和分布式APP，一直在web的小概念里打滚。见我的《打造一个Applevel虚拟化，内置plan9的rootfs:goblin（1）》，我认为这是因为我们始终把“真正的分布式”做到APP层次。短板理论下所以一切都是伪的。

分布式架构的本质问题是**OS**和**APP**的选型和实现问题

分布式架构本质是巨大的问题，这是我们bcxszy一直努力选型与实践的目标,因为有前面文章的铺垫，我们现在深入分析这个问题:

在前文《兼容多OS or 融合多OS?打造基于osxpe的融合OS管理器》中，我们看到，有时为了实用。我们宁愿牺牲少量的性能，选择融合这种折中的方案，而不是严肃意义上的那些破哪补哪直面问题解决问题的打法。 还比如:

1，为了促成能用好用的兼容多OS，从技术层面有二种流派和方案，一种是靠虚拟化，发明一套“OS的OS”，利用这样的元OS来管理guest os，一种是利用经典的再造OS的方法，像龙井，wsl，虽然它也是发展host os/guest os，guest os as subsystem，但与虚拟化流派的做法有着绝对的不同。前者依然是用传统的OS叠加OS，除了hypervisor，容器技术毫无创新，而后者，尝试从问题的源头重新去解决问题，从本生态内去解决问题而不是不断复合/融合（这里没有贬低该文OSXPE+PD方案的意思），如果问题出在最初考虑不周，抽象不够，那么就再抬高一个层次。结果是出来一个单一创新了的全新的OS ------ 这才是主体，其它子OS只是rootfs附属技术。

2，带着这些观点再谈分布式APP，我们发现这种历程正有点像业界对于分布式APP的挣扎和创新，在前面我们不断谈到分布式应用模型的讨论，其中最重要的地方《Plan9：一个从0开始考虑分布式，分布appmodel的os设计》中讨论得最为深刻，在那里，我们提到plan9正是这样的创新。它有别于用传统OS叠加的云化分布式 ——— 正如那些单调的兼容多OS技术一样，而是一种从协议级去创新，去重新发明不同理念OS的一种努力，plan9协议是一种从开发，从内核理念到语言，到APP的变革，去促成不同于现今可见的任一分布式APP的全新分布APP，而谈到协议，它必定是某种上种抽象。刚好在《用开发本地tcpip程序的思路开发webapp》，我们还讲到协议化。使用本地方式开发web程序。将web首先升阶抽象变为协议化的东西这样可以融进“静态web仅为资源”这样的webapp。那文中，我们还坚持一惯贬低了webapp vs 通用分布式app的那些优缺对比，因为它使用传统os，利用appstack层打洞，所以它是分布式OS中极为狭隘的一支，Web绝对是分布式appmodel中最浅层的一个典型。

无论如何，从这二大段说的正是：a,虽然能融合也很不错，不过话说回来。这并不表示能直接解决问题也并不是就不能实现效果最大化，b, ——— OS与APP，本来就是问题的一对最原始中心 ——— 只要先解决这类问题并创新，挖掘发现“真正的分布式APP”才能稍后解决开发，语言和工具。c,而抽象和协议升阶，是软工解决新问题出现的手段。

其中,b是中心的中心。

所以，我们为什么不能做一个真正的OS和分布式呢？

首先来看兼容**OS**解决多硬件，多主机软硬平台融合:一个多主机环境，跨本地远程，多**os**兼容的多主机方案设计

前文《兼容多OS or 融合多OS?打造基于osxpe的融合OS管理器》中，我们谈到的是单主机多OS的方案，稍微提到但没有深入多主机方案，我们提到，多OS的需求跟多显示器，多桌面机制一样可笑却实际存在。但其实细说起来一点也不奇怪，我可以在一台电脑上装三系统，一桌面系统用于生产，一类似dsm的系统用于同步文件，人手三件套手机电脑路由器，OS各各不一，未来公共使用的物联网（华为hm os?）。基于容器，

OS也不会少。我经常看到和能想到的是，有些人还有二部手机，开发者有二个主机甚至多个电脑，

> PS:多主机PC，这些主机可以要么是Computer as unit，便携显示器加nano itx小主机打造新式便携pc(用一个双系统机箱，或定制一个类双盘位的nas铝盒把这二个主板装上，nano itx 二个是24*12,下面刚好有空位上电池+整机箱上面覆盖键盘键盘就更好了，,打造多主机笔电式机箱）或者直接多nano itx PC集群,随身网吧（直接定制长vesa挂架挂墙）,都可以带着出远门。，最现实的方案：要不就2台pc，一台带typec的笔记本,一台无屏带typec供电的便携显示器。也可以组成至少二主机环境。要么一台有双vesa的显示器，主机在显示器后，二个位。

无论是使用实体现，还是这些实体主机配合使用一台云主机，或全云主机，或云主机加一台实体主机，兼容多OS管理器都能使它们协同工作，我样可以将这些云主机中的一部分分出来一些，比如，负责某些任务较重的APP可以独占一个CPU较强大主机。负责存储的可以独占一台配有大容量存储的主机，因为我们要谈到的兼容多OS主要是在一台机器上装多个OS，负责路由的分在另外一个台机，etc… 如何将这些平台上的OS，以及平台本身用兼容平台/兼容os方案整合起来，使之协调一体，前提是：这一切受兼容OS管理器的管理，它本身是个融合OS。

前面我讲到使用macmini+oray控控的方案，wifi局域网纯软或借助硬件的投屏技术，内网穿透+VNC桌面，或ip kvm显示技术，都可以被舍弃了。因为这是个：远程OS也能被统一的虚拟机以桌面虚拟机同样的模式被分布式管理的架构，如下：

一种mirror to local的远程投屏操作系统，可放在pd

我们在前面《聪明的Mac osx本地云：同一生态的云硬件，云装机，云应用，云开发的完美集》谈到mac的云是一种很聪明的云，它主要将浏览器直接作为云存储的同步客户端，而且通过server app+描述文件管理器，利用caching as sync在苹果不同终端间建立私有线上线下打通的云，而且客户APP和服务性APP同宿一个osx，有别于群晖这种分二端APP的做法。而且它的IDE自带devops，也是一体化客户/服务性APP模式 —— 利用本地浏览器作同步客户端，利用sync+caching同时建立云content streaming，提倡使用客服一体化app ——— 正是Mac显得聪明的几个地方， 一言以概之，mac将云理解为传统桌面的附属，所以在实现上尽量向它靠近。

为什么就不能有一种OS：它将远程的一切无缝mirror到本地，如果远程OS可以被管理，可以像mac server一样，它作为分布式远端的OS，不只是一个远程桌面，而是一个实实在在的与其它OS等同的OS，只是被mirror到这里，所以在本地，有跟远程一模一样的运行状态和所有资源，如果放到PD下，就跟其它虚拟机一样，也即，我们本地会mirror远程机一模一样的状态，如果可以这样做到，为什么不呢？当然，mac也做得并不完善，反正，群晖这种webos，需要急待改进了。

缺少一个真正的分布式OS外观,正是分布式问题得不到真正解决，麻烦开始的地方，必须要在这里攻克它。

是不是还想到了plan9？它将API和运行状态都用本地/远程统一的协议来保存。它可以将nativeapp的四栈全部协议化,包括上面提到的投屏和界面。最主要还是其存储协议化，这种协议不光是面向本地的，也面向分布式。所以plan9是一种协议化升阶OS，它解决的是API,但是这种抽象和协议要向下兼容，即，把远程OS也弄为跟本地一样。

真正的分布式云程序，其行为要类似本地，不仅要从开发层透出API，api as services.而且要mirror出当时的host环境，才能透出整个对应native app的文件系统，等信息。这样才能以类似编程本地app的方式去编程dist app，比如，上传一个文件，你有界面，也有upload api，但是没有远程机器的磁盘信息。更具体点，远程桌面要有上传下载文件。所以，新的applvl runtime，要整合一个xaas环境，而不仅仅是api这些app本身的东西，要么在每个applvl像go塞进去一个vm一样，可以为每个app像plan9一样集成一套9p rootfs，也可以像plan9一样用相同协议的OS来保证这类远程APP有同样的本地/远程互操作性。

其中，第三种方法就可以将这种OS作为管理远程OS的实例，塞入PD，实现本地管理，而不必受到远程桌面之类的限制。有相同的外观。下面谈APPLVL的文件系统外观。

云APP/本地APP融合:一种uniform navtive/cloud appmodel抽象的appmodel设计

我们现在一些分布式文件系统，或oss，或文档数据库最大的毛病，是因为它们在操作系统的粒度上，没有给用户呈现一个类本地文件的打包视图。这些必须做在OS层面。才能给后来的APP提供统一的外观。应该上升到os层作为os的理念。和applvl虚拟化程度。

就如同plan9协议化了分布式OS交流用的存储协议一样，mac osx用finder作云同步客户端一样，统一的文件系统外观，使得可以结合plan9以操作本地文件和界面的方式来操作这些存储，形成真正的storage backend distapp。比如，用于web,可以有，基于化远程可观文件系统的网盘，可以有基于网盘同步的通用webapp设计。

基本，搞定了PD管理器，uniform navtimve/cloud 存储和界面协议。一个app的栈就被完全定义下来了，关键是，这些是按本地原来的用户开发和应用习惯设计的。而不是像web一样打洞，像虚拟化一样造更多雷同的东西（虚拟化只是解决了通往分布式APP问题的平台层及一些层面。更多其它的工作需要完成），而是重新从0开始，重发明，创新包容。也即，为了制造那种nativecloud体验合一的分布式，唯有像plan9一样。从经典的内核，API这些源头做起。

这样。从装机，OS到语言，开发，app，这些最终才能做到最好融合，对于促成一种真正的一种native/cloud融化方案平台和app这一最终目的，才能达成。

Bcxszy用goblin这种被创造以整合plan9rootfs，工作在applvl虚拟级，可用于linux kernel based兼容多OS装机环境的分布式集群环境和类PD管理器，做到平台和OS级的融合。Bcxszy将最终打造一个网盘app用于说明云app/本地app融合的概念的实践。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

hyperkit:一个full codeable,full dev support的devops及cloud appmodel

本文关键字：**app level vm,hypervisor as component, hypervisor as api, langsys as api，cloud infrastructure as code，Devops is only tools but not appmodel, Lua as configuration language**

在bcxszy和bcxszy demos的历史选型上我们一直希望得到一个在native上面改良云/分布式程序/cloud的设想，接上学习，开发和应用的断层，实现学习上的最小投入和应用上的去重化 —— 并寄希望于它们是一块自成一体的(from xaas,langsys,appstack to discrete app)，在我们的集成思路和实践中，我们忠实地 1,发明了一个os:dbcolinux并用packer生成了它 — 它代表xaas中的云os，2, 提出了一种语言:terralang — 它代表devops langsys, 3, 参考过openresty那种用发明传统服务器程序来写web程序的方式:enginx — 它代表appmodel —— 我们最终想得到一个创新了的集成三大件组成的fullstack devops in a box-我们叫他engitor,这一切都有别于我以前选择过，现在强烈鄙弃的现行web集成方向的东西:它们使用web appstack,使用js，与nativedev断层，这是一种在栈上造栈的行为，造成了学习成本加大和资源浪费，而且与未来的5g脱节。

> 我们来理一下:
>
> 以上terralang是C runtime and compile as a lib，整个C as a lib可以被lua codeable，这就接上了devops，这个lua可以作为c的shell语言，ide语言，甚至给c内部增强，extending c使用，openresty也是lua codeable的nginx。terralang可以直接调用和发现c dll as api（通过头文件，免binding），也可以直接编程一个binary和配置一个binary本身，以类似shell的方式调用他们（而不需要任何封装工作）。——— 这一切都有了devops的可能，而且是lua的devops。
>
> 这跟packer封装构建的dbcolinux式的devops却有不同:dbcolinux虽然也是可以packer配置出来的，也具备devops的可能，然而，它使用的语言是yaml。我们在前面谈到hashicorp的packer是利用非编程的方法，一种配置语言HCL is the HashiCorp configuration language，配置的devops，就像docker file，使用的yaml。这二种语言在现实生活中却都需要。
>
> 因为一个运维者有时不是一个开发者，按照复杂度，一个语言必须至少分化出configuration language和script language，前者供运维人员或普通配置用户，部署人员使用。后者程序员使用scripting的情况下，更需要更为复杂一点的专业语言。而有些天才开发者，可能需要更为复杂的语言，但这就需要在保证语言核心尽量不变的情况下，给他们封装各种各样的sugar或增进语言机制（这会造成语言核心膨胀），但第一种人，明显是一条一条顺序语句的配置适合他们使用，甚至数据填充data drivern式的配置更适合它们。像html就是标识语言，lua nul就是标识语言。这就是为什么有了js还要有html的原因。

那么有没有作为库存在的xaas devops呢，而不是像packer那样的工具。使得1，它也可以 lua coding 的方式用于devops呢，2，而且同时提供配置式开发接入运维者。

hyperkit及其支撑产品刚好是用来解决上述二种问题的。

> hyperkit
>
> HyperKit is a toolkit for embedding hypervisor capabilities in your application. It includes a complete hypervisor, based on xhyve/bhyve, which is optimized for lightweight virtual machines and container deployment. It is designed to be interfaced with higher-level components such as the VPNKit and DataKit. HyperKit currently only supports macOS using the Hypervisor.framework. It is a core component of Docker Desktop For Mac.
>
> Hardware-facilitated virtual machines (VMs) and virtual processors (vCPUs) can be created and controlled by an entitled sandboxed user space process, the hypervisor client. The Hypervisor framework abstracts virtual machines as tasks and virtual processors as threads. —— 这跟colinux的思路有点相像。

下面来详解它的意义和用处：

1，有助于打造一个可编码的用于devops容器。

也正是linux在各个层次/各种模式的可存在性：kernel ring0 space,usr space,cooperative mode, as subsys, as sandbox,vm space,hypervisor space,甚至于浏览器中(有用js发明的linux)，hyperkit —— 它居然可以存在于app中。

这就可以把全套devops放在app级，内欠式地去做。甚至以lua编程的方式去做，因为hyperkit是基于c api的。，而hyperkit就相当于可编程的xaas

这使得我们集成三大件的路径变得有了更多的取巧，因为hyperkit小而且紧凑，与linux kernel,lua runtime ,nginx一样以小为美，设计正交，我们何不将其放进一块soc中，形成为一个box应用。hyperkit和其打造的生态各种kit即是这样做的。

2，打通运维配置式编程from the same scripting langsys和binary api service发现

如果说terralang能发现api免binding，那么hyperkit可以从成品中制造api不需要设置成开发接口免声明。

一般地，dll是开发者用的东西，其服务和api要通过某种语言，被编程才能透出来，binary可以被配置，其实一些组件机制可以用xml的方式直接配置，形成应用，像pme加持久的xml文件，但是它终究不够通用，且借用绑定专门的语言机制，因为我们用了hyperkit,相比之下使用hyperkit要通用合理得多，它可以借助一个叫datakit的产品。

datakit: Connect processes into powerful data pipelines with a simple git-like filesystem interface

而现在，进一步地，binary可以被进一步以更灵活的方式用于配置：借助hyperkit，可以对二进制进行输入输出的shell式再编程成为可能。

datakit只是提供了一种可能，使得多个kitbox可以通讯，在这个层次上重编程，至于所用语言，肯定就是terralang,这样就可用同一种语言完成shell编程和一般编程,形成一个运维与开发统一的开发思路设计:以c+lua为中心，所有的逻辑都可包装为shell调用或sugar语法

还是那个问题，如果要兼顾运维的话，还是需要提供配置式语言，这个terra同样可以办到了。lua有专门的类packer yaml的配置语言生成器。Universal configuration library parser for nginx.

有了配置语言，我们甚至可以裁剪式将terralang+各种DSL发布成各种语言的发布版，就像发布linux自定义了rootfs后的结果一样。

3，真正的appstack的事,Devops is only tools,appstack is the essential

其实devops只是工具，它并不改变任何东西。它并没有带来appmodel的创新，那么，我们最后来思考appstack的事。四大件中，xaas，langsys，appstack影响appdev，那么，真正的云程序是什么程序呢？

我一直在寻找真正的native cloud能完美融合的统一appstack，从来就没有出现过真正的云程序，Web拖慢了真正的分布式程序达几十年之久，真正的云软件是saas，它是传统软件的服务化。web并没有按这个标准发展。web的api规范其实也有，wsdl等,但不是废弃就被证明不好用。我曾经以为虚拟化，devops，这些规范下的app就是云程序。但其实，分发层的所有东西都不足于区分web与saas app的区别。横跨本地app/webapp之间的鸿沟不可smooth，不能在web之上做saas app，这方面的例子，实在太多了：

比如，内容同步，见cloud wall。不但pause/start式的工具，其同步是表层，pouchdb的自动式也是。 Page ui也只是表层。整个web appstack也不能。是web设计中的链接可点击到达么，也不是。Devops也不是，那只是工具

如果没有接上开发，那么一切都不算数。先有云开发才有云APP,必须首先为云app建立起开发的最底层的那些api的定义，甚至硬件层的定义，然而将这一切接上一门语言，用于开发出可部署的东西：这就是最终软件的定义。

以前的RMI，broker，DCOM，这些，才是远程，分布式程序的正确思考方向，然而他们都被废弃了，WEB+脚本，代替了一切：因为移殖了VM，就可以在本地和远程保留一样的api service，无谓区分是代理或打桩过来的还是本地的。—— 可是这一切成全的WEB有太多缺陷了，最大的隐患就是它接不上未来的高速5G，5G可以streaming一切，web本来就是适配低速网络提出的js，现在它要把它还给高速时代的那些开发语言和方式了。

比如，在开发层，甚至OS的内核调用，本身就考虑了远程调用。甚至机器硬件级内置了远程调用，才可以。还比如像sync 内容一样sync api。建立一个api service server per app level模拟本地远程无差的api服务与结果。即建立一个sync api的维持结果正确，或称反作弊过程，类似fps网络游戏的消息同步模块。

4，一种可能行得通的appmodel设想：P2p 客服同体，只需sync api结果即可的app

因为有了前面的1，2，3讨论，所以这里的讨论方便多了。因为这是xaas,langsys,appstack到final appdev的自然路径。

最典型的就是git，Git，一个客服同体,基于内容同步的云程序,fetch,pull original，都在同一个端同一个操作.在这种情况下，web只是一个cgi转换服务，或类似静态网页生成器的工具，根本不必为它发明一整个orm的大型domainstack中间件。

部署的时候，如果要用上多节点，可以将服务端的部分装在一个xaas os，另一个装在纯粹客户端。

未来会为整个文章描述的devops建立一个engitorbox，其下的开发是all lua codable的：terra/Lua/c，lua scripting, Lua shell，lua sugar.

这一篇应该是覆写序言部分关于xaas,langsys,appstack综合选型的而准备的《一个nativecloudbox:围绕openresty精简工具箱:nginxbox》，我们的新demo会取名egxbox，直接在openresty上强化成,一个围绕着openresty的真native/cloud appstacks与devop工具集，并以此为中心设计实作

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

群晖+DOCKER，一个更好的DEVOPS+WEBOS云平台及综合云OS选型

本文关键字：**dualrunning os bootloader**设想，**dockerized os subsystem appmodel,**云**devops**学编程

经过前面对于群晖的讨论，我们知道它是一个从bootloader到os都很有特色的系统，我们重点讨论了黑群晖的bootloader能安装到不同平台的方式，我们还讨论了如何更好更省事地使用常见群晖套件实现单文件夹同步+同步套件省事同步，及配合访问点服务器使用(frp,公网IP盒子)的方式。，甚至讨论了使用webstation作code snippter空间及利用docker实现类似live code snippter hosting空间的功能，后来我们知道这是devops

综合上群晖总归是一个好用的web化的云OS，这要归结于它可以安装到远端，平台管理和APP都是基于WEB的，也要归结于它支持VMM和docker可以分虚机同时运行多个VM，最重要的还是docker ------ docker是一个同时支持应用和平台虚拟化的东西，docker可模拟subos lxqt效果,也可实现devops.下面我们细细道来。

关于云OS的bootloader pe，从diskbios到cloudbios

pe和bootos越做越复杂的情况有很多，如群晖的webassit，它实际上是一个纯净的pe和liveos,我们称其为dsmpe，包含了大量大容量存储和网卡的驱动的OS，本身并不作为直接可用的系统存在一个安装到实机的驱动解包和适配过程，负责系统功能的是pat，webassit负责的是建立可用的磁盘结构安装和引导pat并后续更新，另外，一台机器开二个同时运行的OS是很现实的需求（one client,one server,非dualboot而是dual running），实际上在xaas系列中我们不断看到过这类系统，如host/guest os(colinux cooperative but not dual running),虚拟机，docker OS，subsystem OS,crossover模拟器,cloudwall(VS前面的方案，唯有它建立起cloudbox,cloudos,clouddevops,cloudappmodel全包的方案)那些，但我们并没有接确到一种能在系统启动时支持双OS会同时启动并运行的PE或工具。

diskbios即是这方面的努力，在diskbios设想中我们提到在一个类似WINPE的环境中实现虚拟机管理的功能，且在发布dbcolinux时我们在一个linuxpe中实现了集成vps管理器的功能，结合OpenVZ Web Panel管理这样的东西，我们可以实现dsmpe类似的东西，多/system(x)这种方案既保留了虚拟机方式也保留了docker方式的虚拟粒度还很自然化。------ 但那文并没有讲到如何使这些VPS运行起来，如何引导进入的方式，------ 提供PE和如何提供双OS，其实这可以是一个相关的问题。

> 当然如果直接采用PE中套虚拟机管理器+集成VPS WEB plane的想法会更简单，但除了WEB plane,还有其它更优雅的方式吗，传统类grub boot的方式会更简单有效么？比如，它或许可是一个强化的grub loader，比如为dbcolinux增加dualos bootloader running功能，可以直接单次boot二个系统，一个OS是linux+vnc thin client，然后选择性boot into guest os through grub.....这些留在后面讨论。

关于云OS本身app和硬件。从minportalbox到minlearnbox,从WEB appmodel到私有gui appmodel

经过前面一系列的xaas讨论，我们明白我们要追求的generic os其实是一种涵盖支持realhw（见《利用colinux打造云环境》）,云主机(见《阿里云winpevirtio装ISO》)，无屏小主机(见接下来文尾《mac mini 2014上装黑群晖》)，虚拟机等硬件装机环境,提供支持os subsystem(win10 wsl,fydeos anriod container,wine appcontainer,linux container) app，webapp,remote x11 gui app,local gui app（dsm lxqt）的云OS，它有这么几个特点，1，要支持多硬件平台，要能以web方式(准确来说，远程都可以,分布式指代远程，也指代一种可负载容错的多节点结构)支持安装分发程序与OS系统本身，2，要支持虚拟多OS APP和多种远程APP，----- 而这，其实就是云OS的一般特征。

我们一直提到和实现的diskbios tinycolinux，就是这个最终目的和generic os的概括，所以现在，这个diskbios不妨称为cloudbios os。

群晖作为一个很好用的WEB化的cloudos。支持web appmodel(page ui appmodel),它还有硬件方案，符合cloudbox->cloudos->cloudappmodel全包的方案，但却没有clouddevops支持，需要挂靠docker，论更符合传统方式的云OS或更集成化的云OS，还有fydeos和cloudwall这种，fydeos虽然是客户端的但是也可以用在服务端，它支持docker os as subsystem/guestos,和多种subsystem appmodel

群晖利用docker很轻松能实现这种docker os as subsystem, subsystem appmodel,linux视图形为APP gui model，加入了协议和网络，形成了remote app model，即x11架构，这对于本地游戏没有优化，然而对建立远程程序天然形成优势。我们在前面说过，任何一种逻辑栈配上一种GUI栈，就是APPMODEL，vs vnc和远程桌面，x11可以直接从docker subsystem中透出，如利用xmingw，或dockerized x11 gui appmodel这种remote app结合teamviewr app窗口模式这种或硬件化的oraykvm这种，如果能做到这个，我们就可以将没有port到那个平台的客户端以remote app viewer的形式投射到那里。类远程桌面。等5G一提速，云streamable游戏和remote gui app就兴起来了，web兴许不用了它的时代就终结，因为WEB始终是一种过渡方案，它体验不如原生，它的优点是管理难实用，像云游戏做成webgame其体验就十分不好。

> 还有，fydeos这种docker as subsystem将docker维持在os subsystem级，还有一个好处，因为每一个docker app总要带系统镜像然后才是分级的APP联合文件系统，docker app总是有点粒度太大，as subsystem复用就强多了，类似anriod container app,linux container app,crossover windows app(docker wine appmodel)，就可以充分抵消一个APP一个OS的docker aufs带来的性能和存储损耗。
>
> 甚至还可以有，私有APP model，unform server/client app model,可以使一个程序的界面都托管在云上，就不用专门的客户端开发了，甚至瘦终端可以是仅带x11和vpn的手机。

而cloudwall作为云操作体统的特点在前面我们讲到是多端原生同步化，还有其devops特性。这利用群晖加DOCKER也能达成。见下。

关于云OS的devops,从yunappmodel到ci backend yunappmodel

云OS最鲜明的一个特别是其对DEVOPS的支持，linux+docker可轻易实现，docker in docker和docker可透露volume1服务可以很容易使之支持CI变身devops。其实本地也有devops,像cpp的sandstorm用的那套ekam，https://github.com/capnproto/ekam可以是devops,gitlab runner based也可以是。关键是有一个可以编译源码且自动化的程序存在，无关乎你将它做进docker还是什么东西，而docker不但能CI编译也能运行构建后的APP。docker很容容易被做成ci builder。

在群晖上利用DOCKER也能实现DEVOPS。

这应该是xaas系列最后一篇关于群晖的文章了。整书关于整个现代编程开发的选型，就是围绕，云OS，云devops开发，云APP，展开的,整个demos选型也是这样，所以新demo中先提出一个os,再jupyterengitor再APP这种，但其实cloudwall这种才是最简易和全包的方案,为了学习起见我们做的是从基础做起的方式，或许我们以后会权宜先在群晖上把dbcolinux docker化，在群晖上把deprecated demos迁移到docker,再考虑后续为它建立jupyterbackend devops支持，甚至自有硬件支持BOX化的方式,所以bcxszy不如叫cloudlearnprogramming。mineportalbox不如叫cloudlearnprogrammingbox

---

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

Plan9:一个从0开始考虑分布式，分布appmodel的os设计

本文关键字：**plan9,Inferno,limbo,Plan 9 from User Space:plan9port**

在《除了UNIX，我们真的有可选的第二开源操作系统吗？》中，我们讲到那些传统的os之争是集中于游戏好不好支持，桌面好不好体验，发行够不够流行，总体好不好用这些方面。而从x86 cpu从0开始的抽象全栈，他们都是一样的 ------- 换言之，某种意义上他们都是一样的OS。

这种共同点在于哪里呢？对于最终的APP和APP开发来说，它们都是基于单PC+单PC下网络程序设计的。于是在这种架构下有了我们现在处处见到的web,云，—— 我们发现，当今的集群和分布式是放在云这个普化架构来做的：集群就是好多好多的PC通过网络计算起来，附带一些PC监控节点 —— 它们还是PC，在每一台PC内部运行的APP，都是从socket开始（更抽象一点，也许还有DCOM，消息件）的“云”程序:这种程序其实还是网络程序，这种总架构下的APPDEV，以OS来看，其实本质都是单机环境下网络交互的程序。

而这些都不是究极的分布式和分布式开发设计。

在《一种开发发布合一，语言问题合一的shell programming式应用开发设想》中，我们讲到了对于任何programming的设想，其实都是一个四栈从0开始叠加的设计。每一个appmodel，都是从hardware从0抽象来的，OS是大件。——— 把这种源头尽早控制在OS的源头，则每一个OS和单机则都天然具有云属性。包括开发。就得到了创新的appmodel设计 — p9，它是一个统一问题，语言，平台的总设计:

> Plan 9不是一个很知名的作品，但是它的前身Unix是世人皆知的。而Plan 9是Unix的几位作者在AT&T职业生涯的一件巅峰之作，是被设计来超越Unix的。 实际上，Plan 9在1992年第一次发布时，就同时实现了Google Docs、Dropbox、Github、Remote Desktop等目前很火爆的互联网产品的功能。 Plan 9能做到这些，是因为它把所有内容都注册到一个称为9P的文件系统里。 举个例子，一个Acme编辑器进程会对应9P中的一个目录acme——我们可以用9p ls acme命令看到这个目录；这个编辑器中的每个窗口对应一个子目录，而窗口标题，编辑内容分别是这个子目录里的文件——我们可以通过修改文件内容（比如通过调用一个shell script）来改变标题和编辑内容。 因为9P是个分布式的文件系统（类似后来的Google GFS和Hadoop HDFS），所以不管用户身在何处（公司、家里、旅馆、咖啡馆）都能看到同一个文件系统。甚至可以在家里的电脑上修改办公室电脑上运行的一个ACME的某个窗口里的内容。或者回家之后，让家里的电脑上运行的ACME访问办公室电脑上的ACME对应的目录，就看到了和办公室电脑上同样的界面——比远程桌面加上Dropbox更加远程桌面和Dropbox。 Plan9没有推广起来，一个原因是它的思想太过领先——在用户还没有意识到存在这样的问题的时候，就把问题解决了。

9p，every problem/app is file io,这也是我们在《bcxszy series》中一直在寻求的分布式方案。

plan9的曲线回归

开发是源于平台到语言到问题的总工程，由设计贯穿，设计包括对OS的设计，OS下编程本身的设计，OS下编程语言的设计，编程方法的设计。—— 所以，先哲学后理论再实现的思路没有错。

9p下的语言。也异于常类。它使用专有的语言limbo作为app langsys，使用c作为toolchain。

> Inferno操作系统是Plan9的姐妹操作系统。它的思想和Plan9基本相同，都是基于文件的。但是它只有内核是C编写，其他的应用程序都是Limbo编写的。所以它和Plan9不同的地方就是在这个系统上运行的程序都是Limbo程序而不是C或C衍生程序了。后来Rob Pike又开发出的Go语言有一些地方的思想就是借鉴于Limbo语言。

Go是limbo语言在linux的再生者,Go 语言的实现带有9p的深重痕迹，即使在x86上，也使用Plan 9的汇编器,为了实现所谓的语言自举，硬是绕开glibc去自己用汇编封装linux系统调用。可见plan9一开始就想彻头彻尾的自立门户，对传统OS和GNU没有任何依赖。

虽然历史上都选择了C family as toolchain和unix as os，没有选择9p和limbo，go，然而这不是9p的错。是工业和市场的错。

plan9 under linux

虽然历史上都选择了C family和unix，没有选择9p，但9p可以是一种附加而不是替代。bell labs的9p是主，其支流也有一些。在linux下使用9p的方案，有Plan 9 from User Space:plan9port

或许我们以后在新ebcolinux rootfs 设计中编译plan9port，我们用plan9 under linux，terracing for go

---

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

打造一个Applevel虚拟化，内置plan9的rootfs:goblin（1）

本文关键字：**applvl Virtualization**。去中心化，无架构**APP**

我们知道云计算最大的特点就是虚拟化，将云虚拟化，将裸金属虚拟化，将CPU虚拟机vmm，将平台虚拟化，将语言运行时虚拟化，将appstack虚拟化。这种“云化，虚拟化，xaas化”动作其实正是传统最基本的抽象，正如保护模式划分CPU，进程划分任务之于”单机运算“一样，这些只是在传统架构上造云的步骤，本不足为奇 —— 但这些所有虚拟化，却唯独少了最后一个层面，将app本身虚拟化，这种工作似乎鲜少进行。

这是因为我们以为程序都是堆成的，一层一层，只要解决了上面的虚拟化，得到的最终会是最小的虚拟化，从上而下地虚拟化，和从下而上地将APP虚拟化，，这二者的复杂度和工作量都一样，但其实，对于最终开发者来讲，这二种方案造成的复杂度其实有着天地之隔。传统从上而下，开发者需要面对所有的复杂度，只是它们被抽象简化了，而从下而上，最终开发者仅需要面对领域逻辑。虚拟化方向和粒度的不同导致的差别很大。

golang的去中心无架构设计

在《golang》中我们谈到golang是一种将portable langvm植入app层次的语言系统，由于这个langvm是与x86无关的非常小，而且go语法是类过程C的非常精简，所以它带来了分布式和分布式开发的新语言规范，—— 大概没人想到会将一个单APP demo和运行层作为架构来集成，但事实是，go将语言运行时虚拟化集成到单app的方式也证明十分好用。我们知道，golang在源码层有着日常可见的那些deps，在发布层开始显露其本质不同：每个Go发布出去的单元，只是发布级/二进制级无依赖。无须依赖：对于移植度，它将解决移殖的方案集成到app内部而非一个共享虚拟机，对开开发和通常行为，这种运行层的去依赖，去堆栈化，打破了传统程序是一层一层堆起来的现实，没有一个对OS，langlibs,3rd party libs的复用，引用中心。也就把架构去掉了，这像极了去中心化和无架设设计 —— 这种“去中心，无架构设计”什么好处呢？这种使得所有跟APP有关人的活动局限于本app demo运行时内部，它使最终开发者不需要面对一层一层继续下来的复杂度，再开发者不需要管任何东西。—— 这才是一种一了百了的工作，这种虚拟化才是不需要管的东西，只要保证每个app内部的这个虚拟部分足够小足够可用，反之，一层一层的虚拟化，这种复杂化成本还是给了开发者。这在源码级也是一样的，我们以为一层层的抽象解决了大部分问题，领域逻辑变成了最终开发者需要面对的，但最终现实中开发者却看到不一样的事实。

applvl全虚拟化设计

虚拟化可以交叉设计。关键是找到正确好用的粒度，那么，除了golang，这种APP级的虚拟化重设计还有哪些呢？相似的设计还有：在前面《hyperkit》一文中，我们谈到这是一种将虚拟机植入app层次的设计，我们在《plan9》中还谈到，plan9是一种定义了9p分布协议的分布式OS，原生的plan9 os是受plan9 kernel支持的，但也存在一种usr space plan9 rootfs，可以用上9p协议。那么，能否将它也欠入到具体app中，，比如，将plan9 roofs 作为busybox欠入到每个app中。。而plan9和busybox shell式programming，如果还能加上go,其好处是：go这种本地程序，本地开发和远程程序，分布式开发合一的语言系统，无依赖，无架构，才能集成，无架构，才能post as app，程序不是预置的长设计蓝图，而是实时的调用，自包含，但并不意味没有复用。

这样就带来了一种applevel的全虚拟化，前提是，保证产生的每一个APP要尽量小。

———

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

云APP，virtual appliance：unikernel与微运行时的绝配,统一本地/分布式语言与开发设想

本文关键字：云时代没有软件，只有服务，虚拟**app**，虚拟**OS**，虚拟**APP**开发**,metarootfs as service,container as service,virtual appliance**，可**devops**编程**os**，**Redox OS**，融合**app**

我们知道，OS的选型，其实关乎着开发，因为它位于开发四栈中的平台栈部分，（系统开发语言对”系统平台“进行开发）平台与语言产生联系的方式首先是支持该OS开发中的toolchain language中的runtime（kernel space或user space中的libc），它以OS为宿主，将系统服务做成API件，将系统实现系统开发接上语言和开发。

与之产生比对的，就是该OS上的各种应用开发层次上的开发语言的后端，比如脚本语言，它可以是各种port到该OS平台上的虚拟机平台作为runtime接入到os，这种软件"虚拟机os"+“虚拟机平台上的语言，包装来自原OS toolchain的API或OS可开发服务即可，照样可以作平台上的系统开发。（比如c,cpp虽然可以桌面应用开发，但是它有开发效率不快学习曲线问题和内存泄漏问题而桌面开发不要求指令密集且不能跨平台，那就可以完全可以另起灶炉发明语言和一整套开发系统换成py这些）。甚至最后，虚拟机语言和脚本更多面向WEB，因为这种“API服务可以来自不同OS不同进程。的虚拟平台，不是OS也可以“。，云时代没有软件，只有服务。在《编程语言选型之技法融合，与领域融合的那些套路》和《一种matecloudos的设想及一种单机复杂度的云mateapp及云开发设想》中，我们都谈到，传统OS和语言是面向nativedev的提供语言系统和开发API的。，对于web和分布式来说，因为API和语言都不需要来自这类平台。更适用这些虚拟机平台和服务开发。----- 平台，语言后端，这二者的界限模糊了，语言可对虚拟，脱节平台进行开发。

后来又出现了容器，“语言后端即服务”。（baas,容器化，对语言来讲容器其实还是OS或虚拟机，它类似一种wsl，只不过它可视为分出来的子OS，这种子OS作用是为语言提供分离服务，而不是其它目的，比如pyenv,和serverless后端）。当然它们不是语言后端仅可视为后端包装器 ----只是这次，os平台，后端，容器，这三者的界限又变得模糊了。

以上这些都是平台与语言可完全脱节存在的例子。实际上，只要是脱离平台，该应用语言能对任何系统进行开发，甚至架空平台。

但是，它们本质上，它们都没有做到完全脱离。语言后端作为语言在这种具体平台上的实现支持语言在这种平台上存在，就是关系的纽带，不管何种形式存在（是对OS作后端还是OS上的虚拟机作后端），都有一个后端，都有一个OS存在：无论这种开发体和托管体，是single OS和APP之间，还是云服务时代，容器这种单元，只要它装配了语言后端，就可以解释为RUNTIME与OS的关系，这样这种后端与APP之间存在完备的运行支持关系才使之成为可能。即，它们是多平台而不是真正的跨平台，语言也并不是跨web/native开发，至少存在一种toolchain lang和一种appdev lang。（这里要明白一点：web不是平台是业务，lamp-l=amp才是runtime targetee,browser也是）

那么有没有完全脱离平台的语言呢，或者极大可能这样做的语言呢，这样做的好处又是什么呢？能给最终的开发（尤其分布式，WEB）带来帮助或颠覆性好处吗？

语言与平台脱离：极小化语言，微语言

这需要从语言运行时本身进行。

这种语言和开发生态就是go和rust，这种极小运行时的语言。这就是我在《Golang，一门独立门户却又好好专注于解决过程式和纯粹app的语言》谈到的。go为每一个它生成的APP都内置了一个“go runtime as os”，而这种go runtime是不依赖任何平台甚至最初级的libc。

这种极小化生态，可以让语言更容易跨平台，甚至集成到浏览器（这是一种除了硬件嵌入开发的WEB嵌入式开发）。比如rust micro runtime之于wasm开发。而带GC的语言，往往不能用于嵌入式。

它带来的更大的好处之一，还可以使得virtual appliance成真。

virtual os与virtual appliance: 内置OS与language runtime的app

在云计算机和虚拟化领域，有virutal iaas，就会有virtual os，和virtual appliance。见《一个隔开hal与langsys，用户态专用OS的设想》，《一个设想基于colinux,the user mode osxaas for both realhw and langsys》。

上面谈到容器，作为language backend包装器存在，它也包装了OS。比如docker，实际上每个docker实例都共享着一套OS模板在最开头，其aufs文件系统的开头，也引用着一个OS rootfs。而现在，随着unikernel的出现。这种专门为OS kernel进行虚拟化（而不是在OS内部共享内核针对rootfs虚拟化的docker）的方案出现了，它使带OS的容器可以真正变得极小。它针对解决的问题是针对容器每一个模板过大的问题，就如同rust之于runtime一样,从源头解耦。

再来谈virtual app,最初的virtual appliance是本地虚拟机跨OS出现的一种APP管理机制，如vmlite的那种，pd的融合APP那种。而rust的微内核，使得用这种语言这种开发出来的APP，是无须调用或依赖任何OS导出的API或服务作为宿主的：不是可不可移殖，是根本无须移殖（OS与APP可脱离且最小化，这就给APP虚拟化提供了基础）。

如果说应用开发领域，OS可以虚拟化，模糊它作为app backend的意义，那么app为什么就不呢？关键是这二者一旦结合，将带来真正的virutal appliance和开发。

1）首先，Unikernel才能促成application virtual，微内核将容器做到os管理的级别跟openvz和docker方案不一样。微内核可以集成到APP级别。使得virtual appliance真的跨OS有存在意义。

2）其次，使得虚拟OS可以被集成在APP内，见《hyperkit:一个full codeable,full dev support的devops及cloud appmodel》，实现真正的virtual appliance，meta rootfs中集成虚拟机管理器。

3）1，2相加，针对那个最终的问题，可以带来更好编程的OS和分布式开发，使分布式真正集成到OS，在《一种matecloudos的设想及一种单机复杂度的云mateapp及云开发设想》，我们谈到现在分布式app都是跨OS的，但也存在一种原生分布式，即plan9,x11这种类似的“os绑定式分布式APP”，

该如何理解这种新appstack呢？传统的app是完成了gui,db,net之后在这上面，面向本地，堆应用，作为应用件，传统webapp是提出一系列关于db.gui,net的新规范（其中net部分固定为http）实现真正跨端，而新appstack是os限定的。跟传统os一样(受限于os透出的api/分布api)。新app是gui,db,net这三者，每一个都api化。且面向分布api化。作为开发件复合堆net关于分布网络。这里的思想类似于直接利用原生的os存储功能提供API和网盘服务。开发上，os即客服一体机器。语言库即os组件，见《一种matecloudos的设想及一种单机复杂度的云mateapp及云开发设想》第二节。

1，2，3可以使得nativedev适用于新的分布式开发，也使得传统web和web语言更容易拥有virtual appliance能力:webapp是初级的分布式和融合开发和融合栈的一种方式，因为它还没有涉及到virtual appliance和微内核容器化。任何应用编程，都是栈选型和归纳行为。归纳为有限几个栈。业务领域自有轮子。却没有真正的virtual appliance，因为web自身就是一种byond os的虚化平台，因此与web最搭的就是这种virtual appliance的appmodel。b端甚至可以不是浏览器直接是普通GUI。甚至结合rust microruntime和interpreter to js(这实现也是elmlang的特色)，可以用rust统一前后端开发。

---

rust的生态在慢慢涵盖我上面提到的那些rust coreboot，redox os微内核，User-space drivers 用户空间驱动，wasm,都在慢慢成真。目前这些只有molliza在做。感觉技术也是烧钱割韭菜。不过这套方案也确定是够slim省事。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一种设想：为**linux**建立一个微内核**,**融合**OS**内核与语言**runtime**设想

本文关键字：**os**之争。微内核**,language based os,language on bearmetal not on os**，华为鸿蒙，语言即**OS**，类脚本语言，把原生应用变语言模块。

我们知道，OS兼容跟语言兼容一样，是一项几乎不太可能完成的事情，因为OS的使命就是作为闭环竞争的商业产品出现，成就造出它他们的公司。从文件系统的多样化和存在的互访鸿沟就知道，见《一个设想：基于colinux，去厚重虚拟化，共盘直接文件系统安装运行的windows,linux》。osx有apfs,windows有ntfs，linux有btrfs/ext，难得出一个exfat，又仅是作为数据盘存储格式不能作为系统区存在而且也有坑（不过也听说过在exfat上成功装系统的）。os之争与语言之争一样，技术上不是不可能融合，而是厂商各自为政。不想那么去做。

但幸运的是，软件是抽象的堆栈，业界方面这些融合一直在进行，只是比较缓慢。融合可以从源头(对于os是kernel)完成，还可以在别的层次完成，类似《编程语言选型之技法融合，与领域融合的那些套路》谈到的那些语言融合。

OS/OS内核/OS的融合经过了很多发展阶和变形形式，如日常我们使用的PD,VB虚拟机管理器里的os，wsl(as os subsystem)，虚拟机管理器on baremetal(qubeos,etc..)，虚拟firmware内嵌虚拟机管理器，os级的虚拟化openvz，简单的chroot沙盒os，iaas大数据级别的kvm，最后，就是这篇文章要谈到的：从0开始，和从os kernel源头进行融合OS和语言系统的os。即“微内核+unikernel”。

> OS的融合其实和语言/语言runtime的融合其实可以是一个相关的过程，如《基于colinux的metaos for realhw，langsys和一体user mode xaas》如《一个隔开hal与langsys，用户态专用OS的设想》中关于让os的子系统直接服务于语言runtime的思想。我们在《云APP，virtual appliance：unikernel与微运行时的绝配,统一本地/分布式语言与开发设想》中也讲到语言的后端和容器和OS的关系(通用容器和语言虚拟机/语言沙盒/baas/serverless，这些东西的存在，其实是”可移殖可隔离runtime“的功能重合。正是面向不同方向但同一目的的某些意义上的“重造轮子”。)

安全的内核级语言：**unikernel**化

os与语言的关系从来不是先有蛋还是先有鸡，而是先有蛋（语言），再有OS（鸡），自举语言和自举OS特殊情况另当别论，历史上和我们现实中出现过的常见OS，其kernel都是C开发的（我仅指osx,linux,windows kernel），，c runtime作为toolchain系统实现层面存在。c库存在用户空间和程序空间的部分是有特权不同的，它们之间要跨syscall互访。应用开发层次的语言系统处在用户空间(由此,官方py版本这种语言是不可能作内核编程的，抽象层次太高运行时也巨大)。我们把内核中的c称为“bearmetal或toolchain”语言，因为它们最开始的作用是给硬件平台提供软件管理层的作用。

由于语言放在OS之前设计（没办法，先得用起来，那个时候还没有出现既能保证开发效率运行效率又能保证安全的语言如rust），架构于os kernel之上。于是kernel的二大机制（内存管理和任务进程）会继承C语言的固有缺点，比如运行在这种OS上的程序会发生内存泄漏，这对于系统实现和APPDEV级都是延续的影响。我们现在的OS，如果它们自我宣称它们是“安全的OS”，其实是某些高阶层次的完全，比如OS应用方面的安全。并不是“内存管理和任务进程”方面的绝对安全。运行在这种OS上的APP还是会发生内存泄漏，因为“程序申请内存”这个功能，无论处在编程的哪个层次，都是从OS继承的能力（任何编程都是某种意义上的系统编程）。GC只是自动释放并不能改变OS内核这种可能的缺陷（这只是一种可能，OS内核的质量会保证这种情况很少出现）。

那么，为什么不采用一种从一开始就足够安全的bearmetal语言呢，如rust？

当然可以，语言放在OS之前设计可以直接省掉OS发明的很多问题，如cpu的保护模式甚至都可以略去，特权模式无关紧要，一种内核天然是bearmetal和usermode通用的（驱动和hal层放在用户空间）。你甚至还可以将rust的stdlib整个实现在bearmetal层,使得内核编程跟系统开发编程/系统应用编程共享同一门语言。

这些特点加起来，可以直接用语言作为虚拟机管理器（这种语言系统和OS最绝佳的关系，正是另一种os kernel-libos的特点：它可作为lib的os被嵌入语言使用，用户态无须经过虚拟机管理器。直接为支持“一app一os”而提出的os）。因为你可以把语言系统作为OS的源。直接compile and run一个kernel出来，使得语言架构于OS之上，做成devable os。在《云APP，virtual appliance：unikernel与微运行时的绝配,统一本地/分布式语言与开发设想》中我们提到unikernel是docker等lanauge runtime的代替品。这里是“more safer unikernel”。

当然，为促成microkernel+unikernel，这还需要一些额外工作。比如，将这种内核裁剪下，去掉“不那么重要的部分”。

采用了安全的语言，我们还要给内核“微内核化”

首先，这种内核要小。仅保留必要的“内存管理和任务管理”。如上，我们知道，内核中最关键的是“内存管理和进程管理”,因为这二个部件，足够可以运行某种“APP”(内存和并发，刚好它们也是语言runtime的重大问题之一，所以这里“语言runtime即OS，这种特征又一次出现了”），除此之外都是addon,非essential的。hal,drives，文件系统,gui都是第二级别的。我们可以把这种OS的“驱动”保障效率的前提下放在user space（如上，实际上，这种OS kernel下，没有内核空间和用户空间的区分了）。传统上，我们将hal和driver一部分放在(firmware或boot层)，一部分放在os kernel层。

这就是微内核。OS层的其它组件可以通过ipc通讯放到其它具体os kernel下，它本身可以很小（作为meta os kernel与其它内核合作，管理这些内核并融入这些内核,这也使这些内核可以实现真正的跨OS）。比如给linux kernel,这种monolitch一体化宏内核作上层，复用它里面的东西。

综合上的microkernel和unikernel，如果融入现在的linux，比"用虚拟firmware内嵌虚拟机管理器打造applevel虚拟化的容器和开发方案"这样的办法要强多了。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

WinPE VirtIO云主机版 支持west263 阿里云aliyun 送精简win2k3镜像

关键字：**winpe kvm,winpe xen pv,winpe virtio,winpe ecs,winpe vps,winpe**云主机版

亲是不是有维护云主机的经历？

云主机虽然是虚拟的，跟本地电脑系统有诸多不同。但其实在安装维护方面还是有很多相同之处的，比如，一般服务端后台都提供了系统重装/全盘备份/恢复功能，对于各种镜像的需求，甚至阿里云还有镜像市场（west263是按镜像收费的）。可是总感觉好像少了什么？

是的，，以上都是服务商默认功能和第三方开放给你的镜像，，，你始终不能像本地一样方便随时随地在本机切换镜像，备份镜像。或存储，制作，还原自己的镜像。按特定盘符指定备份还原。

你或许会认为云主机的ghost功能可有可无，但其实事实不是这样:服务商的后台备份功能不是100%保证成功的(它们只会在页面给你显示成功字样)你最好自己能备份一份。别人给你共享的镜像是未知的，由于各种定制因素的存在你终究需要自己做镜像：

为什么这么说呢？由于服务端后台备份不可能保存二份备份，所以下一次你恢复的镜像可能是带冲突或残破的，这种情况下保留一份自己的备份就额外重要。事实上，这种情况很常见，我在west263上恢复的镜像虽然能工作，但总与我一次备份的有出入，而且我终究需要封装有特定功能的镜像，且能像GHO文件一样的多版本历史镜像或保存不同的系统GHO文件来切换当前系统而不致于一次次重装。

这使我下决心在云主机上安装带ghost的WINPE，这就是如下本软件给你带来的功能和便利。可以做到自己操作，本地/可视，安心且功能更灵活，可以脱离服务商备份机制作二次备份。

功能特点：

软件封装了virtio硬盘控制器驱动，可进入桌面后会自动挂载你所有的virtio硬盘和分区，其它跟普通WINPE中操作ghost相当。见图片区。我一般将镜像保存到第二块数据硬盘(不会被后台故意的,或误恢复之类的操作覆盖)，再在服务商后台备份一次。

集成iso文件，支持全新安装/gho二种方式。在开机菜单，选择对应操作即可。进入系统后程序有PQ,ghost,ntpassword,winrar等。

注意事项：

此WINPE支持阿里云，west263的是kvm,virtio驱动，阿里云普通非优化机型使用的是xen vm,pv 驱动。优化机型使用的正是virtio驱动。 除了以上这二种，支持包括使用virtio的全系列主机。

而且，你的云主机须支持vnc或管理页面/终端管理可以在启动时选择菜单项。west263自带VNC 此版本对鼠标支持不太好，甚至有些只能键盘快捷键操作！！

安装ISO和送的GHO文件是基于61适度精简的win2k3。安装包仅178M，镜像不到500M，包含全部服务器功能，没(mei)有administrator密码。全部免费送！！ 下载地址（站内下载）：

http://www.shaolonglee.com/owncloud/index.php/s/yeKnfbK67f4MXo8

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

将virtio集成slipstream到windows iso,winpe – 原生方法和利用0pe

本文关键字：集成**srs**到**windows**镜像**,slipsrteam driver to windows iso,**集成**virtio**到**winpe,winpe**集成**virtio**

目标：

制作一个winpe，使得其在virtio主机环境中可用，比如阿里云ECS或WEST263，godaddy云主机。

这种技术的本质是将srs驱动带入/植入需要这种驱动的镜像，让其在安装/启动过程中发现硬盘不蓝屏，且辅助其它安装过程继续到完成。这一般有二个步骤，第一步是将驱动手术式的植入iso，实现安装程序的对应这种srs设备的内部驱动支持和发现。

> srs驱动是boot time driver中一类特殊的驱动，因为安装过程往往需要接触到硬盘对其识别，否则会BOSC，且往往是txt mode 的驱动，有别于pnp把inf和driver .sys直接往windows\下一抛就可以的方式，那属于次级驱动，即进入系统后增强系统的后期 设备驱动。在windows安装过程中，这类srs驱动往往是已经存在于注册表和文件系统配置文件中的，所以为了增加一个SRS，我们 必须手动slipstream相应的文件和设置（甚至要修改注册表到.hiv文件）。
>
> 如果ISO能自发现带入的驱动 — 比如它是一个安装iso,安装过程有F3选择驱动接口，这种情形比较干净，可以实现从外部将驱动带入到ISO而不需要修改它（当然你也完全可以事先集成它实现自动发现，这样情况跟要谈到的livecd是一样的了），但如果是livecd iso – winpe则是一种windows livecd,则需要集成这个驱动，且保证正确加载驱动后的镜像结果是预期正确能工作的–对应能驱动你那个需要驱动的安装盘）

第二步，实现外部发现光驱镜像过程，即用grub4dos配合winvblk从外部带入光驱镜像。

可见植入srs,与配合grub4dos+winvblk导入光盘镜像是这二大通用过程，这种情况下0pe就是一个强有力的工具。因为它几乎是专门针对这个问题提出的一个整合方案。当然也可书写grldr菜单和手动集成驱动。

注意：目前所提到的全部针对windows iso。因为它识别grub4dos+winvblk带入的驱动。你可以参照其它尚不支持这种方案的ISO带入SRS驱动的方案，比如本站reactos0.4.x增强系列。

下面继续，我们将在winxpsp2下植入virtio驱动到deepinxp iso来描述这个过程,植入到winpe是同样过程，只是pe加载的grldr稍不同：

值入驱动到ISO

手动方法：

修改镜像中的txtsetup.sif（WINPE和windows安装盘中的i386中），加入驱动盘中的对应txtsetup.oem中的chunks到txtsetup.sif，一般有files段，scsi段，scsi.load段，hardwareiddatabases段。注册表的部分好像并不需要（hivexxx.inf->setup.hiv）。把驱动放到system32\drivers下。保存改过的iso.

自动方法，准备素材，利用工具：

- Deepin XP SP3 完美精简版 V6.2 ISO文件，11/20/2013,51.65.104.7400版virtio netkvm和viostor驱动for winxp
- 其它工具：nlite,grub4dos，WinBuilder0.78.exe和定制的vistape脚本，还有一些for linux，在linux下将ext变成windows winpe盘的工具，在wwwroot下
- 准备补丁：对于for deepinxpsp3.iso的补丁：HIVESFT.INF,LAYOUT.INF,SETUPREG.HIV,etc..由于下载来的deepin iso信息不完整,winbuilder处理它时会出现好多信息不全的情况出现，包括一些文件大小写错误，故需要修正。请下载全部工具尝试得出修正差异。

处理方法：

- 解压deepinxpsp3.iso到一个目录，比如我这里是D盘，利用nlite工具，将virtio 驱动 slipstream到镜像中。
- 打开winbuilder，生成winpe。

grub4dos+winvblk引导

准备peboot，文件组织情况参照提供的peboot.rar，注意的是引导文件中的这几条：

```
title (Winvblock) Boot RamPE From ISO -- filename 0pe.iso
find --set-root /boot/imgs/0pe.iso
map --mem /boot/imgs/winvblock.img.gz (fd0)
map --mem /boot/imgs/0pe.iso (0xff)
map --hook
chainloader (0xFF)/I386/SETUPLDR.BIN

title (Winvblock) Boot WindowsSetup From ISO -- the 1st step,filename winxpsp3.iso
map --mem /boot/imgs/winvblock.img.gz (fd1)
map --mem (md)0x6000+800 (fd0)
```

```
find --set-root /boot/imgs/winxpsp3.iso
map /boot/imgs/winxpsp3.iso (0xff)
map --hook
dd if=(fd1) of=(fd0) count=1
chainloader (0xff)

title (Winvblock) Boot WindowsSetup From ISO -- the 2st step,filename winxpsp3.iso
map --mem /boot/imgs/winvblock.img.gz (fd1)
map --mem (md)0x6000+800 (fd0)
find --set-root /boot/imgs/winxpsp3.iso
map /boot/imgs/winxpsp3.iso (0xff)
map --hook
chainloader (hd0)+1
```

使用0pe

在0pe中植入srs驱动我们用它的自动选择方案，即在ope\srs\FREQUENT\放一个viostor.sy_，0pe\src\CHKPCI.TXT放一条(具体值即打开viostor\txtsetup.oem查看)

$PCI\VEN_1AF4&DEV_1001&SUBSYS_00021AF4 VIOSTOR 在CHKPCIDB.GZ->PCIDEVS.txt中你也可看到0pe对redhat virtio有支持。

至于引导过程，它在加载驱动后会发现optdesk.wim，然后继续winpe的加载，最终完成进入过程。当然你也可以用下一步菜单实现二步安装windows，或自写更多的菜单实现更多功能（这完全是grldr编辑问题。）

对比virtio winpe，与众不同的是virtio 0pe版本的winpe可以借助netkvm连网。且有更多外置工具可用。

而virtio winpe支持将linux winpe盘变成windows filesystem的winpe盘。

注意事项

使用w2k或winxp内核产生的winpe在进入系统时，鼠标可能会出现不能使用的情况（这好像是虚拟机USB驱动冲突通用情况）。至于netkvm完全不必像viostor那样麻烦完成可以采用pnp的方式把对应inf和sys放到光盘驱动镜像中。

生成的virtio winpe：

http://www.shaolonglee.com/owncloud/index.php/s/yeKnfbK67f4MXo8

0pe virtio winpe:

http://www.shaolonglee.com/owncloud/index.php/s/dD8dm8c9FPcAUje

---

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一个设想：基于**colinux**，去厚重虚拟化，共盘直接文件系统安装运行的**windows,linux**

关键字：**uniform windows,linux diskbios**，虚拟机作为装机系统，元操作系统**host**，共用盘**windows,linux**设计。。**diskbios**，带**iaas**的云装机。。

在《除了LINUX，我们真的有可选的第二开源操作系统吗》中的末尾说过，windows起码有一个先天优势是它整合图形到内核，这当然不仅是发布方式上的区别（windows整块地发布交到用户手里，而linux厂商各自为政，内核和桌面分别发布生态碎片化），还可能是导致更深层的区别发生的地方（比如渲染性能和游戏生态，而windows2008后才有类xming的remoteapp） —— 可综上，一定程度上它的确直接导致了linux依赖太多专业的配置，好用的linux发布版太少。都不及windows那样“亲民”的现象。根本上，windows,linux这二者分歧从生态的开始处就过大，除了界面，使用的文件系统都不一样，文件系统这种鸿沟尤其巨大，导致从装机阶段开始不同的磁盘格式不能共存一区（forget the virtual filesystem or image），进而导致了后来体验等一系列后来难题。这二者鸿沟是天生就巨大而存在的。——- 这是它们的哲学和选择问题 。

可，之于用户，我们总希望有一套统一的从头开始的体验方案，至少，我们希望有一致的可安装到的文件系统而不致于要分二个区安装不同的OS处理复杂的硬盘分区和互访问题。我们还希望天生集成界面拥有图形文件浏览器的装机和使用环境，能直接以统一直观的方式操作计算机本身和最重要的文件资源，然后视需求使用不同的操作系统（windows,linux只是一个内核导致的区别而已，如果能共存并同时运行，那么即使linux不能拿来玩游戏开开服务器程序也是好的,合理共存才是大流。），这就有点虚拟机as装机系统的味道了。——- 更科学使用PC，在PC资源允许范围内，使不同的OS完全可以像安装应用一样被安装进来且运行多开运行，这个考虑的诞生是无比自然的，这就要求一套统一的装机和容器/虚拟方案。

没有一个平坦的统一入口反而正是win系,unix系的最大区别，我们的工作是提出一套方案，让windows,linux从入口变得体验平坦，统一，包括上面说到的图形集成发布和统一的文件系统，都可能是我们要解决或集成的对象： 图形问题好解决，往linux集成就够了。 从装机端，搞一个共用的文件系统出来。。这个umsdos已经做过了。然而方案并不完善。

至于共用,有龙井核心这样的方案，或WINE或CYGWIN这样的东西，这些都是OS内部集成层面的，虚拟/虚拟机和多开OS，才是我们这里谈到的外部全局，“PC装机”级相关的那些问题。 这些虚拟多开方案有像COLINUX这样的往实机同时安装不同的操作系统这种自然，简单原始的方案，和现在虚拟机，和基于虚拟机和云计算下的IAAS一样的方案。前者colinux将是我们要采取的。

虚拟机**or colinux**？

虚拟机有虚拟机的缺点，IAAS也是如此，iaas中，必然的一步，是将传统的os分裂成虚拟子计算单元的os部分。分布式运算从OS环境开始的计算，称为云计算，它往往与kvm,qmeun,vmware,hyperv等虚拟机方案结合使用。

云计算基本就是一个将本地计算能力虚拟化的过程，比如平台虚拟化iaas,语言后端虚拟化baas+baas，应用容器虚拟化docker,甚至库中间件化，software+api服务化saas+microservie app化 … —-其中的平台和应用虚拟化是重头，在不同层级有不同实现，然而它们都属于过设计，单生态和虚拟化的问题就是：使一切步入单生态，不断虚拟化的”怪圈”。

而很多单生态设计有问题，比如java langsys，同理，有很多虚拟化，比如iaas,paas，(docker,虚拟机，vboot,vm langsys,云端软件系统，)都是过设计，不当设计，会导致问题。。虚拟化这种东西，除非有好的封口，否则不宜做在用户层，要封好做在用户不可见的系统层，要么干脆不做虚拟，因为始终会导致问题。“不要过早优先”始终是编程界的格言，编程其实是语言系统作为软件的应用的过程，这句话的底音依然是：各软件抽象应该合理推迟到它们应该处在的地方，不要过早地替用户下决论。

所以，关键是如何去“整合”。

而回归整合才是本质，抽象不应提出新的中间层，而是掩埋旧中间层的过程。colinux这种才是从原始的方案着手，是“在入口处提出更多平坦化”和非厚重虚拟化方案。

我们要达成的方案：去虚拟，尽量平坦，推迟面向虚拟的集成

我们要解决的问题有三：

1，我们要能在PC终端机上，物理实机，VPS上，和虚拟过一次的云主机上，不具备VT的任意机器上，IAAS母机上，都具备像安装应用一样安装OS的能力。

colinux完全可以是hadoop+kvm之类的东西，甚至openstack的iaas,paas，所幸它用的不是vm技术。而且，最新版的colinux特性支持的操作系统除了windows系列，居然还多了个Linux 2.6.x，它支持linux as host

可它居然用了windows为元系统且只运行于32下。。这就尴尬了。。64位能裂变出32位的子系统，32位甚至不能用尽4G的内存。。还是linux hosting windows好啊。。而不是反过来。。 2，为了在入口处足够平坦化，将虚拟化这种非到必要不必做的方案推迟到用户或后来，我们要保证linux/windows文件系统要共盘，最好像安装程序一样安装到同一个盘的不同目录。这样就解决了统一装机和以后多系统共享数据的问题。

umsdos还是不够好用的。然而这是一个很好的项目，我不知道它为什么在linux内核中被放弃了。反向使windows具备能安装到ext2的能力也是一个考虑方向。在umsdos这样的方案达成后，可以使colinux直接mount文件夹从中启动。 3,making colinux to be the meta os,not only the hosting os:

未来可能会做一个diskbios装机核心，以colinux为基础作为metaos,使windows也可以作为colinux的guest os。

———————

如果以上1，2，3最终能形成一个diskbioslinux装机系统的东西，或虚拟平台，那么它支持任意环境的多OS多开，对于应我之前提出的anti应用虚拟化enginx，它是anti平台OS虚拟化方案。而基于qtcling的langone可以使不同语言共享一个运行时和一个生态，用的也是anti语言虚拟机的方案，这三者共同组成1ddsoft demo series的基础建设部分（平台，语言，应用）。都是为了解决业界不断过设计导致的问题而提出的。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

阿里云上利用**virtiope+colinux**实现**linux**系统盘动态无损多分区

本文关键字：利用**colinx+virtio winpe**定制**aliyun**多分区**linux**系统盘,在**winpe xp winpe**中运行**colinux**,在**windows pe**下真正操作**linux**分区，利用**colinux**作单硬盘分区扩容无损分区， **bootice**安装**grub2-00** 到硬盘,云主机越狱装自定义镜像

在《发表virtiope》《在阿里云上装自定义ISO》和《定制virtio winpe镜像》系列中，我们用类似手机越狱，刷机的hacker workarounds（围绕virtio pe为中心,利用一系列小工具组合）的方式达到了使云主机装机变得像本地PC一样方便流行的维护/管理方式，，那里我们侧重讲解的是windows，任务结果是将自定义ISO装成了系统盘，将系统盘分成二个区并把备份后的镜像装在第二个区内，这样借助virtiope和一系统一数据区的双分区设置可以恢复一个全新的系统。

在《一个设想：共盘文件系统的windows,linux》一文中提到过windows与linux二大OS文件系统存在巨大鸿沟的现状，及我们希望得到一个共盘的uniform windows linux fs的要求。本文中，我们将在装机领域，探索一种“在winpe下自由操作linux分区”的目标与可能。----- 文章最后，探索为单硬盘单分区下的云主机linux分裂为二个分区，打造一个类PC和手机recovery的可恢复rom机制，只要这样，在装机和实用阶段，都能完成某种“共盘，实用的windows,linux融合方案”，那文提到的设想才能基本变得“像那么回事”，也算有技术参考方向。

> 在winpe下操作linux分区的难点，在于它不如ntfs受windows中的磁盘工具如diskgen,pqmaigc之类与其结合支持得好，在windows下用此类工具操作EXT3，要么不受支持（需要特定驱动且这类驱动往往很原始），要么能读不能写ext分区，要么能写但是频频蓝屏，更别说动态对其调大小，与类gho方式恢复镜像等（diskgen493开始支持格式化EXT3，也不行，稍后会讲到）。甚至格式化都很久
>
> 关于单分区linux动态扩展出新分区有LVM这样的方案，但是要求在业已分好标识为8e的分区格式的情况下进行。

我们的总目标，还要打造一个windows,linux二合一的pe维护盘(保证一切在该xp based winpe下完全，且不需要二次进不同的ISO环境，比如合盘的windows+linux pe)。这一切我们将在1g内存的阿里云预装了ubuntu14.04 32bit的一台机器上完成。下面开始：

在阿里云上利用**noimagecolinux**实现**linux**系统盘的动态分区扩容

这里我们额外用到的virtiope工具有（除了原来封装于virtiope的四个:showdriver,ext234reader,bootice,ramdisk），还有：winpm 7 服务器版本for winpe，它用来分出新ext3区。，还有colinux noimage（busybox我们能用到的工具有mount,tar,cp等等）用来重建系统：众所周知colinux，根据我的《发表colinux》，它被定位于guestos，可是它本身也是工具，colinux可以nomiage的配置形式运行，可加载windows目录为分区也可加载本地硬盘为分区。不加载任何镜像的colinux自带busybox，可以实现在windows下操作linux硬盘分区，实现真正的重新格式化，分区，扩容等效果。最后还需要从网上找一份新grub boot文件包,用来重建grub2.0。

1)准备工作，按照《在阿里云上自定义安装iso》的原系统是linux情况下的方法，将以上几个工具和boot文件包上传放到boot/tools下，然后tar整个根目录

```
cd /
tar -cvpzf backup.tar.gz --exclude=/backup.tar.gz --one-file-system /
```

看到打包后的大小是570m，这个就是原系统镜像。

2)然后，启动进入virtiope，利用ramdisk建立一个590m的内存盘(size=0000250 hex)。利用234extreader将/boot/tools和backup.tar.gz放进来这里的暂存盘是T：(为了操作234extreader你最好要有一个带右键菜单的键盘)，利用winpm删除整个40G分区然后分二个小ext3分区，一个10G用来作新的系统盘，其它30G用作自由空间日后作数据和镜像存放。打开colinux conf文件夹，noimage.conf中设置如下：

```
cobd0="\Device\Harddisk0\Partition1"
cofs0="..\..\..\"   (因为tools与backup.tar.gz并列放在T:中，回退3级才能看到T盘根)
保持mem=128，方便稍后的复制解压，也不能开得过大，因为1G的内存开了用得差不多了
```

现在portable_colinux.bat打开，提示enter激活busybox时，mount 2个盘到noimage colinux：

```
mount /dev/cobd0 /mnt/temp (10g盘)
mount -t cofs 0 /mnt/win (注意cofs与0中间有个空格)
```

(以上2个mnt点是colinux自带的)

3)然后，就是利用busybox中的工具：

```
cp mnt/win/backup.tar.gz mnt/temp/backup.tar.gz
chdir mnt/temp
tar -xvpzf backup.tar.gz -C / --numeric-owner 解压
```

用bootice安装新的mbr grub2.0到硬盘，从网上下载grub的boot文件包替换现有的boot文件夹（除了保留boot下原有的10个内核文件）。

4)，最后重启，进入分区调整后的linux。

如果看到新的grub2启动界面，就说明基本要完成了

```
set root=(hd0,msdos1)
linux /boot/vmlinuz-4.4.0-85-generic ro root=/dev/vda1 （注意阿里云是vda）
initrd /initrd.img
boot
```

进入新的系统，成功！！

打造**linux**和**windows**二合一的**winpe**装机维护方案

一些失败的尝试：

> 我曾尝试7zip直接解压或gnu windows tar解压到ext2sd形成的分区中，但都会蓝屏，这就是为什么我开头就说windows下处理linux分区是非原生的。大部分时间它只是辅助用一下。据说比ext2sd,ext2ifs更好的是Paragon_ExtFS之类，但是上传后无法运行，也无心去试了。不过(要是virtiope日后直接集成了ext2sd就不用这步了)这倒是另外一个极好的尝试方向.
>
> 我也曾试过diskgen是4.9.3的（4.9.3的开始支持对ext2/3的分区，它虽然比较大，但是它综合了bootice,234extreader的全部，且鼠标操作好。），跟上面一样它们甚至在xp winpe上无法运行。只有这个winpm 7 服务器版本for winpe很好支持手标操作。
>
> 我曾试过mount -t tmpfs -o size=590m tmpfs /mnt/tmp，内部fdisk,直接DD，等等，都不够直观或根本行不通。

其余，就可以不断发挥了吧。比如，前面还有《我们有第二开源操作系统吗》和《一个设想：共盘windows,linux中》，后来我们还提到了《reactos可用版》《colinux as xaas》等等，如果说这篇文章是装机领域的这类使linux,windows天然融合的努力，那么那四篇文章纯粹是是实用领域的融合windows,linux的尝试。

恩恩写够多了，就这样了

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

0pe单文件夹，grub菜单全外置版

1, 0pe与镜像定制启动的绝配

本地SRS驱动需要在镜像启动前后注入到系统的情形往往都很常见，0pe就是一个强有力的工具。因为它几乎是专门针对这个问题提出的一个整合方案。包括集成驱动和用winvblk驱动镜像部分。及注入驱动部分。

本站这方面的例子有《高可用virtio reactos svr版本发布-集成生产环境和开发环境》，和《将virtio集成slipstream到windows-isowinpe-原生方法和利用0pe》

0pe有统一PE版本，比较固化，和0penb版本，适合定制。

2，0pe纯定制版

P大的xpe/2k3pe核心的0pe即使在现在WIN nt6系列的pe核心时代，，也是有极大实用和学习意义的

其实对0pe早有耳闻，也用过，只是本人最近无事，闲来wuyou，发现0pe除了他本尊还有一变体0penb，所以兴趣上来搞了个《0penb纯粹DIY版》，，学名叫《0pe单文件夹，grub菜单全外置版》

> 0penb单文件夹，grub菜单全外置版
>
> 基于0PE_NBv1.4.3(2012-06-19)27MB_UDM.7z得到
>
> 0，将一切移到boot文件夹 1，改变grldr内容为加载\BOOT\0PE\GRUB\menu.lst（menu.lst由原menu.0pe改名而来） 2，将所有根目录下的0pe,srs移到boot 3，将所有GRUB相关的菜单外置，做成不藏入形式，并归到一个文件夹，方便了定制位置。 4，保留dos.gz到0pe core,xp文件夹改为xpe，并整个放入imgs 5，还有一些细节的改变。。。。。。

技术细节不多，难度也不大，只是自己需要这个一么绝对DIY的0PE，这样一来，也方便了需要的人。。故共享

下载地址：

http://www.shaolonglee.com/owncloud/index.php/s/dD8dm8c9FPcAUje

---

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

共享在阿里云ecs上安装自定义iso的方法

本文关键字：阿里云 自定义**iso**，阿里云 自定义镜像

应用场景：

首先，最基本的目的：你想在云主机上安装自定义iso,比如一份精简或优化过了的镜像/高版本的系统镜像，而不是运营商提供给你的那些，或你想在在云主机上安装/ghost还原镜像变得跟本地一样方便而不用总是依赖于后台备份。

还比如，你想在阿里云海外linux主机上安装windows，但又不想花一月多出来的那20多元，这就要求winpe具备从linux完全转换到windows磁盘格式和系统的功能，再在这基础上安装自定义ISO。借助winpe virtio内置的grub4dos这能轻松办到。下面详述：

第一步，将**winpe virtio**放到你已有系统中。

下载winpe virtio，如果你的原来系统和磁盘格式就是WINDOWS/ntfs系列， 1.上传peboot.rar,0pe.iso,winxpsp3.iso到你的云主机，直接将下载到的peboot.rar/boot解压到C盘；(默认系统盘为C盘讨论，这里只讨论安装winxpsp3.iso-实际它是61精简版本的win2k3，其它iso类推)

2，将c:/boot/windows/.全部复制到C盘根目录。根据4win.txt调整boot.ini内容，timeout=5要大些，比如调成50，这样vnc显示延迟能方便点到。(如果提示文件覆盖，请自行根据情况处理。一般如果你的系统是准备弃用的，全是就可以了) 把0pe.iso和，winxpsp3.iso放到c:/boot/imgs/下。

如果是linux下，按与windows相同的方式和结构解压peboot.rar到/boot/下，和复制二个镜像文件到/boot/imgs/下 因为linux通常内含grub2,所以/boot/linux/4lin.txt指出要修改的地方，会与windows下boot.ini不同。

然后就是开机。VNC进入。

第二步 **VNC**开机进入**winpe**，处理

如果你的原系统是**windows**

VNC选单，选选grub4dos->grub.exe，如果你的运营商支持通过tigervnc这种vnc进入的，比如west263，那么进入下一个单后可以直接用peboot中的安装菜单安装winxpsp3.iso。这里完成第一步，第二步，基本可以直到windows安装完毕。

但如果你是网页VNC，比如阿里云ecs，那一般如果按上一种方法第一步从iso复制完文件后，安装程序会将boot.ini改为3秒左右，下一次自动重启后等待时间过短，你一般点不到grub4dos第二步的选单出现(系统就循环自动进入第一步了，执行不了第二步，安装失败)。

这就需要进入WINPE。手动复制文件，不须用到peboot的第一，第二步。 （其实只要能进入winpe，在winpe下能看到镜像文件，之后的思路基本就很确定了，什么？还看不出来，那好，我们继续）

将winxpsp3.iso解压，然后执行i386.bat安装到C，回到文头所提，如果是其它版本的iso可以利用winpe下的ntsetup统一完成复制文件。重要的步骤来了： 将C盘已复制的系统文件中的boot.ini timeout改为50，这样重启后它就会自动找到第二步安装需要的目录了。整个安装顺利完成。

如果原系统是**linux**

如果你的是LINUX,别担心，依赖WINPE virtio版，依然可以顺利安装WINDOWS镜像。利用好wwwroot中的linux工具即可。

1.依然是vnc开机，选grub4dos->grub.exe,进入winpe后打开inetpub\wwwroot下的showdriver.exe，确认显示C盘(linux下的/)，再打开bootice，将C盘mbr弄为ntldr。pbr也是。

ps:为什么这步需要首先完成呢？因为鼠标在virtio下可能一会变得没用（原因未明），而bootice不支持快捷操作，(所幸除bootice外wwwroot其它工具都支持键盘，下面的步骤如果鼠标没用就键盘操作吧。。好吧，挺有点小麻烦。。)，所以要趁着鼠标可用的情况下先完成这步。 ps:将mbr/pbr弄为ntldr后，以下第3步复制文件的时候有20%的概率会卡死，那么整个系统就启动不了了，可能需要重来。-_-。

2.TmpRamStorage/ramdisk.exe虚拟出一个256m或512m的内存盘(我默认将设置文件改成了512m，你也可以改动)，这里即将作为临时区存放第三步中复制自C盘boot下的整个文件夹（大约2，300M） ps:说到这，要求你的云主机内存至少512m，这应该是最低配了吧。

3,rdrext23.exe，从linux分区/，复制出整个boot到第二步创建的临时盘。其中有一些grub2的大文件，可以不复制过来。

4,然后，格式化C盘，将临时盘中的boot文件弄到C盘，再按开头第一步准备文件的那些步骤弄好winpe。这样，就完全完成将linux变成ntfs和windows安装盘了。。接下来的问题，完全就是上面说过的了。

最后，设置网卡和静态路由(仅**aliyun ecs)**

最后，对于aliyun ecs,安装好的windows可能在正确配置了IP信息后不能上网。这是阿里云设有双网卡导致的特殊情况。

难点来了。如何设置静态nat 路由：

```
route delete 0.0.0.0
route add -p 0.0.0.0 mask 0.0.0.0 47.88.3.247(你的IP)
route delete 10.0.0.0
route add -p 10.0.0.0 mask 255.0.0.0 10.117.239.243（你的内网网卡网关）
route delete 100.64.0.0
route add -p 100.64.0.0 mask 255.192.0.0 10.117.239.243
route delete 172.16.0.0
route add -p 172.16.0.0 mask 255.240.0.0 10.117.239.243
route delete 10.117.232.0
route add -p 10.117.232.0 mask 255.255.248.0 0.0.0.0
route delete 47.88.0.0
route add -p 47.88.0.0 mask 255.255.252.0 0.0.0.0
```

反正我成功了,bingo!!

这步也属一个挑战了，其实你可以先备份你原来的设置，然后一条条通过route add填到这里即可。填这里的时候，始终要提醒自己的是：临时和你改为永久路由的，都要有效化（即显示在命令行route表里面）。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

使用群晖作**mineportalbox（1）**：合理且不折腾地使用群晖硬件和套件

本文关键字：群晖**,mineportalbox**

在前面《免租用云主机将mineportal2做成nas》《使用树莓派和cloudwall打造个人云》中，我们从软硬结合的角度，提出了二种代替云主机，使云设施不再受远程托管的方案，毕竟，数据在身边才是安心的，这样的一个可同步体和手边的云盒子它更符合我们一直想给它定的名字：mineportal 为什么呢？因为它是优先在内网使用的，只有在对公服务时才需要寻求内网穿透的这样的方案，所以更适合portal to public这个“portal”。----- 因为现在加了盒子的概念，所以我们进一步叫它mineportalbox。

业界早有这样的作品，比如群晖，比如智能路由器系统（可装APP的），在前面我们已讲到过群晖与各大paas的对比（默认我们指群晖的dsm），群晖是GPL开源的,其源码在sourceforge上可找到，应用也是可源的（而且还有synocommunity支持一体crosscompile构建spk的开源方案），本身并没有特殊的技术，遵循从我们bcxszy partii系列1ddcodedemobase中提到的从定制linux开始，准备容器，appstacks，apps这样的方案和路子。(它像oc塞到树莓派这样的方案，而oc没有对linux定制,sandstorm这样的东西最接近群晖dsm，但是也没有配备硬件,etc..区别仅此而已)

> synology是最好用的nas硬件(存储方面)和paas主机环境（软件方面），虽然贵但它是以好用出了名的，最流行的用法就是架网盘，较同类产品它更好用和美观。符合真正的产品级体验。 它的套件也很人性和全面，比如它的note station,编辑界面能自动消隐标题这样可真正当note用，且同时可视为blog/snippter/note工具（textallinone）。它的multimedia station(photo,audio,video)可以进行流媒体播放。webstation中的wordpress也有直接从共享中插入图片到贴子的能力，就不用再hack出一套《wordpress以owncloud为图床》这样的方案了。

可以把群晖理解为：手机伴侣，本地型云服务器（VS 远程型VPS），装机恢复器，同步器。webos，能运行php codesnippter的学习空间，甚至上升到宅男神器这样的东西。

而这一切都是以足够不折腾地的方式下去进行。不折腾原则就是让黑盒的该黑盒，不涉及到对群晖二次开发或改造，满足使用群晖现有功能和它对我们塑造的同步数据的习惯，比如最终，一个人可以不再需要PC或电脑。仅chromebook+手机+装了足够丰富应用的群晖来提升前二者的离线能力这样的三件套就可解决工作生活绝大部分问题即可 -- 这是我们的追求之一。

这就需要讲到如何不折腾地使用群晖。

1）我们能用好的方面：使用好群晖的套件和同步

我一向对云有二种要求：群晖要像cloudwall一样，可存储数据，可运行代码。这二部分可以统一同步和备份。所以我将同步文件夹分为media,softs，_current三个文件夹，media就是备份notestation,photostation,ffsyncserver里面的打包或导出的备份数据,chatlog数据，而softs就是webstation里面的代码，_current是收集每个终端来的临时同步数据，其下设各种子目录，比如frommobile，2018work，这样分别同步时指定_current/终端文件夹，就可以了，比如在家的时候仅同步_current/2018home，在公司时同步_current/2018work,从云端收集的东西放到_current/fromcloud。

这里注意不要使用drive而使用cloudstation，因为只有使用cloudstation才能获得cloudsharesync支持。而且以上用于统一备份的共享文件夹不要放到home/cloudstation中因为sharesync不适用于home，我倾向于把以上放到一个顶层的cloud共享文件夹中对应cloudstation这个名字(与那些顶层的videostation使用video共享文件夹,photostation使用photo一样，这些可作为_current之外又一外层的临时或fromcloud目录被sharesync使用同步到另一台nas)

可以双群晖同步(sharesync)，也可以对云同步(cloudsync加密备份到外部云)，后者可以免除双盘位群晖做raid的必要。当然对于个人来说是这样 - 我买的群晖就是单盘位的ds118。

同步的时候不要去动目标端。同步模式选择为仅上传。本地有同步不要进file station操作数据,其实我一直觉得同步就是个为概念，这就不得不谈到《发表cloudwall中》我们大谈特谈到的同步与云设备的关系。深刻思考这里面的关系你就会发现，永远不要依赖双向同步。仅单向备份同步。可以省却不少麻烦和版本冲突。

2)hack和强化群晖系统？我们目前还不能很好控制的地方

内网同步和nas云存储是群晖的基本作用。群晖是不完善的，对群晖进行深度定制和更改永远是疲于奔命的，业界有黑群晖这样的东西企图使之成为一个通用产品，主要是定制了群晖的kernel和启动技术，使之从不同硬件启动。我也曾解决定制群晖kernel使云硬盘从sda开始认而不是vda开始，实现过把黑群晖装到vps中。甚至考虑过是否一定需要分离式的群晖与终端设备这样的问题（比如asus有一种平板PC二合一电脑，主机是i5系统可装黑群，可分离的屏幕装的andriod可作为终端这样的组合，我还设想过一台大工作站里面有二台电脑，一台装chromebook可取出，一台装黑群固定，二者共享的主机是docker的），这些都最后觉得没有现在群晖一个黑盒子+各种终端好用。

当然在上面提到的DSM的一些固有不便的确存在，比如以上套件其数据有的不能在homes中，video不能像photo一样放到home,sharesync不支持home，考虑将顶层的video,photo做到cloud下统一sync（像moments和drive一样），webstation只持php不支持java开虚拟主机等等，这些还是值得解决的。

当然还有一件事是最迫切需要做的事情。那就是在可选和按需文件下，向外网发布服务使它更像个cloudportal和云主机：为群晖出外网和提供SS支持（这样局域网不用装SS客户端仅http代理即可），以DS118为例。下面谈到的方案适合最苛刻的内网环境（路由器甚至不能DDNS，分配的地址也是ISP的内网IP，只能穿透）：

我们可以下载arm64版的frp,在穿透云主机上装x86的服务器程序(打开群晖ssh登录设置好权限)，打开相应安全组端口，然后把arm代理端装在群晖的cloud->softs->某目录中，配置好，并在群晖控制面板中指定其开机启动:

配置文件：

```
frp
[common]
server_addr = mineportal.xxxx.com
server_port = 7000
token = 这里填服务端设置的token
[22]
type = tcp
local_ip = 127.0.0.1
local_port = 22
remote_port = 7002
[5000]
type = http
local_ip = 192.168.1.118 路由器上已固定此静态IP
local_port = 5000
custom_domains = mineportal.xxxx.com
[80]
type = http
local_ip = 192.168.1.118
local_port = 80
custom_domains = www.xxxx.com

cow
listen = http://0.0.0.0:1080
proxy = ss://aes-256-cfb:xxx@ss.xxxx.com:8000 ss.xxxx.com为ss所在服务器域名/ip
```

任务计划->开机脚本:

```
frp
cd /volume1/cloud/softs/tools/frp/
nohup ./frpc -c ./frpc_mineportal.ini &
cow
cd /volume1/cloud/softs/tools/ss/
nohup ./cow -rc ./rc &
```

穿透云主机开的安全组：

```
frp
7000:frpserver
7001:frpadmin
cow
8000
```

设置邮件通知，接收命令运行后的结果。

这样，平时走在路上手机设置quickconnect id，回到家同步会自动转为局域网同步方式。想快速访问或外网看视频就用frp方式。平时工作产生的文件也小用quickconnect足够。 群晖功率小，可以一直放心放在家里或宿舍，要租那种移动不限上传带宽的个人宽带。

之所以自《DISKBIOS2》隔了三个月才发表这么一篇文章，是因为我已经学会使用群晖和满足地使用群晖了，还是那句话，好用的东西不要再去折腾。

下一篇《使用群晖作mineportalbox（2）：利用工具编译群晖spk-以ffsync为例》，整个1ddcodeanddemobase将揭示如何一步步将一个类似群晖的东西用dbtinylinux,cloudwall,rapbian pi三者来具现的过程。与minlearn programming一起组成bcxszy

---

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

使用群晖作**mineportalbox（2）：把webstation打造成snippter空间**

本文关键字：网盘作**github**空间，网盘空间作演示空间，网盘空间作**code snippter**程序学习空间，群晖当**github**

在使用群晖作mineportalbox（1）中，我们提出了一些省事省心使用群晖的想法和经验（为资源设置合理的文件夹结构，设计单向同步的省心策略,etc..），我们还谈到：郡晖不但能用于媒体存储和同步，还可用于codesnipter hosting和同步空间，用网盘来存储数据和运行代码，“网盘即webos”，在整个系列的前面，我们不断谈到过类似概念：对于前者，文章《利用oc+wp打造backend storage oriented cms：oc静态网站空间》，《oc微博记事本》，都是企图将所有数据用文档存储来以网盘存储的方式呈现和同步的努力 ------ 网盘空间即webos fs,这相对好理解。对于后者codesnippter空间，文章《web:visual instant demo run and debugging》,《post as app，paas.engitor one as demolet engine》中的jupyter notebook即是一个好例子：它实际上把.pynb当成了服务端脚本使之成为web空间，web空间天然就是一种codesnippter空间，里面运行的codesnippter即是app,applet ------ 网盘空间即程序后端。

如此看来，codesnippter空间，这听起来像是语言后端+虚拟web空间或者baas+paas容器？甚至docker,git这样的东西，托管在github中的代码并不会执行，dockerhub呢？它更强调存储和运行单元的虚拟化和容器化，超越了仅仅需要同步这样的需求。----- 但其实空间和其中运行什么语言的程序，其实这些都不是质，我们只是追求“可存储为可同步的codesnippter空间”而已。所以一个网盘也是可以的。甚至把owncloud当git空间管理项目codesnippters也是可以的。----- 如此种种，不一而足。看似不一，其实都有相通点。而装配了webstation的群晖也可以是这种webos，它使用的就是虚拟主机概念（加上它自身就是个网盘）。

PS：backend storage oriented webapp，面向以存储为后端的webapp更符合PC的使用习惯，设想在群晖webos上存储后端即是网盘空间，存储在其上的媒体或软件媒体即软件，软件即媒体，都可以统一同步，备份，只是后者可以执行，被hosting，每一个codesnippter都可成为一个应用app，这样的空间+空间上的一份codesnippter as app，即是backend storage oriented webapp ---- 这一切都像极了PC。 还比如我们在前面提到的cloudwall，它以文档存储为FS，其上的js可以是文档也可以是代码app，可以在客户端执行也可以在服务端发挥服务端脚本的作用。所以cloudwall说它自己是webos有一定的意义。

1，如何省心使用群晖的**codesnippter**空间方面

使用群晖作mineportalbox(1)中谈到了省心使用群晖的基础方面，如果那些只是基础,那么，如何依然还能省事省心地用好群晖托管程序的这一方面呢？本文即更进一步，拟讨论稍高级的这一论题。

如（1）文所讲，使群晖能同时存储媒体托管程序才是合理的。且要能统一备份和同步。好了，下面让我们开始利用群晖的webstation（群晖目前支持的一虚拟空间语言和WEB后端），来搭建一个wordpress.

准备工作： 第一步，安装官方的php5,7,mysql5,10,webstation,apache2,wordpress(它要求mysql10),phpmyadmin等套件，将wordpress,phpmyadmin默认安装在web下，在phpmyadmin中建立数据库，把wordpress安装好，确保一切都运行起来(把wordpress地址填成frp转发后的地址，最终能进入wp)，这只是准备工作，最终我们仅需得到webstation和wordpress,phpmyadmin的源码。phpmyadmin套件,wordpress，mysql10套件要删除掉（保持仅mysql5,webstation这样清希的套件结构，是为了节省资源，也是为了体现将webstation作为上述的codesnippter空间来承载php源码集的方式，比如承载我们下述过程中提取出来的wordpress,phpmyadmin源码） 第二步，wordpress源码，phpmyadmin源码全部从web下移过来到cloud->softs->www下。（所以其实原来的安装方式也是利用虚拟主机这个原理，只是它安装到了web下，我们需要将其移到cloud下的www新目录下统一和cloud下的->media备份）。 通过phpmyadmin备份导出mysql10的数据库，导入到mysql5的数据库，备份下/var/packages/phpMyAdmin/target/synology_added/etc/servers.json，然后卸载wordpress,phpmyadmin,mysql10这3个套件，

现在准备新的虚拟主机： 第三步，把servers.json上传到www,.htaccess从wordpress中放到www根，打开webstation，选择apache2.2,php5.6新建一个虚拟主机，端口81，删除wordpress/wp-include/wp-config.php中群晖新加的东西，否则到时它会调用81端口产生资源失效，而且把数据库调为调用mysql5的pid文件。虚拟主机目录指向到cloud->softs->www ，提示转换权限，注意群晖自动处理的权限不方便，所以我们还需要自动调下，否则无法编辑后保存，也会出现no input files等显示错误。手动调权限：www文件夹自身，和递归子目录权限都定义好为用户http，权限加你当前登入的管理员用户读写全控制。 最后，在frp配置转发文件中定义一个类型为http，local端口81，转发到xxx.xxx.com，然后运行。 成功，wordpress和phpmyadmin都正常运行。

2：继续把群晖用于管理**snippter**和**code note:**

这不用我说了吧，继续新建虚拟主机，往里面写.htaccess，放.php文件直接设置即可。你可以视整个www为codesnippter空间，也可新建一个codesnippter与www并列以它为基础新建虚拟php空间。

PS: 其实我以前是不同意在群晖这样的mineportalbox上搭网站的，但现在看来，对于个人这不失为一种省事，打包带走，数据全在身边的省事方案，至少我们关站关掉电源即可。而且，我们做在mineportalbox上的网站可以仅是一个中转，比如上面提到的wp，那么做一个wordpress中转的意义在哪呢？ 比如，平时你可在群晖的notestation写文章，然后发表到这里，让它跟外面的网站同步。 还比如，普通情况下，这样的wordpress做成的新站不利于收录，但如果你写的都是原创，就可以申明原创，可以作同步到百度熊掌这样的原创平台。就不怕权重高的网站抢你原创了。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

使用群晖作**mineportalbox（3）：在阿里云上单盘安装群晖skynas**

本文关键字：将群晖安装在阿里云主机上，**skynas**系统盘和数据盘同盘。**mineportal3**硬件选型,**webpe**，提取群晖**webpe**，使用**webpe**代替**diskbios**

在前面我们讲到省事直接使用白群晖的方式，和提到一些需要hacker的地方，有些时候，正规一成不变的方式和硬件的方式有时的确是难于忍受的。从本篇开始我们探究一些群晖的奇特面，比如将其安装在云主机上的方式。比如阿里云。

阿里云上的群晖官方发布了一个skynas，我们知道群晖是靠卖软件卖硬件的，那么阿里云当硬件的情况下，它只能对镜像收费了，阿里云上的skynas是收费的，且套件支持不完全（比如非常适合当php虚拟主机管理器的webstation都没有。），且正常使用除系统盘外还需要至少一个数据盘.其里面使用的引导技术，其实跟其它群晖是一样的（白群晖硬件引导器是集成到机器ROM上的uboot，黑群则是U盘上的xpenboot之类的东西）。这种利用引导器来安装恢复系统live linuxpe,平时自身却是个dsm引导器（它有二部分，如果/dev/sda5不能引导，它会自动安装install/upgrade，因为集成好多SRS驱动所以通用性很强）的思路，就是前面我们提到的DISKBIOS的效用之一（这实际上也是我们前面追求的类苹果在线恢复系统式的PE），我们将其称为web装机系统，webpe(群晖叫法webassistant)，其一般思路就是，

在引导器下，系统dsm实际上是一个完整的linux发行版升级包。是系统也是数据，引导器会划分第一块硬盘空间，将DSM挂载到根下。这里有一段复杂的启动脚本完成挂载和升级转接（直接以tar为系统镜像进行恢复或全盘增量式升级）。分启动和安装，第一次安装系统和以后升级系统甚至引导都是同一个过程。。它仅需要这个第一块硬盘为系统盘和数据盘安装系统（对的，这里的数据盘说法上严格来说是volume1，白群和黑群的引导器可以独立于这块硬盘外置USB等方式启动，直接在onthefly环境下操作这块盘，而aliyun ecs环境特殊，引导器要事先集成安装在40G系统盘下，我们得不到一个类外置启动U盘之类的环境，引导器不能对这块系统盘直接作分区操作完成DSM安装/升级，所以一切都是事先集成好的，引导器内部的逻辑不同）。

失败的尝试，我曾想通过安装virtio类grub4dos仿真光盘的方式加载黑群引导器/skynas引导器（将其加载到纯内存），企业制造类黑白群晖的装机环境，但是不行，问题有2： 1，发现不了硬盘，黑群的xpenboot能认virtio盘，然而其将阿里云盘读成vda而非sda，引导器根本发现不了磁盘。 2，我将提取的skynas引导器做成ISO，它有认盘然而在操作硬盘分区时出现35错误，同样失败，意料之中，因为winvblk这样的东西只对windows镜像有效，是它们的驱动而非linux认识的。

如何提取skynas引导器：从这里下载它的系统http://update.synology.com/autoupdate/genRSS.php,搜索alidsm，发现DSM_SkyNAS_15254.pat并下载，7z打开，我们需要的二个引导文件是zImage和rd.gz,在ubuntu 14.04 下创建grub2可引导的iso，准备grub2的文件，将这几个文件和boot/grub文件夹放进一个文件夹假设是test，然入这里，将其做成黑群式的iso：grub-mkrescue -o test.iso test 如出现：grub-mkrescue: warning: Your xorriso doesn’t support `–grub2-boot-info’. Some features are disabled. Please use xorriso 1.2.9 or later.. 安装apt-get install xorriso

好了，因为重新编译黑群晖引导器和调整其发现硬盘的逻辑目前还没有尝试，下面来探究正确安装skynas的方式：

警告，以下过程为了学习起见，不要用于将其用于其它目的！！

准备测试环境

我们先准备测试环境(注意这个接下来的linuxpe并不是webpe，我们用webpe指syno live bootstraper)，为什么要准备这个测试辅助环境，因为webpe的ssh是进不去的我们不能直接在里面工作，我们采用从《使用virtiope安装iso》《在硬盘上安装tinycolinux as linxpe》中的方式在云主机上装上这个环境。并为这个linuxpe准备openssh和lvm支持，将其打造成实用的linuxpe版本。主要就是采用《为tinycolinux创建应用》的方式，在live tinycolinux的microcore.gz中加入这几个3.x的应用包gcc_libs,tcz,openssl-0.9.8.tcz,openssh.tcz,/ ncurses.tcz,readline.tcz,udev-lib.tcz,libdevmapper.tcz,raid-dm-2.6.33.3-tinycore.tcz,liblvm2.tcz,lvm2.tcz。并把shadow处理好，openssh运行一次配置文件也集成到这个PE中去。

好了，最终启动这个live linxpe，我们可以通过ssh进入并作lvm分区操作，如果将其做成上面的可启动ISO，这样就不再需要《利用virtiope加colinux noimage完成云主机linux的动态无损分区》这样的课题了，而我们的目的要稍微轻量一点，我们不打算利用这个PE创LVM分区，而只是复原启动器能用的磁盘结构：

现在上传2个skynas启动器文件（而并非与grub2一起做成ISO）并加入skynas启动器的启动，进入linuxpe,其启动菜单与tinycolinux livepe类似，加一条进去到/boot/grub/grub.cfg，类似

```
menuentry “skynas bootstraper webpe” –unrestricted {
set root=(hd0,msdos1)
set prefix=(hd0,msdos1)/boot/grub
linux /boot/zImage ro root=/dev/sda5 (硬性指定从sda5启动)
initrd /boot/rd.gz
boot
}
```

启动它，访问云主机ip:5000出来web assitant，我们发现它依然不能对内置第一块硬盘进行格式化，这是因为启动器只认集成了分区布局的情况（如果认到，它将不尝试分区），除非，那真是一块在启动器眼中“干净”的内置硬盘（很明显地，启动器也在这里，所以它不算干净）。

备份这二个文件和boot/grub/grub.cfg到其它区。

准备分区布局

然后，准备分区布局，进入linuxpe,fdisk /dev/vda,先键入u，由柱面计算方式换成sector，删除所有现有结构，新建下列布局（不必要求大小一一对应甚至不用格式化，其实布局对了webpe就能继续）

```
Device Boot Start End Sectors Size Id Type
/dev/sda1 * 2048 34815 32768 16M 83 Linux
/dev/sda2 34816 239615 204800 100M 83 Linux
/dev/sda3 239616 20964824 20725209 9.9G 5 Extended
/dev/sda5 239679 9676862 9437184 4.5G 83 Linux （扩展分区第一个分区号永远都是扩展分区号+1之后的计数得来）
/dev/sda6 9676926 19114109 9437184 4.5G 82 Linux swap / Solaris
```

除了把备份的webpe和启动逻辑应用到第一分区sda1，其它分区甚至不需要格式化，如上所讲，正常安装选择在线更新（你也可以上传那个DSM_SkyNAS_15254.pat），你就会发现安装过程就会继续，大小也不一定一定相同，布局相同就会继续了。完工！

你可以利用lvm把剩余空间用起来。

数据盘的问题

进入dsm，你会发现，webstation没有，套件很有限，而且最最关键的一个问题，数据盘所在的volume1，除非另加一个阿里云盘，在系统盘上，不管你在上面的布局如何新增sda7,etc..，都没有在这里被识别为volume1.

启动过程中可以看到一条：volumemamnager.cpp no target disks to be creTED AS VOLUMES AT Vdsm bootup。这尚不能确定除了内核定制，在脚本层就能改变判断volume1的逻辑，如果能做到，那么就能成功。 skynas的安装逻辑主要在，rd.gz\rd\usr\syno\sbin\installer.sh,upgrade.sh，还有/etc/synogrinst.sh,rc.volume等文件中，里面有一条Create data=no的判断。

也可以参照黑群在同一个系统分区上可创建volume1的方式来修改。只是网上有为kernel改virtio为sda的破解，有重新打包pat的破解。定制xpenboot的源码却找不到。

以后解决。

————————

接下来的文章我们要为这个skynas准备一些重要的套件了

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

利用整块化自启镜像实现黑群在单盘位实机与云主机上的安装启动

本文关键字：单盘安装黑群，黑群硬盘镜像，云主机单盘安装黑群。

在《阿里云上单盘安装skynas中》我们谈到群晖的安装方式是基于至少双盘的，新买的白群在内部一小块ROM上存有bootloader(这是第一个盘)，.pat dsm只是作为数据被安装/升级在某个volume1上的某个分区（这个volume1便是第二盘），以后进入DSM在系统中新增volume2,volume3时，与volume1相同的分区结构和分区上的数据都以相同的方式在这些volumex(x=2,3...)上复制建立一次。------- 正常的DSM安装需要双盘，这导致没有双盘的云主机或需要单盘安装黑群的实体机无法在单盘环境下安装（前者除非再购一盘否则没有可用的volume1盘，后者至少浪费一个另外的盘单独放bootloader或要外挂USB作bootloader盘）。但是，事情的转折点是，在实机上已经经过测试，安装完的黑群是有可能在安装完成后，将bootloader移到volume1的某个未划分分区上，做成单盘启动的。这里的未划分分区是群晖为了对齐硬盘在volumes中设置的一个空分区。这样就可移除最初那个bootloader盘，将整个黑群做成单盘的。

既然没有正常的纯单盘安装方法 - DSM安装脚本没有提供行之有效的方法，hacking后的结果又是可以运行的，这说明至少群晖支持单盘运行单盘启动的（这时，它已经被安装上了，整个群晖bootloader,群晖系统，和配置，及对volume1的设置，都已经被固化在这个系统上了）。那么把整个硬盘系统都img dump下来然后用在同类型机器上总是可以启动的，dmg后的硬盘镜像里会有一套区确的分区和群晖.pat安装后的数据，群晖配置，，其中bootloader是一个干净的预安装环境，（它总会是把第一个盘认识的盘作为volume1）pat所在区里的数据和配置也是无关任何机器的。猜想它总能工作起来。

这就是本文要讲的在阿里云上单盘位安装skynas及制作一个可用的硬盘镜像供以后安装。这是对《阿里云上单盘安装skynas中》的强化和补充。

1.准备硬盘镜像

建立在对群晖的etc/fstab配置和启动脚本没有任何研究的基础上，那么不妨一步一步来按照最简单的一般通用过程来还原这个过程，流程是先安装bootloader，再安装pat，再进系统，看bootloader所在盘是否会作为整个volume1被识别 ----- 按在《阿里云上单盘安装skynas中》中的方法，先准备工具，这是因为在阿里云上不能插USB，只有一个系统盘，如何在单盘环境上载入loader，可以使用iso或定制grub2，就如我们在《发布virtiope》一文一样，它在被加载后相当于一个内存盘和外置bootloader。我们现在需要得到的是一个synope，我们使用的是tinycolinux和其引导菜单：

进入工具，重新建立skynas的分区结构（这个是我们反工程skynas安装过的磁盘结构得出的结论），先安装bootloader到对应分区，然后启动上传.pat系统安装，我们以为它会将.pat放进正确的分区，结果出现了pat会提示损坏无法安装，其实，即使最终完成安装，最终，我们进入系统，也发现不了其所在loader盘可作为volume1，更别说在上面建立存储空间。----- 这里可能有二个过程会影响安装脚本对硬盘结构的判断，使得安装不能继续，1，skynas在第一次发现volume1安装.pat的过程中企图改造磁盘结构（volume1），失败，除了不能自己改造自己，它应该还有别的问题,毕竟不能期待安装脚本现在能在单盘上也像双盘上一样进行过去，2,即使完成，在建立volume1上的存储空间时又会改造一次分区和磁盘结构。

好吧，我们来看一下pat会提示损坏无法安装，磁盘到底出了什么问题，但其实这些也有人解决过了，

> 下面是nasyun上的老骥伏枥对于解决这个问题的一段话：
>
> 对此错误，我对代码做了分析，终于查出了原因。这是因为ISO盘是只读的工作模式。不同于USB盘，和硬盘启动。在导入群晖的pat包时，群晖的 update package 程序需要向它的 flash 中写入几个文件。其实这些操作对于黑裙来说是毫无意义的。但又不能不让它进行，否则就要修改它的 update package 程序。但如果群晖系统已经导入，ISO启动盘就可以正常工作。
>
> skynas与黑群dsm可能会有所不同，但道理应该一样。研究安装脚本可以得到方案。
>
> 除此之外，他在nasyun上先后推出了针对黑群自启安装的整个系列，见nasyun上的老骐伏励的系列贴子，他从原理（利用grub启动机制和各版本黑群下的硬盘分区结构，配合脚本判断插入镜像文件到闲置的分区）和实作上提出了一个synope的东西(img for hd,usb+setboot.sh或iso,其脚本原理一样)，还支持.pat的原生安装，我们不打算从自动化的角度来写二篇类似《发布virtiope》与《在阿里云利用virtiope安装ISO》的文章 for syno，因为老骥伏枥已经做得很好了。我们这里要达到的效果是：提出一套类似nasyun的124718842提出的二合一硬盘镜像为最终的产出品和安装启动方法。虽然效果不同但其思路和原理和老骥伏枥系列synope作品是一样的：利用synope工具急救环境，和脚本处理各个过程中出现的问题，即上面提到的，使安装不能继续的1，2，手动的方式比老骥伏枥发布的的自动化更安全。因为不是synope，它与老骥伏枥的安装的另一区别是，不支持老骥伏枥式的.pat安装和自动化过程。

那么在skynas下和在《阿里云上单盘安装skynas中》提到的工具支持下，如何解决pat会提示损坏无法安装的问题呢 。

我们的重点是如何在得到一个可让.pat安装过去的初始硬盘镜像。根据老骥伏枥的方案，挂载买一个新的盘或者利用GRUB2+ISO来虚拟内存盘来使那个临时IDE出现。解决了就卸载那盘。

如果这套方案已经能使黑skynas在云主机上顺利安装了，可以进入系统了，目前为止，它还是skynas眼中的bootloader盘，那么还剩下一个问题，如何将发现volume1也做进里面,同样是挂载再次新买的一个盘，进去把存储空间管理器配置好，定为raid group->basic group brtfs这样会把当前bootloader拷贝到新买的那个盘并配置好volume1指向数据。但是关于第二个问题出，重启你会发现进入不了dsm，进入synope发现，刚才的操作把然后你就会发现这次对存储空间的操作把efi引导区给破坏了，在急救pe工具中把它重新恢复回来，对新数据盘也做同样的操作，即可。重启进入。

这样在新买的那个盘中，会存有所有的能工作和存有配置的skynas系列，包括重要的volume1的设置，整个问题就解决了。我们要最终得到的正是它，反而不是系统盘。

2,dump整个镜像。

因为bootloader所在的那个盘是系统盘阿里云不让你卸载，你可以再次进synope。在这里把整个新购的那个硬盘镜像都dump下来（这个镜像是预计未来整盘恢复的，不能按分区恢复，分区还原时会改造GUID，会破坏整个已安装好的群晖关于引用盘符的逻辑）。因为是做模板的，volume1和存储空间越小越好。如果是在本机上，尽量用一个最小的盘，8G U盘都可以，在阿里云上只能作40G的快照或镜像。后者你可参照nasyun的124718842集成的《918+6.21 二合一引导启动系统盘》UEFI,DS3617xs-6.17 UEFI MBR版等等。

3,在新ECS上测试还原整个镜像

再次讲下老骐伏励系列作品与集成dsm和volume1设置的此整块化自启镜像作品的区别，------ 用此镜像还原的黑群不能重装。也不能对volume1进行添加删除volume1上存储空间的操作，原因如上所述。,那么一旦源镜像与目标盘大小不一样怎么办，目标很大，还原过去的存储空间的设置用不完所有的剩余空间，实机上124718842集成的二合一系列就会有这个问题。

124718842的硬盘镜像是针对16GU盘的，存储空间定为9G大小，假设我使用的是uefi版二合一引导启动系统盘，恢复方法是diskgenuis把目标盘所有分区都删了，然后整个镜像恢复，恢复完后进synope，发现它其实是mbr分区表，只是它有一个激活状态的efi分区，里面有bootloader，那个存储空间就是9G的linux raid，当用此镜像还原到大盘时，为了把存储空间利用完，所以我们得重新利用急救空间来hacking扩展这个存储空间。回到dsm，把固定大小的存储空间和raid group1都删了，然后再重新划分raid group1和存储空间，进入synope发现，在急救pe工具中把那个efi重新恢复回来，即可。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

Dsm as deepin mate：将skynas打造成deepin的装机运维mateos

本文关键字：**ecs**在线安装自定义镜像，通用无限制云主机**dd**安装**windows**，**Debian live os as packer**，用群晖统一打造**Time Machine**和**iCloud server**

本文是《ubuntu touch as deepin mate os,second pc os》中用dsm作为deepen mate的继续。

当mac产品之间变得越来越融合(osx10.15,ipad os,etc..)，我们抛弃它的成本就越来越高，对于我，我最不能舍弃的是它的icloud与finder无缝的方式。还有它能timemachine整盘备份/网络恢复,osx recovery在线重装os的方式。

所以我一直在寻找脱离osx生态和替换以上产品的方式，当然，要以不牺牲体验为前提。比如，它可以是一个类unix系统，如日益变好的deepin和一些第三方的cloud sync产品。见《ubuntu touch as deepin mate os,second pc os》。

可deepin刚好缺少这几大功能。它的recovery os是传统windows PE式的不支持网络安装。它又没有全盘镜像。deepin dde也没有集成cloud,要寻找另外的方案。

那么有没有在网络环境中通用的安装iso的手段呢？网络安装系统的一些实践和通用技术是什么呢。netboot?pxe?tftp?ghost network?

那么用户数据重装恢复呢。这基本可想象是现今常见的cloud服务+客户端同步，对于linux的桌面，有没有无需客户端的cloud mounter呢且能与explorer尽可能融合的方案存在呢？甚至，它能与系统重装统一吗？

好了不深究它了。关于系统重装,运维，这绝不是一个小课题，以前我们集中于单点和实机，现在我们有云和分布式节点频繁重装和异地灾备这样的需求,还有更多未来需求 —— 我们将这一切上升到一个OS，这一切我们称为mirror os,mate os。

我们一步一步来。依然是采用skynas和deepin作例子，我们首先要将skynas打造成deepin的装机运维mateos。

网络重装系统，与用户数据恢复，整盘快照增量备份恢复

在前面《共享在阿里云ecs上安装自定义iso的方法》我们讲到在ecs这种异于实机的装机环境中换ISO，那时我们利用virtiope在linux ecs中操作硬盘释放系统安装。大部分都是手动过程。利用的是windows pe这种图形GUI的中间维护系统。

现在我们有linuxpe和自动化的方案，moeclub在《从零开始:在 Linux VPS 上覆盖安装WINDOWS通用教程》中用Debian live os描述了手动利用DD备份和恢复镜像的方式，如以下：

```
dd if=/dev/sda | gzip > vdaimg.gz
wget -qO- http://xxx/vdaimg.gz |gunzip -dc | dd of=/dev/vda
```

除此之外，moeclub还并提出一个整合性的脚本installnet.sh，并附有修正ip,远程桌面等的作用。它可以dd方式恢复windows镜像(你可以想象osx pe的在线安装系统)，也可以以安装方式在当前云主机安装常见的linux（你可以想象为syno pe的webassit局域网安装pat），使用的是debian livecd，它发挥二个作用，1，livecd可以将驱动注射到维护系统，Debian netboot mini.iso https://wiki.debian.org/DebianInstaller/NetbootFirmware https://www.debian.org/devel/debian-installer/这相当于在线生成virtiope的自动版，2，它类packer,可以为即将安装的系统按模板生成目标实体。这二个完成之后，它会全盘DD或安装。

这个Debian liveos+moeclub的脚本即提出了一种通用网络安装/恢复的方法。当然它还没有自动/手动备份和恢复用户数据的方式。系统的更新和系统配置数据，，应用数据，用户数据往整盘的增加，即镜像的多版本，都要求timemachine支持增量，

在osx中，timemachine 也是有局限的，它只备份osx所在区。至于用户数据恢复，其实timemachine在完成一次恢复之后，icloud数据是默认被排除不恢复的。在osx中，系统和用户数据分开备份。是因为apple认为timemachine数据在本地时间胶囊之类的东西上局域网云，icloud永远在远程云上，用混合备份的方式。

而现在我们统一了系统和数据都在远程。趁现在它没有像timemachine那种持续增量DD和恢复的方式，我们可以改进一下，把timemachine发展为cloud share sync的方式。用户数据和系统数据彻底分开。只要求备份某个有数据变动的分区。

比如，从用户名开始的以后部分全部是backup的。彻底分离的usr/system设计,system为readonly，这其实是安卓类手机系统的一惯做法。可以是一个pe.这也是类群晖系统更新的关键所在。updateable的部分不触动webassit部分。Mac osx 10.15之后也是将系统作为只读镜像,tinycorelinux也有cloud模式，所以以后，osx timemachine估计不会备份系统除非单盘方式。

好了。以上是问题的高级部分。回过头来：

下面我们来讲解使用它在阿里云安装skynas的方法。再来讲集成dsm的cloudsync服务到本地的方式。moeclub的InstallNET.sh脚本针对windows镜像的，才有改IP的效果。如果是skynas这种，需要手动去改。或事先在镜像中做好。或通过定制脚本来实现。不过skynas是可以自动配置IP的。

在阿里云上网络安装skynas镜像和搭建类timemachine的服务器

在《使用群晖作mineportalbox（3）：在阿里云上单盘安装群晖skynas》和《利用整块化自启镜像实现黑群在单盘位实机与云主机上的安装启动》中我们讨论了手动抠出skynas镜像并单盘安装的方式。那么现在我们将使用上述InstallNET.sh来自动恢复它。这三篇文章中都有类似linuxPE，脚本这样的组合手段和相似思路。

所以我们这里直接DD出skynas镜像并不需要按上二文处理，你可以按需买一个双20g盘阿里云（这里仅作为测试，数据盘可以买大一些，系统盘和数据盘都不要使用ssd），vda用skynas615-15254初始化在初始化过程中一旦出现已运行，在控制台立马快速强行停止云主机，然后作快照（使用阿里云自己的导入镜像没用它是没有分区表的），恢复这个快照到数据盘vdb，然后重置vda为ubuntu，在其中dd并gzip vdb，将得到的文件scp上传到可以下载的地方，然后利用installnet.sh将其恢复到任何ECS的vda上./installnet.sh -dd 'http://xxx.com/img.gz'，如果有VNC可以看到Starting the partitioner 会停留很久,就是在执行上面二句，一定要等它完成。(其实你照样在这里可以得到系统盘+数据盘单盘安装skynas方式，提示一下：把数据库作为临时系统盘启动，让系统盘中的镜像临时认数据盘为vda，然后卸载数据盘。)。

以上或许还支持将skynas安装到其它服务商的ecs上。

好了。skynas支持TC和cloud station。然后找到一个叫cloudmounter的软件，cloudmounter支持将dsm webdav 方式的cloud托管服务集成到osx的finder，还支持linux, windows，并将缓存的文件放入一个可选的本地位置，当然，它跟提供了finder sync api的osx finder的可定制性还是不能比。不过它跟原生的iCloud in finder最接近了。

———————

下次我们将把群晖打造成一个uniform 运维，使用，开发者统一的os设计。配合浏览器录屏把群晖打造成收藏os，能收集所有类型文档的，聚合os,,像以前的tumblr一样，所有的网上视频等，不必经过下载。收藏视频的方法是录制，像note crapper一样，并将让tinycorelinux支持netboot和网络安装。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

Dsm as deepin mate(2)：在阿里云上真正实现单盘安装运行skynas

本文关键字：单盘群晖，本验证码版黑群

现在是2019新冠疫情时期，记得关于sars的描述的一句话吗，这种病在早期人类历史上，可以杀死一个大洲的人都不止，所以，对这种病有解的医生，等同于神。

在《使用群晖作mineportalbox》文3和《利用整块化自启镜像实现黑群在单盘位实机与云主机上的安装启动》，及《dsm as deepin mate》文1中，我们着重提到使skynas在云主机上单机安装使用的方法，这些文章中，研究点经历了从“黑群webassit，从0开始安装的方法”到“从镜像开始，能启动就行”的路径变化，但这二者我们都没有完全完成，达成最终目的。下面继续这个课题，实现最终在阿里云上单盘安装运行skynas。

制造并导出镜像的方法

我们重新开始，首先去阿里云开一台系统盘20G，数据盘20G，镜像为skynas615-15254的按量机器，等进去了（因为我们不尝试导出纯净可安装镜像，所以这里不必跟以前一样等一启动就急停，做快照…，或者使用阿里云最新的卸载系统盘功能,恢复系统盘初始状态不重启…，其实这个时候导出的镜像也没用，会失效，稍后会讲到），新建一个admin，密码自设，这个admin会是管理员。我们发现应用只有系统区的二个,filestation和universal search，没有关系，打开控制面板，外部访问中的ssh，我们可以ssh admin@yourip从命令行用admin登录，然后sudo synouser --setpw root xxx，然后exit或su root切换到root。

其它的应用去http://synology.cn/zh-cn/support/download/VirtualDSM#packages复制其spk地址。通过命令行cd到/tmp，wget这个spk，通过synopkg install xxx.spk来安装，这种方式下你需要自己探寻app的依赖关系。启动失败一般是依赖没安好。

但是我们这里并不急着安装APP，因为我们优先考虑尽早把系统设置保存下来，这些将来用于作为模板初始化新volume，——— 系统启动，安装APP会在数据盘volume1中产生文件，分析产生的文件有volume1/@appstore,volume1/@databases,/tmp，volume1/@tmp，这些文件与APP与系统本身都绑定，如果不打包打走，等volume1被释放，这些APP或系统本身，在新volume上启动，运行都会产生设定丢失或权限隐患等问题。比如tmp中包含有系统关于volume1的空间设定（这个设定会让没有任何volume的系统在启动时误认为volume1损毁，但这正是我们需要的,在以前的文章中，我们那个tmpdata,tmproot只是安装/升级时解包的逻辑。并不是进入系统后判断用作数据盘的逻辑。），相应的，@appstore应该就是app的数据。—— 为了不使这个模板文件过大，也是为了不带入因集成APP产生的不确定因素，我们不必事先在这个20G的数据盘volume上装太多APP，后面在增强部分会讲到APP。

> 我们打包这些初始设置文件并替换volume所在盘： /volume1/@database/pgsql(它实际上/var/services/pgsql,app和系统都会使用它) /volume1/@tmp（它实际上是/var/services/tmp,app和系统都会使用它） /tmp（这个目录极为重要，系统使用） cd到/，tar cvpzf template.gz tmp volume1/@tmp volume1/@databases，带原用户权限打包，这样这个模板文件就有了。

事实上，在这里我们就可以把镜像导出来了。因为此时群晖是视数据盘为volume1的，我们盘和模板都有了，所以将在下一次重启时永久卸载这个volume1，让它用上新volume就行了。

来测试一下：1，制造新volume。因为群晖中有parted，比起fdisk它是即时生效的，所以sudo parted /dev/sda mkpart primary 4 100%，然后格式化sudo mkfs.btrfs /dev/sda4，完成。2，sudo reboot 重启，在阿里云ECS的存储管理中，卸载volume1，重启进来，登录web端。空间管理器提示没有任何存储，这是意料之中的，那么没有什么也没有提示空间损毁呢，这是因为tmp是每次启动都会清空的，保存在其中的空间信息会丢掉。3，我们把template.gz恢复到tmp，web端存储管理器立刻提示空间损毁，这是与原来数据盘volume1的挂钩的tmp/space数据，这个时候我们只需要把/dev/sda4挂载到/volume1就可以了。我们在命令行中尝试root登录，sudo mount /dev/sda4 /volume1挂载，web端已显示存储空间可用，这里其实格式sda4为Ext4也可以直接挂载。

然后你就可以导出这个自动镜像为别的机器所用。导出时可以在线nc导出(需配合virtio tinycorelinux，因为群晖没有nc也不能在其中直接nc):1，dd if=/dev/vda | gzip -c | nc -v -l -p 5000 , 恢复处用nc -v xxx.xxx.xxx.xxx(这里最好使用阿里云内网地址) 5000 | gzip -dc | dd of=/dev/vda，也可以2:保存为一个本地gz的方式然后使用ossutil+oss内部endpoint地址导出或者scp xx.gz root@xxx.com:/var/www上传到一个远程空间，oss最快。

恢复到不同的机器时，需要手动产生sda4及重复上述测试过程（使用非custom linux镜像如果是custom linux，需要配合virtio tinycorelinux修改/boot/aliyun_image/os.conf），1，wget -qO- http://xxx/vdaimg.gz（配合oss内网endpoint和virtio tinycorelinux）| dd of=/dev/vda 或者2用InstallNET.sh -dd，重新测试前，记得把网络改了，你需要改，etc/sysconfig/network，etc/sysconfig/network-scripts，/etc/resolv.conf，并增加一条gateway：route add default gw xxx.xxx.xxx.xxx。

增强：在镜像中直接封装sda4 as volume1，及提前封装app

事实上，我们可以对镜像进行一些强化，因为上述过程在每一次换机或重启都要进行一次，为了追求制造一个自动的镜像。我们需要自动产生坏盘和自动挂载sda4，但又不能直接把这个sda4封装进去(因为这样在不同机器上不能自动扩展空间)，所以还需要一些内容

> 自动产生损毁盘。自动产生并挂载sda4 要echo 自动挂载规则到fstab。这就需要绕开syno的自动cfgen，重命名/usr/syno/cfgen/s00_synocheckfstab to k00_synocheckfstab这样etc/fstab就不会改了。 如果不存在sda4,sudo parted /dev/sda mkpart primary 4 100%，然后格式化sudo mkfs.btrfs /dev/sda4 然后sudo vi /etc/rc,在开头加上： echo “/dev/sdb4 /volume1 btrfs

> nospace_cache,synoacl,relatime 0 0” > /etc/fstab mount -a 如果不存在/tmp/space,volume1/@databases,volume1/@tmp，则tar zxvf /template.gz -C / 如果是ext4，直接在/etc/rc开头挂载mount /dev/sda4 /volume1，即可,mount -a不用加

这是代码：

```
mkdir -p /tmp/space
cat >/tmp/space/space_mapping.xml<<EOF
<?xml version="1.0" encoding="UTF-8"?>
<spaces>
    <space path="/dev/sda4" reference="/volume1" uuid="/dev/sda4" device_type="3" drive_type="0" container_type="2" limited_ra
idgroup_num="0" space_id="" >
        <device>
        </device>
        <reference>
            <volume path="/volume1" dev_path="/dev/sda4" uuid="/dev/vda6" type="ext4">
            </volume>
        </reference>
    </space>
</spaces>
EOF
```

进一步封装App，你可以手动将APP移到系统区：先停止app，这里以TextEdtior为例，先sudo synopkg stop TextEditor，然后sudo mv /volume1/@appstore/TextEditor /usr/local/packages/@appstore，再然后sudo ln -sfn /usr/local/packages/@appstore/TextEditor /var/packages/TextEditor/target

但最好是直接安装app到volume1做app封装，因为app启动与volume1/@tmp,volume1/@databases相关,直接转移到系统分区的APP绝大多数不依赖有没有一个volume1，但有些会要求有volume1，这时即使有原来的volume1/@tmp,@databases，启动也会出错。另外注意，把所有的依赖和核心安装全：

> 这里我总结了spk的依赖规律。 一些语言类的： PHP5.6-x86_64-5.6.40-0059.spk(webstation需要，nginx是系统自带的) PHP7.0-x86_64-7.0.33-0028.spk(photostation需要) Perl-x86_64-5.24.0-0074.spk,PythonModule-x86_64-0114.spk(perl和pymodule cardavserver需要,py2是系统内置的)，Node.js_v8-x86_64-8.9.4-0005.spk(这个就是chat server缺失的v8,calendar也需要) PHP7.2-x86_64-7.2.24-0006.spk(这个calendar也需要) 和基础类的： SynologyApplicationService-x86_64-1.6.2-0431.spk(calendar也需要)的都是最好要安装的。 然后就是各种其它APP了。

本地验证版

还记得开头说的“其实这个时候导出的镜像也没用，会失效”吗？上一步导出的镜像有可能失效。在另外的机器上启动时会出现仅加载到webassit的情况，这应该是群晖的验证过程。但是我测试时从一台新阿里云+skynas的按量组合产生的镜像，在24小时之内是不会的。所以要赶在你做好镜像的24小时内”食用“这个镜像，很沮丧是吗?我们可以把黑群弄成纯粹的本地验证。

验证的过程肯定存在于rd.gz，rd.gz相当于usb pe(它会把绑定的网卡或sn与官方比对？异或是上面的tmp?)。而且我发现，普通群晖与skynas的rd.gz中的逻辑脚本组织等，都很接近。skynas的rd.gz并没有经过深度定制。只是没找到vda变sda的地方。是否可以直接相互置换制造云上的黑群？测试安装普通黑群到阿里云，

但是安装好了的，绝对没事(这分二个过程,验证的过程永远在sda1中，而不是sda5中)，不然，安装好了的syno会重启后boot不动，这是毁灭性的。没有出现过黑群被删volume1。

> 一些本质上的问题,任何黑群晖都不是绝对的本地验证版
>
> 就像微软假sn一样，你固然可以在淘宝上买到，但如果涉及到使用某些在线服务时（比如黑苹果之于icloud，黑群晖之于quickconnect，windows之于office，共同的在线升级等），注册在官方的库中，还是可以被辨别并删掉中止服务的，一切只>> 是官方出于某些目的并没有去针对反破解，这些反破解措施中有没有包括删掉你的数据不可得知。技术上可以绕过机器和本地，但是你没法绕过远程和人 — 除非你满足于使用封闭的本地服务，所以，任何使用破解OS这类大件都是有风险的，黑群也是。

——

读到这里的小伙伴都赢了。让我在这里看到你的头像

接下来还会有一篇文章。Dsm as deepin mate(3)：更安心全面地使用syno as icloud,我们将讲解app中那些关于个人办公及日常使用的portal,erp性质的app（email,cardav,calendar,etc..），探讨其可以无缝代替iCloud的方式，如何利用cloudstation+cloudsync作三地异地，探讨其同步算法与原理。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

Dsm as deepin mate(3):离线编辑初始镜像，让skynas本地验证启动安装/升级

地球变得越来越危险了，20年前地球大部分地区可以用荒凉来形容，而现在每一寸都被开发，疫情，自然灾害频发，如果不及时止损，估计100年都用不了，人类只需要20年就能从气候和生态上毁灭地球

本文关键字：啥是真正的黑群，压缩**skynas**磁盘布局为**5G**内

在《dsm as Deepin mate(2)》中，我们讲到了使skynas镜像脱离aliyun ecs真正能运行起来的方法。但是我们用的是倒叙的手法，我们还没到得到安装镜像，这里继续。讲解离线创建和编辑纯净镜像，使之能像普通黑群一样启动，完成安装/升级的方法。

Ps:啥是真正的黑群

———— 所谓黑群晖,是由于群晖基于linux gpl要求开源了早期内核和toolchain在sourceforge：http://sourceforge.net/projects/dsgpl/files/，而它的rd.gz,pat升级/安装包是很容易解包打包的https://github.com/andy928/synochecksum。因此。各种黑群技术有机会修改zImage,rd.gz和重新打包定制pat。你可以在这里定制内核插入你的驱动，作各种脚本增强，使群晖适用于更多机型，甚至通用硬盘。 故，其实正真被黑的部分并不是bootloader。正真被黑的应该是部分内核zImage与位于initrd ramdisk中的各种外挂模块。至于系统中的安装包/升级包中的正常具体逻辑，是不需要破解的。虽然如此，大部分黑群版本命名大都是xxxboot之类。 具体技术方面，（1）破解的第一关是rd.gz中的synobios：群晖是通过一个叫做synobios.ko的外挂模块集中处理硬件绑定问题的。一旦破解了它,很容易解除硬件绑定关系。所谓的破解，主要就是改一下这个接口的代码，使他在黑群晖的硬件上也能返回一个合法的机型代码（例如DS3612xs的代码是42）。因此，大家现在所安装的黑群晖，实际上synobios.ko文件已经修改过了，镜像pat也用的是重新打包过的pat，synochecksum-emu1 hda1.tgz rd.gz updater VERSION zImage >checksum.syno这里有一个对pat内文件验证打包的过程。(2)破解的第二关，群晖在sourcefourge的开源内核驱动仅支持官方，本身也是有点错误的，需要一定定制，加入自己的驱动，直接编译后在引导系统的时候会出现很多错误，例如加载synobios时候出现找不到符号的错误、synoacl实现并没有完全开放等等。因此，需要对synology开放的内核代码做一些修改，目前网上能够找到的对内核的修改只有旧内核的(3.2.11等）。加入自己的驱动，等等。（3）还有一些边角的破解或增强，如修改u盘vid和pid,填写正确的U盘 VID和PID可以在DSM界面隐藏U盘，修改sn和mac地址可以用于"洗白"。可见sn破解是无关紧要的，除非你要用上官方的服务如quickconnect,ffmpeg缩略生成等（见前文后面，本地验证版黑群，官方只是没有对这些假sn开刀而已）。 http://xpenology.com/wiki/en/building_xpenology可以说是一篇最详细和权威的资料（编译内核的内容，还包括了synobios的破解和rd.gz的制作，etc..），不过它是是针对4.1的老版本的。不过其用于指导黑群制作的思路是不变的。 视上面二方面，加入了不同驱动，作了不同增强，因此有很多黑群版本，虽然定制synobios是不可避免的，但有的版本，如gnoboot支持使用官方的PAT文件（依然是xpenology的思路），gnoboot编写了一组脚本，巧妙的实现了在rd.gz引导系统后自动替换需要破解的文件，不仅如此，gnoboot还包含了大量的驱动，并且可以很方便的加载和卸载驱动。 ————

问题回到出发点，对于skynas,破解目标是什么?对应上面提到的(1)（2）（3）有哪些地方需破解呢？第一个问题，不经过破解直接运行的结果，就是用你提取的安装镜像换机安装时，不会启动sda5中的系统安装无法继续，而是进入web assit重装界面（即用当初收费镜像机dd出来的机器镜像，换机后也有绑定过期验证不通过退回webassit重装界面），这个绑定在哪？是不是验证？这就是第二个问题。

作初步猜想，可能在grub的那个checksum（zimage,rd.gz,update.pat/hda1.tgz三个都是阿里云绑定的因为其内都有aliyun_xxx相关的脚本文件,不排除是这个原因导致的安装验证不通过。rd.gz/synobios应该不需要破解因为都是阿里云机。那么那个checksum呢？），也有可能在sn绑定（因为skynas是个在线机器,有先sn网络验证后安装的条件，不过群晖并不常用sn验证安装过程,物理机上的黑群boot通过了就通过了）。

无论如何，skynas用的并不是严格普通黑群的那套(不是synobios过了，pat就会过)。而是只有zimage,rd.gz,update.pat三个都过了，才过，所以破解会是（1），（3）我们需要测试验证，下面会讲到。

镜像及镜像离线编辑测试环境

我们先来尝试建立一个纯净镜像，去阿里云开一台系统盘为20g的skynas615-15254(选择可卸载)按量机器作测试环境（为什么不在本地虚拟机测试呢？因为我们还没有精力去考虑因换本地可能带来的其它情况），开机后关机，恢复系统盘初始状态为skynas615-15254选择不重启，建立快照。 —— 这样我们就得到了一个初始skynas61515254的快照。重置系统为ubuntu18，并启动，为只有一个系统盘的系统建立一个20G的数据区，建立时恢复skynas快照并mount。

反转数据区与系统区启动逻辑(为什么要反转呢，因为我们未来要在系统区安装skynas运行测试镜像，启动到数据区上安装的ubunt18里离线编辑，grub1,2可以互通)，在ubt18中初步编辑skynas/vda/boot/grub.cfg建立启动到第二硬盘的菜单，像这样：

timeout由15改为1500,如果这个不改，从vnc重启，如果过快，会默认启动到skynas。 加一个菜单boot ubt set root=(hd1,msdos1) 这里是vdb中的ubt,从grub1 chainload grub2，反之需要insmod ext2 linux /initrd initrd /vmlinux boot

删掉原来快照重新建立数据区的新快照 ——— 这样我们得到一个初步修正了的skynas的快照。同时，系统区也做快照。—— 这样我们得到了一个ubuntu18的快照。

现在，使用阿里云最新的功能卸载系统盘和数据盘，为没有任何盘的系统重新建立一个20G的系统和50G的数据（这里为什么先前要卸载系统盘呢，因为不是新建的盘不可以从快照中恢复，我们也不想使用镜像功能因为那会涉及到计费），建立时恢复新skynas的镜像，如上所述，我们将在这个ubunt18里离线编辑(为什么ubuntu18里面不同时装我们以前文章中处处使用到的tinycorelinuxpevirtio呢因为tcpe毕竟有些工具版本有限不

太方便)，在skynas盘运行测试镜像（为什么不在数据盘里运行skynas呢，因为skynas kernel的root必须是sda系列，而ubt怎么都可以），这样我们就得到一个离线编辑运行镜像的测试环境了。

vnc重启，进入skynas的菜单，选择启动到vdb上的ubuntu,在ubunt里，dd(+ status=progress可看进度)数据区到系统区root下形成img.gz文件。得到一体化灌装好的一个初始镜像615-15254.gz，这也是阿里云初始sky nas 镜像用的方式，(并非用任意pat通过webassit直装)，mount /vdb5可以看到它是如下这样组织的，像这样：

.SynoUpgradeIndexdb.txz .SynoUpgradePackager 猜出现在package center的是丢在pat/package升级包中提取出来的 .SynoUpgradeSynohdpackImg.tcz .NormalShutdown ….. 然后是hda1.tgz释放到整个sda5中的内容

其实不嫌麻烦，你可以利用前几篇文章中的方法或重新建立分区，研究手动释放DSM_SkyNAS_15254.pat升级包到vdb5得到如上组织的方式。

绕过验证逻辑

这样三者(kernel,rd.gz,pat)都有了，synobios略过，来看grub中的checksum，它相当于物理机中pat中的checksum，checksum是check rd.gz+zimage的。如果修改rd.gz，checksum会failed，但它会不会影响rd.gz中的逻辑呢。如果验证逻辑主要在rd.gz中呢？（这个在gui桌面环境下用7z就能打开查看,里面有大量的可能验证逻辑），不论如何，先验证一下修改rd.gz使checksum失效吧，看会有什么影响。

我们首先测试来为rd.gz中的root去掉内置默认密码，以证实我们的想法和利于接下来的分析，:etc/passwd去掉第一对::中的x,/etc/shadow去掉第一对::中的*，如下作重新打包即可：

(我们现在在/root，mount /dev/vdb2 mnt/vdb2, cd mnt/vdb2,vdb2空间足够，可以直接在这里修改和重打包,首先把rd.gz改名为rd.gz.xz,不然接下来xz解压不成功)

```
(解压)
mkdir rd.dir
cd rd.dir
xz -d -c -k ../rd.gz.xz  | cpio -id

(压缩)
cd rd.dir
find . | cpio -o -H newc | xz -9 --format=lzma > ../rd.gz
```

（测试1） 重启测试新的rd.gz,即boot sda2启动菜单中去掉checksum和vendor。此时进去/sda5/etc/会发现大部分rc都是加密的。原来checksum只够影响这里。

（测试2） 重启测试新的rd.gz,即boot sda2启动菜单中保留checksum和vendor。此时进去/sda5/etc/会发现rc都是正常的，只是显示checksum处显示failed，而且linuxrc.syno failed on 4。但此时系统依然无法启动。，启动后。依然如意料只会停留在web assit界面。

看来checksum并不是最终验证或唯一的过程。使得这个升级过程不能继续。我们需要探索更多rd.gz中的地方。接下来就是初步分析整个rd.gz，去得到绕过验证的逻辑，然后动态修改rd.gz，并用新的rd.gz用在系统区中启动。一点一点测试，这是我们的新思路。

我们结合dmesg,从linuxrc.syno来一步一步分析。这里的流程是rd.gz/linuxrc.syno(rd.subr)->/etc/synogrinst.sh(rd.gz/usr/syno/sbin/installer.sh,update.sh)->sda5/usr/syno/etc.default/aliyun_sn_init.sh,sn_generater在第一次启动后（无论成功或失败）会自删掉

我们从上面找到了，从rd.gz到sda5 privot root的地方(rd.gz/linuxrc.syno，rd.gz/etc/rc中的mount /dev/root /)。

貌似在upgrade输出，准备升级开始的地方。 如果找不到/dev/root就找不到/tmp/root等 如果找不到分区，就找不到/dev/sda 找不到/dev/sda，针对sda就全盘cfgen不了，就导致linuxrc.syno启动失败(upgrade过程) 如果upgrade不了，就形成不了整个有效可启动的/dev/sda5

正是这里导致了linuxrc.syno failed on 4，最终就只能进入rd.gz

因此，所有的问题都变成，如何触发 让rd.gz过程找到/dev/root(就会启动/dev/sda,事实上绕过了验证)

（最终成功测试） 解发能找到分区的关键是，安装过程中要有一个/dev/sdb，跟前文一个思路和技术。

(最后来说那个sn)

其实那个序列号只是系统内一个叫sn_generater的本地工具随机生成的。有没有那个sn没关系,系统可以以空sn安装启动 那个aliyun_sn_init.sh和aliyun_sn_generater是临时生成的。

封装和后期离线编辑过程

最后是封装：dd到ubut分区中,及后期离线编辑：

```
 (没有kpartx, 我们可以用loop)
losetup -f
losetup -o 1048576 /dev/loop0 skynasinit
```

```
mount /dev/loop0 /mnt/volume1
losetup -o 17825792 /dev/loop1 skynasinit
mount /dev/loop1 /mnt/volume2
losetup -o 122715648 /dev/loop2 skynasinit
mount /dev/loop2 /mnt/volume5
umount /volume5losetup -d /dev/loop2
上面-o 之后的数据就是skynas的四个分区开始乘于512得到的

(如果volume5 umount不了,我们可以找出占用volume1的在执行进程并终止它们)
lsof +f -- /volume5
Kill -s 9 pid
```

———————

dsm的web后台技术做得很流畅很轻量不费资源，还有webassit在线安装升级是syno的二个特色（从这里我们可以分析得到其边分区边升级、安装的技术）。结合前文《Dsm as deepin mate(2)：在阿里云上真正实现单盘安装运行skynas》，甚至你还可以压缩skynas磁盘布局为5G内，这样镜像做出来也会更小,集成更多内置spk和一个叫move_syno_pkgs.sh的脚本，etc…..前文遗留问题：有一些spk如vide,photo只能在有单独hd as volume的情况下安装,把sdb1改成sda4

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一键pebuilder，实现云主机在线装dsm61715284

虽然群晖上云没有什么很大意义，因为云主机附属的带宽和存储成本现在还没有做到类似onedrive不限速，几百元1T一年的程度。类似cos,oss这种有api的”类od网盘“成本也偏高实用性也不大，群晖这种OS也没有完全做到以网络硬盘为后端存储，它倒是支持一种分布式硬盘的iSCSI只不过这种分布式是大架构用的海量存储方案。在应用方面，群晖大量采用开源技术，也有一些自研的应用比如xxstation组合，但是cloudstation的sync算法经常会导致冲突，但是anyway，群晖的webstation是个很好的php虚拟主机管理面板，也可以作为网盘间内容转存器，因为它有cloudsync(不过百度盘最近又限制了cloudsync的配额，其一惯作风~)。所以还是来折腾一下：

技术背景

1,aiminick在nasyun上发布http://www.nasyun.com/thread-31988-1-1.html的《DS3615xs直升DSM6.1.4U2【KVM平台虚拟机】加强版引导镜像》：作者结合利用了jun大引导和老骥伏枥二合一引导（这二个是平等作用的东西，前者有外挂驱动完成了黑群的主要部分，后者主要是多合一，可以用jun大引导中的三个文件extra.lzma、 rd.gz 、 zImage替换老骥伏枥/boot/grub/DS3615xs 下同名文件，主要是extra.lzma，这个文件以沙盒补丁的形式对黑群文件系统进行patching），重新编译了syno内核和virtio modules替换了相关文件,在本地kvm上完成测试了最新版黑群614系列pat的安装，(除此之外他还更新了一篇3615xs 6.1.7的kvm安装，除更新了版本支持值得一提的是这个版本去掉了老骥伏枥引导中的grub1,tinycorelinuxpe等，仅保留了grub2部分)。

这证明kvm黑群（类似《在阿里云上单盘安装群晖skynas》和《dsm as deepin mate》系列针对的skynas。但纯粹的kvm黑群方案相对更原生。）也是可以做出来的（虽然接下来我们会谈到，使用aiminick的包在kvm上折腾还是有人碰到很多坑），但是aiminick并没有解决sda->vda的问题，使用的virtio ide方案。云主机所属的kvm使用virtio blk或scsi。这个问题也在第一篇黑群文章中就提到了。

2,作者tiandishi在nasyun上发布http://www.nasyun.com/thread-70722-1-1.html的《群晖上云服务器,完成度90%！！》，搞定了上述问题，并进一步修复了阵列问题，得出了最新版单盘安装黑群的方法并介绍在了另外一个贴子里。

tiandishi没有直接采用aiminick的包在kvm上折腾，他仅采用aiminick的选型和编译方案（615机型和linux-3.10.x.txz内核），倾向使用原始的老骥伏枥DS3615xs黑群晖 6.1-15152版中的文件制作启动，却使用了jun大ds3615xs 6.1.7引导来做，仅最小替换virtioblk来测试，他重新编译了能识别vda为sda的dsm6.1内核和驱动模块，他修改了 linux-3.10.x.txz 中源码的drivers/block/virtio_blk.c,修改 virtblk_name_format("vd", index, vblk->disk->disk_name, DISK_NAME_LEN); 这一行的vd为sd。先从virtio ide+虚拟机下（proxmox kvm下）开始，按正常IDE盘格式和折腾aiminick的3617的包在kvm上安装617pat并进该硬盘的系统，不断打包进新引导包替换测试，如果失败则回退到原处再次测试，最终发现新的引导可以产生与aiminick测试类似的结果：安装 3615xs, 6.17,15284 版本的系统至完结.虽然至少此时正确能识别硬盘为sda，但这也是新问题的开始，因为他导致了不能重启后正常使用（值得一提的是在tiandishi的包中，他保留了其中tinycorelinuxpe，因为它日后可以用来修改grub中的mac）：

1）现象：启动时发现无法识别群晖的/dev/md0 /dev/md1 软raid系统,导致认定系统未安装,重新进入安装界面.作者分析 2）原因：发现这是因为单盘系统盘是一个软raid1的软阵列，黑裙引导的时候需要识别这个阵列并加载阵列的系统。但当硬盘类型为virtio_blk的时候，内核启动检测阵列使用的 mdadm --auto-detect 这个命令不会映射virtio_blk到阵列专用的设备符/dev/md0,进一步导致内核执行/usr/syno/bin/synocheckpartition 的时候返回结果为Partition Version=0，报错Partition layout is not DiskStation style.;需要替换为 mdadm --assemble --scan ,这样可以正常识别阵列分区, 使得/usr/syno/bin/synocheckpartition 返回正常结果 Partition Version=8. 临时解决。 3）方法：去掉virtio盘的虚拟化blk格式，按正常IDE盘格式和折腾aiminick的3617的包在kvm上安装617pat并进该硬盘的系统，仅为得到一次磁盘信息，正常安装下头二个区一个2G一个2.4区为软raid区，这相当于在系统内mdadm --detail --scan > mdadm.conf，获得的关于镜像盘的如下md/blkid相关信息，

```
ARRAY /dev/md0 metadata=0.90 UUID=57d437b5:798159fc:3017a5a8:c86610be   你需要手动修改uuid为你的盘的uuid
ARRAY /dev/md1 metadata=0.90 UUID=7580a324:5f709df7:dcd287d5:29e3aff2
ARRAY /dev/md2 metadata=1.2 name=yunnas:2 UUID=b96e00a8:dab9a2b6:c99e9210:f50c9dbf (这个盘是数据盘请略过)
```

这个mdadm.conf将被放进extra.lzma/juno/etc/中作为解决新系统不能重启运行的补丁,手动喂给booter调用：修改linuxrc.syno的补丁文件jun.patch ，在内核挂载阵列设备 /dev/md0 前执行 mdadm --assemble --scan，该命令优先读取上面的配置文件，识别出硬盘阵列，然后后面的挂载系统能顺利执行。

```
mdadm -S /dev/md0 && mdadm --assemble --scan
 echo "Mounting ${RootDevice} ${Mnt}" 放这句上面
```

打包extra.lzma ，扔进引导，重新启用virtio盘的虚拟化blk格式。测试成功进入系统！还有二个后期小问题：1，启动后其它数据盘部分无法识别，需要添加一个临时数据盘组raid（这个可结合我《Dsm as deepin mate(2)：在阿里云上真正实现单盘安装运行skynas》处理tmp/space.xml的方法尝试解决），2，启动后添加的网卡拿不到IP，需要修改grub引导中mac的地址。但都非关键问题了

下面直接制作镜像，用tiandishi的路子,找对版本,先搞定识别sda安装完系统，再搞定阵列修复进入系统：

制作镜像

准备工作：我们直接使用jun大ds3615xs 6.1.7引导(注意这混乱的版本命名，3615是硬件平台，617是内核，15284等是pat)和原始的老骥伏枥DS3615xs黑群晖 6.1-15152版中的文件（用jun大617替换里面的原615引导的相关文件，/boot/grub/ds6135xs就不改成ds6137xs了以避免需另改grub.cfg），和tiandishi弄好的http://down.nasyun.com/forum/202005/17/064451tfwjwlvwawylwwgj.zip?_upd=kvmdrive.zip,q外面的那个virt_blk.ko仅仅适用于3.10.102内核编译的系统。前者是引导，后者是补丁，

制造下面的617kvmboot：

```
qemu-img create -f raw 617kvmboot 32M

fdisk 615kvmboot自动dos mbr，n自动一区，a自动激活
losetup -fP --show 617kvmboot
mkfs.vfat  /dev/loop0p1
mkdir bootmnt
mount /dev/loop0p1 bootmnt/
grub-install --boot-directory=bootmnt/boot /dev/loop0

...(复制boot所需文件进来，不要把各种tcpe,legacy grub复进来，，先kvmdrive.zip中重命名extra.lzma中的etc/madam.conf(deepin下可以直接编辑名字自动gui打包)，以防其它非意料因素出现，用extra.lzma直接替换617kvmboot中的extra.lzma即可，另外修改grub.cfg中virtio mac地址一定要525400开头否则一会识别不到。有调试需求，则可以在grub.cfg中将虚拟串口设置为开机不自动启用。)..

umount bootmnt
losetup -d /dev/loop0
```

替换后的镜像也都是mbr虚拟机/云主机适用的，依然是我们前面的在pd中装deepin，开kvm，再启动qemu制造主镜像，二步走，第一步用qemu-system-x86_64 制造镜像，第二步用qemu-system-x86_64 测试启动镜像，如果不分二步，直接上来virtioblk，会提示格式化失败

第一步:

```
qemu-img create -f raw dsm61715284 20G

qemu-system-x86_64 -enable-kvm \
 -machine pc-i440fx-2.8 \
 -cpu Penryn,kvm=off,vendor=GenuineIntel \
 -m 2120 \
 -device cirrus-vga,bus=pci.0,addr=0x2 \
 -usb -device usb-kbd -device usb-mouse \
 -hdc ./617kvmboot \
 -hda ./dsm61715284 \
 -device virtio-net-pci,bus=pci.0,addr=0x3,mac='52:54:00:c9:18:27',netdev=MacNET \
 -netdev bridge,id=MacNET,br=virbr0,"helper=/usr/lib/qemu/qemu-bridge-helper" \
 -boot order=dc,menu=on
```

这一步会出现网上很多人拿DS3617xs_DSM6.1_Broadwell.img在kvm下折腾都出过一些问题，新的booter也会有，如下：

引导进入dsm，安装时局域网扫不出ip，除了可能开头就提到的你grub.cfg中virtio mac地址没有改成525400开头否则识别不到,如果dsm已识别网卡，只是扫不出而已。也可以这么做：从宿主机看guest分到的ip(arp -a)，或者找个脚本扫virbr0所在的网段了（192.168.122.2到254）。我这里得出是192.168.112.32或78，以后几乎固定都是这个不用再扫，如果还是得不出ip，重启宿主deepin再来(我发现aiminick的网卡识别比较好和效果稳定)，deepin中用chrome打开，在webassit上显示对应的ip也不是上面指定的mac，被改变了，如果连不上重启虚拟机重来，连上了webassit，下载pat,准备手动安装的https://cndl.synology.cn/download/DSM/release/6.1.7/15284/DSM_DS3617xs_15284.pat。安装的时候提示格式化 1 2 硬盘，勾选安装就可以了，引导盘不会被格式化，如果pat版本没匹配，会出现各种安装到57%左右文件毁损，

如果没有问题，系统安装完成，进去命令行mdadm --detail --scan > mdadm.conf,检查输出(本来设计对镜像作blkid和tune2fs -U写入uuid，但那个2.4g的是个swap fs，无法这样完成，故还是修改mdadm.conf)，生成的文件重新整合到xtra.lzma，则可以继续第二步，启动测试。

第二步:

```
qemu-system-x86_64 -enable-kvm \
 -machine pc-i440fx-2.8 \
 -cpu Penryn,kvm=off,vendor=GenuineIntel \
 -m 2120 \
 -device cirrus-vga,bus=pci.0,addr=0x2 \
 -usb -device usb-kbd -device usb-mouse \
 -hdc ./617kvmboot \
 -device virtio-blk-pci,bus=pci.0,addr=0x5,drive=MacHDD \
 -drive id=MacHDD,if=none,cache=writeback,format=raw,file=./dsm61715284 \
 -device virtio-net-pci,bus=pci.0,addr=0x3,mac='52:54:00:c9:18:27',netdev=MacNET \
 -netdev bridge,id=MacNET,br=virbr0,"helper=/usr/lib/qemu/qemu-bridge-helper" \
 -boot order=dc,menu=on
```

最后，第三个分区20G-4.5G开起来。且开第四个分区用于在主镜像中制造硬盘引导为日后脱离617kvmboot也能引导，即并losetup挂载将617kvmboot中的内容以及引导再次离线写入到生成的镜像的第四分区，tar cvpzf它，并将最终镜像上传到onedrive互联。

完工！

利用**pebuilder**在云主机上安装

由于pebuilder.sh足够智能就不用保留tcpe对于mac的替换，况且这需要在具体目标系统中完成。比如放在pebuilder完成dd后要重启的位置，你可以直接将处理mac的逻辑放这里：(grub.cfg中可以写function，为什么在那里不做？一定要在进入系统后？)，以至于这里也可能可以跟处理mac一样放处理tune2fs的逻辑：

```
d-i partman/early_command string $PIPECMDSTR; \
sed -i 's/mac1=52540066761B/mac1=yourmacaddr/g' \$(list-devices disk |head -n1)4/boot/grub/grub.cfg \
sbin/reboot
```

要看pebuilder.sh gif演示的。去我的github，下文，用开源包替换cloudstation和用videosstation云转码

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

普化群晖将其改造成正常磁盘布局及编译源码打开**kernel message**

本文关键字：群晖显卡支持，**dsm 6 console tty,**打开**console**，让**3.x linux**使用**graphical terminal as console**，改变黑群原生磁盘布局，一种动态调整压缩镜像文件的方法。压缩**0**空间，及在**linux**上离线操作**raid/llvm**分区的方法，离线折腾黑群镜像,调整黑群剩余空间为存储空间。**initrd**加载二个**gz,dd**会复制**uuid**吗,**debian**上**grub-install**会自带**device-map**导致**grub rescue**

在《Dsm as deepin mate(3):离线编辑初始镜像，让skynas本地验证启动安装/升级》中我们讲过离线编辑镜像的需要，但是黑群是一种使用了软raid的系统（在云上，软raid是没有意义的。因为云盘可靠性很高）。它的镜像是一个多区段镜像有三个raid，主启动分区倒不是raid，然而被放在末尾而且很小只有32MB（nasyun老骥伏枥正是因为发现这个32m可以变动一下发明了它的脚本，稍后我们会贴出其镜像布局）另外内置在镜像中的存储空间也未能被认到，由于这个镜像的系统里并不推荐也不能重建存储分区，（本来就是一次性镜像），那么无妨将其纠正为正常的布局，把启动前变1G并提前，顺便将整个镜像当初定的50G缩小至一个较小的尺寸比如20G更为适用，识别/重建可用存储空间的工作也可以一起给做了，

> 以上我们正是为了把原生群晖改造成正常的linux系统的方式,方便以后集成PE和通用booter。实际上群晖是专用硬件附属系统，但因为它用了linux，普化群晖为linux发行也变得可能，比如还有人在群晖上加入其它系统移殖过来的binary，运行sshd,telnetd，甚至整个debain chroot fs等，我们本文一部分也正是这样的路子，我们先为群晖开tty（dsm61启动时停在booting the kernel，但serial口还在正常输出，群晖的serial output我们当然能在虚拟机上加个--serial或redirect to a file能试出来。我们尝试把它还原为直接在console上输出，类普通linux kernel）以便接下来调试，然后再变更原生磁盘格局。

建立本地测试环境**local pve img create/edit/hosting and restore enviroment**

我们将之前的《一键pebuilder，在云主机上安装deepin20，云主机在线安装dsm61715284》创建/编辑/下载镜像的方案，现在我们转到pve上来，我是在osx->pd上安装pve6.3上的（定义一个虚拟机打开nested虚拟化），实际上这货应该安装在baremetal上来（与exsi一样）。----- pve使用了我最喜欢的debian与js界面(只是这货安装时没有使用di有点遗憾)，而且它支持很精细的cpu类型，这就可以为我们之前《kvm上安装osx》的时候无法确定cpu类型提供帮助，

> 如何将以前的创建/编辑/下载方案弄到单点pve上来。而且做到一一对应 切换使用localstroage来存储镜像： pve安装完后创建vm存放disk images的位置是默认的local lvm存储，这是一种虚拟逻辑存储，为了测试我们希望镜像直接存放在filesystem中，供我们dump用。 使用pvesm set local --content images或在web gui 的serverview->data center->storage的local上那里双击为local增加一个images options(这个默认是没有打开的)，这个打开的情况下，就可以从local storage中make disk image了，否则只有一个local lvm供你选。除此之外，你也可以删掉local lvm改挂载逻辑为local，将local lvm改为local，这个不推荐因为lvm存储在真正部署时还是有优势的。 只是这个界面不能像上传iso一样上传diskimage file，你可以使用qm (Qemu/KVM Virtual Machine Manager)pvesm (“Proxmox VE Storage Manager”)等工具通过命令行操作上传/转换镜像。 控制镜像细节：1,定义一个20G的磁盘，在定义虚拟机时填入20就可以了,2,或者进pve的命令行(ssh root@pve ip)，在里面打命令制造特殊布局的raw image。3,上传已有raw image，那个lxc就是raw image的上传处，不要怀疑,模板iso可以直接在web界面上传，也可wget到这里 /var/lib/vz/template/iso/ ，在定义虚拟机时就能看到了 如何获得pve中经过编辑的dd镜像,还是我们在ssh pve中的python -m SimpleHTTPServer大法(停虚拟机先)。 另外一个问题就是这种虚拟机中套虚拟机的方式，使得磁盘性能大大降低，没办法。忍受一下只是制作镜像用，另外据说pve维护一个仓库，可用普通debian安装这里的源，自己造一个pve，不过我们关注app级的虚拟化，并把它集成到os（造nativecloudstack），而不是os的虚拟

除了这个pve用来建镜像并servering，我们还得一个用来dibuilder.sh安装前一个VM restore做出来的镜像的虚拟机，测试dibuilder.sh的有效性（实际上pve上测试镜像通过就通过了我们这里只是测试dibuilder：利用《将pebuilder变成dibuilder.sh,将di tools集入boot层(2)》中的dibuilder.sh支持raw url as dd image src，可以在其中直接dibuilder -dd 'local result img'），用任意linux即可

一个不大不小的问题：我们发现在debian jessie 8.11上创建初始布局镜像 时发现：debian grub-install（apt search grub-pc，显示grub2-common/oldoldstable,oldoldstable,now 2.02~beta2-22+deb8u1 amd64 [installed,automatic]）与前面《利用增强tinycorelinux remaster tool打造你的硬盘镜像及一种让tinycorelinux变成Debian install体的设想》tinycorelinux 11上的grub-install表现不一,虽然最终grub-install都是no errors，但是debian上做的启不动，提示no such device,只认到hd0,ls没有分区。可能是debian上的grub集成的grub/i386-pc里面的脚手架文件不全，也可能是已完成安装的grub2在grub-install到镜像文件上时会带当时系统上自身device map，debian下的vi也有问题退格删除傻傻不分，但是有nano editor代替可用，故在一台tc11上而不是debian 811中，完成后的镜像我们上传到pve，

为了可以循环重来，我一般是在PD虚拟机中新建上面二台vm的当前快照，这样快，失败了也可以迅速恢复重来，

还原**console**

为什么把这个放在前面，因为它可以使控制台观察群晖的输出变可能，作调试用。------ 在之前的文章中我们多次提到busybox init对tty的处理(1，我们主要是创建/dev/console(实际的终端）和/dev/tty0(主虚拟终端）设备文件,实际上tty0这个文件动态生成可以不要手刨，否则为什么叫虚拟终端，2，然后在/etc/inittab文件中添加如下实现：ttyS0::respawn:/sbin/getty -L ttyS0 115200 vt100 # GENERIC_SERIAL(用vt100类型模拟):这里面的说法是，init使用生成（respawn）一个守护tty的getty过程，负责登录开一个终端（注意getty只是开终端不负责内核输出）你登录时会看到了login。最后 grub cmdline中append一条"console=tty0 console=ttyS0,115200")将必须的参数传给内核，完成内核信息开机启动所有的过程和登录接入，包括《avatt2编译》及最近的《将pebuilder变成dibuilder.sh,将di tools集入boot层(1)》中，我们都处理过。

我们来综合介绍一下：

> TTY 是 Teletype 或 Teletypewriter 的缩写，原来是指电传打字机，为裸机作输入输出交互服务，后来在出现了OS之后，这种设备逐渐被键盘和显示器取代。虽然现在物理终端实际上已经灭绝了，但它们都由硬件概念，逐渐演化成了os里软件模拟的终端概念(比如用graphic console/tty模拟text console或physical serial console,这实际上也是一种必要，比如我们需要ssh终端这种东西)，------ 其实，不管是电传打字机还是键盘显示器，还是软件模拟的显示设备，都是作为计算机的终端设备存在的起输出输入交互作用，所以 TTY 也泛指计算机的终端(terminal)设备。 提到终端就不能不提控制台 console。控制台的概念与终端含义非常相近，其实现在我们经常用它们表示相同的东西(比如显示器键盘是终端也是控制台，但在计算机的早期时代，控制台指大型控制设备终端仅限输入输出设备，基本上，控制台是计算机的基本设备，，而终端是附加设备，但它们都有输出信息的终端能力，比如系统启动时的消息只能首先输出到第一个控制台终端上，你也可以接入其它终端来定位这些信息)。所以说法上，有终端作控制台也有控制台as终端的说法。各倾向于指主要输出设备和次要输出设备的意思。 终端和控制台可能会有多种形式，，比如，除了上面提到虽然SSH,Telnet 等等作为p终端实现了对服务器的管理通过远程访问进行，还有比如模拟的物理串行s终端，ssh出来的p终端(用户级的)，本地显卡显示器出来的虚拟终端（os视角下的终端，(我们把它称为c终端，这也是一种kernel级tty)），他们各有各的用处对应各种场景，比如有些时候有些情况下，使用远程访问是无法解决全部问题的，如处理一些导致系统crash的错误或者bug的信息，这些需要通过Serial console来访问，它们需要模拟成软件层的虚拟终端接入OS。所以我们最终面对的是OS视角下的各种终端， linux说法下的专用虚拟控制台终端term=linux,有一些设备特殊文件与之相关联：tty0、tty1、tty2等。当用户在控制台上alt+f1-6切换时，使用的文件始终是/dev/console，或者你启动时没有设定 console= option (或者使用console=/dev/tty0),此时的/dev/console就是/dev/tty0，------ To use a serial port as console you need to compile the support into your kernel - by default it is not compiled in. For PC style serial ports it's the config option next to menu option: Character devices ‣ Serial drivers ‣ 8250/16550 and compatible serial support ‣ Console on 8250/16550 and compatible serial port，You must compile serial support into the kernel and not as a module. ------to use a graphic framebuffer console,你可以关注这些内核配置项，VGA text console,Support for frame buffer devices (EXPERIMENTAL),VESA VGA graphics console编译完成后，你的内核就支持framebuffer驱动了
>
> 对于init，有sysvinit和upstart这种，linux有0-7几种启动级别，在sysvinit中，/Linuxrc 执行init 进程初始化文件。主要工作是把已安装根文件系统中的/etc 安装为ramfs，并拷贝/mnt/etc/目录下所有文件到/etc，这里存放系统启动后的许多特殊文件；接着Linuxrc 重新构建文件分配表inittab；之后执行系统初始化进程/sbin/init。/mnt/etc/init.d/rcS 完成各个文件系统的 mount，再执行/usr/etc/rc.local；通过rcS 单用户脚本可以调用 dhcp 程序配置网络。rcS 执行完了以后，init 就会在一个 console 上，按照 inittab 的指示开一个 shell，或者是开 getty + login，这样用户就会看到提示输入用户名的 提示符(getty就是守护登录而已一个作用，外带开启终端)。/usr/etc/rc.local 这是被init.d/rcS 文件调用执行的特殊文件，与Linux 系统硬件平台相关，如安装核心模块、进行网络配置、运行应用程序、启动图形界面等。/usr/etc/profile rc.local 首先执行该文件配置应用程序需要的环境变量等。 但是很可能会发现在您的计算机上找不到/etc/inittab 文件了，这是因为它是sysvinit的，现在的新发行版要么使用一种被称为 upstart 的新型 init 系统要么使用systemd。sysvinit的缺点是串行同步的，这大大影响效率，UpStart 解决了之前提到的 sysvinit 的缺点。对于设备接入导致的复杂init变动等这些业务需求都可以满足，它采用事件驱动模型,工作配置文件存放在/etc/init 下面，是以.conf 作为文件后缀的文件。 目前不仅桌面系统 Ubuntu 采用了 UpStart，甚至企业级服务器级的 RHEL 也默认采用 UpStart 来替换 sysvinit 作为 init 系统。此外云计算等新的 Server 端技术也往往需要单个设备可以更加快速地启动。

我们的参考：

先去sf下载源码，怎么选呢？黑群5的XPEnoboot mod是可以直接像普通linux一样显boot message的，我们要修改的是61，6之前最通用的版是5.2，因此比较这二个版本：

> 它的源码分类是这样的(按硬件软件发布组合命名)， toolchain下是dsm xx版本/cpu架构+linux版本+cpu型号的文件夹。比如我们选择dsm6.1/Intel x86 linux 3.10.102 (Bromolow)，在里面下载toolchain,https://sourceforge.net/projects/dsgpl/files/Tool%20Chain/DSM%206.1%20Tool%20Chains/Intel%20x86%20linux%203.10.102%20%28Bromolow%29/bromolow-gcc493_glibc220_linaro_x86_64-GPL.txz/download nas source是具体版本分支号/cpu型号，比如我们选择15152branch/bromolow-source，在里面下载kernel source，https://sourceforge.net/projects/dsgpl/files/Synology%20NAS%20GPL%20Source/15152branch/bromolow-source/linux-3.10.x.txz/download 群晖的系统（硬件与采用的linux版本，只有一个版本号如ds3615xs）与硬件系统（系统磁盘上的rootfs，有二个版本号如5.2,6.1，和15152,5644）分离跨度很大是分开开发的，比如很多硬件系统还停留在引用3.x的linux版本上，兼容很稳定不冒进。

编译61源码，以下来自https://xpenology.com/forum/topic/7341-tutorial-compile-xpenology-drivers-in-windows-10/，在debian系上是一样的操作：

```
apt-get install mc make gcc build-essential kernel-wedge libncurses5 libncurses5-dev libelf-dev binutils-dev kexec-tools makedumpfile fakeroot lzma
sudo sh -c "echo alias dsm6make=\'make ARCH=x86_64 CROSS_COMPILE=/root/x86_64-pc-linux-gnu/bin/x86_64-pc-linux-gnu-\' > /etc/profile"
source /etc/profile
cp synoconfigs/bromolow .config
dsm6make menuconfig

# 所幸，发现群晖把他们对linux的增强的地方都放在meunconfig enhancement section跟linux主体分开的，在serial driver栏可以看到tts0,ttys1是被保留使用的而且swap了,另外这些项也要关注一下，CONFIG_FRAMEBUFFER_CONSOLE,SYNO_X86_TTY_CONSOLE_OUTPUT,CONFIG_TTY_PRINTK,CONFIG_EARLY_PRINTK，调整这些参数，如果这个命令发现不了，可能你用了sudo)
# 在群晖中，上面提到的serial tty都是打开的，由于群晖这种东西早期是没有视频和视频驱动支持的，所以它只能用串行s终端作console来输出信息。或者ssh出来的p网络终端。而不倾向于使用本地显卡显示器framebuffer出来的控制台tty终端。所以一般linux都会有默认的kernel级tty支持和显驱支持，群晖内核配置中也当然有相关的选项。

dsm6make(这个时候千w不能想当然make否则调用的不是cross compile)
```

```
arch/x86/boot/bzImage

#出来的结果就可以用来置换内核了，大家都是x86 ds3615xs出来的内核+jun mod所以可以互换，测试时用任何rootfs as initrd arg即可，其实单纯一个linux k
ernel就可以启动计算机无须initrd，
```

发现61不带dmesg输出，同理下载5.2-braswell-gcc473_glibc217_x86_64-GPL.txz，5644-braswell-source.txz：这个由1.6G的重打包而来，编译5.2的源码(5.2碰到locale.c assert错误执行export LANG=C)并调整，发现也是不带dmesg输出的。看来52能输出是xpe mod的修改导致的：

于是去看52 iso(其实不必，在linux kernel启动就可以看到它支不支持console output dmesg，跟rootfs无关，我这里只是列举细节)，

由于xpenology的XPEnoboot_DS3615xs_5.2-5644.4.iso是将initramfs打包在内的(它用的lilo，而grub2和linux内核有tty驱动/mod支持，比如在grub2中也可设置fb分辨率)，难于解包，所以直接装一个进里面查看里面的文件，我们可能还要准备
https://global.download.synology.com/download/DSM/release/5.2/5644/DSM_DS3615xs_5644.pat，得出观察结果：

群晖5使用upstart(在/etc/init/tty.conf中发现了exec tty，各处conf中发现了console output，dev/下发现了正常的/dev/tty*,ttyS，tty显示/dev/console,export 看到term=linuxecho dsfdsf>>/dev/console出现在当前，启动中发现ttyS0 enabled字样, synobios open /dev/ttyS1 success)，,dsm6版本之后使用用了一套很精简的init机制比如synoservice -restart ssh-shell,来启动ssh,6没有telinit,upstart和tty.conf，6 ps有getty，但6貌似在crontab启动它？

> https://smallhacks.wordpress.com/2012/04/17/working-with-synology-hardware-devsynobios-and-devttys1/这篇文章提到synobios和/dev/ttyS1是群晖内置二驱动二硬件，比如ttyS1用于响应面板上的按压操作。还提到它们与闭源的scemd的关系。 顺带去了xpenology的坛子，junmod这个折腾之源的出处(它的原理是，在grub cmdline中initrd一次加载二个rd.gz和extra.lzma，这样这二个就二合为一了。跟群晖menuconfig一样，在单独的patch中放置修改跟主体分开。）发现quicknick的mod ： XPEnology.Configuration.Tool.v3.0.Bootloader.DSM.6.1.x.zip，它似乎提到enable tty方面的东西。 https://xpenology.com/forum/topic/6566-xpenology-configuration-tool-bootloader-dsm-602-8451/ 这个家伙把文件藏好深。 img里第4分区有一个.quicknick（包含quicknick.lzma）隐藏，xz quicknick.lzma还有一个.quicknick，这里是它主要定制的地方。他们俩的东西基本在这你都可以下到https://xpenology.club/downloads/

针对修改我们的dsm62尝试改动：

```
#修改extra.lzma中的init
#以下放在bin/patch后
/bin/sed -i '/SYNOLoadIPv6()/i\\TERM=linux' /etc/rc
/bin/sed -i '/SYNOLoadIPv6()/i\\export TERM' /etc/rc
#启动服务直接守护在console
/bin/sed -i '/SYNOLoadIPv6()/i\\/sbin\/getty 115200 console &' /etc/rc
#以下放在exec /sbin/init之后作测试
echo "test" >> /dev/console
echo "test" >> /dev/console

#修改grub.cfg中的函数
function showtips {


terminal --timeout=3 console serial

    terminal_input --append console
    terminal_output --append console

    if [ -n $has_serial ]; then
        terminal_output --remove serial
    fi
    echo "Screen will stop updating shortly, please open http://find.synology.com to continue."
   .....
    echo
    if [ -n $has_serial ]; then
        terminal_output --append serial
    fi
}
```

失败，mark一下，以后解决！

新建一个镜像文件重建磁盘布局,重建p1,还原p2,p3内容(按扇区dd)

进入我们的目标讨论：将启动分区前提并扩大，缩容存储分区和整个镜像，有二种路子：除了按照《一键pebuilder，实现云主机在线装dsm61715284》从0开始产生定制尺寸的镜像，也可以直接针对现有镜像作缩小镜像并压缩0空间处理作出目标镜像（但是linux上的磁盘扩展压缩没有在windows下方便和工具支持多，如果不是当初设计为lvm镜像来的，后期的resizefs之类的程序操作并不保险。parted也支持用它分区出来的镜像的动态扩展但局限也多不是一种好方案），除了前面二种，实际上也可以在原来的镜像上重建分区格式然后DD得到，这种方式最简单直接而且干净，也是本文要讲述的方案。

我们新建一个20G的空白镜像，（其实可以把原镜像重分区缩为20G(不需要的部分不分区进mbr表，或删掉不让它们进入分区表)。然后直接dd，不用在新镜像上进行。）但为了过程更清楚化，我们还是利用新建镜像然后格式化的方式。我们用的是parted而不是fdisk，仅用fdisk -l,你如果在fdisk中n手动创建，可以创建4再创建3，这样可以跨过3自由定制4的大小。。原有50G镜像格式为：

```
sudo fdisk -l ./dsm61715284
/dev/loop0p1         2048   4982527               4980480  2.4G fd Linux raid autodetect
/dev/loop0p2      4982528   9176831             4194304    2G fd Linux raid autodetect
/dev/loop0p3      9437184 104652799           95215616 45.4G fd Linux raid autodetect
/dev/loop0p4 *    9177088   9242624             65537   32M  e W95 FAT16 (LBA)
```

我们看到很不美观，我们将启动分区提前为扩展为1G，方便塞入dipe和更多东西，其它分区不变，猜想这虽然不是黑群的原生布局，但猜肯定能启动和正常工作

准备三个脚本create.sh,mount.sh和umount.sh：

这是create.sh,part one:

```
# 脚本用的tce-load -iw parted dosfstools util-linux, losetup要换成/usr/local/sbin/losetup, home/tc而不是/home/td)。
#sudo apt-get install parted dosfstools

sudo dd if=/dev/zero of=dsm61715284new bs=1024 count=20971520
dev_dsm61715284new=`sudo losetup -fP --show ./dsm61715284new | awk '{print $1}'`
dev_dsm61715284ori=`sudo losetup -fP --show ./dsm61715284 | awk '{print $1}'`

# 分区1, 分配2097152个扇区=1G, 为了保持以后的分区对齐
sudo parted -s "$dev_dsm61715284new" mktable msdos
sudo parted -s "$dev_dsm61715284new" mkpart primary fat16 2048s 2099199s
sudo parted -s "$dev_dsm61715284new" set 1 boot on
# 从这里开始的是dsm的磁盘布局(跳xxx, 跨xxx)
# 分区2, 从上个分区的最后一个簇+2048+1作为开始（跳2048）, +4980480-1作为结束(跨4980480)
sudo parted -s "$dev_dsm61715284new" mkpart primary 2101248s 7081727s
sudo parted -s "$dev_dsm61715284new" set 2 raid
# 这里有一个没对齐warning, 没关系
# 分区3, 从上个分区的最后一个簇+1作为开始(不跳), +4194304-1作为结束(跨4194304)
sudo parted -s "$dev_dsm61715284new" mkpart primary 7081728s 11276031s
sudo parted -s "$dev_dsm61715284new" set 3 raid
# 分区4, 从上个分区的最后一个簇+260352+1作为开始(跳260352), 100%剩余空间作为结束(跨至尾-1)
sudo parted -s "$dev_dsm61715284new" mkpart primary 11536384s 100%
sudo parted -s "$dev_dsm61715284new" set 4 raid

# 我们看到结果是正确的, mkfs有一个小warning没关系
sudo fdisk -l "$dev_dsm61715284new"
sudo mkfs.msdos "$dev_dsm61715284new"p1
```

安装grub2复制boot文件开始，sudo apt-get install curl binutils，bintuils是为了运行dibuilder.sh时能有ar 然后wget,chmod并运行dibuilder.sh -dd 'http://d.shalol.com/mirrors/dsm61715284.gz'，在完成等待10秒时ctrl+c，在/boot下得到initrd.img和vmlinuz,开始安装grub2

脚本part2:

```
#sudo apt-get install curl binutils
#sudo wget -cq http://10.211.55.2:8000/dibuilder2.sh 镜像中用的 http://10.211.55.2:8000/mirrors/debian/
#dibuilder.sh -dd 'http://d.shalol.com/mirrors/dsm61715284.gz'

mnt_dsm61715284new="/home/td/newp1"
mnt_dsm61715284ori="/home/td/orip4"
sudo mkdir -p "$mnt_dsm61715284new" "$mnt_dsm61715284ori"
sudo mount "$dev_dsm61715284new"p1 "$mnt_dsm61715284new"
sudo mount "$dev_dsm61715284"p4 "$mnt_dsm61715284ori"

sudo grub-install --boot-directory="$mnt_dsm61715284new"/boot "$dev_dsm61715284new"
sudo mkdir -p "$mnt_dsm61715284new"/boot/DI
sudo cp /boot/vmlinuz "$mnt_dsm61715284new"/boot/DI
sudo cp /boot/initrd.img "$mnt_dsm61715284new"/boot/DI

#然后你就可以继续往grub中增加东西,原来老骥伏枥那个是grub2的啊, 一直以为是grub1的
sudo cp -r "$mnt_dsm61715284ori"p4/boot/grub/grub.cfg "$mnt_dsm61715284new"p1/boot/grub/grub.cfg
sudo cp -r "$mnt_dsm61715284ori"p4/boot/grub/DS3615xs "$mnt_dsm61715284new"p1/boot/
sudo cp -r "$mnt_dsm61715284ori"p4/boot/grub/themes "$mnt_dsm61715284new"p1/boot/
#sudo vi grub.cfg,把timeout改为50, 把调用ds3615xs的地方也修正下, 三条有意义的set ds3615xs img的menuentry最上面加个set root=(hd0,msdos1)
, 把判断tcpe7.2处的$cmdpath改为cmdpath, 把调用$cmdpath的地方改为($cmdpath), 改为调用/DI/xxx

sudo dd if="$dev_dsm61715284"p1 of="$dev_dsm61715284new"p2
sudo dd if="$dev_dsm61715284"p2 of="$dev_dsm61715284new"p3
#不能直接sudo mkfs.ext4 "$dev_dsm61715284new"p4, 要等到变成md2时格
#sudo mkfs.ext4 "$dev_dsm61715284new"p4
sudo umount "$mnt_dsm61715284new"
```

```
sudo losetup -d "$dev_dsm61715284new"
```

以上p2-p3按扇区复制，p1也处理了只有p3没有处理(大小不一不能dd)，我们重新还原这个布局里的文件,sudo apt-get install mdadm:

这里是mount.sh，调用方法 mount.sh 镜像文件名

```
# 以下参照《https://www.howtoforge.com/how-to-set-up-software-raid1-on-a-running-system-incl-grub2-configuration-ubuntu-10.04-p2
》
# 从这里开始弃用dsm61715284ori和create.sh
[ $1 = dsm61715284new ] && dev_dsm61715284new=`sudo losetup -fP --show ./dsm61715284new | awk '{print $1}'`
[ $1 = dsm61715284 ] && dev_dsm61715284ori=`sudo losetup -fP --show ./dsm61715284 | awk '{print $1}'`
sudo mkdir p1ori p3ori p2new p4new

# 群晖会把每个新加的盘的第一第二分区都作为md0/md1成员加进去作raid
[ -b /dev/md0 ] ||  echo 'y' | sudo mdadm --create /dev/md0 --metadata=0.90 --uuid=5d279172:1ed20bf2:3017a5a8:c86610be --level
=1 --raid-disks=2 missing "$dev_dsm61715284new"p2
sudo mdadm --detail /dev/md0 | grep "UUID"
# 如果dd过，那么下面一句可省
# mkfs.ext4 /dev/md0
sudo mount -t ext4 /dev/md0 p2new

[ -b /dev/md1 ] || echo 'y' | sudo mdadm --create /dev/md1 --metadata=0.90 --uuid=188d0705:2995b6c6:3017a5a8:c86610be --level=
1 --raid-disks=2 missing "$dev_dsm61715284new"p3
sudo mdadm --detail /dev/md1 | grep "UUID"
# mkswap /dev/md1
# swap区mount不了
# sudo mount /dev/md1 p3new

# 单盘basic，注意与上面的不同
[ -b /dev/md2 ] || echo 'y' | sudo mdadm --create /dev/md2 --metadata=1.2 --name=yunnas:2 --uuid=61203c7b:83ed31ed:29334798:82
e3a353 --level=1 --raid-devices=1 --force "$dev_dsm61715284new"p4
sudo mdadm --detail /dev/md2 | grep "UUID"
# 这个被mkfs.ext4过一次，因此可以mount
# mkfs.ext4 /dev/md2，也可以brtfs
sudo mount -t ext4 /dev/md2 p4new

# 现在把dsm61715284ori也mdadm --create，sudo mount所有/dev/mdxxx,如果挂载失败你可以把looppx单分区按开头结束chs挂载，像《《Dsm as deepin
mate(3):离线编辑初始镜像，让skynas本地验证启动安装/升级》》文一样，最后复制文件-p把权限也带上：

[ -b /dev/md3 ] || echo 'y' | sudo mdadm --create /dev/md4 --metadata=1.2 --name=yunnas:2 --uuid=61203c7b:83ed31ed:29334798:82
e3a353 --level=1 --raid-devices=1 --force "$dev_dsm61715284ori"p3
sudo mdadm --detail /dev/md3 | grep "UUID"
sudo mount -t ext4 /dev/md3 p3ori

# 最后，p4按文件复制,p2其实也可以这样只是被DD过了# cp -dpRx "$mnt_dsm61715284ori"p3ori "$mnt_dsm61715284new"p4new

sudo cat /proc/mdstat
```

在mdadm.conf上使用的是RAID设备的UUID,而不是文件系统blkid等出来的UUID.supperblock应该就是相当于raid的“pbr”之类的东西，在create时就会写入，DD会保留这种块信息

以上uuid是从打包到extra.lzma中获取的（解压： sudo sh -c 'xz -dc extra.lzma | cpio -id' 压缩： find . | cpio -c -o | xz -9 --format=lzma > extra.lzma），也是p3/etc/mdadam.conf中的内容,最好自己解包确认下。

以上脚本，还有就是我也不能确定超级快信息不能用多次create来覆盖，这是否一个once写入动作，所以首先查看原来的superblock,使用mdadm -E命令查看一下/dev/loop0px的情况：mdadm -E /dev/loop0px,如果有，可以使用：mdadm -As /dev/md0 --config /xxx/mdadm.conf重新启动这个raid，而不是重新创建

这是unmount.sh

```
sudo umount p2new
sudo mdadm -S /dev/md0
#sudo umount p3new
sudo mdadm -S /dev/md1
sudo umount p4new
sudo mdadm -S /dev/md2
sudo umount p3ori
sudo mdadm -S /dev/md3

sudo cat /proc/mdstat
```

umount.sh可多运行几次直到所有md显示stopped(-S不会抹除md块信息),我是直接在umount.sh前就tar dsm.gz dsm.raw的，这样镜像会有点不干净,打包的时候会显示镜像文件已更新，重打包即可)。

管不了那么多，进入系统(md0,md1不认出来也会停在安装设置界面)，结果失败，mark！以后再弄！

看来，要让群晖承认它的非原生布局，必须要绕过/usr/syno/bin/synocheckparition?它在闭源的scemd中,或者暴力修改linux.syno脚本中的判断，另外，(群晖里/usr/syno/sbin/synopartition --check也可在开放ssh的群晖里面使用。你可以用它check下列镜像的一个缩小版本)，至于那个存储分区(md2)（volume识别是无关紧要的问题，应该主要是修改/etc/mdadm/mdadm.conf里的md2.不过听说存储区重建还有《利用整块化自启镜像实现黑群在单盘位实机与云主机上的安装启动》结尾处提到技术，《群晖二合一版本无需进PE系统或DiskGenius 扩容的方法》，《引导二合一群晖系统的一键脚本动态扩容！》等技术）。

我们可以把deepin,winsrv2019core,osxkvm都可以做成这样的含dipe,tdpe的多区段镜像。grub2也有windows版，甚至ntboot也做了一个类似linux/initrd的命令(http://bbs.wuyou.net/forum.php?mod=viewthread&tid=417545 《NTBOOT & wimboot for UEFI GRUB2》不过可能与winsrvcore19的bootmgr配合不了，所以可以用类似以前《黑苹果》方案中用memdisk+clover.iso的方式用memdisk+ntboot.iso

在阿里云上安装黑苹果的一种设想

本文关键字：在阿里云上安装黑苹果，省事方案之把**osx icloud**当**nas**用

我们为什么执着于在云上安装黑苹果，其实我们的目的不是使用osx的工具链和编译环境，以我个人来说，我只是喜欢iclouddrive。它有着极快的同步速度，和极科学的同步算法。而且与finder集成。使得osx即nas，可以很容易做成异备，而且它的客户端app即服务端app的结构，使得如果本地osx结合云上osx，能容易打造出一台省事的nas。这些在前文《聪明的Mac osx本地云：同一生态的云硬件，云装机，云应用，云开发的完美集》中我们都说过。

在《在阿里云上装黑苹果（1）》和《在阿里云上装黑苹果（2）》文我们讲了一些有关这个话题的可能性和资源。

uefi可以理解为就是一类pc（支持uefi代码的），另一类就是我们用了很久的bios机器，强调机器是不是bios或uefi，主要是它们与os安装这个关系上产生了联系，有些os同时支持bios和uefi下的安装，有些os仅支持uefi（最新的osx），有些pc可以同时工作在bios或uefi下。（通过开机设置切换）有些只能工作在bios下(如guest云主机）。——— 即，bios,uefi是机器从固件上如何工作与os安装支持发生关系的二种方式。

要让这个uefi发挥作用，在安装os时，特定uefi firmware往往安装在硬盘中某特定分区，所以它又涉及到了分区格式。一般地，bios机用mbr，uefi机用gpt，现在的机器，一般可以这二种方式，现在的os也一般同时支持bios+mbr方式或uefi+gpt方式安装（2者居1）。当然，也存一些混合的方式和分区格式可以允许某os从mbr+gpt分区上安装启动（如bootcamp）。但要实现直接让一个不支持uefi的os支持bios方式安装则要难得多，—— 这里的本质问题，首先是firmware，并非分区格式。虽然相关但前者是机器本身的问题，后者无足重轻。

那么，黑群晖能不能在云主机上安装，问题的关键在于，1，云主机是没有uefi的。只有bios，除非那种能nested kvm的。能以软件方式喂给uefi。2，能不能有一种方式，使得改造osx或找一个特定的osx，使之在bios+mbr也能完成安装，(osx catalina 10.15能经过patch以mbr的hfs+启动)，3,如果2能解决，那么在guest云主机是不是还满足这个osx的其它硬件支持，如cpu是不是支持sse，osx是不是支持virtio盘。4，如果2不能解决，那么安装一种clover类似的启动器能不能帮忙解决(clover这类virtual uefi的作风就是在bios或uefi机上利用软件生造出一个新的类uefi层———能引导osx的固件层。原理与uefi类似)。依赖它guest 云主机不能虚拟成支持uefi，但能引导osx所需要的固件环境就够了。5，启动了osx rececvery是不是就算成功了呢？不是的。osx本体跟黑群晖不一样，不是所有的驱动都做在loader中。系统只是升级数据包。osx recovery跟osx有独立的os级的驱动。正是osx本体的驱动判断cpu是不是支持sse4这些能不能运行的后续过程。

如果2，3，4都能解决，实际上在guest上安装黑osx是完全可能的。我们选择的机型是阿里云的至强cpu类型支持sse4的，kvm+qemu是半虚拟的，所有的硬件中，cpu是host的，但是是直通到guest的，所以也算是guest的，virtio是guest自己的，这样的guest机是完成可以安装osx 10.10以上系列的(裸金属是调度器级别的有专门的硬件，不存在虚拟化，而阿里云机属于流行的半虚拟机和全虚拟化，都在os内核中做了手脚外加CPU vtd支持，kvm,xen属于半虚拟化)。

验症了事物的充要性，（模拟硬件和适配镜像，始终是问题最大的二个坑），已初步排除一个，下面就是完成镜像，制造一个能bios+mbr安装运行的osx dd直装式镜像，这二个过程实际是一个过程。我们可以在本地建造kvm+quem虚拟环境。我们选择的是osx 10.10，据说这个版本最温和可驯，实用性也接近最新推出的osx10.15,其它的版本也可以依理尝试。测试好了。就导出。

ps:这个其实google一下，网上有。 搜索Catalina MBR HFS Firmware Check Patch 10.15.x可得。

话说回来，为什么我们不提倡在云上完成安装过程，使用“正规黑苹果”的那种方式（这个说法不可笑，黑苹果黑的是硬件和loader，跟黑群晖一样，不黑安装包）。因为我们有installNET.sh这样的东西，这更符合云装机的风格。我们在本地，也有工具和资源模拟这样的制造环境。——而且，结果跟一步一步安装得到的结果并不矛盾。—— 更并且，有了一个dd image，这样以后在白苹果上安装更方便（不用再依赖recovery或usb那套了，而这，dd这不就是ghost或timemachine吗）。

最终继续实验以实证。把镜像上传到云主机实测，如果不行，反复测试。

————

下文就是《省事方案之把osx icloud当nas用》，这样省事个人云系列1，黑群，2，黑果，3，自实现，都有了，就算基本有了精神本体了。恩恩

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在阿里云上装黑苹果（1）：黑苹果基础

本文关键字：云**mac**当局域网第二**mac**用，本地盘与网盘组**raid**

我们在很多地方谈到，苹果之所以好用，是因为mac产品家族间注重融合，比如ipad os与osx是完全不一样的系统，但它们可以交互，他们是生态层面，加以技术辅助的融合，并不是从0开始就纯粹技术融合的，比如共用完全一样的内核,etc..—— 不管如何，这也使得苹果的产品非常难于部分替代的方式去整体代替，除非整个弃用。

苹果在现代OS软硬件的各个层次都有自己的定制，小到硬件（nvme硬件,disk format，bootcamp， NVRAM。etc..）大到OS，软硬层次的融合性深定制，这是一种闭环竞争力。

所以有各种黑苹果技术出现，企图将x86的黑苹果装在普通PC上。这称为Hackintosh,最开始的黑苹果技术是修改系统文件 — 这属于破解和反工程，已经过时，后来发展到利用loader这条合法正途上来：最终要能使黑苹果目标完装官方无修发布的原生安装包。

何谓loader？最初的loader是grub这种bios向os的传手，仅是一个硬盘主分区上的bootstrape，用来boot OS，但是复杂化的grub2可以干很多其它事情，纯粹可以发展为一套软件。这样，bios作为固件和grub作为软件的界限就不是那么重要了。所以后来干脆统一了强化的grub作为标准，这就是EFI，可扩展固件接口 (EFI) 依然跟grub强化一样是一个介于操作系统与硬件平台固件的软件接口。只不过它更强大，这里有各种厂商写好的EFI，引导计算机的硬件逻辑。驱动（为什么这里要用驱动呢，这个驱动不是为了EFI用它是稍后注入到OS的，苹果有iokit接口供各种硬件接入，开源的黑驱动本质也是合法的白苹果驱动）等。注意它们实际是软件可以被置换重写，不是固件它们只是引导固件为OS所用，所以完全可以在这里欺骗OS。

我们在以前谈到黑群，它有一个pe层，loader和所有的驱动都放在那里，系统仅为数据和升级包。而苹果没有pe层决定引导的因素存在，黑苹果要做的工作要更底层一些，因为它对硬件也有诸多规范（安装完的黑苹果只能是macbook,mac mini,imac,etc..型号系列的硬件组合，以及五码同一），而苹果原生loader是闭源不可抠的或patch的，抠出来也不能转移的，转移了也不能用的，即使能用也是不合法的，即使合法且能用上，它也不是决定发行包最终能安装成功的唯一因素。

所以复杂的第三方efi loader技术完全可以在这里发挥作用，统一承担所有的功能更多作用，比如伪装苹果承认的硬件规范和五码合一负责欺骗硬件的工作及更进一步的引导安装工作（注意这里进一步这三字）。

甚至这种第三方loader发展到通用EFI伪装器的境界，它不仅能伪装苹果机，还可以伪装任何类型的硬件。你可以将这种强化了的loader想象成”虚拟的efi as loader”。因为它有配置文件可以对应任何硬件平台的硬件组合。除了引导和驱动，对于黑苹果需求它甚至可以提供很多专门工具如四叶草的F4提取原生硬件码。

—— 随着技术的发展，黑苹果的loader大致分为：变色龙引导，CLOVER四叶草引导，Ozmosis刷BIOS的方法安装(相对而言它使安装完的PC更象白苹果，不过它本质是clover应用的局限类型。)。截止到发稿日期，主流的安装方式是CLOVER四叶草引导。

以下都是摘录：

CLOVER的主要任务

SMBIOS（DMI）充满了模拟真正的Apple Macintosh的数据 - 运行macOS的要求。序列号是假的，但有效。

ACPI表 - 包含在PC的ROM中 - 通常不能正确编写并且可能包含错误，主要是因为制造商很懒惰：APIC表中的CPU核心数不正确，NMI数据丢失，表FACP中缺少重置寄存器，错误的电源配置文件，缺少SSDT表中的EIST数据，最好甚至不提DSDT表。Clover试图解决这些问题。

OS X试图通过所谓的EFI字符串从引导加载程序获取数据，描述视频，以太网或声卡等附加设备。Clover生成此类数据。

基于BIOS的计算机将在初始启动过程中在传统模式下使用USB，这在将控制权传递给操作系统时会成为问题。Clover将改变USB模式。

macOS使用称为NVRAM的特殊内存进行信息交换，该内存包含在RuntimeServices中（不存在于传统加载程序中）。Clover提供此类信息交换，支持正确的Firewire功能和使用“启动磁盘”首选项面板。此外，NVRAM用于注册iCloud和iMessage服务。

ConsoleControl协议是必需的，在DUET中不存在。

有必要通过DataHub协议填充EFI / Platform中的某些数据，该协议在DUET中不存在，并且不总是存在于UEFI中。此外，设置了极其重要的FSBFrequency值，有时是错误的或完全缺失的。

在工作之前必须正确初始化CPU，但由于主板通用以匹配大量不同的CPU，因此内部表不包含任何正确的CPU数据。Clover执行已安装CPU的完整检测，更正表和CPU本身。一个副作用是工作涡轮模式。

另一个小问题：DUET和EDK2源通用写入以匹配不同的硬件，但硬件依赖性本身依赖于常量。这意味着一个特定平台的编译过程。Clover旨在实现通用性并提供自动平台检测。

一些资源

一些黑苹果常用的软件或者驱动开发者的主页，希望大家能及时更新驱动和软件，驱动需要自己去对应驱动开发者的主页去更新。

RehabMan 维护了很多黑苹果驱动和相关补丁 https://bitbucket.org/RehabMan/

Vit9696 lilu和相关插件、applealc的主要开发或维护者 https://github.com/acidanthera

Clover团队更新 clover的主要发佈渠道 http://sourceforge.net/projects/cloverefiboot/

我们的工作

你可以去百度查找利用clover装机，打包集成clover的镜像,甚至进阶到高级kext制作等课题。

而我们要做的云主机上的黑苹果，除了正确的colover配置和virtio驱动的准备，都是必须的。还有更多工作要做。我们最终要实现一个云端的mac机，就跟本地局域网的mac mini作为第二mac一样，我们最终需要的是为上文《Dsm as deepin mate：将skynas打造成deepin的装机运维mateos》准备的timemachine盘，与本地一个盘/文件夹组同步，类raid作用或群晖的share sync。

这样的混合云方案兼顾本地业务和远程，比局域网的mac mini业务更偏向多一些建站和远程灾备。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在阿里云上安装黑苹果（2）:本地虚拟机方案研究和可行资源参考

本文关键字：虚拟机黑苹果，**osx kvm guest**

在《在阿里云上安装苹果(1)》中我们讲到实机上安装黑苹果是一件难事，但也不是不可能，因为有全套的clover方案和不断丰富的kexts（kexts是EFI专属驱动,不是OS用的），如果硬件有问题，比如CPU不支持osx特有的指令扩展（低标准CPU不能运行OSX,To run MacOS Mojave, you need a processor with support for sse4.1 and sse4.2 instructions），主板不支持UEFI，显卡不受osx支持。我们换了它们就是。而云主机，比如阿里云，（1）我们除了不能控制它的硬件类型如CPU等，（2）还由于它是一种Qemu/kvm/virtio架构。是一种远端host架构，我们没有办法控制host，充其量只能在本地支持VT的实机上自建Qemu/kvm host/guest的虚拟机环境里捣鼓测试，(3)虽然本地都是自己可以控制的全套host/guest虚拟机方案，但虚拟机本身也有它特殊的局限，它不同于实机，是没有UEFI的（4）最后，还由于云主机是虚拟机的一种局限类型和特殊类型，我们除了得不到它的UEFI控制只能得到其guest之外，还要面对其仅支持从mbr+bios中启动系统（5）不论是实体虚拟机，还是云guest only虚拟机，我们都面对需集成virtio的事实。4,5关联紧密。因为UEFI中的驱动跟OS中的驱动联系紧密。

故在云主机上安装黑苹果就更是一件难事了，每一个新出现的大问题。都有可能使我们的最终目的泡汤。—— 但好在市面上支持osx vm的现行虚拟化方案(使用clover或不使用clover方案的黑苹果，甚至白苹果）也不少，如un-locker+vmware也可以，parallesdesk也可以是最好用的（它是白苹果），qemu/kvm下面的osx guest方案当然就更可以了。最接近我们这里要讲的阿里云的要算qemu/kvm/virtio了 ，在云主机上安osx的方案似乎并没有。但—— 但我们的努力总不致于毫无希望。

解决虚拟机上黑苹果的安装条件:软件/硬件/固件测试环境准备

对于(1)，我们可以换配置的方式来解决，如果不能就换服务商。这个略过。我们仅首先考虑本地虚拟机的情况（2）我们差不多不可能找到一家提供nested kvm和再虚拟化的云主机，除非裸金属，不过其投入很高。同2略过（3）虚拟机上有OVMF这样的东西代替UEFI，OVMF "is a project to enable UEFI support for Virtual Machines”(它依赖edk2的跨平台uefi方案)。不过OVMF只能用在HOST端，做成一文件作为参数传递给虚拟机启动程序，（4）如果说1,2,3都是在实体虚拟机上的解决之法，那么对于4，云主机，我们只能用上clover方案了，因为ovmf在这里不可用，且只有clover是同时支持bios和uefi系统的。那么是否它能用在阿里云机器上启动clover？（5）驱动方面，有一些第三方uefi virtio driver的这些可以被以clover常用的方式用到黑苹果上（将virtio xxx.pkg放到kexts中使用），最近的Mojave 10.13.3/4官方才开始有virtio支持，它是OSX内部的virtio。不过blk只能工作在legacy模式。那么问题来了，是否采用clover的传统模式下绕过uefi，依然可以用上virtio？osx是一定要用到efi的。

PS:（4）（5）这二个问题的结果直接关系到我们接下来在阿里云安装黑苹果的可行性，首先，集中第一个难题：在阿里云上bios+mbr安装clover并使之启动

https://www.reddit.com/r/hackintosh/comments/atliaw/clover_mbr_or_gpt_for_legacy_bios_system/ https://www.reddit.com/r/VFIO/comments/av736o/creating_a_clover_bios_nonuefi_install_for_qemu/ https://superuser.com/questions/1382154/clover-boot-efi-file-on-non-uefi-computer

> Your question touches two interlocked issues:
>
> BIOS vs. UEFI boot
>
> GPT vs. MBR boot
>
> The problem you run into is, that Clover needs a BIOS boot, but BIOS boot implies MBR boot. So obviously there needs to be some magic involved - turns out, this is quite straight forward: A disk can carry both, an MBR and a GPT partition table, and they don't need to carry the same information - this is called a hybrid partition table.
>
> So what you need to do is create a GPT partition table to your needs (including one partition for Clover - best put this first to make the next step easier)
>
> Then create a "protective" MBR-style partition table, that contains the Clover partition as bootable, and everything else in a single primary partition of type "EF".
>
> After you have installed Clover, when the BIOS-style boot runs via the MBR (installed by Clover), it will start Clover, which will in turn read the GPT partition table to boot the rest. This implies, that while BIOS sees only one bootable partition, Clover can see many.
>
> I have successfully used this method to triple-boot between Ubuntu, Windows and MacOS.

总结上面就是说可以连锁boot，实现在云主机上启动clover，然后，virtio驱动方面，Google “clover virtio”,我们在网上还是找到了一些讨论：

http://philjordan.eu/osx-virt/ 不过只有for network没有for blk,这是UEFI层的。 https://github.com/pmj/virtio-net-osx https://passthroughpo.st/mac-os-adds-early-support-for-virtio-qemu/ 这是OS级的

> the VirtIO drivers are only in the virtualization firmware OVMF. It's not part of clover
>
> Then you can't use VirtIO drivers. They are part of VM firmware to allow faster access to resources via the virtualization features of the CPU. If the VM is booted legacy then clover emulates the firmware over legacy interfaces, even if you built the VirtIO drivers they would not work because there is no access outside of the EFI firmware of the VM. I don't understand why your VM would be restricted to legacy

> booting, what VM are you using because I have QEMU, VirtualBox, windows server hyperV and multiple versions of different VMWare products and none are restricted to just legacy booting...

总结来说，clover下的virtio目前没有，不过此话的正确性有待查证，因为下面我们看到https://github.com/kholia/OSX-KVM通过clover用上了virtio pkgs.

osx guest方面的例子和自动化安装项目有:

在虚拟机上使用OVMF的例子有，它们是不使用到clover的：

http://www.contrib.andrew.cmu.edu/~somlo/OSXKVM/ https://www.contrib.andrew.cmu.edu/~somlo/OSXKVM/index_old.html https://github.com/foxlet/macOS-Simple-KVM

也可以在虚拟机上结合OVMF+clover作UEFI，这方面的例子有：

https://www.kraxel.org/blog/2019/06/macos-qemu-guest/ https://www.kraxel.org/blog/2017/09/running-macos-as-guest-in-kvm/ https://github.com/kholia/OSX-KVM

不管如何，下文开始，我们准备硬碰硬，自己实践出真知。彻底实现在云主机上安装黑osx.

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

云主机装黑果实践（1）：deepin上qemu+kvm装黑果

本文关键字：用**ubi**在**osx**上打造一分区一安装启动U盘，**deepin**上**qemu+kvm**装黑果

在上文《云主机装黑果》中，我们讨论了其可能性，或许这里面的坑远远不止那文提到的，甚至最终不能成功。—— 但不管如何，实践总要先行。在理论与实践组成的一对学习矛盾中，先见独到的理论思考，与慢悠盲目的实践，都必不可少。现在让我们开始正式的实践。

我们在前面说到，现在最新的流行OS，如windows,ubt发行，osx，都是基于uefi机考虑的。同一个系统，安装在同一台pc的mbr或uefi模式下，表现结果和所需驱动都是不一样的，比如你会发现有些驱动不能工作而有些可以，有些都有驱动的表现也不尽相同，这是因为bios和uefi定义了同一硬件的不同规范。之所以再次提到这个，是因为继续云机装黑果前，我们需要制作一套实践环境。需要在PC上制作一套省事的，统一的osx,windows,linux多系统安装和启动法 ——— 这些同样与uefi相关：

我们知道，uefi提供了将固件放在磁盘上的方式（与OS同一处理方式），这带来的好处就是，所有的启动，不必再mbr式写启动动到磁盘结构了，也不必动到整盘，大家只需要在gpt的efi分区中写入uefi文件即可安装，也可运行，而且这对U盘硬盘，（写入安装）和（启动系统）都是一样的。在各OS运用不同磁盘格式的现实情况下，这也许是是在PC上u盘安装启动多系统的最省事方法。—— 但是历史上，有些os,如Windows uefi的安装程序不安分（会创造额外分区，msr,等等），osx uefi和deepin uefi之类的os倒是十分安分,苹果是支持installmedia 刻录到分区安装和启动的（你可能不知道，osx最初的标语是打造一个人性化的os，从note，日历，到icloud，个人认为日常都离不开这些，都是很人性的。所以就连它的createinstallmedia安装盘方式和本地盘方式也是基于分区的。），deepin也可以分区安装。这二者共存就是我们想要的结果：你会在一个装有多OS的GPT盘的efi分区发现多个系统的efi文件夹。至于windows，你可以pass使用它的安装程序，用winpe代替，集成wim安装包，pe里面会有一个nt6setup工具选择efi分区和系统分区的选项，从而达成我们要的结果。

1）用ubi在osx上打造一分区一安装启动U盘，一分区一系统的本地盘格局

下面介绍在osx下运用unetbootin677来制作这样的安装U盘的的方法：Unetbootin的特别之处在于它将安装镜像写入分区而不是整盘，有别于市面其它产品，原理就是syslinux+uefi+fat32，使用时选择镜像后对应分区要事先格式为fat32。但fat32的缺点就是一旦文件有超过4g。就不能使用ubi，但是dvd也有容量限制呢，所以这只是一种折中。

在osx的图形化磁盘工具中，我们将一个32gU盘分成3个10g(或osx 10g,deepin15 4g,windows 10 2019 ltsc 6g，10G作exfat data区放点其它东西），每一个分区写入一个安装iso，efi分区是自然生成的，，Ubi是不支持windows的仅支持linux，但是实测也可以按同样对linux的方式强行将windows pe带安装包wim iso刷入，至于安装到的本地多系统，我们也将本地盘分为三到四个区（osx留大一点，最后一个区可以是exfat data放点东西，我是256g,80g给osx,2x40分别给deepin15和windows,90g做data），这样，我们可以把OS X主系统下图形硬盘工具作出的这套结构作为主要的日常视图。日后在finder->位置->下看到清爽的磁盘结构结果,也方便我们接下来的工作（我们需要在这些系统间切来切去而不能使用pd虚拟机来完成以下测试）。

注意，只有osx系统才有开机按alt(option)显示U盘各分区的启动。也许未来在没有这种支持的电脑上也可实现，比如我们可以做个智能U盘，当某系统所属的分区需要被当前使用到，就设置一个把这个分区推出来的功能。来代替平果机的这种功能，而不用使用mbr统一菜单在分区间chain来chain去的那种。（除非某天所有系统都能安装到同一个分区使用同一种格式）

好了，下面继续我们的重点工作，在deepin15中安装qemu+kvm测试安装黑果。

2)deepin上qemu+kvm装黑果：准备工作

由于我的macbookpro是一台2015，cpu是支持虚拟化特性的，所以直接：

Sudo apt-get update Sudo apt-get install virt-manager bridge-utils libvirt-clients qemu qemu-kvm

发现我的deepin-15.11-amd64在uefi下屏幕开机高亮的问题得到了解决，但是无线网卡没用，弄了个360usb wifi插好对付着用，发现下载到的qemu是2.81的。它的中心是一个libvirt加qemu，加一些类似管理机控制台的命令行和GUI工具，Qemu就是类似osx上bvyve一样的东西。后者也可以直接运行kernel。打开控制台，它里面有一系列虚拟的存储池，网络泡（配置文件和硬盘文件都在/var/lib/libvirt/image，/var/lib/libvirt/network/default.xml，/etc/libvirt/qemu/networks/default.xml)），你可以在GUI里新建一般复杂度配置的vm，但gui功能有限，更多更专业的参数需要命令行完成，如今天谈到的黑果。

根据前文《在阿里云上装黑苹果（1）：黑苹果基础，在阿里云上安装黑苹果（2）:本地虚拟机方案研究和可行资源参考》，https://www.contrib.andrew.cmu.edu/~somlo/OSXKVM/index_old.html和http://www.contrib.andrew.cmu.edu/~somlo/OSXKVM/ 这二文基本是有关在kvm上装黑果的技术起源。后者都是别人基于这些研究的一些脚本工具集成。这二文中，old那文是直接用apple boot-132->变色龙(停更)->enoch对变色龙的修改，后来那文是正儿八经地统盘修改ovmf(更干净和接近uefi定制)，，再后来，别人就用clover这个更强大免编译的uefi来代替ovmf的编译了作出更简单的osxkvm方案（如https://www.kraxel.org/blog/2017/09/running-macos-as-guest-in-kvm/的kraxel和https://github.com/kholia/OSX-KVM的kholia）。

PS：Boot-132 is a bootloader provided by Apple for loading the XNU kernel，但是它并不用在真正的Mac上。而Chameleon is a Darwin/XNU boot loader based on Apple's boot-132.Enoch对变化龙的增强可以启动早期osx，甚至到10.11,10.12为止的osx，说到底，enoch与clover做的工作是一样的，也都是efi的，(但enoch是对seabios朝uefi的过渡层，与普通bios机和云主机的bios接近，云主机是seabios)，具体到kvm，enoch是kernel方式喂

给qemu的,是bios上浅浅的一层。对Qemu/kvm guest方要求的更改最小，vs clover对uefi的要求最低。可能更适合我们未来将其用受限方式集成到受限的kvm环境如利用linuxboot等方式将其喂给普通阿里云ecs，以上是固件上的，其它工作方面，pc上黑平果的工作就像黑群，从loader着手，必要的时候，也重新打包安装包(vs synology upgrade pat)。因为这里要处理固件驱动注入，修改原包中正常启动参数到适合特定硬件启动，kvm guest机器其实可以类比通用pc上黑果技术的一组硬件特例处理,cloud kvm更是仅明确使用virtio驱动。

所以enoch这种，最接近pc黑果技术的起点，所以我们选择从它开始而不是clover

由于kholia在他的仓库中中把somlo的old文中的前期方案的相关脚本和工具也做了进来，所以这里为了方便我干脆采用它的一并说明，网上搜索kvmqemu osx出来的教程多用enoch_rev2839_boot和Install_OS_X_10.11.3_El_Capitan，所以这里也准备这样做。我clone到的2019.10.25的https://github.com/kholia/OSX-KVM/commit/64fcc1ef4d800197e8f4fc1421dd0f7060a72166，在整个脚本的backup文件夹内（backup之外的则对应最新利用clover的那些工具和脚本）。先看这个https://github.com/kholia/OSX-KVM/blob/master/notes.md，Enoch Bootloader (obsolete)是宣称过时的，意料之中，继续，1）脚本第一步，是create_install_iso.sh(for 10.11.0-10.11.4 and 10.12.0-10.12.5,只能在osx bash下运行)，它将从appstore下载到application/install OS X xxx app中的安装程序进行再打包，使之成为qemu适用的iso。如果你下载的OS X离线安装包，可以手动释放到application，这个过程中，一些必要的驱动和变动，也将被加进去，也就是破解kernel的方式。（这也是常见制作懒人包的方式 under pc黑果s），在安装过程中会用到。2）我下载到的是OS_X_10.11.3 El Capitan,运行，会出错，说tar 解压不到正确的格式包，下载安装Pacifist,用它打开install osx ei capitan.app/contents/sharesupport中的InstallESD.dmg，切换到”软件包内容”“资源”的“资源”，把Essentials.pkg释放到backup下，然后修改create_install.iso.sh中的对应处，成为python "$script_dir/parse_pbzx.py" "$script_dir/Payload" | cpio -idmu ./System/Library/Kernels || exit_with_error "Extraction of kernel failed”，重新运行脚本，得到Install_OS_X_10.11.3_El_Capitan.iso —— 与此同时，发现backup中，没有enoch_rev2839_boot，我从网上搜索下载到这个文件，放到backup中，值得注意的是网上下载到宣称2839的不一定就是对应版本.3）再在backup下创建一个40g硬盘，用来安装，qemu-img create -f qcow2 osxkvm.qcow2 40G。

3)deepin上qemu+kvm装黑果：启动

三大件准备完毕，我们需要先执行一些准备工作：

打开kvm参数：echo 1 > /sys/module/kvm/parameters/ignore_msrs，不这样，就会显示boot:不断重复重启。

网卡和网络设置部分： Kvm会自己创建一个virbr0和br0（见控制台虚拟网卡default栏），为nat（Installing "virt-manager" automagically creates the "virbr0" local private bridge :-)）。我们把osx网络与主机网络组成tap： sudo ip tuntap add dev tap0 mode tap sudo ip link set tap0 up promisc on sudo brctl addif virbr0 tap0

仓库脚本最后一般是用复杂命令行定义的VM和启动逻辑，这也是网上的教程常讲的主体部分（除了命令行，也有libvirt.xml版，但我们讲命令行版方便说明）：

```
qemu-system-x86_64 -enable-kvm -machine pc-q35-2.4 -smp 4,cores=2 \

      -cpu Penryn,kvm=off,vendor=GenuineIntel \
      -m 4096 \

      -smbios type=2 \
      -kernel ./enoch_rev2902_boot \

      -device isa-applesmc,osk="ourhardworkbythesewordsguardedpleasedontsteal(c)AppleComputerInc" \

      -usb -device usb-kbd -device usb-mouse \
      -device ich9-intel-hda -device hda-duplex \

      -device ide-drive,bus=ide.2,drive=MacHDD \
      -drive id=MacHDD,if=none,file=./osxkvm.qcow2 \
      -device ide-drive,bus=ide.0,drive=MacDVD \
      -drive id=MacDVD,if=none,snapshot=on,file=./'Install_OS_X_10.11.3_El_Capitan.iso'

      -netdev tap,id=net0,ifname=tap0,script=no,downscript=no -device e1000-82545em,netdev=net0,id=net0,mac=52:54:00:c9:18:27
\

      -monitor stdio

kvm=off，并没有使用kvm半虚拟化加速。
penryn,pc-q35-2.4，这是qemu虚拟出来尽量按近mac的cpu和主板机型,q35使用osx硬件采用的ich9芯片组。
-smbios type=2 seabios patch,required for booting [Mountain]Lion
我们注意到device isa-applesmc,osk=“”这行，对应somlo old文，这是Mac专属硬件，用程序算出来的
urhardworkbythesewordsguardedpleasedontsteal(c)AppleComputerInc这是固定的。
修改好boot,qcow2,iso各文件对应位置。
如果是-usb -device usb-kbd -device usb-tablet要改成-usb -device usb-kbd -device usb-mouse，否则moniter stdio下没有鼠标（usb-tablet后期在系统运行时可以有另外的开源驱动用）。
Ich9-intel-hda声卡
注意到虚拟网卡-netdev 开始的部分，这是最大的坑，作为黑果特例，网卡单一e1000是osx普遍内置的，net0是osx内部用的名字。52:54:00是osx合法的开头，见https://github.com/kholia/OSX-KVM/blob/master/networking-qemu-kvm-howto.txt,你还可给osx guest设置更复杂的bridge网络。
（也可以后期装virtio net pkg）
-monitor stdio使用原生kvm窗口。也可以spice或vnc
```

（这是整个osxkvm黑果的技术主体，也是pc黑果的技术主体https://www.insanelymac.com/forum/topic/278638-enoch-bootloader/，之所以以上都能轻描淡写做进命令行，是因为它们被解决了，从中我们看到somlo old文对应所提到的大大小小的坑,还有日渐功能丰富的qemu也会将它们整合包含(somlo文中一些对kvm本身编译更改的在qemu更新版中不必再做了)，故，以上在考虑将osx移到云主机时就需要逐个攻破，从qemu重新转到guest内，让kvm guest自包含而不是外部喂给，或依赖host qemu）

启动脚本boot-macOS.sh，显示虚拟机启动窗口。开头显示r2839字样，ehci warning: guest requested more bytes than allowed，接processing error - resetting ehci HC,但可以继续。如果不是r2839就启动不了。显示osx的recovery安装界面，进去，格式化磁盘，取个名，cp -av /Extra /Volumes/磁盘名（这个时候我们处在iso的文件系统中，extra就是写入的那些扩展,其实也就org.chameleon.boot.plist一个文件Chameleon configuration:timeout,To automatically boot into OS X without needing to press enter at the prompt，The EthernetBuiltIn and PCIRootUID keys fix the “Your device or computer could not be verified. Contact support for assistance.” error when logging in to the App Store.，主要是配置一些启动参数，没有kexts注入），继续安装。

如果在这里碰到No packages were eligible for install，一种解法是网上说的修改时间，如果点选左下角定制的时候，注意到osinstallsciprt为0kb，应试是镜像做错了，再次做好镜像即可。

——

至于加速的和硬件直通的kvm osx。你需要更深入。这样的方式使用黑果是不是违法的，因为硬件就是 apple规约中的那些，apple并没有规约硬件的外层或外延。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

云主机装黑果实践（2）：在deepin kvm下测试mbr方式安装的黑果10.15最新版

本文关键字：**QEMU 免kernel firmware**运行黑**osx**，把**enoch**做进**iso normal boot,**建立**10.15 enoch**懒人安装包

在前面云主机装黑果系列文章中，我们知道了变色龙Chameleon,四叶草clover作为黑果技术的本质，其原理就是重写了白苹果的引导。但它们不是一般的1）boot to os loader，而且也是2）firmware to os(fake devices, inject kexts，patch to kernels,配置tricks），即firmware as loader，黑果的技术基础就是依靠它们将kexts和配置写入安装包或系统运行时的内存代替官方,操作起来复杂度不低，因为它们都是为不同硬件写的，所以大量第三方kexts的注入选择，硬件平台适配和osx版本对应是一个考究，集成大量kexts可能使一台PC上的黑果”负重“（采用了过多的固件驱动），所以这类固件也不要在白果上用，可能损坏硬件，最有可能损坏的就是视网膜屏，绝非危言耸听。

它们都同时支持bios(mbr,mbr+gpt)和uefi（gpt）,可以启动windows,linux但又都对osx专门做了工作,都有相关工具，如gui wizard tools。但一般来说，clover更灵活更强大,支持更新。《实践（1）》文前面我们是实现了装在本地kvmguest上，使用了Chameleon。好了，我们继续讨论用它让黑果装在云上，阿里云云主机一般是qemu standard i440fx pii(x3)老intel机型，Xeon E5-2682 v4 以上cpu，bios机(如seabios8c24b4c)没有uefi，只能从mbr上启动，而且驱动部分是virtio。所幸硬件上这完全可以运行黑果最新版，经过一些破解，软件上也完全可以，我们要做的工作就是，将前文章第二部分启动脚本里的enoch做的固件和引导(包括支持某osx版本安装运行所有做的全部破解工作)的工作集成到osx内部/guest vm内部：

首先是那个显而易见的大问题：bios云主机是不能以外部喂给kernel的方式提供firmware的，那么enoch能不能像普通boot如grub那样，集成firmware到os镜像内呢，这是可以的，因为变色龙是一种firmware as bootloader，我们在grub2的命仅行下也能浏览文件，这里的loader本该是os的功能但居然可以共存，loader就是第一层os，linux是第二层os，可以前后启动，变色化也可以是一种grub2 loader，被安装在硬盘上且充当osx的传手，一前一后启动，—— 这就是说，mbr和bios机，也支持将一部分firmware放在硬盘上与os一起（就如同efi文件夹与os一样）。

不管如何，让我们实践吧，这次我们直接测试10.15。因为这次我发现PD居然有嵌套虚拟化，这次我们在OSX+PD虚拟机DEEPIN测试黑果,不用跟上文一样重启来重启去了。

1）让qemu免kernel在kvm guest主机上运行黑果

工作依然是建立新的可启动iso开始。(最终我们要得到一份能InstallNET.sh用在云主机的安装后的磁盘image dd)，但是我们需要在kvm/qemu上测试一下，所以还是按常规流程做成iso。因为10.15比前文osx-kvm脚本中的createinstalliso.sh提到的10,11,12,13更高，较10.11，12，事情有了不同：10.14开始，osx强制运行在uefi和apfs上，做了这方面的硬性选择，我们需要破解其为mbr和安装在普通hfs+上。能找到的开始支持10.15的引导是enoch2921 https://www.insanelymac.com/forum/files/file/71-enoch/。—— Crazybirdy有一个仓库和脚本专门 https://www.insanelymac.com/forum/files/file/985-catalina-mbr-hfs-firmware-check-patch/解决这二大问题，搜索Catalina MBR HFS Firmware Check Patch 10.15.x可到。下载，里面有一个MBR-Manual-Method15-20191209.dmg，解包打包成MBR-Manual-Method15-20191209.zip本地存储。还有一个enoch2922有几个配置文件，也下载。

研究里面的EasyMBR-Installer1015和说明文件，提到破解了让安装程序不作mbr和convert to apfs方面的检查，重点是脚本中的打包iso逻辑（Work on 10.15.txt What's the difference between createinstallmedia method, MBR-Manual-Method, and MBR-Automatic-Method?），使用上拖入一个大于10G的安装U盘分区和一个install 10.15.app就可以，内部逻辑上，它其实跟前文osx-kvm创造iso的逻辑差不多：都是提取安装包install xxx.app中的basesystem.dmg（这是安装程序，recovery），对kernel进行补丁，kexts注入，Extra写入，将install xxx.app本体写入（因为对kernel有patch和kexts注入，对标准createinstallmedia逻辑流程有侵入，所以并不同于供vmware+unlocker虚拟机用的，用写入到虚拟镜像盘或U盘的方式，然后createinstallmedia形成一个新的懒人cdr的方式。），最终生成一个cdr。

但是，1）这个脚本都设想enchos是外置的，引导本身并没有写进去，因为脚本作者都是设想这类脚本生成的镜像是供在osx下使用和u盘安装使用的，最后，变色龙引导以post install方式被安装，—— 这也就是为什么enchos文件crazybirdy也是分开放的，也是Enoch-rev.2922.pkg这类发行打包方式使然，，只有osx能option识别这类方式制作的“可启动”U盘。只有先运行enoch作为固件或loader才能识别其中写入的安装（本身并不包括enoch）。2）问题2，整个cdr也不是一个标准的可启动的iso（放到windows下或qemu -boot d -cdrom './install.iso.cdr' -m 512M 是不能启动的），因为本质上，这个cdr是dmg转换过来的,在osx和qemu+ kernel enoch机眼中，是可启动硬盘，而不是可启动iso

好了，我们现在要把EasyMBR-Installer1015里的脚本改造成生成镜像供qemu免kernel参数用上mbr patched了的10.15cdr，而不仅是U盘和实体机，最好是windows下也能用的bootable iso。。未来还要考虑云主机的接受方式。

2)变色龙mbr启动iso制作，在osx上制作变色龙mbr启动iso。将kernel集成到iso内部。

在2.4前变色龙是wowpc.iso发布的。但是r2922发布的是一个在osx上运行的“事后”package(追随clover系列风格)，离线安装到本地需要破解的黑果镜像或U盘。wowpc.iso里的boot加kexts加plist,后者gui方式需要手动选择大量项目（其实也不多）注意这并不是Chameleon wizard工具。

先创建一个虚拟U盘，供脚本使用。JHFS+即MacOS扩展（日志式），第一行创建的-fs是临时的，因为马上要在第3行重分区，第二行挂载，第三行分区为mbr盘中的一个hhfs+区要求10G大小保险，整个镜像11G，注意，去磁盘工具中查看dev node，我这里是disk3：

hdiutil create -o “install" -size 11g -layout SPUD -fs HFS+J hdiutil attach "install.dmg" -noverify -mountpoint /Volumes/install_build diskutil partitionDisk disk3 MBR JHFS+ Recovery 10G

未来这个recovery区即是脚本执行时要用的虚拟U盘。打开它。

然后安装r2922引导，用Pacifist打开Enoch-rev.2922.pkg提取其中的Core.pkg中的所有enoch引导文件到一个临时文件夹，cd到这个文件夹的/usr/standalone/i386下，1)首先Install boot0 to the MBR（我们这里是为硬盘生成引导，如果是要作成windows下的iso：用hdiutil makehybrid -o mywowpc.iso 从wowpc中或Enoch-rev.2922.pkg/Core.pkg解压出来的所有文件加你的Extra配置组成的所在文件夹/ -iso -hfs -joliet -eltorito-boot 从wowpc中或Enoch-rev.2922.pkg/Core.pkg解压出来的所有文件加你的Extra配置组成的所在文件夹/usr/standalone/i386/cdboot -no-emul-boot -hfs-volume-name "Chameleon" -joliet-volume-name "Chameleo" -iso-volume-name "Chameleo”），2)然后Install boot1h to the partition’s bootsector到disk3s1(boot1h为for hfs+,boot1f32为for fat32,boot1fx为for exfat，如果用这个虚拟U盘生成的cdr启动时提示Boot0 done,boot1 error，那就是boot1文件选错了)，3)最后Install boot to the partition’s root directory:

```
sudo fdisk -f boot0 -u -y /dev/rdisk3
sudo dd if=boot1h of=/dev/rdisk3s1
sudo cp boot /Volumes/Recovery
```

你还要把crazybirdy里面推荐的Extra配置文件复制到打开的虚拟U盘(这个U盘是可写挂载的)。

最后，打开EasyMBR-Installer1015编辑（这个盘跟脚本里的U盘实体属性有一点是不等的，比如它不能动态erase，故）：

diskutil eraseVolume JHFS+ Disk1mbrJHFS ${devDisk1mbrInstaller} 从这里开始改

diskutil rename "${devDisk1mbrInstaller}" "Disk1mbrJHFS" echo "rename virtual disk to disk1mbrJHFS done"

运行脚本后，选择我们准备的install 10.15.app和虚拟U盘拖入，运行到成功结束(中间会有lzvn需要放行。但脚本也不会等待，直接复制失败，所以放行后再次运行脚本从头建立虚拟U盘生成即可。)，它进行了mbr10.15黑果上前述脚本处理引导破解和firmware注入的工作（A1-A4），如果需要，你需要在镜像制作完成后额外手动完成以下动作还要二步：

A5. (Needn't with Clover, Need only if you use Chameleon with -f to boot Disk1mbrInstaller, use Pacifist v3.2.14+ to access Core.pkg.) Make directory of /Volumes/Disk1mbrInstaller/System/Library/Kernels first. Copy InstallESD.dmg/Packages/Core.pkg/System/Library/Kernels/kernel to /Volumes/Disk1mbrInstaller/System/Library/Kernels/kernel

A6. (Needn't with Clover, Need only if you use Chameleon with -f to boot Disk1mbrInstaller, use Pacifist v3.2.14+ to access Core.pkg.) Replace InstallESD.dmg/Packages/Core.pkg/System/Library/Extensions to /Volumes/Disk1mbrInstaller/System/Library/Extensions

这样，这个虚拟U盘就成为mbr patched,（注意它并不是脚本运行后要安装到的“系统目标盘”），等脚本完成，将其转成iso，${mbrdiskname}是脚本中的名字，实际名字会是Volumes/10153EasyMBR19D76之类：

```
hdiutil detach "/Volumes/${mbrdiskname}"
hdiutil convert “install.dmg" -format UDTO -o “install“
```

现在可以执行上文中的Boot-macOS.sh了，进入PD中的qemu-kvm，，这次，可以免kernel参数了，修改脚本删掉，内含enoch启动，它会自动找到分区中的installmedia：

以后，你完全可以re Detach re convert来重新修改dmg和生成新iso来解决启动中出现的问题或深度定制你的黑果。

———

其实黑果上登icloud也不会被封，如果你实在不放心，买个100元不到的iphone4，硬件上也登着。恩恩

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

云主机装黑果实践（3）:得到云主机安装镜像

本文关键字：**qemu kvm mojave enoch**，，**grub4dos** 手动**mbr**安装到**fat32**，**linux** 挂载多区段镜像分区

前面二文讲解了在kvm/qemu上尝试安装osx的工作，基本上，变色龙，四叶草，opencore的使用，就是在具体平台上尝试不同的patch参数(acpi,dsdt,etc)和大量第三方kexts，调试是最主要的工作，这三个loader支持都不相同，最弱的是变色龙，只是变色龙的mbr支持是最简单的和云机友好的。那么，作为一类特殊硬件，在云主机上主机启动遇到问题，该如何调试呢。又有哪些特殊呢？第一步是创建出云主机能用的镜像，上传测试。

前文我们使用的是PD开nested虚拟化实现了安装mbr patched 10.15。接前面的测试环境，我们继续来讲解。为了简化工作，我们要在deepin linux上事先创建这个镜像并处理，用raw格式而不是qcow2，得到的raw我们可以直接打包供InstallNET.sh用，我们这次选用的是10.14，因为10.15镜像有点大，而且我们尽量在linux下完成大部分工作 ——— 在osx recovery上分区很麻烦。

为了更方便调试引导区，我们要设置一个200m的引导区，并加入grub2和virtiope，然后chainloader变色龙iso。 —— 变色龙直接作为第一层引导好像在云主机硬盘上会有启动问题。

制作镜像结构和启动

在《离线编辑skynas镜像》文中我们讲过挂载一个多区段的镜像，但是现在我们是从0开始的镜像,是空盘和空分区，不是一个导出的有结构的镜像，我们采用20G，10.14安装后约有11-12G，云硬盘最小也是20G：

qemu-img create -f raw osxkvm 20G (或dd if=/dev/zero of=osxkvm bs=1024 count=20971520)

我们现在可以通过fdisk osxkvm, mkfs.vfat osxkvm直接操作这个镜像。使之有mbr和分区信息。但是注意：这是针对常规盘已有设备挂载点的（整盘已被挂载）那些操作，这里的镜像文件，虽然内部有分区信息（但没有被挂载），所以分区也得不到挂载设备点，不能经过格式化获得文件系统信息，也就不能实现挂到文件系统。

对于空白镜像，我们可以通过loop将整盘镜像虚拟成块，统一完成分区和格式化/挂载这样的工作：

losetup -fP --show osxkvm (P参数是带分区的镜像，show参数可以查看刚losetup到的盘，省掉一次losetup -a)

fdisk /dev/loop0进去分区,会自动转为mbr和msdos,n一个2048起的+200M分区，并a为活动，这就是我们的启动设定盘，其余空间作为一个分区，设置卷标BOOT,OSXKVM (注意不要生成在PD的/media/psf/xxx，否则打不开, 始终要注意，镜像文件的不同步,这时你要重启deepin。或者将镜像移到deepin里面)

现在才可以格式化和挂载分区：

mkfs.vfat /dev/loop0p1 mount /dev/loop0p1 /osxkvm1 （下回再挂载分区就是《离线编辑skynas镜像》文中的-o offsetnumber mount了）

好了，我们现在给它安装启动，我们不会直接在fat32里mbr装变色龙（原因：启动分区是调试必要的，云主机硬盘直接单分区hfs+启不动，4k硬盘装变色龙也有变数,变色龙装在fat32也有特殊：不能直接dd boot1f32，要把原pbr备份下来整个写入新的，再恢复部分老的，dd if=opbr of=/dev/rdisk1s1 bs=1 count=86 skip=3，dd if=opbr of=/dev/rdisk1s1 bs=1 count=90 skip=422 ）：

所以为了省事我们用grub2引导wowpc.iso这些原来在windows下用的：

grub-install --boot-directory=/bootpart /dev/loop0 grub-mkconfig -o /osxkvm1/grub/grub.cfg (这里/dev/loop0为虚拟块对应设备，/bootpart为fat32启动分区的挂载点,deepin的grub.cfg有延时设置不用update-grub)

为wowpc定制grub.cfg,加个menuentry "chameleon”，这里的wowpc.iso在接下来一节要准备好：

set root='(hd0,msdos1)’ linux16 /syslinux/memdisk iso raw (记得在系统内apt-get install syslinux拷相关文件到boot：cp -av /usr/lib/syslinux/ /bootpart/syslinux) initrd16 /wowpc.iso

把boot-macOS.sh去掉cdr发现这个raw image是可以grub2启动的,你还可以定制grub逻辑，加入tinycorelinuxpe。接下来就是制作镜像主体：那个osxkvm分区，和wowpc.iso引导文件了

Recovery里安装OS X得到镜像文件和引导

跟据上文，准备mbr patched 10.14 install.cdr（提一下，替换A5,A6会出现apfs stop），全程记得用https://github.com/chris1111/Chameleon-macOS-Mojave-USB/releases/tag/V2 chris1111的安装包ChameleonEnochv2.4svnr2922-Post.pkg在EasyMBR-Installer1013.sh脚本完全后写入，不要用其它地方来的。

ps: 自定chris1111的配置：

FakeSMC-Extra-Options->FakeSMC-NULCPU->FakeSMC&HWMonitor: HWsensors.V6.26.146，安装 FakeSMC,安装(Fakesmc是为把boot-macOS.sh中的device smc参数消减，所以接将kexts做进osx来解决，必备黑果驱动) CPUSensors,GPUSensors,LPCSensors都取消 FakeSMC-Extra-Options->Extra-smbios: Extr,安装 SMBIOS PC on Laptop->imac14,2，勾选 跳到最后Chameleon Enoch v2.4svn r2922->Chameleon Standard 勾选standard mode 安装变色龙引导到虚拟U盘，，它默认以-Xmpc -v等参数启动。删掉主题default后启动会更清爽。

在osx下虚拟U盘复制以下引号里提到的文件到某一目录，准备制作wowpc.iso，并把它放到虚拟U盘Extra下

sudo hdiutil makehybrid -o wowpc.iso “boot,Extra,Extra/extesions等chris1111的变色龙引导配置文件所在目录”/ -iso -hfs -joliet -eltorito-boot “虚拟U盘standalone/i386/cdboot”/cdboot -no-emul-boot -hfs-volume-name "Chameleon" -joliet-volume-name "Chameleo" -iso-volume-name "Chameleo"

cdr和wowpc.iso准备好了，启动boot-macOS.sh进入recovery,如果启动有问题，加-f重建缓存启动，

在recovery里界面的磁盘工具中查看未分区的盘号，这里是disk1s2，发现不是很好看，二个区凑一起了，但是没关系，能用就行(要得到好看的，diskutil partitiondisk disk1 MBR fat32 “BOOT” 200M HFS+ “OSXKVM” R，在linux里按上面过程返工吧),

退出磁盘工具进入命令行，打入：sudo diskutil eraseVolume JHFS+ OSXKVM disk1s2

拷wowed.iso到boot分区根下：cp /Extra/wowpc.iso /Volumes/BOOT/

进入安装界面,开始安装OSX到上一步准备的镜像osxkvm，因为是在PD中是虚拟套虚拟，安装性能有点慢，安装接近完全，最后一分钟要等好久,把最后得到的osxkvm测试启动，最后关掉PD，把得到的raw image gzip -c xx>xx.gz打包，完工！。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

云主机装黑果实践（4）：阿里轻量机上变色龙bootloader启动问题

本文关键字：处理云主机上大镜像安装问题，编译 **enoch** 变色龙

在《云机装黑果实践系列》1-3中，我们完成了直到生成镜像的所有准备工作，现在要上机测试了，进入最困难的boot-机型适配调试了，这也是黑果技术最典型的实践密集点，结合搜索引擎从最最小依赖一点一点添加配置是唯一的流程（基本上，变色龙是appleboot+fake efi as bios发展来的，具体机型千千W，云主机又特点，这种适配和调试工作变数和坑很多），我选的是一台阿里云轻量云主机，第一步是把镜像传上去。对于一个7G展开20G的打包镜像，moeclub的installnet.sh其内部使用的是wget gunzip输出到stdout再dd的管道，gzip版本太低，解压到前面很少一部分就会hang，脚本自动重启，找到那句将其改成wget -qO- '$DDURL' |tar zOx |/bin/dd of=$(list-devices disk |head -n1);Tar 不要加f和其它参数,版本不够。正常边untar边dd在我这（港区oss与轻量）要50来分钟，镜像正常启动grub2,进入tinycorelinux virtiope，fdisk /dev/vda，p显出正常hfs+分区，或者直接grub2 insmod hfs hfsplus part_apple，ls (hd0,msdos2)/ 也可以看到osx分区上的文件。之后迅速做一个快照，以防接下来的调试破坏系统。

如果说上面解决installnet.sh的脚本问题是小问题，那么大boss来了，r2922的cdboot在实机和kvmqemu机上可以运行，在云主机上根本无法运行（grub2进进入tinycorelinux virtiope，sudo mount /dev/vda1,sudo wget生成的iso，重启进入），不能显示引导界面，也是自动重启。(除了cdboot,按教程直接boot0h,hfs启动hfs分区的硬盘系统不行，用mbr+boot1f32也不行。)

问题可能在哪？这就是前面提到的ignore_msrs,我从memdisk版本问题开始排除，最后聚焦在boot本身上（cdboot这个stage2基本是一个boot+2560kib），猜想可能是版本问题导致的，利用排除法，先在网上海找了一通，能找到的最流行的低版本，就是v5.0.132 enoch r2839，，其cdboot写iso可以启动主机。最大r2841也可以，2842开始就不行。

翻看http://forge.voodooprojects.org/p/chameleon/source/changes/2842/，发现经过了三个commit，重点是源码上的变化:

Commit 2842, by ErmaC : General update Commit 2841, by Bungo : 1) Dropping DMAR (DMA Remapping table) to use stock AppleACPIplatform.kext - resolves stuck on "waitForSystemMapper" or "[PCI configuration begin]" 2) Added "ACPI" (all capitals) path 3) Small cosmetics Commit 2840, by Bungo : Sync

剩下就是实际编译出cdboot判断问题到底出在哪个commit了

从enoch的源码中找出变化，实际编译

编译环境是pd上的xcode 8.2.1 for EL CAPTAN 10.11，这套配置可以编译2841->2922，其它的都会有问题。苹果的开发链都很封闭自由度不高。只能选择这个配置了。

下载一个10.11，把它作成pd能安装用的镜像，原理跟mbrpatch打包类似，适合在PD安装老版本osx使用。

```
(以下差不多任意版本都适用)
hdiutil attach /Applications/Install\ macOS\ Sierra.app/Contents/SharedSupport/InstallESD.dmg -noverify -nobrowse -mountpoint
/Volumes/install_app

hdiutil create -o /tmp/Sierra.cdr -size 7316m -layout SPUD -fs HFS+J
hdiutil attach /tmp/Sierra.cdr.dmg -noverify -nobrowse -mountpoint /Volumes/install_build

asr restore -source /Volumes/install_app/BaseSystem.dmg -target /Volumes/install_build -noprompt -noverify -erase

rm /Volumes/OS\ X\ Base\ System/System/Installation/Packages
cp -rp /Volumes/install_app/Packages /Volumes/OS\ X\ Base\ System/System/Installation/
cp -rp /Volumes/install_app/BaseSystem.chunklist /Volumes/OS\ X\ Base\ System/BaseSystem.chunklist
cp -rp /Volumes/install_app/BaseSystem.dmg /Volumes/OS\ X\ Base\ System/BaseSystem.dmg

hdiutil detach /Volumes/install_app
hdiutil detach /Volumes/OS\ X\ Base\ System/

hdiutil convert /tmp/Sierra.cdr.dmg -format UDTO -o /tmp/Sierra.iso
```

再https://developer.apple.com/download/more/下载Command_Line_Tools_macOS_10.11_for_Xcode_8.2.dmg装上。

最后下载源码：svn co -r 2841 http://forge.voodooprojects.org/svn/chameleon/trunk/，svn co -r 2842 http://forge.voodooprojects.org/svn/chameleon/trunk/，svn co -r 2922 http://forge.voodooprojects.org/svn/chameleon/trunk/

它们的编译都是cd trunk，然后直接make，经过几次尝试，从最初直接替换libsaio/cpu.c,cpu.h,platform.c,platform.h，到最后仅在2842 src中libsaio/cpu.c找到以下二句并注释掉

```
//    case CPUID_MODEL_SKYLAKE_AVX:
//    case CPUID_MODEL_CANNONLAKE:

Re make clean
```

```
Re make
```

其产出的cdboot都是可以用在云主机上作正常启动的。这也可以被用在2922上。

再一步步调试出能启动云主机的变色龙配置：

到现在为止，终于进入对类似现实机器调试变色龙的流程来处理针对云主机安变色龙的问题了，这就是在上述一次次的“修改参数+打包iso+tinycorelinux上传”的循环（这也是我们当初使用grub2+memdisk的基本考虑）重复调试参数了：云主机较特别，可能会因为一个问题无法最终成功，但至少希望就在眼前。因为我们解决了大量关键问题。

注：碰到上传了正确的cdboot打包的iso，也启不动云主机到界面的问题，但有一个workarouds：2841和2922的wowpc.iso都上传，发现2922不能启动到界面，先启次一次2841，之后2922总可以成功。猜是loader改变了相关mbr参数，是残留的硬件作用。启动一次2841可以将其复位。（或许整个替换cpu和platform编译会根本解决）

最小配置是这样的：

```
org.chamalon.boot.plist

    <key>Kernel Flags</key>
    <string>-v -f</string>
        <key>Timeout</key>
        <string>30</string>
```

下文继续详解。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

云主机装黑果实践（5）：重得到镜像和继续强化前置启动过程

本文关键字：**,hackintosh Cirrus Logic GD5446**，**OS X,GD5446 kext**，黑苹果 **vga**兼容 花屏，黑苹果 虚拟机 显卡驱动

现在是4.4全国哀悼日，没有从天而降的超人只有敢于挺身而出的凡人

在前文中，我们解决了大量启动测试前的关键问题，但还不算进入了真正的黑果适配机型的关键步骤，因为有一些前置问题依然存在， 1，关于前文那个编译后的变色龙有时在grub2+memdisk下启不动的问题，有没有永久解决法，有没有解法使kvm的ignore_msrs在guest端实现？ 2,鉴于前面文章提到的云机适配问题，有没有方法在qemu中尽可能虚拟出接近云的guest配置，在本地深度测试后再上传镜像 3，我们在前面编译的trunk2922不是enoch的，跟社区的chris1111（能用于在文章2,3中正常安装启动的boot,cdboot）的boot存在大小相差，引导效果也不一样，前者启动后只进行了很小的起步。尝试chris1111(基于enoch)的可以进行到更多的进度,不加EasyMBR A5,A6可进系统。 4,在镜像中放installer还是完成安装后的osx，osx有没有类winsetup的驱动部署过程。 5,Mojave 1014是否真正选型正确？是否搭载了必备关键驱动。留给我们的工作多不多，有没有不可克服的？ 6,是否有改用clover的必要？

我们将继续一一处理：

新qemu配置方案

对于1，暂时没找到永解法，但找了一个新的暂时解法，把linux16,kernel16后面的16删掉ctrlx会halt，然后重新启动，系统进入到变色龙的概率会更大，

对于2，，一般的qemu/osxkvm用的都是q35机型，云主机是1996年的if440x，这种芯片影响了osx的支持设备，与能不能启动直接相关，比如机型的插槽都绑有具体功能，if440x不支持pci上的virtioblk启动iso，会卡在waiting for root device这是硬件导致It doesn't boot with the disk in virtio and scsci-virtio mode but boot in scsi.（启动分二段，第一段是变色龙的启动，稍后会详细说明kernel之后的启动调试也有卡root这里仅是变色龙启动），变色龙检出Cpu is intel xeon platinum 8163 cpu,Intel Xeon Platinum 8163（Skylake）,这是阿里云第四代服务器采用的CPU，但本机模拟不出这种U，vga不能用std，只能用cirrus-vga,但这种显卡在虚拟机中实测启动花屏，但安装过程依然可辨，在云主机上直接黑屏不能继续调试，但还是努力尝试一下，找出一套尽可能在保留if440x全局设定下启动osxinstaller的配置（毕竟阿里云不能改，其它的可以通过在osx端和变色龙端软性更改，比如尝试让virtio在osx端发挥作用前面说到这属于kernel之后的启动调试后来解决），所以以下，硬盘加了achi，显卡暂时没处理：

```
qemu-system-x86_64 -enable-kvm \
 -machine pc-i440fx-2.8 \
 -smp 4,cores=2 \
 -cpu Penryn,kvm=off,vendor=GenuineIntel \
 -m 4096 \
 -usb -device usb-kbd -device usb-mouse \
 -device cirrus-vga \
 -device ahci,id=ahci \
 -device ide-drive,bus=ahci.0,drive=MacHDD \
 -drive id=MacHDD,if=none,format=raw,file=./osxkvm \
 -device virtio-net-pci,mac='52:54:00:c9:18:27',netdev=MacNET \
 -netdev bridge,id=MacNET,br=virbr0,"helper=/usr/lib/qemu/qemu-bridge-helper" \
 -boot order=dc

#暂时不支持未实现
#-device virtio-blk-pci,drive=MacHDD \
#-drive id=MacHDD,if=none,cache=writeback,format=raw,file=./osxkvm \
#使用bridge网络,对于网卡，主机要有tap0先行创立起来，记得启动virtmanager中的那个virbr0哦，最好设为自动启动onboot，上面的device前drive后端定义才起作用。osx支持virtio net才起作用。
sudo chmod u+s /usr/lib/qemu/qemu-bridge-helper
mkdir /etc/qemu
touch /etc/qemu/bridge.conf
echo 'allow virbr0' > /etc/qemu/bridge.conf
#简单起见，除了ahci,pci都没有写slot位，功能位等，有默认值也可手动boot进tcpe。tce-load -wi pci-utils.tcz，然后lspci或lspci -t：以树状结构显示PCI设备的层次关系，包括所有的总线、桥、设备以及它们之间的联接,pci' domain='0x0000' bus='0x00' slot='0x07' function='0x2' 有四个域，或lspci -nn，或命令是lspci -vmm查设备ID与供应商ID，然后填上,我看到的是vga2 nic3 hd5
```

未来考虑使用Libvirt xml file format

新enoch变色龙

对于3，我们需要重新checkout编译enoch branch的源码而不是trunk的(变色龙作为Appleboot https://opensource.apple.com/release/macos-10133.html 和开源系统OpenDarwin的继承被社区用来作引导白果，通过KernelPatcher和FileNVRAM，变色龙已经支持纯白苹果免改iso只在内存中patching的安装方式https://www.insanelymac.com/forum/topic/299296-kernelpatcher-source-code/)，除了enoch branch,Chimera是另外一个有名的branch,变色龙 Enoch r2880 版本起 功能 变更如下: 1.内嵌入 FakeSMC.kext 2.新增 KextsToPatch,使用文件 /Extra/kexts.plist 3.新增 KernelToPatch,而trunk2922依

然没有这些。它的kernelpatcher module和filenvram是外置的（vs enoch内置的功能很欠缺内置只有一个acpi patcher，而且丢进extra/modules，会卡在tmsafey.kext或AppleKextExcludeList.kext ）,启动时没有[KERNEL PATCHER START][KEXT PATCHER START]。好多都没有发挥作用，—— 奇怪的是，它的make pkg结果居然是有跟enoch 2922一样的kernel patcher项的。有趣的是，我发现chameleon也有menuconfig。

svn -co HEAD http://forge.voodooprojects.org/svn/chameleon/branches/ErmaC/Enoch,Makerule中的CFLAGS = $(CONFIG_OPTIMIZATION_LEVEL) -g -Wmost -Wno-expansion-to-defined中的Werror去掉,修改一些源码加入从enoch删掉的-v debug部分：

```
在boot2/boot.c中能发现verbose() 的地方加个if (gVerboseMode) pause();
// ErmaC
verbose("\n");
if (gVerboseMode) pause();

kernel_patcher_internal.c中：
verbose("[ KERNEL PATCHER START ]\n");
if (gVerboseMode) pause();
verbose("[ KEXTS PATCHER START ]\n");
if (gVerboseMode) pause();

把加载kext的详细过程verbose出来并适当停顿，因为有时候在这里也会发生黑屏，卡屏，重启情况。这样肉眼可以看到。
drivers.c中：
verbose("Attempting to load drivers from standard repositories:\n");
if (gVerboseMode) pause();
verbose("Attempting to generate and load custom injectors kexts:\n");
if (gVerboseMode) pause();

我们把加载kext的verbose也做出来,并且10条一暂停。Enoch的Hfs.c里面Read HFS+ file被注释了，我把把它弄回来，依然在drivers.c中,这几句放FileLoadDr
ivers（）index=0和 ret=getdirentry开始处:
index = 0;
int plistverbosecount=0;
while (1)
{
    if ((plistverbosecount > 9) & gVerboseMode)
    {
        pause();
        plistverbosecount = 0;
    }
        plistverbosecount++;

        ret = GetDirEntry(dirSpec, &index, &name, &flags, &time);
        …
    }
….
}
这几句放long LoadMatchedModules( void )处:
int exeverbosecount=0;
while (module != 0)内：
    if ((exeverbosecount > 9) & gVerboseMode)
    {
        pause();
        exeverbosecount = 0;
    }

    exeverbosecount++;

对于启动过程中存在System uuid 不存在，fixing 00112233-4455-6677-8899-AABBCCDDEEFF的错误，这并不是导致卡waitting root device的根本原因。
应该是硬件驱动不起来或驱动没识别。但是可以去掉这个错误：
在smbios.plist中加入以下：
<key>SMsystemuuid</key>
<string>88D1F108-02EE-3622-99CE-EA53BEA03BD9</string>
或Libsaio,smbios.c->uint8_t *fixSystemUUID()中修改
```

最后，sudo make clean，sudo make pkg或dist，得到新boot和cdboot

新installer镜像和tcpe，测试启动

对于4，虽然不能找到osx安装没有驱动部署过程的证据，但找到了一个新的简便办法制作installer镜像，发现osx下操作raw image也很方便(linux处理镜像之后的结果在osx的磁盘工具中结构视图是mixed一团的也不好看qemu-img convert -f dmg -O raw也不好用)，可以直接：

```
dd if=/dev/zero of=osxkvm bs=512 count=52428800
diskutil partitionDisk disk3 MBR fat32 BOOT 100M JHFS+ OSXKVM R JHFS+ OSXKVMINSTALL 10G
（installer本来8G就够，但需要手动替换1G的Extensions故10G，installer放最后可在安装完系统可磁盘工具把它合并到OSX，OSX留15G刚好总25G刚好也是轻量
的磁盘大小）
hdiutil attach -imagekey diskimage-class=CRawDiskImage -nomount osxkvm
（仅attach,没有可装载的文件系统，一定要使用nomount）
挂载OSXKVMINSTALL,安装EasyMBR-Installer1013，不要安装变色龙bootloader,安装后，OSXKVMINSTALL会由日志式变成非日志式hfs
```

```
fdisk -e /dev/rdisk3,flag 1,write,yes,quit。
（fdisk: could not open MBR file /usr/standalone/i386/boot0: No such file or directory 没事，略过）
然后整个卸载回到linux装grub2,memdisk和tinycorelinuxpe
（sudo deepin-editor grub2.cfg，删掉所有内容，只保留二段menuentry，和一个全局set timeout久一点）。

在这里，你还可以制作一个带mount hfsplus免linux中关闭hfs+ journal内核（>3.4）的新tinycorelinuxpe。tc7是4.2的kernel，也有一个hfyprog.tcz(仅限x86)，进64bit deepin15编译32bit kernel里，gcc -v是630版，从http://mirrors.163.com/tinycorelinux/7.x/x86/release/src/kernel/Wget linux-4.2.9-patched.tar.xz和config-4.2.9-tinycore，sudo apt-get install gcc-multilib g++-multilib module-assistant libncurses5-dev libncursesw5-dev genisoimage( genisoimage 已取代之前的 mkisofs ),cd linux src根，Make mrproper,make menuconfig,加载config-4.2.9-tinycore,加上virtio(按/查找virtio，发现有virtio_blk,virtio_net,virtio_pci),再集成hfs文件系统(/查找hfs,发现hfs_fs,hfsplus_fs和hfsplus_fs_posix_acl)，save成.config，再额外打个防vmlinuz启动时出现“failed to allocate space for phdrs system halted”的bug补丁（见https://bugzilla.kernel.org/show_bug.cgi?id=114671t和https://bugzilla.kernel.org/attachment.cgi?id=209601&action=diffr按这个手动改下源码）

之后就可以make bzImage了，cp to vmlinuz,用http://mirrors.163.com/tinycorelinux/7.x/x86/release/distribution_files/core.gz和它组成新的virtiope,然后制作iso:把tc7的isolinux提取出来，cd tcpe,sudo genisoimage -R -J -T -v --no-emul-boot --boot-load-size 4 --boot-info-table -V "tcpe" -b isolinux/isolinux.bin -c isolinux/boot.cat -o ../tcpe.iso . (genisoimage很奇葩，-b 和 -c里面不能写绝对路径，应该写的是相对iso的相对路径，否则找不到boot catalog)

grub2也可直chainloader boot0，比如fat32装变色龙dd if=/dev/rdisk1s2 count=1,bs=512 of=origbs,cp boot1f32 newbs,dd if=origbs of=newbs skip=3 seek=3 bs=1 count=87 conv=notrunc,dd if=newbs of=/dev/rdisk1s2 count=1 bs=512，menuentry写insmod hfsplus,set root='(hd0,3)’,search --no-floppy --fs-uuid --set beec0ed4-a131-3ad3-9406-35cd8ebfb1b4,parttool (hd0,3) boot+,chainloader (hd0,1)/boot/chameleon/boot0,blkid显示uuid或tune2fs -U c1b9d5a2-f162-11cf-9ece-0020afc76f16 /dev/sda5更改uuid，有些电脑不支持 grub2下uuid的搜索，会Error：no such device:3c7c1d30-86c7-4ea3-ac16-30d6b0371b02，不过我们还是使用自己的grub2+memdisk+iso方案(iso中的文件系统是个hfs+,en0 0，它并不是fat32)。
```

镜像做好虚拟机可以直接用，上传到云机，installnet.sh如果https你要支持https，要wget --no-check-certificate，轻量的vnc后台窗口如需滚动窗口，请将鼠标移动到窗口左右两侧空白处，再进行滚动操作。。

现在要启动了，org.chameleon.Boot.plist几乎是总控文件，所有参数都在这里，kernel.plist,kext.plist只是patchs定义（注意KernelBooterExtensionsPatch to load extra kexts besides kernelcache），Themes文件夹和gui相关就不要加了。Extras/Extensions中不要加fakesmc，因为被enoch内置了，如果你碰到，ebios error，Startup error pause 5s,一般都是不正确kext导致的。

```
重点的org.chameleon.Boot.plist中的启动参数:

<key>Instant Menu</key>
<string>Yes</string>
<key>Default Partition</key>
<string>hd(0,3)</string>
<key>Timeout</key>
<string>10</string>
<key>Kernel</key>
<string>kernel</string>
<key>CsrActiveConfig</key>
<string>103</string>
<key>UseKernelCache</key>
<string>Yes</string>
<key>Kernel Flags</key>
<string>darkwake=0 npci=0x3000 debug=0x100</string>
```

同时kernel flags千W不要加-v，也不要加-f，毕竟这些都可以启动时按任意键手动加，也是后期调试必须手动敲入喂给kernel的部分。加debug=0x100（Don't reboot on panic，pause），但千W不要加wait加了wait=yes会黑屏，至于其它参数也是手动加的不要加进文件内（“-F”可以忽略org.chameleon.Boot.plist中的kernel flags）：

"-x" 安全模式，载入最少的kext。which ignores all cached driver files. "-f" (默认是 No) Lion 专用，选用 Yes 将载入预链接的 KernelCache，并忽略 /Extra/Extensions 和 /System/Library/Extensions 及 Extensions.mkext。建议在 KernelCache 已内含所有必要的驱动时，才启用。 使用 (-s) 单用户模式进入，可于在开机进入命令模式排除问题。也许bdmsg就是在这里用的，不过这仅限能启动进入console kernel

基本上，解决和完成了问题1，2，3，到这里，变色龙级别的启动是可以无阻进行的，出现问题的，大部分都是darwin kernel启动之后出现的，像上面提到的waitting root device。我们把变色龙的启动和kernel的启动分开，前面讲到的都是变色龙的，这里开始我们讲kernel的启动，，After the KEXTs decompression phase, you should see the kernel booting ，TMSafetynet.kext 是最后一个,之后便会starting Darwin kernel（它本来进入显卡的图像模式，应该换一个分辨率显示文字，）。启动文字在虚拟机上花屏变色（正常是白色），最后IOConsoleUsers: time(0) 0->0, lin 0, llk 1, IOConsoleUsers: gIOScreenLockState 3, hs 0, bs 0, nov 0, sm 0x0错误无法继续，然后apfs module stop，在云主机上黑屏，无后续，免v下，虚拟机图形有进度条，云机无。

看来，即使是这样，虚拟机跟云主机也不是一一对应的，云主机属于一个更小的集合。qemu机->qemu guest机->qemu云主机 就像同显卡qemu方案云主机会黑屏虚机不会。

驱动，以及影响启动的综合未完成问题

对于问题5，前面说的黑屏，卡住都是驱动问题（gIOScreenLockState错误原因是系统无法识别出你的显卡驱动,所以这个黑屏，是kernel的问题，不是变色龙的。），是1014不含这些驱动吗？而且10.14包含virtio,virtgraphics这些云机驱动（只是我们可能作很多黑果方面的工作使它工作：识别和驱动，在前面，我们按EasyMBR-Installer1013脚本A5,A6做的工作也把virtio,virtgraphic这些驱动做到S/L/E中去了，SLE:system,library,extensions）。

所以这问题大了去了，（为什么我们执着于10.14，14较新icloudrive支持高我们黑果的目的主要就是这，10.10-10.15的UI是一系的不排除内部实现有版本大更），我们从综合高度来思考这类问题：

硬件与bios与boot与os包含：QEMU uses the PC BIOS from the Seabios project and the Plex86/Bochs LGPL VGA，Seabios与qemu绑定bundle SeaBIOS 1.9.4 binaries with QEMU v2.7.1，就是变色龙no gui启动时显示的那个qemu，由qemu仿真的cirrus VGA就是一种典型的SVGAhttps://www.kraxel.org/blog/2018/10/qemu-vga-emulation-and-bochs-display/，https://www.kraxel.org/blog/2019/09/display-devices-in-qemu/.但是云主机lspci -vnn | grep VGA，虚拟是的是一块Cirrus Logic GD5446，为1996年pci vga的显卡VBE 2.00(VESA BIOS EXTENSION)，显存2-4mb(云机的bios是什么版本?)。Mojave ships with a driver for qemu stdvga and qemu cirrus vgahttps://www.kraxel.org/blog/2019/06/macos-qemu-guest/.osx从Yosemite开始不支持VGA,从skylake平台开始砍了原生vga通道，最新的Mac电脑都是没有VGA接口的，所以苹果的驱动里面理所当然就没有VGA接口的描述信息，macOS Mojave 10.14.x开始不支持NVIDIA显卡,苹果早就把 Mac下 Intel核显驱动 中 vga接口代码 全部移除了,最惹人注意的就是A卡的一堆、hd3000的几个、N卡的几个加上高通的无线网卡驱动，(没驱动的时候，vga有画面，只要驱动就黑屏。集显DVI没有问题，集显vga有问题)对于已有驱动，如AppleHDA.kext，Apple也已从其中删除了大量的Layouts，苹果就是这样一个追新弃旧派。

驱动工作模式与os支持：vga兼容模式一般是显卡在刚开始启动系统还没有加载显示驱动的时候，ramfb is a very simple framebuffer display device. It is intended to be configured by the firmware and used as boot framebuffer, until the guest OS loads a real GPU driver.If your guest OS supports the VESA 2.0 VBE extensions (e.g. Windows XP) and if you want to use high resolution modes (>=1280x1024x16) then you should use this option.对于linux：You have two Cirrus graphic options in the kernel, CONFIG_DRM_CIRRUS_QEMU is the KMS framebuffer that fit with qemu-kvm with a Cirrus virtual video card. The help say that it's only appropriate for virtualisation with the use of the modesetting Xorg video driver. So here, the use of the cirrus Xorg video driver is not what they recommand. You have also the CONFIG_FB_CIRRUS option that is a framebuffer driver for real Cirrus graphic cards. This framebuffer driver support the Qemu Cirrus emulated graphic card, but only tests can say fi it work with. 对于osx：https://www.kraxel.org/blog/2019/06/macos-qemu-guest/中提到virtualgraphics.kext也有Cirrus framebuffer，但并没有深入讲。这有可能是个efi驱动,那个simple osx kvm仓库中*.fd的补丁，就是OVMF（UEFI对qemu的实现）,Note that OVMF has a Display Resolution Setting too.那个virtio blk可能也是。

boot与os的关系：我们知道，能启动osx是qemu+firmware/boot(grub,chameleon)+硬件+osx的合力效果,grub有grub的文件，osx有osx的文件，变色龙有变色龙的文件，grub2与变色龙都属于osx前面一级的boot，这里grub2+wow.pc变色龙，grub2只负责boot，没有 firmware的作用，对于linux它有，比如它传给kernel的参数（这种关系就像变色龙之于osx），对于osx没有，我们发现grub2的mods很多与变色龙相同，而且用memdisk启动，存在依赖关系。会模拟出一个en(0,0)，Grub2和变色龙，也都可以在boot级就改变显示模式和效果。共享同样的boot技术。比如GRUB2也有insmod vga,insmod video_cirrus，it uses VESA BIOS Extensions to display the boot menu with enhanced resolution (and color depth), 当变色龙加osx出现Rooting via boot-uuid from xxx,Waiting on ioproviderclass时我们看到也提示了uuid，跟grub2一样no such device。 结合前面《linux as boot》我们讲了linuxboot和一个集成开发层到boot的东西，这样os下来就全是这种语言的app，甚至不必处理驱动硬件，python能在bootloader运行无须os也是一个例子BIOS Implementation Test Suite(BITS)。可以看出：其实boot就是firmware，它作为boot的意义是很浅一部分，驱动硬件，才是最主要的。

对于问题6，不能，理由无

对于现在的问题，virtio不能在云机osx驱动，虚机cirrus-vga花屏云机黑屏，有没有继续的必要？苹果的驱动里面没有VGA（但不是不可以从别处找到用于Intel GPU的VGA接口的代码，甚至第三方）Apple也已从其中删除了大量的信息，（但修改Injector这些kext修补缺失部分可以设备正常工作。）甚至修改dsdt，edid注入 —— 所以还是值得试下。

下文将力求解决显卡和硬盘驱动问题，真正实现云上黑果。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

云主机装黑果实践（6）：处理云主机上变色龙启动后置过程：驱动和黑屏

本文关键字：无显驱**vesa**方式驱动**osx10.14,mojave vga**黑屏，云主机的显示器，非**n**非**a**卡黑果，**waitting for root device**，**apfs modules stop 1432,appleexclude.kext**，**can't determine on the same uuid,qemu virtual display,qemu vga glitch**，**starting darwin x86**就黑屏**, osx cascadelake** 黑屏**,** 变色龙 **skylake** 黑屏

在前面5篇系列文章中，我们讲到了云上装黑果的基础工作和碰到的前置调试问题，我们采取的是用本地尽可能接近阿里云参数的情况下模拟镜像在阿里云端可安装运行（安装镜像/安装，安装后镜像/运行）的测试路径，假设这些工作完成之后，就可能肯定基本离成功在阿里云轻量上运行黑果这个目标很接近了，文1到文5中间做的都是顺利的准备工作，除了准备镜像都是前置启动故障排解，文章5末尾我们描述的一些1,驱动调试相关的问题和2,虚拟机不黑/轻量云黑屏的特别情况，这些都属于变色龙启动后darwin开始接手的地方，属于后置过程了，进入到机型适配了，被普遍认为是机型适配碰到的BOSS级的问题，本文开始，继续文5提到的问题排解和调试，其中有一条是在i440fx下我们无法从virtio blk驱动，因此也无法从其中的系统启动，启动中出现waitting for root device（我们初步判断这种问题有可能是前文说的pci功能位不对，也可能往往是驱动加载不正确，启动不起来，不要动不动就联系到AppleAHCIPort）。

这是因为文章2-4适配的是安装后镜像，是正确的路径，但到了5我们企图得到一个安装镜像，偏离我们原来的测试路径，其实，安装与运行，这二者不能二全，installer镜像是不含任何virtio和virtualgraphics驱动的，安装后的镜像才有，只要我们在qemu配置文章中手动加入功能位，而且使用没有受破坏的安装后的镜像，即，在变色龙启动完成后的kernel verbose界面保证不出现显示virtio驱动加载错误（如果你强行把virtio,virtgraphics相关和依赖kext丢进E/E或S/L/E，这往往又破坏镜像加的权限和缓存，照样会出现apfs modules stop 1432，修复往往不起作用），就可以实现识别virtio blk驱动，从其中启动的，而cirrus-vga显卡驱动，在启动时有花屏，进入系统后却是可以识别并驱动的（只是是800x600模式的SVGA兼容模式，此模式由VESA为IBM兼容机推出的标准。分辨率为800x600，查看设备为00:02.0 "Class 0300" "1013" "00b8" "1af4" "1100" ，1013 Cirrus Logic，00b8 GD 5446，1af4 1100 QEMU Virtual Machine，这说明virtualgraphic也发挥作用了）

驱动问题并没有发展到需要patch kext处理显卡驱动，更没有涉及到加起QE/CI显卡加速来，因为我们是云主机无须显卡加速，但黑屏依然是一个暂时无解的问题，可能是CPU指令集问题，也可能是黑屏与修改edid法修复显示器参数,dsdt法，但这些都可以作为猜想，都需要慢慢找出问题。

重整理的测试参数

我们重新设计一下测试参数：

创建安装镜像cdr:

```
hdiutil create -o "osxkvminstall10146" -size 8g -layout SPUD -fs HFS+J
hdiutil attach "osxkvminstall10146.dmg" -noverify -nomount
diskutil partitionDisk disk2 MBR JHFS+ OSXKVMINSTALL R
直接用mbrpatch脚本免Q5Q6，不要手动复制basesystemdmg/s/l/e或pacifist解压，否则有权限问题，会导致apfs module stop
及时安装变色龙启动,这个dmg和cdr卸载后会变成只读，所以尽早安装变色龙，
hdiutil convert "osxkvminstall10146.dmg" -format UDTO -o "osxkvminstall10146"
```

创建安装后镜像：

```
dd if=/dev/zero of=osxkvm10146 bs=512 count=52428800
hdiutil attach -imagekey diskimage-class=CRawDiskImage -nomount osxkvm10146
diskutil partitionDisk disk2 MBR fat32 BOOT 100M JHFS+ OSXKVM R
（mbr的在安装后的osx不可动态调整因此没有必要在尾端保留一个osxinstaller，做成osxkvm,osxkvminstall镜像为一体，用尽25G）
fdisk -e /dev/rdisk3,flag 1,write,yes,quit。
在linux中安装grub2,syslinux,tcpe,wowpc
```

用这套安装,按esc进,用cdr上的启动就可以避免安装时osxkvm所在卷不可卸载:

```
qemu-system-x86_64 -enable-kvm \
 -machine pc-q35-2.8 \
 -cpu Penryn,kvm=off,vendor=GenuineIntel \
 -m 5120 \
 -usb -device usb-kbd -device usb-mouse \
 -device ide-drive,bus=ide.2,drive=MacHDD \
 -drive id=MacHDD,if=none,format=raw,file=./osxkvm10146 \
 -device ide-drive,bus=ide.0,drive=MacDVD \
 -drive id=MacDVD,if=none,snapshot=on,file=./osxkvminstall10146.cdr \
 -boot order=dc,menu=on
```

用这套启动安装后的系统，注意去掉了smp，且其中新手动加的pci插槽和功能位,tcpe下lspci就能看到:

```
qemu-system-x86_64 -enable-kvm \
 -machine pc-i440fx-2.8 \
 -cpu Penryn,kvm=off,vendor=GenuineIntel \
 -m 5120 \
 -device cirrus-vga,bus=pci.0,addr=0x2 \
 -usb -device usb-kbd -device usb-mouse \
 -device virtio-blk-pci,bus=pci.0,addr=0x5,drive=MacHDD \
 -drive id=MacHDD,if=none,cache=writeback,format=raw,file=./osxkvm10146 \
 -device virtio-net-pci,bus=pci.0,addr=0x3,mac='52:54:00:c9:18:27',netdev=MacNET \
 -netdev bridge,id=MacNET,br=virbr0,"helper=/usr/lib/qemu/qemu-bridge-helper"

chameleon.boot.plist中删掉：
<key>UseKernelCache</key>
<string>Yes</string>
```

启动的时候一定要手动打上UseKernelCache=Yes，否则还是识别不了virtio,-v 可以看到识别了virtio blk，在Rebuild caches after update，early boot complete,continue后会出现：notice - new kext com.apple.driver.kextexcludelist, v14.0.3 matches prelinked kext but can't determine if executables are the same (no uuids)，等漫长的timeout之后(我等了五分钟)。可进界面（又是漫长的左上角滚动海浪球），下次启动不会等这么久。

完成！可顺利加载virtio和显卡驱动进入系统，封装osxkvm10146为gz，上传到云主机。你可以把那个第一次进入等待过久的过程固化在镜像中打包，云机中以后就不用等这么久了。

至于如何让Applevirtio手动变成一个boot time driver集成在install镜像或osxkvm镜像中免去使用这个prelinked kernel(一般情况下避免使用prelinked cache,-f是用来强制重新从磁盘s/l/e加载kext跳过缓存的。但并不会重建prelinked缓存，注意把kext与kernel的缓存分开)，我尝试把kext提到变色龙或s/l/e,权限修复工具，手动修复，kextcache 实现的几种方法都无效，KCPM Utility Pro，kext utility 2.6.1这类工具无法给离线系统注入kext，更提示无法给恢复系统注入kext和make cache。都没有成功。

对cache机制的理解：

The kernel is the first component of the operating system to start. It has no other tools available. In particular there is no way to check code signatures, and all file system access is very hard at this point. Apple therefore decided to prelink the bare kernel with all kernel extensions every time the kernel or one of the extensions is updated, and to start only that prelinked kernel at boot time.

Since the prelinked kernel is on a read-only volume, it cannot be updated directly. Apple had to conceive a new mechanism for updates.

When you reboot or shut down your machine, launchd stops all processes. Then it remounts the system volume in read/write mode. This is possible because launchd has the entitlement com.apple.private.apfs.mount-root-writeable-at-shutdown. Then it runs /var/install/shove_kernels to copy the new kernel. But apparently this doesn’t actually work. So to update a kernel extension you need to disable System Integrity Protection or manually trigger a kernel update after booting into macOS Recovery.

Note: This is fixed in macOS 10.15.1

我用的脚本：

```
#!/bin/sh
sudo chmod -Rf 755 /S*/L*/E*
sudo chown -Rf 0:0 /S*/L*/E*
sudo chmod -Rf 755 /L*/E*
sudo chown -Rf 0:0 /L*/E*
sudo rm -Rf /S*/L*/PrelinkedKernels/*
sudo rm -Rf /S*/L*/Caches/com.apple.kext.caches/*
sudo touch -f /S*/L*/E*
sudo touch -f /L*/E*
sudo kextcache -Boot -U /
```

驱动解决，也不用考虑安装镜像和安装问题，接下来只需探索黑屏问题。

云主机变色龙黑屏探索1：cpu兼容

我们在上文说到，在云轻量上，变色龙读取s/l/e，加载完plist和kext中的可执行文件，直到TMsafetyNet.kext ，从这里开始就黑屏了(控制台可以看到磁盘和CPU都没有读写，应该是halt了。而不仅是虚拟vnc显示器没有信号)，我们上文也讲到Wait=Yes不起作用，可能也是在执行wait=yes前就黑了，因此我们有必要修改变色龙源码，在结束-v所有输出98%内容后输出starting darwin kernel前（或许是判断文件名tmsafetynet后），每条输出都pause一下，看到底在哪里黑屏了。

上面的qemu启动配置，就cpu和display没有真正做到与云轻量对应。如果没有cpu直通，kvm虚拟出来的是vcpu，客户机看到的是基于 KVM vCPU 的 CPU 核，而 vCPU 作为 QEMU 线程被 Linux 作为普通的线程/轻量级进程调度到物理的 CPU 核上。

进轻量云，tcpe cat /proc/cpuinfo，对于cpu，我们发现CPU是变化的，有时是Intel(R) Xeon(R) Platinum 8269CY CPU @ 2.50GHz（Cascadelake），有时却是：Xeon Platinum 8163（SkyLake），这是一个1C的主机：processor: 0，physical id: 0，siblings: 1，core id: 0，cpu cores : 1，也可以看到它的feature指令集。

我们cpu设为最基本的客户机 CPU 模型qemu64,把所有的cpu flags都加上。后面加check,enforce来查看与host的比较情况，最后上面的cpu变成：

```
-cpu qemu64,kvm=off,vendor=GenuineIntel,+fpu,+vme,+de,+pse,+tsc,+msr,+pae,+mce,+cx8,+apic,+sep,+mtrr,+pge,+mca,+cmov,+pat,+pse36,+clflush,+mmx,+fxsr,+sse,+sse2,+ss,+ht,+syscall,+nx,+pdpe1gb,+rdtscp,+lm,+constant_tsc,+rep_good,+nopl,+pni,+pclmulqdq,+ssse3,+fma,+cx16,+pcid,+sse4_1,+sse4_2,+x2apic,+movbe,+popcnt,+tsc_deadline_timer,+aes,+xsave,+avx,+f16c,+rdrand,+hypervisor,+lahf_lm,+abm,+3dnowprefetch,+fsgsbase,+tsc_adjust,+bmi1,+hle,+avx2,+smep,+bmi2,+erms,+invpcid,+rtm,+mpx,+avx512f,+avx512dq,+rdseed,+adx,+smap,+avx512cd,+avx512bw,+avx512vl,+xsaveopt,+xsavec,+xgetbv1,+xsaves,+arat,check,enforce
```

放在本地启动，发现2.8的qemu-kvm -cpu help，Qemu Recognized CPUID flags，是不支持以下的： qemu-system-x86_64: Property '.constant_tsc' not found qemu-system-x86_64: Property '.rep_good' not found qemu-system-x86_64: Property '.nopl' not found qemu-system-x86_64: Property '.tsc_deadline_timer' not found

```
而且，我的主机cpu是不支持以下的（这并不影响什么，只要qemu支持模拟就对我们的实验结果没有影响）：
warning: host doesn't support requested feature: CPUID.01H:EDX.ht [bit 28]
warning: host doesn't support requested feature: CPUID.07H:EBX.hle [bit 4]
warning: host doesn't support requested feature: CPUID.07H:EBX.erms [bit 9]
warning: host doesn't support requested feature: CPUID.07H:EBX.rtm [bit 11]
warning: host doesn't support requested feature: CPUID.07H:EBX.mpx [bit 14]
warning: host doesn't support requested feature: CPUID.07H:EBX.avx512f [bit 16]
warning: host doesn't support requested feature: CPUID.07H:EBX.avx512dq [bit 17]
warning: host doesn't support requested feature: CPUID.07H:EBX.avx512cd [bit 28]
warning: host doesn't support requested feature: CPUID.07H:EBX.avx512bw [bit 30]
warning: host doesn't support requested feature: CPUID.07H:EBX.avx512vl [bit 31]
warning: host doesn't support requested feature: CPUID.80000001H:EDX.pdpe1gb [bit 26]
warning: host doesn't support requested feature: CPUID.0DH:EAX.xsavec [bit 1]
warning: host doesn't support requested feature: CPUID.0DH:EAX.xgetbv1 [bit 2]
warning: host doesn't support requested feature: CPUID.0DH:EAX.xsaves [bit 3]
warning: host doesn't support requested feature: CPUID.0DH:EAX [bit 3]
warning: host doesn't support requested feature: CPUID.0DH:EAX [bit 4]
warning: host doesn't support requested feature: CPUID.0DH:EAX [bit 5]
warning: host doesn't support requested feature: CPUID.0DH:EAX [bit 6]
warning: host doesn't support requested feature: CPUID.0DH:EAX [bit 7]
```

暂时先在上面的cpu指令集中删掉以上4条。正常启动Qemu，果然，这个cpu下发生了黑屏，跟云机表现一致,重新在deepin15编译qemu高版本，QEMU 3.1.0 in added the Cascadelake-Server CPU,我们下载的4.2.0：

sudo apt install libpixman-1-dev bison flex libsdl2-dev(不加这个会卡在vnc server) ./configure —prefix=/usr —target-list=x86_64-softmmu —enable-sdl make install

cat /usr/share/libvirt/cpu_map.xml |grep Cas -A100，发现有Cascadelake-Server，但是我们不把cpu指定为cascade lake-Server，我们更相信指定具体的指令集，删掉的4条可以加上了，重新运行qemu，可以运行，但依然会出现host cpu缺失指令集warning，虚拟机表现同样黑屏，跟云机一致。

看来，这个CPU太先进了，只能尝试CPUID fix了（如https://www.insanelymac.com/forum/topic/335650-kernelandkextpatches-1013x1014x1015x-x99x299/），就跟处理前面的msrs一样，变色龙集成了一部分cpu patch，但需要我们做更多工作。不能确定是哪个指令集缺失或什么其它原因导致的问题，多开几台不同CPU的云主机试试，或者在本地不断调整指令集参数作Clear Test测试，然后在相关kext处作patching针对解决。

云主机变色龙黑屏探索2：虚拟显示器注入

在探索1提到，驱动和显示器问题可能并不是黑屏的原因(nv_diable让vesa生效没用,radvesa没用，删s/l/e驱动让Vesa生效也没用，一般不必也导致权限问题，resolustion915 fix也不是)。安装过程中花屏和vnc中glitch现象（https://ostanin.org/2014/02/11/playing-with-mac-os-x-on-kvm/）是24位与36位混乱形成的，但不影响进入系统(想到这里，突然也记起以前用向日控控的时候，笔记本有屏幕，需要拔掉，才能那个网络界面中显示,控控对台机好用自配屏的笔记本不行)。但为了完善我们的测试过程，我们还是考虑可能的显示器edid问题：

The EDID (Extended display identification data) data structure have all the info of your graphic card and other video sources. EDID是由VESA——视频电子标准协会定义的，并在1994年和DDC标准1.0版一起推出了1.0版本，qemu也支持以下虚拟显示器：

-display sdl - Display video output via SDL (usually in a separate graphics window). -display curses - Displays video output via curses. -display none - Do not display video output. This option is different than the -nographic option

- display vnc, 这就是我们云机支持的，

再看几条爬贴参考：

https://www.tonymacx86.com/threads/black-screen-after-boot-menu.165117/

https://www.tonymacx86.com/threads/guide-general-framebuffer-patching-guide-hdmi-black-screen-problem.269149/

新版本的qemu支持edid，但是云主机的qemu是我们无法控制的。考虑这条方向上的尝试不实用。总之探索2最好在探索1完成后再进行。

———————

也许我们以后要深度修改变色龙源码，把usekernelcache固化进变色龙，或加入云专用的kexts，定制一下whatevergreen之类的东西与云主机适配，修改resolustion src为cirrus logic vga所用。尽量避免动到镜像本身。还有那个msr ingore问题，希望在变色龙端彻底解决：msr 0x35 which qemu/kvm not implemented yet then ..mojave和catalina都越来越大了，未来精简osx为更小的镜像比如至仅命令行。据说还有mojave，AppleQEMUHID.kext…

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

云主机装黑果实践（7）：继续处理云主机上黑果前后置问题，增加新boot

本文关键字：**fakecpuid apple logo stuck,变色龙fakecpuid,Cascadelake fakecpuid,qemu define cpu model,Skylake-Server qemu hackintosh**

我们知道，云上能启动黑果是上下一条路径上包括硬件，bios,boot,os在内合力的效果，硬件上的CPU又是这条路径上产生变数最多的因素，变色龙处理cpu兼容是一关，kernel处理cpu兼容是它独立的一关，这二个都存在这样的过程。而变色龙与kernel对cpu的定义不一样，存在不一样的处理过程，所以可能导致不能把cpu信息传给kernel最终发生黑屏这样的过程（更何况硬件/模拟器上，qemu对指令集和CPU支持也不一样），因此我们需要重新全盘统筹。

新的qemu和新加的boot

如何判断阿里云上qemu的版本？除了搜索引擎，也许只有实测靠谱，如何在guest体内判断host的qemu版本。好在阿里会把qemu版本信息写在系统里，我们在轻量上执行dmidecode，出来BIOS Information Vendor: SeaBIOS Version: 8c24b4c Release Date: 04/01/2014，System Information：Manufacturer: Alibaba Cloud Product Name: Alibaba Cloud ECS Version: pc-i440fx-2.1。

因此我们重新编译qemu with deepin1511中的gcc630：

```
wget http://wiki.qemu-project.org/download/qemu-2.1.0.tar.bz2
sudo apt install libpixman-1-dev bison flex libsdl-dev autoconf automake libtool
(不加sdl会卡在vnc server，注意这里是sdl不是sdl2)
(不加auttools会autoreconf: not found，不加flex bison在install时会出错)
./configure --prefix=/usr --target-list=x86_64-softmmu --enable-sdl
make install
```

这个出来的bios是BIOS rel-1.7.5-0-ge51488c-20140602_164612-nilsson.home.kraxel.org，离20140401很接近了。就这样吧。这样qemu版本就适配了

除了qemu，enoch2922本身就是个问题，因为其中根本就没有对最新cascadelake的定义和支持，查看src/libsaio/platform.c,platform.h我们看到它支持的CPU和指令集，三src驱动高重我们的调试思路是确保这三者从上而下，都有同样的关于cpu正确逻辑的继承实现或屏蔽（或者像Penryn一样能启动就行，可能仅仅因为cpu的问题在这这三者不冲突）。但另一个有名的boot：clover可能有:

Clover is a later revision of Tianocore. Both are 'firmware in RAM' replacements for UEFI firmware. It allows you to boot in MBR\CSM mode and then run Clover which acts as a 'pseudo-UEFI boot manager', allowing you to boot to a UEFI OS from an MBR\CSM boot.2015-10-16 Rev 3289 新增 Skylake CPU 及 核显 及 SMBIOS 支持。而且，clover更强大有更先进的gui wizard，高于mbrpatch指定的Clover r4514+ boot 10.14 fine的版本选择也多，clover下也有直接把log保存在第一个分区的功能（而变色龙仅能得到bdmdg，Bdmsg其实是仅直到变色龙启动完就停止的。/var/log/system.log下有kernel启动的log）。不妨再添加一个boot同时测试？

尝试再添加一个boot为四叶草，直接从http://sourceforge.net/projects/cloverefiboot/files/Bootable_ISO/下载以wowpc.iso同样的方式启动。我们知道clover同时支持引导bios和uefi，也可以在实机或qemu（+pc-i440fx）下启动https://passthroughpo.st/hackintosh-kvm-guide-high-sierra-using-qemus-i440fx-chipset/，一般地，我们使用grub2+memdisk+clover.iso这种干净的老方案（网上还有把clover做成dmg，或grub chainloader boot0ss文件或chainloader pbr文件的方案），然而最新的clover.iso在qemu（+pc-i440fx+hd or virtio）时，会在引导clover时进入一个头部带有上述dmidecode部分信息的图形化虚拟UEFI BIOS 启动界面（这实际上是没有发现盘，因此无法加载EFI文件夹，因此无法加载/EFI/BOOT/BOOTX64.efi这个bios启动文件，在图形化efi界面文件浏览，你也根本看不到盘符），这实际上跟clover iso版本采用的boot也有关系（clover iso有一个cdboot，这相当于clover install较旧版本的默认boot，clover install较新版本没有默认boot，只有多选方案，但也只有二个，一个boot6,一个boot7，Clover ISO cdboot集成的默认是boot6，boot6,7启动效果不同），跟盘性质也有关系(一般clover更好支持bios+gpt+普通ide+fat32，当然它也承诺支持bios+mbr+hfs+)，跟你的硬件/qemu也有关系(除非你编译qemu2.1时加了CONFIG_VIRTIO_BLK_DATA_PLANE，否则virtio在bios下是无法识别的，grub2 insmod hfs hfsplus part_apple ->memdisk>clover.iso也不能把这种效果传递给clover。)

我们最终选择是Clover-v2.4k-4630-X64.iso，分别在实机下测试和云上测试，盘性质为普通ide hd，且EFI位于这个分区内，可以进入到clover选单界面，不受支持的virtio就只能停在图形化efi界面，如何实现让其在virtio或云机上也能用呢？。

我们直接dd把文件弄出来，也不用去编译clover的源码了。 Cdboot:452,608-boot6:450,048=2560字节Byte，512Byte是1簇，是5簇 把cdboot与boot7放一起：然后 dd if=boot7 of=cdboot seek=5 bs=512 conv=notrunc Fat32 BOOT根分区是不需要放BOOT文件的,只需要把EFI从ISO中复制到这里来，其实移动也可。(对比变色龙，EFI就相当于其Extra) 然后mbrpatch cfg用变色龙那套Remaster Clover ISO

测试可以启动。以后我们就变色龙和四叶草一起测试了。如果你想测试clover install pkg，或得到相关文件，在10.12,13虚拟机guest osx中进行，，复制过去osxkvm10146然后挂载这个raw image,安装到虚拟出的boot区得到相关文件。，不要在host osx中安装（有保护也装不上）也不要在guest osx本机中安装。如果你想替换clover.iso，直接在host osx端挂载raw image编辑boot区放入新调的clover.iso(修改完成不需要卸载，pd端就能反应出变化)，用不着一台具有网络能力的qemu tcpe了。

新的libvirt xml和boot.plist

我们为什么deprecate前文qemu4.2的思路，前面的增删指令集思路没用，因为一个CPU涉及到架构，不是简单的指令集的叠加。因此我们从别的方向前进。我们用上了libvirt xml和vnc，我们使用libvirt是看中了它能自定义cpu，这样我们可以坚守在2.1下，一步一步自定义cpu成skylake或Cascadelake:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  <name>osxkvm</name>

  <os>
    <type arch='x86_64' machine='pc-i440fx-2.1’>hvm</type>
  </os>

  <features>
    <acpi/>
    <kvm>
      <hidden state='on'/>
    </kvm>
  </features>

  <cpu mode='custom' match='exact' check='none'>
    <model fallback='allow'>Penryn</model>
  </cpu>

  <memory unit='KiB'>4194304</memory>

  <devices>

   <emulator>/usr/bin/qemu-system-x86_64</emulator>

   <input type='keyboard' bus='usb'/>
   <input type='mouse' bus='usb'/>

   <video>
      <model type='cirrus'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0'/>
    </video>

    <graphics type='vnc'/>

    <disk type='file' device='disk'>
      <driver name='qemu' type='raw' cache='writeback'/>
      <source file='/media/psf/DATA/aliyunosx/osxkvm10146'/>
      <target dev='vda' bus='virtio'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0'/>
    </disk>

    <interface type='bridge'>
      <mac address='52:54:00:c9:18:27'/>
      <source bridge='virbr0'/>
      <model type='virtio'/>
      <address type='pci' domain='0x0000' bus='0x0' slot='0x03' function='0x0'/>
    </interface>

  </devices>

</domain>
```

网络还是用回文章1的tap0而不是使用helper，因为我们想免去boot.sh

```
sudo ip tuntap add dev tap0 mode tap
sudo ip link set tap0 up promisc on
sudo brctl addif virbr0 tap0（这句跟xml中bridge chunk中的target作用重复,如果不删掉<target dev='tap0'/>你每次都需要sudo brctl delif vi
rbr0 tap0,PD一旦休眠或暂停，tap0会自动消失因此这个绑定作用失效需要重新执行然后把virbr0停止再开一下，所以测试过程中要把pd中deepin设为永不休眠，否
则qemu tcpe网络不稳定)
```

继续作源码修改强化变色龙的调试功能：

因为10.14开始已经不支持32 sdk了。所以最新可用的10.13，PD直接application下的安装app直接做成镜像，过程中要产生一个cdr，这个原理就是我们在文4中提到的，其实可以手动直接createinstallmedia产生供pd使用的可安装镜像,只是在10.15不能执行32位createinstallmedia，制作好的安装程序已损坏，暂时更改主系统的time，例如date 102516242016回车,并禁网络直到安装完成(guest中这样做比较麻烦第一阶段可以过去，但第二阶段选不到菜单调不出命令行，过不去)

10.13支持的是Command Line Tools (macOS 10.13) for Xcode 9.2.dmg，它也没有10.11 command tools下不支持c99 vsscanf等函数发生“ld vsscanf : symbol(s) not found for architecture i386”等的问题:

graphics.c中Bouncing ball .oOo.把动画图标换成文字小棍。prompt.c中加个by 自己的标识 前面说到变色龙的bdmsg就是信息内容复杂点的-v + pause()，它其实是#define DBG(x...) msglog(x)，可以改动把它保存在可访问的fat32 boot分区内。

sys.c long GetFileInfo，可以得到文件名，用于hfs.c中提取更简单的文件名，你也可以在hfs.c中用vsscanf正规提取*.kext的部分，-v最后出现的加载drive，只要在启动时命令行中同时指定UseKernelCache=No才出现，变色龙默认使用prelinkedkernel，也就是usekernelcache=yes(-f=Yes)，此时不加载s/l/e，一定要指定usekernelcache=no才加载

在qemu src的pcbios下，我们找到acpi-dsdt.aml(i440fx),q35-acpi-dsdt.aml，变色龙也支持DSDT，因为变色龙逻辑中只写了bt(0,0)而我们的dsdt.aml在en(0,0)中，所以丢到extras还不够，还要在boot.plist中写一条key DDST,string dsdt.aml

其实变色龙也支持和clover一样的自定义kernel patch。在kernel.plist中写KernelPatches,blaaa..

二个boot，调试cpu尝试让mojave在Skylake-Server下运行解决黑屏问题

Clover和变色龙下，cpu设为Penryn可进入我们前面做好的镜像。但换cpu就进不去。这也在意料之中，cpu不兼容。继续看一段关于处理CPU兼容的问题，你可以三者全实现也可以保持不冲突部分实现，只要能最终启动mojave：

Why Penryn is recommanded beforehttps://forums.unraid.net/topic/84430-hackintosh-tips-to-make-a-bare-metal-macos/:

After digging a lot of code, I have some conclusions why they recommanded Penryn as prefered CPU Model:

1. Penryn is classic, and it missing some features compared to newer generation, which lead to a similar situation with VM.
    i. Penryn do not have a msr 0x00000198 leaf to read the perfstatus (like bus ratio, cpu frequency) which the same as a VM.
    ii. Penryn do not have a msr 0x35 leaf to read topology structure, which also most hypervisors haven't implemented yet.Instead, the MacOS will try to get the topology from acpi when it detect a Penryn process, which the same as a VM.
    iii. Some bootloaders do have some compatibility issues when using a newer generation in a VM, some causing dividing by zero errors(They can't get correct frequency from acpi or msr, so they may be zero).
2. Some articles have outdated so long from now.
    i. It don't have some cpuid features like avx/avx2/bmi/fma so MacOS won't recognized those features even thought you just passed through.

osx下的cat /proc/cpuinfo：sysctl machdep.cpu，sysctl -a | grep machdep.cpu，sysctl -a | grep hw.optional

更多在clover和变色龙下patch cpu的选项和功能：https://github.com/AMD-OSX/AMD_Vanilla/blob/master/17h/patches.plist：

```
<string>algrey - commpage_populate -remove rdmsr</string>
<string>algrey - cpu_topology_sort -disable _x86_validate_topology (10.14.4+)</string>
<string>algrey - cpuid_set_generic_info - disable check to allow leaf7</string>
<string>algrey - cpuid_set_cpufamily - force CPUFAMILY_INTEL_PENRYN</string>
<string>xlnc - cpuid_set_cpufamily - force CPUFAMILY_INTEL_PENRYN</string>
<string>algrey - i386_init - remove rdmsr (x3)</string>
<string>algrey - tsc_init - remove Penryn check to execute default case</string>
<string>algrey - tsc_init - grab DID and VID from MSR</string>
<string>xlnc - tsc_init - grab DID and VID from MSR</string>
<string>algrey - tsc_init - skip test and go get FSBFrequency</string>
<string>xlnc - tsc_init - skip test and go get FSBFrequency</string>
<string>algrey - lapic_init - remove version check and panic</string>
<string>algrey - lapic_interrupt - skip checks and prevent panic</string>
<string>xlnc - lapic_interrupt - skip checks and prevent panic</string>
<string>algrey - mtrr_update_action - fix PAT (x2) (10.14.4+)</string>
```

很不幸，这篇文章依然不是《最终实现在云机上运行黑果》，依然是前文的继续，我们的目的只是把osx装上云当个nas，代替《聪明的Mac osx本地云》文使之成为iphone,osx的nas mate os，可能这种尝试有点付出太多了。但无论如何，读到这里的小伙伴都赢了：目标已经十分接近了。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

云主机装黑果实践（8）：利用clover，离我们的目标更接近了

本文关键字：**clover** 黑底白苹果 无进度条**,clover white screen,clover legcay bios video, clover force vesa mode, prevent Mojave switch video boot while booting,Resolution for General Protection Exception/ X64 Exception Type 0D Error**

在文章7中我们讲到了使用clover来代替chameleon，因为似乎进行到上一文chameleon已经遇到了瓶颈，1，其调试功能严重不足，遇到黑屏，我们只能大致猜想是CPU不兼容或显卡乱配问题，不能从更细粒度上去确定问题所在，指导接下来实践，2，变色龙只有一种模式，即bios，通过这种方式作fake efi来（填充满足需要efi引导相关结构的OS数据块）引导黑果，因此对于现实主机/云主机功能上受制于从legcay pc的层面去解决问题，功能单一，（比如驱动）靠爬贴能得到解决问题的机会少。对于这二个问题，clover都有解决，clover可以从bios,csm,纯uefi去解决问题，提供黑果功能，有更多的驱动乱配和具备更多解决驱动带来的引导问题的方向，其调试功能一流，直接集成为工具。本文详细描述在qemu标准云主机型号（文章6，7所述命令行和libvirt版本i440fx虚拟机方案）上利用clover5070 iso引导运行黑果mojave。

前文简单描述了利用boot7做cdboot的思路，这里详述，我们利用变色龙源码来生成clover的cdboot7，注意到变色龙的srccdboot下有$(SYMROOT)/cdboot，我们复制这一段为：

```
$(SYMROOT)/cdboot7:
    @echo "    [NASM] cdboot.s"
    @$(NASM) cdboot.s -o $(SYMROOT)/cdboot7
    @dd if=$(SYMROOT)/boot7 of=$(SYMROOT)/cdboot7 conv=sync bs=2k seek=1 &> /dev/null

    @# Update cdboot with boot file size info
    @stat -f%z $(SYMROOT)/boot7 \
        | perl -ane "print pack('V',@F[0]);" \
        | dd of=$(SYMROOT)/cdboot7 bs=1 count=4 seek=2044 conv=notrunc &> /dev/null
```

all embedtheme optionrom: $(DIRS_NEEDED) $(SYMROOT)/cdboot后面加一条$(SYMROOT)/cdboot7,make dist前把clover的boot7放进去$(SYMROOT)，cd到src,cdboot下，sudo make all，就会发现symroot下有cdboot7了，使用这个cdboot,就加载了biosblk driver，在virtio blk下可以看到fat32中的EFI,cdboot6不行。

然后准备5070clover iso的config.plist,kexts和drivers下不动任何东西(5070 iso，kexts/other下有一个fakesmc,drivers/bios下有apfs,audiodxe,fsinject,ps2mousedxe,smchelper,usbmousedxe,xhcidxe)：

熟悉clover的调试信息与界面,虚拟机调试进系统

Clover的原则是尽量少添加参数，不懂的或不能确定其意义的不要添加(与chameleon一样他们被集成设计成尽量不必添加任何非必要参数就能启动一份常见系统，除非真的必要)，我们一步一步来，写出这个config.plist，原则是保留尽可能多的debug。而且与前文写的变色龙chameleon.boot.plist尽量一一对应，这样好理解，最终我们写成这样：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>

    <key>RtVariables</key>
    <dict>
        <key>BooterConfig</key>
        <string>0x28</string>
        <key>CsrActiveConfig</key>
        <string>0x67</string>
    </dict>

    <key>GUI</key>
    <dict>
        <key>ScreenResolution</key>
        <string>1024x768</string>
        <key>TextOnly</key>
        <true/>
        <key>ConsoleMode</key>
        <string>Min</string>
    </dict>

    <key>Boot</key>
        <dict>
                <key>Arguments</key>
                <string>Kernel="kernel" "Graphics Mode"="1024x768" "Boot Graphics"=No "Text Mode"=Yes -v debug=0x100</string>
        <key>DefaultVolume</key>
        <string>OSXKVM</string>
                <key>Timeout</key>
                <integer>10</integer>
```

```
            <key>Debug</key>
              <true/>
        <key>Log</key>
        <true/>
        </dict>

    <key>SystemParameters</key>
    <dict>
        <key>InjectKexts</key>
        <string>Detect</string>
        <key>NoCaches</key>
        <string>No</string>
    </dict>

    <key>KernelAndKextPatches</key>
    <dict>
        <key>KernelToPatch</key>
        <array>
            <dict>
            </dict>
        <array/>

        <key>Debug</key>
        <true/>
    </dict>

    <key>SMBIOS</key>
    <dict>
        <key>BiosReleaseDate</key>
        <string>12/22/2016</string>
        <key>BiosVendor</key>
        <string>Apple Inc.</string>
        <key>BiosVersion</key>
        <string>IM142.88Z.0118.B17.1612221936</string>
        <key>Board-ID</key>
        <string>Mac-27ADBB7B4CEE8E61</string>
        <key>Family</key>
        <string>iMac</string>
        <key>Manufacter</key>
        <string>Apple Inc.</string>
        <key>Manufacturer</key>
        <string>Apple Inc.</string>
        <key>ProductName</key>
        <string>iMac14,2</string>
        <key>SerialNumber</key>
        <string>C02KK5W9F8J2</string>
        <key>Version</key>
        <string>1.0</string>

        <key>Mobile</key>
        <false/>
    </dict>

</dict>
</plist>
```

解释一下：

开头，RtVariables是全局的开SIP，任何黑果引导的先决条件，为了好理解，所以把它写最前面，GUI和BOOT是菜单选单界面配置（当然，这里也有本质上的引导参数的喂给逻辑），引导后出现在你面前的就是这个界面，所以写在第二前面，GUI段面向clover作preboot用，BOOT段面向darwin接手后booting用，注意到我们加了尽可能多的debug支持的情况下，还尽量使用的是文字界面，和preboot/darwinboot一致的分辨率，这是为什么呢？

因为chamleon或clover引导时，要么用图形要么用文字，chameleon有纯文字界面而clover只有图形模式（文字界面是图形模拟的textonly,consolemode都是图形），mojave在启动时会有二个阶段不断换分辨率，这所有的过程，都涉及到分辨率带来的显卡驱动对启动过程的影响讨论，有很多花屏，黑屏，包括前面我们说到的云机上黑屏，都可能是显卡驱动导致的，第一要素就是分辨率（vesa显驱用vesa mode）：

vga std显卡vesa下mode 800x600时，clover preboot没有进度条，这就是上面GUI段手动设为1024x768以上的原因,猜可能就是800x600与匹配的vesa分辨率任何一种都不符，，逻辑上就不让过去了(要进入dmsg工具查看log中起作用的vesa mode才能发现原因可是进不去就没法使用dmsg)，为避免显卡问题在preboot时就halt掉我们的调试过程。这里应该尽量使用保守的模式（一般clover和变色龙够智能会帮你选好一般不用干预），进去mojave时，受到virtualgraphic.kext的支持，分辨率有更多选择，系统也会自己换分辨率。

所以原则上可以尽量手动保持preboot和darwinboot stage1,stage2二过程，这三个过程尽量使用一致的分辨率（如果有办法的话，比如修改源码也行，chameleon->boot2->graphics.c static int setVESAGraphicsMode中默认将vesa设为280，我将它改为267 https://en.wikipedia.org/wiki/VESA_BIOS_Extensions#Modes_defined_by_VESA，注意，vesa3支持刷新率定义但不要用，以免造成意外），这样的

预先处理可以免去很多问题。所以BOOT段OS显驱发挥作用时再次喂给prelinkedkernel同样的显示模式。注意到我还喂了尽量少用graphics mode和使用text mode的参数都是为了规避可能的显卡问题(喂给kernel as flags的照样可以设置在chameleon的boot.plist中)。

此时，直到BOOT的所有配置只能完成vga std下的preboot，接下来的 SystemParameters段有大讲究，也是显示相关的，本来启动darwin后的stage1很顺利，开始切换分辨率到stage2时，花屏过不去了，后来加了虽然也会花屏，但一会就过去了，实际起作用的是SystemParameters段，我在里面加了injecexts和nocache，但实际起作用的是SystemParameters空体这句，不是里面的任何东西。而且好像它的位置也要放在下面一个平等位。比如它放在SMBIOS下面一个位就没用。而且这个空体，对vga cirrus下clover preboot也有用，不加这个空段或放到不适当位置时，vga cirrus在clover preboot时显示End random seed ++++ 白屏，而且一直没进展，加空段放到BOOT对等位下时，也是白屏，但一会就进入到stage1,stage2。直到这里为止，已完成clover的引导进入到了darwin的verbose，虽然花屏或白屏，但最终进入系统，这说明这二种显卡的vesa都生效了,cirrus进入后是800600,vga进入后是19201440，只是进入前都有瑕疵一个开始花屏，一个后面白屏(对比系统报告发现不同,clover下bios变成imac10.1，chameleon保持了按原样14.2,clover下smbios有好多是可以自动计算的，写法上可以忽略)

注意到boot段中的Log true似乎并没有生效。不如直接把dmsg.efi弄出来为tools直观,好像LogEveryBoot有用。kernel听说如果不加入此参数"Kernel=..."，将默认加载系统缓存（kernelcache）启动，作用等同于启动菜单的"without kernelcache"选项。但似乎实测证明也不正确。SystemParameters那二段(clover下没有UseKernelCache=yes，这二段组合使用模拟了类似的效果，但似乎SystemParameters二条怎么调clover都是用prelinkedkernel,换成kernel cache这个kernel flags倒是可以https://www.insanelymac.com/forum/topic/99891--/不过没必要用)，prelinkedkernel是自动-f生成或手动使用 kextcache 将 kernel 和 kexts from /System/Library/Caches/com.apple.kext.caches/Startup/kernelcache共同 link 成 PrelinkedKernel，由System/Library/CoreServices/boot.efi启动，boot.efi 是平台特定的 EFI Driver(application), 由cloverx64.efi或真实系统的EFI 固件加载, 主要作用是加载 Kernel, 并将控制权移交给 KernelEntry.boot段中的kernel flags都是喂给/Library/Preferences/SystemConfiguration/com.apple.Boot.plist再给kernel或prelinkedkernel的。但是不要直接修改它。会造成文件系统损坏，修改这个文件很容易破坏镜像。可用clover在内存中patching。

上述启动过程中的花屏白屏，可能需要专门的驱动和patch过程来解决（vs clover，chameleon在这方面没有优势，这就是我们选clover的理由，clover可有csmvideodxe,qemuvideodxe，不过云机不能用这些，因为csmvideo需要开csm，qemuvidedxe需要开uefi，clover在mbr机上并不可以纯uefi，因此也不能使用uefi drivers，比如CsmVideoDxe-64.efi:Clover图形界面的图像驱动，可以有更多的分辨率选择。（仅限于启动界面）。他基于UEFI BIOS的CSM模块，因此需要CSM可用。UEFI GOP VBIOS只适用于UEFI引导的系统的,即UEFI引导的Clover和Ozmosis,用传统BIOS引导系统的同学请绕行）。不过不对引导造成问题，我们就不管了。

搜索花屏白屏，在网上可以爬到很多贴，单一个黑屏就这么多可能和可能的解决方案。也是初学者学习黑果第一大拦路虎，这里有趣也最容易让人弃坑，黑苹果文化博大精深，到底都是方案，可是不一定适用你那套。

我们可以在GUI段通过custom和scan将dmsg.efi工具形成菜单。查看debug log就更快捷了。跟Uefishell一样，Uefishell,就是把启动脚本化，而且是在线脚本语言化，跟grub一样，接上shell，都可以。但可运行一些.efi程序,load一些驱动，cp,ls一些文件，注意与cloverx64分开，它不可用于引导OSX，cloverx64才是osx lister和OS X booter efi(boot6,7 is the booter itself)，osxclover提供了二个efi booter，其实都是同一个文件，就是bootx64和cloverx64，如果充分利用CLOVER，甚至可以打造装机上的"统一实机云主机firmware"和统一BOOT环境,让bootx64是总boot。也许我们以后要把各种OS（包括boot6,7）做成boot+pbr，做成各种os的loader文件，让所有fat32 boot下的所有boot都呈一个样归进EFI.

云主机上的clover通用保护错误和黑屏问题

如果说实机上的白屏，花屏不影响最终启动，那么云主机上的黑屏就是一个实实在在的先决问题了。这之前，还有一个错误，就是cloverx64在云主机上执行会报错General Protection Exception/ X64 Exception Type 0D Error。这是clover层面的导致出错。但也有可能是显卡导致的。The reason is the incompatibility between UEFI graphic drivers & OS（or boot loader）.cloverx64只是相当于Os lister+它可以引导osx,Cloverx64一开始在preboot就用了显卡驱动，不像变色龙在preboot用的是text console。由于大量使用vesa，所以它访问graphics driver时会发生type 64 exception通用保护错误。而变色龙往往在完成preboot启动后，darwinboot接手涉及到大量vesa时才会黑屏，既然从clover中得知，黑屏可能不是cpu而是从qemu模拟的vga（vgabios inside seabios）到os显卡驱动导致的，本地就不黑，云主机就黑，clover也是切换到这种图形模式就无法输出了，那么这样这种某种cpu保护错误还与显卡问题同源。

那么，最终这是CPU兼容问题还是第一节显卡问题在云主机特殊环境上的新问题？如果我们在chameleon有限的debug下看不到，那么在clover下丰富的debug支持下能不能看到问题的蛛丝马迹呢？

不过，可以肯定的是，clover下解决问题尝试的方向会有很多，在源码定制上，它基于omvf实现，有很多开源驱动可以定制，摆弄，众多黑果开发者维护了很多驱动成果。可寻求解决问题的尝试路径会更多。

让我们试目以待。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

云主机装黑果实践(9)：继续处理云主机上黑果clover preboot问题

本文关键字：**clover fake display，unbound variable python_command, manual build compiling Clover from source,EDK2_REV,mac Merge lines between two patterns using sed, gnu sed vs bsd sed，sed**同时匹配多个模式**1**管道符方式**2**多**e**选项**3**分号连接**4**大括号一次执行,**sed**同时匹配多个模式异与或模式，**sed**模式空间匹配多个模式循环内欠结构模式空间

在前面，我们确定了一切重新开始从clover突破的思路，由于能驱动mojave的最低是4514+，因此，我们最终希望得到一个4514+ 云机上能过preboot阶段的clover(vs chameleon，clover分开了bootstrape和firmware，其核心部件不是boot6,boot7而是cloverx64.efi，clover的boot是已编译好的就在源码包里无须编译得到且不常变，得到编译结果后，我们只须保持boot和前面文章中的镜像文件中的大部分不变，替换cloverx64.efi就可以测试)，能从souceforge clover project files栏直接下载到的clover版本是断续的，实测过preboot的版本是3799其下一个版本3811开始的稍高版本在轻量云机上就报type x64 0D通用保护错误。而且，上述能下载到的二进制版本，其说明文件或执行结果中(clover about menu)除了自身并没有关于任何其它源码组件，比如edk2的rev号，而edk2是clover的一个大件，所以这给我们判断异常发生在那层对应关系的哪个部件中带来了麻烦 ，比如我们无法判断下载到的3799可能并没有使用其发布日最新的edk2,type x64 0d是否主要由cloverx64中的edk2源码部分导致而来。—— (尽管如此，我们在其论坛中https://www.insanelymac.com/forum/topic/304530-clover-change-explanations/发现了一些信息，比如，Rev 3264 Compatibility with new EDK2 18475 (and more?).Rev 3338 Compatibility with recent EDK2, because of changes in Autogen.py build tool.Rev 3471 ，It is now possible to build Clover with -xcode5 using versions of Xcode prior to 7.3 - as long as DevicePathToText.c is patched from Patches_for_EDK2.Rev 3952 New patch_for_edk2 rev 23215+ preventing load efi drivers and applications.With this patch we can use recent EDK2 at least 23447.)。Rev 4331 : 使用25687，REV 4496 : Synchronized Patches_for_EDK2 with r27233,Rev 5068 : Clover at sourceforge is synced to EDK2 tagged as edk2-stable201908 https://github.com/CloverHackyColor/edk2/archive/edk2-stable201908.zip，Rev 5104 Clover switched to C++ programming language

PS：edk2与ovmf与clover的关系。 EDK, EDK II是intel关于uefi实现:tianocore的一个开源实现。clover是作为edk2的一个子组件存在编译出来的，clover用它作虚拟uefi界面与引导osx的主要工作。OVMF "is a project to enable UEFI support for Virtual Machines，面向虚拟机的。clover就是使用它的。BIOS/CSM,EFI,UEFI,现在的UEFI就是它的升级版本，即：tianocore->edk2->ovmf（UEFI对qemu的实现）->clover。clover中有一个rEFIt_UEFI，应该是clover引导osx的中心模块，而edk2是用来提供firmware的中心模块。

clover也是一个十几年的项目了，早期clover的编译默认同步当时最新的edk2，后期clover edk2对应已经时间上脱节很多了。且都移到了github了，只能凭有上述insanelymac上clover change explanations上记载的来确定版本对应关系。最终我们要得到一个接近前文用的5070左右过preboot的编译结果，且修正上述不保留revs对应的版本，只能像当初处理chameleon一样，从源码级编译出一个能在云机上克服preboot就出问题的版本，我们只能亲自动手，手动控制这二个rev关系，从编译中寻求答案。

准备工作

早期clover的编译http://clover-wiki.zetam.org/Development已挂掉，其步骤是把对应版本的cloverefiboot-code-rxxxx改名为Clover放到edk2源码根中，然后在这二套源码中交替运行脚本，先编译全局性的toolchain,mtoc,nasm,gettext,extras等，然后update clover下的 patchs到edk2根，最后在clover下运行编译脚本，带动edk2下的编译脚本。实际结果cloverx64.efi生成在edk2/build/具体toolchain命名的文件夹结果中(我们并不需要全套clover，直到make pkg)。以上这是流程，工具链方面，Currently, 01.02.2016 it is possible to compile full Clover package only with gcc-4.9.Since April 2016 the best compilation is possible with XCODE5 toolset, so no more gcc-4.x needed.Tested on Xcode 4.4.1 and Xcode 7.3.1，Gcc49 toolchain被设定在linux上运行和osx下都可运行，但osx下默认Xcode，我们也使用Xcode。—— 而且我们不使用其它的第三方的整合编译方案build command（micky968,cloveglow,etc..）直接用源码包的。这样方便我们从0开始学习，下面开始：

我们先准备好虚拟机环境，文章1，文章4装的是10.11.4，10.12，文章7用了10.13，由于clover推荐xcode731,这里我们使用10.11.6搭配xcode7.3.1（注意非command tools），从10.11.6开始苹果安装包好像加了厚厚的验证，系统内部也加了sip，往PD中安装时很麻烦，跟文章7一样，果断关PD host网设好时间，一切重新开始，不过100%的时候卡了好久我强制重启，进入系统（如果你还碰到没有符合资源的软件包，安装时需要登录icloud之类，果断一切重来），但是我在10.11.6上安装xcode7.3.1失败，提示Xcode[45010]: DVTDownloadable: Failed to preflight installation Error Domain=PKInstallErrorDomain Code=102 "The package “MobileDeviceDevelopment.pkg” is untrusted." 又是时间问题，把控制面板中自动设置时间去掉，在客户机断网(使用服务使其不活跃)，Edit and set the date of your Mac as Jan 1st, 2016.或者设置pd客户机与主机时间不同步统统上，最后完成安装xcode。在mac下sudo 拷贝和删除文件时提醒Operation not permitted，无法显示隐藏文件，。是因为Max OS X El 中增加了rootless功能， 即sudo也不能操作部分文件目录， 利用configuration window -> Hardware -> Boot Order，boot into OS X Recovery Mode on Parallels Desktop,Enable Select boot device on startup，重启进入发现这也是一个虚拟EFI之类的东西，csrutil disable即可。

前面测出3811就报异常了，于是这里先从如下3810，22803源码组合开始，从https://sourceforge.net/p/cloverefiboot/code/3800/tarball?path=以及http://sourceforge.net/projects/edk2/22803/tarball?path=）下载源码包。解压到一个文件夹内。使用如下脚本先预处理脚本,这个updatesrcforonce.sh脚本放在下载解压后的edk2 src和clover src组成的文件夹并列的位置，修改下面开头的二个rev号，执行它，正确的结果是没有任何回显，不能多次执行，多次执行破坏源码重新解压并update即可，我都是在host OS X 10.15上执行然后将处理过后源码放进pd osx10.11.6编译：：

```
#下面脚本工作在大约rev 4000之前的脚本
export cloverrev=r3811
export edkrev=r22803

#(处理ebuild,保存revs完善源码缺失)
export reporev="repoRev\=\"0000"
export newreporev="repoRev\=\"$cloverrev,$edkrev"
```

```
sudo sed -i "" "s/$reporev/$newreporev/g" cloverefiboot-code-$cloverrev/ebuild.sh
# (处理ebuild,默认使用Xcode，即不使用toolchain仅nasm,与下面的toolchain gcc49二选一)
export nasmprefix='echo[[:space:]]*\"NASM_PREFIX'
export newnasmprefix='export NASM_PREFIX\=\~\/src\/opt\/local\/bin\/ \&\& echo \"NASM_PREFIX'
sudo sed -i "" "s/$nasmprefix/$newnasmprefix/g" cloverefiboot-code-$cloverrev/ebuild.sh
# (处理ebuild,替换toolchain为gcc49)
#export toolchaindir='\"\$CLOVERROOT\"\/..\/..\/toolchain'
#export newtoolchaindir='\~\/src\/opt\/local'
#sudo sed -i "" "s/$toolchaindir/$newtoolchaindir/g" cloverefiboot-code-$cloverrev/ebuild.sh
#（处理ebuild,替换edk2dir）
export edk2dir="cd[[:space:]]*\"\$CLOVERROOT\"\/.."
export newedk2dir="cd \"\$CLOVERROOT\"\/..\/edk2-code-$edkrev-trunk\/"
sudo sed -i "" "s/$edk2dir/$newedk2dir/g" cloverefiboot-code-$cloverrev/ebuild.sh
#(处理ebuild,替换platformfile)
export platformfile='Clover\/Clover.dsc'
export newplatformfile='\$\{CLOVERROOT\}\/Clover.dsc'
sudo sed -i "" "s/$platformfile/$newplatformfile/g" cloverefiboot-code-$cloverrev/ebuild.sh

#(替换patchs引用为edk2根，因为将来要update.sh)
export patchesdir='Clover\/Patches_for_EDK2\/'
sudo sed -i "" "s/$patchesdir//g" {cloverefiboot-code-$cloverrev/Clover.dsc,cloverefiboot-code-$cloverrev/Clover.fdf}
#(替换FLASH_DEFINITION)
export cloverfdffile="Clover\/Clover.fdf"
export newcloverfdffile="..\/cloverefiboot-code-$cloverrev\/Clover.fdf"
sudo sed -i "" "s/$cloverfdffile/$newcloverfdffile/g" cloverefiboot-code-$cloverrev/Clover.dsc
#(替换剩下的Clover\路径引用)
cloverrest="Clover\/"
newcloverrest="..\/cloverefiboot-code-$cloverrev\/"
sudo sed -i "" "s/$cloverrest/$newcloverrest/g" {cloverefiboot-code-$cloverrev/Clover.dsc,cloverefiboot-code-$cloverrev/Clover
.fdf}

#替换.inf,(下面命令无回显，要验证结果 需要  grep -rl "被替换的字符串" *，如果没有结果了,说明替换成功了，后来执行时如果build.py... : error xxx
tbuild error，也是这里没替换成功，好好检查)
export cloverpkgfile="Clover\/CloverPkg"
export newcloverpkgfile="..\/cloverefiboot-code-$cloverrev\/CloverPkg"
export grubfsdir="Clover\/GrubFS"
export newgrubfsdir="..\/cloverefiboot-code-$cloverrev\/GrubFS"
export fsinjectdir='Clover\/FSInject'
export newfsinjectdir="..\/cloverefiboot-code-$cloverrev\/FSInject"
sudo find . -name "*.inf"|xargs grep -rl "Clover/CloverPkg"|xargs sed -i "" "s/$cloverpkgfile/$newcloverpkgfile/g"
sudo find . -name "*.inf"|xargs grep -rl "Clover/GrubFS"|xargs sed -i "" "s/$grubfsdir/$newgrubfsdir/g"
sudo find . -name "*.inf"|xargs grep -rl "Clover/FSInject"|xargs sed -i "" "s/$fsinjectdir/$newfsinjectdir/g"
#替换.inf(for高级一点的版本)
export grubfsdir2="Clover\/FileSystems\/GrubFS"
export newgrubfsdir2="..\/cloverefiboot-code-$cloverrev\/FileSystems\/GrubFS"
sudo find . -name "*.inf"|xargs grep -rl "Clover/FileSystems/GrubFS"|xargs sed -i "" "s/$grubfsdir2/$newgrubfsdir2/g"

#修改-Werror(出错时使用)
#export werror='-Werror[[:space:]]*'
#sudo grep -rl "\-Werror" *|xargs sed -i "" "s/$werror//g"

sudo mv edk2-code-$edkrev-trunk/edk2/* edk2-code-$edkrev-trunk/
sudo rm -rf edk2-code-$edkrev-trunk/edk2
#如果手动在Finder中替换，要选择合并而不是替换。
sudo cp -R -f cloverefiboot-code-$cloverrev/Patches_for_EDK2/* edk2-code-$edkrev-trunk/
sudo rm -rf cloverefiboot-code-$cloverrev/Patches_for_EDK2
```

新建一个download放到与二个src root和update.sh并列，(离线下载好nasm-2.12.02.tar.bz2，修改buildnasm中的export DIR_DOWNLOADS=${DIR_DOWNLOADS:-$MAIN_TOOLS/download}为export DIR_DOWNLOADS=${DIR_DOWNLOADS:-$PWD/download}，在PD osx10.11.6中编译出nasm。

最好clover src下ebuild.sh，如下这几行注释掉：# Bash options set -e # errexit set -u # Blow on unbound variable(不去掉，高版本edk2会出现unbound variable python_command),下面就可以开始了。

编译测试一个直到**rev 5068**不报异常的**cloverx64.efi**

在PD osx10.11.6中cd到src root/clover src root，sudo rm -rf /usr/local/bin/mtoc,mtoc.NEW，sudo ./ebuild.sh cleanall sudo ebuild.sh(-mc不要了，因为我们的测试环境中cdboot就是-mc的,-mc是使用boot7，相当于ebuild.sh -h出来的-D USE_BIOS_BLOCKIO=1，默认没使用-gcc49，默认Xcode5，第一次执行不用执行cleanall，实际这个cleanall是个鸡肋，不要用)

ebuild.sh这是一个企图涵盖linux/osx的脚本，在linux和osx下都可编译使用gcc49 toolchain，(-h查看其参数，也存在一个对最终结果的配置过程)，执行后，clover源码作为一个组件被转到edk2编译过程，在那里会Initializing workspace，根据Clover src root/Clover.dsc(ebuild.sh默认-p Clover.dsc)定义的配置文件转到edk2 srcroot/build/Conf开始编译，最后生成结果cloverx.efi或clover64.efi到edk2 srcroot/build/toolchain named dir下:

如果出现tbuild执行错误就是上面准备工作的脚本没成功替换或重复替换了且preboot通过(通过tcpe上传cloverx64.efi到云机，测试,一般一个版本测试三次，因为有时能preboot的都偶尔会异常，跟chamleon尿性一样，这时直接重启服务器,冷启而不是从tcpe中reboot热启)，

3810 2016-10-14，edk 22803,通过编译和preboot，3811 2016-10-14，edk 22803，通过编译和preboot，利用半分思路测试，成功时大胆跨100，1000个rev，不成功时跟据最近一个成功视版本跨度取一半继续测试：3900，2016-11-02, edk 23078，没编译通过，提示 implicit declaration of function 'ARRAY_SIZE' is invalid in C99，看编译出错提示，在对应的文件中，把#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0])) 加进去(从22943开始)，编译通过，preboot也通过,4000 2017-02-06,23837，2.4k第一个版本,异常,3950 2016-12-01,23452，异常,3925 2016-11-17，23297，异常,3915 2016-11-09,23212，异常,3910 2016-11-07,23098，异常,3905 23091，异常

分析3901-3905，23079-23091异常，考虑主要是edk2 rev的变化导致的，现在用3901 23079-23091来测试,23088通过。23089通过。23090 通过,直到23091有了下面的变化。 https://sourceforge.net/p/edk2/code/23091/log/?path=：by edk2buildsystem: MdePkg/BaseMemoryLib*: check for zero length in ZeroMem () Unlike other string functions in this library, ZeroMem () does not return early when the length of the input buffer is 0. So add the same to ZeroMem () as well, for all implementations of BaseMemoryLib living under MdePkg/ This fixes an issue with the ARM implementation of BaseMemoryLibOPtDxe, whose InternalMemZeroMem code does not expect a length of 0, and always writes at least a single byte.

按上面的log，revert changes since edk r23091即可。我们发现3901-3906,23091可以通过preboot了，本地虚拟机也可。

继续从clover开始,往前几个版本又不行了，又黑屏了 3906 2016-11-07,23098 通过,但是preboot会重启 3907 2016-11-07,23098 ,23098 编译未通过,提示error: use of undeclared identifier 'CHAR_NULL'，，从这个版本23089更改了CHAR_NULL的定义位置。把提示找不到符号的源码加进去如下：

```
#define CHAR_NULL             0x0000
#define CHAR_BACKSPACE        0x0008
#define CHAR_TAB              0x0009
#define CHAR_LINEFEED         0x000A
#define CHAR_CARRIAGE_RETURN  0x000D
```

编译通过，preboot不通过异常。 这次是Commit [r3907] https://sourceforge.net/p/cloverefiboot/code/3907/log/?path= :update CPU definitions by ErmaC,这个commit提到的rEFIt_UEFI/Platform/下几个源码文件里有很多cpu model变化，我们记得变色龙的修正路子也是类似。要revert changes since clover r3907，把Platform.h中的define CPU_MODEL_SKYLAKE_S先删掉，然后把接下来一行CPU_MODEL_SKYLAKE_D改为CPU_MODEL_SKYLAKE_S，其它三文件StateGenerator.c,cpu.c,platformdata.c删除CPU_MODEL_SKYLAKE_D删除所有跟CPU_MODEL_SKYLAKE_D有关的orphaned case skylaked逻辑块。

5068是最接近我们上文5070的，现在直接来尝试编译5068+https://github.com/CloverHackyColor/edk2/archive/edk2-stable201908.zip，这里就不需要用上面的for 4000的预处理脚本了。revert changes since rev 23091也不需要，只需要手动处理3907的情况。toolchaindirs的逻辑还是需要处理一下的，手动去掉ebuild.sh中的nasm判断逻辑，修正其它（所幸我们不需要修正22943,23089的二段）直接改名为Clover放到edk2 src下，osx下手动粘贴合并非替换，手动工作量也小，另外，mtoc也需要编译了。手动准备mtoc，里面的是cctools-921.tar.gz，但是在osx10.11.6上会出错，用回cctools-895.tar.gz，指定xcode5,因为会默认xcode8，即sudo ebuild.sh -xcode5，

出来结果，在轻量上过preboot（表现是过preboot，但提示多处patchs，一段时间会重启，preboot后卡住无提示不重启/重启无提示/出错有提示，都大有讲究。可能都与源码中的cpu处理逻辑有关），本机虚拟机也可代替5070的cloverx64.efi用。

将tinycolinux以硬盘模式安装到云主机

本文关键字：**tinycolinux**安装到阿里云主机，**tinycolinux**实机硬盘模式安装**,vs Frugal tinycorelinux Scatter**模式安装**,**重新定制**tinycolinux**的**rootfs**

在《发布tinycolinux代替docker》一文中，我们将colinux和tinycorelinux结合，打造了一个tinycolinux并装在了windows host上，只要主机装了windows，那么实际上就可以装tinycolinux as guest，这对实机和云主机是无区别的，因为二者都可以装windows。

那么tinycorelinux，如何实现其在实机/云主机以standalone模式安装呢？

在《为tinycolinux发布应用中》我们提到tinycolinux的rootfs:microcore.gz，那里我们对它有一些优化意见，但在那里我们还不想定制它，那么现在我们要面临这个实际问题了。

测试livecd模式和寻址问题

实际上参照tinycolinux as guest for windows的方案和《利用tinycolinux在云主机上为linux动态分区》一文中安装grub2的过程，我们已经有思路了，即我们完全可以在云主机上创建一个包含microcore.cpio内容的grub2 as bootload的分区，然后参照windows host/colinux guest中利用vmlinux和microcore.gz的方式去尝试驱动它，实际上这是完全可能的。

我们先来说livecd模式安装，，即tinycolinux的frugal模式安装。因为在这个基础上可以一步一步很好测试以后的scatter模式是否能成功：

即按《利用tinycolinux在云主机上为linux动态分区》一文中利用virtiope+tinycolinux no image的方法分二个区，第一个区作为系统区并bootice刻上grub2的mbr，然后解压g2files.tar.gz，做/boot/grub的文件结构，把http://mirrors.163.com/tinycorelinux/3.x/archive/3.8.4/distribution_files/下到的bzImage和microcore.gz放进/boot，grub.cfg菜单就写成：

```
menuentry "tinycolinux" --unrestricted {
set root=(hd0,msdos1)
set prefix=(hd0,msdos1)/boot/grub
linux /boot/bzImage ro root=/dev/vda1 local=vda3 home=vda3 opt=vda3 tce=vda1         ;云主机发现2个盘
initrd /boot/microcore.gz
boot
}
```

这样是完全可以驱动进云主机的，但我们很快发现这始终只能让initrd中的内容成为根，进一步上传从microcore.gz中解压出来的microcore.cpio到/mnt/vda1/，cd /boot/，cpio -idmv < microcore.cpio,这时vda1中已经有可以工作的文件系统了，但是重启，去掉或保留那条initrd /boot/microcore.gz ，都不能使菜单中的root=/dev/vda1起作用（去掉会让云主机提示cant mount vfs as root，找不到盘启不动，而本来linux是可以不用initrd启动的.）。目前为止这样的livecd于主机来说不实用，由于livecd写入到根的东西都是占内存的，且由于一些未知的原因（我是不想追究了），我们发现GCC是无法在这种livecd中运行的。

这是因为vmlinux开机时发现不了virtio云硬盘，所以不能这样启动，不同于其在windows hosted的情况下可以在配置文件中直接定义/dev/cobd1=/dev/disk/partion,root=/dev/cobd1。

我尝试用bootloader grub来启动vmlinuz,即在grub.cfg中set GRUB_CMDLINE_LINUX="root=/dev/vda1 rootfstype=ext3"，同样发现不行，看来，

要寻求传统的硬盘根文件系统启动的方式，scatter模式，只能寄希望于先编译出一个支持virtio inside，能在开机时就能发现硬盘并挂载的vmlinuz：

编译virtio驱动模块到tinycorelinux bzimage/vmlinuz

我使用的版本是tinycorelinux 3.8.4，从http://mirrors.163.com/tinycorelinux/3.x/release/src/kernel/处下载config-2.6.33.3-tinycore和linux-2.6.33.3-patched.tbz2

由于在config中集成驱动，各个选项有复杂的依赖关系，是不能直接修改.config文件的。所以进make menuconfig，末尾加载那个config-2.6.33.3-tinycore，按一下/，输入virtio查看依赖关系，发现跟virtualization有关，好了，进入打开，如果你直接在network driver中打开virtio network的y选项会提示有依赖关系，block driver中的virtio block driver也一样，只有解决了依赖才能进行。

然后make mrproper（如果你进行了多次构建尝试，执行一个这个比较好）由于我在gcc481下编译的，所以vdso makfile会提示找不到i386等等，此时按《在colinux上编译openvpn》上处理的方法一样将里面的某句改成m32,m64，继续，得到bzimage在/arch/x86/boot。改个名放进livecd模式下的/boot/中，sudo reboot，在系统启动时进入grub命令行，改菜单，去掉initrd，用新的bzimage名代替bzimage，提示发现vda1，但又出现：runaway loop modprobe binfmt-464c的问题，无论如何，我们问题完成了一半。

网上说这可能是位数冲突，可能我使用编译bzimage的是个64位的ubt主机导致的，于是换回colinux+gcc461编译：

colinux下make meunconfig会用到term设置：export $TERMINFO=/usr/share/terminfo，且要安装ncurse和perl5.tcz，安装，重复make menuconfig继续编译得到bzimage，继续上传放进云主机/boot中测试，问题解决！！

然后就是那个定制**cpio**的问题了

其实这在硬盘模式下可以直接定制根文件系统逻辑了，对于打包的microcore.gz，则可以这样定制再打包，这称为remaster：

```
cd /mnt/vda1/boot/test
ls . | sudo sh -c 'cpio -oH newc -d > ../test.cpio'
不要在boot目录使用find ./test，会保留test
sudo gzip ../test.cpio
```

如果不用sh -c，会出现sudo之后依然无权限，我的busybox cpio是version v1.19.0，仅支持使用以上newc格式。。

其实，我刚一开始的解决方案是企图通过定制microcore.gz来改那个/etc/init.d/tc-config，把挂载到livecd根下的所有目录通过类opt,home,usrlocal,tce的方式挂载到目标硬盘，这样在有硬盘时使用硬盘，没有硬盘时就使用livecd。livecd模式也不是完全没用的，livecd模式也称为装机模式，，忘掉它云端模式的另一个名字吧我感觉没什么大用，只在这个装机模式下，它整个都在内存，所以即使tinycorelinux initrd所在硬盘可以拿来格掉/覆盖，且tinycorelinux比起virtiope来还有一个联网功能，云端模式的名字就来源于此，其实完全可以代替virtiope的所有功能，打造类mac电脑在线恢复系统的功能，我未来会把它做进diskbios--一个整合化的装机系统。

还有，linux kernel一个微小中心+shell脚本的多元发行设计使得其发行版很常见，运用到语言的设想就是用terralang这样的东西组装langtech级可定制组装/剪裁的发行版语言系统，这样就不需要聚集于传统的库方式，也不需要大量非C的脚本DSL了，当然，当这个terralang是terracling的时候就是这样，因为terralang中的lua是非C的。。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

为**tinycolinux**制作应用包

本文关键字：**tinycolinux**自定义应用包，**tinycolinx**内存运行，**mysql**重建**/tmp/mysql.sock**

在前面《发布tinycolinux》中，我们重点描述了将tinycolinux安装到硬盘的情况，及处理安装应用到硬盘的情况，这也是大部分情形下的场景，其实，完全可以采取其rootfs放在livecd ram中运行而应用依然安装到硬盘的方式，这样更有利于vm container iaas环境建server farm，这样rootfs是加载到ram中去的。只要重启，一切对系统的更改将撤消。用户就不会轻易破坏系统。

ram中运行**rootfs**

首先在conf/colinux.conf中root=/dev/ram0,initrd=microcore.gz,cobd0=/imgs/tinycolinux1g,这样启动起来的colinux其rootfs在/dev/ram0中，硬盘介质中仅用于保存用户数据，即conf/colinux.conf中定义的四个挂载地址/opt=cobd0,/home=cobd0,local=cobd0,tce=cobd0，这四个可持化挂载点是colinux那些当且仅当需要修改的地方，所以需要被挂载持久，，我们还可以再定义几个变量加强这个live rootfs的强度，如norestore,

启动，运行。成功进入到tc用户的cmdline.

当然，虽然这个live rootfs系统启动起来了，这个rootfs还是有点raw form和不便的。比如：

通过df命令我们发现定义的四个挂载点，仅挂载了三个到/缺了/tce，但是硬盘中依然生成了四个文件夹/opt,/home,/tclocal对应/usr/local，和/tce，通过tce-load -w发现下载的包在/mnt/cobd0/tce中，这是正确的行为，能用但不好看，这四个挂载点的加载逻辑全在/etc/init.d/tc-config中，所以我们甚至可以重新打包microcore.gz修改tc-config加入缺失的/tce条目。

modprobe也会出错，因为readonly live rootfs是不能加载原initrd.gz注入的lib/modules的，不过同样地，我们可以重新打包microcore.gz手动加入这些文件。

还有一些必要的系统级持久无法完成，比如用户密码更改，它保存在readonly rootfs /etc/shadow中，我们必须这样来完成：

sudo passwd root

输入密码二次

cp /etc/shadow /opt/shadow （做一次备份到硬盘中/opt）

然后修改下/opt/.bootsync.sh，加入以下：

```
cp /etc/shadow /etc/shadow_old
cp /opt/shadow /etc/shadow
```

其实我们完全可以替换busybox中的passwd，改变/etc/shadow路径到其它外部可持久位置，还比如，vm container子机环境不需要关机，可以去掉busybox中的halt，还比如我们可以编译加入dropbear支持，毕竟sshd是最基本的发行包支持了。

我们就不定制microcore.cpio包了。太累。

组建复合应用

官方提供了很多镜像，这些包都很正交。且还有构建源码，可往往我们还需要lnmp这样的组合包，我们可以按《发布tinycolinux》part2中的硬盘安装应用方法来组合一次性安装包（当然，这样它就不正交了但对一台vm container通常情况下仅需承载安装一次lnmp的情形下来说，非常合理和实用），以下是组合应用逻辑，举例我们用了lnmp，组合到一个lnmp.tar.gz中。

首先，tce-load -w nginx,php5,sqlite3，发现会下载大量tcz到/mnt/cobd0/tce/options中：bsddb.tcz,bzip2-lib.tcz,curl.tcz,gmp.tcz,libgdbm.tcz,libiconv.tcz,libltdl.tcz,libmcrypt.tcz,libpng.tcz,libxml2.tcz,libxslt.tcz,mysql.tcz,ncurses.tcz,ncurses-common.tcz,nginx.tcz,openssl-0.9.8.tcz,pcre.tcz,perl5.tcz,php5.tcz,readline.tcz,sqlite3.tcz，这些都是我们要组合进一个大应用包的基础。一个一个解压它到my文件夹，sudo unsquashfs -f -d /mnt/cobd0/my/ /mnt/cobd0/tce/optional/xxx.tcz

作一些更改（这是因为原tcz全是绿色dropin包）：

nginx conf/nginx.conf，root index加个index.php,把关于php的三条注释去除注释化使其有效，其中SCRIPT_FILENAME改成$document_root$fastcgi_script_name;且把最大脚本内存由128m改为64mb

usr/local/etc加个my.cnf，内容如下：

```
[mysqld]
socket     = /tmp/mysql.sock
port       = 3306
pid-file   = /tmp/hostname.pid
datadir    = /usr/local/var/mysql
```

```
language   = /usr/local/share/mysql/english
user       = tc
```

好了，现在重建数据库，sudo /usr/local/bin/mysql_install_db，，尝试启动mysql: sudo /usr/local/bin/mysqld_safe & ，成功

然后我们cd /mnt/cobd0/my，打包它们sudo tar zcf lnmp.tar.gz *，，，安装这个大应用测试下：cd到/，然后tar zxvf /mnt/cobd0/my/lnmp.tar.gz，然后在/opt/bootlocal.sh中启动它们：

sudo nginx；sudo php-cgi -b 127.0.0.1:9000；sudo mysql_safe

成功。

这个我本来还要集成memcached (bootlocal中启动用memcached -d 64m限制最大使用内存)和postfix的。postfix适合另外起一台vm container建一个emailserver的组合包。而不是放到lnmp中。

当然，如果自己要从源码构建php等的新版本，而不是直接利用官方包组合，这需要处理好多东西。恩恩

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

为tinycolinux创建应用包-toolchain和编译方法

本文关键字：**tinycorelinux编译gcc套件,live,vhd二合一colinux,tinycorelinux lnmp**

在前面我们提到，一个linux发行包只要提供了核心部分和cui的基础toolchain部分才算是一个基本完整的linux发行包，因为扩展将来都由这套toolchain编译而来。在《为tinycolinux创建应用包》中我们用简单解压组合tcz的方式组建了一个lnmp环境包(mysql5.1+php5.3)，在这里，我们准备为tinycolinux建立一个toolchain环境，并用源码编译的方式产生高版本的mysql+php的lnmp包,而这也是更通行和更灵活的办法。

> 关于编译新gcc套件及处理glibc移殖的问题
>
> 编译GCC可能面临二种需求环境：1) 从本地产生,比如你需要一个bootstrap的gcc低版本来产生高版本，2) 从外部crosscompile而来。
>
> 默认gcc第一遍只需要gmp,mpc,mpfr加gcc，这样--enable-language=c,c++编译出来的gcc支持stdlibc++-dev却不带libc-dev,甚至binutils都不需要，如果目标环境中没有支持是没有实用的。
>
> 完整可用的gcc套件要经过多遍，除了gcc,binutils，甚至还需要附加编译flex,bison这些，
>
> 最重要的问题来了：
>
> 默认gcc仅带libstdc++，这个可以后期添加新版本替换/叠加系统原有版本因为它是built into toolchain的，而glibc的版本是一个linux发行版rootfs中集成的built into rootfs，是最为基础的被引用部分，不可升级/替换,是一个不可移殖项。你需要另外准备平台依赖的libc-dev(glibc-dev)，这可能需要在其它遍次pass,phase的gcc编译中完成。
>
> 其实GCC也算是一个类kernel的复杂包了，ng-crosstools有用类kernel的menuconfig方式产生.config环境。

以下测试过程全在硬盘版的tinycolinux下测试，live版的不方便。请下载tinycolinux live hd一体包后继续：

组建bootstrap toolchain

以下tcz默认全是4.x的，从4.x的compiletc.tcz的meta包的dep中提取而来，以下底部部分eglibc_base-dev就是glibc开发包，glibc runtime已经在tinycolinux的/lib中了，底部其它的那些是可选开发包，因为比较基础都保留了，gcc为461版本，请手动从某个镜像的4.x/tcz目录下载这些包到/mnt/cobd0/tce/optional，然后在console-nt中直接paste执行以下脚本。

```
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/gmp.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/libmpc.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/mpfr.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/gcc.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/binutils.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/m4.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/make.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/pkg-config.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/bison.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/flex.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/gawk.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/grep.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/sed.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/patch.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/diffutils.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/findutils.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/file.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/eglibc_base-dev.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/gcc_base-dev.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/linux-3.0.1_api_headers.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/util-linux_base-dev.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/imlib2_base-dev.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/e2fsprogs_base-dev.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/fltk_base-dev.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/freetype_base-dev.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/jpeg_base-dev.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/libpng_base-dev.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/libsysfs_base-dev.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/zlib_base-dev.tcz
```

解压完毕后测试 gcc -v,g++ -v，可能你需要sudo reboot重启一次tinycolinux系统才能发现已安装但缺失包。

遵从以上谈到的编译gcc的难点，其实你完全可以用这套GCC461作bootstrap编译出新的GCC如gcc483 gcc491 etc..。以下我们用它测试编译新lnmp：

编译新lnmp

不可直接用lnmp.org的一键包，因为系统集成的工具扩展不一样，一般地，先编译mysql，再php，再nginx，这样php的--with-mysq=就可以引用到mysql的开发包了。以下包都跟上面一样下载来自4.x，整篇文章的tcz都来自4.x（注意cmake例外）：

1) mysql5.5

```
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/cmake
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/libidn.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/libiconv.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/ncurses-dev.tcz
```

5.5之后的mysql需要cmake。cmake.tcz是3.x的。4.x的cmake需要安装4.x的额外libstdc++，因此我舍弃4.x cmake选择了混合3.x的tcz。

ncurses nginx和mysql都需要（nginx需要运行库部分，mysql需要dev pkg部分）

mkdir b ; cd b ; cmake -DCMAKE_INSTALL_PREFIX=/usr/local/mysql -DWITH_MYISAM_STORAGE_ENGINE=1 ..

2) php5.6

```
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/curl.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/curl-dev.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/libxml2.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/libxml2-bin.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/libxml2-dev.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/liblzma.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/libssh2.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/openssl-1.0.0.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/libpng.tcz
```

./configure --prefix=/usr/local/php --enable-fpm --enable-zip --enable-mbstring --with-mysql=mysqlnd --with-pdo-mysql=mysqlnd --with-zlib --with-gd --with-curl

(我们选取了能安装owncloud需要的那些包,mysql就不引用实际编译的5.5的包了用了src自带的mysqlnd)

3) nginx1.11

```
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/pcre.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/pcre-dev.tcz
sudo unsquashfs -f -d / /mnt/cobd0/tce/optional/ncurses.tcz
./configure --prefix=/usr/local/nginx
```

以上编译过程中，如果解压发现不了实际已解压的引用包的，一般是一些含.so的包，需要sudo reboot重启一次guest系统

配置运行部分：

上面php和mysql显然没指定my.cnf和php.ini的目录，但它们默认分别都在/usr/local/mysql/和/usr/local/php/lib/php.ini，自己建2个即可,需要配置php.ini这二个文件，tz.php中才能显示smtp支持和控制php更多行为的那些选项如上传max upload size。

其实大多数可以参照《为tinycolinux创建应用包》中的做法，但还有一些附加处理部分：

mysql中新建一个tmp用来放mysql.sock，其权限要和data一样，都设为0755且归staff下的tc用户所有。这样mysql_install_db才能正确产生初始数据库+pid文件和mysqld_safe产生mysql.sock文件

启动的方法都可以在/opt/bootlocal.sh下加二条：

/usr/local/nginx/sbin/nginx ; /usr/local/php/sbin/php-fpm ; /usr/local/mysql/bin/mysqld_safe &

当然你还可能需要额外编译mta和redis这些，我发现网上chenall(grub4dos作者)有一个类似的tinycolinux的发行包，使用的是473。 它能把/home,/tce都挂载到本地share文件夹中，虽然权限不能继续，但在一定意义上，这是“共盘windows,linux”，这应该不是官方的支持的功能(vmlinux or initrd473)，只是不知道怎么实现的。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在tinycolinux32上装tinycolinux64 kernel和toolchain

本文关键字：高版本**gcc cross compile** 交叉编译低版本**gcc,boostrap,**为**tinycolinux**低版本**linux kernel**生成**gcc**，在**32**位**linux cross build gcc target for linux64 execution**，**32**位**64**位混合**rootfs**制作，运行**cross build**的应用。

在《为tinycolinux创建应用包-toolchain和编译方法》中我们谈到gcc作为一套完善工具链的中心（编译套件），它从源码级支持被boostrap构建，和被外来地cross compile构建，一般地，当制作一个linux可用发版时toolchain支持是必要的。当然它也是难于构建的，它难于被构建是因为它绑定了binutils,kernel,libc这样的东西。且这些东西分别分布于三个关联平台，即：build平台,host平台,target平台 ----- build,host,target是GCC的三元组，它表达了这样一种通用工具链产生流程：用a gcc编译出能运行在b上的gcc，且这个gcc能产生c上运行的代码，这里abc依次即为build,host,target，GCC自举统一使用build平台已有的binutils,kernel,libc，且通常build=host=target。这里没有任何涉及到先有鸡还是先有蛋的问题（当然第一个GCC肯定不是GCC产生的，这其实是个演化问题），但是一旦涉及到cross compile（cross compile正是通用流程，bootstrap build只是个例）这里的复杂关联就无穷无尽了：这三个平台的位数，CPU类型，OS类型，OS版本，都不相同，具体GCC也要求不同具体版本的bin和平台上的header,和来自平台调用到libc的封装，GCC产生的程序需要运行在配有当初与GCC一起产生的binutils中的LD的host平台中运行等,如此种种,etc..... 没有一个single gcc source tree that with binutils,kernel,libc in a boundle能覆盖这些(当然指定版本到足够相符程度还是可以的)，通常情况下，我们都是依赖配置生成具体三位组所指的GCC TOOLCHAIN。考虑到复杂性，这也是为什么GCC这样的基础套件一般被设计成极度selfcontained的-仅引用binutils,kernel,libc，在以前的文章中我们还谈到升级libstdc++的情况，但要升级libc这基本很难，这也是因为作为基础的它们越基础的东西绑定越紧的原故，就像kernel一样。

好了，在以前的文章中我们一直使用的是3.x的tinycolinux32，现在，我们编译tinycolinux3.x 64和其完善的toolchain支持。其中，我们会涉及到比较多的坑。

cross compile tinycolinux3.x 64

我们是在一台ubuntu14.04 64bit上的gcc485交叉编译出如下tinycolinux 3.x 64的(不要直接用tinycolinux上32位的gcc编译这个kernel):

参照《将tinycolinux以硬盘模式安装到云主机》一文的相似做法，我们从http://mirrors.163.com/tinycorelinux/3.x/release/src/kernel/下载64位的src和patch，打开virtualization中的virtio pci选项，编译进virtio block和network驱动，轻易在arch/x86/boot下得到bzimage，放到boot中启动。发现可以跟原有的rootfs一起正常启动。uname -m显示x86_64。file /boot/bzimage，显示x86 bootable kernel。猜这是因为在.config文件中同时开启了32和64支持，32位程序能运行在64位上，且原来的rootfs中的32位binutils和gcc未变。

如果把64位某linux的程序拷进来file它显示64bit elf，执行它会提示not found，这是因为它依赖的binutils ld没有，调用gcc -o helloworld.c -64m，提示unimplemented,这是因为3.x的rootfs是没有对应的GCC 64的。接下来需要cross compile一个：

在32 tinycolinux上boostrap compile gcc 4.4.3 64 三件套 for tinycolinux 3.x 64 target

一般地，GCC支持从高向低crosscompile，反过来要难一点。所以这里我们采取最简单的方法：从同版本的32位GCC bootstrap编译出同版本的GCC，采用本地的32位gcc bootstrap式cross compile出64位的gcc，不再使用外来cross compile的方案（直接那样也行）。当然这种方案是设想了tinycolinux上本来就存在GCC的事实基础上，如果追求更通用的实践目的，还是从外面的系统cross compile进来好。

这样产生出来的GCC仅是一个target到x86_64-pc-linux-gnu的gcc 443版本，因为在本机上构建，所以这个build和host都不变，为本机系统HOST，但是并不影响我们的工作继续，至于以后你要用这个GCC作鸡生蛋蛋生鸡的事，比如可以用这个再次自举GCC443到host也为443的版本inplace覆盖，这都是以后的事。GCC支持从32到64或反过来的交叉构建。

我们选用2.x repos的make.tcz(3.81版，为什么不使用3.x的make 382接下来会涉及到)和选用3.x repos的gcc443 32位(为什么不用4.x的gcc471：因为4.x后采用eglibc，在编译很多程序时会遇到重复定义错误，这个时候就应该想到是版本问题)，走从GCC443 32位编译出GCC443 64的方案，要保证系统绝对干净，否则可能会遇到各种坑(比如cant computer object file prefix,etc..)，介绍一下制作纯净tinycolinux系统的方法：

按《在硬盘上安装tinycolinux》的方法重新安装rootfs，相当于重装系统，除了保留第一步的64 bzimage在boot下引导不变，你可能需要额外安装openssh。然后下载3.x的toolchain并安装:

```
sudo unsquashfs -f -d / /tce/gccbase/gmp.tcz
sudo unsquashfs -f -d / /tce/gccbase/libmpc.tcz
sudo unsquashfs -f -d / /tce/gccbase/mpfr.tcz
sudo unsquashfs -f -d / /tce/gccbase/ppl.tcz
sudo unsquashfs -f -d / /tce/gccbase/cloog.tcz
sudo unsquashfs -f -d / /tce/gccbase/binutils.tcz
sudo unsquashfs -f -d / /tce/gccbase/bison.tcz
sudo unsquashfs -f -d / /tce/gccbase/diffutils.tcz
sudo unsquashfs -f -d / /tce/gccbase/file.tcz
sudo unsquashfs -f -d / /tce/gccbase/findutils.tcz
sudo unsquashfs -f -d / /tce/gccbase/flex.tcz
sudo unsquashfs -f -d / /tce/gccbase/gawk.tcz
```

```
sudo unsquashfs -f -d / /tce/gccbase/gcc.tcz
sudo unsquashfs -f -d / /tce/gccbase/grep.tcz
sudo unsquashfs -f -d / /tce/gccbase/m4.tcz
sudo unsquashfs -f -d / /tce/gccbase/make.tcz
sudo unsquashfs -f -d / /tce/gccbase/patch.tcz
sudo unsquashfs -f -d / /tce/gccbase/pkg-config.tcz
sudo unsquashfs -f -d / /tce/gccbase/sed.tcz
sudo unsquashfs -f -d / /tce/gccbase/base-dev.tcz
#sudo unsquashfs -f -d / /tce/gccbase/gcc_libs.tcz
#sudo unsquashfs -f -d / /tce/gccbase/linux-headers-2.6.33.3-tinycore.tcz
```

然后下载以下并准备，都解压到一个目录。

http://mirrors.163.com/tinycorelinux/3.x/release/src/compiletc_other/ (mpfr-2.4.2.tar.xz,gmp-4.3.2.tar.xz,binutils-2.20.tar.xz)

http://mirrors.163.com/tinycorelinux/3.x/release/src/glibc-2.11.1.tar.bz2 (不用2.11.1了，到ftp.glibc.gnu.com下载2.12.1，以后有用)

http://mirrors.163.com/tinycorelinux/3.x/release/src/gcc-4.4.3.tar.bz2(从GCC-4.3起，安装GCC将依赖于GMP-4.1以上版本和MPFR-2.3.2以上版本。如果将这两个软件包分别解压到GCC源码树的根目录下，并分别命名为"gmp"和"mpfr")

1）首先编译binutils:

```
cd binutils-2.20 && sudo make b && cd b
sudo ../configure --prefix=/usr/local/gcc443 --target=x86_64-pc-linux-gnu --disable-multilib
```

高版本GCC可加--disable-werror以免导致各种警告错误

sudo make sudo make install

2）然后导出linux头文件到工具链:

```
cd linux-2.6.33.3
sudo make ARCH=x86_64 INSTALL_HDR_PATH=/usr/local/gcc443/x86_64-pc-linux-gnu headers_install
```

要用到perl.tcz

3) 构建GCC工具框架，不带任何库。

```
cd gcc-4.4.3 && sudo make b && cd b
sudo ../configure --prefix=/usr/local/gcc443 --target=x86_64-pc-linux-gnu --enable-languages=c,c++ --disable-multilib
```

如果使用的3.x的make 3.8.2会出现configure错误：mixed rule

sudo make all-gcc sudo make install-gcc

4) 生成glibc的基础部分

第三步已经将工具生成了，现在最重要的基础库的基础部分，注意还不是整个glibc

预先export PATH=$PATH:/usr/local/gcc443/bin帮助接下来的configure找到新编译出的x86_64-pc-linux-gnu-gcc，虽然configure会自动找到，手动一下更保险

```
cd glibc-2.12.1 && sudo make b && cd b
sudo ../configure --prefix=/usr/local/gcc443/x86_64-pc-linux-gnu --build=$MACHTYPE --host=x86_64-pc-linux-gnu --target=x86_64-
pc-linux-gnu --with-headers=/usr/local/gcc443/x86_64-pc-linux-gnu/include --disable-multilib libc_cv_forced_unwind=yes libc_cv
_c_cleanup=yes
```

$MACHTYPE在正常的linux32上会输出i686-pc-linux-gnu字样，在tinycolinux上输出为空，继续

如上语句在tinycolinux上一次通过，但在普通linux上configure似首很容易把glibc源码目录被破坏，即使是cd 到b中，比如你也许会碰到：cannot compute suffix of object files或者： invalid host type: $CXX unregconnize -c，并不网上说的解决办法能解决的，往往重新准备glibc源码目录重新按上面的configure来配置就好了，在普通linux上，glibc源码目录下的scripts/gen-sorted.awk 19行以后会出现需要将√+$ 改成 √√+$的BUG，修正就好了。

然后就是make了：

a) sudo make install-bootstrap-headers=yes install-headers

在tinycolinux上一次通过，在普通linux上，你或许需要在make后额外加CFLAGS="-O2 -U_FORTIFY_SOURCE" cross-compiling=yes以分别应付下列可能出现的错误。

cross-compiling=yes : No rule to make target `elf/soinit.os' error CFLAGS=02 : glibc cant continues without opt error -U_FORTIFY_SOURCE : inlining failed in call to ‘syslog’ error

如果在不纯净的tinycolinux上执行a)，可能会出现需要tls support error

b) sudo make csu/subdir_lib

如果在不纯净的tinycolinux上执行b)，会继续出错

c) install csu/crt1.o csu/crti.o csu/crtn.o /usr/local/gcc443/x86_64-pc-linux-gnu/lib

d) x86_64-pc-linux-gnu-gcc -nostdlib -nostartfiles -shared -x c /dev/null -o /usr/local/gcc443/x86_64-pc-linux-gnu/lib/libc.so

e) touch /usr/local/gcc443/x86_64-pc-linux-gnu/include/gnu/stubs.h

如果在纯净的tinycolinux上，可以无误一直执行到e)，一般到这接下来二步都能完成。

5)生成GCC的LIBGCC

重新cd gcc-4.4.3/b sudo make all-target-libgcc sudo make install-target-libgcc

6)最后一步，生成完整的glibc和gcc中的stdc++lib

重新cd glibc-2.12.1/b sudo make sudo make install

重新cd gcc-4.4.3/b sudo make sudo make install

其实如上三部曲的编译还有很多联合构建的选项。但是本文不深究了。

测试编译**64**位程序并运行

先写一个CPP的helloworld,test.cpp

```
#include <stdio.h>
int main() {
    printf("Hello World");
    return 0;
}
```

然后分别/usr/local/gcc443/bin/x86_64-pc-linux-gnu-g++ test.cpp -o a,/usr/local/gcc443/bin/x86_64-pc-linux-gnu-g++ -static test.cpp -o b,file a,file b，发现都是64位程序，我们发现b可以直接运行，而a显示not found，跟文章开头说的没有64位的GCC和binutils一样原因，那么现，我们讨求用新编译出的工具链让它运行的方法：

其实原因就是找不到共享库，error cant find share libs, ELF64CLASS，我们不能用32位的LDD分析它的依赖关系，但我们可以cd a所在的目录，x86_64-pc-linux-gnu-readelf -a ./a | grep "Shared"或x86_64-pc-linux-gnu-objdump -p ./a | grep NEEDED的方式查看，发现它引用了libc.so.6(->libc-2.12.1.so),libstdc++.so.6(->libstdc++.so.6.0.13),libgcc_s.so.1,libm.so.6(->libm-2.12.1.so)，当然了，这些库都要从新编译出的工具链的lib或lib64中找，放到a的目录，然后在a的目录下写个runa，加起执行权限，内容为：

```
D=$(dirname $0)
$D/ld-linux-x86-64.so.2 --library-path $D:$D/lib:$D/usr/lib ./a $@
(ld-linux-x86-64.so.2也是从工具链中找到的，它其实可以被执行，你也可以定制上面的--library-path)
```

执行./runa，输出跟静态b一样的结果。

这从理论上说明，只要系统支持默认的ld-linux-x86-64，它就支持运行一切由这个新工具链产生的程序。

现在，64位的kernel有了，生成64位程序的toolchain有了（它本身还是32位程序只是也能处理64位生成的事），但是整个ROOTFS还是基本上32上的，连运行它生成的64位程序的事都管不了，应该要重新编译busybox让它支持新的64位ld。

还有，可以以这个TOOLCHAIN为基础，不断bootstrap高版本的GCC，或者inplace覆盖，或变动BUILD，HOST进行，产生新的toolchains.

关注我。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在tinycolinux上组建子目录引导和混合32位64位的rootfs系统

本文关键字：**mount subdirectory as linux root,boot linux from root subdirectory**，从子目录引导**linux root**，**separated system and usr extend under linux root**

在前面《在tinycolinux32上装tinycolinux64 kernel和toolchain》中我们讲到了组建一个linux发布版的二大基本部件：kernel和toolchain部分，虽然通常提到linux发行指的是一个包含了所有打包的linux ---- 体积外观上最大的主要是其rootfs部分，即那个/下的部分，，但往往kernel才是一个发行版的表征：它提供了能bootable起硬件使之变成OS的部分，它定义了PC能带起什么硬件，能支持几位程序的部分，基础之二的toolchain是支持这个表征放大的基础：它提供了用户能开发和运行应用以扩展这个OS的支持部分---- 它们属于从外而内产生一个linux发行版的必要和基础部分---------- 这二部分要先行从外部cross built而来，相对来说，rootfs只是kernel要搭配起哪些toolchain产生的程序完成什么工作的事，所以可以放在以后，甚至一个至简的rootfs就是一个busybox+一些init脚本就可以了。

本篇我们将讨论这个rootfs组建的过程，以测试运行前文产生的kernel和gcc toolchain产生的程序的过程。最终的目的，将会是一个支持64位/32们混合的文件系统，和一个高度自定义，system和用户扩展文件夹分开的，这样一个linux发行版。

> 这究竟会是一个什么样的LINUX呢？
>
> 现在的linux发行版，基本是根文件系统挂在/下的，这样一个发行版就占用一整个硬盘分区，外观上也很不雅观，业界竟然也没多少人注意到这个问题，要是能进行一下改造：在不破坏这个根目录是挂不挂在/下这个事实的基础上，如果我们能让系统从/下的一个子目录启动就好了。比如从/system启动。这样有很多好处，外观清爽不说，还可以在一个分区中准备多个发行版并从中引导运行（有没有一点像虚拟化？），每个rootfs对应一个发行版/system1,/system2,etc..，这样还可以被独立打包备份。除了/system1,/system2,我们还可以有与/systemx并列的/usr，/usr就是用户扩展，应用安装后所在的目录，里面会有bin,include,lib这些用户扩展。这样就分开了传统linux发行版将所有一切放到/傻傻不分的情况了，最终/下会有/boot,/system,/usr三个文件夹，如果说我们可以有混合32/64文件系统，那么在这里，我们甚至可以混合使用这三个文件夹的组合，达到一种“system和usr extend能分开能组合的高度自定义化文件系统”。
>
> 上述说法中，承认我们没有破坏根目录挂载在/下的事实是很重要的，因为我们仅是想做个trick，让系统文件归档在/system下使之变得好看，并做到能启动就好了，事实上，这仅是改造busybox的事我们的目的就能达到 - 仅是改造busybox源码中硬编码的文件路径，并不需要关注“改造根目录挂载位置”这样重大的问题，而那其实是一个难度甚高的事情。比如可以利用initramfs+privo root达到。---- 但这样的方案就有点重了。

好了，让我们来看是怎么回事吧，先来看32/64位混合文件系统。

在tinycolinux上组建32/64位混合文件系统

在《在tinycolinux32上装64位toolchain》文中，我们提到产生的64位程序不能运行，甚至ldd都不能分析出其引用，仅提示wrong elf64class，直接执行也提示not found,这是因为它找不到64位共享库，由于ldd无法使用，我们通过其它手段分析，发现最终原因其实是因为默认64位GCC产生的glibc，将GCC产生的程序对loader，即ld-linux-x86-64.so的引用，放在了/lib64中（至于其它基础库libc-2.12.1.so，libcrypt-2.12.1.so，libm-2.12.1.so，libpthread-2.12.1.so，你可以把它做起对应软链一同放在/lib64中，其实不做也可以，因为它们被引用在了/usr/local/gcc443/x86_64-pc-linux-gnu/lib这个是由编译工具链时hardcoded指定的，还有libstdc++.so.6.13,libgcc_s.so.1也要放/usr/local/gcc443/x86_64-pc-linux-gnu/lib）。因此我们仅需：

首先把这个文件和它引用的真实文件ld-2.12.1.so复制到/lib64下，并把ld-2.12.1.so加起执行权限来（这个至关重要，否则会提示access corrupt shared libraries），然后把上述的文件各自复制到其所在目录。执行64位测试程序，发现能成功运行！

这样，tinycolinux就拥有了二套GCC支持开发和运行的程序，所在的文件系统，一套在/lib下，一套在/lib64下。分别同时支持32位和64位。

在tinycolinux上组建system和usr extend分开的高定文件系统

还记得我们开头谈到至简的rootfs就是busybox+一些init脚本吗，我们不断提到的busybox是一个产生rootfs的基础和中心，总管，它自包含我们建立这个测试环境需要的一切，我们来使用它建立这个最简的rootfs样本：

我们是在tinycolinux本身带有GCC481的环境下测试的，为了方便测试使用云主机，使用快照随时准备备份恢复重来，使用的tinycolinux它自己就有rootfs。根目录下有个init,/bin下有个busybox，注意到这些细节后，我们来构建自己的busybox：

首先下载busybox源码http://mirrors.163.com/tinycorelinux/3.x/release/src/下载busybox-1.19.0.tar.bz2,busybox-1.19.0-config和9个patch并运用到解包后的源文件,按《tinycolinux上硬盘安装》一文准备sudo make menuconfig并运用config，为了我们的分离式文件夹系统，busybox事先是被静态链接的，静态链接可以免去对lib目录的依赖,且编译menuconfig配置时设置了把/system/usr/bin,/system/usr/sbin一起合并到/system/bin,system/sbin中， ---- 因为lib我们要将其做在usr文件夹中，与system,boot并列，sudo make install 编译好后复制_install为根目录下的/system，那个/system/linuxrc不要删。

然后我们按照《将tinycolinux安装在硬盘上》一文中的grub启动/boot下的kernel，具体我们测试用的kernel启动参数是：linux /boot/bzImage ro root=/dev/vda1 swapfile=vda1 local=vda3 home=vda3 opt=vda3 tce=vda3 init=/system/init，，，完整的grub菜单文本请参照那文查看。注意到init=/system/linuxrc，这是新加的一条参数。它定义了系统在引导系统时发现root=/dev/vda1后，完成系统将执行权交给PID0来初始化文件系统的那个PID0，root只能是设备，对应文件系统中的/，而init pid0可以是/下任意路径下的一个可执行程序，一段脚本。这段参数其实就是kernel转手给通往rootfs init的连接器(其实你可以patch kernel中的init/main.c让它加载你自己的init)。

业界有很多复杂化的init，如systemvinit等，tinycolinux也定义了它的脚本化init，在tinycolinux中，init是根下的init是一段脚本，但对于简单的init，你可以将它直接链接到busybox中的init，在我们的测试环境，就是这么用的:linuxrc链接到system/bin/init,busybox init仅链接就能作为init使用是因为它其实包含了一系列默认动作，就像传统init能做的那样：它首先会查找etc/inittab，这个文件可以没有，没有的话,busybox init会执行/etc/init.d/rcS，在这里它要执行一些必要工作，所以我们还要准备一些把busybox当init用脚本和作一些初步工作：a)在/system下建立dev，etc，proc,sys四个空目录，b)dev下准备二个设备文件 mknod console c 5 1和mknod null c 1 3，然后: c)etc下提供fstab,inittab,init.d/rcS，其中inittab,rcS都加起执行权限,内容分别为:

fstab:

```
proc /system/proc proc defaults 0 0
sysfs /system/sys sysfs defauts 0 0
```

inittab:

```
::sysinit:/system/etc/init.d/rcS
console::respawn:-/system/bin/sh
::ctrlaltdel:/system/sbin/reboot
::shutdown:/system/bin/umount -a -r
```

init.d/rcS:

```
#!/system/bin/sh
/system/bin/mount -a
```

好了，仅是这样就OK了(你可以先不用/system，将上面的rootfs打包成initrd.gz在普通方式下测试，证明这个文件系统是完善的，最终结果是进入无误进入命令行。)。下面我们试着让基于附加了/system的rootfs运行，直接改名原来tinycolinux的/bin/busybox，让新的busybox生效，继续如下测试，如果失败有下列原因之一，在下列失败可能和解决方案循环间不断恢复云主机重新尝试：

1)失败可能:

提示kernel panic，说提供的init不可执行，系统尝试执行tinycolinux /下的默认init

warning:cant start default console

sbin/gettty not found之类之类

可以看到sh，但ls ,which not found

可见光是脚本文件不足于影响busybox的行为，由于我们企图将etc这些东西归类到/system/etc下，所以我们需要定制busybox中的路径硬编码部分以继续测试:

2)解决方案：

改动源码：

include/libbb.h

```
1690: define bb_default_path      (bb_PATH_root_path + sizeof("PATH=/system/bin:/system/sbin:/sbin:/usr/sbin"))
1717: #define LIBBB_DEFAULT_LOGIN_SHELL  "-/system/bin/sh"
1725: #define CURRENT_TTY "/system/dev/tty"
1726: #define DEV_CONSOLE "/system/dev/console"
```

init/init.c

```
137: #define INIT_SCRIPT  "/system/etc/init.d/rcS"   //为全程定制busybox
init的行为起见，这句必须要搭配下面一句inittab使用
638: parser_t *parser = config_open2("/system/etc/inittab", fopen_for_read);
684: tty = concat_path_file("/system/dev/", skip_dev_pfx(tty));
996: putenv((char *) "SHELL=/system/bin/sh");
1018: new_init_action(SYSINIT, "mount -t proc proc /system/proc", "");
```

继续编译。

cd /tce/busybox sudo make clean sudo make install sudo cp _install/system/bin/busybox /system/bin

不断测试，最终成功。系统启动过程无误，最终出现正常命令行。当然还有很多需改动使这个rootfs变得更完善的空间。

为了维护这套干净强大的文件系统设计，用户要注意在编译程序时将其产生到/usr下，永远不要采用./configure 默认无prefix的情况。

你可以整合tinycolinux的现有init逻辑，把tinyclinux的根文件系统改造成高定文件系统，以如上在tinycolinux内部循序渐进地改动进行的方式。

关注我。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

基于虚拟机的**devops**套件及把**dbcolinux**导出为虚拟机和**docker**格式

本文关键字：**hashi packer,devops backend cloudcomputing,**制作**dbsubcolinux**的**docker**模板,零配置，自动化开发件

在前面我们多次谈到cloud ide & liveeditor,用nas+docker作devops等集成思想，所谓devops，它是一个集成语言系统，开发件，运维件，IDE tools，容器环境的所有自动化开发相关的东西,Devops是一个很巨大的工程，它越来越作为IT的基础存在，比如甚至这个过程和生态整合进了系统和运维 —— 这里的需求是：统一开发，多节点分布式部署，持续发布，大型网站的负载均衡架构，有点接近devops backend computing，——但更显然，它最开始主要服务于开发devops backend appdev。

我在bcxszy选型中，与之对应的概念就是engitor （这个命名带有点使cloud回归native的概念），vs devops 我的思想要更超前一点，尽量零配置，自动化，使得用户仅依赖它，就可以动手开发不管其它的东西。这个东西要像linux rootfs一样可以配置出来,作为某种系统基础件built into a os就好了,有点类似devops ci backend os的意思。。除了xaas的dbcolinux，在《选型》中devops的engitor它就是大头了。

> 在现实生活中，devops虽然出现较晚，但对比物很多。这些都在我们以前与devops和cloud ide相关的文章中提到过，比如，gitlab是一个集成化的ci,cd平台。它是基于git的devops，这里git只是分布版本提交器与devops没有太大关系，gitlab runner才开始有devops，我们以前也提到docker可用devops，单纯的docker只是一个容器还没有持续的概念存在docker composer才刚有持续构建的概念存在。另外一个例子，就是mac osx的xcode ide,它是ide的devops可启用build bot不过它也是对其它工具的调用,还有vagrant,Jenkins,chef这些。——— 所以，往往devops是对所有这些工具的结合运用，最终达到一种称为CD，CI的过程，注意这个持续C —— 这其实本质就是一种代码化加自动化思想，CI的思想本质就是就是把一切代码化。就像脚本语言一切化一样，碎片化了就可以做任何集成层的持续过程事情,比如docker有composer这是语义化了。实际上是可编程部署的容器（写yal语法）和自动化部署（这就有了devops），vagrant这种用命令行vagrant up,etc..和可脚本配置方式控制虚拟机，所以也有devops可能。
>
> 我们还谈到terralang是一种devops语言，其实所有脚本语言都可源码文档化，可集成式发布和构建应用。比如实现terralang，使编译期和运行期分离的那个macro系统，它就是一个命令行IDE。就是一种devops的运用。当然，还有很多，很多。。。。。。

基于虚拟机的**devops:hashicorp tools**

最近我用上了osx也使用了parallels desk，主要是冲着business和pro版有devops支持去的，它主要用的是vagrant，上面说到它使一切有了devops可能，vagrant有for parallels desk插件，不过它自己也有虚拟机部件，Vagrant就相当命令行代替了parallels desk的图形环境，vagrant 甚至还可以与docker 结合来用，parallels desk在osx上也就变成了一个基于虚拟机的full devops软件。—— 这也是《聪明的osx云》那文我们讲到的它自带devops，而实际上它的xcode ide也是.

说到这个vagrant，它是hashi corp的东西，它自己也有一个七件套，涵盖虚拟机为中心展开devops的方方面面。以前我们主要谈到docker,git为中心的devops，docker作为轻量虚拟机性能固然好，但是它不能管从0开始，内核定制方面的事情，而vagrant可以。我们这里只讲构建，即与vagrant联系紧密的一个叫packer的工具的使用，它可以制造供vagrant使用的镜像。—— 这里只讲构建，而且用virtualbox代替vagrant，因为packer支持广泛 ——— 支持docker,支持各种虚拟机，甚至云主机，而且vb又有一个图形界面。

我们采用的例子是《将tinycolinux以硬盘模式安装到主机》，在那文中，我们以前手工构建是先发明一个liveos，现在我们不需要了，因为借助packer我们可以直接在虚拟机上构建，试错，调试。因为packer工具就一套以虚拟机为中心的devops语法支持的，持续构建器。

了解原理，准备基础环境

我使用的是osx上的packer_1.4.1_darwin_amd64.zip，VirtualBox-6.0.8-130520-OSX.dmg，Packer的文档在官网上都有，原理大约是先提供一个host os - 一个在其中构建的宿主，我们是在tinycolinux live os上定制新的tinycolinux hd，所以要先准备一个ISO，和各种它的tczs，生成的硬盘镜像就是我们需要的。

在启动构建过程中，packer会自动生成硬盘和虚拟机配置信息（下述基础脚本已写明），然后就是准备builders，和执行provisioners脚本了。然后将这些写入硬盘。

- 安装virtualbox
- 下载packer放入本地os的/usr/local/bin
- 按下面基础脚本要求准备microcore_3.8.4.iso和各种tczs材料。

准备基本脚本

```
{

  "_comment": “整个builders段，是定制这个iso出来的live os，  直到产生一个ssh登录服务，然后 provisioners接手。此时并未重启，除非你在脚本或inli
ne中指定reboot”,
  "builders":
    [{
    "type": "virtualbox-iso",
```

```
    "_comment": “Linux_64这个virtualbox的默认模板，里面有默认的配置，当然你可以定制加入其它参数，具体翻packer文档”,
    "guest_os_type": "Linux_64",
    "iso_url": "./microcore_3.8.4.iso",
    "iso_checksum": "41cbc86443cc12bfbf7b03c4965e4a171ac1aa993017aeef8d04c78db73c6afb",
    "iso_checksum_type": "sha256",
    "ssh_username": "tc",
    "ssh_password": "tc",
    "boot_wait": "4s",
    "shutdown_command": "sudo poweroff",
    "_comment": "http_directory会产生一个主机上的http服务器，下面pkgs里会有3.x,tcz这样的文件夹结构,tce-load -w才能正常下载到”,
    "http_directory": "pkgs/",

    "boot_command":
        [
          "<enter><wait10>",
          "ifconfig",
          "<return>",
        "sudo rm /opt/tcemirror && sudo touch /opt/tcemirror<return>",
        "_comment": “这个10.0.2.2是主机相对于虚拟机nat网卡能访问到的地址”,
        "sudo sh -c 'echo http://10.0.2.2:{{ .HTTPPort }}/ > /opt/tcemirror'",
        "<return>",
        "_comment": “每个tcz要从官网下好dep,md5.txt文件”,
          "tce-load -iw openssh.tcz<return><wait10>",
          "sudo passwd tc<return>",
          "tc<return>",
          "tc<return>",
        "sudo cp /usr/local/etc/ssh/sshd_config.example /usr/local/etc/ssh/sshd_config<return><wait>",
          "sudo /usr/local/etc/init.d/openssh start<return><wait>"
          ],

      "export_opts":
          [
          "--manifest",
          "--vsys", "0",
          "--description","{{user `vm_description`}}",
          "--version", "{{user `vm_version`}}"
          ],
      "format":"ova",
      "vm_name":"dbcolinux"
    }],


  "_comment": “整个provisioners段，开始为这个guestos作各种写入硬盘（那个linux_64自动生成的一块40g的硬盘）作上述export前的过程，当然，如何写入
，脚本里要写明”,
  "provisioners":
    [{
        "type": "shell",
        "pause_before":"1s",
        "execute_command": "echo '' | sudo -S sh -c '{{ .Vars }} {{ .Path }}'",
        "inline":
        [
        "cp -R /tmp/tce ~/"
        ]
    },

    {
        "type": "shell",
        "pause_before":"1s",
        "inline":
        [
        "tce-load -iw parted.tcz",
        "tce-load -iw grub2.tcz"
        ]
    },

    {
        "type": "shell",
        "pause_before":"1s",
        "execute_command": "echo '' | sudo -S sh -c '{{ .Vars }} {{ .Path }}'",
        "scripts":
        [
        "./scripts/phase1.sh"
        ]
        }]

}
```

假设上面构建文件保存为dbcolinux-pe.packer,调试和启动构建的方法：

进入正确的目录，执行packer build ./dbcolinux-pe.packer(for test and debug, you can use -debug -on-error=ask after build command)，用了ask，它会问，你调试完后选择clean即可。不要立即作答。如果用了-debug会走一步卡一步让你确认。

Scripts中就是如何写入硬盘，写入哪些东西形成硬盘系统的逻辑，这是一个字符数组，目前只有phase1一个文件

scripts->phase1中的内容

```
export PATH=$PATH:/usr/local/sbin:/usr/local/bin

echo PREPARE HD
parted /dev/hda mktable msdos
parted /dev/hda mkpart primary ext3 1% 99%
parted /dev/hda set 1 boot on
mkfs.ext3 /dev/hda1
parted /dev/hda print
rebuildfstab
mount /mnt/hda1

echo COPY SOFT
echo /usr/local/etc/init.d/openssh start >> /opt/bootlocal.sh
echo usr/local/etc/ssh > /opt/.filetool.lst
echo etc/passwd>> /opt/.filetool.lst
echo etc/shadow>> /opt/.filetool.lst
/bin/tar -C / -T /opt/.filetool.lst -cvzf /mnt/hda1/mydata.tgz
mv ~/tce /mnt/hda1/
cp -R /opt /mnt/hda1

echo INSTALLING GRUB
grub-install --boot-directory=/mnt/hda1/boot /dev/hda
mkdir /mnt/cdrom/
mount /dev/cdrom /mnt/cdrom
cp /mnt/cdrom/boot/microcore.gz /mnt/hda1/boot/microcore.gz
cp /mnt/cdrom/boot/bzImage /mnt/hda1/boot/bzImage
echo set timeout=3 > /mnt/hda1/boot/grub/grub.cfg
echo menuentry \\\"dbcolinux\\\" { >> /mnt/hda1/boot/grub/grub.cfg
echo  linux /boot/bzImage com1=9600,8n1 loglevel=3 user=tc console=ttyS0 console=tty0 noembed nomodeset tce=hda1 opt=hda1 home
=hda1 restore=hda1 >> /mnt/hda1/boot/grub/grub.cfg
echo  initrd /boot/microcore.gz >> /mnt/hda1/boot/grub/grub.cfg
echo } >> /mnt/hda1/boot/grub/grub.cfg

#reboot
```

————————

懂得了原理，基本够用的语句，或许以后，我们会把所有的xaas->dbcolinux上写过的文章中的构建全部按lesson实例写一次。

关注我

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

利用hashicorp packer把dbcolinux导出为虚拟机和docker格式(2)

本文关键字：**Cross-compile 64-bit kernel on 32-bit machine**

在《将tinycolinux以硬盘模式安装到云主机》和《在tinycolinux32上装tinycolinux64 kernel和toolchain》2篇文章中我们手动构建了32到64的tinycolinux的基础部分，而且用的是非cross compile，即直接在一台ubt64上产生这个kernel，再后来，针对文章1 —— 以硬盘方式安装iso到主机，我们利用虚拟机devops的构建工具packer自动构建了基本的tinycorelinux pe，现在我们继续文章2，利用devops packer自动构建直接从32位编译出64位的kernel。

这就需要用到cross compile。废话不说，直接上源码。这些源码可直接附在《基于虚拟机的devops套件及把dbcolinux导出为虚拟机和docker格式》文章所提到的源码的适当部分，然后以相同的方式构建，结果会产生一个可用的64 kernel+64 toolchain的dbcolinux：

新加的**script1**部分

注意这里的不同，root=/dev/hda1

```
#HD INSTALL
cd /mnt/hda1/
gunzip -c /mnt/hda1/boot/microcore.gz > /tmp/microcore.cpio
cpio -idmv < /tmp/microcore.cpio
echo menuentry \\"dbcolinux hd\\" { >> /mnt/hda1/boot/grub/grub.cfg
echo  linux /boot/bzImage com1=9600,8n1 loglevel=3 user=tc console=ttyS0 console=tty0 noembed nomodeset root=/dev/hda1 tce=hda
1 opt=hda1 home=hda1 restore=hda1 >> /mnt/hda1/boot/grub/grub.cfg
echo } >> /mnt/hda1/boot/grub/grub.cfg
rm /tmp/microcore.cpio
```

packer脚本部分

将这些放进provisioners中的适当部分。注意下面的_comments会引起错误，这里只是为了说明，使用时得去掉。

```
{
        "type": "shell",
        "pause_before":"1s",
    "_comment": “以下安装32位gcc443的官方tczs，因为grep,gcc_libs在scripts1中安装过了，所以这里不必另外安装，另外,scripts1生成的tmp得改一下
权限才能在下一步上传文件”,
        "inline":
        [
        "tce-load -iw gcc.tcz",
        "tce-load -iw binutils.tcz",
        "tce-load -iw bison.tcz",
        "tce-load -iw diffutils.tcz",
        "tce-load -iw file.tcz",
        "tce-load -iw findutils.tcz",
        "tce-load -iw flex.tcz",
        "tce-load -iw gawk.tcz",
        "tce-load -iw m4.tcz",
        "tce-load -iw make.tcz",
        "tce-load -iw patch.tcz",
        "tce-load -iw pkg-config.tcz",
        "tce-load -iw sed.tcz",
        "tce-load -iw base-dev.tcz",
        "tce-load -iw linux-headers-2.6.33.3-tinycore.tcz",

        "tce-load -iw perl5.tcz",
        "tce-load -iw ncurses.tcz",
        "tce-load -iw ncurses-dev.tcz",

        "sudo sh -c 'chown tc:staff /mnt/hda1/tmp/'"

        ]
    },

    {
    "_comment": “这里开始上传编译所需的源码文件”,
      "type": "file",
      "source": "pkgs/3.x/src/64kernelandtoolchain",
      "destination": "/mnt/hda1/tmp"
    },

    {
```

```
    "_comment": “由于这里用了sudo，在scripts,2,3中所有命令默认都是经过了sudo的”,
        "type": "shell",
        "pause_before":"1s",
        "execute_command": "echo '' | sudo -S sh -c '{{ .Vars }} {{ .Path }}'",
        "scripts":
        [
        "./scripts/2.compile64toolchain.sh",
        "./scripts/3.compile64kernel.sh"
        ]
        }
```

compile64toolchain and compile 64 kernel

将这些放进scripts/2.compile64toolchain.sh文件中，不要轻易动这里的东西，否则格式不对产生的tls.h不能发现错误很诡异。

```
export PATH=$PATH:/usr/local/sbin:/usr/local/bin

#我们现在是在32位上创建for 64的交叉，从64到32不叫交叉，反之要交叉
#三元组,machtype在在正常的linux32上会输出i686-pc-linux-gnu字样,tinycorelinux上是空。但是不产生问题
export BUILD=$MACHTYPE
export HOST=x86_64-pc-linux-gnu
export TARGET=x86_64-pc-linux-gnu

#我们全程不必用sudo，因为packer文件中exe command写好了
cd /mnt/hda1/tmp/64kernelandtoolchain/
tar jxvf linux-2.6.33.3-patched.tar.bz2
tar zxvf binutils-2.20.tar.gz
tar zxvf glibc-2.12.1.tar.gz
tar zxvf mpfr-2.4.2.tar.gz
tar zxvf gmp-4.3.2.tar.gz
tar jxvf gcc-4.4.3.tar.bz2
mv gmp-4.3.2 gcc-4.4.3/gmp
mv mpfr-2.4.2 gcc-4.4.3/mpfr

cd linux-2.6.33.3
make headers_install ARCH=x86_64 INSTALL_HDR_PATH=/mnt/hda1/usr/local/gcccross/$TARGET

cd ../binutils-2.20 && mkdir b && cd b
../configure -prefix=/mnt/hda1/usr/local/gcccross -target=$TARGET -disable-multilib
make
make install

#编译器执行文件
cd ../../gcc-4.4.3 && mkdir b && cd b
../configure -prefix=/mnt/hda1/usr/local/gcccross -target=$TARGET -enable-languages=c,c++ -disable-multilib
make all-gcc
make install-gcc

export PATH=$PATH:/mnt/hda1/usr/local/gcccross/bin

#标准C库头文件和一些必要启动文件
cd ../../glibc-2.12.1 && mkdir b && cd b
#在glibc眼里，这里的build和host一个意思，与标准的其它三元组意义不同
../configure -prefix=/mnt/hda1/usr/local/gcccross/$TARGET -build=$MACHTYPE -host=$HOST -target=$TARGET -with-headers=/mnt/hda1
/usr/local/gcccross/$TARGET/include -disable-multilib libc_cv_forced_unwind=yes libc_cv_c_cleanup=yes
make install-bootstrap-headers=yes install-headers
make csu/subdir_lib
install csu/crt1.o csu/crti.o csu/crtn.o /mnt/hda1/usr/local/gcccross/$TARGET/lib
#利用新编的64gcc处理
x86_64-pc-linux-gnu-gcc -nostdlib -nostartfiles -shared -x c /dev/null -o /mnt/hda1/usr/local/gcccross/$TARGET/lib/libc.so
touch /mnt/hda1/usr/local/gcccross/$TARGET/include/gnu/stubs.h

#编译器本身的库文件
cd ../../gcc-4.4.3/b/
make all-target-libgcc
make install-target-libgcc

#标准C库
cd ../../glibc-2.12.1/b/
make
make install

#标准C++库
cd ../../gcc-4.4.3/b/
make
make install
```

将这些放进scripts/3.compile64kernel.sh文件中

```
export PATH=$PATH:/mnt/hda1/usr/local/gcccross/bin

#使用刚编译出来的gcccross，记得末尾的-
export ARCH=x86_64
export CROSS_COMPILE=x86_64-pc-linux-gnu-

cd /mnt/hda1/tmp/64kernelandtoolchain/
# 这里奇怪之处：1，不作把/usr/local/ncurses*相关的库移到/usr/下会大量符号undefined，2，在guest os中测试时如果不用sudo，依然会提示无法找到ncurses相关库，3，sudo make clean没用，一定要sudo make mrproper。4，在virtual box play make menuconfig那行时会一直闪烁，在guestos中测试是正常的,5.所以我们用了make defconfig的方法,但这里还是保留menuconfig相关的脚本
cd linux-2.6.33.3
cp -R /tmp/tcloop/ncurses-dev/usr/local/include/*.* /usr/include/
cp -R /tmp/tcloop/ncurses-dev/usr/local/lib/lib*.a /usr/lib/
cp -R /tmp/tcloop/ncurses/usr/local/lib/*.* /usr/lib/
#直接放文件可能需要重启才能被引用生效，所以这里ldconfig一下
ldconfig
export TERMINFO=/usr/share/terminfo
export TERM=linux
make mrproper
cp -f ../config-2.6.33.3-tinycore64 arch/x86/configs/x86_64_defconfig
make x86_64_defconfig
make bzImage


cp -f arch/x86/boot/bzImage /mnt/hda1/boot/bzImage64
echo menuentry \\"dbcolinux 64\\" { >> /mnt/hda1/boot/grub/grub.cfg
echo  linux /boot/bzImage64 com1=9600,8n1 loglevel=3 user=tc console=ttyS0 console=tty0 noembed nomodeset root=/dev/hda1 tce=hda1 opt=hda1 home=hda1 restore=hda1 >> /mnt/hda1/boot/grub/grub.cfg
echo } >> /mnt/hda1/boot/grub/grub.cfg

rm -rf /mnt/hda1/tmp/64kernelandtoolchain/

#这是一条触发ask的错误语句,放在你猜想可能出现错误作为中断点的地方，但是它还会继续
Dsfasdfdf
```

好了，scripts3文件结束，产生的镜像文件可能很大。需要针对处理一下。

————

关注我

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

利用hashicorp packer把dbcolinux导出为虚拟机和docker格式（3）

本文关键字：**gcc enable shared cross compile,multiarch gcc,multilib gcc,cross compile gcc under tinycorelinux**

在 《利用hashicorp packer把dbcolinux导出为虚拟机和docker格式》1，2中，我们实现了基本的hd pe环境和释放到hd /能启动的硬盘环境，且编译出了for 64的gcc cross toolchain和64 kernel，那么现在，我们需要使之释放到hd /system下还能启动，且未来将继续这个/system下的32/64混合rootfs的一些强化工作，将之打造成一个高可用的系统.

我们先来完成最基本的效果，即实现《在tinycolinux上组建子目录引导和混合32位64位的rootfs系统》一文提到的效果：初步组建这个/system rootfs，并验证/system下的rootfs能不能够启动。——— 本文完成的我们称它为3264rootfsbase，未来继续强化至3264systemfull.

我们的计划是，按上面的安排，分二步，形成phase1->phase2的效果，phase1即dbcolinuxbase，=前面的toolchain+kernel+基本rootfs的打造。phase2即build3264systemfull,在以后的文章中《打造子目录引导和混合32位64位的rootfs系统全部》再讲述和丰富。第一步的dbcolinuxbase做成一个pvm，即parallels的"type": "parallels-iso”模式，第二步的后续dbcolinuxfull要从这个iso生成的pvm生成，即parallels的"type": "parallels-pvm" builder模式，对应2个phases，这样便于集成，和调试。尝试packer的新东西。且phase2可以直接复用phase1的结果持续集成.

1,packer文件，其它准备工作

由于换了osx加parallelsdesk，我们需要改变一下基本的packer文件,以下只提到加的部分，文章1，2中有的部分会省略：

builders中是这样：

```
"type": "parallels-iso",
"guest_os_type": "linux",
"hard_drive_interface": "ide",
……
"parallels_tools_flavor": "lin",
……
"boot_wait": “4s",
"shutdown_command": "sudo poweroff",
……
"prlctl": [["set", "{{.Name}}", "--startup-view", "window"]],

"boot_command":
[
……
"tce-load -iw parted.tcz<return>",
"tce-load -iw grub2.tcz<return>",
"tce-load -iw squashfs-tools-4.x.tcz<return>",
……

整合了这里为compiletc,base-dev包含linux header
"echo 'setup gcccore pkgs...'<return>",
"tce-load -iw compiletc.tcz<return>",

"echo 'setup gccextra pkgs...'<return>",
"tce-load -iw perl5.tcz<return>",
"tce-load -iw ncurses.tcz<return>",
"tce-load -iw ncurses-dev.tcz<return>"
……
]
```

可见，为了调试，1,2步方便持续集成，我们还把整个gcc放进了bootcommand，还整合进了一些必要库和工具。

再准备文件，与packer文件同层的src文件夹中，分成了/base和/others中，这样的文件夹安排，base对应phase1,是原来的编译toolchain,64kernel所用的源码文件夹+.x tinycorelinux 中的 src repos/busybox-1.19.0patched.tar.gz,busybox-1.19.0-config。，而others对应phase2所要用到的源码包。有autoscan-devices.openssl-1.0.1.tar.gz,openssh-5.8p1.tar.xz,sudo-1.7.2p6.tar.gz,libzip-0.10.tar.xz,make-3.81.tar.gz,etc..

所以，"provisioners”部分会是这样：

```
{"type": "shell","pause_before":"1s","execute_command": "echo '' | sudo -S sh -c '{{ .Vars }} {{ .Path }}'","inline":["cp -R /
tmp/tce ~/"]},
{"type": "shell","pause_before":"1s","execute_command": "echo '' | sudo -S sh -c '{{ .Vars }} {{ .Path }}'","scripts":["./scri
pts/1.bootstrap.sh"]},

由于我们稍后的bootstrap中会不再释放microcore.cpio,这个tmp要事先生成。
……
"sudo sh -c 'mkdir /mnt/hda1/tmp/'",
"sudo sh -c 'chown tc:staff /mnt/hda1/tmp/'"
……
```

```
{"type": "file","source": "src/base","destination": "/mnt/hda1/tmp"},

{"type": "shell",
"pause_before":"1s",
"execute_command": "echo '' | sudo -S sh -c '{{ .Vars }} {{ .Path }}'",
"scripts":
[
"./scripts/2.1.build64toolchainandkernel.sh",
"./scripts/2.2.build3264rootfsbase.sh"
]
}
```

可见原来的2，3整合了成了2，这里的script3是要新加的,为了方便调试，在packer的输出窗口中简便查看起见，包括上面的packer尽量将简单的树形xml写在行内，接下来的脚本中，我们也作了一些小的改变和与原来文章中的不同调整：

我们把重要的注释换成了echo “ing …”，脚本中，解压文件的都不加verbose,涉及到编译toolchain,kernel的地方。都./configure CC=“gcc -w”，且以上统一都 > /dev/null，还有，toolchain单独一个文件夹，与基础rootfs所需的/lib,/lib64分开，我们还将切换native gcc/刚建的cross gcc的方法不再使用export CROSS_COMPILE，而直接用文件名。

调试语句上，发现前面文章写的，一句letitdebug的无义语句，放在位置放在脚本尾或强行断点处并不能让on error ask断下来，reboot会产生断点让后面的语句无法运行，但也会丢失当前环境，packer -debug还是最可用的，除了它每执行一步都问一句有点烦之外，packer不支持-debug-on-error。

这些大部分都需要在脚本层更改，我们一件一件来：

2,bootstrap脚本的变动

因为我们要形成纯净的/system,所以去掉了/下的系统(以下注释掉了部分)，整个bootstrap会是这样:

```
echo "HD INSTALL..."
#以下适合/的情况
#cd /mnt/hda1/
#gunzip -c /mnt/hda1/boot/microcore.gz > /tmp/microcore.cpio
#cpio -idmv < /tmp/microcore.cpio

#tce-load不能sudo,只能unsquashfs方式安装
#cp -R /tmp/tcloop/squashfs-tools-4.x/usr/ /mnt/hda1/
#unsquashfs -f -d /mnt/hda1/ /mnt/hda1/tce/optional/gcc_libs.tcz
#unsquashfs -f -d /mnt/hda1/ /mnt/hda1/tce/optional/openssl-0.9.8.tcz
#unsquashfs -f -d /mnt/hda1/ /mnt/hda1/tce/optional/openssh.tcz
#ldconfig
#tar zxf /mnt/hda1/mydata.tgz -C /mnt/hda1/

#/下，tce=hda1实测运行tce-load是无效的
#echo menuentry \"dbcolinux hd\" { >> /mnt/hda1/boot/grub/grub.cfg
#echo  linux /boot/bzImage com1=9600,8n1 loglevel=3 user=tc console=ttyS0 console=tty0 noembed nomodeset root=/dev/hda1 swapfi
le=hda1 tce=hda1 opt=hda1 home=hda1 norestore >> /mnt/hda1/boot/grub/grub.cfg
#echo } >> /mnt/hda1/boot/grub/grub.cfg

#以下适合/system的情况
mkdir /mnt/hda1/system/

#除了grub条目，所有其它东西要一点点靠组装，或编译出来,注意这里加了root,init，以及norestore,copyfiles tce是为了restore，这里用不着
echo menuentry \"dbcolinux systemhd\" { >> /mnt/hda1/boot/grub/grub.cfg
echo  linux /boot/bzImage64 com1=9600,8n1 loglevel=3 user=tc console=ttyS0 console=tty0 noembed nomodeset root=/dev/hda1 init=
/system/linuxrc swapfile=hda1 tce=hda1 opt=hda1 home=hda1 norestore >> /mnt/hda1/boot/grub/grub.cfg
echo } >> /mnt/hda1/boot/grub/grub.cfg
```

3，patch源码部分

这些文件是经过修改的，在接下来的scripts会被释放覆盖相应的源码文件夹中的对应层次，形成patch的效果。所以它的结构保留了原源码文件夹的结构。然后打包成xx patch.tgz文件，有：linux-2.6.33.3_patchs，glibc-2.11.1_patchs，busybox-1.19.0_patchs,patched的部分请去文章3中去对照。也可以直接去源码库中找。

4, /scripts/2.build64toolchainandkernel.sh

这里主要是文章2，原来脚本2，3的强化和修正：

```
……
```

```
echo "compiling binutils ......"
#这里设成shared，因为lib64中的ld要突出so，ld-*稍后需要复制一份出来
#shared模式下编译64bit bfd会出现ar:file truncated问题，所以要编译二次，fix一次tce-load compiletc里那个,它会安装到/usr/local/bin/
cd ../binutils-2.20 && mkdir b && cd b
../configure CC="gcc -w" -enable-64-bit-bfd > /dev/null
make > /dev/null
make install > /dev/null
cd ../../binutils-2.20 && mkdir b2 && cd b2
#只要binutils -enable-shared，那么用它编译的gcc必定也是-enable-shared,，会影响接下来c libs.cpplibs全是shared
../configure CC="gcc -w" -prefix=/mnt/hda1/system/toolchain -target=$TARGET -enable-shared -disable-multilib -enable-64-bit-bf
d > /dev/null
make > /dev/null
make install > /dev/null
……

echo "compiling c lib startups ......"
#标准C库头文件和一些必要启动文件
#毕竟，通过export cross compile切换cross compile是不好的，我们干脆手动指定，一了百了
#如果cross compile gcc出错，出现fenv.h找不到那就是你改动了一些prefix文件夹，如果出现tls.h找不到，那么就是x86_64-pc-linux-gnu-gcc与gcc你没用
对,compiling c libs，二处都是x86_64-pc-linux-gnu-gcc而不是gcc
tar zxf /mnt/hda1/tmp/base/glibc-2.11.1_patchs/Archive.tgz -C /mnt/hda1/tmp/base/glibc-2.11.1/
cd ../../glibc-2.11.1 && mkdir b && cd b
#在glibc眼里，这里的build和host一个意思，与标准的其它三元组意义不同
../configure CC="x86_64-pc-linux-gnu-gcc -w" -prefix=/mnt/hda1/system/toolchain/$TARGET -build=$MACHTYPE -host=$HOST -target=$
TARGET -with-headers=/mnt/hda1/system/toolchain/$TARGET/include -disable-multilib libc_cv_forced_unwind=yes libc_cv_c_cleanup=
yes > /dev/null
make install-bootstrap-headers=yes install-headers > /dev/null
make csu/subdir_lib > /dev/null
install csu/crt1.o csu/crti.o csu/crtn.o /mnt/hda1/system/toolchain/$TARGET/lib > /dev/null
#利用新编的64gcc处理
x86_64-pc-linux-gnu-gcc -nostdlib -nostartfiles -shared -x c /dev/null -o /mnt/hda1/system/toolchain/$TARGET/lib/libc.so > /de
v/null
……

echo "compiling c++ libs ......"
#标准C++库
#这是一个bug1
ln -s /usr/local/bin/file /usr/bin/file
cd ../../gcc-4.4.3/b/
……
make CC="x86_64-pc-linux-gnu-gcc -w" bzImage > /dev/null
make CC="x86_64-pc-linux-gnu-gcc -w" modules > /dev/null
make modules_install INSTALL_MOD_PATH=/mnt/hda1/system > /dev/null
ln -s /system/lib/modules/2.6.33.3-tinycore64/kernel /mnt/hda1/system/lib/modules/2.6.33.3-tinycore64/kernel.tclocal

cp -f arch/x86/boot/bzImage /mnt/hda1/boot/bzImage64
```

5,scripts/3.build3264rootfsbase.sh

这是新加的整个脚本3：

```
export PATH=$PATH:/usr/local/sbin:/usr/local/bin

cd /mnt/hda1/tmp/base/
tar zxf busybox-1.19.0patched.tar.gz

echo "making basic rootfs lib,lib64......"
#gcc test.c一个demo，/usr/bin/ldd测试它，就可以找出要运行它的dys
#注意这里，复制32 ld-*,dylibs部分
#/mnt/hda1/system/lib/在前面脚本安装lib/modules时被创建了
cp /lib/lib*.so* /mnt/hda1/system/lib/
cp /lib/ld-2.11.1.so /mnt/hda1/system/lib/ld-2.11.1.so
chmod +x /mnt/hda1/system/lib/ld-2.11.1.so
#再来套/lib64下的
mkdir /mnt/hda1/system/lib64/
cp /mnt/hda1/system/toolchain/x86_64-pc-linux-gnu/lib/lib*.so* /mnt/hda1/system/lib64/
cp /mnt/hda1/system/toolchain/x86_64-pc-linux-gnu/lib/ld-2.11.1.so /mnt/hda1/system/lib64/ld-2.11.1.so
chmod +x /mnt/hda1/system/lib64/ld-2.11.1.so
cp /mnt/hda1/system/toolchain/x86_64-pc-linux-gnu/lib64/lib*.so* /mnt/hda1/system/lib64/
#链接,file ./ld-linux.so.2可查出来
ln -s /system/lib/ld-2.11.1.so /mnt/hda1/system/lib/ld-linux.so.2
ln -s /system/lib64/ld-2.11.1.so /mnt/hda1/system/lib64/ld-linux-x86-64.so.2

echo "compiling busybox ......"
tar zxf /mnt/hda1/tmp/base/busybox-1.19.0_patchs/Archive.tgz -C /mnt/hda1/tmp/base/busybox-1.19.0/
cd busybox-1.19.0
make mrproper > /dev/null
```

```
cp -f ../busybox-1.19.0-config configs/i686_defconfig
make i686_defconfig > /dev/null
#还可以加CROSS_COMPILE=
#有些文件编码问题会导致下面的-变成...，如果出现/usr/local/bin/ld rpath=/system/lib no such file no such file or directory，可能就是这类错误了,很诡异
make CC="gcc -Wl,–rpath=/system/lib -Wl,–dynamic-linker=/system/lib/ld-linux.so.2" CONFIG_PREFIX=/mnt/hda1/system/ install > /dev/null

echo "make basic rootfs data..."
mkdir /mnt/hda1/system/dev
cd /mnt/hda1/system/dev
mknod console c 5 1
mknod null c 1 3

mkdir /mnt/hda1/system/etc
cd /mnt/hda1/system/etc

touch fstab
echo proc /system/proc proc defaults 0 0 >> fstab
echo sysfs /system/sys sysfs defauts 0 0 >> fstab

touch inittab
chmod +x inittab
echo ::sysinit:/system/etc/init.d/rcS >> inittab
echo console::respawn:-/system/bin/sh >> inittab
echo ::ctrlaltdel:/system/sbin/reboot >> inittab
echo ::shutdown:/system/bin/umount -a -r >> inittab

mkdir /mnt/hda1/system/etc/init.d
cd /mnt/hda1/system/etc/init.d
touch rcS
chmod +x rcS
echo #!/system/bin/sh >> rcS
echo /system/bin/mount -a >> rcS

mkdir /mnt/hda1/system/proc
mkdir /mnt/hda1/system/sys

#clean
#rm -rf /mnt/hda1/tmp/base/
#reboot
```

最后，进入新生成的grub系统条目，出现命令行要先export PATH=$PATH:/system/bin:/system/sbin，应该是patchs没做好。下回做。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

发布一统**tinycolinux**，带**openvz**，带**pelinux,**带分离目录定制（**1**）

本文关键字：**tinycolinux**上装第二套非标准路径下的**gcc toolchain**

在前面《在tinycolinux上组建可子目录引导和混合32位64位的rootfs系统》一文中，我们讲到了定制可启动linux rootfs结构的可能：我们测试的是最简的情况，发现仅是修改busybox和启动脚本就可以达到目的，，，由于linux根文件系统大多基于busybox有同样的启动脚本流程，，于是在那文的文尾我们还谈到这种思路可以用于一般linux，比如tinycorelinux，只是在tinycolinux中，预想除了busybox,inital scripts，可能还要涉及到从kernel到glibc,甚至一些系统必要工具中的硬编码路径的修改（因为tinycolinux默认最简情况下由这些组成）。下面，就让我们来完成这个定制，并记录下整个细节。最终我们要得到从/system中启动的tinycolinux的发行版。

而且，，，而且我们还将一并完成某些额外工作，比如运用liveos技术运用在基于这个rootfs的一个linux发行版中，并希望整合openvz到这个定制内核。

为什么要这么做呢？如果一直关注我的《bcxszy part2》->《xaas》系列，读者一定会发现这是由colinux as xaas到纯tinycolinux路线的改变，比如原来由winpe(winpe+tinycolinux下busybox维护硬盘分区)变成了这里的统一live tinyclinux+tinycolinux hd，除了装机方案还有cui toolchain-----我们deprecated了《hostguest nativelangsys及uniform cui cross compile system》换成了统一的tinycolinux内的toolchain《在tinycolinux32上装tinycolinux64 kernel和toolchain》，且不久前还谈到《DISKBIOS：一个统一的实机装机和云主机装机的虚拟机管理器方案设想》，而在这里，我们还将进一步完善这个统一的tinycolinux内的rootfs和toolchain建设,且整合进openvz，使之变成一个高可用的综合xaas体。

所以最终，我们完成得到的是一个带pelinux和分离目录定制的rootfs和带虚拟化openvz的tinycolinux发行。

首先我们来准备这个liveos环境和一个带gcc443的tinycolinux硬盘系统环境，liveos部分会不断在接下来的定制rootfs的过程中被调用：

> 按《将tinycolinux以硬盘模式安装到云主机》的方法安装一套硬盘tinycolinux 3.8.4到硬盘，menuentry tinycolinuxhd(/)启动脚本可写成：
>
> set root=(hd0,msdos1) set prefix=(hd0,msdos1)/boot/grub linux /boot/bzImage ro root=/dev/vda1 local=vda3 home=vda3 opt=vda3 tce=vda3
>
> 准备microcore.gz一份也放到/boot下，作为liveos在grub启动脚本中被启动,menuentry "tinycolinuxlive"：
>
> set prefix=(hd0,msdos1)/boot/grub linux /boot/bzImage ro initrd /boot/microcore.gz
>
> 有了这个livecd和二个分区，然后我们启动进liveos把整个/mnt/vda1下的/打包起来放进/mnt/vda3下的某个tar.gz中，在增删hd rootfs时有出错了我们随时可以在livecd中恢复回来，命令就不用提供了吧。
>
> 再来准备我们要测试的对象rootfs：
>
> 把备份的xxx.tar.gz解压一份到/system下，并用如下脚本设置启动menuentry "tinycolinuxhd(/system)"：
>
> set prefix=(hd0,msdos1)/boot/grub linux /boot/bzImage base ro root=/dev/vda1 local=vda3 home=vda3 opt=vda3 tce=vda3 init=/system/init
>
> 注意它与上面grub条目的区别，这里是启动/system/init下的rootfs，这样，系统中就有二套hd rootfs,/system下为我们要得到的rootfs，这个/system我们要让它成为根目录下唯一文件夹也能启动tinycolinux

目前为止，这2个hd rootfs都可能启动，我们首先为这个新rootfs准备toolchain和基本的openssh支持。，过程可能要求你在前二个系统中不断reboot

编译另一套安装到/**system**下的工具链

先按《在tinycolinux32上装tinycolinux64 kernel和toolchain》一文中准备bootstrap new gcc用的gcc443，并编译新的GCC，由于这里是在32位准备另一套装到/system的gcc不是那文中的装64位toolchain，，所以这里与那文中的步骤有类似但不完全相同，主要三件套中的某些步骤会有所不同（比如去掉了x86_64相关的参数，省掉了glibc三步步骤为一步），依次完成下列工作：

安装make.tcz 381,perl5.tcz binutils : sudo ../configure --prefix=/system --disable-multilib linux header : sudo make INSTALL_HDR_PATH=/system headers_install gcc:sudo ../configure --prefix=/system --enable-languages=c,c++ --disable-multilib export PATH=$PATH:/system/bin

定制glibc 2.12.1源码中的路径，将其中引用/lib和/lib64的部分改成引用/system/lib,/system/lib64，和一些由/移到/system的通用路径修正:

```
glibc-2.12.1\sysdeps\generic paths.h,ldconfig.h
glibc-2.12.1\sysdeps\unix\sysv\linux\x86_64 ldconfig.h
glibc-2.12.1\elf readlib.c
glibc-2.12.1\sysdeps\unix\sysv\linux\ia64 ldconfig.h
```

以上更改主要是让新的glibc的ldconfig会发现/system下的lib。当然我们现在还处在旧rootfs中，还没有条件来测试具体引用过程。

编译时先在源码目录下mkdir b，cd b,需要建立一个configparms引导接下来的configure，文件内容为CFLAGS += -march=i486 -mtune=i686 -pipe，否则会出问题__sync_bool_compare_and_swap_4问题。

然后sudo ../configure --prefix=/system --disable-multilib sudo make install

然后就是整个gcc了。重新cd gcc-4.4.3/b sudo make sudo make install

由于libz极为基本，binutils中的as和openssh都要引用到libz，所以需要编译这个先，从3.x的release->src下取得libz的src，编译这个libz：

sudo ./configure CC="gcc -Wl,--rpath=/system/lib -Wl,--dynamic-linker=/system/lib/ld-linux.so.2" --prefix=/system

(注意不要拼错。否则出错，最好直接在winputty中直接右键粘贴，上面这句是用原rootfs中的gcc，新gcc中的库来产生libz)

然后是make381 src，同样用上面的configure同样构建make到/system下

然后依次是:openssl,openssh,openssh需要openssl,我们下载的是《在tinycolinux上安装sandstorm davros》一文中的版本，openssl的configure文件较特殊我们需要把其中的gcc:批量替换成gcc -Wl,--rpath=/system/lib -Wl,--dynamic-linker=/system/lib/ld-linux.so.2，然后

cd openssl-1.0.1/ sudo ./Configure（大写的） CC="gcc -Wl,--rpath=/system/lib -Wl,--dynamic-linker=/system/lib/ld-linux.so.2" --prefix=/system linux-elf sudo make install

openssh的编译 cd openssh-5.8p1/ sudo ./configure CC="gcc -Wl,--rpath=/system/lib -Wl,--dynamic-linker=/system/lib/ld-linux.so.2" --with-ssl-dir=/system --prefix=/system/usr sudo make

这样整个新的GCC就好了，这样其它的扩展都可以由这个新GCC而来的而不再需要上面的一系列参数了。当然，这建立在一个假设上，我们最终要能boot进/system rootfs，且运行起/system下的gcc来编译程序。而我们还没能达到这个目的 - 我们还没有一个这样的环境。

我们先进入liveos打包一下/system到/mnt/vda3/systemgcc.tar.gz，然后我们重新启动到原rootfs，我们先来完成第一步，为能boot进/system new rootfs使用GCC和openssh改一些路径，注意以下工作依然是在原rootfs中完成的。

建立能用**openssh**和新**gcc toolchain**的新**rootfs**环境

继续《在tinycolinux上组建子目录引导和混合32位64位的rootfs系统》下的工作，修改busybox中的硬编码路径，安装到/system下与toolchain一起，然后是大量的脚本，基本上，把能看到的原/下所有硬编码的路径都加个/system，这个工作量比较大，找到文本批量替换搞一下，注意能启动openssh的那些脚本要改好路径。修改后reboot到/system hd rootfs下，当然这样你是看不到失败的，因为这个/system依然引用大量原/ rootfs中的所有目录。我们的最终目的是脱离所有的/下的文件夹能启动。

注意到lib是openssh和gcc都要引用的目录，这是我们消除的原rootfs下所有目录的第一个目录。以便做到删除原rootfs/lib也能启动openssh和使用gcc，

我们先来完全删除/lib，发现进入/system rootfs失败，没事，失败了可以reboot进livecd还原。

最终，精简/lib至安全版本，仅含几个文件：ld-2.11.1.so,ld-linux.so.2,libc.so.6,libc-2.11.1.so,libnsl.so.1,libnss-2.11.1.so,libnss_dns.so.2,libnss_dns-2.11.1.so,libnss_files.so.2,libnss_files-2.11.1.so

我们也不知道这些文件为何还在引用，无论如何，新的/system rootfs终于能进了能tc登录了，openssh终于可以成功了，不过有好多errors和warnings。

先用起new GCC来，方法是：重新解压那个第一次制作的系统打包，并解压systemgcc.tar.gz到其中，这样就是一个全新的系统了。 再删除原来的/usr/local下的gcc，/usr/bin/ar的链接去掉，去掉旧的GCC引用，这样，新的gcc在export PATH=$PATH:/system/bin后终于可以用了。

当务之急是，利用这个新GCC，进去再编译几个工具。

利用新**gcc**编译**tinycolinux rootfs**必要工具

以下源码全在3.x release src下下载，以下编译过程会自动使用新的/system/lib：

sudo gcc rotdash.c -o rotdash sudo gcc autoscan-devices.c -o autoscan-devices

e2fsprogs的编译:cd /system/mnt/vda3/tce/e2fsprogs-1.41.11/ sudo ./configure --prefix=/system sudo make install

sudo的编译:cd /system/mnt/vda3/tce/sudo-1.7.2p6/ sudo ./configure --prefix=/system --without-pam (注意这个withoutpam) sudo make install

最后是udevm这个东西，比较烦琐，基本上这个引用是这样的：udevm需要gperf,acl需要attr,attr需要gettext

a) 第一次尝试configure udevm，发现不了gperf cd gperf-3.0.4/ sudo ./configure --prefix=/system sudo make install b) 第二次尝试configure udevm cd gettext-0.17/ sudo ./configure --prefix=/system sudo make install

由于attr,acl的configure风格比较相似，文件都不够聪明，所以我们需要指定它引用到的工具路径给它 sudo unsquashfs -f -d / /system/mnt/vda3/tce/gccextras/libtool.tcz

cd attr-2.4.43/ sudo ./configure --prefix=/system MAKE="/usr/local/bin/make" LIBTOOL="/usr/local/bin/libtool" MSGFMT="/system/bin/msgfmt" MSGMERGE="/system/bin/msgmerge" XGETTEXT="/system/bin/xgettext"

并且它的安装要二步完成:sudo make install sudo make install-dev

好了，由于我们把attr库安装到了system/lib,system/include下，所以:

cd acl-2.2.52/ sudo ./configure --prefix=/system LDFLAGS="-L/system/lib" CPPFLAGS="-I/system/include"

但是这样是不行的，我们还是乖乖地将attr多加一次安装到/usr/下以便现在这个gcc发现。

c) 第三次尝试configure udevm，需要lusb

cd libusb-1.0.8/ sudo ./configure --prefix=/system cd usbutils-002/ sudo ./configure --prefix=/system

发现不了pci-ids数据文件，但usb-ids有了,直接下载3.x中的lpci.tcz解压上传吧。

lusb和lpci风格比较相似，

sudo ./configure --prefix=/system LIBUSB_LIBS=/usr/lib LIBUSB_CFLAGS=/usr/include LIBUSB_LIBS=/usr/lib/libusb-1.0.la LDFLAGS=-lusb-1.0

发现不了usb.h，下载lusb的tcz，把usb.h放进来

d) 第四次尝试configure udevm，发现需要glibo

sudo unsquashfs -f -d /system /system/mnt/vda3/tce/libffi.tcz sudo unsquashfs -f -d /system /system/mnt/vda3/tce/gobject-introspection.tcz sudo unsquashfs -f -d /system /system/mnt/vda3/tce/glib2.tcz

可见，如果不将一切都编译到/system/lib，会导致对原/lib和原rootfs依然有依赖。

e) 第五次尝试,完

sudo ./configure --prefix=/system LIBUSB_LIBS=/usr/lib LIBUSB_CFLAGS=/usr/include LIBUSB_LIBS=/usr/lib/libusb-1.0.so LDFLAGS=-lusb-1.0 --disable-extras

还有一个工具rzcontrol不弄了

基本上，这样之后，/system roots能很好启动了（较以前少很多errors,warnings）。

到现在为止,/lib依然没有没完全精简掉。

你需要像对待精简/lib的过程一样对待其它文件夹。一步一步来。先找出它的所有硬编码可能存在于原rootfs中的某处依赖。改正它。尝试删除它，反现不行就在liveos中改回来继续尝试。

最终完成精简掉整个原rootfs /下文件夹的目的。

好了，其它的工作就是逐步修改，不断reboot to liveos备份增量集成修改的过程了，最终你会得到一个无误完善版本的rootfs

我发现这个新rootfs，以前登录winscp或winputty会停顿的现象现在不会了哈哈，不知是不是编译openssh时disable pam了的结果。。

下一篇我们将在这个tinycolinux hd中装openvz，另外，克服程序员技术更新过快和后天健忘（要记的东西太多了）的特点：只能是写博记录。或微博记事。

其它的留到本文第二篇再写吧。

关注我

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

发布一统tinycolinux，带openvz，带pelinux,带分离目录定制（2）

本文关键字：从0开始搭建linux rootfs,在tinycolinux上安装openvz，openvz源码编译任意linux

在前面《发布一统tinycolinux，带openvz，带pelinux,带分离目录定制（1）》中，我们在tinycolinuxhd上集成了pelinux，使之成为一个pre install的linux维护环境，而且我们还改造了其rootfs，使之成为一个文件结构更合理化的tinycolinux发行版，在本篇我们将实现将openvz也集成到这个tinycolinux，实现将其基本变成一个完全适合用单机和集群服务器环境的tinycolinux的承诺，由于进行了深度定制，一路上碰到的大小坑不断，不过实践证明这完全是可行的。不费话了，继续吧：

更彻底的 clean /system rootfs

1)去掉/lib的方法：

前文我们谈到那个/lib中至少需要保留几个文件，这是因为busybox有一条配置CONFIG_USE_BB_PWD_GRP:

If you enable CONFIG_USE_BB_PWD_GRP , BusyBox will use internal functions to directly access the /etc/passwd, /etc/group, and /etc/shadow files without using NSS . This may allow you to run your system without the need for installing any of the NSSconfiguration files and libraries.

只要将它打开编译进BB，就可以删掉这些文件了，纯粹以/system/lib下的文件启动。只是/etc下的passwd,shadow还在被引用，待以后处理。

2)去掉/bin的方法:

登录tc时login总是引用/bin/sh，而不是/system/bin/sh

这是因为没有用上/system/etc/skel中的登录脚本，在其中设置好PATH变量到/system/bin,/system/sbin，并保证内核传过的home=vda3正确挂载在了/mnt/vda3/home/tc（保证其中的内容是由/system/etc/skel中复制过来的），然后系统会自动用上/system/bin/sh

以后，我们将陆续去掉/下其它文件夹。

但本文以测试openvz为优先，接下来的测试，会有tcz不断被安装到/usr/local/bin。所以这一节与接下来的一节可以不相关。用户最好在原来的/rootfs的tinycolinuxhd中进行，接下的讲解也是默认以原先的/ rootfs的测试环境为准进行的。

构建openvz内核和支持工具

谈到从源码构建openvz2.6 series针对任意发行版本，网上的文章都是从https://openvz.org/Download/kernel/2.6.32/2.6.32-feoktistov.1/下载的，有完整的patch打包和.config配置示例，不过我下载的时候都显示404，就连各个mirrors中的这些文件都失效，只好从它的git中下载了https://src.openvz.org/projects/OVZL：这个版本中的kernel是经过patch的，我们需要linux-2.6.32-openvz 74c87ab8a48 ,ploop d42955558fb,vzctl 8b9e1c15fce,vzpkg a05e9f503f9,vzquota 5834f7a1fcb，这几个repos，注意二点，1） 一定要git，不要下到windows打包再上传到linux，这是为了防止稍后编译kernel会出现unkown option error，2） 由于/system rootfs的tinycolinux没有了tmp，而kernel编译脚本中默认还是使用/ rootfs中tmp的，所以暂时恢复建立tmp否则gen initramfs_data.cpio error 1

首先来编译kernel，我们需要tinycolinux 2.6.33中的.config，为了免除更多工作，我们没有将tinycolinux 2.6.33中的patchs搬过来打补丁，实际上tinycolinux对原版vault的kernel 2.6.33也没进行太多影响大局的修正。因此，我们完全可以使用openvz的kernel在其上工作，好了，现在进入src root,make menuconfig，进入发行openvz选项，加载2.6.33的.config，virtio按以前文章中的处理方式处理编译到内核，openvz中的项是星是模块的都不要改动，因为openvz tools会默认以加载模块的方式来启动openvz，sudo make开始编译kernel，完成后在arch/x86/boot中找到bzImage，复制到boot，写个menuentry让其在下一节稍后启动,sudo make modules_install，把安装到lib/modules/tinycolinux-2.6.32.8中的kernel文件夹改为build，因为编译出的kernel要从那里找modules

然后是工具部分，准备安装autotools包含libtool.tcz，vzctl需要用到，这个libtool-dev.tcz也需要，否则在编译vzctl时./autogen.sh时，会出现LToptions not found，sudo ./autogen.sh开始配置过程，发现需要libxml2-dev.tcz,liblzma.tcz，将libxml2.tcz和lzma unsquashfs -f -d安装/进去sudo ldconfig (以前文章中我们不断提到安装了*-dev.tcz的tinycolinux需要重启使lib文件生效，但实际上仅需要这一句就可免重启)，继续，发行还需要ploop，编译ploop不成功。但是没关系。仅需要头文件。继续，还需要libcgroup，但是好像vzctl内含libcgroup，我下载的是sourceforge中的libcgroup-0.41。cgroup配置过程中需要yacc，用flex,bisn也可以，make install后sudo ldconfig

好了，现在可以安心编译vzctl了，sudo ./autogen.sh，需要gmp,pam，一一解决，直到出现配置正确完成。make install会安装主体vzctl文件和支持脚本,make install-debian会安装vzctl控制openvz自启的脚本，由于debian最接近tinycolinux，所以我们安装这个。然后是接下来的vzpkg和vzquota，相对比较简单，直接sudo make install就可以了

到现在为止，内核和工具部分都编译完成了，由于tinycolinux是特殊版，为了稍后能启动，所以我们还需要稍微定制一下：

lsmod在bin下不在sbin下,将它复制到sbin下, 手动建一个/var/lock/subsys,coreutils.tcz中含的stat(coreutils需要libcap,acl,attr,gmp，下载到的attr不含so，按前文的方法编译得到so或直接复制这个so和二个链接出来到/lib下),bash.tcz(直接打开提取上传到/usr/bin),iproute中的ip command都需要否则会出现ip command not found,/etc/init.d/vz脚本也稍微修改一下，将其中sysctl -p后面的-p去掉，/opt/bootlocal.sh加一段使vz随系统自启的脚本，与openssh自启条目放一起，内容为：

```
rm -rf /var/lock/subsys/*
/etc/init.d/vz start
```

现在重启。进之前为新kernel建的grub menuentry，让系统加载新kernel并尝试接下来启动整个openvz

启动openvz并建立一个vps

好了，如果重启后显示running kernel is not an openvz kernel就表明没有进入正确的kernel，反之输出openvz的一系列信息直到显示openvz is running(上面bootlocal.sh脚本中可以加个>null使之不显)则可以继续下一步测试：

下载一个template，我选择的是ubuntu-12.04-x86-minimal.tar.gz，上传到/vz/template/cache,然后安装它：sudo vzctl create 101 --ostemplate ubuntu-12.04-x86-minimal --layout simfs --config basic，发现busybox默认提供的gzip和tar都缺少选项功能，显示invalid option --l等错误。从 http://ftp.gnu.org/gnu/下载tar-1.30和gzip-1.9.tar,编译tar ./configure FORCE_UNSAFE_CONFIGURE=1，和gzip替换busybox中的旧版本，再次执行上面的sudo vzctl create，正确完成！

启动sudo vzctl start 101,sudo vzctl enter 101，显示ubuntu的root用户的命令行。完工！

我们其实可以把vz映射到/mnt/vda3/vz，像tinycolinux的local,tce,usr/local,opt文件夹一样。我们还可以定义自己的os template，这个os template同样可以是基于tinycolinuxhd的。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

发布一统**tinycolinux**，带**openvz**，带**pelinux,**带分离目录定制（**3**）

本文关键字：从**0**开始编译**tinycolinux,busybox openssh nss**，改变**openssh**中的**passwd**文件引用路径

在《发布一统tinycolinux，带openvz，带pelinux,带分离目录定制》系列的前二篇中，我们以修改BB和GLIBC的方式创建了一个/system下的rootfs并编译了诸多工具使之得以正常启动和进行开发（gcc以提供toolchain,openssh以能本地登录和ssh登录）并为它配备了一个livelinuxpe环境以调试，并在这个rootfs中安装实现了openvz，组建这样一个文件系统还是需要实践者具备相当的动手的决心的，因为你需要亲自解决N多大小问题，但所幸结果是在成功预期里的，接下的问题依然是使这个rootfs不断完善，比如将/下还需依赖原文件系统的某些文件夹消除，在本系列文章第二篇中，我们消除了/bin,/lib在那里我们谈到/etc下的passwd,shadow还在被引用，那么在这篇中，我们将要消除的就是这个/etc和/dev

来分析问题，我们现在的状况是：在/etc下和/system/etc下都存在passwd和shadow，通过su root，adduser 或 passwd root发现，实验结果修改了/system/etc/passwd但是没有修改/system/etc/shadow却修改了/etc/shadow，adduser xxx的时候提示语也表现为：没有在/etc/shadow中找到当前创建用户名的条目使用的是/system/etc/passwd，当初编译的时候，BB的include/libbbb.h中所有的#define _PATH_XXX都改变成为了/system/etc/xxx包括passwd和shadow，而glibc/sysdeps/generic/paths.h中的_PATH_SHADOW也是改变成了/system/etc/shadow的，可是我们没有找到_PATH_PASSWD的选项。glibc和bb在使用passwd,shadow方面，当bb配置为CONFIG_BB_PWD_GRP和ENABLE_BB_SHADOW时理应使用的是新定义的/system/etc/xxx下的文件。BB按理那些关于使用自身shadow,passwd的选项（在login management中）应该足于支撑让它无须任何GLIBC的NSS机制独立完成。

实现本地登录引用/system/etc/passwd,shadow

继续来做测试，如果我们删掉/etc/passwd，是可以在本地登录的，但删除/etc/shadow本地不可登录，删掉/system/etc/shadow则无关紧要。于是，问题就是：1）本地登录机制引用/system/etc/passwd是我们预期的结果，但还在引用/etc/shadow而非/system/etc/shadow是意外，是不是libbb.h中那条#define _PATH_SHADOW没有起作用呢，被glibc冲突了呢，不知道是不是冲突，但BB配置SHADOW没起作用这个事实至少是真的。那句adduser xxx的提示语也说明了一切。

于是把BB中所有提到_PATH_SHADOW的都硬编码为“/system/etc/passwd”，我的方式是windows下用7z打开busybox.src.tar，里面打开那个主文件夹，然后搜索压缩包内所有含“PATH_SHADOW”的文件，发现有chpasswd.c,adduser.c,passwd.c,deluser.c,pwd_grp.c,libbb.h,,一一硬编码替换掉，然后重新编译生成BB复制到/system/bin，特别是那个pwd_grp.c，这个文件中的三处硬编码直接关乎我们最终想要的结果。

事实是，修改完的BB完全实现了我们的预期：本地登录引用/system/etc/passwd,/system/etc/shadow就能登录。

实现远程SSH登录也引用/system/etc/passwd,shadow

继续来做测试，我们发现删除/etc/shadow或者/etc/passwd中任一个，远程登录都登录不了，这其实仅仅是openssh导致的问题：是否它被设置成使用系统密码验证时，使用shadow,passwd的方式是偏向GLIBC的，分析它的源码就可发现使用了getpwnam这样的nss函数。

于是追溯源头，用7z搜索glibc2.12.1.src.tar，发现nis/nss_compat/compat_pwd.c,compat_spwd.c中都有"/etc/passwd"字样，nss作为一个注册表查找程序，从这里定义查找文件位置，openssh引用是glibc的shadow,passwd机制，于是改成compat_pwd.c中将其改成“/system/etc/passwd”，compat_spwd.c中将/etc/shadow字样改成/system/etc/shadow，编译覆盖原/system下的glibc

发现远程也能直接使用/system/etc/xxx验证了。

---

openvz实际也是一种应用空间。。因为它的系统包可以是一个不完整的rootfs，是不是这样呢？一个设想，日后求验。

本篇过后，本系列应该不会再出新文章了。关注我。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在tinycorelinux上安装lxc，lxd (1)

本文关键字，在**tinycorelinux**上安装**lxc**，**lxd,gcc4.4 self-reference struct typedef**

在前面的文章中我们讲到过内置虚拟化的os设计，它可以使包括裸金属，云主机在内的等所有虚实主机实现统一装机，且统一各层次的虚拟化(os容器/app容器一体化)，这就是diskbios->cloudbios的设想，在一些文章中我们还讲到利用这样的一套架构打造vps server farm(最初我们尝试的是在windows上利用colinux打造vps farm),甚至打造一个portable cloud environment image file的思想。

那时我们考虑的主要是单纯的xaas：类coreos，但是更偏向接近native的去虚拟化，我们为此建立了一个极小的linux distro，在《发布一统tinycolinux，带openvz，带pelinux,带分离目录定制》系列中我们实现了这样一个linux distro的基础部分：dbcolinux。

在更稍晚的文章中，我们发现了devops,如，《利用packer把dbcolinux导出为虚拟机格式和docker格式》谈到的Provision a dbcolinux img。。我们在《hyperkit：一个full codeable,full dev support的devops,及cloud appmodel》还谈到一种利用轻量hypervisor来做虚拟层的工具和devops OS设计：bhyve。我们甚至还谈了支持编程扩展DSL和devops的语言。

最后，考虑到虚拟化和devops结合，结合四大件，我们承诺实现一个带xaas,langsys,domainstack的全功能到busybox的设想，这里的思想是这样的：—— 基本上，只要把这些xaas,devops尽早尽量上升到上流，而且保持尽量小。就可以在下流domainstacks,apps中再度被抽象。而busybox设想的最终目的，是要实现一个native/cloud可以本地远程无差运维开发的appstack,这是后话。

但是现在，为了得到一个这样可用的东西，我们会采用一些更为实际的方法。—— 比起使用真正的hypervisor，我们可能会继续使用ovz这样的东西，这是因为深思之后考虑到：一些云主机没有intelV硬件功能不可再虚拟化，而且，hypervisor它也比较重。虽然bhyve比较轻量不过比起os级的虚拟(openvz,etc)来说还是比较重，而且它只工作在freebsd，

> 分清二种平台虚拟化containerisation vs virtualisation：
>
> 拿bhyve的衍生品smartos来说。
>
> SmartOS is a specialized Type 1 Hypervisor platform based on illumos. It supports two types of virtualization: OS Virtual Machines (Zones): A light-weight virtualization solution offering a complete and secure userland environment on a single global kernel, offering true bare metal performance and all the features illumos has, namely dynamic introspection via DTrace KVM Virtual Machines: A full virtualization solution for running a variety of guest OS's including Linux, Windows, *BSD, Plan9 and more

Ovz做的主要是第一种，而kvm,bhyve是第二种，我们偏向采用os level的虚拟化，因为它也能devops，而OVZ的devops功能不足。所以我们考虑用lxc/lxd来代替ovz，它的优点有：

1,lxc兼容linux 2.6之上,利用linux本身机制，与docker技术统一。2,lxc作为操作系统级的containerisation技术，它的使命却在于模拟普通虚拟机和hypervisor。3,它也有LXD这样的上层，LXD is a container "hypervisor"。可以Provision生成，甚至休眠。缺点：资料少,有一些与虚拟机的功能不能一一对应，缺失

我们先来讲在dbcolinux安装它，好了，开始吧。

基础工作，安装toolchain增强工具

按《在tinycolinux上编译seafile》的方法，安装3.x的autotools，包括autogen,automake,autoconf,libtool,libtool-dev,etc.. 还要安装tclsh.tcz

如果允许，你也可以把下面的给做了

```
File systems  --->
   Pseudo filesystems  --->
   [*] /proc file system support
That enables the /proc virtual filesystem; read the help file for more on that. Then enable the following:
General setup  --->
   [*] Kernel .config support
   [*]   Enable access to .config through /proc/config.gz

When editing the file by hand, say Y to CONFIG_PROC_FS, CONFIG_IKCONFIG, and CONFIG_IKCONFIG_PROC.
```

重编内核不是必要工作，这是测试lxc需要的。本篇只讲编译。

编译lxc

然后下载lxc-lxc-2.0.11.tar.gz的src,2.0.x是lxc2,选择2是因为它从linux kernel 2.6.32开始，与系统所用kernel接近

1,错误：expected specifier-qualifier-list before sa_family_t 在macro.h中，把所有 include linux/*.h 放include directory的最后 2,错误：ms_shared undeclared here 在conf.c中引用这二个头文件，linux/fs.h and limits.h，也放后面

这里的问题大部分都是因为我们所用的gcc443是c99以内的标准，而lxc源码用了部分c11，所以需要如上的workarounds，实际上编译lxd的时候也会看到好几种相似的情形。需要一一针对处理。

Sudo ./autogen.sh,sudo ./configure,sudo make install

————————

lxd放在接下来一篇讲，因为lxd编译要复杂得多，而且它们二者应该分开，因为lxd作为管理可以不跟lxc一样集成在host中而是一个guest中。有没有。。 至于运用lxc和lxd provision的方式，这些都在网上可以找到。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在tinycorelinux上安装lxc，lxd (2)

在《在tinycorelinux上安装lxc，lxd（1）》中我们讲到源码适配gcc443，由c11退回c99的一些处理，这里依然要处理大量gnu11的事。

准备工作，及编译golang

Grub 加个swapfile=hda1进去。编译go1.12.6内存起码1g。准备git，git我们用4.x的，需要expat2.tcz和openssl-1.0.0.tcz,都用3.x的, 按《在tinycolinux上安装sandstorm davros》编译openssl1.0.1覆盖1.0.0 —prefix=/usr/local,make install,sudo ldconfig，再编译curl 7.30.0 —with-ssl=/usr/local,make install,sudo ldconfig,不用编译git,为防出现unable to get local issuer certificate git，运行git config --global http.sslVerify false

安装bash.tcz，下载并解压go1.4-bootstrap-20171003.tar.gz，Go 1.4 was the last distribution in which the toolchain was written in C，cd go,sudo ./make.bash，不要export GOROOT_BOOTSTRAP=/mnt/hda1/tmp/go，这个没用，还是得mv /mnt/hda1/tmp/go /home/tc/go1.4，下载go1.12.6.tar.gz,cd go-go1.12.6/src,sudo ./make.bash没有之前的swap设置这里过不去， 为了让go生效。export PATH=$PATH:/mnt/hda1/tmp/go-go1.12.6/bin

lxd源码处理

安装libcap.tcz,acl-dev.tcz,下载并解压lxd-3.0.4.tar.gz,cd lxd-lxd-3.04,处理一下lxd src:

> 第一个问题，还是那个问题，我们使用的gcc443不是gnu11，go默认调用gnu11，会出现Unknown command line -std=gnu11 在lxd src中，找到// #cgo 有-std=gnu11的去掉它，对，注释的起作用的，大约有16个文件,然后，在/home/tc/go/src中新建github.com->lxc文件夹,cd lxc,直接mv 修改过的lxd到这里，保证名字是lxd /lxd/shared/idmap/shift_linux.go中， /lxd/shared/netutils/netns_addrs.c中，

然后是makefile:

> Sudo vi Makefile最上面加shell=/bin/bash,default中去掉deps的判断ifeq ($(TAG_SQLITE3),)中的ifeq改成ifneq，进一步来分析一下makefile中这个默认make deps的逻辑：
>
> 它以home/当前用户/go/为GOPATH,维护这样一种结构(GOPATH)/deps/，所以我们mkdir -p ~/go/src,cd ~/go/src,mv /mnt/hda1/tmp/lxd-lxd-3.04 ~/go/src
>
> 继续分析makefile，有5个deps:sqlite,uv,raft,co,dqlite，文件中有4个地址，没有libuv的，稍后处理，但因为这5个deps都可能编译出错，make deps一执行，总是会强行从0开始拉取（sqlite无条件拉取，其它四个判断拉取），所以不可能通过本地修改deps sqlite的相关文件，调试影响make deps使之最终通过。我们只能定制sqlite仓库，然后在makefile中替换其地址：
>
> Sqlite: 2019.4.19左右的sqlite：https://github.com/CanonicalLtd/sqlite/3c5e6afb3d8330485563387bc9b9492b4fd3d88d,你必须fork 它的github仓库，作如下修改，并改动makefile中的GitHub repo调用地址参数来跳过这个 在src/sqlite3.h.in中： 删掉这句 typedef struct sqlite3_wal_replication sqlite3_wal_replication; 然后下面typdef struct sqlite3_wal_replication{…}的sqlite3_wal_replication的前面统统加个struct，有五行 才能避免make deps编译时可能出现redefinition of typedef ‘sqlite3 wal replication’，gcc 4.7之后才支持c11的typedef重定义-Wtypedef-redefinition,，gcc 443是不支持的,

其它四个deps可以分别git到/mnt/hda1/tmp修改，尝试make install，

> libuv: Git clone 2019.6.28左右的https://github.com/libuv/libuv/commit/1a06462cd33fb94720d639f40db3522313945adf Sudo ./autogen.sh,./configure,make install
>
> Raft: Git clone 2019.6.26左右的，https://github.com/CanonicalLtd/raft/commit/ee097affa3dfff53f0c5af096a55d8b7dacecdc3 会出现error implicit declaration of function aligned_alloc，因为C11中添加的函数aligned_alloc（） 你可在configure.ac 161行找到implicit-function-declaration相关行注释掉，这样它就是一个warning而不是error ./configure —disable-example,否则会有TIME_UTC is a macro in C11,TIME_UTC is macro in glibc 2.16
>
> libco: https://github.com/freeekanayaka/libco,目前是v18,没有make install 复制 lib.pc到 /usr/lib/pkconfig/ 手动复制安装下/usr/为prefix然后ldconfig

当然你也可以像对待sqlite一样将修过改的后4个deps的新仓库地址放进makefile中，尝试Sudo make deps，找不到libuv时到那个deps下make install下再sudo ldconfig重新make deps，这样更方便统一。

以上lxd src和dep的src处理，因为go或makefile会将文件不断下到go path，调试的时候，如果有新的错误，记得清空/deps/或src/github.com/中相应的文件夹让makefile或go get重新应用新逻辑。

编译deps和lxd

Make dep,最终成功!!之后需要设几条export，编译完后会提示： Export CGO_CFLAGS="-I/home/tc/go/deps/sqlite/ -I/home/tc/go/deps/dqlite/include/" Export CGO_LDFLAGS="-L/home/tc/go/deps/sqlite/.libs/ -L/home/tc/go/deps/dqlite/.libs/" Export LD_LIBRARY_PATH="/home/tc/go/deps/sqlite/libs/:/home/tc/go/deps/dqlite/.libs/" 如果是手动生成的，对应地址会是/mnt/hda1/tmp/xxx

最后，在整体make (default)前需要处理一下：

> 在这里会有很多陷阱和挑战，主要是golang的包下载需要用到外网线路而且go没有一个可以换mirror的准法。为省事我们将手动补全：src中新建golang.org文件夹->x文件夹，cd x,依然git clone github.com/golang/sys/,github.com/golang/net/, github.com/golang/crypto/,这是因为golang.org的包全部被墙,还有一些虽然没被墙但是较大的包，手动下载,比如下到gopkg.in的mgo v2，cd gppkg.in,git clone https://github.com/go-mgo/mgo/,mv mgo mgo.v2,cd mgo.v2,git checkout v2，v2是它的一个branch

sudo make。会自动下载其它包，没被墙的gopkg.in被依然下载到/home/tc/go/src/gopkg.in。然后自动开始编译，如果在这里出现找不到deps的h,lib往往是make deps后的几条export没设好，没关系，这里可以进一步export覆盖补全。

最后，lxd也编译完成。完工！

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在云主机上手动安装腾讯PAI面板

本文关键字：云主机上装管理面板

在前面，我们介绍过lnmp,sandstorm paas，还有黑群晖，docker管理面板，这些都是云OS上的面板扩展和APPSTACK扩展，分散在不同级别被实现，（像群晖这种是OS和面板一体的），包括这里要介绍的pai和未来可能要介绍的openfaas云函数面板，基本可以分为二类，一带无devops无隔离，没有明显的虚拟化APP打包特征，像宝塔，lnmp，pai，APP直接在baremetal上运行，一带devops，基于容器。像docker管理面板，openfaas。都可以做所有通用服务应用不限web，里面基于容器的技术也都类同，不过vs sandstorm paas，openfaas是用标准容器方法达成的(vs真正的用统一语言和统一微内核做的那套，容器是我们当代伪applvl的virtualappliance做得最完善的了。)，更开放更devops。

所以谁说openfaas,sandstorm,lamp,dsm这样的规模的东西不是个os?openfaas cloud还有整合存储Minio，类似自建云函数+云存储的方案只不过个人难于负担存储部分只能寻求OD这样的替代。

好了不废话。

上次我们搞好了云主机装机的pebuilder.sh。这次来介绍云主机装机常用的服务器套件，一般这类产品有宝塔，wdcp,lnmp等，但是鉴于我们近期在研究云函数和serverless，这次我们找到了PAI，https://cloud.tencent.com/solution/pai，它在一台云主机上自动绑定一个cloudbase域名，并做了对小程序的自动鉴权（大约小程序对xxx.pai.cloudbase.com的域名自动鉴权，否则需要去小程序后台填自定义域名），集成了git拉取pai项目，自动certbot作ssl验证，当然，tx的servless产品主要有cloudbase（里面有云函数云存储云数据库）和wx ide。这个PAI并不能达到官方cloudbase提供的服务那么完整(自建云函数机制，支持云函数的event,context写法)，也不能做到让wx ide完全无缝对接（比如管理PAI上的云函数），这货吧有点像nodejs做的容器和devops，目前它只是自动鉴权方面有点强而已。其它只是一个通用服务器和不使用云函等的小程序后端，没发现什么亮点。

这个PAI它不是一个镜像也不是一个软件，而是需要购买时绑定的。下面我们把它安装在任意云主机上，甚至不是tx cvm也可以。这样我们就失去了那个免费xxx.pai.cloudbase.com三级域名和自动鉴权的好处，但是实际上用自己的域名和自动鉴权也不费事。关键是我们想看看pai有哪些程序可用。直接给脚本：

基础

注意使用说明：云主机事先开5523，并域名绑好到这个云主机上。以便程序内自动申请证书等工作。

一些变量：

```
MIRROR_PATH="http://default-8g95m46n2bd18f80.service.tcloudbase.com/d/demos"
# the pai backend
SERVER_PATH=${MIRROR_PATH}/pai/agent/stable/pai_agent_framework
PAI_MATE_SERVER_ROOT_PATH=${MIRROR_PATH}/pai/mate
PAI_MATE_SERVER_PATH=${MIRROR_PATH}/pai/mate/stable/install
TOOLS_PATH=${MIRROR_PATH}/pai/tools
```

安装依赖

apt-get install git nginx gcc python3.6 python3-pip python3-virtualenv python-certbot-nginx golang -y

单独安装node语言件：

```
# install node.js
installNodejs() {

    echo "====================node.js progress======================="
    msg=$(wget -q ${TOOLS_PATH}/node-v10.16.2-linux-x64.tar.xz
    tar -Jxvf node-v10.16.2-linux-x64.tar.xz -C /usr/local/
    ln -sf /usr/local/node-v10.16.2-linux-x64 /usr/local/node
    rm node-v10.16.2-linux-x64.tar.xz -f
    # for manual launch node in shell maybe in the later
    echo "export PATH=/usr/local/node/bin:$PATH" >> ${HOME}/.bashrc

    wget -q ${TOOLS_PATH}/pm2-3.5.1.tgz
    PATH=/usr/local/node/bin:$PATH npm install -g pm2-3.5.1.tgz
    PATH=/usr/local/node/bin:$PATH npm install -g serve-handler
    rm pm2-3.5.1.tgz -f

    wget -q ${TOOLS_PATH}/sqlite3-4.1.1.tgz
    PATH=/usr/local/node/bin:$PATH npm config set user 0
    PATH=/usr/local/node/bin:$PATH npm config set unsafe-perm true
    PATH=/usr/local/node/bin:$PATH npm install -g sqlite3-4.1.1.tgz
    rm sqlite3-4.1.1.tgz -f 2>&1)
    status=$?
    updateProgress 10 "$msg" "$status" "node.js"
}
```

```
installNodejs
```

pai前后端基础支持

后端5523会透出管理页面，/data/pai-mate-workspace中的应用代理到nginx 3000，first time renew也是为了生成一个/etc/letsencrypt/renewal/下的模板文件供certbot-renew.service服务使用。 安装中，请保证certbot renew务必成功。否则后面的二个pai服务绝对启动不了。但如果成功，基本安装就能很好完成。

```
configngінx() {

    echo "====================certbot renew progress======================="
    systemctl enable nginx.service
    systemctl start nginx

    cp -f /lib/systemd/system/certbot.service /etc/systemd/system/certbot-renew.service
    cp -f /lib/systemd/system/certbot.timer /etc/systemd/system/certbot-renew.timer

    # sed -i "s/renew/renew --nginx/g" /etc/systemd/system/certbot-renew.service

    msg=$(
    #first time renew
    certbot certonly --standalone --agree-tos --non-interactive -m ${EMAIL_NAME} -d ${DOMAIN_NAME} --pre-hook "systemctl stop
nginx"

    systemctl daemon-reload
    systemctl enable certbot-renew.service
    systemctl start certbot-renew.service
    systemctl start certbot-renrew.timer 2>&1)
    status=$?
    updateProgress 40 "$msg" "$status" "certbot renew"


    echo "====================nginx reconfig progress======================="
    # add nginx conf
    rm -rf /etc/nginx/conf.d/default.conf
    cat << 'EOF' > /etc/nginx/conf.d/default.conf

server {
    listen 443 http2 ssl;
    listen [::]:443 http2 ssl;

    server_name DOMAIN_NAME;

    ssl on;
    ssl_certificate /etc/letsencrypt/live/DOMAIN_NAME/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/DOMAIN_NAME/privkey.pem;
    ssl_session_timeout 5m;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256:HIGH:!aNULL:!MD5:!RC4:!DHE;
    ssl_prefer_server_ciphers on;

    location / {
        proxy_redirect off;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_pass http://localhost:3000;
    }
}

server {
    listen 80;
    server_name DOMAIN_NAME;

    if ($host = DOMAIN_NAME) {
        return 301 https://$host$request_uri;
    }

    return 404;
}
EOF

    sed -i "s#DOMAIN_NAME#${DOMAIN_NAME}#g" /etc/nginx/conf.d/default.conf

    # restart nginx
    msg=$(systemctl reload nginx.service
```

```
    systemctl restart nginx 2>&1)
    status=$?
    updateProgress 50 "$msg" "$status" "nginx reconfig"

}

configngінx
```

安装pai,paimate

```
installPai() {

    echo "====================paimate install progress======================="
    mkdir -p ${HOME}/pai
    echo "export PATH=/usr/local/node/bin:$PATH" > ${HOME}/pai/pai-mate-env

    rm -rf /data/logs
    sudo mkdir /data/logs

    echo "Start installing PAI Mate!"
    echo ${PAI_MATE_SERVER_PATH}
    echo ${DOMAIN_NAME}

    INSTALL_DIR="${HOME}/pai-mate"

    # prepare directory
    mkdir -p ${INSTALL_DIR}

    msg=$(# download package
    wget -qO- ${PAI_MATE_SERVER_PATH}/pai-mate-latest.tar.xz > ${INSTALL_DIR}/pai-mate-latest.tar.xz

    # unzip
    tar -Jxvf ${INSTALL_DIR}/pai-mate-latest.tar.xz  -C ${INSTALL_DIR}
    mv ${INSTALL_DIR}/pai-mate-latest.tar.xz ${INSTALL_DIR}/pai-mate-latest.tar.xz.old

    cd ${INSTALL_DIR}

    # config
    echo "UPDATE_PATH: ${PAI_MATE_SERVER_PATH}" > config.yml
    echo "DOMAIN_NAME: ${DOMAIN_NAME}" >> config.yml
    echo "CERT_PATH: /etc/letsencrypt/live/${DOMAIN_NAME}/fullchain.pem" >> config.yml
    echo "KEY_PATH: /etc/letsencrypt/live/${DOMAIN_NAME}/privkey.pem" >> config.yml

    # prepare
    source ${HOME}/pai/pai-mate-env # get node/npm binary path
    #npm install --production --unsafe-perm=true --allow-root
    # download from cos
    wget -qO- ${PAI_MATE_SERVER_ROOT_PATH}/libs/node_modules.tar.xz | tar -Jxf -
    npm run migrate:latest

    # prepare workspace
    mkdir -p /data/pai_mate_workspaces

    # systemd service start
    rm -rf /etc/systemd/system/tencentcloud-pai-mate.service
    cat << 'EOF' > /etc/systemd/system/tencentcloud-pai-mate.service

[Unit]
Description=Tencent Cloud Pai Mate
After=network.target

[Service]
Type=simple
Restart=always
RestartSec=1
User=root
Environment=PATH=/usr/local/node/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
WorkingDirectory=/root/pai-mate
ExecStart=/root/pai-mate/bin/start.sh

[Install]
WantedBy=multi-user.target
EOF

    rm -rf /etc/systemd/system/tencentcloud-pai-mate-update.service
    cat << 'EOF' > /etc/systemd/system/tencentcloud-pai-mate-update.service

[Unit]
Description=Tencent Cloud Pai Mate Update
```

```
After=network.target

[Service]
Type=oneshot
User=root
Environment=PATH=/usr/local/node/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
WorkingDirectory=/root/pai-mate
ExecStart=/root/pai-mate/bin/update.sh
EOF

    rm -rf /etc/systemd/system/tencentcloud-pai-mate-update.timer
    cat << 'EOF' > /etc/systemd/system/tencentcloud-pai-mate-update.timer

[Unit]
Description=Tencent Cloud Pai Mate Update

[Timer]
OnCalendar=daily
RandomizedDelaySec=5minutes
Persistent=true

[Install]
WantedBy=timers.target
EOF

    chmod +x ${INSTALL_DIR}/bin/*
    systemctl daemon-reload
    systemctl enable tencentcloud-pai-mate.service
    systemctl start tencentcloud-pai-mate.service
    systemctl start tencentcloud-pai-mate-update.timer 2>&1)

    status=$?
    updateProgress 90 "$msg" "$status" "paimate install"


    echo "====================pai install progress======================="
    CONFIG_INSTALL_DIR=${HOME}/pai/etc
    BINARY_INSTALL_DIR=${HOME}/pai/bin

    mkdir -p ${CONFIG_INSTALL_DIR}
    mkdir -p ${BINARY_INSTALL_DIR}

    echo "server_path: ${SERVER_PATH}" > ${CONFIG_INSTALL_DIR}/pai.yml
    echo "domain_name: ${DOMAIN_NAME}" >> ${CONFIG_INSTALL_DIR}/pai.yml

    msg=$(# Note: `agent` binary will update and run this time. `baker` binay will be run next time.
    # cannot overwrite binay, error: text busy
    # mv -f "${BINARY_INSTALL_DIR}/pai_agent" "${BINARY_INSTALL_DIR}/pai_agent.old"
    # mv -f "${BINARY_INSTALL_DIR}/pai_baker" "${BINARY_INSTALL_DIR}/pai_baker.old"
    wget -q "${SERVER_PATH}/bin/pai_agent" > "${BINARY_INSTALL_DIR}/pai_agent"
    # curl "${SERVER_PATH}/bin/pai_baker" -sSf > "${BINARY_INSTALL_DIR}/pai_baker"
    chmod +x "${BINARY_INSTALL_DIR}/pai_agent"
    # chmod +x "${BINARY_INSTALL_DIR}/pai_baker"

    rm -rf /etc/systemd/system/tencentcloud-pai-agent.service
    cat << 'EOF' > /etc/systemd/system/tencentcloud-pai-agent.service

[Unit]
Description=Tencent Cloud Pai Agent
After=network.target

[Service]
Type=simple
Restart=always
RestartSec=1
User=root
ExecStart=/root/pai/bin/pai_agent

[Install]
WantedBy=multi-user.target
EOF

    rm -rf /etc/systemd/system/tencentcloud-pai-baker.timer
    cat << 'EOF' > /etc/systemd/system/tencentcloud-pai-baker.timer

[Unit]
Description=Tencent Cloud Pai Baker

[Timer]
OnCalendar=daily
```

```
RandomizedDelaySec=5minutes
#OnCalendar=*-*-* *:*:00
Persistent=true

[Install]
WantedBy=timers.target
EOF

    systemctl daemon-reload
    systemctl enable tencentcloud-pai-agent.service
    systemctl start tencentcloud-pai-agent.service 2>&1)
    # systemctl restart tencentcloud-pai-baker.timer
    status=$?
    updateProgress 100 "$msg" "$status" "pai install"


}

installPai
```

安装完成后，打开域名:5523，用你的云主机帐号，最好root登录。其它就没有什么了，/root/pai,/root/pai-mate是程序目录 /data是数据，，测试了下，只有一个当前应用能起作用（鸡肋？）。，，并没有太深入去了解这个工程的细节。只是追求能做到可用即可。恩恩

我们的下一文，打造yet another cloudbase:在云主机上安装cloudide(jupyter)为pai面板所用

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

利用openfaas faasd在你的云主机上部署function serverless面板

本文关键字：自建云函数后端。**self build serverless function as service,single node serverless**

在前面《云主机上手动安装PAI面板》中我们讲到了在云主机上安装某种“类似baota xx语言项目管理器”的虚拟主机管理面板，也提到它并不是cloudbase版的云函数面板，后者这种方案要重得多：

function serverless最初也是由一个专家一篇文章给的思路，然后业界觉得好用就流行起来了。vs 传统虚拟主机管理面板和language backend as service，它至少有下面几个显著的不同特点：1），它将服务托管细粒化到了语言单位，即函数调用，故名faas，2），它与流行的API分离前后端结合，对这种webappdev有支持。3），它利用了devops docker, 可scaleable集群的部署，还记得我们《利用colinux打造serverfarm》一文吗？。4）运营上它支持按需按调用计费，将语言按调用次数收费。5) 它面向来自内部外部多种不同服务交互的混合云，构成的API调用环境。

> 它自动化了好多部署和开发级的东西以devops，以容器为后端，Triggers是一个重要组件，从GATEWAY代理中提取函数。根据触发从容器中fork一个process出来（因此与那些纯k8s和swarm的管理面板直接提供docker级别的服务粒度不同）。这个process就是watchdog 它是一种similar to fastCGI/HTTP的轻量web服务器，提供函数服务。由于支持多种环境多种不同服务交互，因此main_handler()中总有event指定事件来源，支持event,content为参数的async函数书写方式（而这，是nodejs的语言支持精髓）。。

综上，它是某种更倾向于“云网站管理面板”的思路。and more ...开源界的对应产品就是openfaas这类。

openfaas一般使用到k8s这种比较重的多节点docker管理器。注重集群可伸缩的云函数商用服务。那么对于个人，只是拿来装个云主机搭个博客，不想用到服务端的云函数（虽然有免费额度，不过总担心超）的用户，有没有更轻量的方案呢？

这就是faas containerd serverless without kubernetes:faasd，它其实也是一种openfaas的后端，只不过它使用containerd代替后端容器管理，因此它也可以To deploy embedded apps in IoT and edge use-cases，项目地址，http://github.com/openfaas/faasd/ ，作者甚至在树莓派上运行了它。

好了，下面在一台1h2g的云主机上来安装它，测试在ubuntu18.04下进行。

基础

以下脚本从项目的cloudinit.txt提取，有改正和修补。注意使用说明：外网访问云主机需开8080，如果提示Get http://faasd-provider:8081/namespace=: dial tcp: i/o timeout之前，把你的云主机对外的8081打开，最好都打开。

一些变量

```
MIRROR_PATH="http://default-8g95m46n2bd18f80.service.tcloudbase.com/d/demos"
# the openfaas backend
OPENFAAS_PATH=${MIRROR_PATH}/faasd
```

安装依赖

```
apt-get install nginx golang python git runc python-certbot-nginx -qq -y
不安装runc会导致containerd可能出现oci runtime error, 导致启不动faasd
```

安装faasd

1.3.5有个link错误，所以换用1.3.3。

```
# install faasd
installOpenfaasd() {

    echo "====================containerd install progress======================="
    msg=$(wget -qO- ${OPENFAAS_PATH}/containerd/v1.3.3/containerd-1.3.3-linux-amd64.tar.gz > /tmp/containerd.tar.gz && tar -xv
f /tmp/containerd.tar.gz -C /usr/local/bin/ --strip-components=1

    cat << 'EOF' > /etc/systemd/system/containerd.service

[Unit]
Description=containerd container runtime
Documentation=https://containerd.io
After=network.target local-fs.target

[Service]
ExecStartPre=-/sbin/modprobe overlay
ExecStart=/usr/local/bin/containerd

Type=notify
```

```
Delegate=yes
KillMode=process
Restart=always
# Having non-zero Limit*s causes performance problems due to accounting overhead
# in the kernel. We recommend using cgroups to do container-local accounting.
LimitNPROC=infinity
LimitCORE=infinity
LimitNOFILE=1048576
# Comment TasksMax if your systemd version does not supports it.
# Only systemd 226 and above support this version.
TasksMax=infinity

[Install]
WantedBy=multi-user.target
EOF

    systemctl daemon-reload && systemctl enable containerd
    systemctl start containerd 2>&1)
    status=$?
    updateProgress 50 "$msg" "$status" "containerd install"

    echo "====================cni install progress======================="
    msg=$(/sbin/sysctl -w net.ipv4.conf.all.forwarding=1
    mkdir -p /opt/cni/bin
    wget -qO- ${OPENFAAS_PATH}/containernetworking/v0.8.5/cni-plugins-linux-amd64-v0.8.5.tgz > /tmp/cni-plugins-linux-amd64-v0
.8.5.tgz && tar -xvf /tmp/cni-plugins-linux-amd64-v0.8.5.tgz -C /opt/cni/bin 2>&1)
    status=$?
    updateProgress 60 "$msg" "$status" "cni install"

    echo "====================faasd install progress(this may take long and finally fail due to network issues,you can manual
 fix later)======================="
    msg=$(wget -qO- ${OPENFAAS_PATH}/openfaas/faasd/0.9.2/faasd > /usr/local/bin/faasd && chmod a+x /usr/local/bin/faasd

    export GOPATH=$HOME
    rm -rf /var/lib/faasd/secrets/basic-auth-password
    rm -rf /var/lib/faasd/secrets/basic-auth-user
    rm -rf $GOPATH/go/src/github.com/openfaas/faasd

    mkdir -p $GOPATH/go/src/github.com/openfaas/
    cd $GOPATH/go/src/github.com/openfaas/ && git clone https://github.com/openfaas/faasd && cd faasd && git checkout 0.9.2
    cd $GOPATH/go/src/github.com/openfaas/faasd/ && /usr/local/bin/faasd install
    sleep 60 && systemctl status -l containerd --no-pager
    journalctl -u faasd-provider --no-pager
    systemctl status -l faasd-provider --no-pager
    systemctl status -l faasd --no-pager 2>&1)
    status=$?
    updateProgress 90 "$msg" "$status" "faasd install"


    echo "====================faas-cli install progress======================="
    msg=$(wget -qO- ${OPENFAAS_PATH}/openfaas/faas-cli/0.12.9/faas-cli > /usr/local/bin/faas-cli && chmod a+x /usr/local/bin/f
aas-cli && ln -sf /usr/local/bin/faas-cli /usr/local/bin/faas
    sleep 5 && journalctl -u faasd --no-pager
    cat /var/lib/faasd/secrets/basic-auth-password | /usr/local/bin/faas-cli login --password-stdin 2>&1)
    status=$?
    updateProgress 100 "$msg" "$status" "faas-cli install"
}
```

整个脚本跟pai安装脚本的风格很类似。可以像pai一样把nginx也整合起来作为总前端(openfaas+faasd也是前后端的一种说法)，把8080转发到nginx，要知道，nginx是通用协议转发器不只http，见《基于openresty前后端统一，生态共享的webstack实现》。

以上这些如果无误完成。在云主机上可以打开8080(faasd),8081(faasd-provider)等。打开8080需要登录。

如果打不开8080，可能是脚本faasd up时从docker.io下载的几个必要小images时timeout了。cd /var/lib/faasd/ && /usr/local/bin/faasd up（一定要观察看到几个小images下完，可能会提示8080已被占用）。重启即可访问8080。

在云主机上sudo cat /var/lib/faasd/secrets/basic-auth-password得到网关密码。用户名是admin，然后部署云函数：faas-cli store deploy figlet --env write_timeout=1s。系统可能依然会开二个实例，设成仅1个也可以。由于faas-cli都是一样的，其它相关适用的高级用法可以继续关注faasd相关文档得到。

然后，，，就是把运行在cloudbase的云函数移过来，可能需要一些补正，跑在自己的服务器上，好处是不用再担心额度了，省心省事。

不过说真的我对于这种docker做的虚拟化不放心，最好不要存数据。所以还是选择pai，未来整合pai,faas试试？

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

panel.sh：一个nginx+docker的云函和在线IDE面板,发明你自己的paas(1)

本文关键字：**Cannot connect to the Docker daemon at，containerd cannot properly do "clean-up" with shim process during start up**，用标准方法实现的类群晖**paas**，**with debugable appliance inside built**

在前面《利用openfaas faasd在你的云主机上部署function serverless面板》中我们介绍了用从https://github.com/openfaas/faasd/tree/0.9.2/cloud-config.txt提取的脚本安装openfaas（后来我们用上了0.9.5），和在云主机上使用它的方法，见《在openfaas面板上安装onemanager1，2》，如果说这3文定位主要是基本安装，排错，和调试，那么本文开始就着重于增强和提高脚本的体验了。前3文的成果和努力依旧有效。

第一个问题，脚本要能在一台干净的ubuntu1804的机器上安装，尽量一次成功，如果不能成功，那么它也要求能多次覆盖安装不致于弄坏系统。这就要求脚本中安装的组件分开，各组件包括其配置要standalone方式放置，这样可以重装时拔插和替换，覆盖。

第二个问题，虽然我前3文中从来没遇到过，但是后来的尝试中，我发现在ubuntu1804同样的安装方式和组件版本（v1.3.3containerd+cni0.4.0+cniplugins0.8.5+faasd0.9.5），居然gateway那个container开启一会之后就会停止，导致8080根本不能访问。

不废话了，直接上新的脚本：

前置

更新了安装说明。集中化了全局变量，注意deps prepare部分，bridge-utils是为了控制cni制造的那个openfaas0虚拟网卡用。安装docker.io，是ubuntu上它可以同时安装containerd1.3.3和runc

> 从Docker 1.11开始，Docker容器运行已经不是简单的通过Docker daemon来启动，而是集成了containerd、runC等多个组件。如果去搜索一番，就会发现：docker-containerd 就是 containerd，而 docker-runc 就是 runc。containerd是真正管控容器的daemon，执行容器的时候用的是runc。 为什么 要分的七零八散呢？为了防止docker一家独大,docker当年的实现被拆分出了几个标准化的模块,,标准化的目的是模块是可被其他实现替换的,其实也是为了实现类llvm的组件化可开发效果(软件抽象上，源头如果有分才能合，如果一开始就是合的就难分)。也是为了分布式效果。docker也像git一样做分布式部件化了，分布式就是设置2个部件，cliserver，这样在本地和远程都可这样架构。
>
> 而为什么是dockerio而不是docker-ce: 事实是我还发现，有些系统上，安装了docker-ce再安装containerd。会导致系统出问题Cannot connect to the Docker daemon at 。docker与containerd不兼容，所以只好安装ubuntu维护的docker.io这种解决了containerd依赖的，它默认依赖containerd和runc(不过稍后我们会提到替换升级containerd的版本)。我们选用加入了最新cn ubuntu deb src后apt-get update得到的sudo apt install docker.io=19.03.6-0ubuntu1~18.04.1，sudo apt-cache madison docker.io出来的版本.

```
#!/bin/bash

## currently tested under ubuntu1804 64b,easy to be ported to centos(can be tested with replacing apt-get and /etc/systemd/sys
tem)

## How to use this script in a cloudhost
## su root and then: ./panel.sh -d 'your domain to be binded' -m 'email you use to pass to certbot' -p 'your inital passwords'
(email and passwords are not neccessary,feed email only if you encount the "toomanyrequestofagivetype" error)
## (no prefix https/http needed,should bind to the right ip ahead for laster certbot working)


export DOMAIN_NAME=''
export EMAIL_NAME='some@some.com'
export PANEL_TYPE='0'
export PASS_INIT='5cTWUsD75ZgL3VJHdzpHLfcvJyOrUnza1jr6KXry5pXUUNmGtqmCZU4yGoc9yW4'

MIRROR_PATH="http://default-8g95m46n2bd18f80.service.tcloudbase.com/d/demos"
# the pai backend
SERVER_PATH=${MIRROR_PATH}/pai/pai-agent/stable/pai_agent_framework
PAI_MATE_SERVER_PATH=${MIRROR_PATH}/pai/pai-mate/stable/install
# the openfaas backend
OPENFAAS_PATH=${MIRROR_PATH}/faasd
# the code-server web ide
CODE_SERVER_PATH=${MIRROR_PATH}/codeserver

#install dir
INSTALL_DIR="/root/.local"
CONFIG_DIR="/root/.config"
# datadir only for pai and common data
DATA_DIR="/data"


while [[ $# -ge 1 ]]; do
    case $1 in
```

```
      -d|--domain)
        shift
        DOMAIN_NAME="$1"
        shift
        ;;
      -m|--mail)
        shift
        EMAIL_NAME="$1"
        shift
        ;;
      -t|--paneltype)
        shift
        PANEL_TYPE="$1"
        shift
        ;;
      -p|--passinit)
        shift
        PASS_INIT="$1"
        shift
        ;;
      *)
        if [[ "$1" != 'error' ]]; then echo -ne "\nInvaild option: '$1'\n\n"; fi
        echo -ne " Usage(args are self explained):\n\tbash $(basename $0)\t-d/--domain\n\t\t\t\t\-m/--mail\n\t\t\t\t\-t/--pane
ltype\n\t\t\t\t-p/--passinit\n\t\t\t\t\n"
        exit 1;
        ;;
      esac
    done

[[ "$EUID" -ne '0' ]] && echo "Error:This script must be run as root!" && exit 1;

beginTime=$(date +%s)

# write log with time
writeProgressLog() {
    echo "[`date '+%Y-%m-%d %H:%M:%S'`][$1][$2]"
    echo "[`date '+%Y-%m-%d %H:%M:%S'`][$1][$2]" >> ${DATA_DIR}/h5/access.log
}

# update install progress
updateProgress() {
    progress=$1
    message=$2
    status=$3
    installType=$4

    # echo "====================$installType progress======================="
    echo "======================$installType progress=======================" >> ${DATA_DIR}/h5/access.log
    writeProgressLog "installType" $installType
    writeProgressLog "progress" $progress
    writeProgressLog "status" $status
    echo $message >> ${DATA_DIR}/h5/access.log

    if [ $status == "0" ]; then
      code=0
      message="success"
    else
      code=1
      message="$installType error"
      # exit 1
    fi

    cat << EOF > ${DATA_DIR}/h5/progress.json
{
    "code": $code,
    "message": "$message",
    "data": {
        "installType": "$installType",
        "progress": $progress
    }
}
EOF

    if [ $status == "0" ]; then
      code=0
      message="success"
    else
      code=1
      message="$installType error"
      # exit 1
```

```
    fi

    if [ $status != "0" ]; then
      echo $message >> ${DATA_DIR}/h5/installErr.log
    fi
}

echo "====================begin .....====================="
echo "PANEL_TYPE: ${PANEL_TYPE}"
echo "DOMAIN_NAME: ${DOMAIN_NAME}"
echo "SERVER_PATH: ${MIRROR_PATH}"
echo "OPENFAAS_PATH: ${OPENFAAS_PATH}"
echo "PAI_MATE_SERVER_PATH: ${PAI_MATE_SERVER_PATH}"
echo "CODE_SERVER_PATH: ${CODE_SERVER_PATH}"
echo "INSTALL_DIR: ${INSTALL_DIR}"


rm -rf ${DATA_DIR}/h5
mkdir -p ${DATA_DIR}/h5
rm -rf ${DATA_DIR}/h5/index.json

rm -rf ${DATA_DIR}/logs
mkdir -p ${DATA_DIR}/logs

mkdir -p ${INSTALL_DIR}/bin
mkdir -p ${CONFIG_DIR}

echo "====================deps prepare progress(this may take long...)======================="
msg=$( #begin
    if [ $PANEL_TYPE == "0" ]; then

        apt-key adv --recv-keys --keyserver keyserver.Ubuntu.com 3B4FE6ACC0B21F32
        echo deb http://cn.archive.ubuntu.com/ubuntu/ bionic main restricted universe multiverse >> /etc/apt/sources.list
        echo deb http://cn.archive.ubuntu.com/ubuntu/ bionic-security main restricted universe multiverse >> /etc/apt/sources.
list
        echo deb http://cn.archive.ubuntu.com/ubuntu/ bionic-updates main restricted universe multiverse >> /etc/apt/sources.l
ist
        echo deb http://cn.archive.ubuntu.com/ubuntu/ bionic-proposed main restricted universe multiverse >> /etc/apt/sources.
list
        echo deb http://cn.archive.ubuntu.com/ubuntu/ bionic-backports main restricted universe multiverse >> /etc/apt/sources
.list
        apt-get update
        apt-get install docker.io=19.03.6-0ubuntu1~18.04.1 --no-install-recommends bridge-utils -y
        apt-get install nginx python git python-certbot-nginx -y
        # sed '1{:a;N;5!b a};$d;N;P;D' -i /etc/apt/sources.list
        # apt-get update

    else
        apt-get update && apt-get install git nginx gcc python3.6 python3-pip python3-virtualenv python-certbot-nginx golang -
y
    fi 2>&1)
status=$?
updateProgress 30 "$msg" "$status" "deps prepare"
```

基础组件代码:nginx front and docker backend

这部分虽然写死了各条转发。但重点在于如何根据具体的转发需要布置代码。这里的理论在于：如果代理服务器地址（proxy_pass后面那个）中是带有URI的，此URI会替换掉 location 所匹配的URI部分。 而如果代理服务器地址中是不带有URI的，则会用完整的请求URL来转发到代理服务器。

```
configngnix() {

    echo "====================certbot renew+start+init progress======================="
    systemctl enable nginx.service
    systemctl start nginx

    #cp -f /lib/systemd/system/certbot.service /etc/systemd/system/certbot-renew.service
    #echo '[Install]' >> /etc/systemd/system/certbot-renew.service
    #echo 'WantedBy=multi-user.target' >> /etc/systemd/system/certbot-renew.service
    #cp -f /lib/systemd/system/certbot.timer /etc/systemd/system/certbot-renew.timer

    # sed -i "s/renew/renew --nginx/g" /etc/systemd/system/certbot-renew.service


    rm -rf /etc/systemd/system/certbot-renew.service
    cat << 'EOF' > /etc/systemd/system/certbot-renew.service
```

```
[Unit]
Description=Certbot
Documentation=file:///usr/share/doc/python-certbot-doc/html/index.html
Documentation=https://letsencrypt.readthedocs.io/en/latest/
[Service]
Type=oneshot
ExecStart=/usr/bin/certbot -q renew
PrivateTmp=true
[Install]
WantedBy=multi-user.target
EOF

    rm -rf /etc/systemd/system/certbot-renew.timer
    cat << 'EOF' > /etc/systemd/system/certbot-renew.timer

[Unit]
Description=Run certbot twice daily

[Timer]
OnCalendar=*-*-* 00,12:00:00
RandomizedDelaySec=43200
Persistent=true

[Install]
WantedBy=timers.target
EOF

    msg=$(
    #first time renew
    certbot certonly --quiet --standalone --agree-tos --non-interactive -m ${EMAIL_NAME} -d ${DOMAIN_NAME} --pre-hook "systemc
tl stop nginx"

    systemctl daemon-reload
    systemctl enable certbot-renew.service
    systemctl start certbot-renew.service
    systemctl start certbot-renrew.timer 2>&1)
    status=$?
    updateProgress 40 "$msg" "$status" "certbot renew+start+init"


    echo "=====================nginx reconfig progress======================="
    # add nginx conf
    rm -rf /etc/nginx/conf.d/default.conf
    cat << 'EOF' > /etc/nginx/conf.d/default.conf

server {
    listen 443 http2 ssl;
    listen [::]:443 http2 ssl;

    server_name DOMAIN_NAME;

    ssl on;
    ssl_certificate /etc/letsencrypt/live/DOMAIN_NAME/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/DOMAIN_NAME/privkey.pem;
    ssl_session_timeout 5m;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256:HIGH:!aNULL:!MD5:!RC4:!DHE;
    ssl_prefer_server_ciphers on;

    location / {
      proxy_pass http://localhost:PORT;

      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection upgrade;
      proxy_set_header Accept-Encoding gzip;
    }

    location /pai/ {
      proxy_pass http://localhost:5523;

      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection upgrade;
      proxy_set_header Accept-Encoding gzip;
    }

    location /faasd/ {
      proxy_pass http://localhost:8080/ui/;

      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection upgrade;
```

```
      proxy_set_header Accept-Encoding gzip;
    }

    location /codeserver/ {
      proxy_pass http://localhost:5000/;
      proxy_redirect http:// https://;
      proxy_set_header Host $host:443/codeserver;

      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection upgrade;
      proxy_set_header Accept-Encoding gzip;
    }

}

server {
    listen 80;
    server_name DOMAIN_NAME;

    if ($host = DOMAIN_NAME) {
        return 301 https://$host$request_uri;
    }

    return 404;
}
EOF

    sed -i "s#DOMAIN_NAME#${DOMAIN_NAME}#g" /etc/nginx/conf.d/default.conf

    if [ $PANEL_TYPE == "0" ]; then
        sed -i "s#PORT#8080/function/#g" /etc/nginx/conf.d/default.conf
    else
        sed -i "s#PORT#3000#g" /etc/nginx/conf.d/default.conf
    fi



    # restart nginx
    msg=$( #begin
    [[ $(systemctl is-active nginx.service) == "activating" ]] && systemctl reload nginx.service
    systemctl restart nginx 2>&1)
    status=$?
    updateProgress 50 "$msg" "$status" "nginx reconfig"

}

confignginx
```

为了让docker能覆盖安装，接下来脚本开头处逻辑的清空了配置，这里的重点问题是containerd与cni，与openfaasd的复杂关系：

Container Network Interface (CNI) 最早是由CoreOS发起的容器网络规范，是Kubernetes网络插件的基础。其基本思想为：Container Runtime在创建容器时，先创建好network namespace，然后调用CNI插件为这个netns配置网络，其后再启动容器内的进程。现已加入CNCF，成为CNCF主推的网络模型。CNI负责了在容器创建或删除期间的所有与网络相关的操作，它将创建所有规则以确保从容器进和出的网络连接正常，但它并不负责设置网络介质，例如创建网桥或分发路由以连接位于不同主机中的容器。

这个工作由openfaasd等完成。docker的这些组件->containerd+cni+ctr+runc，是由faasd来配置运行的。单独启动第一次安装完的containerd+cni+ctr+runc并不会启动cni和开启网卡(单独启动containerd提示cni conf not found没关系它依然会启动)，需要openfaasd中的动作给后者带来cni和网卡配置。但这种结合很紧密，使得接下来容器的完全清理工作有难度。

对于容器的清除，用ctr tasks kill && ctr tasks delete && ctr container delete可以看到ps aux|grep manual看到主机空间的shim任务和/proc/id号/ns都被删掉了，但还是某些地方有残留。这是因为这二者很难分开,shim开启的task关联容器和/var/run/containerd无法清理，导致前者很难单独拔插/进行配置卸载，也难于在下一次覆盖安装时能从0全新开始。

而这其实是一个bug导致的,containerd cannot properly do "clean-up" with shim process during start up? #3971（https://github.com/containerd/containerd/issues/3971），直到1.4.0beta才被解决（https://github.com/containerd/containerd/pull/4100/commits/488d6194f2080709d9667e00ff244fbdc7ff95b2），但我测试了(cd /var/lib/faasd/ faasd up)，只是效果好点，1.3.3是提示id exists不能重建container，1.40是提示/run/container下的files exists，同样没解决完全清理以全新覆盖安装containerd的需求，所以我脚本中提示了“containerd install+start progress(this may hang long and if you over install the script you may encount /run/containerd device busy error,for this case you need to reboot to fix after scripts finished”，这个基本如果你遇到了var/run删不掉错误，等安装程序跑完，重启即可。

因此我选择了1.40的containerd，它也同时解决了我开头提到的，gateway失效的问题。用的cni plugins还是0.8.5，本来想用那个cri-containerd-cni-1.4.0-linux-amd64.tar.gz,但里面的cni是0.7.1，与faasd要求的0.4.0不符。

对于cni的卸载和清除，则不属于ctr的能控制范畴，cni没有宿主机上的控制，除非将进程网络命名空间恢复到主机目录，或在在容器网络空间内运行IP命令来检查网络接口是否已正确设置，都挺麻烦，用上面删容器的ctr tasks kill && ctr tasks delete && ctr container delete三部曲删可以看到ifconfig中五个task对应的虚拟网卡也被干掉了，所以我也就没有再深入研究cni的卸载逻辑。

```
configdocker() {



    [[ $(systemctl is-active faasd-provider) == "activating" ]] && systemctl stop faasd-provider
    [[ $(systemctl is-active faasd) == "activating" ]] && systemctl stop faasd
    [[ $(systemctl is-active containerd) == "activating" ]] && ctr image remove docker.io/openfaas/basic-auth-plugin:0.18.18 d
ocker.io/library/nats-streaming:0.11.2 docker.io/prom/prometheus:v2.14.0 docker.io/openfaas/gateway:0.18.18 docker.io/openfaas
/queue-worker:0.11.2  && for i in basic-auth-plugin nats prometheus gateway queue-worker; do ctr tasks kill -s SIGKILL $i;ctr
tasks delete $i;ctr container delete $i; done && systemctl stop containerd && sleep 10
    ps -ef|grep containerd|awk '{print $2}'|xargs kill -9
    rm -rf /var/run/containerd /run/containerd

    [[ ! -z "$(brctl show|grep openfaas0)" ]] && ifconfig openfaas0 down && brctl delbr openfaas0
    rm -rf /etc/cni

    echo "================================cniplugins installonly================================="
    msg=$( #begin

    if [ ! -f "/tmp/cni-plugins-linux-amd64-v0.8.5.tar.gz" ]; then
        wget --no-check-certificate -qO- ${MIRROR_PATH}/docker/containernetworking/plugins/v0.8.5/cni-plugins-linux-amd64-v0.8
.5.tar.gz > /tmp/cni-plugins-linux-amd64-v0.8.5.tar.gz
    fi

    mkdir -p /opt/cni/bin
    tar -xf /tmp/cni-plugins-linux-amd64-v0.8.5.tar.gz -C /opt/cni/bin

    /sbin/sysctl -w net.ipv4.conf.all.forwarding=1 2>&1)
    status=$?
    updateProgress 50 "$msg" "$status" "cniplugins installonly"


    echo "======containerd install+start progress(this may hang long and if you over install the script you may encount /run/c
ontainerd device busy error,for this case you need to reboot to fix after scripts finished)====="
    msg=$( #begin
    # del original deb by docker.io
    rm -rf /usr/bin/containerd* /usr/bin/ctr

    # replace with new bins
    if [ ! -f "/tmp/containerd-1.4.0-linux-amd64.tar.gz" ]; then
        wget --no-check-certificate -qO- ${MIRROR_PATH}/docker/containerd/v1.4.0/containerd-1.4.0-linux-amd64.tar.gz > /tmp/co
ntainerd-1.4.0-linux-amd64.tar.gz
    fi

    tar -xf /tmp/containerd-1.4.0-linux-amd64.tar.gz -C ${INSTALL_DIR}/bin/ --strip-components=1 && ln -sf ${INSTALL_DIR}/bin/
containerd* /usr/local/bin/ && ln -sf ${INSTALL_DIR}/bin/ctr /usr/local/bin/ctr

    rm -rf /etc/systemd/system/containerd.service
    cat << 'EOF' > /etc/systemd/system/containerd.service

[Unit]
Description=containerd container runtime
Documentation=https://containerd.io
After=network.target local-fs.target
#After=network.target containerd.socket containerd.service
#Requires=containerd.socket containerd.service

[Service]
ExecStartPre=-/sbin/modprobe overlay
ExecStart=/usr/local/bin/containerd

Type=notify
Delegate=yes
KillMode=process
#changed to mixed to let systemctl stop containerd kill shims
#KillMode=mixed
Restart=always
# Having non-zero Limit*s causes performance problems due to accounting overhead
# in the kernel. We recommend using cgroups to do container-local accounting.
LimitNPROC=infinity
LimitCORE=infinity
LimitNOFILE=1048576
# Comment TasksMax if your systemd version does not supports it.
# Only systemd 226 and above support this version.
TasksMax=infinity
```

```
[Install]
WantedBy=multi-user.target
EOF

    systemctl daemon-reload && systemctl enable containerd
    systemctl start containerd --no-pager 2>&1)
    status=$?
    updateProgress 50 "$msg" "$status" "containerd install+start"



}

configdocker
```

未来等containerd的这个bug彻底解决或许有可能让containerd的shim task实现彻底停止和移除。来说点别的，还记得我在《enginx》中对openresty可以脚本编程转发连结游戏服务器的多组件集群，形成demo based programming的能力的设想吗(类似组成openfaas的五个containers，是组建一个单节点集群分布式的典型职责单位。有验证有网关，有业务)。还有基于jupyter的engitor，那么现在，我们用openfaas+vscodeonline来实现它们。我们知道openfaas这种就是构建一个分布式函数的“城市”，让城市组成的世界在二进制级，相互调用分布式API，进行demo组合，构成应用。是真正的demo积木编程。因为它可以Turn Any CLI into a Function，甚至是本地native cli。比如它能使shell完全变成分布式语言。直接在二进制上编程。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

panel.sh：一个nginx+docker的云函和在线IDE面板,发明你自己的paas(2)

继前文，这里介绍faasd和pai二个后端安装启动逻辑，也直接放代码：

faasd

这里主要是将原来cloud-config.txt中cd git source root,faasd install替换成，经分析source/cmd/install.go后得到的几个静态文件，并将它们直接放进代码中，这样可以免除编译代码直接全部静态资源加脚本逻辑处理。

注意几处，1，/var/lib/faasd的权限一定要设成0755，而不是cmd/install.go中指定的0644。否则containerd启动不了，它要写hosts文件，以调用docker cni。2，结尾服务启动时，须留足充足的时间让images都下载完，只有下载完镜像才会启动网卡的。加了一些关于镜像和虚拟网卡是否准备完毕和唤起的判断，重点是离线镜像。

由于影响脚本失败的地方和可变因素主要来自镜像的拉取（及之后cni网络唤起，faasd up建立cni有一定机率让openfaas0网络出现很慢。），故设置了预拉取离线镜像并判断（本来之前计划是预配置cni，但与镜像部分结合紧密，且后来考虑到镜像拉取如果OK，cni过程应该无须再考虑）。使得onedrive+om可代替docker register仓库（github也有一个git bundle create offline-repos master，这对应我在《在openfaas面板上安装onemanager》文尾把git和docker仓库静态化的设想）。

```
# install faasd
installOpenfaasd() {

    [[ $(systemctl is-active faasd-provider) == "activating" ]] && systemctl stop faasd-provider
    [[ $(systemctl is-active faasd) == "activating" ]] && systemctl stop faasd

    ps -ef|grep faasd|awk '{print $2}'|xargs kill -9
    rm -rf ${INSTALL_DIR}/bin/faas*
    rm -rf /var/lib/faasd /var/lib/faasd-provider

    echo "=====faasd install+start progress(this may finally fail due to incorrect folder permissions or network issues,you ca
n maunal fix it later with systemctl restart faasd)====="
    msg=$( # begin

    if [ ! -f "/tmp/faas-cli" ]; then
        wget --no-check-certificate -qO- ${OPENFAAS_PATH}/faasd/0.9.5/faasd > /tmp/faasd
    fi

    cp /tmp/faasd ${INSTALL_DIR}/bin/faasd && chmod a+x ${INSTALL_DIR}/bin/faasd && ln -sf ${INSTALL_DIR}/bin/faasd /usr/local
/bin/faasd


    #export GOPATH=$HOME
    # ......
    #systemctl status -l faasd-provider --no-pager
    #systemctl status -l faasd --no-pager

    mkdir -p /var/lib/faasd && chmod 0755 /var/lib/faasd && mkdir -p /var/lib/faasd-provider && chmod 0755 /var/lib/faasd-prov
ider && mkdir -p /var/lib/faasd/secrets


    rm -rf /var/lib/faasd/docker-compose.yaml
    cat << 'EOF' > /var/lib/faasd/docker-compose.yaml

version: "3.7"
services:
  basic-auth-plugin:
    image: "docker.io/openfaas/basic-auth-plugin:0.18.18ARCH_SUFFIX"
    environment:
      - port=8080
      - secret_mount_path=/run/secrets
      - user_filename=basic-auth-user
      - pass_filename=basic-auth-password
    volumes:
      # we assume cwd == /var/lib/faasd
      - type: bind
        source: ./secrets/basic-auth-password
        target: /run/secrets/basic-auth-password
      - type: bind
        source: ./secrets/basic-auth-user
        target: /run/secrets/basic-auth-user
    cap_add:
```

```
    - CAP_NET_RAW

nats:
  image: docker.io/library/nats-streaming:0.11.2
  command:
    - "/nats-streaming-server"
    - "-m"
    - "8222"
    - "--store=memory"
    - "--cluster_id=faas-cluster"
  # ports:
  #    - "127.0.0.1:8222:8222"

prometheus:
  image: docker.io/prom/prometheus:v2.14.0
  volumes:
    - type: bind
      source: ./prometheus.yml
      target: /etc/prometheus/prometheus.yml
  cap_add:
    - CAP_NET_RAW
  ports:
     - "127.0.0.1:9090:9090"

gateway:
  image: "docker.io/openfaas/gateway:0.18.18ARCH_SUFFIX"
  environment:
    - basic_auth=true
    - functions_provider_url=http://faasd-provider:8081/
    - direct_functions=false
    - read_timeout=60s
    - write_timeout=60s
    - upstream_timeout=65s
    - faas_nats_address=nats
    - faas_nats_port=4222
    - auth_proxy_url=http://basic-auth-plugin:8080/validate
    - auth_proxy_pass_body=false
    - secret_mount_path=/run/secrets
    - scale_from_zero=true
  volumes:
    # we assume cwd == /var/lib/faasd
    - type: bind
      source: ./secrets/basic-auth-password
      target: /run/secrets/basic-auth-password
    - type: bind
      source: ./secrets/basic-auth-user
      target: /run/secrets/basic-auth-user
  cap_add:
    - CAP_NET_RAW
  depends_on:
    - basic-auth-plugin
    - nats
    - prometheus
  ports:
     - "8080:8080"

queue-worker:
  image: docker.io/openfaas/queue-worker:0.11.2
  environment:
    - faas_nats_address=nats
    - faas_nats_port=4222
    - gateway_invoke=true
    - faas_gateway_address=gateway
    - ack_wait=5m5s
    - max_inflight=1
    - write_debug=false
    - basic_auth=true
    - secret_mount_path=/run/secrets
  volumes:
    # we assume cwd == /var/lib/faasd
    - type: bind
      source: ./secrets/basic-auth-password
      target: /run/secrets/basic-auth-password
    - type: bind
      source: ./secrets/basic-auth-user
      target: /run/secrets/basic-auth-user
  cap_add:
    - CAP_NET_RAW
  depends_on:
    - nats
```

```
EOF

    sed -i "s#ARCH_SUFFIX##g" /var/lib/faasd/docker-compose.yaml

    rm -rf /var/lib/faasd/prometheus.yml
    cat << 'EOF' > /var/lib/faasd/prometheus.yml

# my global config
global:
  scrape_interval:     15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      # - alertmanager:9093

# Load rules once and periodically evaluate them according to the global 'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
    - targets: ['localhost:9090']

  - job_name: 'gateway'
    static_configs:
    - targets: ['gateway:8082']
EOF

    rm -rf /var/lib/faasd/resolv.conf
    cat << 'EOF' > /var/lib/faasd/resolv.conf
nameserver 8.8.8.8
EOF

    rm -rf /var/lib/faasd-provider/resolv.conf
    cat << 'EOF' > /var/lib/faasd-provider/resolv.conf
nameserver 8.8.8.8
EOF

    rm -rf /var/lib/faasd/secrets/basic-auth-user
    cat << 'EOF' > /var/lib/faasd/secrets/basic-auth-user
admin
EOF

    rm -rf /var/lib/faasd/secrets/basic-auth-password
    cat << 'EOF' > /var/lib/faasd/secrets/basic-auth-password
PLEASECORRECTME
EOF

    sed -i "s#PLEASECORRECTME#${PASS_INIT}#g" /var/lib/faasd/secrets/basic-auth-password

    rm -rf /etc/systemd/system/faasd-provider.service
    cat << 'EOF' > /etc/systemd/system/faasd-provider.service

[Unit]
Description=faasd-provider

[Service]
MemoryLimit=500M
Environment="secret_mount_path=/var/lib/faasd/secrets"
Environment="basic_auth=true"
ExecStart=/usr/local/bin/faasd provider
Restart=on-failure
RestartSec=10s
WorkingDirectory=/var/lib/faasd-provider
# add
# User=root

[Install]
WantedBy=multi-user.target
EOF
```

```
    rm -rf /etc/systemd/system/faasd.service
    cat << 'EOF' > /etc/systemd/system/faasd.service

[Unit]
Description=faasd
After=faasd-provider.service

[Service]
MemoryLimit=500M
ExecStart=/usr/local/bin/faasd up
Restart=on-failure
RestartSec=10s
WorkingDirectory=/var/lib/faasd
# add
# User=root

[Install]
WantedBy=multi-user.target
EOF

    # in saftey,first downloads image and import them offline by advance
    if [ ! -f "/tmp/faasd-containers.tar.gz" ]; then
        wget --no-check-certificate -qO- ${OPENFAAS_PATH}/faasd/0.9.5/faasd-containers.tar.gz > /tmp/faasd-containers.tar.gz &
& tar -xf /tmp/faasd-containers.tar.gz  -C /tmp
    fi
    ctr --address=/run/containerd/containerd.sock image import /tmp/faasd-containers/basic-auth-plugin-0.18.18.tar
    ctr --address=/run/containerd/containerd.sock image import /tmp/faasd-containers/nats-streaming-0.11.2.tar
    ctr --address=/run/containerd/containerd.sock image import /tmp/faasd-containers/gateway-0.18.18.tar
    ctr --address=/run/containerd/containerd.sock image import /tmp/faasd-containers/queue-worker-0.11.2.tar
    ctr --address=/run/containerd/containerd.sock image import /tmp/faasd-containers/prometheus-v2.14.0.tar

    systemctl daemon-reload && systemctl enable faasd-provider faasd
    systemctl start faasd-provider --no-pager
    systemctl start faasd --no-pager  2>&1)
    status=$?
    updateProgress 65 "$msg" "$status" "faasd install+start"

    # then, wait extra 30 for cni taking effort
    sleep 30

    # finally check to ensure every container services and cni network being ready
    for i in 1 2 3; do [[ ! -z "$(ctr image list|grep basic-auth-plugin)" ]] && break;sleep 3;echo "checking basic-auth ($i),i
f failed at 3,it may require a reboot"; done
    for i in 1 2 3; do [[ ! -z "$(ctr image list|grep nats)" ]] && break;sleep 3;echo "checking nats ($i),if failed at 3,it ma
y require a reboot"; done
    for i in 1 2 3; do [[ ! -z "$(ctr image list|grep prometheus)" ]] && break;sleep 3;echo "checking prometheus ($i),if faile
d at 3,it may require a reboot"; done
    for i in 1 2 3; do [[ ! -z "$(ctr image list|grep gateway)" ]] && break;sleep 3;echo "checking gateway ($i),failed at 3,it
 may require a reboot"; done
    for i in 1 2 3; do [[ ! -z "$(ctr image list|grep queue-worker)" ]] && break;sleep 3;echo "checking queueworker ($i),if fa
iled at 3,it may require a reboot"; done
    for i in 1 2 3 4 5; do [[ ! -z "$(brctl show|grep openfaas0)" ]] && break;sleep 3;echo "checking openfaas0 ($i),if failed
at 5,it may require a reboot"; done


    echo "====================faas-cli install+login progress======================="
    msg=$( #begin

    if [ ! -f "/tmp/faas-cli" ]; then
        wget --no-check-certificate -qO- ${OPENFAAS_PATH}/faas-cli/0.12.9/faas-cli > /tmp/faas-cli
    fi

    cp /tmp/faas-cli ${INSTALL_DIR}/bin/faas-cli && chmod a+x ${INSTALL_DIR}/bin/faas-cli && ln -sf ${INSTALL_DIR}/bin/faas-cl
i /usr/local/bin/faas && ln -sf ${INSTALL_DIR}/bin/faas-cli /usr/local/bin/faas-cli

    mkdir -p /var/lib/faasd/.docker
    ln -sf ~/.docker/config.json /var/lib/faasd/.docker/config.json

    sleep 5 && journalctl -u faasd --no-pager
    cat /var/lib/faasd/secrets/basic-auth-password | /usr/local/bin/faas-cli login --password-stdin 2>&1)
    status=$?
    updateProgress 70 "$msg" "$status" "faas-cli install+login"
}
```

pai

然后是pai的了,感觉没什么好说，就是把node跟vscode一样处理，将语言做进应用。因为node是带modules发布的。node是一个bootstrap，不像go的containerd，cni这些，是单个exe。我把pai整合了一下：把node-v10.16.2-linux-x64和pm2-3.5.1.tgz,serve-handler,sqlite3-4.1.1.tgz整合了，再把pai-mate-latest.tar.xz和node_modules.tar.xz整合了。最后把二者顶级文件夹整合的结果形成新的pai-mate-latest.tar.xz。这样也方便简化pai的脚本逻辑

```
# install paiserver
installPai() {

    [[ $(systemctl is-active tencentcloud-pai-mate) == "activating" ]] && systemctl stop tencentcloud-pai-mate
    [[ $(systemctl is-active tencentcloud-pai-mate-update) == "activating" ]] && systemctl stop tencentcloud-pai-mate-update
    [[ $(systemctl is-active tencentcloud-pai-agent) == "activating" ]] && systemctl stop tencentcloud-pai-agent
    rm -rf ${INSTALL_DIR}/pai ${INSTALL_DIR}/pai-mate

    echo "====================paimate install+start progress======================="
    msg=$(
    # prepare directory
    mkdir -p ${INSTALL_DIR}/pai-mate

    # prepare
    mkdir -p ${INSTALL_DIR}/pai
    echo "export PATH=/root/.local/pai-mate/bin:$PATH" > ${INSTALL_DIR}/pai/pai-mate-env
    source ${INSTALL_DIR}/pai/pai-mate-env # get node/npm binary path

    # prepare workspace
    mkdir -p ${DATA_DIR}/pai_mate_workspaces

    # download package
    if [ ! -f "/tmp/pai-mate-latest.tar.xz" ]; then
        wget --no-check-certificate -qO- ${PAI_MATE_SERVER_PATH}/pai-mate-latest.tar.xz > /tmp/pai-mate-latest.tar.xz
    fi

    # unzip
    tar -Jxf /tmp/pai-mate-latest.tar.xz  -C ${INSTALL_DIR}/pai-mate
    mv /tmp/pai-mate-latest.tar.xz /tmp/pai-mate-latest.tar.xz.old

    cd ${INSTALL_DIR}/pai-mate

    # config
    echo "UPDATE_PATH: ${PAI_MATE_SERVER_PATH}" > config.yml
    echo "DOMAIN_NAME: ${DOMAIN_NAME}" >> config.yml
    echo "CERT_PATH: /etc/letsencrypt/live/${DOMAIN_NAME}/fullchain.pem" >> config.yml
    echo "KEY_PATH: /etc/letsencrypt/live/${DOMAIN_NAME}/privkey.pem" >> config.yml

    #npm install --production --unsafe-perm=true --allow-root
    npm run migrate:latest

    # systemd service start
    rm -rf /etc/systemd/system/tencentcloud-pai-mate.service
    cat << 'EOF' > /etc/systemd/system/tencentcloud-pai-mate.service

[Unit]
Description=Tencent Cloud Pai Mate
After=network.target

[Service]
Type=simple
Restart=always
RestartSec=1
User=root
Environment=PATH=/root/.local/pai-mate/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
WorkingDirectory=/root/.local/pai-mate
ExecStart=/root/.local/pai-mate/bin/start.sh

[Install]
WantedBy=multi-user.target
EOF

    rm -rf /etc/systemd/system/tencentcloud-pai-mate-update.service
    cat << 'EOF' > /etc/systemd/system/tencentcloud-pai-mate-update.service

[Unit]
Description=Tencent Cloud Pai Mate Update
After=network.target

[Service]
Type=oneshot
User=root
Environment=PATH=/root/.local/pai-mate/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
WorkingDirectory=/root/.local/pai-mate
ExecStart=/root/.local/pai-mate/bin/update.sh
```

```
EOF

    rm -rf /etc/systemd/system/tencentcloud-pai-mate-update.timer
    cat << 'EOF' > /etc/systemd/system/tencentcloud-pai-mate-update.timer

[Unit]
Description=Tencent Cloud Pai Mate Update

[Timer]
OnCalendar=daily
RandomizedDelaySec=5minutes
Persistent=true

[Install]
WantedBy=timers.target
EOF

    chmod +x ${INSTALL_DIR}/pai-mate/bin/*
    systemctl daemon-reload
    systemctl enable tencentcloud-pai-mate.service
    systemctl start tencentcloud-pai-mate.service
    systemctl start tencentcloud-pai-mate-update.timer 2>&1)

    status=$?
    updateProgress 90 "$msg" "$status" "paimate install+start"


    echo "====================pai install+start progress======================="


    msg=$(#begin
    mkdir -p ${INSTALL_DIR}/pai/etc
    mkdir -p ${INSTALL_DIR}/pai/bin

    echo "server_path: ${SERVER_PATH}" > ${INSTALL_DIR}/pai/etc/pai.yml
    echo "domain_name: ${DOMAIN_NAME}" >> ${INSTALL_DIR}/pai/etc/pai.yml

    # Note: `agent` binary will update and run this time. `baker` binay will be run next time.
    # cannot overwrite binay, error: text busy
    # mv -f "${INSTALL_DIR}/pai/bin/pai_agent" "${INSTALL_DIR}/pai/bin/pai_agent.old"
    # mv -f "${INSTALL_DIR}/pai/bin/pai_baker" "${INSTALL_DIR}/pai/bin/pai_baker.old"
    wget --no-check-certificate -qO- "${SERVER_PATH}/bin/pai_agent" > "${INSTALL_DIR}/pai/bin/pai_agent"
    # curl "${SERVER_PATH}/bin/pai_baker" -sSf > "${INSTALL_DIR}/pai/bin/pai_baker"
    chmod +x "${INSTALL_DIR}/pai/bin/pai_agent"
    # chmod +x "${INSTALL_DIR}/pai/bin/pai_baker"

    rm -rf /etc/systemd/system/tencentcloud-pai-agent.service
    cat << 'EOF' > /etc/systemd/system/tencentcloud-pai-agent.service

[Unit]
Description=Tencent Cloud Pai Agent
After=network.target

[Service]
Type=simple
Restart=always
RestartSec=1
User=root
ExecStart=/root/.local/pai/bin/pai_agent

[Install]
WantedBy=multi-user.target
EOF

    rm -rf /etc/systemd/system/tencentcloud-pai-baker.timer
    cat << 'EOF' > /etc/systemd/system/tencentcloud-pai-baker.timer

[Unit]
Description=Tencent Cloud Pai Baker

[Timer]
OnCalendar=daily
RandomizedDelaySec=5minutes
#OnCalendar=*-*-* *:*:00
Persistent=true

[Install]
WantedBy=timers.target
EOF
```

```
    systemctl daemon-reload
    systemctl enable tencentcloud-pai-agent.service
    systemctl start tencentcloud-pai-agent.service 2>&1)
    # systemctl restart tencentcloud-pai-baker.timer
    status=$?
    updateProgress 100 "$msg" "$status" "pai install+start"


}

if [ $PANEL_TYPE == "0" ]; then
    installOpenfaasd
else
    installPai
fi
```

vscodeonline和结束部分

```
# install codeserver
installCodeserver() {

    [[ $(systemctl is-active code-server) == "activating" ]] && systemctl stop code-server
    rm -rf ${INSTALL_DIR}/bin/code-server ${INSTALL_DIR}/lib/code-server-3.5.0 ${CONFIG_DIR}/code-server

    echo "====================codeserver install+start progress======================="
    msg=$(# begin

    mkdir -p ${INSTALL_DIR}/lib/code-server-3.5.0 ${INSTALL_DIR}/bin ${CONFIG_DIR}/code-server

    if [ ! -f "/tmp/code-server-3.5.0-linux-amd64.tar.gz" ]; then
        wget --no-check-certificate -qO- ${CODE_SERVER_PATH}/v3.5.0/code-server-3.5.0-linux-amd64.tar.gz > /tmp/code-server-3.
5.0-linux-amd64.tar.gz
    fi

    tar  -xf /tmp/code-server-3.5.0-linux-amd64.tar.gz -C ${INSTALL_DIR}/lib/code-server-3.5.0 --strip-components=1
    ln -s ${INSTALL_DIR}/lib/code-server-3.5.0/bin/code-server /usr/local/bin/code-server

    # systemd service start
    rm -rf ${CONFIG_DIR}/code-server/config.yaml
    cat << 'EOF' > ${CONFIG_DIR}/code-server/config.yaml

bind-addr: 127.0.0.1:5000
auth: password
password: PLEASECORRECTME
cert: false
EOF

    sed -i "s#PLEASECORRECTME#${PASS_INIT}#g" ${CONFIG_DIR}/code-server/config.yaml

    # systemd service start
    rm -rf /etc/systemd/system/code-server.service
    cat << 'EOF' > /etc/systemd/system/code-server.service

[Unit]
Description=code-server
After=network.target

[Service]
Type=exec
ExecStart=/usr/local/bin/code-server
Restart=always
User=root

[Install]
WantedBy=default.target
EOF

    systemctl daemon-reload && systemctl enable code-server
    systemctl start code-server 2>&1)
    status=$?
    updateProgress 100 "$msg" "$status" "code-server install+start"
}

installCodeserver

echo "====================finished .....====================="
if [ $PANEL_TYPE == "0" ]; then
```

```
    echo "the final admin panel url you can access(login with user name admin and passwd given): https://${DOMAIN_NAME}/faasd/
,password is '$(cat /var/lib/faasd/secrets/basic-auth-password)',thevscodewebide server at:https://${DOMAIN_NAME}/codeserver/,
password is ${PASS_INIT},thank you!!"
else
    echo "the final admin panel url you can access(login with your valid cvm account): https://${DOMAIN_NAME}/pai/,datadir is
${DATA_DIR},thevscodewebide server at:https://${DOMAIN_NAME}/codeserver/,password is ${PASS_INIT},thank you!!"
fi


# count time
endTime=$(date +%s)
echo 'Total Time Spent: '$((endTime-beginTime))'s'
```

如果安装后你更改了密码，可以systemctl restart faasd faasd-provider containerd使之生效。刚刚收到短信20201012,腾讯scf免费要改为一年之后收费了，还是自备云函的好

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一个fully retryable的rootbuild packer脚本,从0打造matecloudos(1):实现compiletc tools

本文关键字：云**packer**类**cloudinit,pebuilder.sh**本地版，**tc as general rootbuild,**子**shell**启动脚本,可在**cloudide terminal**下运行，**bash** 数组 包含反引号命令替换会被执行，**bash**将任意命令放在数组中却能正常调用的方法**,bash** 命令字符串**cmstr**换行显示

在生成黑群，黑苹果的文章中，我们用的都是在deepin+kvm/qemu的架构中进行的。除了依赖客户机desktop环境其导出镜像的过程也不是纯粹自动化的（vs命令行下一条命令跑到底的方式）。在以前《利用hashicorp packer把dbcolinux导出为虚拟机和docker格式》系列文章和buildtc系列脚本中，我们用的是packer+本地虚拟机的方式，这种方式可以一条命令跑到底得到最终镜像。但同样依赖具体OS下的具体虚拟机还有一个iso(这相当于pebuilder.sh packer开始时的主机os)，这二种方式都存在环境条件假定，比如你不能在一台无法开kvm虚拟也没有桌面的linux服务器上进行，在前面pebuilder.sh中我们描述过一种debianinstall os，pebuilder.sh里面的脚本其实只是预处理和打包，其实真正的pack(provision)过程在于启动时的喂给它的那段preseed.cfg as boot command，借助pkg仓库这成为在线安装os/使os变得可在线化安装的良好方法(非dd方式)但它局限于debian且不是预生成方式，最后，以上都不是从0的源码构建的。

那么为了追求从0开始更彻底更可控的效果，有没有一种纯云上packer生成并从源码构建并得到offline镜像结果并为pebuilder.sh所用的方式呢？------ 即三个要求，1，在服务器环境和纯命令行中也能进行，2，offline预生成镜像，3源码编译的方式进行 ------ 所以现在，让我们来探求在任何环境下命令行生成镜像的方法，使用通用跨平台packer+qemu的组合(qemu独立使用是一个通用虚拟机而以不跟kvm一起)，服务于本文目的：从0实现tclinux based mateclous离线镜像。这段代码取自http://mirrors.163.com/tinycorelinux/11.x/x86_64/release/src/toolchain/compile_tc11_x86_64。,与前面面向cross compile和分离目录定制的方向不同，这里只面向在64tc11从0开始源码自举构建一个干净的64tc11。

思路是将compile_tc11_x86_64中所有编译目标弄成一个查表的参数数组，然后统一调用编译，为什么一定要弄成数组呢，这样可以清晰化所有的pass。

在云主机装上packer和qemu(我用的>2G ubt1804,编译gcc需要多点内存,packer为以前文章的1.4.1版本没变，qemu为apt-get install出来的qemu4)，根据上面的代码，准备所有的源码文件和目录，新建一个rootbuild.sh里面是packer build -force -on-error=ask ./matecloudos-$1.packer，以后有多个packer只需换后缀参数就可以了，本文是matecloudos-tools.packer，其它的packer逻辑在另外的文章中按需给出(设立多个packer不致于让所有工作都放在一步，可以持续集成，当然在一个packer中设好retry也是一样但这需要更多工作)，(等所有的packer都能成功可考虑合成一个packer)。

运行代码rootbuild.sh packer文件名，会执行本文的主体代码matecloudos-tools.packer，（如果你是tnt-pd16(用的sudo修复网络启动的pd)，需要sudo build.sh xxxx才能唤起prctl window view），直接给代码：

matecloudos-tools.packer：压缩boot-command和切换shell

qemu相当于qemu-iso，其实qemu支持linux as firmware(xhyve也支持)，即参数中直接喂入-kernel,-initrd,-append，我们可以实现直接用linux作为iso，然后通过 qemuargs = [["-kernel", "/boot/xxx"],["-initrd", "/boot/ixxx"],["-append", " 'xxx'"]]喂入，但packer规定必须要有一个iso启动不能完全控制启动过程。所以不准备模拟这效果了

因为我们定位于在服务器和任意网络环境生成，所以所有资源全是wgetable的。还有个问题是：因为它是打字形式的命令输入，不好处理要协调延时，所以写短一点省事，本来bootcomomand phase可以写很多。这次不行,,所以这样的故意延时很重要，因为iso中并没有安装kvm等加速器，在boot阶段大量启动显示信息会很慢。这也使得bootcommand要尽量短。，能放的都放到接下来的provisioners中去,这对整个脚本-on-error-ask后选择能有机会retry而避免涉及到格盘这种操作也是一种好的支持。压缩bootcommand也是为了将一切可能的输出结果转移到provisioners阶段，也可以省掉prlctl window view或促成headless而主要输出保留在packer那个窗口，这个阶段也是正常的输出主要场所而我们希望直接在livecd中一次完成所有事情，。2G内存是为了接下来编译gcc不致内存耗光，"headless": true保证了在无x11的服务器环境也能运行其实还有个disable_vnc可用。里面只有一条export是为了与接下来切换shell讲解部分中的其它export作对应。

> CorePure64-11.1-openssl-remaster.iso是在TinyCorePure64-11.1.iso中进桌面使用ezremaster集成openssh,parted,grub2-multi(这个不集成接下来的tce-load会失效因为没有分区)生成的(其中openssh集成到extract to initrd其它二个outside initrd on boot，而以前文章中，我们在packer中写到硬盘版tc中集成,这很麻烦比如inbuilt需求在packer中集成tcz你需要unsquashfs tczs和复制ld.conf.cache（值得一提的是boot_command中允许写sudo reboot不会退出packer这让我们可以在硬盘上准备一个非最终目标的tc不断reboot后构建，用于保存系统中间状态再构建），，ezremaster能自动处理依赖还让你进入extract目录定制的机会，我给ssh复制了一个sshd_config和mkdir了一个/var/lib/sshd目录写入了分区用的parted -s /dev/vda mktable msdos;parted -s /dev/vda mkpart primary ext3 2048s 100%;parted -s /dev/vda set 1 boot on;mkfs.ext3 /dev/vda1;rebuildfstab;mount /mnt/vda1;grub-install --boot-directory=/mnt/vda1/boot /dev/vda和一句tce-setup最后启用ssh用的/usr/local/init.d/openssh start到bootlocal.sh(注意这3个先后，这个文件里默认是sudo的)，还sudo定制了tc密码,给bootcode增加了tce=vda1 opt=vda1 home=vda1 restore=vda1到extract/..../isolinux.cfg还另外为vda1版本写了一份覆盖f2)，这样就充分压缩了boot-command

```
"builders":[{
      "type": "qemu",
  "iso_url": "http://www.shalol.com/d/mirrors/tinycorelinux/CorePure64-11.1-ezremasterd.iso",
  "iso_checksum_type": "sha256",
  "iso_checksum": "ec6173e4ee9d64276ce07f4bb362cd7d9dfb2964ce13e9a6e0695fcdeab89b11",

  "vm_name":"tclinuxforselfbootstrape",
  "net_device": "virtio-net",
```

```
        "disk_interface": "virtio",
        "disk_size": 20000,
    "format": "raw",
        "cpus": 2,
        "memory": 2048,

    "headless": true,
    "vnc_port_min": 5960,
    "vnc_port_max": 5960,
    "boot_wait": "10s",

    "boot_command":[
          "<enter><wait20>",
          "export<return>"
    ],

    "ssh_username": "tc",
    "ssh_password": "tc",

    "shutdown_command": "echo 'tc' | sudo poweroff",
        "output_directory": "/Users/minlearn/packer/iso"

  }],
```

matecloudos-tools.packer provisioners:在这里我们处理即时shell和shell变量嵌套，我们不将逻辑写在packer中，packer主文件只用来设置框架。主要的逻辑放在接下来的sh中进行，这样才能避免packer json不断转义限制，packer中每一段shell chunk都会生成一个目标机/tmp下的sh，这里的问题是切换shell，父shell和子shell的变量是隔离的。sh方式运行脚本，会重新开启一个子shell，脚本中export输出的SHLVL可以看到，无法继承父进程的普通变量，能继承父进程export的全局变量。source或者．方式运行脚本，会在当前shell下运行脚本，相当于把脚本内容加载到当前shell后执行，自然能使用前面定义的变量。compile_tc11_x86_64主要使用bash，因此必须保证所有的脚本shebang都是bash的而不是tc默认的ash。

注意此时还是tclinux的ash。

```
  "provisioners": [
    {
        "type": "shell",
        "pause_before":"1s",
        "inline":[
        "export",

        "sudo mkdir -p /mnt/vda1/opt /mnt/vda1/home",

        "sudo rm /opt/tcemirror && sudo touch /opt/tcemirror",
        "sudo sh -c 'echo http://www.shalol.com/d/mirrors/tinycorelinux/ > /opt/tcemirror'",
        "sudo sh -c 'sed -i s/wget[[:space:]]*-c[[:space:]]/\"wget -cq \"/g /usr/bin/tce-load'",

          "echo ADD BASH PKG",
        "tce-load -iw -s bash",
          "#sudo rm -rf /mnt/vda1/boot/tmp/bin/sh",
          "#sudo ln -sf /usr/local/bin/bash /mnt/vda1/boot/tmp/bin/sh",
          "#sudo sh -c 'sed -i s#bin/sh#usr/local/bin/bash#g /etc/passwd'",

          "echo PRE SAVE STATE",
          "sudo sh -c 'echo opt/tcemirror >> /mnt/vda1/opt/.filetool.lst'",
          "#sudo sh -c 'echo etc/passwd >> /mnt/vda1/opt/.filetool.lst'",
          "sudo /bin/tar -C / -T /mnt/vda1/opt/.filetool.lst -czf /mnt/vda1/mydata.tgz",

        "echo fix the system ar",
        "sudo rm -rf /usr/bin/ar",

          "echo prepare for uploading",
        "sudo mkdir -p /mnt/vda1/tmp",
        "sudo chown tc:staff /mnt/vda1/tmp/",
        "#sudo rm -rf /mnt/vda1/tmp/src"

        ]
    },
```

注意为了测试，src我暂时放在了本地跟脚本逻辑一起，未来download src的方式会放到主体脚本逻辑中一一wget。还注意到qemu虚拟网卡网络很不稳定，前面装tce都有可能出错，更别说这里的wget src.tar了，故暂时用本地上传。这是packer141 bug?qemu bug?换版本组合会好点？

```
    {
      "type": "file",
      "source": "./src",
      "destination": "/mnt/vda1/tmp"
    },
```

这里将shebang切成了bash，且开始用sudo，如果这里不用sudo，那么接下来要打大量sudo，且接下来脚本中echo >,sed 这样的语句sudo sh -c ''发挥不了作用，很诡异，没有继续研究。

```
    {
```

```
        "type": "shell",
        "pause_before":"1s",
        "execute_command": "sed -i s#bin/sh#usr/local/bin/bash#g {{ .Path }} && sudo {{ .Path }}",
        "scripts":
        [
        "./src/2.buildtc/buildtools"
        ]
        }

  ]
```

tc脚本

这里的又一个跟命令有效性相关的问题在于，将任意命令放在数组中，当展开的时候，其作用并不是像展开的字符串一次喂入的效果一样。有时候不能正常调用转义不掉，猜测是变量替换会把空格隔开的单位分行放置，比如custproc如果前面有环境变量会当成语句被执行。所以我尽量把所有的正常proc都拆开放置，并把命令字符（排除参数）放到非数组部分。尽量不做allinoneproc到数组中。这里着重要提到的是``，相当于$()，它是优先调用符（vs bash的变量替换，这是bash的命令替换），它如果放在数组中会被定义数组的时候就给优先执行了，所以必须用手动替换的方法设置CMDSEP绕过自动替换，在数组外想办法让它正常执行。

注意到shebang是手动加的

```
#!/usr/local/bin/bash

#tc11_x86_64 (on corepure64)

su - tc -c 'tce-load -iw -s compiletc rsync bc'
su - tc -c 'tce-load -iw -s compiletc perl5 ncursesw-dev bash mpc-dev udev-lib-dev texinfo coreutils glibc_apps rsync gettext
python3.6 automake autoconf-archive'
slient1=w
slient2=/dev/null
slient3=s
export MAKEFLAGS="-j 2"

#set +h
#umask 022
#用了/tmp而不是/tmp/tc，这样build src tools可以并列
TC=/mnt/sda1/tmp
LC_ALL=POSIX
TC_TGT=x86_64-tc-linux-gnu
PATH=/tools/bin:/usr/local/bin:/bin:/usr/bin
export TC LC_ALL TC_TGT PATH

export

#rm -rf $TC/tools $TC
mkdir -p $TC $TC/tools
chown tc:staff $TC/tools
#只要mount才能隐藏二级地址类301,302？
ln -sf $TC/tools /

#bash数组的每一条（成员）其实都是单条独立语句，可以放数组外分开定义。所以可以使用\换行放置，不会被归入到echo cmdstr结果中。

# 对于以下cmdstr
# 只要是语句，加不加DEFERSUBME都会被数组外bash eval当成语句发生变量替换/展开（即使是echo cmdstr都会发生变量替换，动态展开）,加个DEFERSUBME会更
清楚
# 如果是非环境变量的替换/展开，需要把$转义一下，如下面的for file ... $file要转成\$file，如果是环境变量如$TC则不需要，后者在不执行时就被静态展开了

declare -A PROCESSTABLE

export PROCESSTABLE=(
    ["binutils_p1_confhz"]="--prefix=/tools --with-sysroot=$TC --with-lib-path=/tools/lib --target=$TC_TGT --disable-nls --dis
able-werror"

    ["gcc_p1_precmds"]="rm -rf ../mpfr;cp -rf ../../mpfr-4.0.2 ../mpfr;rm -rf ../gmp;cp -rf ../../gmp-6.1.2 ../gmp;rm -rf ../m
pc;cp -rf ../../mpc-1.1.0 ../mpc; \
    for file in ../gcc/config/linux.h ../gcc/config/i386/linux.h ../gcc/config/i386/linux64.h; \
    do \
        cp -f \$file \$file.orig;rm -rf \$file; \
        sed -e 's#/lib\(64\)\?\(32\)\?/ld#/tools&#g' -e 's#/usr#/tools#g' \$file.orig > \$file; \
        echo '#undef STANDARD_STARTFILE_PREFIX_1' >> \$file;echo '#undef STANDARD_STARTFILE_PREFIX_2' >> \$file;echo '#define
STANDARD_STARTFILE_PREFIX_1 \"/tools/lib/\"' >> \$file;echo '#define STANDARD_STARTFILE_PREFIX_2 \"\"' >> \$file; \
        touch \$file.orig; \
    done; \
    sed -e '/m64=/s/lib64/lib/' -i.orig ../gcc/config/i386/t-linux64"
    ["gcc_p1_confhz"]="--target=$TC_TGT --prefix=/tools --with-glibc-version=2.11 --with-sysroot=$TC --with-newlib --without-h
eaders --with-local-prefix=/tools --with-native-system-header-dir=/tools/include --disable-nls --disable-shared --disable-mult
```

```
ilib --disable-decimal-float --disable-threads --disable-libatomic --disable-libgomp --disable-libquadmath --disable-libssp --
disable-libvtv --disable-libstdcxx --enable-languages=c,c++"

    ["linux_p1_custproc"]="make proper && make INSTALL_HDR_PATH=dest headers_install"
    ["linux_p1_postcmds"]="cp -r -f dest/include/* /tools/include"

    #["glibc_p1_precmds"]="#edit manual/libc.tcexinfo;#remove @documentencoding UTF-8"
    ["glibc_p1_confhz"]="--prefix=/tools --host=$TC_TGT --build=DEFERSUBME../scripts/config.guessDEFERSUBME --enable-kernel=4.
19.10 --with-headers=/tools/include"
    #["glibc_p1_postcmds"]="echo 'int main\(\){}' > dummy.c;$TC_TGT-gcc dummy.c;readelf -l a.out | grep ': /tools';rm dummy.c
a.out"

    ["gcc_p2_confhz"]="--host=$TC_TGT --prefix=/tools --disable-multilib --disable-nls --disable-libstdcxx-threads --disable-l
ibstdcxx-pch --with-gxx-include-dir=/tools/$TC_TGT/include/c++/9.2.0"

    ["binutils_p2_confqz"]="CC=$TC_TGT-gcc AR=$TC_TGT-ar RANLIB=$TC_TGT-ranlib"
    ["binutils_p2_confhz"]="--prefix=/tools --disable-nls --disable-werror --with-lib-path=/tools/lib --with-sysroot"

    #["gcc_p3_precmds"]="#rm -rf DEFERSUBMEdirname DEFERSUBME$TC_TGT-gcc -print-libgcc-file-nameDEFERSUBMEDEFERSUBME/include-f
ixed/limits.h;cat gcc/limitx.h gcc/glimits.h gcc/limity.h > DEFERSUBMEdirname DEFERSUBME$TC_TGT-gcc -print-libgcc-file-nameDEF
ERSUBMEDEFERSUBME/include-fixed/limits.h"
    ["gcc_p3_confqz"]="CC=$TC_TGT-gcc CXX=$TC_TGT-g++ AR=$TC_TGT-ar RANLIB=$TC_TGT-ranlib"
    ["gcc_p3_confhz"]="--prefix=/tools --with-local-prefix=/tools --with-native-system-header-dir=/tools/include --enable-lang
uages=c,c++ --disable-libstdcxx-pch --disable-multilib --disable-bootstrap --disable-libgomp"
    #["gcc_p3_postcmds"]="ln -sf gcc /tools/bin/cc;echo 'int main\(\){}' > dummy.c;$TC_TGT-gcc dummy.c;readelf -l a.out | grep
 ': /tools';rm dummy.c a.out"

    ["m4_p1_precmds"]="sed -i 's/IO_ftrylockfile/IO_EOF_SEEN/' lib/*.c;echo '#define _IO_IN_BACKUP 0x100' >> lib/stdio-impl.h"
    ["m4_p1_confhz"]="--prefix=/tools"

    ["ncurses_p1_precmds"]="sed -i s/mawk// configure"
    ["ncurses_p1_confhz"]="--prefix=/tools --with-shared --without-debug --without-ada --enable-widec --enable-overwrite"
    ["ncurses_p1_postcmds"]="ln -sf libncursesw.so /tools/lib/libncurses.so"

    ["bash_p1_confhz"]="LDFLAGS=-L/tools/lib"
    ["bash_p1_confhz"]="--prefix=/tools --without-bash-malloc"

    ["bison_p1_confhz"]="--prefix=/tools"
    ["bzip2_p1_installhz"]="PREFIX=/tools install"

    ["coreutils_p1_confqz"]="FORCE_UNSAFE_CONFIGURE=1"
    ["coreutils_p1_confhz"]="--prefix=/tools --enable-install-program=hostname"

    ["diffutils_p1_confhz"]="--prefix=/tools"
    ["file_p1_confhz"]="--prefix=/tools"
    ["findutils_p1_confhz"]="--prefix=/tools"
    ["gawk_p1_confhz"]="--prefix=/tools"

    ["gettext_p1_confhz"]="--disable-shared"
    #["gettext_p1_postcmds"]="#cp gettext-tools/src/msgfmt /tools/bin;#cp gettext-tools/src/msgmerge /tools/bin;#cp gettext-to
ols/src/xgettext /tools/bin"

    ["grep_p1_confhz"]="--prefix=/tools"
    ["gzip_p1_confhz"]="--prefix=/tools"

    #["make_p1_precmds"]="sed -i '211,217 d; 219,229 d; 232 d' glob/glob.c"
    ["make_p1_confhz"]="--prefix=/tools --without-guile"

    ["patch_p1_confhz"]="--prefix=/tools"

    ["perl_p1_custproc"]="sh Configure -des -Dprefix=/tools -Dlibs=-lm -Uloclibpth -Ulocincpth && make"
    #["perl_p1_postcmds"]="#cp perl cpan/podlators/scripts/pod2man /tools/bin;#mkdir -p /tools/lib/perl5/5.30.0;#cp -R lib/* /
tools/lib/perl5/5.30.0"

    ["Python_p1_precmds"]="sed -i '/def add_multiarch_paths/a \\        return' setup.py"
    ["Python_p1_confhz"]="--prefix=/tools --without-ensurepip"

    ["sed_p1_confhz"]="--prefix=/tools"
    ["tar_p1_confhz"]="--prefix=/tools"
    ["texinfo_p1_confhz"]="--prefix=/tools"
    ["xz_p1_confhz"]="--prefix=/tools"
)

#以下代码来自cloverboot，作了修改和增强，比如接受tarballs数组
#直接用

export DOWNLOADPREFIX=${DOWNLOADPREFIX:-http://www.shalol.com/d/mirrors/buildmatecloudos}
export DIR_GCC=${DIR_GCC:-/mnt/sda1/tmp/build}
[ ! -d ${DIR_GCC} ]        && mkdir ${DIR_GCC}
```

```
### Download and Extract ###

# Function to extract source tarballs
DownloadandExtractTarballs ()
{
    exec 3>&1 1>&2 # Save stdout and redirect stdout to stderr

    tarballs=(`echo $1 | tr ',' ' '`)

    #local maintarball=${tarballs[0]}
    #local package=${maintarball}
    local package_top_level_dir=""

    for i in "${!tarballs[@]}"; do

        tarball="${DIR_DOWNLOADS}/${tarballs[i]}"

        if [[ ! -f ${tarball} ]]; then echo "Status: ${tarballs[i]} not found.";wget --no-check-certificate -qO- ${DOWNLOADPRE
FIX}/${tarballs[i]} > ${tarball} || exit 1; fi


        local filetype=$(file -L --brief "${tarball}" | tr '[A-Z]' '[a-z]')
        local tar_filter_option=""

        case ${filetype} in # convert to lowercase
            gzip\ *)  tar_filter_option='--gzip' ;;
            bzip2\ *) tar_filter_option='--bzip2';;
            lzip\ *)  tar_filter_option='--lzip' ;;
            lzop\ *)  tar_filter_option='--lzop' ;;
            lzma\ *)  tar_filter_option='--lzma' ;;
            xz\ *)    tar_filter_option='--xz'   ;;
            *tar\ archive*) tar_filter_option='';;
            *) echo "Unrecognized file format of '${tarball}'"
           exit 1
           ;;
        esac

        # Get the root directory from the tarball
        local first_line=$(dd if="${tarball}" bs=1024 count=256 2>/dev/null | \
            tar -t $tar_filter_option -f - 2>/dev/null | head -1)
            local top_level_dir=${first_line#./}  # remove leading ./
        top_level_dir=${top_level_dir%%/*}    # keep only the top level directory

        # save the main tarball topleveldir name
        [[ $i -eq 0 ]] && package_top_level_dir=${top_level_dir}

        [ -z "$top_level_dir" ] && echo "Error can't extract top level dir from $tarball" && exit 1

        if [[ ! -d "${DIR_GCC}/$top_level_dir" ]]; then
            echo "- ${tarballs[i]} extracting..."
            rm -rf "${DIR_GCC}/$top_level_dir" # Remove old directory if exists
            rm -rf "$DIR_GCC/tmp"   # Remove old partial extraction
            mkdir -p "$DIR_GCC/tmp" # Create temporary directory
            tar -C "$DIR_GCC/tmp" -x "$tar_filter_option" -f "${tarball}"
            mv "$DIR_GCC/tmp/$top_level_dir" "$DIR_GCC/$top_level_dir"
            rm -rf "$DIR_GCC/tmp"
            echo "- ${tarballs[i]} extracted"
        fi

    done

    # Restore stdout for the result and close file descriptor 3
    exec 1>&3-
    echo "${DIR_GCC}/$package_top_level_dir" # Return the full path where the main tarball has been extracted
}

### Compiling package ###

export DIR_LOGS=${DIR_LOGS:-/mnt/sda1/tmp/logs}
[ ! -d ${DIR_LOGS} ]       && mkdir ${DIR_LOGS}

CompilePackage ()
{

    local package="$1"
    # 百分号是去掉右边，二个百分号最大模式直到遇到最后一个-
    packagebase=${package%%-*}
    # 井号是去掉左边，二个井号最大模式直到遇到第一个:
    packagepass=${package##*:}
    packageentry=$packagebase"_"$packagepass
```

```
    # 百分号是去掉右边，一个百分号最小模式直到遇到最后一个:
    local packagefiles=${package%:*}
    local packagedir=$(DownloadandExtractTarballs "${packagefiles}") || exit 1
    local packagecheckfile=$packageentry".ok"

    packageprecmds=${PROCESSTABLE[$packageentry"_precmds"]} ; [[ $packageprecmds =~ "DEFERSUBME" ]] && packageprecmds=${packag
eprecmds//DEFERSUBME/'`'}
    packageconfqz=${PROCESSTABLE[$packageentry"_confqz"]} ; [[ $packageconfqz =~ "DEFERSUBME" ]] && packageconfqz=${packagecon
fqz//DEFERSUBME/'`'}
    packageconfhz=${PROCESSTABLE[$packageentry"_confhz"]} ; [[ $packageconfhz =~ "DEFERSUBME" ]] && packageconfhz=${packagecon
fhz//DEFERSUBME/'`'}
    packagemakeqz=${PROCESSTABLE[$packageentry"_makeqz"]} ; [[ $packagemakeqz =~ "DEFERSUBME" ]] && packagemakeqz=${packagemak
eqz//DEFERSUBME/'`'}
    packagemakehz=${PROCESSTABLE[$packageentry"_makehz"]} ; [[ $packagemakehz =~ "DEFERSUBME" ]] && packagemakehz=${packagemak
ehz//DEFERSUBME/'`'}
    packageinstallqz=${PROCESSTABLE[$packageentry"_installqz"]} ; [[ $packageinstallqz =~ "DEFERSUBME" ]] && packageinstallqz=
${packageinstallqz//DEFERSUBME/'`'}
    packageinstallhz=${PROCESSTABLE[$packageentry"_installhz"]} ; [[ $packageinstallhz =~ "DEFERSUBME" ]] && packageinstallhz=
${packageinstallhz//DEFERSUBME/'`'}
    packagepostcmds=${PROCESSTABLE[$packageentry"_postcmds"]} ; [[ $packagepostcmds =~ "DEFERSUBME" ]] && packagepostcmds=${pa
ckagepostcmds//DEFERSUBME/'`'}

    packagecustproc=${PROCESSTABLE[$packageentry"_custproc"]} ; [[ $packagecustproc =~ "DEFERSUBME" ]] && packagecustproc=${pa
ckagecustproc//DEFERSUBME/'`'}

    if [ ! -f $DIR_LOGS/"$packagecheckfile" ]; then

        cd ${packagedir}
        #[[ "$packagepass" -gt 1 ]] && rm -rf $packagedir

        cd $packagedir;mkdir -p "build"$packagepass;cd "build"$packagepass

        startBuildEpoch=$(date -u "+%s")
        echo "precmds:" "$packageprecmds"
        eval "$packageprecmds"
        if [ ! -n $packagecustproc ]; then eval "$packagecustproc" > $slient2;[[ $? -eq 0 ]] && touch $DIR_LOGS/$packagecheckf
ile; else
            logfile="$DIR_LOGS/${packageentry}.configure.log.txt"
            echo "-  ${packageentry} configure..."
            #echo "configure:" "$packageconfqz ../configure $packageconfhz"
            eval "$packageconfqz ../configure $packageconfhz" > "$logfile" 2>&1
            [[ $? -ne 0 ]] && echo "Error configuring ${packageentry} ! Check the log $logfile" && exit 1

            logfile="$DIR_LOGS/${packageentry}.make.log.txt"
            echo "-  ${packageentry} make..."
            eval "CC=\"gcc -$slient1\" $packagemakeqz make -$slient3 $packagemakehz" 1> "$logfile" 2>&1
            [[ $? -ne 0 ]] && echo "Error making ${packageentry} ! Check the log $logfile" && exit 1

            logfile="$DIR_LOGS/${packageentry}.install.log.txt"
            echo "-  ${packageentry} install..."
            eval "$packageinstallqz make -$slient3 install" 1> "$logfile" 2>&1
            [[ $? -ne 0 ]] && echo "Error installing ${packageentry} ! Check the log $logfile" && exit 1 || touch $DIR_LOGS/$p
ackagecheckfile
        fi
        echo "postcmds:" "$packagepostcmds"
        eval "$packagepostcmds"
        stopBuildEpoch=$(date -u "+%s");buildTime=$((stopBuildEpoch - startBuildEpoch));if [ $buildTime -gt 59 ]; then timeToB
uild=$(printf "%dm%ds" $((buildTime/60%60)) $((buildTime%60))); else timeToBuild=$(printf "%ds" $((buildTime))); fi;printf --
"-  %s %s %s\n" "$packageentry" "Total Time Spent: " "$timeToBuild"

    else
        echo $package is already compiled and installed!
    fi
}

echo ==============compile tools...============================


for packagedef in binutils-2.33.1.tar.xz:p1 gcc-9.2.0.tar.xz,mpfr-4.0.2.tar.xz,gmp-6.1.2.tar.xz,mpc-1.1.0.tar.gz:p1 linux-5.4.
3.tar.xz:p1 glibc-2.30.tar.xz:p1 binutils-2.33.1.tar.xz:p2 gcc-9.2.0.tar.xz:p2 gcc-9.2.0.tar.xz:p3 m4-1.4.18.tar.gz:p1 ncurses
-6.1.tar.gz:p1 bash-5.0.tar.gz:p1 bison-3.4.2.tar.xz:p1 bzip2-1.0.8.tar.gz:p1 coreutils-8.31.tar.xz:p1 diffutils-3.7.tar.xz:p1
 file-5.37.tar.gz:p1 findutils-4.7.0.tar.xz:p1 gawk-5.0.1.tar.xz:p1 gettext-0.20.1.tar.gz:p1 grep-3.3.tar.xz:p1 gzip-1.10.tar.
xz:p1 make-4.2.1.tar.gz:p1 patch-2.7.6.tar.gz:p1 perl-5.30.0.tar.gz:p1 Python-3.8.0.tar.gz:p1 sed-4.7.tar.xz:p1 tar-1.32.tar.x
z:p1 texinfo-6.7.tar.gz:p1 xz-5.2.4.tar.gz:p1

do

  CompilePackage $packagedef || exit 1
```

```
done
```

此脚本还需要调试，这段脚本仅是compile_tc11_x86_64的前三分之一的部分。未来会加入新的processcmd数组和处理逻辑的组合。 调试方法，我们调用处设置了echo cmdstr，如果发现被截断或无法原形显示，就在cmdstr数组条目中进行处理修正，不断整合正确结果到脚本

发现一个一个可以解决云笔记的痛点问题所在。即文本的版本记录应该配合github desktop这种能动态显历史更改的东西来用。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一个fully retryable的rootbuild packer脚本,从0打造matecloudos(2):以lfs9观点看compiletc tools

本文关键字，**bash 命令替换嵌套,bash echo -e输出换行,lfs Constructing a Temporary System,How to uninstall gcc installed from source**

在《一种虚拟boot作通用bootloader及一种通用qemu os的设想》和《一个fully retryable的rootbuild packer脚本,从0打造matecloudos(1):》中，我们都讲到tc11上编译/交叉编译gcc920的实践，其中前者是我基于clover build gcc8修改而来的，这套脚本有很多增强和亮点，如downloadandextracttarballs()脚本与大文件分开，源码包在外部下载，，“all slient compiling”和“判断连续命令最后一条是否出错”的方法。。，后者则是我针对tc官方的compiletc11，结合这些亮点和增强，打造的最终“力求fully retryable”的packer脚本(我们选择了tclinux这类有"动态mountable,包管理重启后没有污染","包被放在硬盘上也不占内存"，“系统与数据分离”优点的云系统来作为m同样是mate"cloud"os的基础，用packer脚本来一步步构建它)。而其实，我们之前很多文章如《把dbcolinux导出为虚拟机和docker格式》《将虚拟机集成在bios中：avatt》，也是这类实践的例子。

> 说到这个cross compile in packer，为什么要花大力气去编译toolchain且做成packer? 因为我们要力求把这个过程做成可观察参考的动态例子（packer中一切都是从0开始provisione起来的），搜索引擎搜索到的那些，都是基于传统的方法（如https://wiki.osdev.org/GCC_Cross-Compilerc能查到的那些（《一种虚拟boot...qemu os的设想》和《dbcolinux》《avatt》也属于这种方法），而lfs9(http://www.linuxfromscratch.org/lfs/downloads/9.0/LFS-BOOK-9.0.pdf)是业界的标准实践路子（《一个fully retryable的rootbuild packer脚本》和compiletc11就是属于lfs9的路子），其实编译gcc也不是什么特别复杂的工作，只是这里面的问题是从来找不到一个必然会成功的例子中间有很多坑你只能去实践：
>
> 1，toolchain是属于基础工具，基于“先用起来主义”随着历史发展起来的gcc很复杂，有很多配置和参数(整个gcc也没有retryable的make uninstall的支持，只有部分有)，版本之间又有差别,binutils这种东西又是独立gcc的，虽然有lfs这种标准教程和很多crossbuild工具/buildroot工具套件，但是除非你像lfs一样一步一步照做，否则离构建成功，该踩的坑你还是要踩（实际上照lfs做你照样会踩坑）。2，如果你选择手动，除去上面提到的过程中出现的坑，结果也会出现一些，如果你不注意，有时会碰到虽然toolchain构建成功了，但是异常运行的情况（实际效果不是我们要的）。而lfs和buildroot这种套件又没有针对构建中每一个pass，pass中的subpass的中间调试信息。保证每一步结果都是正确的到最后必然正确的设施。compiletc11的脚本中有一些。 3，.... 4，这一切都导致了能编译一套toolchain也其实是工作量不小的事情，也是我们这个《一个fully retryable的rootbuild packer脚本》系列要解决的问题。

不废话了。上脚本：

重新改的ezremasterd iso和packer provisioner部分

这个脚本主要是清晰化了做进iso和写进packer文件的界限。被注释的都是做进了ezremasterd iso的，当然，具体方法跟被注释的语句不是一一对应的，但是效果相当。你可以将一切都保留在packer脚本中，去掉注释稍微修改即可，这样可以免ezremaster一次iso，我这样做只是为“最终服务于显式构建tc11主要逻辑，不让packer变得过乱”。

```
"provisioners": [
    {
        "type": "shell",
        "pause_before":"1s",
        "inline":[
        "export PATH=$PATH:/usr/local/sbin:/usr/local/bin",

        "echo 'PREPARE HD'",
        "sudo parted -s /dev/sda mktable msdos",
        "sudo parted -s /dev/sda mkpart primary ext3 2048s 100%",
        "sudo parted -s /dev/sda set 1 boot on",
        "sudo mkfs.ext3 /dev/sda1",
        "sudo rebuildfstab",
        "sudo mount /dev/sda1",

          "echo 'make iso perisit for after reboot useablity(commented statements were remasterd into iso already)'",
        "sudo grub-install --boot-directory=/mnt/sda1/boot /dev/sda",
          "sudo cp /mnt/sr0/boot/corepure64.gz /mnt/sda1/boot/corepure64.gz",
          "sudo cp /mnt/sr0/boot/vmlinuz64 /mnt/sda1/boot/vmlinuz64",
          "sudo sh -c 'echo set timeout=3 > /mnt/sda1/boot/grub/grub.cfg'",
          "sudo sh -c 'echo menuentry \"tclinuxforselfbootstrape for after reboot useablity\" { >> /mnt/sda1/boot/grub/grub.cf
g'",
          "#append bootcodes: loglevel=3 tce=sda1 opt=sda1 home=sda1 restore=sda1 cde",
          "sudo sh -c 'echo linux /boot/vmlinuz64 loglevel=3 tce=sda1 opt=sda1 home=sda1 restore=sda1 cde >> /mnt/sda1/boot/gr
ub/grub.cfg'",
          "sudo sh -c 'echo initrd /boot/corepure64.gz >> /mnt/sda1/boot/grub/grub.cfg'",
          "sudo sh -c 'echo } >> /mnt/sda1/boot/grub/grub.cfg'",

          "echo 'ADD PKG(commented statements were remasterd into iso already)'",
        "sudo mkdir -p /mnt/sda1/opt /mnt/sda1/home /mnt/sda1/tce",
        "sudo tce-setup",
```

```
        "sudo rm /opt/tcemirror && sudo touch /opt/tcemirror",
        "sudo sh -c 'echo http://10.211.55.2:8000/tinycorelinux/ > /opt/tcemirror'",
        "sudo sh -c 'sed -i s/wget[[:space:]]*-c[[:space:]]/\"wget -cq \"/g /usr/bin/tce-load'",
        "#tce-load -iw -s openssh parted grub-multi",
        "tce-load -iw -s bash",

        "echo 'add neccessy and fix the small system issues(commented statements were remasterd into iso already)'",
        "sudo rm -rf /usr/bin/ar",
          "#sudo passwd tc",
          "#tc",
          "#tc",
        "#sudo cp /usr/local/etc/ssh/sshd_config.ori /usr/local/etc/ssh/sshd_config",
        "#sudo mkdir -p /var/lib/sshd/",
        "#sudo mkdir -p /usr/local/etc/ssl/certs/",
          "#sudo /usr/local/etc/init.d/openssh start",

          "echo 'PRE SAVE STATE(commented statements were remasterd into iso already)'",
          "sudo sh -c 'echo opt/tcemirror >> /mnt/sda1/opt/.filetool.lst'",
          "#sudo sh -c 'echo usr/local/etc/ssh > /mnt/sda1/opt/.filetool.lst'",
          "#sudo sh -c 'echo etc/passwd >> /mnt/sda1/opt/.filetool.lst'",
          "#sudo sh -c 'echo etc/shadow >> /mnt/sda1/opt/.filetool.lst'",
          "sudo /bin/tar -C / -T /mnt/sda1/opt/.filetool.lst -czf /mnt/sda1/mydata.tgz",
          "#sudo mv ~/tce /mnt/sda1/",
          "#sudo cp -R /opt /mnt/sda1",

          "echo 'prepare for uploading'",
        "sudo mkdir -p /mnt/sda1/tmp",
        "sudo chown tc:staff /mnt/sda1/tmp/"
        ]
    }
```

修正的CompilePackage

注意这个脚本被作为bash module，被下一个脚本以. /mnt/sda1/tmp/src/common/common形式调用，注意这个.后面有一个空格，它表示source后面的脚本。然后里面的全局变量和全局函数就可以被父shell调用了，也就是下一个脚本。

```
#!/usr/local/bin/bash

### Download and Extract ###

......


### Compiling package ###

export DIR_LOGS=${DIR_LOGS:-/mnt/sda1/tmp/logs}
[ ! -d ${DIR_LOGS} ]       && mkdir ${DIR_LOGS}

CompilePackage ()
{

    local package="$1"
    packagebase=${package%%-*}
    packagepass=${package##*:}
    packageentry=$packagebase"_"$packagepass

    local packagefiles=${package%:*}
    local packagedir=$(DownloadandExtractTarballs "${packagefiles}") || exit 1
    local packagecheckfile=$packageentry".ok"

    packageprecmds=${PROCESSTABLE[$packageentry"_precmds"]} ; [[ $packageprecmds =~ "DEFERSUBME" ]] && packageprecmds=${packag
eprecmds//DEFERSUBME/'`'}
    packageconfqz=${PROCESSTABLE[$packageentry"_confqz"]} ; [[ $packageconfqz =~ "DEFERSUBME" ]] && packageconfqz=${packagecon
fqz//DEFERSUBME/'`'}
    packageconfcmd=${PROCESSTABLE[$packageentry"_confcmd"]} ; [[ $packageconfcmd =~ "DEFERSUBME" ]] && packageconfcmd=${packag
econfcmd//DEFERSUBME/'`'}
    packageconfhz=${PROCESSTABLE[$packageentry"_confhz"]} ; [[ $packageconfhz =~ "DEFERSUBME" ]] && packageconfhz=${packagecon
fhz//DEFERSUBME/'`'}
    packagemakeqz=${PROCESSTABLE[$packageentry"_makeqz"]} ; [[ $packagemakeqz =~ "DEFERSUBME" ]] && packagemakeqz=${packagemak
eqz//DEFERSUBME/'`'}
    packagemakehz=${PROCESSTABLE[$packageentry"_makehz"]} ; [[ $packagemakehz =~ "DEFERSUBME" ]] && packagemakehz=${packagemak
ehz//DEFERSUBME/'`'}
    packageinstallqz=${PROCESSTABLE[$packageentry"_installqz"]} ; [[ $packageinstallqz =~ "DEFERSUBME" ]] && packageinstallqz=
${packageinstallqz//DEFERSUBME/'`'}
    packageinstallhz=${PROCESSTABLE[$packageentry"_installhz"]} ; [[ $packageinstallhz =~ "DEFERSUBME" ]] && packageinstallhz=
${packageinstallhz//DEFERSUBME/'`'}
    packagepostcmds=${PROCESSTABLE[$packageentry"_postcmds"]} ; [[ $packagepostcmds =~ "DEFERSUBME" ]] && packagepostcmds=${pa
```

```
ckagepostcmds//DEFERSUBME/'`'}

    packagecustproc=${PROCESSTABLE[$packageentry"_custproc"]} ; [[ $packagecustproc =~ "DEFERSUBME" ]] && packagecustproc=${pa
ckagecustproc//DEFERSUBME/'`'}

    if [ ! -f $DIR_LOGS/"$packagecheckfile" ]; then

        cd ${packagedir}
        #[[ "$packagepass" -gt 1 ]] && rm -rf $packagedir

        cd $packagedir;mkdir -p "build"$packagepass;cd "build"$packagepass

        startBuildEpoch=$(date -u "+%s")
        if [ -n "$packageprecmds" ]; then
            logfile="$DIR_LOGS/${packageentry}.precmds.log.txt"
            echo "-  ${packageentry} precmds...."
            eval "$packageprecmds" > "$logfile" 2>&1
            [[ $? -ne 0 ]] && echo -e "Error processing packageprecmds: \n$packageprecmds" && exit 1
        fi

        if [ ! -n "$packagecustproc" ]; then
            logfile="$DIR_LOGS/${packageentry}.configure.log.txt"
            echo "-  ${packageentry} configure..."
            [[ ! -n "$packageconfcmd" ]] && eval "$packageconfqz ../configure $packageconfhz" > "$logfile" 2>&1 || eval "$pack
ageconfqz $packageconfcmd $packageconfhz" > "$logfile" 2>&1
            [[ $? -ne 0 ]] && echo "Error configuring ${packageentry} ! Check the log $logfile" && exit 1

            logfile="$DIR_LOGS/${packageentry}.make.log.txt"
            echo "-  ${packageentry} make..."
            eval "CC=\"gcc -$slient1\" $packagemakeqz make -$slient3 $packagemakehz" 1> "$logfile" 2>&1
            [[ $? -ne 0 ]] && echo "Error making ${packageentry} ! Check the log $logfile" && exit 1

            if [ ! -n "$packagepostcmds" ]; then
                logfile="$DIR_LOGS/${packageentry}.install.log.txt"
                echo "-  ${packageentry} install..."
                eval "$packageinstallqz make -$slient3 install $packageinstallhz" 1> "$logfile" 2>&1
                [[ $? -ne 0 ]] && echo "Error installing ${packageentry} ! Check the log $logfile" && exit 1 || touch $DIR_LOG
S/$packagecheckfile
            else
                logfile="$DIR_LOGS/${packageentry}.install.log.txt"
                echo "-  ${packageentry} install..."
                eval "$packageinstallqz make -$slient3 install $packageinstallhz" 1> "$logfile" 2>&1
                [[ $? -ne 0 ]] && echo "Error installing ${packageentry} ! Check the log $logfile" && exit 1

                logfile="$DIR_LOGS/${packageentry}.postcmds.log.txt"
                echo "-  ${packageentry} postcmds...."
                eval "$packagepostcmds" > "$logfile" 2>&1
                [[ $? -ne 0 ]] && echo -e "Error processing packagepostcmds: \n$packagepostcmds" && exit 1 || touch $DIR_LOGS/
$packagecheckfile
            fi
        else
            if [ ! -n "$packagepostcmds" ]; then
                logfile="$DIR_LOGS/${packageentry}.custproc.log.txt"
                echo "-  ${packageentry} custproc..."
                eval "$packagecustproc" > "$logfile" 2>&1
                [[ $? -ne 0 ]] && echo -e "Error processing packagecustproc: \n$packagecustproc" && exit 1 || touch $DIR_LOGS/
$packagecheckfile
            else
                logfile="$DIR_LOGS/${packageentry}.custproc.log.txt"
                echo "-  ${packageentry} custproc..."
                eval "$packagecustproc" > "$logfile" 2>&1
                [[ $? -ne 0 ]] && echo -e "Error processing packagecustproc: \n$packagecustproc" && exit 1

                logfile="$DIR_LOGS/${packageentry}.postcmds.log.txt"
                echo "-  ${packageentry} postcmds...."
                eval "$packagepostcmds" > "$logfile" 2>&1
                [[ $? -ne 0 ]] && echo -e "Error processing packagepostcmds: \n$packagepostcmds" && exit 1 || touch $DIR_LOGS/
$packagecheckfile
            fi
        fi
        stopBuildEpoch=$(date -u "+%s");buildTime=$((stopBuildEpoch - startBuildEpoch));if [ $buildTime -gt 59 ]; then timeToB
uild=$(printf "%dm%ds" $((buildTime/60%60)) $((buildTime%60))); else timeToBuild=$(printf "%ds" $((buildTime))); fi;printf --
"-  %s %s %s\n" "$packageentry" "Total Time Spent: " "$timeToBuild"

    else
        echo $package is already built and installed!
    fi
}
```

对应lfs9的Constructing a Temporary System部分的脚本

基本上对于lfs9可以这样理解：

constructing a temp system有二个toolchain main pass(p1-1这样的表示：main pass1中的subpass1)，一个是binutils-2.33.1.tar.xz:p1 gcc-9.2.0.tar.xz:p1-1 linux-5.4.3.tar.xz:p1 glibc-2.30.tar.xz:p1 gcc-9.2.0.tar.xz:p1-2，另一个是binutils-2.33.1.tar.xz:p2 gcc-9.2.0.tar.xz:p2

对于第一个pass,是构建出了x86_64-tc-linux-gnu binutils和x86_64-tc-linux-gcc，属于仅换了名字的“cross compile”，由vendor "pc"换成了tc，其它主要的部件都没变，可见本质还是native compile，第二个是用第一个重新构建出我们机器上tce-load -iw compiletc安装的x86_64-pc-linux-gnu binutils和gcc组成的toolchain，本质也是native compile，这样我们的/tools/下会有x86_64-tc-linux-gnu和x86_64-pc-linux-gnu二套toolchain。包括tce-load -iw compiletc安装的会有三套toolchain。

这样有什么用呢，这是lfs9为了构建出一个足够clean的toolchain，对于每一个pass和每一个subpass都是严格按职责细分的结果，这里仅提示重点部分：

在第一个main pass中，前一个gcc-9.2.0.tar.xz:p1-1使得它编译出来的程序仅调用/tools/lib64/下面的glibc，glibc-2.30.tar.xz:p1中实际构建出了这些libs，并写了一个测试glibc_p1_postcmds。输出结果如果是调用了/tools/lib64下面的lib，就表示通过。gcc-9.2.0.tar.xz:p1-2则是仅构建c++stdlib。注意喂给这些pass,subpass中的build,host,target tripe参数的组合。

相比mainpass1自然的过渡风格，在第二个mainpass中，这里有一个相当大的坑：因为pass2，实际上是用x86_64-tc-linux-gnu-gcc x86_64-tc-linux-gnu-ar x86_64-tc-linux-gnu-ranlib来native编译x86_64-pc-linux-gnu toolchain（由于/tools/bin一开始就被设置为靠PATH前面，gcc脚本会自然识别到x86_64-tc-linux-gnu-*），但是却不直接用--host=x86_64-tc-linux-gnu，而是用了build=host=target=x86_64-pc-linux-gnu的强制组合（这个不是compiletc11脚本中的写法，但是与config guess和实际要求的结果是一致的，只是我为了在这里脚本不会产生可能的异常而强制的，这种可能的异常就是我上面提到的“除去上面提到的过程中出现的坑，结果也会出现一些”，马上就提到），在mainpass2构建成功后，前面的main pass1作为它的接力被一定程度废弃了，现在转为main pass2来发挥主要作用了，因为main pass2构建过后的toolchain负责接下来所有从m4-1.4.18.tar.gz:p1到xz-5.2.4.tar.gz:p1的大量utils的x86_64-pc-linux-gnu native编译，还有，gcc-9.2.0.tar.xz:p2不分gcc-9.2.0.tar.xz:p2-1，gcc-9.2.0.tar.xz:p2-2是因为c++stdlib不用再拆分了。跟mainpass1一样同样要注意喂给这些pass,subpass中的build,host,target tripe参数的组合。

> 在tc11 iso+gcc920 src中，这种强制可能会产生《一种虚拟boot...qemu os的设想》文出现的cant run compiled c programs错误(见那文详解)(实际上这条命令的configure在进入tc11手动是可以的，只是脚本中不行，所以要从脚本中找原因)，《一种虚拟boot...qemu os的设想》文权宜是使用host=x86_64代替host=x86_64-pc-linux-gnu，但是这样虽然可以让脚本继续，但是结果最终是异常的，所以是不对的。
>
> 要在脚本中保证最终结果，就需要强制build=host=target=x86_64-pc-linux-gnu，且使得编译能继续，解决方法是在["binutils_p2_postcmds"]中设置一条ln -sf /tools/lib /tools/lib64"，然后 ["gcc_p2_postcmds"]="ln -sf /tools/bin/x86_64-pc-linux-gnu-gcc /tools/bin/cc;再修正一下)。这样下来，脚本能跑了，最终的结果也是正确的：ldd测试这些out binary会发现正确调用了（如果不修正cc，会自动回退到tce-load -iw compiletc中的/usr/local/bin/cc，导致你会发现用gcc-p2编译的dummy.c运行时会发生ld cant find crt0在对应log中输出为空，这是异常产生的上级原因，产生异常的根源原因就在binutils p2中那条ln修正，tc11官方iso的bintuils没有x86...ar，只有ar，这也是一个可能产生异常的原因）

```
#!/usr/local/bin/bash

#tc11_x86_64 (on corepure64)

su - tc -c 'tce-load -iw -s compiletc rsync bc'
su - tc -c 'tce-load -iw -s compiletc perl5 ncursesw-dev bash mpc-dev udev-lib-dev texinfo coreutils glibc_apps rsync gettext
python3.6 automake autoconf-archive'
slient1=w
slient2=/dev/null
slient3=s
export MAKEFLAGS="-j 4"

declare -A PROCESSTABLE=(
    ["binutils_p1_confhz"]="--target=x86_64-tc-linux-gnu --prefix=/tools --with-sysroot=/mnt/sda1/tmp --with-lib-path=/tools/l
ib --disable-nls --disable-werror"

    ["gcc_p1-1_precmds"]="rm -rf ../mpfr;cp -rf ../../mpfr-4.0.2 ../mpfr;rm -rf ../gmp;cp -rf ../../gmp-6.1.2 ../gmp;rm -rf ..
/mpc;cp -rf ../../mpc-1.1.0 ../mpc; \
    for file in ../gcc/config/linux.h ../gcc/config/i386/linux.h ../gcc/config/i386/linux64.h; \
    do \
        cp -f \$file \$file.orig;rm -rf \$file; \
        sed -e 's#/lib\(64\)\?\(32\)\?/ld#/tools&#g' -e 's#/usr#/tools#g' \$file.orig > \$file; \
        echo '#undef STANDARD_STARTFILE_PREFIX_1' >> \$file;echo '#undef STANDARD_STARTFILE_PREFIX_2' >> \$file;echo '#define
STANDARD_STARTFILE_PREFIX_1 \"/tools/lib/\"' >> \$file;echo '#define STANDARD_STARTFILE_PREFIX_2 \"\"' >> \$file; \
        touch \$file.orig; \
    done; \
    sed -e '/m64=/s/lib64/lib/' -i.orig ../gcc/config/i386/t-linux64"
    ["gcc_p1-1_confhz"]="--target=x86_64-tc-linux-gnu --prefix=/tools --with-glibc-version=2.11 --with-sysroot=/mnt/sda1/tmp -
-with-newlib --without-headers --with-local-prefix=/tools --with-native-system-header-dir=/tools/include --disable-nls --disab
le-shared --disable-multilib --disable-decimal-float --disable-threads --disable-libatomic --disable-libgomp --disable-libquad
math --disable-libssp --disable-libvtv --disable-libstdcxx --enable-languages=c,c++"

    ["linux_p1_custproc"]="cp -f /mnt/sda1/tmp/src/1.buildbase/kernelandtoolchain/config-5.4.3-tinycore64 ../.config;cd ../ &&
```

```
 make mrproper && make INSTALL_HDR_PATH=dest headers_install"
    ["linux_p1_postcmds"]="cp -rf dest/include/* /tools/include"

    #["glibc_p1_precmds"]="#edit manual/libc.tcexinfo;#remove @documentencoding UTF-8"
    ["glibc_p1_confhz"]="--host=x86_64-tc-linux-gnu --prefix=/tools --build=DEFERSUBME../scripts/config.guessDEFERSUBME --enab
le-kernel=4.19.10 --with-headers=/tools/include"
    ["glibc_p1_postcmds"]="echo 'int main(){}' > dummy.c; \
    x86_64-tc-linux-gnu-gcc dummy.c; \
    readelf -l a.out | grep ': /tools'; \
    rm dummy.c a.out; \
    ln -sf /tools/lib /tools/lib64"

    ["gcc_p1-2_confcmd"]="../libstdc++-v3/configure"
    ["gcc_p1-2_confhz"]="--host=x86_64-tc-linux-gnu --prefix=/tools --disable-multilib --disable-nls --disable-libstdcxx-threa
ds --disable-libstdcxx-pch --with-gxx-include-dir=/tools/x86_64-tc-linux-gnu/include/c++/9.2.0"

    ["binutils_p2_confqz"]="CC=x86_64-tc-linux-gnu-gcc AR=x86_64-tc-linux-gnu-ar RANLIB=x86_64-tc-linux-gnu-ranlib"
    ["binutils_p2_confhz"]="--build=x86_64-pc-linux-gnu --host=x86_64-pc-linux-gnu --target=x86_64-pc-linux-gnu --prefix=/tool
s --disable-nls --disable-werror --with-lib-path=/tools/lib --with-sysroot"
    ["binutils_p2_postcmds"]="make -C ld clean; \
    make -C ld LIB_PATH=/usr/lib:/lib; \
    cp ld/ld-new /tools/bin"

    ["gcc_p2_precmds"]="cat ../gcc/limitx.h ../gcc/glimits.h ../gcc/limity.h > /tools/lib/gcc/x86_64-tc-linux-gnu/9.2.0/includ
e-fixed/limits.h"
    ["gcc_p2_confqz"]="CC=x86_64-tc-linux-gnu-gcc CXX=x86_64-tc-linux-gnu-g++ AR=x86_64-tc-linux-gnu-ar RANLIB=x86_64-tc-linux
-gnu-ranlib"
    ["gcc_p2_confhz"]="--build=x86_64-pc-linux-gnu --host=x86_64-pc-linux-gnu --target=x86_64-pc-linux-gnu --prefix=/tools --w
ith-local-prefix=/tools --with-native-system-header-dir=/tools/include --enable-languages=c,c++ --disable-libstdcxx-pch --disa
ble-multilib --disable-bootstrap --disable-libgomp"
    ["gcc_p2_postcmds"]="ln -sf /tools/bin/x86_64-pc-linux-gnu-gcc /tools/bin/cc; \
    echo 'int main(){}' > dummy.c; \
    cc dummy.c; \
    readelf -l a.out | grep ': /tools'; \
    rm dummy.c a.out"

    ["m4_p1_precmds"]="sed -i 's/IO_ftrylockfile/IO_EOF_SEEN/' ../lib/*.c;echo '#define _IO_IN_BACKUP 0x100' >> ../lib/stdio-i
mpl.h"
    ["m4_p1_confhz"]="--prefix=/tools"

    ["ncurses_p1_precmds"]="sed -i s/mawk// ../configure"
    ["ncurses_p1_confhz"]="--prefix=/tools --with-shared --without-debug --without-ada --enable-widec --enable-overwrite"
    ["ncurses_p1_postcmds"]="ln -sf libncursesw.so /tools/lib/libncurses.so"

    ["bash_p1_confqz"]="LDFLAGS=-L/tools/lib"
    ["bash_p1_confhz"]="--prefix=/tools --without-bash-malloc"
    ["bash_p1_postcmds"]="ln -s bash /tools/bin/DEFERSUBMEecho 'sh'DEFERSUBME"

    ["bison_p1_confhz"]="--prefix=/tools"
    ["bzip2_p1_custproc"]="cd ../ && make && make PREFIX=/tools install"

    ["coreutils_p1_confqz"]="FORCE_UNSAFE_CONFIGURE=1"
    ["coreutils_p1_confhz"]="--prefix=/tools --enable-install-program=hostname"

    ["diffutils_p1_confhz"]="--prefix=/tools"
    ["file_p1_confhz"]="--prefix=/tools"
    ["findutils_p1_confhz"]="--prefix=/tools"
    ["gawk_p1_confhz"]="--prefix=/tools"

    ["gettext_p1_precmds"]="cp /usr/local/lib/libgnuintl.so.8 /usr/local/lib/libgnuintl.so"
    ["gettext_p1_confhz"]="--disable-shared"
    ["gettext_p1_postcmds"]="cp gettext-tools/src/msgfmt /tools/bin;cp gettext-tools/src/msgmerge /tools/bin;cp gettext-tools/
src/xgettext /tools/bin"

    ["grep_p1_confhz"]="--prefix=/tools"
    ["gzip_p1_confhz"]="--prefix=/tools"

    ["make_p1_precmds"]="sed -i '211,217 d; 219,229 d; 232 d' ../glob/glob.c"
    ["make_p1_confhz"]="--prefix=/tools --without-guile"

    ["patch_p1_confhz"]="--prefix=/tools"

    ["perl_p1_custproc"]="cd ../ && ./Configure -des -Dprefix=/tools -Dlibs=-lm -Uloclibpth -Ulocincpth && make"
    ["perl_p1_postcmds"]="cp perl cpan/podlators/scripts/pod2man /tools/bin;mkdir -p /tools/lib/perl5/5.30.0;cp -R lib/* /tool
s/lib/perl5/5.30.0"

    ["Python_p1_precmds"]="sed -i '/def add_multiarch_paths/a \\        return' ../setup.py"
    ["Python_p1_confhz"]="--prefix=/tools --without-ensurepip"

    ["sed_p1_confhz"]="--prefix=/tools"
```

```
    ["tar_p1_confqz"]="FORCE_UNSAFE_CONFIGURE=1"
    ["tar_p1_confhz"]="--prefix=/tools"
    ["texinfo_p1_confhz"]="--prefix=/tools"
    ["xz_p1_confhz"]="--prefix=/tools"
)

#########begein main#########

#set +h
#umask 022
TC=/mnt/sda1/tmp
LC_ALL=POSIX
TC_TGT=x86_64-tc-linux-gnu
PATH=/tools/bin:/usr/local/bin:/bin:/usr/bin
export PROCESSTABLE TC LC_ALL TC_TGT PATH

#rm -rf $TC/tools $TC
#mkdir -p $TC
mkdir -p $TC/tools
chown tc:staff $TC/tools
ln -sf $TC/tools /

export

这里就是上一节说的bash module应用。

. /mnt/sda1/tmp/src/common/common

for packagedef in binutils-2.33.1.tar.xz:p1 gcc-9.2.0.tar.xz,mpfr-4.0.2.tar.xz,gmp-6.1.2.tar.xz,mpc-1.1.0.tar.gz:p1-1 linux-5.
4.3.tar.xz:p1 glibc-2.30.tar.xz:p1 gcc-9.2.0.tar.xz:p1-2 binutils-2.33.1.tar.xz:p2 gcc-9.2.0.tar.xz:p2 m4-1.4.18.tar.gz:p1 ncu
rses-6.1.tar.gz:p1 bash-5.0.tar.gz:p1 bison-3.4.2.tar.xz:p1 bzip2-1.0.8.tar.gz:p1 coreutils-8.31.tar.xz:p1 diffutils-3.7.tar.x
z:p1 file-5.37.tar.gz:p1 findutils-4.7.0.tar.xz:p1 gawk-5.0.1.tar.xz:p1 gettext-0.20.1.tar.gz:p1 grep-3.3.tar.xz:p1 gzip-1.10.
tar.xz:p1 make-4.2.1.tar.gz:p1 patch-2.7.6.tar.gz:p1 perl-5.30.0.tar.gz:p1 Python-3.8.0.tar.gz:p1 sed-4.7.tar.xz:p1 tar-1.32.t
ar.xz:p1 texinfo-6.7.tar.gz:p1 xz-5.2.4.tar.gz:p1

do

  CompilePackage $packagedef || exit 1

done

#########end main#########
```

以上脚本经过了调试，全程是可以正确运行的。在Constructing a Temporary System过后，脚本会进一步chroot到/tools/bin，这样就彻底隔离了本机上tce-load -iw compiletc安装的版本带来的脚本能继续但结果异常的情况。

相比前面文章，本文将begin main,end main块放到最后，将那个大大的processtable放在前面（注意到里面的$TC,$TC_TGT都被硬编了），我们这样做，就是为了和接下来讲解chroot下的脚本内容安排时，统一将processtable一起export过去，做到结构一致和清晰化。

一个fully retryable的rootbuild packer脚本,从0打造matecloudos(3):以lfs9观点看compiletc tools in advance

本文关键字：**shell**中的数组作为参数传递且带下标**,bash** 数组作为变量，**bash** 数组作为环境变量，**bash**中的以及多层单双引号转义处理，**bash**中**Shell**嵌套中的**$**转义处理**, chrooted shell $ escape**

在《一个fully retryable的rootbuild packer脚本,从0打造matecloudos(2)》中，我们见到了一种用数组化命令字符串和统一compiletarget()的的方式来构建lfs9的compiletc11基础部分，在那文的结尾，我们提到，为了清晰化这种布局，我们把那个大数组放到了前面，把common部分也作为bash module，这样主脚本文件的后面是逻辑主体，很短小很清晰，我们还提到，利用export可以将上层shell的processtable带入到下层shell，这就是本文要讲到的chroot环境和通过``，$()等命令替换，甚至bash -c ""等方式开启的子shell环境。

> 怎么在上下级shell间传递一个大大的命令数组？为什么要这样做？ 因为bash中，主从shell间的变量是主shell的变量能传到子shell，且数组不是一级公民，不能直接传递，也不好通过传递${arr[@]}的方式进行（因为如果这样，在子shell中还要重新组装一次为新数组），所以我们在主shell中，把数组先设为一个大大的字符串（数组化每条命令字符串和字符串化整个数组的区别正是本文与上文的区别之一），然后通过export导出这个大串到子shell，在子shell中重新构建为数组只需一个declare -A，declare -A是bash中定义关联数组的方法，我们在上二文都用到了，必须显式定义。而且它会进入到shell的环境变量，通过export和declare -A可以看到（注意，如果你在脚本中自动输出，里面的条目都是打乱的，并不影响脚本逻辑使用，如果手动粘贴命令，则可以看到按定义顺序显示的数组条目）。

接上文，本文讲解的是compiletc11的第三部分，将上面涉及到的要点进行讲解（注意上面提到三种环境chroot,substutue和bash -c，都将在接下来涉及）。

废话不说，上脚本。

增加了chroot支持的common

主要是把路径形式按chrootmode设置调用是chroot的/下还是非chroot下的/mnt/sda1/tmp下（文章2的普通模式）

```
#!/usr/local/bin/bash

export chrootmode='0'

while [[ $# -ge 1 ]]; do
  case $1 in
    -c|--chroot)
      shift
      chrootmode="$1"
      shift
      ;;
    *)
      if [[ "$1" != 'error' ]]; then echo -ne "\nInvaild option: '$1'\n\n"; fi
      echo -ne " Usage(args are self explained):\n\tbash $(basename $0)\t-c/--chroot\n\t\t\t\t\n"
      exit 1;
      ;;
    esac
  done

DOWNLOADPREFIX=${DOWNLOADPREFIX:-http://10.211.55.2:8000/buildmatecloudos}

[[ "$chrootmode" == '0' ]] && DIR_DOWNLOADS=${DIR_DOWNLOADS:-/mnt/sda1/tmp/build} || DIR_DOWNLOADS=${DIR_DOWNLOADS:-/build}
[[ "$chrootmode" == '0' ]] && DIR_GCC=${DIR_GCC:-/mnt/sda1/tmp/build} || DIR_GCC=${DIR_GCC:-/build}
[[ "$chrootmode" == '0' ]] && DIR_LOGS=${DIR_LOGS:-/mnt/sda1/tmp/logs} || DIR_LOGS=${DIR_LOGS:-/logs}

[ ! -d ${DIR_DOWNLOADS} ]        && mkdir ${DIR_DOWNLOADS}
[ ! -d ${DIR_GCC} ]        && mkdir ${DIR_GCC}
[ ! -d ${DIR_LOGS} ]       && mkdir ${DIR_LOGS}

然后是二个函数不变
......
```

buildrootbase中的字串化大数组以及多层引号转义处理

buildrootbase是另起的脚本文件，对应lfs9的compiletc11的“make basic filesystem”，以示与上文“constructing a temp system”分开。(如果你要看详细的lfs，从一个低的版本开始，如lfs62，高版本lfs9有相当大的省略)

这里面的重点是：由于是一个大“”括起来的大字串，包括开头(和末尾的)在内，实际上是整个数组定义字串化的全部内容，所以相比文章中pure processtable，这里命名为processtablestr，与前者比较主要的区别在：里面如果有双引号要用\转义，如果双引号中还有双引号，那么双引号要改成单引号，因为bash只允许最大三级的引号嵌套（这是我总结的不知对不对），即3级转义“”：一级不用转，二级用\，三级用'，直观的手段是你可以直接在一个bash shell中粘贴这个"processtablestr="大字串"以进行测试，如果没有出错，差不多就行了，如果有出错，在转化成数组后，调用数组时，会提示出错must use subscript when assigning associative array。

```
PROCESSTABLESTR="(
    [\"linux_p2_custproc\"]=\"cp -f /src/1.buildbase/kernelandtoolchain/config-5.4.3-tinycore64 ../.config;cd ../ && make mrpr
oper && make headers\"
    [\"linux_p2_postcmds\"]=\"find usr/include -name '.*' -delete; \
    rm usr/include/Makefile; \
    cp -rv usr/include/* /usr/include\"

    [\"glibc_p2_precmds\"]=\"cd ../ && patch -Np1 -i /src/1.buildbase/kernelandtoolchain/glibc-2.30-fhs-1.patch; \
    sed -i '/asm.socket.h/a# include <linux/sockios.h>' sysdeps/unix/sysv/linux/bits/socket.h; \
    ln -sfv ../../lib/ld-linux-x86-64.so.2 /lib64; \
    cd buildp2; \
    echo 'CFLAGS += -mtune=generic -Os -pipe' > configparms\"
    [\"glibc_p2_confqz\"]=\"CC='gcc -ffile-prefix-map=/tools=/usr'\"
    [\"glibc_p2_confhz\"]=\"--prefix=/usr --disable-werror --libexecdir=/usr/lib/glibc --enable-kernel=4.19.10 --enable-stack-
protector=strong --with-headers=/usr/include libc_cv_slibdir=/lib --enable-obsolete-rpc\"
    [\"glibc_p2_mdlcmds1\"]=\"find . -name config.make -type f -exec sed -i 's/-g -O2//g' {} \\; ; \
    find . -name config.status -type f -exec sed -i 's/-g -O2//g' {} \\; \"
    [\"glibc_p2_mdlcmds2\"]=\"touch /etc/ld.so.conf; \
    sed '/test-installation/s@DEFERSUBMEPERLDEFERSUBME@echo not running@' -i Makefile\"
    [\"glibc_p2_postcmds\"]=\"cp ../nscd/nscd.conf /etc/nscd.conf; \
    mkdir -p /var/cache/nscd; \
    make localedata/install-locales; \
    sed -i 's@lib64/ld-linux-x86-64.so.2@lib/ld-linux-x86-64.so.2@' /usr/bin/ldd; \
    mv -v /tools/bin/{ld,ld-old}; \
    mv -v /tools/x86_64-pc-linux-gnu/bin/{ld,ld-old}; \
    mv -v /tools/bin/{ld-new,ld}; \
    ln -sv /tools/bin/ld /tools/x86_64-pc-linux-gnu/bin/ld; \
    gcc -dumpspecs | sed -e 's@/tools@@g' -e '/\*startfile_prefix_spec:/{n;s@.*@/usr/lib/ @}' -e '/\*cpp:/{n;s@$@ -isystem /us
r/include@}' > /tools/lib/gcc/x86_64-tc-linux-gnu/9.2.0/specs; \
    sed -i 's@lib64/ld-linux-x86-64.so.2@lib/ld-linux-x86-64.so.2@' /tools/lib/gcc/x86_64-pc-linux-gnu/9.2.0/specs; \
    echo 'int main(){}' > dummy.c; \
    cc dummy.c -v -Wl,--verbose &> dummy.log; \
    readelf -l a.out | grep ': /lib'; \
    grep -o '/usr/lib.*/crt[1in].*succeeded' dummy.log; \
    grep -B1 '^ /usr/include' dummy.log; \
    grep 'SEARCH.*/usr/lib' dummy.log |sed 's|; |\n|g'; \
    grep '/lib.*/libc.so.6 ' dummy.log; \
    grep 'found' dummy.log; \
    rm -v dummy.c a.out dummy.log\"

    ......

)"
```

这里仅给出部分是因为还没有最终调试正确出所有的编译条目，注意与上一文的细节区别。

buildrootbase中的主逻辑以及子Shell和Shell嵌套中的$转义处理

下面的mkdir和mount为了追求脚本在调试时可retryable，都尽量用了判断和-p。mount部分需要重构。

```
#########begin main#########

TC=/mnt/sda1/tmp
mkdir -p $TC/dev
mkdir -p $TC/proc
mkdir -p $TC/sys
mkdir -p $TC/run
[[ ! -e $TC/dev/console ]] && mknod -m 600 $TC/dev/console c 5 1
[[ ! -e $TC/dev/null ]] && mknod -m 666 $TC/dev/null c 1 3
if awk -v status=1 '$2 == "$TC/dev" {status=0} END {exit status}' /proc/mounts; then   echo "yes"; else   mount --bind /dev $T
C/dev; fi
if awk -v status=1 '$2 == "$TC/dev/pts" {status=0} END {exit status}' /proc/mounts; then   echo "yes"; else   mount -t devpts
devpts $TC/dev/pts -o gid=5,mode=620; fi
if awk -v status=1 '$2 == "$TC/proc" {status=0} END {exit status}' /proc/mounts; then   echo "yes"; else   mount -t proc proc
$TC/proc; fi
if awk -v status=1 '$2 == "$TC/sys" {status=0} END {exit status}' /proc/mounts; then   echo "yes"; else   mount -t sysfs sysfs
 $TC/sys; fi
if awk -v status=1 '$2 == "$TC/run" {status=0} END {exit status}' /proc/mounts; then   echo "yes"; else   mount -t tmpfs tmpfs
 $TC/run; fi
```

```
chown -R root:root $TC/tools
ln -sf $TC/tools /

注意这里，这里完成了开头讲的通过env变量export传递processtablestr，和稍后在chrooted bash shell中将这个big strdeclare -A重构为数组的动作，注意$(echo \$PROCESSTABLE)，实际上开启了一个子shell作命令替换，这是一个用$()产生子shell的用法，所以这里要\转义$这个变量替换，所以这里可以解释为，export是没经过转义的原始字符串，而命令替换内部的echo出来的经过了转义的。否则$PROCESSTABLE根本取不到值。

实际上在讲到bash中嵌套的命令替换/变量替换的相关技术中，一般都会讲到内部的变量要经过转义。

chroot $TC /tools/bin/env -i MAKEFLAGS="-j 2" PATH=/bin:/usr/bin:/sbin:/usr/sbin:/tools/bin PROCESSTABLE="${PROCESSTABLESTR}"
/tools/bin/bash -c "
declare -A PROCESSTABLE=$(echo \$PROCESSTABLE); \
export; \

......

那么重点来了，如果上述是二级嵌套，那么这里又重新启动了一个bash -c，属于三级shell嵌套了，联系到上一节的引号嵌套转义，可以用相似的处理方法，将这个shell cmd str用''括起来（实际上''本来也是一种标识性的转义），而里面的命令和变量替换符，如CompilePackage \$packagedef，需要转义一次，否则$packagedef也根本取不到值。所以这里发生了一次三级引号转义和普通命令变量替换符$转义。

exec /tools/bin/bash --login +h -c ' \

touch /var/log/{btmp,lastlog,faillog,wtmp}; \
chmod 664  /var/log/lastlog; \
chmod 600  /var/log/btmp; \

这里的-c 1，使得buildrootbase采用正确的chroot相关路径设定。

. /src/common/common -c 1 ; \

for packagedef in linux-5.4.3.tar.xz:p2 glibc-2.30.tar.xz:p2 zlib-1.2.11.tar.gz:p1 file-5.37.tar.gz:p2 readline-7.0.tar.gz:p1
m4-1.4.18.tar.gz:p2 bc-2.2.0.tar.xz:p1 binutils-2.33.1.tar.xz:p3 gmp-6.1.2.tar.xz:p1 mpfr-4.0.2.tar.xz:p1 mpc-1.1.0.tar.gz:p1
gcc-9.2.0.tar.xz:p4; \
do \
  CompilePackage \$packagedef || exit 1; \
done; \
' \
"

#########end main#########
```

稍后的文章会给出完全调试正确后的结果，这里仅给出技术路线。

在tinycolinux上安装chrome

本文关键字：**chrome as desktop shell,uniform web os for admin and user**

一个APP总是由UI，中间件，业务逻辑组件，但唯有UI足以划分一个appstack，因为UI是一个APP必须的部分，即使是console也有TUI，现今我们看到的UI主要有二种，随OS发布的原生GUI，和随着webapp发展出来的WEBPAGE GUI，但实际上若好好归纳一下，VNC也是一种远程控制专用GUI。 硬件加速GL,DX也是一种UI，它是游戏APP的GUI，概言之，用图形或非图形技术实现的交互，只要它混合其它栈元素组成开发发布单元，它其实就可以是一种UI(你可以看到语言库和大型IDE中项目模板往往就是按appstack和UI类型组织的)，只不过技术实现上，因为WEB的UI往往是一种HTML渲染引擎的东西，所以它其实属于基于原生UI的高级UI，但是，无论如何，一种OS使用某种高级UI并以此建立起全部的APP生态是可能的，如果有这样一种OS，那么就法上它可以称为该UI的OS。

chromeos，webos就是这种东西，它展现的是webpage使用的appmodel完成的是web appstack面向的是webapp，用户可以单纯一个chrome就可以完成整个应用（当然webgame比起硬件加速的native cg game是二个东西），管理员可以用chrome完成维护任务，开发者可以就browser开发网页程序。chromeos就是一个linux系统核心+webkit UI组成的全部可用生态（desktop SHELL，AUI,工具，APP..），如果不存在还需要在这种OS玩大型3D游戏这种需求(况且现在已有webgl,websocket,html5这样的方案)，它其实是一种足够好用且可扩展到任何原原生UI和原生appstack占据的那些业务领域的东西。

其实,linux宏内核设计本来就是面向多样化被发布。它甚至可以per app os。chrome as os desktop all and AUI其实是合理的，它可以答配文尾提到的mineportal demos打造oc专用增强os。

好了，现在让我们在tinycolinux上安装GUI环境，以此原生UI为基础，实际上我们的最终目的不是这个，我们是要安装chrome，把它打造成类chrome os的东西，最终将tinycolinux发展成面向webui和webapp的专用OS。

在tinycolinux上安装x环境

根据http://wiki.tinycorelinux.net/wiki:adding_a_desktop_to_microcore有xvesa和xorg可选，我们安装的是full blown的xorg而不是tinycore.iso中自带的精简的vesa,因为chrome需要xorg，这次我们选择从3.x的tcz repos中下载而不是4.x的。

依次下载解压Xlibs.tcz,Xprogs.tcz,pixman.tcz,fontconfig.tcz,Xorg-7.5-bin.tcz,Xorg-7.5-lib.tcz，Xorg-fonts.tcz,Xorg-7.5.tcz,8个文件解压完先重启一次，不要马上执行startx(startx在Xprogs.tcz中),重启后在home/tc下执行startx，提示发现不了/etc/sysconfig/Xserver，手动准备/etc/sysconfig/Xserver文件，内容就是一行Xorg，保存，重新startx发现已经能够进入桌面（且以后每次重启登TC用户都会进入这个桌面），只是没有窗口管理器和右键菜单。

以上是xorg的configless配置，所有的配置都是用户配置，生成在home/tc，每次重启进入TC都进入桌面，是因为第一次/home/tc下startx已生成了配置文件，重启发现都会自动进入原先那个桌面(xserver文件那行xorg和configless的效果)，，加了新东西后测试或重来可删home/tc所有文件，重新在/home/tc下startx会生成新的一系列配置文件夹。

现在在基础桌面环境里安装flwm和wbar.tcz(mac style docker?),重启依然没有窗口管理和右键菜单，这是因为一直没有启动flwm，看来startx并没有在home/tc配置文件中将启动flwm逻辑加入其中。在tinycorelinux bootcode中加desktop=flwm，重启，现在有桌面和右键菜单了。

安装chrome

我下载的是3.x的32.6 M大小，版本为14.0.835.186的chromium-browser.tcz,在完成安装了x界面后，剩下的基本就是安装chrome和依赖tczs了。依次下载并安装下列18个tczs:

（由于以后每次tc登录都自动进入了桌面，你可以外部开个putty执行以下命令或sudo reboot,也可在桌面右键-terminal）

```
atk.tcz,cairo.tcz,gtk2.tcz,gdk-pixbuf2.tcz,pango.tcz
dbus.tcz
dbus-glib.tcz
libasound.tcz
nss.tcz
libevent.tcz
libcups.tcz
libgcrypt.tcz
libgpg-error.tcz
nspr.tcz
hicolor-icon-theme.tcz
shared-mime-info.tcz
chromium-browser.tcz
chromium-browser-locale.tcz
```

（在此过程中，进入桌面右键-terminal，/usr/local/bin/执行./chroum-browser测试所需tczs.）

全部安装完后重启一次，右键桌面APPS-chrouim，进入chrome，发现弹出对话框是乱码，点最右下角的那个乱码按钮，进入chrome，发现标题栏和地址栏是乱码，就算是在地址栏输入英文，也是乱码。这应该是chrome标题栏和地址栏，工具栏这些地方使用的字体是系统中没有的。非系统编码中缺少网页字体显乱码方块（系统此时是en,chrome也用的en,en-us?在/usr/local/chromium-browser-addons/locales中发现无en但有en-us项，改名也无用，调整系统etc/sysconfig/language也无用）

发现调chrome设置（乱码菜单中那个扳手图标进入）中跟字体，编码有关的选项都不行。这应该是这个prebuilt chrome版本的bug.

不过此时的chrome已能浏览网站，https的浏览不了。应该要源码重新编译。留到以后测试。

此处的为tinycolinux装GUI技术可以运用在将tinycolinux打造成virtiope这样的地方。恩恩

本文也是为《web开发发布的极大化：一套以浏览器和paas为中心技术的可视全栈开发调试工具，支持自动适配任何领域demo》一文作铺垫，这文中的demos设想如果全部完成，那就是bcxszy pt2 mineportal demos总成了，mineportal是一套demos集选型,xaas部分为diskbios，完成mineportal的平台选型,langys部分为engitor，完成mineportal的开发发布选型,appstack,apps部分为deepinoc，完成mineportal的源码选型,这三大demos最终为了使得基于大web的oc装箱可用，在线开发，集成一切必须的学习支持。学习者可以通过研究它的实现获得PHP开发的知识，且积累自己的codebase.

关注我。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一种设想：利用tinycorelinux+chrome模拟chromeos并集成vscodeonline

本文关键字：**Chromium as linux desktop,x11 –** 在没有**GUI**的情况下在服务器上启动**GUI**浏览器**,kiosk mode,Porteus Kiosk,x11 kiosk mode**，**chrome --enable-consumer-kiosk,webkitgtk kiosk**，发明自己的**chromeos**，直接用**vscodeonline**当**textcui,vscode as text os cui**

在《cloudwall：一种真正的mixed nativeapp与webapp的统一appstack》中我们讲到，web是一种从native和nativedev打洞出来的appmodel（它的协议是应用级的http，html是云UI,是一个可以没有一个宿主render的云GUI。其后端可以是lnmp中的nmp都是非平台依赖的APP级的引擎,这是一种"GUI/网络/存储/业务"的四栈云化的结构，在这些层次上它已经脱离了本地开发），webapp后端也可是k8s这种，在《利用openfaas faasd在你的云主机上部署function serverless面板》中，我们介绍过serverless,其实，云和云开发，本来就是serverless和terminal-less的。因为云的属性，强调的就是没有PC，没有一个专门平台，承认docker它是服务性APP中去掉server backend的自然平坦app结构:server"less" virtual cloud appliance，在《一门独立门户却又好好专注于解决过程式和纯粹app的语言，一种类C的新规范》我们讲到golang的极小运行时，和分布式开发中,"平台/语言/设计/人"四栈变三栈的正常现象，没有nativedev和平台依赖，云APP可以是三栈结构。即，云APP有三栈就OK。这些文章提出了对webapp本质的疑惑又一一进行了分析。----- 所以，总体上，web这种app早已经是云APP合理的存在。人们应该接受了这样的离散平台，和新型APP和开发。

> 再说一次，为什么说云开发无OS呢，因为os，已部基础和服务化了。app不直接寄宿在os或虚拟机上。开发变成了服务调用。或者说，这些开发来源不属于你自己的机器，因此你不能控制该服务所在OS（无须传统部署）。

在《cloudwall：一种真正的mixed nativeapp与webapp的统一appstack》中我们还提到chromeos作为webos(终端)的合理性，如果说一种app决定一种os，既然web是一种云UI，那么云OS，如果它基于PC上的OS实现而来，chromeos这样的实践就变得合理了(《minlearnpgramming》整书toc vol1已经重构为云os融合/云app融合二部分)。对于webos ,在《群晖+DOCKER，一个更好的DEVOPS+WEBOS云平台及综合云OS选型》我们还谈到docker based webapp/webos(而其实我们还谈到docker based webos被一种unikernel os代替)，无论如何，现在我们尝试在tinycorelinux上以kiosk mode安装chrome，基于以前也写过的一篇《在tinycorelinux上装chrome》文章，使kiosk mode的chrome成为系统启动时进入的唯一全屏独占应用，打造类似chromeos的实现，未来，我们直接用vscodeonline来代为该chromeos唯一的页面。使这样的linux发行成为vscodeos。

kiosk mode实现chromeos

这个模式的本质原理只需要在startx的末尾启动一个full screen的webkitgtk demo或基于chrome的demo,不需要装桌面管理器(因为它不启动桌面环境)。其实在tinycorelinux界有一个实现了，如webDesktop-v.0.2.iso，https://github.com/joaquimorg/早就不维护了。它基于webkitgtk，自己编译了一个叫desktop的全屏demo browser在/usr/local/bin，然后安装了x11，最后在startx中写入启动：desktop $PHOTOFRAME &。

而chromium（开源版chrome）是自带kioskmode的，因此省去了将它做成全屏独占的模式的工作，可以直接使用chromium –kiosk –incognito http://localhost的命令形式开启全屏模式并导向本地主页（为了安全起见也要禁用xorg设置中的tcp，如果你已有桌面管理器，要设置禁止禁用屏幕保护和自动重启）。你当然也可以使用cfe这样的lib自己编译定制demo，我们采用的测试版本是tc11.x 64bit，仓库中已有chromium-browser.tcz和x11支持。可以直接测试。可直接用ezremastered测试，这样方便。

下面介绍我的一个初步成功尝试：

```
利用ezremaster给tinycorelinux core增加桌面生成新iso

ezremaster.cfg部分：

cd_location = /home/tc/CorePure64-11.1.iso
temp_dir = /tmp/ezremaster
对于窗口不能全屏，中文不能显的处理bootcode,注意它与接下来chrome的lang参数中的连符作辨别
cc = lang=zh_CN vga=791
app_outside_initrd_onboot = chromium-browser.tcz
app_extract_initrd = openssh.tcz
chrome只能用xorg 作x11server,flwm这些用的都是xfdev不是xorg
app_extract_initrd = Xorg-7.7.tcz
extract_tcz_script = ignore

手动的post调整部分：
cd extract
为tc passwd一个密码，然后sudo cp -f /etc/passwd etc/passwd /etc/shadow etc/shadow
sudo touch var/lib/sshd,sudo cp usr/local/etc/ssh/sshd_config.ori usr/local/etc/ssh/sshd_config
顺便opt/bootlocal.sh，加入/usr/local/etc/init.d/openssh start
sudo touch etc/sysconfig/Xserver,写入Xorg，保存

接下来是重要关键部分：
tce-load -w getlocale graphics-4.5.3-tinycore64
仅安装xorg-7.7并startx，会failed in waitforx,waitforx在xlib.tcz中,尝试网上说的：1，注释/etc/sktl/.xsession中的waitforx行，2，isolinux bootcode在corepure64 后append喂vga=791或max_loop=256 iso=UUID=$rootuuid$isofile or after system starts,sudo fromISOfile /mnt/sdb1,startx,make TC boot and be used directly from an ISO file)，都不是解决问题的关键。/usr/local/bin/Xorg发现错误/var/log/xorg.0.log,no screens find，装lspci发现这是一张virtio gpu显卡。于是tce-load -iw graphics-5.4.3-tinycore64.tcz ，failed in startx消失，壁纸一闪成功进入黑屏桌面（因为没有桌面管理器）。
getlocale.sh选择四个zh_CN，生成mylocale.tcz(看来，tinycorelinux除了主框架部分Core-scripts，tce也能成为藏脚本的地方)再安装一个字体wget http://mirrors.163.com/tinycorelinux/4.x/x86/tcz/fireflysung.tczg到/tmp/，字体x86,64，跨版本能通用
这2个tcz和其依赖不能像openssh,xorg-7.7一样成功集成到initrd，原因不明，只好手动释放，字体只能手动：
```

```
sudo unsquashfs -f -d /tmp/tce/optional/glibc_gconv.tcz /tmp/ezremaster/extract
sudo unsquashfs -f -d /tmp/tce/optional/mylocale.tcz /tmp/ezremaster/extract
sudo unsquashfs -f -d /tmp/tce/optional/i2c-4.5.3-tinycore64.tcz /tmp/ezremaster/extract
sudo unsquashfs -f -d /tmp/tce/optional/graphics-4.5.3-tinycore64.tcz /tmp/ezremaster/extract
sudo unsquashfs -f -d /tmp/fireflysung.tcz /tmp/ezremaster/extract

最后，/etc/skel/.X.d，ls -alh,新建一个任意名字脚本（linux中命令行启动与x启动后的运行命令是二个不同的放置过程，命令行下显中文和图形界面中文能不能起作用是另一回事，这个注意），放置启动/usr/local/bin/chromium-browser --start-maximized --kiosk --lang=zh-CN https://www.baidu.com/的逻辑，保存退出。这样会在startx时，在home/tc/.x.d生成实际脚本，，ctl+alt+f1可退出桌面(黑屏)进入命令行界面,ctlc退出chromium再次ctlaltf1返回正常命令行
整包导出后为160多m
```

集成vscodeonline实现cloud dever os和writer's os

实现了kiosk mode还不够，我们的webos，要本地和跨网络都可用。打造一个沉浸式不离开IDE的全能环境和一个vscodeonline based cloud dever os，。我们直接用vscodeonline当这种os的textcui,vscode as text os cui直接把vscodeonline作为shell。,因为vscodeonline本身有一个命令行区web cmd，可以当复合shell使用。而虽然vscodeonline比较笨重，但是它的主要界面实际上是一个复合了的编辑器，可当真正的写作环境如写md文章:vscode没有工程组织，包，用磁盘文件代替工程文件类go用环境变量+文件夹当工程组织，这种方式很适合组织文档。。网络上，当代替备忘录这样的频繁打开/关闭的碎片化写作环境当然不行，但vscodeonline有一个remote ssh，断了也能很好连上，连上能持续连接很久，这放在一台开发终端上可用性还是很高的。

对于webos的shell选型有ssh cui/tui，也有桌面环境有rdp/vnc，这二种是最基本的，本地则有局域网投屏，wifi display这样的方案，跨网络有网页化，也有专门的ctx，remoteapp，虚拟桌面这样的专门方案。vscode online本身就有在远端开端口进行网页服务可用，而设置一个本地终端版的chrome devos，填补了本地也可能需要一个vscodeonlineos的情况。比如可以在利用remotessh的情况下，维持一个远程这种cloud dever os和一个本地终端版的chrome devos，设置一个同步插件同步二者的home目录，这个home区就是网盘。这种实用性还是蛮高的。

未来我们要在这个os中集成panel.sh，在这个panel上去掉pai，为terralang增加c header files装上terralang，为devplane增加后绑定/变更域名。vscodeonline as ide andplugin server+terralang server core+openfaas serverless core,使之成为devpanel.sh。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一种混合包管理和容器管理方案，及在tinycorelinux上安装containerd和openfaas

本文关键字：在tinycorelinux上装docker，virtual appliance vs virtual appstack，no cgroup mount found in mountinfo: unknown，jailing process inside rootfs caused: pivot_root invalid argument: unknown

在《利用openfaas faasd在你的云主机上部署function serverless面板》 和《panel.sh：一个nginx+docker的云函和在线IDE面板,发明你自己的paas(1),(2)》文中，我们谈到openfaas是一种基于containerd(docker官方经过对旧docker进行模块化重构后得到的二进制级的可复用结构，作为容器的运行时实现存在，containerd+runc+cni是默认的代替原docker的选型，https://github.com/containerd/containerd/releases也是把cni,containerd,runc这样打包一起的)的云函数面板，《tinycorelinux上编译安装lxc,lxd》《tinycorelinux上装ovz》这些文章也都是在linux上容器选型的例子。

> 容器化为什么这么重要，因为容器是现在最流行的原生virtual cloud appliance（cloud appliance化是app部署级别的融合，代表着“为云APP造一种包结构”，k8s这些被称为云原生所以你可以将其简单理解为云原生软件包，cloud appliance要与app开发用的内部cloud appstack融合化区别开）代表，作为统一部署方案，它主要关注解决集群和云上那些“软件发行资源配额的隔离”问题，软件的资源配额从来都是一个复杂的关联问题，可巧这些问题在本地和原生的包管理软件中（包主要关注解决依赖问题）也存在，linux上提供了统一的整套内核级支持方案（比如liblxc,libcgroups，基于它们+brtfs可以完全用shell发明一套简单的docker运行时，当然这跟我们需要的，最终完备的容器和容器管理系统containerd,openfaas是没法比的），另一方面，“资源隔离”稍微更进一步，就很容易与“软件怎么样启动”这些问题相关联，变成systemd这类软件要解决的问题（systemd-nspawn可以创建最轻量级的容器），进而变成k8s这类软件要解决的问题。所以，这三大问题融合和关联发生在方方面面，，很容易成为某种“混合包管理和容器管理”融合体系要解决的中心问题，进而需要这样一种软件，core os的https://github.com/rkt/rkt就是这一类软件的代表并为此构建出一个基于容器作为包管理的OS（虽然2020年中期它们准备dreprecate了）。

这就是说，容器化的实现可以简单也可以复杂，不同OS也有集成不同复杂容器管理的方案，除了core os的rkt这种，tc的tce pkg本身就是一种沙盒环境，不过它与上述提到的lxc,ovz,containerd这些真正意义的容器化没有关系。前述与openfaas相关的这些文章都是在流行的linux发行版中实践装openfaas的例子，接下来的本文将介绍尝试在tinycorelinux11中装真正的容器，即containerd和openfaas的安装实践过程。

> 这里的主要问题是，tc本身是一种raw linux发行版，追求小和简单，一般地，跟alpine一样tc往往作为容器guest os如boot2docker，鲜少在tc上装containerd作为容器服务器环境，因此，在tc中装容器可能会因为没有现成参考方案而显得繁琐。比如，tc也没有使用systemd这类复杂的init启动管理系统而是简单的sysv init（虽然systemd提出了一个巨大的init pid 1，但是它只关注“启动”，这点上，它还是符合kiss的）。一般linux发行都是依赖systemd处理容器设置的一些至关重要的基础问题,比如稍后我们会谈到systemd自动管理cgroups，，而这些在tc中都没有,需要手动还不见得能解决，

不管了，下面开始尝试实践，我们的测试环境是tc11。

1，制造一个containerd.tcz和faasd.tcz

准备一个集成了openssh,sudo passwd tc,做好了bootcode tce=sda1，echo过 /opt/tcemirror,/etc/passwd,/etc/shadow > /mnt/sda1/opt/.filetool.lst，tar 进restore=sda1 mydata.gz的基本ezremasterd tc11 iso，即《一个fully retryable的rootbuild packer脚本,从0打造matecloudos(2)》中第一小节那样的iso。我们开启二个引导了这个iso的虚拟机,都用parted格好sda1，这二虚拟机准备一个生成containerd.tcz和faasd.tcz用（生成后在其目录下python -m SimpleHTTPServer 80供以后下载，事先ifconfig看好ip），另一个测试生成的tcz(/opt/tcemirror设成第一台地址+11.x/x86_64/tcz正确的结构)，在第一台虚拟机的/mnt/sda1根目录中，安排这些文件：

```
准备文件夹结构和binaries:
docker采用前述文章的版本组合而成（做二文件夹一个containerd-root，其中cni放在/opt/cin/bin,runc放在/usr/local/sbin,containerd放在/usr/local/bin，一个faasd-root，其中放/usr/local/bin/faasd,faas-cli，最后，前文提到的几个offline docker image也集进来放在faasd-root/tmp/*.tar）。这些exe设好chmod +x，由于这些exe都是go的，都是静态链接的，在tc11上可直接运行(lib64一定要ln -s 一下到lib，否则ctr不能起作用)。

准备几个配置文件：
一些containerd和faasd启动时的动态文件，不用创建。否则会导致只读文件系统无法写入错误
/etc/cni/net.d/10-openfaas.conflist
/var/lib/faasd/secrets/*
/var/lib/faasd/resolv.conf
/var/lib/faasd-provider/resolv.conf
这些文件必须要
/var/lib/faasd/docker-compose.yaml
/var/lib/faasd/prometheus.yml

准备overlay module:
在第一台中tce-load -iw bc compiletc perl5重新编译kernel，在config中把config_overy_fs打开为m，得到overlay module,因为它在tc11中被关闭了。
下载tc11中所需的kernel编译文件http://mirrors.163.com/tinycorelinux/11.x/x86_64/release/src/kernel/到/mnt/sda1
解压并cp config-5.4.3-tinycore64 linux-5.4.3/.config
sudo make oldconfig，提示几个交互项直接回车
sudo make install
sudo make modules_install
把得到的overlay module file(在/lib/modules/fs中)放到准备打包的containerd.tcz文件夹结构中。
```

```
准备2个服务文件并分别chmod +x：
containerd-root/usr/local/init.d/start-containerd:
/sbin/modprobe overlay
/usr/local/bin/containerd
containerd-root/usr/local/init.d/start-faasd:
for i in 1 2 3; do [[ ! -z "$(ctr image list|grep basic-auth-plugin)" ]] && break;ctr --address=/run/containerd/containerd.soc
k image import /tmp/faasd-containers/basic-auth-plugin-0.18.18.tar;echo "checking basic-auth ($i),if failed at 3,it may requir
e a reboot"; sleep 3;done
for i in 1 2 3; do [[ ! -z "$(ctr image list|grep nats)" ]] && break;ctr --address=/run/containerd/containerd.sock image impor
t /tmp/faasd-containers/nats-streaming-0.11.2.tar;echo "checking nats ($i),if failed at 3,it may require a reboot"; sleep 3;do
ne
for i in 1 2 3; do [[ ! -z "$(ctr image list|grep prometheus)" ]] && break;ctr --address=/run/containerd/containerd.sock image
 import /tmp/faasd-containers/prometheus-v2.14.0.tar;echo "checking prometheus ($i),if failed at 3,it may require a reboot"; s
leep 3;done
for i in 1 2 3; do [[ ! -z "$(ctr image list|grep gateway)" ]] && break;ctr --address=/run/containerd/containerd.sock image im
port /tmp/faasd-containers/gateway-0.18.18.tar;echo "checking gateway ($i),failed at 3,it may require a reboot"; sleep 3;done

for i in 1 2 3; do [[ ! -z "$(ctr image list|grep queue-worker)" ]] && break;ctr --address=/run/containerd/containerd.sock ima
ge import /tmp/faasd-containers/queue-worker-0.11.2.tar;echo "checking queueworker ($i),if failed at 3,it may require a reboot
"; sleep 3;done
cd /var/lib/faasd
/usr/local/bin/faasd provider
/usr/local/bin/faasd up

准备mkall.sh放到/mnt/sda1根下chmod +x起来：
mkall.sh的内容(注意到统一作为tc:staff存放到tcz中):
rm -rf containerd.tcz containerd.tcz.md5.txt faasd.tcz faasd.tcz.md5.txt faasd.tcz
mksquashfs containerd-root containerd.tcz -noappend -no-fragments -force-uid tc
md5sum containerd.tcz > containerd.tcz.md5.txt
mksquashfs faasd-root faasd.tcz -noappend -no-fragments -force-uid tc -force-gid staff
md5sum faasd.tcz > faasd.tcz.md5.txt
echo containerd.tcz > faasd.tcz.dep
```

sudo ./mkall.sh打包好的tcz各80多m，准备好后我们就可以在第二台测试了，tce-load -iw faasd后经过测试不满意可删除/mnt/sda1/tce/optional下的tcz重启重来，我们需要不断mkall并测试生成的二个tcz。

2，测试

第一次测试启动start-containerd,start-faasd，出现:no cgroup mount found in mountinfo: unknown这就是上面谈到tc11不具有自动处理cgroups的逻辑。而containerd依赖它们。tc11中的kernel config中提供了linux对容器的内核支持基础，只是没有更进一步。

> CGroup 提供了一个 CGroup 虚拟文件系统，作为进行分组管理和各子系统设置的用户接口。要使用 CGroup，必须挂载 CGroup 文件系统。这时通过挂载选项指定使用哪个子系统。 需要注意的是，在使用 systemd 系统的操作系统中，/sys/fs/cgroup 目录都是由 systemd 在系统启动的过程中挂载的，并且挂载为只读的类型。换句话说，系统是不建议我们在 /sys/fs/cgroup 目录下创建新的目录并挂载其它子系统的。这一点与之前的操作系统不太一样。

针对于此，很幸运我们找到了https://gitee.com/binave/tiny4containerd/blob/master/src/rootfs/usr/local/etc/init.d/cgroupfs.sh,它使用old docker，并且基于https://github.com/tianon/cgroupfs-mount/（这个工程https://gitee.com/binave/tiny4containerd/src/rootfs/usr/local/这里的lvm动态扩展分区脚本和docker服务，cert处理等函数也不错，可为未来所用），里面有几句。

```
mount -t tmpfs -o uid=0,gid=0,mode=0755 cgroup /sys/fs/cgroup

如果你没有上面这句mount 接下来会mkdir: can't create directory 'cpu': No such file or directory, 因为/sys/fs/cgroup只是内核给的fake fs

cd /sys/fs/cgroup;

# get/mount list of enabled cgroup controllers
for sys in $(awk '!/^#/ { if ($4 == 1) print $1 }' /proc/cgroups); do
    mkdir -p $sys
    if ! _mountpoint -q $sys; then
        if ! mount -n -t cgroup -o $sys cgroup $sys; then
            rmdir $sys || true
        fi
    fi
done
```

我们把cgroupfs放跟containerd-root/usr/local/etc/init.d/containerd并排，在containerd脚本中启动containerd前加入/usr/local/etc/init.d/cgroupfs.sh mount这句。打包再测试：出现jailing process inside rootfs caused: pivot_root invalid argument: unknown(我也一直没有测试https://gitee.com/binave/tiny4containerd/中的docker会不会出现这错误，不过听说有遇到了https://forums.docker.com/t/tinycore-8-0-x86-pivot-root-invalid-argument/32633),

> In a system running entirely in memory, after an upgrade from 17.09.1-ce to 17.12.0-ce, docker stopped creating containers, failing with message like docker: Error response from daemon: OCI runtime create failed: container_linux.go:296: starting container process caused "process_linux.go:398: container init caused \"rootfs_linux.go:107: jailing process inside rootfs caused \\"pivot_root invalid argument\\"\"": unknown..

查网上说要使用DOCKER_RAMFS=true环变，我试了没用。

在其它非tc上相似的容器产品也有人遇到了：https://engineeringjobs4u.co.uk/how-we-use-hashicorp-nomad，针对于此它们做了一个内核补丁：https://lore.kernel.org/linux-fsdevel/20200305193511.28621-1-ignat@cloudflare.com/

> The main need for this is to support container runtimes on stateless Linux system (pivot_root system call from initramfs). Normally, the task of initramfs is to mount and switch to a "real" root filesystem. However, on stateless systems (booting over the network) it is just convenient to have your "real" filesystem as initramfs from the start.

对linux543/fs/namespace.c进行手动patch,mnt_init()定义前增加，和中间增加新加的代码。但是没用。因此按《利用hashicorp packer把dbcolinux导出为虚拟机和docker格式(3)》的方法转为传统硬盘安装方式。问题解决。

然后又出现了： Error creating CNI for basic-auth-plugin: Failed to setup network for task "basic-auth-plugin-1210": failed to create bridge "openfaas0": could not add "openfaas0": operation not supported: failed to create bridge "openfaas0": could not add "openfaas0": operation not supported

这个问题其实在意料之中，因为从前面文章的经验来看，我们一直对containerd中的那个cni必须要起作用留了个心，可是faasd up产生了10openfaas.conflist后，我一直尝试ifconfig，都没看到第三个网卡。

网上有人提示说是CONFIG_BRIDGE_VLAN_FILTERING，看tc11的kernel，config_bridge被作为模块了，它的file应该是bridge.ko之类。但modprobe bridge没用，tce-load -iw original-modules-5.4.3-tinycore64，这下成功了。（安装了这个包之后，控制台显示好多设备都认到了）

再测试： Error: Failed to setup network for task "basic-auth-plugin-3894": failed to locate iptables: exec: "iptables": executable file not found in $PATH: failed to locate iptables: exec: "iptables": executable file not found in $PATH，需要tce-load -iw iptables，

至此，faad up启动成功。containerd控制台显示warning,memory cgroup not supported,应该是kernel config，又没设好。

我们接下来要做的，就是把usr/local/etc/init.d下的几个服务文件做完善点。参考https://github.com/MSumulong/vmware-tools-on-tiny-core-linux/blob/tiny-core-6.3/additional-files/etc/init.d/open-vm-tools和https://gitee.com/binave/tiny4containerd/blob/master/src/rootfs/usr/local/etc/init.d/

利用增强tinycorelinux remaster tool打造你的硬盘镜像及一种让tinycorelinux变成Debian install体的设想

本文关键字:增强tinycorelinux remaster tool，tinycorelinux 开机加载module,x509: certificate signed by unknown authority

在前面很多云主机装机相关的文章中，我们都讲到debian的netinstall实现云主机装机，它并不利用pxe这种cs结构和另外的装机服务器之类的东西，而是debian固有装机方式中的一种，即简单利用软件包仓库和chroot机制在线操作硬盘provision出一个ramos pe化os的原理，---- 这在《一个fully retryable的rootbuild packer脚本,从0打造matecloudos》和《把DI当online packer用:利用installnet制作一个云装机packerpe》都讲过。那么它在其它linux dists上有实现吗？

这种替代类似方案之一就是tinycorelinux，它追求小跟di一样，而且它本身就是一个ramos，（tinycorelinux内存os是什么意思呢？其实整个tc也可以通过把initrd.gz cpio -idmv < 到硬盘中运行。但是默认情况下，如果不提供tce=sda1之类的bootcode 及after bootinto system then tce-setup重配置，那么它的包是下载到/tmp这个内存fs和/挂载点的。如果指定硬盘上的tce目录加载，除了一些极端情况，如docker工作在ramfs会现privot_root异常，它实际上跟普通硬盘linux无异，这就是tc的stateless和cloud mount属性），而且tinycorelinux也有在线仓库，它的remaster tool和bootlocal.sh也可以成为简单的preseed和provision机制，可以简单地模拟出debian的netinstall环境。

ezremaster工作在gui下，通过读取仓库下的info.lst.gz形成软件列表（find 11.x/x86_64/tcz/*.tcz -print | sed -e 's/11.x\/x86_64\/tcz\///' > 11.x/x86_64/tcz/info.lst gzip -cf 11.x/x86_64/tcz/info.lst > 11.x/x86_64/tcz/info.lst.gz），但是也可以让它工作在命令行下，提取其中的/usr/local/bin/remaster.sh，然后手动装好依赖tce-load -iw mkisofs-tools advcomp，调用remaster+ezremaster.cfg(这个必须为绝对路径)+命令进行工作，工作原理见脚本本身，过程是给一个模板iso，然后blalalala....最后生成一个新iso。

！！重点来了：我们想通过定制/增强tc的ezremaster tool来讲解tc的增强能力。比如我们想使之生成硬盘镜像呢（以后考虑进一步弄成pebuilder.sh之类的东西，呆会你就会看到，整remaster.sh的逻辑跟pebuilder.sh共享同样的原理和流程,这种可组装能力有点让tc等同metalinux的意思了）？该如何进行。不废话，（我们使用前面《一种设想：利用tinycorelinux+chrome模拟chromeos并集成vscodeonline》和《一种混合包管理和容器管理方案，及在tinycorelinux上安装containerd和openfaas》的实践案例，顺便在这里增强一下上述二文），我们使用的测试环境是上述二文中经过了打包openssh.tcz的corepure64-11.1-remasterd.iso，用它开的一台虚拟机，因为要手动安装ezremaster tool的依赖，所以还要：tce-load -iw mkisofs-tools advcomp parted grub2-multi util-linux ca-certificates（其实其它linux发行也可以运行remaster.sh，只要它们有cpio, tar, gzip, advdef, and mkisofs. Advdef is used to re-compress the image with a slightly better implementation, producing a smaller image that is faster to boot,后四个tcz是新增的，供生成硬盘镜像用，没ca-certificates会出现ctr/faas-cli拉镜像时x509: certificate signed by unknown authority）

不废话了，直接提供脚本：

1,ezremaster.cfg文件

一般用CorePure64-11.1.iso作模板，corepure64中的vmlinuz64，并没有一些像它的http://mirrors.163.com/tinycorelinux/11.x/x86_64/release/src/kernel/config-5.4.3-tinycore64一样，corepure64/lib/modules中为追求小也删了大量modules，我们用集成origmod.tcz的方法代替前面仅集成graphics modules.tcz等部分模块的方法以后，好多驱动认到了，鼠标跟指针分离的现象也貌似解决了？。但还是遗留了一些问题，比如virtio_blk实际上没有做成builtin，在云主机上工作时，需要寻求另外编译vmlinuz或寻求启动时加载modules的相关方案。

其它注意点：1，如果在gui中，loglevel=3和cde（如果你在cfg中写过了extract outside intro）是默认的，不需再写，2，因为要集成的tce比较多，最终的iso比较大，进程耗时，所以在/mnt/sda1/ezremaster(sudo chown tc:staff)上进行。3，faasd.tcz是使用了自己的镜像，（因为用od托管的tce镜像必须要用到wget ssl，因此临时先把/opt/tcemirror切换到mirrors.163.com/tinycorelinux下载openssl-1.1.1，然后切换到od主镜像），4，勾选产生的那个copy2fs.flg目前还考虑不到有什么用。,tc有多个藏tcz的地方，如tce,cde,initrd的集成包，且可以同时起作用，那么它是如何处有多个tcz目录的依赖项和防止冲突的？因为在tce=sda1且root=/dev/sda1硬盘模式下，/usrlocal/tce.installed会导致混乱.

其实我是把cfg文件当字串内置到remaster.sh中供grep处理的（本来是grep "^cd_location = " $input文件 | awk '{print $3}'），如下：

```
input="cd_location = /mnt/sda1/CorePure64-11.1.iso
temp_dir = /mnt/sda1/ezremaster/
cc = home=sda1 opt=sda1 tce=sda1 restore=sda1
app_outside_initrd_onboot = chromium-browser.tcz
app_outside_initrd_onboot = iptables.tcz
app_outside_initrd_onboot = faasd.tcz
app_outside_initrd_onboot = nginx.tcz
app_extract_initrd = openssh.tcz
app_extract_initrd = original-modules-5.4.3-tinycore64.tcz
app_extract_initrd = Xorg-7.7.tcz
extract_tcz_script = ignore
copy2fs.flg"

if [ ! -n $input ]; then
    echo "input is empty"
    exit 2
fi
```

```
......

cd_location=`echo "$input" | grep "^cd_location = " | awk '{print $3}'`
temp_dir=`echo "$input" | grep "^temp_dir = " | awk '{print $3}'`
```

脚本中其它的cat $input,都要改一下为上述形式，参数判断逻辑也改一下，使得一个参数也能通过。

2，改造脚本，使之生成硬盘镜像

把下面这段加在package(){...}后，这部分主要的逻辑是硬盘镜像的挂载初始化与析构。

注意到，remaster.sh rebuild function本身是可以覆盖执行的，extract目录中的initrd在经过第一次iso打包后就已经变化，集成了app_extract_initrd指定的三个tczs，除非extract必变以后它的体积大小不会改变。（但保险起见，你依然可以选择在第一次脚本完成后打包那个/mnt/sda1/ezremaster为ezremaster-inital.gz备用），

我们加入了一些fixandmodinitrd，因此，我们必须保证extract目录也是依然可以覆盖使用的。因此我们把这个函数变成了fixandmodinitrdforonly，用了一个 if [! -f $temp_dir/extractalreadyprocessed]; then判断，这种手法在《一个fully retryable的rootbuild packer脚本,从0打造matecloudos》系列中很常见。

（前面tce-load中这个util-linux主要是因为busybox中的那个losetup太旧，没法用）

```
export BUILD_PACKAGE_MNT_PT=/tmp/buildpackage

fixandmodinitrdforonlyonce() {

    if [ ! -f $temp_dir/extractalreadyprocessed ]; then

        sudo cp -f /etc/passwd etc/passwd
        sudo cp -f /etc/shadow etc/shadow

        sudo cp -f /usr/local/etc/ssh/sshd_config.orig usr/local/etc/ssh/sshd_config
        sudo cp -f -R /usr/local/etc/ssl usr/local/etc/
        sudo cp -f -R /usr/local/etc/ssh usr/local/etc/
        sudo mkdir -p var/lib/sshd/

        sudo cp -av /usr/local/etc/ssl/ etc/ssl

        sudo sh -c "echo Xorg > etc/sysconfig/Xserver"
        sudo sh -c "echo /usr/local/bin/chromium-browser --start-maximized http://127.0.0.1 > etc/skel/.X.d/chromium-browser"
        sudo chmod +x etc/skel/.X.d/chromium-browser

        sudo sh -c "echo /usr/local/etc/init.d/openssh start >> opt/bootlocal.sh"
    sudo sh -c "echo http://d.shalol.com/mirrors/tinycorelinux/ >> opt/tcemirror"

    sudo sh -c "find . | cpio -o -H newc | gzip -2 > $temp_dir/image/boot/corepure64.gz" || exit 22
    sudo advdef -z4 $temp_dir/image/boot/corepure64.gz || exit 23

        sudo touch $temp_dir/extractalreadyprocessed

    else
        echo fixandmodinitrdforonlyonce is already processed!
    fi
}

packagehd() {
    dev_buildpackage=$(mount | grep "$BUILD_PACKAGE_MNT_PT" | awk '{print $1}')
    if [ -z "$dev_buildpackage" ];then
        echo "- Creating new dev_buildpackage disk"
        sudo rm -rf $temp_dir/buildpackage.raw
        sudo dd if=/dev/zero of=$temp_dir/buildpackage.raw bs=1024 count=20971520
        dev_buildpackage=`sudo /usr/local/sbin/losetup -fP --show $temp_dir/buildpackage.raw | awk '{print $1}'`
        echo

        [ -n "$dev_buildpackage" ] && {
                sudo parted -s "$dev_buildpackage" mktable msdos
                sudo parted -s "$dev_buildpackage" mkpart primary ext3 2048s 100%
                sudo parted -s "$dev_buildpackage" set 1 boot on
                sudo mkfs.ext3 "$dev_buildpackage"p1
        }

        [ ! -d "$BUILD_PACKAGE_MNT_PT" ] && sudo mkdir "$BUILD_PACKAGE_MNT_PT"
        sudo mount "$dev_buildpackage"p1 "$BUILD_PACKAGE_MNT_PT"

        sudo grub-install --boot-directory="$BUILD_PACKAGE_MNT_PT"/boot "$dev_buildpackage"
        sudo grub-mkconfig -o "$BUILD_PACKAGE_MNT_PT"/boot/grub/grub.cfg
```

```
        sudo sh -c "echo set timeout=3 >> $BUILD_PACKAGE_MNT_PT/boot/grub/grub.cfg"
        sudo sh -c "echo menuentry \"this is a raw hd\" { >> $BUILD_PACKAGE_MNT_PT/boot/grub/grub.cfg"
        sudo sh -c "echo linux /boot/vmlinuz64 loglevel=3 tce=sda1 opt=sda1 home=sda1 restore=sda1 cde >> $BUILD_PACKAGE_MNT_P
T/boot/grub/grub.cfg"
        sudo sh -c "echo initrd /boot/corepure64.gz >> $BUILD_PACKAGE_MNT_PT/boot/grub/grub.cfg"
        sudo sh -c "echo } >> $BUILD_PACKAGE_MNT_PT/boot/grub/grub.cfg"


    sudo depmod -a -b $temp_dir/extract `uname -r`
    sudo ldconfig -r $temp_dir/extract

    cd $temp_dir/extract
    umount $temp_dir/extract/proc >/dev/null 2>&1
        fixandmodinitrdforonlyonce


    cd $temp_dir/image
        sudo cp -av boot/ "$BUILD_PACKAGE_MNT_PT"/
        sudo cp -av cde/ "$BUILD_PACKAGE_MNT_PT"/tce


    fi
    # Automatically remove DISK on exit
    trap 'echo; echo "- Ejecting dev_buildpackage disk"; cd "$HOME"; sudo umount "$BUILD_PACKAGE_MNT_PT" && sudo /usr/local/sb
in/losetup -d "$dev_buildpackage"' EXIT
}
```

最后，参数处加一条 packagehd) packagehd ;;作调用

测试，修改镜像为od主镜像，运行remaster.sh /mnt/sda1/ezremaster.cfg rebuild生成iso，然后remaster.sh /mnt/sda1/ezremaster.cfg packagehd，生成硬盘镜像，成功，脚本是自带tail -f ezremaster.log效果的，里面会出现一些wget 404，应该是一些无关紧要的info文件等的下载。无妨。

最后你可以你可以tar这个镜像用tce-load -iw python，sudo python -m SimpleHTTPServer 80下载这个镜像，默认0.0.0.0。如果发现卡住，下载不了ctrl c一下python进程。

最后，这个脚本可以保存为remaster.sh，但是因为tc没有arm64，所以准备转debian，这样也可以与pebuilder.sh针对的debian netinstall接上，但是默认提供的最小的netinstall.iso也要几百M，网上看到一篇《基于 debootstrap 和 busybox 构建 mini ubuntu》https://www.cnblogs.com/fengyc/p/6114648.html，据称可以将debian或ubuntu精简到40-50m，或更小，它的思路主要是将启动脚本依赖的占用替换为简单bb和内核模块精简一下，所以以后有机会了尝试一下，还看到一个slax的live系统，我们要尝试用来代替tc的这个新系统一定要像slax和tc一样live加载扩展，而且要/ext模块目录和/os，用来保存数据的/data单独三个文件夹layout到/下。

一种用buildkit打造免registry的local cd/ci工具,打通vscodeonline与openfaas模拟cloudbase打造碎片化编程开发部署环境的设想

本文关键字：如何直接修改**docker**中的文件,从外部编辑**dockernamespace**内文件,**share data between host and container?**,定制镜像和容器，不经过任何**registry**重建/修改/**commit docker**镜像，**Creating an image from a commited snapshot**,把**openfaas**还原为非**docker**结构，可以直接在**docker**内编辑集成，**overlay fs** 读写，把你的**vps**做成**cloudfunction**环境(小程序碎化片前后端支持环境)和配备一个云**IDE**开发环境

在前面《利用onemanager配合公有云做站和nas》，《利用fodi给onemanager前后端分离》我们讲到腾讯cloudbase和scf，提到它的后端管理页面有一个在线IDE。类似内置的简化版vscode online，在《在云主机上安装vscodeonline》，《panel.sh：一个nginx+docker的云函和在线IDE面板,发明你自己的paas》我们讲到自己构建这种online ide+serverless的paas面板技术，现在我们考虑联接起containerd+vscode，打通这二者模拟cloudbase碎片化编程开发部署环境的设想：

我们知道serverless技术背后是容器跑起来的。一般地，dockerd+docker-cli主要被设计成stateless and immutable only as 自动化代码部署工具(部署和生成分离,CI/CD分离)，docker维护一个image build in a place,then in another place的image pull -> image deploy -> container run的cycle（runc还有一个类似进程管理的task命令）是一个以registry为中心的pull->build->deploy循环流程。这就导致了一些问题：

> 如果不作特别说明，本文说到的docker/containerd可以混淆理解，containerd和docker都是容器运行时，docker最早是LXC（Linux Container）的二次封装发行，后来使用的是Libcontainer技术,从1.11开始进一步演化为 runC 和 containerd，其利用的也是Linux内核特性namespaces(名称空间) 、 cgroups (控制组)和AUFS(最新的overlay2)等技术，是操作系统层面的虚拟化技术,containerd重点是继承在大规模的系统中，例如kubernetes，而不是面向开发者，让开发者使用，更多的是容器运行时的概念，承载容器运行。docker偏指docker官方发布的那个运行时版本和dockercli工具。

1）docker aufs/overlayfs被设计成只读，和不变immutable环境(执行df可以看到其挂载到主机上的overlayfs mount，就像tc下可以看到tce用的loop一样，类似/run/containerd/io.containerd.runtime.v2.task/default/xxxx/rootfs)，里面的文件系统虽然是挂载到主机上的但不推荐直接读写因为它会破坏容器元数据，，要修改docker的文件，我们要借助docker提供的命令，进入具体docker容器空间按它正常的命令逻辑，去获得文件系统视图(进入Docker容器比较常见的几种做法如下：1.使用docker attach,2.使用SSH,3.使用nsenter,4.使用exec)后操作,之后docker cli(或其它定制的高级cli如k8s runctr)和dockerd管理器会保证整个应用库中的（/var/lib/docker/image/xxx）的容器元数据不受破坏。------ 但是虽然容器可用这种方法编辑内容，其结果也是不能保存的：下次重启容器，还是会重新进入到这个以pull image开始的循环，除非你重新生成镜像或提交容器即时更改。如上所述docker是部署和生成分离，docker中运行镜像和生成镜像这往往是一个脱机操作 ------ 对于提交即时更改，docker有一个docker commit命令就是干这事的（类似git commit和去主机快照技术），但除非用于特定目的(比如保存现场，比如对于某些容器每次我们只需修改/home/app下的东西并commit，这二种场景下，其实也没什么问题。)这也是不受推荐的，因为它会导致复杂的镜像:首先，如果在安装软件，编译构建，那会有大量的无关内容被添加进来，如果不小心清理，将会导致镜像及其臃肿。此外，使用docker commit 意味着所有对镜像的操作都是黑箱操作，生成的镜像也被称为黑箱镜像，换句话说，就是除了制定镜像的人知道执行过什么命令，怎么生成的镜像，别人根本无从得知，。换言之，docker fs并不是一个flat fs结构，不能按照普通的文件系统的逻辑直接获得路径并读写。docker instant commit虽然受支持但也是不受推荐的，

2) 你也完全可以将需要变动的部分放在外部，然后mount进容器，这个问题的本质其实是打通宿主机和容器中的某些通道，类似我们使用虚拟机时在宿主和虚拟机内共享一个volume或文件夹，类似远程桌面将读写通道重定向到你本机（实际上是下载，对于远程桌面所在服务器是上传），我们将host上可写的数据被作为volume mount(或mount bind)进来到容器，将更改和变动只保存在宿主端容器动态持续CI/CD即可，但是其有权限处理问题(rootless containers...)，我们知道这二套空间的用户和文件都不一样。即我们把可以把装有app的那个/home/app独立出来放到主机。不内置入镜像。containerd层只维护一个到主机相关目录的索引,----- Docker itself does not seem to encourage using host-mounted writable volumes. 共享volume不受推荐。

> 其实docker的应用就是被设计作为执行空间，在很多文章中我们都讲到不要将其作为存放任何数据空间，一是因为它用了奇怪的fs，二是因为它与宿主机share data也是非常不直观的,见《10 ways to avoid in using docker》。。但是这是一种刚需：应用一般都是存在可读写的部分和可变存放数据的部分的，如何让docker组成一个普通的符合可读写的应用环境呢？那么程序上可以做到类似效果吗或有改进方案吗？

如果能做到或能改进，那么在openfaas+vscode上，我们将这二者结合打通,形成cloudbase后端管理页面代码编辑器编辑保存类似的效果，这样我们就可以不依赖openfaas-cli式以registry为中心的那个循环流程（请无视local registry搭建技术，因为它也是以registry为中心的），直接在online ide或web后台管理中，即时地修改代码（这种思路是在经典流程中插入一个commit过程绕过registry 作inplace build using cache，最后按需push deploy回归到经典流程-这步不是必要的）。

buildkit似乎可以完成这个工作。

docker commit方案

buildkit是moby工程的一部分，它是从源码构建到容器的统一构建工具，在buildkit这类工具之前，业界都是用Docker in docker来处理的，后来转为buildkit这类专门工具，它支持多种docker运行时和docker-cli构建时，你可以把buildkit它想象成docker compose的一种上层工具，以达到docker file format无关构建和面向部署各种后端构建(处在中间层插入一个层，正符合我们上述提到的绕过registry,inplace building不push的要求)。即，它是一种统一面向容器的统一CI/CD工具，谈到CI/CD，就是你在各种CI工具和其它提供商产品后台修改git源码容器自动实时构建时那一大块黑色输出框背后用的那一套工具对应的输出界面。因为它有分布式cache构建(传统的Docker build cache only works on the same host)，并行构建，统一

中间格式这些。面向多种前端和多种后端，（buildkitd daemon支持runc后端和containerd后端，By default, the OCI (runc) worker is used. You can set --oci-worker=false --containerd-worker=true to use the containerd worker.你也可以用文件定制它的参数 https://github.com/moby/buildkit/blob/master/docs/buildkitd.toml.md，它是干什么用的呢。将运行时和构建时像dockerd/dock-cli一样形成cs结构，通过 http 通信的方式执行构建。）openfass build就使用了buildkit，gitlab也有一个buildkite，都是统一容器CI/CD工具链上的东西。

> moby就是我们前文《hyperkit:一个full codeable,full dev support的devops及cloud appmodel》介绍hyperkit的时候那家厂商造的，就是docker那家厂商。里面的好多产品，都是docker转正前的试验品。包括moby,和hyperkit等。在《群晖+DOCKER，一个更好的DEVOPS+WEBOS云平台及综合云OS选型》我们谈到用docker构建原生webos,cloudos，在《一种混合包管理和容器管理方案，及在tinycorelinux上安装containerd和openfaas》我们又一次提到这个观点，据说，用docker构建OS的还真不少。除了coreos还有moby。darch这种，Docker 在 DockerCon 2017大会上发布了一个自己的操作系统，宣称LinuxKit，就是kernel+busybox实现的一个微缩linux系统，其中直接安装了containerd和runc服务。其他服务全部都使用容器启动。

前面说到docker偏指docker官方发布的那个运行时版本和dockercli工具和规范，所以oci偏指containerd/runc这样的工具和规范。buildkit都支持它们作为前后端。Open Container Initiative(OCI)目前有2个标准：runtime-spec以及image-spec。OCI(当前)相当于规定了容器的images和runtime的协议，只要实现了OCI的容器就可以实现其兼容性和可移植性。那么它与我们要达到的效果"达到类似web端编辑迅速ci/cd的类似目的吗？不经过registry,就地构建并保存结果给openfaas运行"有什么关系呢？因为buildkit可以以containerd image store直接为后端(containerd image store，The containerd worker needs to be used)。形如：sudo buildctl build --frontend dockerfile.v0 --local context=./ --local dockerfile=./ --output type=image,name=docker.io/minlearn/dafsdf:latest（ctr --namespace=buildkit images ls，To change the containerd namespace, you need to change worker.containerd.namespace in /etc/buildkit/buildkitd.toml）

> buildkit能达到以上效果主要是它掩盖和封装了snapshot和image操作这些中间过程，提供了一个类似docker commit的操作，可以生成含上述oci文件的镜像包，类似直接在镜像上commit，要重新生成镜像，这就不得不谈到容器的snapshot。类似云主机快照的热备原理，ctr 也支持snapshot 操作，比如ctr contained 命令行工具 prepare 命令，基于指定的已提交态快照作为"父"创建一个新快照。还有commit，containerd是用snapshootter来完成这事的，Snapshotter is one of these plugins, which is used for storing extracted layers. During pulling an image, containerd extracts layers in the image and overlays them for preparing rootfs views called “snapshots” in the snapshotter. When containerd starts a container, it queries a snapshot to snapshotter and uses it as the container’s rootfs.基本上在本地重新生成镜像这个过程应该是这样的：1,Use the diff service to create a blob in content store with the committed snapshot 2,Create image config and distribution manifest https://github.com/opencontainers/image-spec in the content store that reference that layer blob 3,Set either the image config or manifest as a target descriptor for containerd image,它涉及到具体容器和镜像的配置文件定制都较繁琐（oci维护一个镜像和容器规范，你可以用ctr oci spec > /etc/containerd/cri-base.json获得这些基础配置。），因为新THE CONTAINER LAYERS和新THE IMAGE LAYERS层处理都很复杂。----所以我们得借助buildkit这样的工具。
>
> 除此之外，你还可以直接push到registry，sudo buildctl build --frontend dockerfile.v0 --local context=./ --local dockerfile=./ --output type=image,name=docker.io/minlearn/dafsdf:latest,push=true 还可以导出为一个OCI tarball buildctl build ... --output type=oci,dest=path/to/output.tar buildctl build ... --output type=oci > output.tar
>
> 对docker OCI runtime对它的理解可以解决前面一个没有解决的问题(ramdisk containerd下不能pivot root的问题，但是ramdisk chroot本身就是不受推荐的所以略过)。

甚至可以导出构建中使用的cache,导出缓存对运营registry的ci/cd服务商非常有用。因为可以加速构建过程。

docker mount方案

上面谈完了skip registry build的docker commit方案，对于mount方案，其实这个似乎更接近cloudbase那套(我们注意到cloudbash那个保存后提交也有一定时间延迟的，但是延迟很小，可能并不是在提交并building。)，这个就是层的概念。docker本身就支持mount主机上的目录并run,buildkit在构建过程中就直接支持Buildkit allows secrets files to be mounted as a part of the build process. These secrets are kept in-memory and not stored within the image:

RUN --mount=type=bind (the default mount type) This mount type allows binding directories (read-only) in the context or in an image to the build container. RUN --mount=type=cache This mount type allows the build container to cache directories for compilers and package managers.

但这些是buildkit是在构建时的mount，与运行时的mount要分开。对volume的定义是可以写在oci runtime-spec配置文件中的。

Volume Permission and Ownership

Volume permissions van be changed by configuring the ownership within the Dockerfile.You can, like Docker, mount volumes from the host to the container. The following mounts /home/Documents/images from the host to the container at /mnt/images with read and write permissions:

```
"mounts": [
    {
        "destination": "/mnt/images",
        "type": "bind",
        "source": "/home/Documents/images",
        "options": [
            "rbind",
            "rw"
        ]
```

```
    }
]

Remember that the UID:GID pair is relative to the user namespace that the user is going to run the container with.
```

这种方案下，只要类似openfaas gateway:8080/ui或cloudbase后台支持，可以完全分离docker构建/运行时所用的容器本身镜像和外来数据。在数据发生变动时，以手动提交方式或自动方式触发一次构建。

具体到将上述猜想做到openfaas+vscodelonline产品和将这个功能放进管理后端中，这实际上是处理openfaas后端如何调用openfaas-cli工具（通过json api）这类问题和https://code.visualstudio.com/docs/remote/containers-advanced所提到的那些问题。除此之外，新的加了edit按钮的后端，必须使得每个容器被depoly时，faas自动为其导出数据目录，不必写明在docker-composer.yml中最好。

在tc上安装buildkit.tcz，vscode.tcz，打通vscodeonline与openfaas模拟cloudbase打造碎片化编程开发部署环境

本文关键字：**rebuild kernel invalid magic number,failed to create diff tar stream: failed to get xattr for : operation not supported**

在《一种用buildkit打造免registry的local cd/ci工具,打通vscodeonline与openfaas模拟cloudbase打造碎片化编程开发部署环境的设想》中，我们介绍了方案和设想，本文将用测试说话，在tc11上实践上文谈到的内容：

> 这里提个上文遗漏的内容，为什么我们要说vscodeonline+openfaas这种环境是碎片化呢？因为这一切发生在云上，我们可以随时用碎片化时间断续编程，而且buildkit commit方案直接快速本地形成“修改->发布”的容器循环，而且如果你在云函数中用的是js这种带universal web/desktop app支持的云原生语言（说实话，只有js是云和web原生的而且实现全栈，因为它是内置语言自带环境browser,v8 nodejs，其它都是走通用化语言->支持webframework，更多关注仅后端的路子出来的，js像是一种编辑器和编辑器脚本语言更具DSL特性，）当然，这些都是后端微服务化，持续集成化，那么前端碎片化呢，全碎片化前端+碎片化后端才是降低程序规模以降低难度的标配，而Electron这样的产品（nodejs+chrome core形成的app front），可以把前端文件托管在后端，动态加载这些”前端资源“，达成wx小程序这样的快速原型，这也是“小”，“碎片”程序的意义表现之一。

buildkit.tcz+instant commit

首先，从下载buildkit最新版，它跟docker,openfaas,containerd一样，都是清一色的go binary，采用《一种混合包管理和容器管理方案，及在tinycorelinux上安装containerd和openfaas》同样的tcz构建方法，将下载到的v0.8.1的buildkit包的所有bin放到一个squashfs-root/usr/local/bin中(不用加chmod + x因为下载包里自带)，然后新建一个squashfs-root/usr/local/etc/init.d/，里面放二个chmod +x ./buildkit ./makedockerconfig:

这是makedockerconfig中的内容，buildkit+containerd代替了整个docker作构建工具和运行时(containerd不能build,ctr image build没有这个命令),但配置文件用的还是对接dockerhub的那一套。

```
read -p "enter your dockerhub username:" DOCKERHUB_USERNAME
read -p "enter your dockerhub personal access token:" DOCKERHUB_TOKEN
rm -rf ~/.docker/config.json
mkdir -p ~/.docker/
cat > ~/.docker/config.json << EOF
{
    "auths": {
        "https://index.docker.io/v1/": {
            "auth": "$(echo -n $DOCKERHUB_USERNAME:$DOCKERHUB_TOKENT | base64)"
        }
    }
}
EOF
cp -f ~/docker/config.json /var/lib/faasd/config.json
```

解释一下，所谓access token就是一种能代替密码，实现有限权限子账户的机制。dockerhub后台可以得到。由于tc11是没有base64的，这个工具在coreutils.tcz中。稍后建成的buildkit.tcz将依赖一条coreutils.tcz

这是buildkit中的内容，/usr/local/bin/buildkitd &，（你可以按上文添加额外参数设置为runc或containerd后端，默认为runc），为应用包在/init.d/写随着系统启动的启动文件，特别要注意，在命令后必要处加&，否则前台命令会block启动流程。在前文中，faasd和containerd都是这样处理的。

写好dep（依赖coreutils和containerd,attr.tcz:buildkit使用xattr相关命令）和md5.txt，打包成tcz,安装在tc11中，现在来测试一下，建立一个测试仓库并：sudo buildctl build --frontend dockerfile.v0 --local context=./ --local dockerfile=./ --output type=image,name=docker.io/minlearn/dafsdf:latest，出错了：

```
......
=> ERROR exporting to image 0.1s
error: failed to solve: rpc error: code = Unknown desc = mount callback failed on /tmp/containerd-mount267804283: mount callba
ck failed on /tmp/containerd-mount706848158: failed to write compressed diff: failed to create diff tar stream: failed to get
xattr for /tmp/containerd-mount267804283/bin: operation not supported
```

这是因为我在tc11中使用的是ext3，构建tc11用的config-5.4.3-tinycore64中并没有开启CONFIG_EXT3_FS,也没有开启CONFIG_EXT3_FS_XATTR，导致buildkit调用xattr相关命令时不成功。因此重新编译内核。注意要sudo make install，不能仅sudo make在arch/boot/x86/compressed下得到vmlinux，而要在install过后的/boot下得到vmlinuz。否则虽然编译成功，但kernel image运行不了，会提示invalid magic number

安装好kernel image,重启，问题解决。你就可以实现在local容器中免registry commit了。最后在命令中按是否需要上传到dockerhub，加个push=true一下。

当然这一切现在只是命令行方式进行，并没有上升到整合为openfaas 8080/ui那个后台的界面功能。但作为上文提到的方案2，接下来的vscode mount就好多了。

vscode.tcz + mount

我们下载的是cdr的code-server-3.8.0-amd64，按 《panel.sh：一个nginx+docker的云函和在线IDE面板,发明你自己的paas(2)》的路子将所有文件解压到squashfs-root的/usr/local/lib/中。然后新建squashfs-root/usr/local/bin,squashfs-root/usr/local/etc/init.d/并依次：

cd squashfs-root/usr/local/bin sudo ln -s ../lib/code-server-3.8.0/bin/code-server code-server sudo chmod +x ./code-server

cd squashfs-root/usr/local/etc/init.d/ sudo touch vscodeonline makevscodeconfig sudo chmod +x ./vscodeonline makevscodeconfig

makevscodeconfig里面放：

```
read -p "enter your desired access token(plain strings ok):" VSCODE_TOKEN
rm -rf ~/.config/code-server/config.yaml
mkdir -p ~/.config/code-server/
cat > ~/.config/code-server/config.yaml << EOF
bind-addr: 127.0.0.1:5000
auth: password
password: "$(echo $VSCODE_TOKEN)"
cert: false
EOF
mkdir -p /home/tc/.config/code-server/
cp -f /root/.config/code-server/config.yaml /home/tc/.config/code-server/config.yaml
```

然后是启动vscodeonline的：/usr/local/bin/code-server --config /root/.config/code-server/config.yaml &

直接打包，直接md5,没有dep引用。 --config ~/.config/code-server/config.yaml是需要的，因为code-server似乎存在一个bug，它启动的时候会在~.config下找配置文件，而不是~/.config少了一个/(多了一个/?没仔细看)，且它识别不了~，找不到会自动在8080启动vscode服务，导致发生异常，故强行指定/root/.config/。

然后你就可以按https://code.visualstudio.com/docs/remote/containers-advanced尝试vscode.tcz + mount方案了

未来我们将code-server中绑定的nodejs发行版本独立出来用独立tcz代替，这才符合tcz一个软件一个包的适当粒度划分。

---

buildkit commit方案已经可以即时将构建修改循环的时间压缩到很短了，但第一次拉取和构建镜像依然是从dockerhub，耗时巨大，我们在后来的文章中将探索用onedrive配合onemanager等程序，打造将OS发行的软件源和程序语言模块仓库，在线商店等，统统托管成平坦结构，放进od的方法。将OD发展为程序员专用，人手一套的私人registry网盘，配合ctr images offline tar import。这样，以后的时间将只耗费在我们本文提到的免registry commit的ci/cd过程中。

把DI当online packer用:利用installnet制作一个云装机packerpe（1）

本文关键字：**pebuilder,**云主机装机就用云镜像装机。直接在网络上组装生成目标系统，模拟群晖的**webassit,wget --spider to stdout not work**

前面《利用hashicorp packer把dbcolinux导出为虚拟机和docker格式》系列中我们讲到了packer，这是一个类虚拟机管理器的工具，它有一个iso模式，允许用户提供一个最原始的iso和一系列脚本。，然后在这个系统上运行脚本内命令行，完成格式硬盘，安装软件包，动态生成目标系统镜像内容。关机即得这个镜像。packer也常被用来在本地生成虚拟机镜像，然后上传导入到云主机供云主机用，然而，这有一个缺点，生成镜像通常很大，本地网络拉取一些特定区的镜像速度也不太好，最后，还得上传，那么，有没有一种在云主机上类packer，边拉取软件边生成目标镜像的工具呢？如果有，是否适用所有系统呢，比如windows不提供linux方式的kernel与rootfs分离，也不支持软件包仓库丰富系统，是不是也同样适用这个过程呢？

————— Ps:debianinstaller，是一种debian netboot system组成的类packer的环境，当可引导安装介质(安装光盘，安装U盘)启动后，可以选择进入文本向导模式或图形向导模式，然后完成如下操作 加载 kernel，initrd，加载额外的Udeb包,完成安装界面初始化环境; 如果启动参数包含 preseed文件，则会按照preseed文件内定义的规则自动执行,如果没有对应的规则,则返回交互界面; 交互界面会引导用户完成键盘，时区，主机名，网络，用户密码，分区等设置，存储在当前安全器环境中; 完成分区格式化等操作后，该磁盘会被挂载到 /target，安装器会调用debootstrap 在 /target 完成核心系统的构建; 安装器通过执行 chroot 操作进入以 /target 为根的系统中，完成软件源的更新，执行 tasksel，弹出可选软件菜单; 安装可选的软件包, 完成上述操作后,最后配置 grub, 将之前保存的全部设置应用，完成安装过程.
—————

可见，整个debian installer 类似packer，只不过这里bootstrap系统是debian（packer iso模式下的iso）。工具是Debian-installer等，脚本是preseed,provision过程是运行这个pe回放preseed。当被用于云主机时，使用debian installerb也可在其中完成识别硬盘格式化等操作，，实际上它开始是一个按preseed自动化的livecd，先启动进livecd，还原回放preseed时，可以在这里继续安装完整系统（preseed）完成即得到目标镜像，debian-installer不光用来生成debian派生系的镜像，对于windows,osx这种没有在线组装能力的系统而言，可直接在preseed早期就通过wget,tar,dd的管理道dd镜像完成安装目标系统 ——— 格盘，装系统，这其实是一种比服务商提代的镜像恢复方式和dd方式更符合云机装机的原生方式。《云主机上装windows iso》《virtio 0pe》这是我早期使用virtiope网络装机的文章，我们讲到过pebuilder，我们还谈到过群晖的webassit都是类比物，在后面一些文章，尤其是《云主机上装黑群，黑果》系列中我们都提到过moeclue的installnet.sh脚本，用它网络装机可在线完成。就相当pebuilder ，生成的pe相当packer ，

下面来完善这个脚本

注意！！！！！！！！：使用此脚本安装镜像默认会抹除你硬盘中的所有内容。请谨慎尝试。 注意！！！！！！！！：使用此脚本安装镜像默认会抹除你硬盘中的所有内容。请谨慎尝试。 注意！！！！！！！！：使用此脚本安装镜像默认会抹除你硬盘中的所有内容。请谨慎尝试。

前置

主要是把selectmirror精简为dd only，把一些全局变量置为局部过程变量，不用作为参数喂给脚本。

```
#!/bin/bash

## License: GPL,Written By MoeClub.org,moded by minlearn for dd purpose only

export tmpDDURL=''
export tmpMIRROR=''
export tmpINSTANTWITHOUTVNC='0'

export ipAddr=''
export ipMask=''
export ipGate=''

export UNKNOWHW='0'
export UNVER='6.4'

while [[ $# -ge 1 ]]; do
  case $1 in
    -dd|--ddurl)
      shift
      tmpDDURL="$1"
      shift
      ;;
    --mirror)
      shift
      tmpMIRROR="$1"
      shift
      ;;
    -i|--instantwithoutvnc)
      shift
      tmpINSTANTWITHOUTVNC="$1"
      shift
```

```
      ;;
    --ip-addr)
      shift
      ipAddr="$1"
      shift
      ;;
    --ip-mask)
      shift
      ipMask="$1"
      shift
      ;;
    --ip-gate)
      shift
      ipGate="$1"
      shift
      ;;
    *)
      if [[ "$1" != 'error' ]]; then echo -ne "\nInvaild option: '$1'\n\n"; fi
      echo -ne " Usage(args are self explained):\n\tbash $(basename $0)\t-dd/--ddurl\n\t\t\t\t--mirror\n\t\t\t\t-i/--instantwi
thoutvnc\n\t\t\t\t--ip-addr/--ip-gate/--ip-mask\n\t\t\t\n"
      exit 1;
      ;;
    esac
  done

[[ "$EUID" -ne '0' ]] && echo "Error:This script must be run as root!" && exit 1;

function CheckDependence(){
  FullDependence='0';
  for BIN_DEP in `echo "$1" |sed 's/,/\n/g'`
    do
      if [[ -n "$BIN_DEP" ]]; then
        Founded='0';
        for BIN_PATH in `echo "$PATH" |sed 's/:/\n/g'`
          do
            ls $BIN_PATH/$BIN_DEP >/dev/null 2>&1;
            if [ $? == '0' ]; then
              Founded='1';
              break;
            fi
          done
        if [ "$Founded" == '1' ]; then
          echo -en "$BIN_DEP";
        else
          FullDependence='1';
          echo -en "[\033[31mNot Install\033[0m]";
        fi
        echo -en "\t\t\t[\033[32mok\033[0m]\n";
      fi
  done
  if [ "$FullDependence" == '1' ]; then
    echo -ne "\n\033[31mError! \033[0mPlease use '\033[33mapt-get\033[0m' or '\033[33myum\033[0m' install it.\n\n\n"
    exit 1;
  fi
}

clear && echo -e "\n\n\n\n\n\n\n\n\n\n\033[36m# Check Dependence\033[0m\n"

CheckDependence wget,ar,awk,grep,sed,cut,cat,cpio,curl,gzip,find,dirname,basename;

function SelectMirror(){

  [ $# -ge 1 ] || exit 1
  AddonMirror=$(echo "$1" |sed 's/\ //g')

  declare -A MirrorBackup
  MirrorBackup=(["Debian0"]="" ["Debian1"]="http://httpredir.debian.org/debian")
  echo "$AddonMirror" |grep -q '^http://\|^https://\|^ftp://' && MirrorBackup[{"Debian"}0]="$AddonMirror"

  for mirror in $(echo "${!MirrorBackup[@]}" |sed 's/\ /\n/g' |sort -n |grep "^Debian")
    do
      CurMirror="${MirrorBackup[$mirror]}"
      [ -n "$CurMirror" ] || continue

      CheckPass1='0';
      DistsList="$(wget --no-check-certificate -qO- "$CurMirror/dists/" |grep -o 'href=.*/"' |cut -d'"' -f2 |sed '/-\|old\|Deb
ian\|experimental\|stable\|test\|sid\|devel/d' |grep '^[^/]' |sed -n '1h;1!H;$g;s/\n//g;s/\///\;/g;$p')";
      for DIST in `echo "$DistsList" |sed 's/;/\n/g'`
        do
          [[ "$DIST" == "jessie" ]] && CheckPass1='1' && break;
```

```
        done
      [[ "$CheckPass1" == '0' ]] && {
        echo -ne '\njessie not find in $CurMirror/dists/, Please check it! \n\n'
        bash $0 error;
        exit 1;
      }

      CheckPass2=0
      ImageFile="SUB_MIRROR/dists/jessie/main/installer-amd64/current/images/netboot/debian-installer/amd64/initrd.gz"
      [ -n "$ImageFile" ] || exit 1
      URL=`echo "$ImageFile" |sed "s#SUB_MIRROR#${CurMirror}#g"`
      wget --no-check-certificate --spider --timeout=3 -o /dev/null "$URL"
      [ $? -eq 0 ] && CheckPass2=1 && echo "$CurMirror" && break
    done

    [[ $CheckPass2 == 0 ]] && {
      echo -ne "\033[31mError! \033[0minitrd.gz not find in $CurMirror/jessie/main/installer-amd64/current/images/netboot/debi
an-installer/amd64/! \n";
      bash $0 error;
      exit 1;
    }
}

sleep 5s

echo -e "\n\n\033[36m# Select Mirror\033[0m\n"
LinuxMirror=$(SelectMirror "$tmpMIRROR")
echo -e "${LinuxMirror}"
```

缓存udebs和kernel,rootfs:

把udebs仓库也缓存下来放进pe，做成本地镜像。因为rootfs中有一个httpd，可以通过preseed启动，做成仓库镜像服务器。可以看到preseed中地址用了127.0.0.1，验证也去掉了By default the installer requires that repositories be authenticated using a known gpg key. This setting can be used to disable that authentication,a udeb就是一个方便在liveos中运行的简化的deb。

```
sleep 5s

echo -e "\n\033[36m# All prerequisites Done,Begin processing [DD URL:$tmpDDURL,Instant without vnc:$tmpINSTANTWITHOUTVNC]\033[
0m\n"

[[ -d /tmp/boot ]] && rm -rf /tmp/boot;
mkdir -p /tmp/boot/usr/bin;
cd /tmp/boot;

LIBC6_SUPPORT='pool/main/g/glibc/libc6_2.19-18+deb8u10_amd64.deb'
WGETSSL_SUPPORT=''
declare -A HTTPD_SUPPORT
HTTPD_SUPPORT=(
["webfs1"]="pool/main/w/webfs/webfs_1.21+ds1-10_amd64.deb"
["webfs2"]="pool/main/g/gnutls28/libgnutls-deb0-28_3.3.8-6+deb8u7_amd64.deb"
["webfs3"]="pool/main/p/p11-kit/libp11-kit0_0.20.7-1_amd64.deb"
["webfs4"]="pool/main/libt/libtasn1-6/libtasn1-6_4.2-3+deb8u3_amd64.deb"
["webfs5"]="pool/main/n/nettle/libnettle4_2.7.1-5+deb8u2_amd64.deb"
["webfs6"]="pool/main/n/nettle/libhogweed2_2.7.1-5+deb8u2_amd64.deb"
["webfs7"]="pool/main/g/gmp/libgmp10_6.0.0+dfsg-6_amd64.deb"
["webfs8"]="pool/main/libf/libffi/libffi6_3.1-2+deb8u1_amd64.deb"
["webfs9"]="pool/main/m/mime-support/mime-support_3.58_all.deb"
)
DDPROGRESS_SUPPORT=''
UNZIP=''

[[ -n "$LIBC6_SUPPORT" ]] && {
  echo -ne 'Add libc6 support(binary-amd64/Package)...'

  wget --no-check-certificate -qO ${LIBC6_SUPPORT##*/} $LinuxMirror/$LIBC6_SUPPORT; \
  ar x ${LIBC6_SUPPORT##*/} data.tar.gz; tar xzf data.tar.gz; \
  rm -rf data.tar.gz ${LIBC6_SUPPORT##*/}

  [[ ! -f  /tmp/boot/lib/x86_64-linux-gnu/libc.so.6 ]] && echo 'Error! LIBC6_SUPPORT.' && exit 1 || sleep 3s && echo -en "[\03
3[32mok\033[0m]\n" ;
  # [[ $? -eq '0' ]] && echo -ne 'Success! \n\n'
}


if [[ -n "$tmpDDURL" ]]; then
  echo "$tmpDDURL" |grep -q '^http://\|^ftp://\|^https://';
  [[ $? -ne '0' ]] && echo 'No valid URL in the DD argument,Only support http://, ftp:// and https:// !' && exit 1;
```

```
  curl -Is "$tmpDDURL" | grep -q 'gzip';
  [[ $? -ne '0' ]] && echo 'not tar or gunzip file !' && exit 1 || UNZIP='0';
else
  echo 'Please input vaild image URL! ';
  exit 1;
fi

echo "$tmpDDURL" |grep -q '^https://'
[[ $? -eq '0' ]] && {

  [[ -n "$WGETSSL_SUPPORT" ]] && {
    echo -ne 'Add ssl support(binary-amd64/Package)...'

    wget --no-check-certificate -qO- $LinuxMirror/$WGETSSL_SUPPORT |tar -x
    mv -f /tmp/boot/usr/bin/wget /tmp/boot/usr/bin/

    [[ ! -f  /tmp/boot/usr/bin/wget2 ]] && echo 'Error! WGETSSL_SUPPORT.' && exit 1 || sleep 3s && echo -en "[\033[32mok\033[0
m]\n" ;
    # sed -i 's/wget\ -qO-/\/usr\/bin\/wget2\ --no-check-certificate\ --retry-connrefused\ --tries=7\ --continue\ -qO-/g' /tmp
/boot/preseed.cfg
    # [[ $? -eq '0' ]] && echo -ne 'Success! \n\n'
  }
    # || {
    # echo -ne 'Not ssl support package! \n\n';
    # exit 1;
    # }
}

#[[ -n "$HTTPD_SUPPORT" ]] && {
  echo -ne 'Add httpd support(webfs binary-amd64 Package)...'

  for pkg in $(echo "${!HTTPD_SUPPORT[@]}" |sed 's/\ /\n/g' |sort -n |grep "^webfs")
    do
      CurPkg="${HTTPD_SUPPORT[$pkg]}"
      [ -n "$CurPkg" ] || continue

      wget --no-check-certificate -qO ${CurPkg##*/} $LinuxMirror/$CurPkg; \
      ar x ${CurPkg##*/} data.tar.xz; xz -d data.tar.xz; tar xf data.tar; \
      rm -rf data.tar ${CurPkg##*/}
    done

    [[ ! -f  /tmp/boot/usr/bin/webfsd ]] && echo 'Error! HTTPD_SUPPORT.' && exit 1 || sleep 3s && echo -en "[\033[32mok\033[0m
]\n" ;
    # [[ $? -eq '0' ]] && echo -ne 'Success! \n\n'
#}


echo -e "Downloading kernel : ${LinuxMirror}/...../debian-installer/amd64/initrd.gz"

IncFirmware='0'

wget --no-check-certificate -qO '/boot/initrd.img' "${LinuxMirror}/dists/jessie/main/installer-amd64/current/images/netboot/de
bian-installer/amd64/initrd.gz"
[[ $? -ne '0' ]] && echo -ne "\033[31mError! \033[0mDownload 'initrd.img' for \033[33mdebian\033[0m failed! \n" && exit 1
wget --no-check-certificate -qO '/boot/vmlinuz' "${LinuxMirror}/dists/jessie/main/installer-amd64/current/images/netboot/debia
n-installer/amd64/linux"
[[ $? -ne '0' ]] && echo -ne "\033[31mError! \033[0mDownload 'vmlinuz' for \033[33mdebian\033[0m failed! \n" && exit 1
MirrorHost="$(echo "$LinuxMirror" |awk -F'://|/' '{print $2}')";
MirrorFolder="$(echo "$LinuxMirror" |awk -F''${MirrorHost}'' '{print $2}')";

if [[ "$IncFirmware" == '1' ]]; then
  wget --no-check-certificate -qO '/boot/firmware.cpio.gz' "http://cdimage.debian.org/cdimage/unofficial/non-free/firmware/jes
sie/current/firmware.cpio.gz"
  [[ $? -ne '0' ]] && echo -ne "\033[31mError! \033[0mDownload 'firmware' for \033[33mdebian\033[0m failed! \n" && exit 1
fi

vKernel_udeb=$(wget --no-check-certificate -qO- "http://jessieMirror/dists/$DIST/main/installer-amd64/current/images/udeb.list
" |grep '^acpi-modules' |head -n1 |grep -o '[0-9]\{1,2\}.[0-9]\{1,2\}.[0-9]\{1,2\}-[0-9]\{1,2\}' |head -n1)
[[ -z "vKernel_udeb" ]] && vKernel_udeb="3.16.0-6"

echo -e "Downloading debs : ${LinuxMirror}/pool/....."

IncUdebrepo='1'

if [[ "$IncUdebrepo" == '1' ]]; then

  mkdir -p /tmp/boot/var/log/debian;

  udeburl=".*pool\/main\(.*\)udeb.*"
  wget --no-check-certificate -qO- "$LinuxMirror/dists/jessie/main/debian-installer/binary-amd64/Packages.gz" |gunzip -dc|sed
```

```
"/$udeburl/!d"|sed "s/Filename: //g"|while read line
  do
    path=${line%/*}
    mkdir -p /tmp/boot/var/log/debian/$path
    file=${line##*/}
    wget --no-check-certificate -qO /tmp/boot/var/log/debian/$path/$file $LinuxMirror/$line
  done

  mkdir -p /tmp/boot/var/log/debian/dists/jessie/main/binary-amd64/
  mkdir -p /tmp/boot/var/log/debian/dists/jessie/main/debian-installer/binary-amd64/
  mkdir -p /tmp/boot/var/log/debian/dists/jessie/main/installer-amd64/current/images/

  wget --no-check-certificate -qO /tmp/boot/var/log/debian/dists/jessie/Release $LinuxMirror/dists/jessie/Release
  wget --no-check-certificate -qO /tmp/boot/var/log/debian/dists/jessie/main/binary-amd64/Release $LinuxMirror/dists/jessie/ma
in/binary-amd64/Release
  wget --no-check-certificate -qO /tmp/boot/var/log/debian/dists/jessie/main/debian-installer/binary-amd64/Release $LinuxMirro
r/dists/jessie/main/debian-installer/binary-amd64/Release,同路径下还有一个Packages.gz
  wget --no-check-certificate -qO /tmp/boot/var/log/debian/dists/jessie/main/installer-amd64/current/images/udeb.list $LinuxMi
rror/dists/jessie/main/installer-amd64/current/images/udeb.list

  chmod -R 0644 /tmp/boot/var/log/debian/
fi
```

————

脚本版权归原作者所有。还可以像群晖web assit一样,https://wiki.debian.org/DebianInstaller/WebInstaller。preseed还可用于网络和qemu启动

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

把DI当online packer用:利用installnet制作一个云装机packerpe（2）

本文关键字：自建**udeb**站镜像。**local repository debian installer preseed,Preseed install from local repository,start http server in a preseed file,debian**中用**NC+SHELL**实现的一个**web**服务,浏览器装机,**Preseed failed with exit 2,d-i preseed/early_command string stuck hang,linux grep -o** 只匹配字串

继把Debianinstaller当online packer用文章1，本文继续讲解生成packerpe。

给前面的脚本增加和镜像格式判断，统一wget和dd progress

增强主要是为了pipecmdstr处整合，稍后会看到(因此也顺便支持了onelist,oneindex这种网盘外链作为ddurl地址，其实质是302跳转体)。这段替换前文的cd /tmp/boot;到echo -e "Downloading kernel : ${LinuxMirror}/...../debian-installer/amd64/initrd.gz"间的内容：

```
LIBC6_SUPPORT='pool/main/g/glibc/libc6_2.28-10_amd64.deb'
declare -A SHARED_SUPPORT
SHARED_SUPPORT=(
["webfs1"]="pool/main/g/gnutls28/libgnutls30_3.6.7-4+deb10u3_amd64.deb"
["webfs2"]="pool/main/p/p11-kit/libp11-kit0_0.23.15-2_amd64.deb"
["webfs3"]="pool/main/libt/libtasn1-6/libtasn1-6_4.13-3_amd64.deb"
["webfs4"]="pool/main/n/nettle/libnettle6_3.4.1-1_amd64.deb"
["webfs5"]="pool/main/n/nettle/libhogweed4_3.4.1-1_amd64.deb"
["webfs6"]="pool/main/g/gmp/libgmp10_6.1.2+dfsg-4_amd64.deb"

["webfs7"]="pool/main/libf/libffi/libffi6_3.2.1-9_amd64.deb"
["webfs8"]="pool/main/m/mime-support/mime-support_3.62_all.deb"
["webfs9"]="pool/main/libu/libunistring/libunistring2_0.9.10-1_amd64.deb"

["wgetssl7"]="pool/main/libi/libidn2/libidn2-0_2.0.5-1+deb10u1_amd64.deb"
["wgetssl8"]="pool/main/libp/libpsl/libpsl5_0.20.2-2_amd64.deb"
["wgetssl9"]="pool/main/p/pcre2/libpcre2-8-0_10.32-5_amd64.deb"
["wgetssl10"]="pool/main/u/util-linux/libuuid1_2.33.1-0.1_amd64.deb"
["wgetssl11"]="pool/main/z/zlib/zlib1g_1.2.11.dfsg-1_amd64.deb"
)

HTTPD_SUPPORT='pool/main/w/webfs/webfs_1.21+ds1-12_amd64.deb'
WGETSSL_SUPPORT='pool/main/w/wget/wget_1.20.1-1.1_amd64.deb'
DDPROGRESS_SUPPORT='pool/main/b/busybox/busybox_1.30.1-4_amd64.deb'
UNZIP=''
DDURL=''

[[ -n "$LIBC6_SUPPORT" ]] && {
  echo -ne 'Add libc6 support(binary-amd64/Package)...'

  wget --no-check-certificate -qO ${LIBC6_SUPPORT##*/} $LinuxMirror/$LIBC6_SUPPORT; \
  ar x ${LIBC6_SUPPORT##*/} data.tar.xz; xz -d data.tar.xz; tar xf data.tar; \
  rm -rf data.tar ${LIBC6_SUPPORT##*/}

  [[ ! -f  /tmp/boot/lib/x86_64-linux-gnu/libc.so.6 ]] && echo 'Error! LIBC6_SUPPORT.' && exit 1 || sleep 3s && echo -en "[\03
3[32mok\033[0m]\n" ;
  # [[ $? -eq '0' ]] && echo -ne 'Success! \n\n'
}

[[ -n "$HTTPD_SUPPORT" ]] && {
  echo -ne 'Add httpd and httpd shared support(binary-amd64/Package)...'

  wget --no-check-certificate -qO ${HTTPD_SUPPORT##*/} $LinuxMirror/$HTTPD_SUPPORT; \
  ar x ${HTTPD_SUPPORT##*/} data.tar.xz; xz -d data.tar.xz; tar xf data.tar; \
  rm -rf data.tar ${HTTPD_SUPPORT##*/}

  for pkg in $(echo "${!SHARED_SUPPORT[@]}" |sed 's/\ /\n/g' |sort -n |grep "^webfs")
    do
      CurPkg="${SHARED_SUPPORT[$pkg]}"
      [ -n "$CurPkg" ] || continue

      wget --no-check-certificate -qO ${CurPkg##*/} $LinuxMirror/$CurPkg; \
      ar x ${CurPkg##*/} data.tar.xz; xz -d data.tar.xz; tar xf data.tar; \
      rm -rf data.tar ${CurPkg##*/}
    done

  [[ ! -f  /tmp/boot/usr/bin/webfsd ]] && echo 'Error! HTTPD_SUPPORT.' && exit 1 || sleep 3s && echo -en "[\033[32mok\033[0m]\
n" ;
  # [[ $? -eq '0' ]] && echo -ne 'Success! \n\n'
}
```

```
if [[ -n "$tmpDDURL" ]]; then
  echo -ne 'setting final ddurl and prepare unzip mode...'
  echo "$tmpDDURL" |grep -q '^http://\|^ftp://\|^https://';
  [[ $? -ne '0' ]] && echo 'No valid URL in the DD argument,Only support http://, ftp:// and https:// !' && exit 1;

  IMGHEADER="$(curl -Is "$tmpDDURL")";
  IMGTYPE="$(echo "$IMGHEADER" | grep -E -o 'raw|qcow2|gzip|x-gzip|301|302')" || IMGTYPE='0';
# [[ "$IMGTYPE" -ne '0' ]] && echo 'not a raw,tar,gunzip or 301/302 ref file, exit ... !' && exit 1 || {

    [[ "$IMGTYPE" == 'raw' ]] && UNZIP='0' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m raw \033[0m]\n";
    [[ "$IMGTYPE" == 'qcow2' ]] && UNZIP='0' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m raw \033[0m]\n";
    [[ "$IMGTYPE" == 'gzip' ]] && UNZIP='1' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m gzip \033[0m]\n";
    [[ "$IMGTYPE" == 'x-gzip' ]] && UNZIP='1' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m x-gzip \033[0m]\n";
    [[ "$IMGTYPE" == 'gunzip' ]] && UNZIP='2' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m gunzip \033[0m]\n";

    [[ "$IMGTYPE" == '302' ]] && IMGHEADERCHECKPASS2="$(echo "$IMGHEADER" |grep 'Location: http'|sed 's/Location: //g')" && IM
GHEADERCHECKPASS2=${IMGHEADERCHECKPASS2%$'\r'} && IMGHEADERCHECKPASS3="$(curl -Is "$IMGHEADERCHECKPASS2" |grep 'content-locati
on: http'|sed 's/content-location: //g')" && IMGHEADERCHECKPASS3=${IMGHEADERCHECKPASS3%$'\r'} && IMGTYPECHECKPASS2="$(curl -Is
 "$IMGHEADERCHECKPASS3" | grep -E -o 'raw|qcow2|gzip|x-gzip|gunzip')" && {
      [[ "$IMGTYPECHECKPASS2" == 'raw' ]] && UNZIP='0' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m raw \033[0m]\n"
;
      [[ "$IMGTYPECHECKPASS2" == 'qcow2' ]] && UNZIP='0' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m raw \033[0m]\
n";
      [[ "$IMGTYPECHECKPASS2" == 'gzip' ]] && UNZIP='1' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m gzip \033[0m]\
n";
      [[ "$IMGTYPECHECKPASS2" == 'x-gzip' ]] && UNZIP='1' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m x-gzip \033[
0m]\n";
      [[ "$IMGTYPECHECKPASS2" == 'gunzip' ]] && UNZIP='2' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m gunzip \033[
0m]\n";
    }

    [[ "$IMGTYPE" == '0' ]] && echo 'not a raw,tar,gunzip or 301/302 ref file, exit ... !' && exit 1;
# }

else
  echo 'Please input vaild image URL! ';
  exit 1;
fi

#echo "$DDURL" |grep -q '^https://'
#[[ $? -eq '0' ]] && {

  [[ -n "$WGETSSL_SUPPORT" ]] && {
    echo -ne 'always Add wgetssl and wgetssl support(binary-amd64/Package)...'

    wget --no-check-certificate -qO ${WGETSSL_SUPPORT##*/} $LinuxMirror/$WGETSSL_SUPPORT; \
    ar x ${WGETSSL_SUPPORT##*/} data.tar.xz; xz -d data.tar.xz; tar xf data.tar; \
    rm -rf data.tar ${WGETSSL_SUPPORT##*/}; mv -f /tmp/boot/usr/bin/wget /tmp/boot/usr/bin/wget2

    for pkg in $(echo "${!SHARED_SUPPORT[@]}" |sed 's/\ /\n/g' |sort -n |grep "^wgetssl")
      do
        CurPkg="${SHARED_SUPPORT[$pkg]}"
        [ -n "$CurPkg" ] || continue

        wget --no-check-certificate -qO ${CurPkg##*/} $LinuxMirror/$CurPkg; \
        ar x ${CurPkg##*/} data.tar.xz; xz -d data.tar.xz; tar xf data.tar; \
        rm -rf data.tar ${CurPkg##*/}
      done

    [[ ! -f  /tmp/boot/usr/bin/wget2 ]] && echo 'Error! WGETSSL_SUPPORT.' && exit 1 || sleep 3s && echo -en "[\033[32mok\033[0
m]\n" ;
    # sed -i 's/wget\ -qO-/\/usr\/bin\/wget2\ --no-check-certificate\ --retry-connrefused\ --tries=7\ --continue\ -qO-/g' /tmp
/boot/preseed.cfg
    # [[ $? -eq '0' ]] && echo -ne 'Success! \n\n'
  }
    # || {
    # echo -ne 'Not wgetssl support package! \n\n';
    # exit 1;
    # }
#}

[[ -n "$DDPROGRESS_SUPPORT" ]] && {
  echo -ne 'Add ddprogress support(binary-amd64/Package)...'

  wget --no-check-certificate -qO ${DDPROGRESS_SUPPORT##*/} $LinuxMirror/$DDPROGRESS_SUPPORT; \
  ar x ${DDPROGRESS_SUPPORT##*/} data.tar.xz; xz -d data.tar.xz; tar xf data.tar; \
  rm -rf data.tar ${DDPROGRESS_SUPPORT##*/}

  [[ ! -f  /tmp/boot/bin/busybox ]] && echo 'Error! DDPROGRESS_SUPPORT.' && exit 1 || sleep 3s && echo -en "[\033[32mok\033[0m
```

```
]\n" ;
  # [[ $? -eq '0' ]] && echo -ne 'Success! \n\n'
}
```

如果你发现启动不了文件系统出现制作文件系统时出现的那些错误，那就是可能glibc低了。注意到dd status progress我并未在接下的preseed中用到。

整合精简几大件,the net, the loader, the preseed file

下面是脚本的继续,准备net设置，grub loader和preseed file供打包：

```
sleep 5s

echo -e "\n\033[36m# Preparing others\033[0m\n"

echo -e "the net"

setNet='0'
interface=''

[ -n "$ipAddr" ] && [ -n "$ipMask" ] && [ -n "$ipGate" ] && setNet='1';
[[ -n "$tmpWORD" ]] && myPASSWORD="$(openssl passwd -1 "$tmpWORD")";
[[ -z "$myPASSWORD" ]] && myPASSWORD='$1$4BJZaD0A$y1QykUnJ6mXprENfwpseH0';

if [[ -n "$interface" ]]; then
  IFETH="$interface"
else
  IFETH="auto"
fi

[[ "$setNet" == '1' ]] && {
  IPv4="$ipAddr";
  MASK="$ipMask";
  GATE="$ipGate";
} || {
  DEFAULTNET="$(ip route show |grep -o 'default via [0-9]\{1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}.*' |head -n1 |sed 's/p
roto.*\|onlink.*//g' |awk '{print $NF}')";
  [[ -n "$DEFAULTNET" ]] && IPSUB="$(ip addr |grep ''${DEFAULTNET}'' |grep 'global' |grep 'brd' |head -n1 |grep -o '[0-9]\{1,3
\}.[0-9]\{1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}/[0-9]\{1,2\}')";
  IPv4="$(echo -n "$IPSUB" |cut -d'/' -f1)";
  NETSUB="$(echo -n "$IPSUB" |grep -o '/[0-9]\{1,2\}')";
  GATE="$(ip route show |grep -o 'default via [0-9]\{1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}' |head -n1 |grep -o '[0-9]\{
1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}')";
  [[ -n "$NETSUB" ]] && MASK="$(echo -n '128.0.0.0/1,192.0.0.0/2,224.0.0.0/3,240.0.0.0/4,248.0.0.0/5,252.0.0.0/6,254.0.0.0/7,2
55.0.0.0/8,255.128.0.0/9,255.192.0.0/10,255.224.0.0/11,255.240.0.0/12,255.248.0.0/13,255.252.0.0/14,255.254.0.0/15,255.255.0.0
/16,255.255.128.0/17,255.255.192.0/18,255.255.224.0/19,255.255.240.0/20,255.255.248.0/21,255.255.252.0/22,255.255.254.0/23,255
.255.255.0/24,255.255.255.128/25,255.255.255.192/26,255.255.255.224/27,255.255.255.240/28,255.255.255.248/29,255.255.255.252/3
0,255.255.255.254/31,255.255.255.255/32' |grep -o '[0-9]\{1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}'${NETSUB}'' |cut -d'/'
-f1)";
}

[[ -n "$GATE" ]] && [[ -n "$MASK" ]] && [[ -n "$IPv4" ]] || {
  echo "Not found \`ip command\`, It will use \`route command\`."

  ipNum() {
    local IFS='.';
    read ip1 ip2 ip3 ip4 <<<"$1";
    echo $((ip1*(1<<24)+ip2*(1<<16)+ip3*(1<<8)+ip4));
  }

  SelectMax(){
  ii=0;
  for IPITEM in `route -n |awk -v OUT=$1 '{print $OUT}' |grep '[0-9]\{1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}'`
    do
      NumTMP="$(ipNum $IPITEM)";
      eval "arrayNum[$ii]='$NumTMP,$IPITEM'";
      ii=$[$ii+1];
    done
  echo ${arrayNum[@]} |sed 's/\s/\n/g' |sort -n -k 1 -t ',' |tail -n1 |cut -d',' -f2;
  }

  [[ -z $IPv4 ]] && IPv4="$(ifconfig |grep 'Bcast' |head -n1 |grep -o '[0-9]\{1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}.[0-9]\{1,3\}' |h
ead -n1)";
  [[ -z $GATE ]] && GATE="$(SelectMax 2)";
  [[ -z $MASK ]] && MASK="$(SelectMax 3)";

  [[ -n "$GATE" ]] && [[ -n "$MASK" ]] && [[ -n "$IPv4" ]] || {
```

```
    echo "Error! Not configure network. ";
    exit 1;
  }
}

[[ "$setNet" != '1' ]] && [[ -f '/etc/network/interfaces' ]] && {
  [[ -z "$(sed -n '/iface.*inet static/p' /etc/network/interfaces)" ]] && AutoNet='1' || AutoNet='0';
  [[ -d /etc/network/interfaces.d ]] && {
    ICFGN="$(find /etc/network/interfaces.d -name '*.cfg' |wc -l)" || ICFGN='0';
    [[ "$ICFGN" -ne '0' ]] && {
      for NetCFG in `ls -1 /etc/network/interfaces.d/*.cfg`
        do
          [[ -z "$(cat $NetCFG | sed -n '/iface.*inet static/p')" ]] && AutoNet='1' || AutoNet='0';
          [[ "$AutoNet" -eq '0' ]] && break;
        done
    }
  }
}

[[ "$setNet" != '1' ]] && [[ -d '/etc/sysconfig/network-scripts' ]] && {
  ICFGN="$(find /etc/sysconfig/network-scripts -name 'ifcfg-*' |grep -v 'lo'|wc -l)" || ICFGN='0';
  [[ "$ICFGN" -ne '0' ]] && {
    for NetCFG in `ls -1 /etc/sysconfig/network-scripts/ifcfg-* |grep -v 'lo$' |grep -v ':[0-9]\{1,\}'`
      do
        [[ -n "$(cat $NetCFG | sed -n '/BOOTPROTO.*[dD][hH][cC][pP]/p')" ]] && AutoNet='1' || {
          AutoNet='0' && . $NetCFG;
          [[ -n $NETMASK ]] && MASK="$NETMASK";
          [[ -n $GATEWAY ]] && GATE="$GATEWAY";
        }
        [[ "$AutoNet" -eq '0' ]] && break;
      done
  }
}

echo -e "the loader"

LoaderMode='0'
setInterfaceName='0'
setIPv6='0'

if [[ "$LoaderMode" == "0" ]]; then
  [[ -f '/boot/grub/grub.cfg' ]] && GRUBVER='0' && GRUBDIR='/boot/grub' && GRUBFILE='grub.cfg';
  [[ -z "$GRUBDIR" ]] && [[ -f '/boot/grub2/grub.cfg' ]] && GRUBVER='0' && GRUBDIR='/boot/grub2' && GRUBFILE='grub.cfg';
  [[ -z "$GRUBDIR" ]] && [[ -f '/boot/grub/grub.conf' ]] && GRUBVER='1' && GRUBDIR='/boot/grub' && GRUBFILE='grub.conf';
  [ -z "$GRUBDIR" -o -z "$GRUBFILE" ] && echo -ne "Error! \nNot Found grub.\n" && exit 1;
else
  tmpINSTANTWITHOUTVNC='0'
fi

if [[ "$LoaderMode" == "0" ]]; then
  [[ ! -f $GRUBDIR/$GRUBFILE ]] && echo "Error! Not Found $GRUBFILE. " && exit 1;

  [[ ! -f $GRUBDIR/$GRUBFILE.old ]] && [[ -f $GRUBDIR/$GRUBFILE.bak ]] && mv -f $GRUBDIR/$GRUBFILE.bak $GRUBDIR/$GRUBFILE.old;
  mv -f $GRUBDIR/$GRUBFILE $GRUBDIR/$GRUBFILE.bak;
  [[ -f $GRUBDIR/$GRUBFILE.old ]] && cat $GRUBDIR/$GRUBFILE.old >$GRUBDIR/$GRUBFILE || cat $GRUBDIR/$GRUBFILE.bak >$GRUBDIR/$G
RUBFILE;
else
  GRUBVER='2'
fi


[[ "$GRUBVER" == '0' ]] && {
  READGRUB='/tmp/grub.read'
  cat $GRUBDIR/$GRUBFILE |sed -n '1h;1!H;$g;s/\n/%%%%%%%/g;$p' |grep -om 1 'menuentry\ [^{]*{[^}]*}%%%%%%%' |sed 's/%%%%%%%/\n
/g' >$READGRUB
  LoadNum="$(cat $READGRUB |grep -c 'menuentry ')"
  if [[ "$LoadNum" -eq '1' ]]; then
    cat $READGRUB |sed '/^$/d' >/tmp/grub.new;
  elif [[ "$LoadNum" -gt '1' ]]; then
    CFG0="$(awk '/menuentry /{print NR}' $READGRUB|head -n 1)";
    CFG2="$(awk '/menuentry /{print NR}' $READGRUB|head -n 2 |tail -n 1)";
    CFG1="";
    for tmpCFG in `awk '/}/{print NR}' $READGRUB`
      do
        [ "$tmpCFG" -gt "$CFG0" -a "$tmpCFG" -lt "$CFG2" ] && CFG1="$tmpCFG";
      done
    [[ -z "$CFG1" ]] && {
      echo "Error! read $GRUBFILE. ";
      exit 1;
    }
```

```
    sed -n "$CFG0,$CFG1"p $READGRUB >/tmp/grub.new;
    [[ -f /tmp/grub.new ]] && [[ "$(grep -c '{' /tmp/grub.new)" -eq "$(grep -c '}' /tmp/grub.new)" ]] || {
      echo -ne "\033[31mError! \033[0mNot configure $GRUBFILE. \n";
      exit 1;
    }
  fi
  [ ! -f /tmp/grub.new ] && echo "Error! $GRUBFILE. " && exit 1;
  sed -i "/menuentry.*/c\menuentry\ \'Packer PE \[debian\ jessie\ amd64\]\'\ --class debian\ --class\ gnu-linux\ --class\ gnu\
 --class\ os\ \{" /tmp/grub.new
  sed -i "/echo.*Loading/d" /tmp/grub.new;
  INSERTGRUB="$(awk '/menuentry /{print NR}' $GRUBDIR/$GRUBFILE|head -n 1)"
}

[[ "$GRUBVER" == '1' ]] && {
  CFG0="$(awk '/title[\ ]|title[\t]/{print NR}' $GRUBDIR/$GRUBFILE|head -n 1)";
  CFG1="$(awk '/title[\ ]|title[\t]/{print NR}' $GRUBDIR/$GRUBFILE|head -n 2 |tail -n 1)";
  [[ -n $CFG0 ]] && [ -z $CFG1 -o $CFG1 == $CFG0 ] && sed -n "$CFG0,$"p $GRUBDIR/$GRUBFILE >/tmp/grub.new;
  [[ -n $CFG0 ]] && [ -z $CFG1 -o $CFG1 != $CFG0 ] && sed -n "$CFG0,$[$CFG1-1]"p $GRUBDIR/$GRUBFILE >/tmp/grub.new;
  [[ ! -f /tmp/grub.new ]] && echo "Error! configure append $GRUBFILE. " && exit 1;
  sed -i "/title.*/c\title\ \'DebianNetboot \[jessie\ amd64\]\'" /tmp/grub.new;
  sed -i '/^#/d' /tmp/grub.new;
  INSERTGRUB="$(awk '/title[\ ]|title[\t]/{print NR}' $GRUBDIR/$GRUBFILE|head -n 1)"
}

if [[ "$LoaderMode" == "0" ]]; then
  [[ -n "$(grep 'linux.*/\|kernel.*/' /tmp/grub.new |awk '{print $2}' |tail -n 1 |grep '^/boot/')" ]] && Type='InBoot' || Type
='NoBoot';

  LinuxKernel="$(grep 'linux.*/\|kernel.*/' /tmp/grub.new |awk '{print $1}' |head -n 1)";
  [[ -z "$LinuxKernel" ]] && echo "Error! read grub config! " && exit 1;
  LinuxIMG="$(grep 'initrd.*/' /tmp/grub.new |awk '{print $1}' |tail -n 1)";
  [ -z "$LinuxIMG" ] && sed -i "/$LinuxKernel.*\//a\\\tinitrd\ \/" /tmp/grub.new && LinuxIMG='initrd';

  if [[ "$setInterfaceName" == "1" ]]; then
    Add_OPTION="net.ifnames=0 biosdevname=0";
  else
    Add_OPTION="";
  fi

  if [[ "$setIPv6" == "1" ]]; then
    Add_OPTION="$Add_OPTION ipv6.disable=1";
  fi

  BOOT_OPTION="auto=true $Add_OPTION hostname=debian domain= -- quiet"

  [[ "$Type" == 'InBoot' ]] && {
    sed -i "/$LinuxKernel.*\//c\\\t$LinuxKernel\\t\/boot\/vmlinuz $BOOT_OPTION" /tmp/grub.new;
    sed -i "/$LinuxIMG.*\//c\\\t$LinuxIMG\\t\/boot\/initrd.img" /tmp/grub.new;
  }

  [[ "$Type" == 'NoBoot' ]] && {
    sed -i "/$LinuxKernel.*\//c\\\t$LinuxKernel\\t\/vmlinuz $BOOT_OPTION" /tmp/grub.new;
    sed -i "/$LinuxIMG.*\//c\\\t$LinuxIMG\\t\/initrd.img" /tmp/grub.new;
  }

  sed -i '$a\\n' /tmp/grub.new;
fi

[[ "$tmpINSTANTWITHOUTVNC" == '0' ]] && {
  GRUBPATCH='0';

  if [[ "$LoaderMode" == "0" ]]; then
  [ -f '/etc/network/interfaces' -o -d '/etc/sysconfig/network-scripts' ] || {
    echo "Error, Not found interfaces config.";
    exit 1;
  }

  sed -i ''${INSERTGRUB}'i\\n' $GRUBDIR/$GRUBFILE;
  sed -i ''${INSERTGRUB}'r /tmp/grub.new' $GRUBDIR/$GRUBFILE;
  [[ -f  $GRUBDIR/grubenv ]] && sed -i 's/saved_entry/#saved_entry/g' $GRUBDIR/grubenv;
  fi

  cd /tmp/boot;
  COMPTYPE="gzip";
  CompDected='0'
  for ListCOMP in `echo -en 'gzip\nlzma\nxz'`
    do
      if [[ "$COMPTYPE" == "$ListCOMP" ]]; then
        CompDected='1'
```

```
        if [[ "$COMPTYPE" == 'gzip' ]]; then
          NewIMG="initrd.img.gz"
        else
          NewIMG="initrd.img.$COMPTYPE"
        fi
        mv -f "/boot/initrd.img" "/tmp/$NewIMG"
        break;
      fi
    done
  [[ "$CompDected" != '1' ]] && echo "Detect compressed type not support." && exit 1;
  [[ "$COMPTYPE" == 'lzma' ]] && UNCOMP='xz --format=lzma --decompress';
  [[ "$COMPTYPE" == 'xz' ]] && UNCOMP='xz --decompress';
  [[ "$COMPTYPE" == 'gzip' ]] && UNCOMP='gzip -d';

  $UNCOMP < /tmp/$NewIMG | cpio --extract --verbose --make-directories --no-absolute-filenames >>/dev/null 2>&1

  echo -e "the preseed"

  [[ "$UNZIP" == '0' ]] && PIPECMDSTR='wget2 --no-check-certificate -qO- '$DDURL' |dd of=$(list-devices disk |head -n1)';
  [[ "$UNZIP" == '1' ]] && PIPECMDSTR='wget2 --no-check-certificate -qO- '$DDURL' |tar zOx |dd of=$(list-devices disk |head -n
1)';
  [[ "$UNZIP" == '2' ]] && PIPECMDSTR='wget2 --no-check-certificate -qO- '$DDURL' |gzip -dc |dd of=$(list-devices disk |head -
n1)';

cat >/tmp/boot/preseed.cfg<<EOF
d-i preseed/early_command string anna-install

d-i debian-installer/locale string en_US
d-i console-setup/layoutcode string us

d-i keyboard-configuration/xkb-keymap string us

d-i hw-detect/load_firmware boolean true

d-i netcfg/choose_interface select $IFETH
d-i netcfg/disable_autoconfig boolean true
d-i netcfg/dhcp_failed note
d-i netcfg/dhcp_options select Configure network manually
# d-i netcfg/get_ipaddress string $ipAddr
d-i netcfg/get_ipaddress string $IPv4
d-i netcfg/get_netmask string $MASK
d-i netcfg/get_gateway string $GATE
d-i netcfg/get_nameservers string 8.8.8.8
d-i netcfg/no_default_route boolean true
d-i netcfg/confirm_static boolean true

d-i mirror/country string manual
#d-i mirror/http/hostname string $IPv4
d-i mirror/http/hostname string httpredir.debian.org
d-i mirror/http/directory string /debian
d-i mirror/http/proxy string

# webfsd -i $IPv4 -p 80 -r /var/log/ -l /var/log/httpd.log

d-i apt-setup/services-select multiselect
d-i debian-installer/allow_unauthenticated boolean true

d-i passwd/root-login boolean ture
d-i passwd/make-user boolean false
d-i passwd/root-password-crypted password $myPASSWORD
d-i user-setup/allow-password-weak boolean true
d-i user-setup/encrypt-home boolean false

d-i clock-setup/utc boolean true
d-i time/zone string US/Eastern
d-i clock-setup/ntp boolean true

d-i partman/early_command string $PIPECMDSTR; /sbin/reboot
EOF

  [[ "$LoaderMode" != "0" ]] && AutoNet='1'

  [[ "$setNet" == '0' ]] && [[ "$AutoNet" == '1' ]] && {
    sed -i '/netcfg\/disable_autoconfig/d' /tmp/boot/preseed.cfg
    sed -i '/netcfg\/dhcp_options/d' /tmp/boot/preseed.cfg
    sed -i '/netcfg\/get_.*/d' /tmp/boot/preseed.cfg
    sed -i '/netcfg\/confirm_static/d' /tmp/boot/preseed.cfg
  }

  #[[ "$GRUBPATCH" == '1' ]] && {
```

```
  #  sed -i 's/^d-i\ grub-installer\/bootdev\ string\ default//g' /tmp/boot/preseed.cfg
  #}
  #[[ "$GRUBPATCH" == '0' ]] && {
  #  sed -i 's/debconf-set\ grub-installer\/bootdev.*\"\;//g' /tmp/boot/preseed.cfg
  #}

  sed -i '/user-setup\/allow-password-weak/d' /tmp/boot/preseed.cfg
  sed -i '/user-setup\/encrypt-home/d' /tmp/boot/preseed.cfg
  #sed -i '/pkgsel\/update-policy/d' /tmp/boot/preseed.cfg
  #sed -i 's/umount\ \/media.*true\;\ //g' /tmp/boot/preseed.cfg

  [[ -f '/boot/firmware.cpio.gz' ]] && {
    gzip -d < /boot/firmware.cpio.gz | cpio --extract --verbose --make-directories --no-absolute-filenames >>/dev/null 2>&1
  }
```

注意上面的脚本中输出至preseed file至EOF止的内容必须顶格写，Ks.cf是installment.sh用来用同样ubuntu支持preseed支持centos的方法。已删，在文件中，我注释了原来使用内置的webfsd服务器作镜像的方案，因为preseed early command string中欠入webfsd启动命令（放anna install前），会导致PE启动时early command运行时，umount时前台却在加载webfsd卡住，需要ctrlc才能继续，但实际上webfsd已成功运行。

所以我将注释webfsd的逻辑放在preseed中某位置，这个位置是它应该启动时的时机。实际上early command string中都不比它合适。未来想办法让它以正确方式起作用。

打包

```
  sleep 5s

  echo -e "\n\033[36m# Packaging \033[0m\n"

  find . | cpio -H newc --create --quiet | gzip -9 > /boot/initrd.img;
  [[ "$tmpINSTANTWITHOUTVNC" == '0' ]] &&  echo "finished, auto reboot after 30s...(if needed, you can press ctrl c to interru
pt to bak the /boot/initrd.img, then manually reboot to continue)";sleep 30s

  rm -rf /tmp/boot;

}

[[ "$tmpINSTANTWITHOUTVNC" == '1' ]] && {
  sed -i '$i\\n' $GRUBDIR/$GRUBFILE
  sed -i '$r /tmp/grub.new' $GRUBDIR/$GRUBFILE
  echo -e "\n\033[33m\033[04mIt will reboot! \nPlease connect VNC! \nSelect\033[0m\033[32m Packer PE [debian jessie amd64] \03
3[33m\033[4mto install system.\033[04m\n\n\033[31m\033[04mThere is some information for you.\nDO NOT CLOSE THE WINDOW! \033[0m
\n"
  echo -e "\033[35mIPv4\t\tNETMASK\t\tGATEWAY\033[0m"
  echo -e "\033[36m\033[04m$IPv4\033[0m\t\033[36m\033[04m$MASK\033[0m\t\033[36m\033[04m$GATE\033[0m\n\n"

  read -n 1 -p "Press Enter to reboot..." INP
  [[ "$INP" != '' ]] && echo -ne '\b \n\n';
}

chown root:root $GRUBDIR/$GRUBFILE
chmod 444 $GRUBDIR/$GRUBFILE


if [[ "$LoaderMode" == "0" ]]; then
  sleep 3 && reboot >/dev/null 2>&1
else
  rm -rf "$HOME/loader"
  mkdir -p "$HOME/loader"
  cp -rf "/boot/initrd.img" "$HOME/loader/initrd.img"
  cp -rf "/boot/vmlinuz" "$HOME/loader/vmlinuz"
  [[ -f "/boot/initrd.img" ]] && rm -rf "/boot/initrd.img"
  [[ -f "/boot/vmlinuz" ]] && rm -rf "/boot/vmlinuz"
  echo && ls -AR1 "$HOME/loader"
fi
```

脚本结束。

——

这个脚本还有其它很多需要增强的地方，比如dd progress显示。还比如改成了生成双分区。保留packerpe到第一分区。整盘其实可以preseed/partman成二个分区，一个分区200M放生成的packerpe自身（未来，还可以在这个packerpe集成dbcolinux或集成linuxbpot,coreboot，clover等），

还比如，扩展镜像到整个磁盘空间，实际上windows通过ntfs fuse是实现了的。对于linux磁盘，要处理镜像分区表适配到目标整个磁盘。这不光适用于dd方案。也适用于将磁盘打包成dd镜像（脚本中改变pipecmdstr，镜像扩容缩容）。然后nc 监听，供其它机器下载其镜像或直接dd。这是这个脚本的反向功能。

还比如，利用webfsd的cgi改dd地址。也可用grub的auto url变换dd地址。Ddurl参数保留到下一次。且读取位置不再在preseed中。

但是我不想做了，以后继续吧。这些天做windows server 2019 core的dd镜像也较累了，用了跟做osx云镜像差不多的方法：在linuxkvm中套虚拟机。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

利用onedrive加packerpebuilder实现本地网络统一装机

本文关键字：**bash line string to array, sed** 按匹配次数替换，**sed**替换指定次数的某字符串,**bash**变量替换，即时赋新值，**bash** 数组下标变量

在《利用installnet制作一个云装机packerpe》1,2中我们谈到了packerpebuilder.sh用于云主机装机的用处，它的原理是产生一个基于d-i的pe，然后这个pe会使用当初封装的dd url去网络下载镜像并dd到硬件，在《利用od做站和nas》我们谈到了其与终端作pcmate,mobilemate的那些方面（这个系列只讲了做站，即配合client pc as page server mate的方面---想象一下，在客户端我们可以用servering pages，只是拥有了网盘backend的page server app才真正使远程和本地成为一对mates，产生本地远程合一的mate app，未来我们还会讲配合client pc作离线下载，网盘转存等mateable的方面），今天，我们将用onedrve结合packerpebuilder实现本地也能像云主机一样装机，使远程成为本地装机app，实际上这个思路自packerpebuilder一开始就有了，只是一直没有找到合适能用的网盘。直到对onedrive的直链有了研究之后，才有了本文。

不废话了

前置改动

把tmpmirror也消除了。调整了一些注释，如整合checkdeps和selectmirror为prepare prerequisites，selectmirror经过重构变成select1stvalidmirrorfrom3():

```
function Select1stValidMirrorFrom3(){

  [ $# -ge 1 ] || exit 1

  declare -A MirrorTocheck
  MirrorTocheck=(["Debian0"]="" ["Debian1"]="" ["Debian2"]="")

  echo "$1" |sed 's/\ //g' |grep -q '^http://\|^https://\|^ftp://' && MirrorTocheck[Debian0]=$(echo "$1" |sed 's/\ //g');
  echo "$2" |sed 's/\ //g' |grep -q '^http://\|^https://\|^ftp://' && MirrorTocheck[Debian1]=$(echo "$2" |sed 's/\ //g');
  echo "$3" |sed 's/\ //g' |grep -q '^http://\|^https://\|^ftp://' && MirrorTocheck[Debian2]=$(echo "$3" |sed 's/\ //g');

  for mirror in $(echo "${!MirrorTocheck[@]}" |sed 's/\ /\n/g' |sort -n |grep "^Debian")
    do
      CurMirror="${MirrorTocheck[$mirror]}"

      [ -n "$CurMirror" ] || continue

      # CheckPass1='0';
      # DistsList="$(wget --no-check-certificate -qO- "$CurMirror/dists/" |grep -o 'href=.*/"' |cut -d'"' -f2 |sed '/-\|old\|D
ebian\|experimental\|stable\|test\|sid\|devel/d' |grep '^[^/]' |sed -n '1h;1!H;$g;s/\n//g;s/\//\;/g;$p')";
      # for DIST in `echo "$DistsList" |sed 's/;/\n/g'`
        # do
          # [[ "$DIST" == "jessie" ]] && CheckPass1='1' && break;
        # done
      # [[ "$CheckPass1" == '0' ]] && {
        # echo -ne '\njessie not find in $CurMirror/dists/, Please check it! \n\n'
        # bash $0 error;
        # exit 1;
      # }

      CheckPass2=0
      ImageFile="SUB_MIRROR/dists/jessie/main/installer-amd64/current/images/netboot/debian-installer/amd64/initrd.gz"
      [ -n "$ImageFile" ] || exit 1
      URL=`echo "$ImageFile" |sed "s#SUB_MIRROR#${CurMirror}#g"`
      wget --no-check-certificate --spider --timeout=3 -o /dev/null "$URL"
      [ $? -eq 0 ] && CheckPass2=1 && echo "$CurMirror" && break
    done

    [[ $CheckPass2 == 0 ]] && {
      echo -ne "\033[31mError! \033[0minitrd.gz not find in $CurMirror/jessie/main/installer-amd64/current/images/netboot/debi
an-installer/amd64/! \n";
      bash $0 error;
      exit 1;
    }
}

MIRROR=$(Select1stValidMirrorFrom3 'http://httpredir.debian.org/debian' 'http://www.shalol.com/cn/d/debian' 'http://http://arc
hive.debian.org/debian')
[ -n "$MIRROR" ] && echo -en "Select Mirror ......:" && echo -en "[\033[32m ${MIRROR} \033[0m]\n" || exit 1
```

主体改动

整合prepare parepare dist files包括downloading basic kernel and rootfs files（将它提前，逻辑更合理。）和downloading repo pkgs files，，以及接下来的PrepareDDessentials(其原来内部下载deb的逻辑整合到与下载full udeb一起)，，并将它们都变成可复用的函数和函数调用buildrepo()和PrepareDDessentials().

prepare parepare dist files与prepare others是并列的：前三者是大资源文件，后三者是小参数文件，将二者中间延时变量变成2s，各内部延时3s（内部还去掉了细节方面，肯定情况下的一些echo输出，改为直接exit 1，改为由主要的几句话来echo，界面输出更清），共5s

wget要调用ssl client才能tls certificate已完善，buildrepo()更强大，支持sed "s/(+|~)/-/g"处理链接中的+号和~号(tcb上的onemanager不支持这类特殊符号)，和更强大更逻辑清楚的拉取安装deb pkgs支持:

```
IncPkgrepo='1'

declare -A OPTPKGS
OPTPKGS=(
  ["libc1"]="pool/main/g/glibc/libc6_2.28-10_amd64.deb"
  ["fmtlibc"]="xz"
  ["binlibc"]=""

  ["common1"]="pool/main/g/gnutls28/libgnutls30_3.6.7-4+deb10u3_amd64.deb"
  ["common2"]="pool/main/p/p11-kit/libp11-kit0_0.23.15-2_amd64.deb"
  ["common3"]="pool/main/libt/libtasn1-6/libtasn1-6_4.13-3_amd64.deb"
  ["common4"]="pool/main/n/nettle/libnettle6_3.4.1-1_amd64.deb"
  ["common5"]="pool/main/n/nettle/libhogweed4_3.4.1-1_amd64.deb"
  ["common6"]="pool/main/g/gmp/libgmp10_6.1.2+dfsg-4_amd64.deb"
  ["fmtcommon"]="xz"
  ["bincommon"]=""

  ["busybox1"]="pool/main/b/busybox/busybox_1.30.1-4_amd64.deb"
  ["fmtbusybox"]="xz"
  ["binbusybox"]="bin/busybox"

  ["wgetssl1"]="pool/main/libi/libidn2/libidn2-0_2.0.5-1+deb10u1_amd64.deb"
  ["wgetssl2"]="pool/main/libp/libpsl/libpsl5_0.20.2-2_amd64.deb"
  ["wgetssl3"]="pool/main/p/pcre2/libpcre2-8-0_10.32-5_amd64.deb"
  ["wgetssl4"]="pool/main/u/util-linux/libuuid1_2.33.1-0.1_amd64.deb"
  ["wgetssl5"]="pool/main/z/zlib/zlib1g_1.2.11.dfsg-1_amd64.deb"
  ["wgetssl6"]="pool/main/o/openssl/libssl1.0.0_1.0.1t-1+deb8u8_amd64.deb"
  ["wgetssl7"]="pool/main/o/openssl/openssl_1.0.1t-1+deb8u8_amd64.deb"
  ["wgetssl8"]="pool/main/w/wget/wget_1.20.1-1.1_amd64.deb"
  ["fmtwgetssl"]="xz"
  ["binwgetssl"]="usr/bin/wget"

  ["webfs1"]="pool/main/libf/libffi/libffi6_3.2.1-9_amd64.deb"
  ["webfs2"]="pool/main/m/mime-support/mime-support_3.62_all.deb"
  ["webfs3"]="pool/main/libu/libunistring/libunistring2_0.9.10-1_amd64.deb"
  ["webfs4"]="pool/main/w/webfs/webfs_1.21-ds1-12_amd64.deb"
  ["fmtwebfs"]="xz"
  ["binwebfs"]=""
)

function buildrepo(){

  if [[ "$IncPkgrepo" == '1' ]]; then

    echo -e "Downloading full udebs pkg files..... [\033[32m ${MIRROR}/dists/jessie/main/debian-installer/binary-amd64/Package
s.gz \033[0m]\n"

    repodir='/tmp/boot/var/log/debian'
    mkdir -p $repodir

    udeburl=".*pool\/main\(.*\)udeb.*"
    wget --no-check-certificate -qO- "$MIRROR/dists/jessie/main/debian-installer/binary-amd64/Packages.gz" |gunzip -dc|sed "/$
udeburl/!d"|sed "s/Filename: //g"|while read line
    do
      path=${line%/*}
      mkdir -p $repodir/$path
      file=${line##*/}
      wget --no-check-certificate -qO $repodir/$path/$(echo $file|sed "s/\(+\|~\)/-/g") $MIRROR/$line
    done

    mkdir -p $repodir/dists/jessie/main/binary-amd64/
    mkdir -p $repodir/dists/jessie/main/debian-installer/binary-amd64/
    mkdir -p $repodir/dists/jessie/main/installer-amd64/current/images/

    wget --no-check-certificate -qO $repodir/dists/jessie/Release $MIRROR/dists/jessie/Release
    wget --no-check-certificate -qO $repodir/dists/jessie/main/binary-amd64/Release $MIRROR/dists/jessie/main/binary-amd64/Rel
ease
    wget --no-check-certificate -qO $repodir/dists/jessie/main/debian-installer/binary-amd64/Release $MIRROR/dists/jessie/main
```

```
/debian-installer/binary-amd64/Release
    wget --no-check-certificate -qO $repodir/dists/jessie/main/debian-installer/binary-amd64/Packages.gz $MIRROR/dists/jessie/
main/debian-installer/binary-amd64/Packages.gz; \
    orisize=$(cat $repodir/dists/jessie/main/debian-installer/binary-amd64/Packages.gz | wc -c); \
    orimd5=$(md5sum $repodir/dists/jessie/main/debian-installer/binary-amd64/Packages.gz| awk '{ print $1 }'); \
    orisha1=$(sha1sum $repodir/dists/jessie/main/debian-installer/binary-amd64/Packages.gz| awk '{ print $1 }'); \
    orisha256=$(sha256sum $repodir/dists/jessie/main/debian-installer/binary-amd64/Packages.gz| awk '{ print $1 }'); \
    gunzip -c $repodir/dists/jessie/main/debian-installer/binary-amd64/Packages.gz > $repodir/dists/jessie/main/debian-install
er/binary-amd64/Packages; \
    rm -rf $repodir/dists/jessie/main/debian-installer/binary-amd64/Packages.gz; \
    sed -i "s/\(+\|~\)/-/g" $repodir/dists/jessie/main/debian-installer/binary-amd64/Packages; \
    gzip -c $repodir/dists/jessie/main/debian-installer/binary-amd64/Packages > $repodir/dists/jessie/main/debian-installer/bi
nary-amd64/Packages.gz; \
    rm -rf $repodir/dists/jessie/main/debian-installer/binary-amd64/Packages; \
    cursize=$(cat $repodir/dists/jessie/main/debian-installer/binary-amd64/Packages.gz | wc -c); \
    curmd5=$(md5sum $repodir/dists/jessie/main/debian-installer/binary-amd64/Packages.gz| awk '{ print $1 }'); \
    cursha1=$(sha1sum $repodir/dists/jessie/main/debian-installer/binary-amd64/Packages.gz| awk '{ print $1 }'); \
    cursha256=$(sha256sum $repodir/dists/jessie/main/debian-installer/binary-amd64/Packages.gz| awk '{ print $1 }')

    toreplace="main\/debian-installer\/binary-amd64\/Packages.gz"
    linenoarray=($(grep -n $toreplace $repodir/dists/jessie/Release |cut -f1 -d:))

    sed -i ${linenoarray[0]}s/$orimd5/$curmd5/ $repodir/dists/jessie/Release
    sed -i ${linenoarray[0]}s/$orisize/$cursize/ $repodir/dists/jessie/Release
    sed -i ${linenoarray[1]}s/$orisha1/$cursha1/ $repodir/dists/jessie/Release
    sed -i ${linenoarray[1]}s/$orisize/$cursize/ $repodir/dists/jessie/Release
    sed -i ${linenoarray[2]}s/$orisha256/$cursha256/ $repodir/dists/jessie/Release
    sed -i ${linenoarray[2]}s/$orisize/$cursize/ $repodir/dists/jessie/Release

    wget --no-check-certificate -qO $repodir/dists/jessie/main/installer-amd64/current/images/udeb.list $MIRROR/dists/jessie/m
ain/installer-amd64/current/images/udeb.list

    chmod -R 0644 $repodir/
  fi

  echo -e "Downloading optional deb pkg files..... [\033[32m ${MIRROR}/dists/jessie/main/binary-amd64/Packages.gz \033[0m]\n"
;

  for pkg in `echo "$1" |sed 's/,/\n/g'`
    do

      [[ -n "${OPTPKGS[$pkg"1"]}" ]] && {
        for subpkg in $(echo "${!OPTPKGS[@]}" |sed 's/\ /\n/g' |sort -n |grep "^$pkg")
          do
            cursubpkgfile="${OPTPKGS[$subpkg]}"
            [ -n "$cursubpkgfile" ] || continue

            cursubpkgfilepath=${cursubpkgfile%/*}
            mkdir -p $repodir/$cursubpkgfilepath
            cursubpkgfilename=${cursubpkgfile##*/}
            cursubpkgfilename2=$(echo $cursubpkgfilename|sed "s/\(+\|~\)/-/g")

            wget --no-check-certificate -qO $repodir/$cursubpkgfilepath/$cursubpkgfilename2 $MIRROR/$cursubpkgfile; \
            [[ "${OPTPKGS["fmt"$pkg]}" == "tar" ]] && ar x $repodir/$cursubpkgfilepath/$cursubpkgfilename2 data.tar.gz && tar
xzf data.tar.gz && rm -rf data.tar.gz
            [[ "${OPTPKGS["fmt"$pkg]}" == "xz" ]] && ar x $repodir/$cursubpkgfilepath/$cursubpkgfilename2 data.tar.xz && xz -d
 data.tar.xz && tar xf data.tar && rm -rf data.tar

          done
            [[ -n "${OPTPKGS["bin"$pkg]}" ]] && mv -f /tmp/boot/${OPTPKGS["bin"$pkg]} /tmp/boot/${OPTPKGS["bin"$pkg]}2
            # [[ ! -f  /tmp/boot/${OPTPKGS["bin"$pkg]}2 ]] && echo 'Error! $1 SUPPORT ERROR.' && exit 1;
      }

    done

}
buildrepo libc,common,busybox,wgetssl;
```

PrepareDDessentials()也更强大，支持sharepoint和office365个人的302跳转风格，强化《利用installnet制作一个云装机packerpe》2中关于仅支持office365style相关方面功能 --- 其实sharepointstyle和office365 style也可自动公判断，但是我不想折腾了。

```
UNZIP=''
DDURL=''
OFFICE365STYLE='0'
SHAREPOINTSTYLE='1'

function PrepareDDessentials(){

  if [[ -n "$tmpDDURL" ]]; then
```

```
    echo "$tmpDDURL" |grep -q '^http://\|^ftp://\|^https://';
    [[ $? -ne '0' ]] && echo 'No valid URL in the DD argument,Only support http://, ftp:// and https:// !' && exit 1;

    IMGHEADER="$(curl -Is "$tmpDDURL")";
    IMGTYPE="$(echo "$IMGHEADER" | grep -E -o '200|302')" || IMGTYPE='0';

    # [[ "$IMGTYPE" -ne '0' ]] && echo 'not a raw,tar,gunzip or 301/302 ref file, exit ... !' && exit 1 || {

      [[ "$IMGTYPE" == '200' ]] && IMGHEADERCHECKPASS2="$(echo "$IMGHEADER" |grep -E -o 'raw|qcow2|gzip|x-gzip')" && {
        [[ "$IMGTYPECHECKPASS2" == 'raw' ]] && UNZIP='0' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m raw \033[0m]"
;
        [[ "$IMGTYPECHECKPASS2" == 'qcow2' ]] && UNZIP='0' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m raw \033[0m
]";
        [[ "$IMGTYPECHECKPASS2" == 'gzip' ]] && UNZIP='1' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m gzip \033[0m
]";
        [[ "$IMGTYPECHECKPASS2" == 'x-gzip' ]] && UNZIP='1' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m x-gzip \03
3[0m]";
        [[ "$IMGTYPECHECKPASS2" == 'gunzip' ]] && UNZIP='2' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m gunzip \03
3[0m]";
      }

      [[ "$IMGTYPE" == '302' && "$OFFICE365STYLE" == '1' ]] && { \

        IMGHEADERCHECKPASS2="$(echo "$IMGHEADER" |grep 'Location: http'|sed 's/Location: //g')" && IMGHEADERCHECKPASS2=${IMGHE
ADERCHECKPASS2%$'\r'} && \
        IMGHEADERCHECKPASS3="$(curl -Is "$IMGHEADERCHECKPASS2" |grep 'content-location: http'|sed 's/content-location: //g')"
&& IMGHEADERCHECKPASS3=${IMGHEADERCHECKPASS3%$'\r'} && \

        IMGTYPECHECKPASS2="$(curl -Is "$IMGHEADERCHECKPASS3" | grep -E -o 'raw|qcow2|gzip|x-gzip|gunzip')" && {
          [[ "$IMGTYPECHECKPASS2" == 'raw' ]] && UNZIP='0' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m raw \033[0m
]";
          [[ "$IMGTYPECHECKPASS2" == 'qcow2' ]] && UNZIP='0' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m raw \033[
0m]";
          [[ "$IMGTYPECHECKPASS2" == 'gzip' ]] && UNZIP='1' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m gzip \033[
0m]";
          [[ "$IMGTYPECHECKPASS2" == 'x-gzip' ]] && UNZIP='1' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m x-gzip \
033[0m]";
          [[ "$IMGTYPECHECKPASS2" == 'gunzip' ]] && UNZIP='2' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m gunzip \
033[0m]";
        }
      }



      [[ "$IMGTYPE" == '302' && "$SHAREPOINTSTYLE" == '1' ]] && { \

        IMGHEADERCHECKPASS2="$(echo "$IMGHEADER" |grep 'Location: http'|sed 's/Location: //g')" && IMGHEADERCHECKPASS2=${IMGHE
ADERCHECKPASS2%$'\r'} && \

        IMGTYPECHECKPASS2="$(curl -Is "$IMGHEADERCHECKPASS2" | grep -E -o 'raw|qcow2|gzip|x-gzip|gunzip')" && {
          [[ "$IMGTYPECHECKPASS2" == 'raw' ]] && UNZIP='0' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m raw \033[0m
]";
          [[ "$IMGTYPECHECKPASS2" == 'qcow2' ]] && UNZIP='0' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m raw \033[
0m]";
          [[ "$IMGTYPECHECKPASS2" == 'gzip' ]] && UNZIP='1' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m gzip \033[
0m]";
          [[ "$IMGTYPECHECKPASS2" == 'x-gzip' ]] && UNZIP='1' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m x-gzip \
033[0m]";
          [[ "$IMGTYPECHECKPASS2" == 'gunzip' ]] && UNZIP='2' && DDURL="$tmpDDURL" && sleep 3s && echo -en "[\033[32m gunzip \
033[0m]";
        }
      }



      [[ "$UNZIP" == '' ]] && echo 'didnt got a unzip mode, exit ... !' && exit 1;
      [[ "$DDURL" == '' ]] && echo 'didnt got a ddurl, exit ... !' && exit 1;

      [[ "$IMGTYPE" == '0' ]] && echo 'not a raw,tar,gunzip or 301/302 ref file, exit ... !' && exit 1;
    # }

  else
    echo 'Please input vaild image URL! ';
    exit 1;
  fi

}
echo -e 'prepare DDessentials ......';
PrepareDDessentials;
sleep 3s
```

使用方法

中途提示备份,会给你30s上传/tmp/boot/var/log/debian仓库到onedrive或其它服务器创建镜像，。

```
   [[ "$tmpINSTANTWITHOUTVNC" == '0' ]] &&  echo "finished, auto reboot after 30s...(if needed, you can press ctrl c to interru
pt to bak the repodir:$repodir, then manually reboot to continue)";sleep 30s
```

preseed中的mirrorhost换成你的od上传地址。

然后就然后了。。。

未来，我们要往packerpebuilder中集成不死booter。这样装机永远都不会因为抹了第一个硬盘哭了。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一键**pebuilder**，实现云主机在线装**deepin20beta**

本文关键字：**deepin**云主机版本，在**onedrive**里装机

国产系统deepin现在已经很精美了，国内很多软硬厂商在慢慢联手uos，。手机os华为方面2020.9月份会发布第一个可用版鸿蒙，这也是一个"uniform os"，最近听说美控台积电又要给华为断供了，但其实技术和生态也是俗物，只要钱慢慢到位，时间都是第二位的，也就没有不可克服的问题，毕竟华为芯片都能自研了，这种从1到2的事情其实反而好办。

我们《minlearnprogrammingv2:选型与实践》作为《minlearnprogramming:理论与实践》的替代与延续，也力在打造一个统一os：mateos,matecloud os，最初定位于从tinycorelinux开始深度定制，最近决定转为基于deepin深度定制，这是一个集成云booter,面向云应用，由客服一体的netdisk backend apps组成的生态。---- 这种集成虚拟机和linux的云booter和kernel exec，真正让rootfs as os container content,使得app和os,subos一个性质都是rootfs。从最开始和源头就自带强烈的虚拟化特性，它类似qubeos,不过qubeos的用处在于利用上述设施制造安全系统，它把联网代码转置入非安全域所以用到vtd，据说还有集成quubeos的librem安全笔记本。而我们的目的在于容器和app container定义和建设。这样deepin os被转换成pure rootfs，放置在rootfs的位置。一个虚拟机上可以运行多份大小os。---- 况且为了这种os下的开发，我们还准备了可视开发和网盘内devops支持（这是后话）

好了不废话了。先在云主机上装上deepinos20beta，经实践，deepin20的界面（非特效）在gd5446下也非常流畅，它的界面效果也是四角圆白色风格，类web还有darkmode。所以放到云主机也是很和谐的效果。

首先我们要创建可用的镜像，还要增强一下《利用onedrive加packerpebuilder实现本地网络统一装机》以来的在od里装机的方案：

创建云主机镜像

利用的是云主机上装黑果系列文章中制造镜像的方法(而非packer结合oss,cos)，在pd中装deepin，开kvm，再启动qemu制造镜像：

```
qemu-img create -f raw deepin20 20G

qemu-system-x86_64 -enable-kvm \
 -machine pc-i440fx-2.8 \
 -cpu Penryn,kvm=off,vendor=GenuineIntel \
 -m 5120 \
 -device cirrus-vga,bus=pci.0,addr=0x2 \
 -usb -device usb-kbd -device usb-mouse \
 -device ide-drive,bus=ide.0,drive=MacDVD \
 -drive id=MacDVD,if=none,snapshot=on,media=cdrom,file=./20.iso \
 -device ide-drive,bus=ide.1,drive=MacDVD1 \
 -drive id=MacDVD1,if=none,snapshot=on,media=cdrom,file=./3.iso \
 -device virtio-blk-pci,bus=pci.0,addr=0x5,drive=MacHDD \
 -drive id=MacHDD,if=none,cache=writeback,format=raw,file=./deepin20.raw \
 -device virtio-net-pci,bus=pci.0,addr=0x3,mac='52:54:00:c9:18:27',netdev=MacNET \
 -netdev bridge,id=MacNET,br=virbr0,"helper=/usr/lib/qemu/qemu-bridge-helper" \
 -boot order=dc,menu=on
```

手动全盘挂载到/不要自动全盘否则小于64g安装程序不让你过去。 以上脚本方案当然也适合windows，我用类似的方案制造了一个winsrvcore2019.gz,对于windows你还需要一个iso virtio-win-0.1.171.iso，上面的3.iso就是。如果是for deepin可以去掉。当然windows需要定制，比如防止ping localhost出现ipv6，disablectlaltdel登录等等。

这样装好的镜像在还原到云主机上或在pd kvm/qemu中运行，都是可以正常联网的。

强化**pebuilder.sh**

加了一个export tmpGENMIRRORBAK='0'全局变量，即-g开关，直接pebuilder.sh -dd '你的镜像文件地址'，使用-g开关可以生成本地镜像仓库以供上传到你自己的onedrive用

加入了当Downloading basic kernel and rootfs files时：

```
if [[ "$tmpGENMIRRORBAK" == '1' ]]; then
  bakdir='/tmp/boot/var/log/debian/dists/jessie/main/installer-amd64/current/images'
  mkdir -p "$bakdir/netboot/debian-installer/amd64"
  cp "/boot/initrd.img" "${bakdir}/netboot/debian-installer/amd64/initrd.gz"
  cp "/boot/vmlinuz" "${bakdir}/netboot/debian-installer/amd64/linux"
  wget --no-check-certificate -qO $bakdir/udeb.list $MIRROR/dists/jessie/main/installer-amd64/current/images/udeb.list
fi
```

接下来的downloading full udeb pkg当然也用这个开关控制。

新的自动化prepare ddessentials：

```
UNZIP=''
IMGSIZE=''

function PrepareDDessentials(){

  if [[ -n "$tmpDDURL" ]]; then
    echo "$tmpDDURL" |grep -q '^http://\|^ftp://\|^https://';
    [[ $? -ne '0' ]] && echo 'No valid URL in the DD argument,Only support http://, ftp:// and https:// !' && exit 1;

    IMGHEADERCHECK="$(curl -Is "$tmpDDURL")";
    IMGTYPECHECK="$(echo "$IMGHEADERCHECK"|grep -E -o '200|302'|head -n 1)" || IMGTYPECHECK='0';

    #directurl style,no addon headcheckpass,1 addon typecheckpass to the final
    [[ "$IMGTYPECHECK" == '200' ]] && \
    IMGTYPECHECKPASS_DRT="$(echo "$IMGHEADERCHECK"|grep -E -o 'raw|qcow2|gzip|x-gzip'|head -n 1)" && {
      # IMGSIZE
      [[ "$IMGTYPECHECKPASS_DRT" == 'raw' ]] && UNZIP='0' && sleep 3s && echo -en "[\033[32m raw \033[0m]";
      [[ "$IMGTYPECHECKPASS_DRT" == 'qcow2' ]] && UNZIP='0' && sleep 3s && echo -en "[\033[32m raw \033[0m]";
      [[ "$IMGTYPECHECKPASS_DRT" == 'gzip' ]] && UNZIP='1' && sleep 3s && echo -en "[\033[32m gzip \033[0m]";
      [[ "$IMGTYPECHECKPASS_DRT" == 'x-gzip' ]] && UNZIP='1' && sleep 3s && echo -en "[\033[32m x-gzip \033[0m]";
      [[ "$IMGTYPECHECKPASS_DRT" == 'gunzip' ]] && UNZIP='2' && sleep 3s && echo -en "[\033[32m gunzip \033[0m]";
    }

    # refurl style,(added 1 more imgheadcheck and 1 more imgtypecheck pass in the middle)
    [[ "$IMGTYPECHECK" == '302' ]] && \
    IMGHEADERCHECKPASS2="$(echo "$IMGHEADERCHECK"|grep 'Location: http'|sed 's/Location: //g')" && IMGHEADERCHECKPASS2=${IMGHE
ADERCHECKPASS2%$'\r'} && IMGHEADERCHECKPASS2="$(curl -Is "$IMGHEADERCHECKPASS2")" && \
    IMGTYPECHECKPASS2="$(echo "$IMGHEADERCHECKPASS2"|grep -E -o '200|302'|head -n 1)" && {

      #sharepoint style,1 addon typecheck pass to the final
      [[ "$IMGTYPECHECKPASS2" == '200' ]] && \
      IMGTYPECHECKPASS_SPT="$(echo "$IMGHEADERCHECKPASS2"|grep -E -o 'raw|qcow2|gzip|x-gzip'|head -n 1)" && {
        # IMGSIZE
        [[ "$IMGTYPECHECKPASS_SPT" == 'raw' ]] && UNZIP='0' && sleep 3s && echo -en "[\033[32m raw \033[0m]";
        [[ "$IMGTYPECHECKPASS_SPT" == 'qcow2' ]] && UNZIP='0' && sleep 3s && echo -en "[\033[32m raw \033[0m]";
        [[ "$IMGTYPECHECKPASS_SPT" == 'gzip' ]] && UNZIP='1' && sleep 3s && echo -en "[\033[32m gzip \033[0m]";
        [[ "$IMGTYPECHECKPASS_SPT" == 'x-gzip' ]] && UNZIP='1' && sleep 3s && echo -en "[\033[32m x-gzip \033[0m]";
        [[ "$IMGTYPECHECKPASS_SPT" == 'gunzip' ]] && UNZIP='2' && sleep 3s && echo -en "[\033[32m gunzip \033[0m]";
      }

      # office365 style,1 addon headercheck and 1 addon typecheck pass to the final
      [[ "$IMGTYPECHECKPASS2" == '302' ]] && \
      IMGHEADERCHECKPASS3="$(echo "$IMGHEADERCHECK"|grep 'Location: http'|sed 's/Location: //g')" && IMGHEADERCHECKPASS3=${IMG
HEADERCHECKPASS3%$'\r'} && IMGHEADERCHECKPASS3="$(curl -Is "$IMGHEADERCHECKPASS3")" && \
      IMGHEADERCHECKPASS4="$(echo "$IMGHEADERCHECK3"|grep 'content-location: http'|sed 's/content-location: //g')" && IMGHEADE
RCHECKPASS4=${IMGHEADERCHECKPASS4%$'\r'} && IMGHEADERCHECKPASS4="$(curl -Is "$IMGHEADERCHECKPASS4")" && \
      IMGTYPECHECKPASS_OFE="$(echo "$IMGHEADERCHECKPASS4"|grep -E -o 'raw|qcow2|gzip|x-gzip|gunzip'|head -n 1)" && {
        # IMGSIZE
        [[ "$IMGTYPECHECKPASS_OFE" == 'raw' ]] && UNZIP='0' && sleep 3s && echo -en "[\033[32m raw \033[0m]";
        [[ "$IMGTYPECHECKPASS_OFE" == 'qcow2' ]] && UNZIP='0' && sleep 3s && echo -en "[\033[32m raw \033[0m]";
        [[ "$IMGTYPECHECKPASS_OFE" == 'gzip' ]] && UNZIP='1' && sleep 3s && echo -en "[\033[32m gzip \033[0m]";
        [[ "$IMGTYPECHECKPASS_OFE" == 'x-gzip' ]] && UNZIP='1' && sleep 3s && echo -en "[\033[32m x-gzip \033[0m]";
        [[ "$IMGTYPECHECKPASS_OFE" == 'gunzip' ]] && UNZIP='2' && sleep 3s && echo -en "[\033[32m gunzip \033[0m]";
      }

    }


    [[ "$UNZIP" == '' ]] && echo 'didnt got a unzip mode, you may input a incorrect url,or the bad network traffic caused it,e
xit ... !' && exit 1;
    #[[ "$IMGSIZE" -le '10' ]] && echo 'img too small,is there sth wrong? exit ... !' && exit 1;
    [[ "$IMGTYPECHECK" == '0' ]] && echo 'not a raw,tar,gunzip or 301/302 ref file, exit ... !' && exit 1;

  else
    echo 'Please input vaild image URL! ';
    exit 1;
  fi

}
echo -en '\n\nprepare DDessentials ......';
PrepareDDessentials;
```

加入了定制grub，以提供给用户手动执行pebuilder的机会：

```
sed -i 's/timeout_style=hidden/timeout_style=menu/g' $GRUBDIR/$GRUBFILE;
sed -i 's/timeout=[0-9]*/timeout=30/g' $GRUBDIR/$GRUBFILE;
```

重要的prepare others->the preseed部分：

```
[[ "$UNZIP" == '0' ]] && PIPECMDSTR='wget -qO- '$tmpDDURL' |dd of=$(list-devices disk |head -n1)';
[[ "$UNZIP" == '1' ]] && PIPECMDSTR='wget -qO- '$tmpDDURL' |tar zOx |dd of=$(list-devices disk |head -n1)';
[[ "$UNZIP" == '2' ]] && PIPECMDSTR='wget -qO- '$tmpDDURL' |gzip -dc |dd of=$(list-devices disk |head -n1)';
```

和packaging部分：

```
  echo -e "\n\033[36m# Packaging \033[0m\n"

  sleep 2s &&   echo -en "make a safe wget wrapper to inc --no-check-certificate\n"

  rm -rf /tmp/boot/usr/bin/wget
cat >/tmp/boot/usr/bin/wget<<EOF
#!/bin/sh
rdlkf() { [ -L "\$1" ] && (local lk="\$(readlink "\$1")"; local d="\$(dirname "$1")"; cd "\$d"; local l="\$(rdlkf "\$lk")"; ([
[ "\$l" = /* ]] && echo "\$l" || echo "\$d/\$l")) || echo "\$1"; }
DIR="\$(dirname "\$(rdlkf "\$0")")"
exec /usr/bin/env wget2 --no-check-certificate "\$@"
EOF
  chmod +x /tmp/boot/usr/bin/wget

  sleep 2s && echo -en "packaging\n"
  find . | cpio -H newc --create --quiet | gzip -9 > /boot/initrd.img;

  [[ "$tmpGENMIRRORBAK" == '1' ]] &&  echo -en "packaging finished,and all done! auto reboot after 9999s...(if needed, you can
 press ctrl c to interrupt to bak the repodir under tmp/boot/, then manually reboot to continue)" && sleep 9999s

  rm -rf /tmp/boot;
```

还有一些小的修补。

给pe配不死booter的工作还在设想和进行中，可能开始会提供一个较简易的方案，而非与制造开头所提到的云booter一起完成。如果能，当然最好

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一个设想，在统一**bios/uefi firmware**，及内存中的**firmware**中为**pebuilder.sh**建立不死**booter**

本文关键字：**firmware in RAM' replacements for UEFI firmware**，虚拟**efi,**编译类**colinux**的**linuxboot**

在《云主机装黑果实践》上我们反复提到一种在bios上也能运行的uefi，这就是变色龙和四叶草。它在内存中模拟一份虚拟的firmware和efi机器环境，以对接需要efi环境运行的os（如果是bios机，可以模拟一份全新efi，如果是uefi机，可以替换或修改相关uefi项），------这样无论是legacy系统还是新机器是实机还是云环境，都可以制造一层一致的中间层机器固件环境和功能来定义os运行所需环境 .

在一致的中间层**firmware**中我们能干啥

在这种”中间层固件，软件化的固件“中，除了硬件加载部分，我们可以写入各种payload（即uefi的app,如一份efi shell，如一份linuxboot中的vmlinux和rootfs，实际上，linuxboot即是这种技术，它保留原vendor firmware的硬件加载部分，把uefi app部分替换成linux+linux rootfs，比如headfirmware：一份集有虚拟机管理器的linux或uboot：一份集有userland golang busybox替代的linux）。，------ 然后，在实机上，这样的一份firmware可以被直接刷入bios rom或spi rom，在云主机或虚拟机中，它们可以被刷入分区代替firmware硬件rom区(相当于强化了的booter,with firmware included)，为bios机重新定义firmware，---- 实际上，群晖中，由initrd和updater.pat组成的二层rootfs，它的第一层initrd即是一个meta rootfs，适合放在这样的firmware区中。所以黑群的引导，从以上的观点中，可以理解为整个引导所在区组成的firmware(也可以理解为linuxpe)，而实际上，这一部分，在白群中，的确是被刷入主板的spi中的。

为**pebuilder.sh**建立不死**booter**

而我们最想做的，就是利用上述技术为云主机上pebuilder.sh建立一个不死booter，我们知道，现在都流行一键还原和折腾安装系统，老旧的电脑，一折腾错磁盘就费了，又插usb又插光盘的 到现在为止，这种问题依然没有解决，大家还是靠U盘救电脑，其实新的电脑也一样，如果能做到类mac电脑一键网络恢复的功能，它的原理是将网络支持，网络恢复写进了firmware中就好了，在这里提供系统在线恢复所需的环境(这个环境局限受payload大小限制往往只有几M到十几M。)，，如果可以，这一切可以日后可发展到实机上作为linuxpe（不用这个firmware刷机）。因为linux的srs驱动支持很强大。

如果是刷机方式，这种booter是天然刷不死的。除非spi或bios rom坏了。如果是仅放在引导区，只要严格要求镜像市场中的镜像格式审核，也是可以极大程度以免刷死的。

技术原理上，四叶草它们用开源的tiancore和ovmf(qemu)达成。由于变色龙和四叶草实际上就是tiancore模拟内存中的efi加强化了rUEFit引导osx的特化（以引导的形式存在的虚拟firmware，功能面向为引导osx），而linuxboot也面向产出firmware，却跟clover不同的地方在于：（它不面向引导的形式，而面向产出一个能被刷入的rom，linuxboot功能面向于：自动化保留vender中硬件加部分和替换payload的功能），除此之外它们一样。

所以实际上，也可以按照linuxboot的方式在四叶草中里面集成linux和linux rootfs。

由于qemu环境和云主机环境始终是最容易虚拟得到的环境和最方便接确到的实验体，所以。最开始制造for qemu pc3.5,i440fx等的这种不死booter支持最符合现实。

等条件成熟了就动手做一下吧。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

软件即抽象

本文关键字：抽象是软件的本质，设计是编程的本质

首先，什么是编程，这或许要先问，什么是软件，因为具体编程就是一种“在某平台下，使用某语言，针对解决某个需求进行实现，某个问题进行解决，由程序团队完成最终递交给客户并维护的整个过程，产生的结果就叫软件”，人们只注意到了作为结果的软件，但其实这里提到了很多实体和对象 ----- 整个软件和软件开发，它是一种寄涉众多的人类工程，每一个都不可分别而论。

具体到工业层次的软件开发和应用上，人们最终追求的始终是如何控制软件的生产周期，降低成本。更灵活地处理新出现的需求和软件本身改良的代价 ------ 历史上存在过的，和发展到现在，我们见过的不同编程形式都可以泛性地归入这个讨论。对于大中型软件开发，存在著名的关于银弹问题的讨论。这就是软件工程和开发的标准定义

那么，什么是抽象呢? 抽象，顾名思义,抽象抽象,抽取事物形象的一面, 那么，为什么需要抽象呢?首先,计算机科学和编程是一门复杂性很高的科学，人脑往往不适于长辐记忆或直接面对复杂的二进制底层,人们在面对根本无法控制的事情时,往往把它们转化为另外一件可控的事 - 这对于W事W物都是这样的。其次,从问题到解决不是一步而就的,所以需要建立中间层,先完成这诸多中间层,当中间的逻辑被解决的时候,最终的事情自然就变得简单了，还有更多，抽象源于一个简单的事实,把事物从逻辑上分开,这样就会解偶它们之间的联系.只有把接口拉高,向高层抽象,那么就可以忽视平台逻辑(实际上正是正面绕过)，面对一个抽象了的简单化的结果 比如抽象是历史上在开发中解决移殖问题最好的方法,而实际上，增加抽象是解决一切编程问题的方法！！（by某某牛人）软件即抽象。抽象是软件的本质。**抽象足可以被称为“软件原理”**

总之，既然软件与开发在工程层面源于上面四大件（平台，语言，问题与需求，设计与人），抽象就在软件的这些各个层面都存在。下面，我们就从大的工程方面，从编程的最初形态开始，来讲解抽象在这四个层面各有哪些关联和体现。

1)平台和语言层中的抽象

最初,人们想得到一台能高速模拟数学计算的自动化机器，试想只要有了这样一套机器和数学方法，只要机器的速度足够快，形式化的问题有解，它就能在人类有意义的等待时间内产生结果，图灵机的运作原理为自动机理论，它的计算方式被称为形式计算,形式化了的问题就叫“算法”，图灵机理论揭示并解决了现代计算机在抽象层次运行的本质即形式计算.这对计算机发展和工业化的意义不言而喻,.这不光是硬件上的，还是软件上的，

硬件上，出现了冯氏机.冯氏架构将执行指令的CPU和存放程序的内存分开,其带来了可置换软件的雏形 - 程序机器分离。CPU作为主导，集成了对内存的管理,是计算机中的总控制中心,机器运行的过程就表现为通电状态下的CPU从同样通电状态下的主存里取指令并高速译码串序执行的过程.在CPU的眼里一切都是内存地址,指令也是内存地址. 可见，这里**CPU**不但执行指令,还管理组织数据的事.-------- 这深刻描述冯氏模型的特征和导致了冯氏编程的统一性.冯氏的哲学观就是数据和操作数据的指令。数据即指令，指令即数据。

软件上，就是算法。既然在CPU眼里,一切都是内存地址和指令,而且这导致了冯氏模型的统一性.这些仅仅使机器知道如何寻址和译码，至于计算机如何形成任务,它能完成什么样的伤务内容呢? 冯氏计算机是如何把问题-这里是科学计算问题，形式化的呢？最初的人们是如何使用最原始的计算机的呢？------ 这就是CPU级的执行路径，由数据和指令组成的程序分片断在"栈式机"里运行，每个片断在一个个高速生成-消亡的可执行单元里进出，占据CPU的时间资源，CPU有机制直接表达这种栈帧 - 它里面的寄存器，CPU也用科学计算处理单元来完成指令的执行，空间上,CPU用寄存器编址内存地址，CPU也支持编址操作内存中存储的整型和浮点型数据，形成最终组成应用和程序的指令序列和数据址位，最终完成程序的执行。------ 这时的计算机应用当然首先是在一些大型主机上作科学计算不能作通用计算。机器是裸机，这时的软件形式就是一些用汇编写的应用，最初的软件就是问题尾端的应用，,开发上，没有复用，写程序一切都要从头开始，程序是离线开发（电传在另一台同类型机上开发调试），烧录到ROM的，开发过程就是直接电传写裸机上的应用（没有开发支持和安装支持）。软件也不能称为软件（我们把出现OS之后软件称为软件）。

总之，这便是最初可编程(科学)计算机的基本雏形以及开发和应用的雏形。它是图灵理论算法的最原始表现形式(算法可以是一切最简的可形式化，到任何复杂化了的可执行，开发，发布程序形态，但始终不离图灵计算本质的那些计算问题，以下会不断提到)。直到出现真正的软件系统使之变得现代化，这就是现代OS的出现。

那么为什么要出现软件系统呢？

最初的大主机的处理能力太过于空闲,一个当前使用机器的用户对于这台机器的处理时间和处理能力是独占的,机器内没有管控这些程序该如何更好运行，出错了怎么防崩之类的逻辑，而且，不仅机器本身没有自理能力人力管理和使用机器也麻烦，很多很多年前，那时，会编下的操作员和程序员,要完成先控制机器的任务(机器上有重启,暂停等按钮,在机器内也有相应的控制指令)，再使用机器，效率不高也不够方便。更别说开发了。

于是急需出现一种能代替人管理机器的层面和机器管理程序的层面，将其显式地整合进一个随着机器存在的层面直接常驻裸机之上，这个层面就是最初的软件层面，（最初的分时系统，这些层面就逐渐演变为“软件OS”的原型。此时，它可能并不叫“操作系统”。），再后来，人们自然想到，由于计算机的硬件速度足够快，为什么不到所有能想到的，都软件化整合进这个层面呢？比如为什么不把编程都搬到这个机器和集成到软件层面来呢？这样，各行各业使用计算机的人都可以面对一个统一的，大而全的，更强大的软件层,，普通用户可以面对一个更强大的功能，PC可以接入更强大的外设，供更多人使用，计算机终于开始像个PC了。对于开发者，高级语言完全也可以实现在软件层，软件开发发布就可也在软件层更方便进行，程序员可以直接在软件层实现应用，使用平台上的API，（这里并没有一个显式的OS和高级语言谁先谁后出现的问题，但肯定一前一后紧密，系统和系统开发是一对自然而然的结果和现象，比如OS本身也需要用更好的语言来开发完善，更强大的软件要求出现更强大的开发手段。。）。。

高级语言系统同样基于图灵理论，编译原理即是这样一种学科，发明语言本身用的是”图灵算法“，它为源端的源码建立一套形式表示，映射为可运行的目标逻辑，并传给目标运行（这里的目标即OS执行栈），语法表示和语义映射这样的问题，是高级语言前端的重要课题，由于是图灵原理，它要尽量避免二义性，用计算机来识别的语言只需是(不是只能,而是只需)某种形式语言而非我们的自然语言(如果硬要这样,那反倒是不合理和不必需的,因为自然语言是一种形式语言的超形式,在语法语义层面存在很多歧义,比如一个句子有二层语义,虽然这是现实中一语双关的美妙使用,但是在编程中这正是要极力避免的,仅仅因为不需要那么做,计算机本来就被设计成只处理确定性的算法。

在OS层集成高级语言，不管如何，事实上人们也这样做了，此处略去1W字有了操作系统和高级语言之后的好处 ----- 我们分别着重来说这二者给开发带来的新的复杂性和便利性，

对于开发，OS层为开发新增了一个平台和基础抽象的层面。平台上使用高级语言进行开发，发布，必须遵从和适配OS上的以上服务，调用和处理与此相关的所有抽象，对照原来只有汇编和硬件平台是直接对应原始的图灵运算模式的原始形式，它产生的程序本身就成了某种高级的混合算法态--我们暂时把它称为OS层的功能抽象。如果说OS那些抽象给开发带来的是功能层面的，那么，语言给开发带来的影响更为明显，它带来的是映射层面的整个叠加。------ 比起汇编和汇编语言是直接对应机器逻辑和语言机器逻辑一一映射的。用语言进行开发时，它产生的程序本身也是多种复合了的算法态。

> a) OS为开发贡献的新的功能抽象层有：
>
> OS带来的程序运行栈：OS为软件准备了一个过程和函数栈，高级语言（我们稍后会讲到）的后端链接产生的程序会链接到OS运行可执行体的服务（它们在语言中被作为内存分配等函数）。 OS层封装软硬平台的服务：硬件的特性会被反映到编程上，编程语言上作为抽象，比如原子操作作为语言关键字。平台的并发有时也会成为软件的直接支持设施。 OS层的API服务：OS为应用准备了一套API，所有语言产生，运行在这个OS上的程序，都可以调用其API服务。 OS层的组件支持：OS上的应用可以以二进制组件复用体的形式出现，进行链接，生成，和运行。 OS也为应用准备一个appmodel，（这个放在领域问题部分来讲。）
>
> b) 高级语言叠加了一层映射计算方式作为抽象的叠加参与开发。
>
> 首先，高级语言中的类型抽象。
>
> 在前面说过冯氏模式就是处理逻辑和数据的，甚至这二者本来就是一个东西，高级语言会有一个类型系统，这里指的主要是简元类型系统（UDT，ADT这些扩展语言类型的放在跨越语言的复用部分讲），简元类型模拟了汇编下CPU的类型，而函数定义，过程式流程命令则显式化了原先无义执行序列中的部分，使之成为有意义的逻辑单元，冯氏开发就二个东西，数据和操作数据的逻辑,到了这里，类型已经成了一种综合数据结构抽象和代码结构抽象的东西（甚至它还综合了API和接口这些复用的机制。甚至语言的类型系统，OO使扩展语言的库成为一棵关于类型的大树。这放在复用部分），这一切都是抽象，让语言映射呈现出靠人便于使用方面的抽象。
>
> 而语言后端与OS运行栈一起，，完成语言系统的根本任务。
>
> 可见，OS和语言层为开发引进了抽象的叠加和复杂。是不断丰富的平台和语言系统给开发带来的方便性和不方便性产生的新矛盾体，它在带来方便的同时，也带来了这些复杂性。那为什么不抛弃它呢，这个问题问得好，问出了抽象的哲学出身：这其实是一对矛盾。是抽象的特点。抽象是用更强大的方便性掩盖和代替相对不那么重要的方便。

2)超越语言的抽象层，及问题域和人的工程需要给开发叠加的那些抽象层

语言逻辑将到现在为止讲到的一切，包括语言本身和平台抽象的部分，都统一于语言服务，或封装在语言标准库，或封装在第三方库,在计算机看来,都是图灵原始算法不断复合，演变的复杂形式和高级形态.最终统一于高级语言开发。

那么，除了以上OS和语言本身带来的新的复杂性和抽象层，还有哪些会随着语言进入开发呢？这就是超越语言的抽象层，及问题域和人的工程需要给开发叠加的那些抽象层。它们超越语言，但也可以作为语言服务存在。比如，将加入人类工程和问题抽象的一次性设计，做成框架，这属于复用部分。

这首先是高级语言时代的算法抽象。

当图灵能解决数学问题的时候，它实际也能解决这种形式化表达的任何通用问题。试想“1个男人+1个女人=？”这样的抽象问题如果能得于形式化，在图灵机眼里都是可计算的，等价的，跟“1+1=？”一样，冯氏机自以前汇编时代就有这种能力，之前汇编的算法我们讲过，就是根据编程上的数据类型（汇编下是CPU的类型）然后按计算机的方法一定套路操作数据形成结果，只是在汇编时代，开发上没有让图灵算法一次演变到这种复杂形成的可能，那么现在正是时候，装配了软件层面和高级语言的冯氏机就正好可以在程序中抽象这些并映射解决这类的复杂抽象形成新的复合算法。并在软件和高级语言时代，算法形成了真正自己的学科。

所以算法，它首先也是个数学和科学问题（用计算机作科学计算）。它超越于具体语言语句和类型抽象之上，（因为它是跨越语言的，算法是计算机的学科，不是编程的学科），是开发界实现最初抽象，利用计算机原始的解法，建立问题域抽象的手段。

> a) 算法层和实现抽象层。
>
> 那么这又是一种什么样的抽象开发过程呢？ 这里面的抽象就在于，VS机器和汇编时代CPU类型和操作法，这里的相当于高级语言中的变量和类型。算法相当于数据结构+数据结构操作。 它将算法用语言的简元类型来表达成数据类型抽象，进而可以表达成数据结构抽象，然后结合处理这些数据结构的模式，可以提出ADT类型供复用。 但当谈到ADT的时候，实际上进入了下一步，跨越具体语言的复用层级了。
>
> b) 语言的接口和复用层,扩展层，继续抽象层。

> 分清语言的实现抽象层和扩展抽象层
>
> 在前面不断提及过语言的复用层，一不小心它就总是和语言的简元类型和技法重合，高级语言最基本的类型和用法，过程式和简元类型抽象实际上就是最基础的可复用设施复用支持。（对类型的抽象就是编程设计的冯氏数据抽象思想为中心的开发方式，类型抽象和数据结构是高级语言用语言来解决计算，描述算法的最基础的工具。而代码结构就是复用手段，工程手段。所以提出类型本身，它就是一种复用，甚至这是语言用类型来完成语言本身的手段，比如运行时和库树），不过也有复杂和显式的（比如算法科学中，简元类型可以完成算法，但ADT则是用户抽象扩展的分水岭。）。现在我们正式来归类复用层面给开发增加的新抽象，------ 为什么复用这么重要，因为高级语言在映射层能解决的问题所有问题，就是使用算法实现抽象和使用复用连接已有实现或第三方抽象，扩展用户抽象，除了算法，高级语言的复用支持是语言对接人方面工程目需求那层的 -- 接口和复用。它是高级语言能连接抽象的第二种手段。
>
> 更复杂的接口和复用层有：高级语言习惯用法。tricks，技法，语言的UDT，ADT扩展系统，高级语言的API和二进制复用机制。库级扩展系统，组件扩展系统。过程范式，等
>
> c) 问题域
>
> 在软件时代，这里的软件和软件开发分系统软件和系统应用软件，系统编程领域，包括OS的发明，语言的实现本身，都是系统软件。上面提到的算法和数据结构，就频频服务于系统实现与系统应用开发。
>
> 问题域为开发引入了什么新的抽象呢？ 首先，一个现代的OS，基本涵盖了对GUI，graphics,network,socket,持久，io，等领域的基本抽象建立。然后，一个具体APP中，最大的肯定是问题本身所处领域逻辑。 最后，一般OS基本还建立了２个APPMODEL层，一个是console appmodel，一个是gui appmodel，你可以将appmodel理解为程序的模板，其作用将domainlogic+appmodel形成最终该OS下的一个具体app。
>
> d) 人带来的新的设计抽象层面。
>
> 当考虑到人的需求，它又是一个更高级的，超越语言和问题域的话题，归纳一下，它为现代开发引入的新抽象有：设计模式，比如设计模式，可以在很高层次上，为问题建立一套MVC的框架。
>
> 甚至更多的语言，如带来了脚本语言，更强大的虚拟机语言这样的课题。和领域语义，如xml, 中间件，更细分的appmodel和问题域抽象，web appmodel,game appmodel这样的东西。，你可以将它看作为是对前面1)a,b,2)a,b,c的全盘设计，修正，选型，创新，和附加，和全盘颠覆。或部分全部直接采用。 如QT框架库，它是对CPP语言，平台库，等等，这四个东西的封装，修正增强，这里有鲜明的重设计特征和工程总层面进行开发的痕迹。

3) 总结：软件和开发的原理和模式-如何建立抽象，给问题建模，复用（扩展模式）和映射抽象，处理平台抽象

所以，看出来了吗，软件和开发的原理和总模式，其实就如何2)中abcd一样，已经证明是这些抽象不断叠加或修正的过程呈现。而人类设计，它是对1)a,b,2)a,b,c。的部分全部直接采用。或重设计。最终，通用OS和高级语言如何解决新时代下，解决程序运行和开发的问题，就是解决如上抽象和设计的问题。

比如设计模式，当谈到设计是，表明这是一种人类设计。将加入人类工程和问题抽象的一次性设计，做成框架，这属于高级复用部分，同时又是典型的设计。

这是《编程新手真言vol1:编程新手抽象与理论》下文我们谈设计，编程即设计，

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

语言选型通史：快速整合产生的断层

关键字：兼容后端多语言体系,免**binding**一体化后端，**llvm cling:**全栈全范式语言系统

在编程语言选型上，初学者往往希望选到一门最热门最简单最全面最省事的语言。更并且，他们经常寄希望找到的学习项目恰好就是这门语言的，这样他们可以坚守一门语言积累起自己的一套codebase，供学习也是以后工作上的开发而不用变动。—— 可是往往事与愿违，于稍微高级一点的需求领域及现在web,mobile,native开发现状和日益分裂的语言生态现状，，这样的“方案+学习项目”的组合，往往很难存在。于更全面的领域，这更是不可求得。

因此，人们很自然地会从C系转到C++（或许会发现CPP有点复杂没有动态消息转QT式CPP），再转到动态脚本如PY（发现编译语言太复杂转动态脚本），或py到C#,java（被宣称是通用语言），最后又转到js这种看似全栈的“大一统语言”，到最后，发展到发现每一种语言都其实需要学的境界 ——- 这绝不仅是语言各各内部语法的区别，更是语言间所处阶层产生断层导致的。是业界不断变动的开发方向带来的自然结果：不断提出的新项目/新语言/新开发体系都希望自治，以致学习上的生态碎片化过大，交集众多。

本文接下来的部分会展示出这些需求的出现的断层变化部分和语言选型的殊途同归大方向。

1，从孤立语言体系到统一后端免**binding**语言体系：学习曲线的断层

大部分科班或自学的人都是从C系开始的，这个时候他们往往关注语言的写法，学习的成本主要是语法即语言的前端应用部分。这个时候仅是一些结构语法，问题部分，就是数据结构这些简单的封装/实现手段，然后开发更复杂本地应用的时候，他们遇到了学习CPP和OO的情景，所幸，CPP和C共享很多语法和库，且后端兼容。换言之：C，CPP毕竟是同生态的语言。

情况开始变得很糟：虽然OO这些语言内手法是被标准化了，但他们开始需要学习一些非C的语言。比如新出现的web开发需求之于python,虽然python解决web问题良好，但当一旦涉及到C系/py交互时，需要做复杂的binding. —— 这只能求助swig这样的工具或boost.python这样的库，不再享有C系内部CPP对C，或C对CPP的调用方式。总结：这是后端带来的断层。而前端的差别实际上可以忍受（实际上强调多语言风格的开发反而正是好事）：多语言结合开发才是主流。

> 后端带来了巨大的断层，使得维护一套共享 srclvl codebase的多语言项目成为不能实现的事情(无缝开发部署)，这根本是因为新语言（他们往往是一些脚本语言）有自己的后端，不再属于C系，除了语法前端上巨大的与C的不同的那些差别，后端带来的巨大学习/开发沟壑开始变得尤其巨大。而且开始出现偏增强后端化的特点：VM式语言正是软件后端，而动态语法是受这种后端支持下的新语法。比如后端与前端不再离线，在脚本上可以调用内省这样的语言服务。— 这本质是集成而已。

情况开始变得稍微好了一点：从Cpython这样的孤立后端，再后来他们发现了java,.net这种统一后端的语言和强大的统一类库如j2se,j2ee,.netfx sdk。比如它甚至可以统一编译式/动态式语法前端。如.net的DLR。这种统一后端毕竟无非就是VM式后端的极大化，这里导致的主要变化是：在统一后端语言下可发展多种语言前端。多语言在统一后端内可免binding相互调用，多个不同语言组成的demos或学习项目可以在统一后端下形成一套codebase。但与C系相互调用情况并没有变化多少，是伪统一后端，因为它没有从C系整合。没有考虑进C系后端和没有从C系要解决的那些native开发问题开始整合，在那里看依旧是断层的。

无论如何，在特定统一后端下，多语言开发成为主流，这可以分散CPP所谓的单语言多典范带来的压力。学习成本转移到了后端。

> 其实，面对多种语言选型，人们大部分缺少和需要的仅是一种免binding的一体化后端方案。因为他们找到一些开源的学习品往往由多语言开发，需要做binding，这时出现一种免binding的方案可以迅速为他们积累起一套codebase. 一种兼容后端，多种开发前端。并将他们做成一个体系。—— 这对学习上统一一套学习重点也是有帮助的，即：后端类型技术和库才是他们要关注的编程学习重点。这样可以结合编程教育不致于太突出语法：可以任意选一门他们喜欢的更简单的语言。

情况开始变得更好了一点：还提出了js这种“全栈”语言。这可以视作是从问题域整合BS/CS开发架构的一种努力。

然而这二种方向都没有以解决最根本的问题为前提，即：也没有统一起所有问题，是断层的。遗留的问题依然巨大：js面对的问题依然不能统一起native开发的那些问题—只要它是虚拟机后端实现的，且一门语言无clr,jvm这样的大规范，根本上不能代表任何稍微大一点的完整生态。

2，更轻更好更全的整合方案：**maybe llvm based clang/cling?**

llvm可以发展包括c系,cpp在内的语言，其本身类.net cls的特点，使得任何现有语言前端可以将后端筑在llvm上，并为之生成ir码。可以说是类.net,jvm的不二代表。如果基于llvm的多语言成为现实，再加上一个统一API大类库，可以完全实现类.net的宠大体系。

第二，clang/cling，cern的cling可以完全达成像js那样的全问题栈用一种语言来开发的效果，且能达成保持C系语法和C系后端都不用改变的平滑编程学习曲线。root sdk内置sginal/slot可模拟qt动态OO效果。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

编程语言选型之技法融合，与领域融合的那些套路

本文关键字：**oneforall** 编程语言真的存在吗，多语言统一学习法，统一**native/web**语言和领域**,the web api/service and serverside/cliside web programming essentials**

在以前的文章《语言选型通史：快速整合产生的断层》中，我们一直讨论one for all编程语言，开发是一个四栈（语言系统，平台，人，问题/应用）的综合过程，因此对语言的讨论涉及到这四者的边界，只能综合而论。本文档集也一直在探索一种自然不断层方式的native/web，本地/分布APP统一的语言系统和编程模型(其它三栈我们统一称为为appstack/appmodel编程模型，因为它是被开发的主要对象，和语言要处理的目标体)。业界从来都是“先用起来再融合”原则，这使得新旧语言和编程appstack层出不穷。支流分分合合，语言系统在融合appmodel也在融合。事实上，我们开发本来就需要多门语言综合，但业界显然也在探求语言融合之道，先来说语言的融合，它主要分为二部分：

语言技法的融合：语言内的**DSL**化方向融合

（1）首先是统一前端语言，这是主要的融合方向，我们知道，代表一种语言本身最具辨识度的就是它的写法，也就是技法。新语言都是技法融合的代表，兼容(源码或者api/abi二进制发布级)和多范依然是语言内融合的主要手段，如cpp融合c,go融合lua,rust融合c/cpp，只是，在处理工业需求和学习曲线上，CPP的多范型固然没错，就是复杂化的痕迹很明显。所以会有go这样的语言来简化。这样的结果是出现了多种语言前端和后端组成的语言生态系统（注意这四字，语言从来处在其它三栈的边界，所以是一个生态的一部分,比如pme适合GUI，async适合分布异步，微后端化可以被用于嵌入编程，而唯有C系才能进入系统内核编程:内核空间与用户空间的编程环境非常不一样只支持C语言这就将任何基于脚本的原型语言和设计排除在外。静态编译类语言安全，而解释脚本类开发时效率高但不安全不能用于高精密领域，语言的逻辑热更新，编译型语言也能办到，这与语言是什么属性毫无必然关系，------ 这些都是语言与其它三栈无层次联系的几个例子，代表了它们的适用领域和人群需求，发明初衷）

（2）另一种方向是统一后端，如jvm体系，clr体系，各种技法和语言前端也被分散在了前端语言中，如jvm上不但有java也有jpython，clr上也有多种语言，这种统一后端方向的融合工作也没错。相比统一后端，统一前端要复杂得多才是真本事，真根本。这样的结果是仅出现多种语言，但是生态得到了极大简化（比如问题域统一/库抽象比较统一可以多语言共享）。

这二种融合方向，其实都基于一种最基本的事实和本质：随着新出现的业务需求和人的要求，任何语言都需要不断DSL化。而任何语言，它们的内部和外部扩展能力。都是这门语言出生时的“最小使命”，来归纳一下现今现语言的元编程能力：

> ps: dsl化方向的融合：
>
> 语言系统最小应该提供什么，然后才是其它什么东西？即自我扩展的能力：DSL能力和元编程设施 如CPP有范型，这种参数化类型成就的泛型和元编程，这使得CPP可以在编译期仅靠语言本身技法就可以实现DSL，辅助程序员完成“库是用来扩展语言的这种说法，但更重要的是需要语言本身也可以扩展自己”，另外二个运行期语言系统的对比，就是js和py了。js可以在语法技上用函数机制直接在抽象ast上写程序（任何语言机制都可以用函数抽象达成，而且这种抽象受语言核心支持,都可变成一级对象）具备元语言能力。而py可以用“meta object”之类的语言级抽象来提供元语言能力，辅助程序员完成“库是用来扩展语言的这种说法语言自身也可以”，后二者在运行期追求语言结构的变化（比如内省，这对GUIprogramming抽象亲和但不够安全），提供灵活性，但有得有失，不够安全也增加了运行期的维护调试成本。这是 cpp/rust vs js,py这二门派语言在借助编译器提供DSL能力上的显式区别。------ 否则。为了完成以上目的，语言需要提供显式化的“用语言本身来发明语言”，需要使得语言本身被服务化，可以被作为库调用。比如像terralang,racket-lang（A programmable programming language，language oriented programming language）。这种显式宣称自己有DSL能力的语言。---- 但是它们将这种能力放到了语言设施外。

但是，无论如何，上面的任何一种，都是”语言系统“级别的融合，其实这都使得语言学习成本增多。没有统一学习法,一门语言应该核心稳定，为了长久的应用或学习曲线，，，但是开头说了，业界从来都是先把想到好用的先用起来，再慢慢解决不那么重要的麻烦事。即融合使之进入同一个生态更好用。

那么能不能改变开发中的其它三栈行为，使得语言跨域实现oneforall呢（如同上面的lang本身DSL化一样）？

appmodel融合：语言外对语言融合的要求与实现

业界终于想到对融合APPMODEL下手了。这就是上面不那么重要的麻烦事之一。

我们知道，到现在为止，我们的常见领域主要有欠入式os实现类/native/web领域，语言都宣称它们是通用类，但实际上都被设计成领域实用和领域适用。现实中语言无论上面提到的哪一种，其实都可以DSL无限化，但都是在一种语言生态内完成的都不能跨应用域，那么有没有真正跨域的语言呢，实现语言真正的one for all？比如最难攻克的native/web融合....

分布式应用现在最典型的就是web，很难为分布式框定APPMODEL，因为这是一种跨OS跨进程的“APP”，---- 分布，是一种产品分离，也是抽象的一种，同时关乎产品设计和语言程序设计，可以用多种方法来达成。web就是一种。但是这种领域不能做本地计算密集的游戏应用。而且要求的语言跟nativedev大相径庭，那么有没有让他们共享同一种语言系统呢？

> PS：the web api/service and serverside/cliside web programming essentials

> web作为业界分布式程序的第一种广为流传的尝试和成功案例，其实它带来的误导性和建设性(支持页面链接，碎片化程序发布即page app)一样多，web误导了真正的分布式很多年。使得plan9和x11这类原生分布式都失败，单就其bs web技术选型中的浏览器技术，基本用可编程的clientside UI(js->page)代替了app ui，一条道走到黑。先不说这个，
>
> 现在的web/webstack技术中，服务端的主要任务主要集中在MVC框架内的东西，即数据方面orm协议方面url regex route/request/respone模板template filter，包括了部分客户端逻辑,这部分客户端逻辑可以完全交给客户端来做。服务端即MC。客服之间交互的，或者是request/respone这种传统带状态的协议，或者post json作为请求的webapi。 至于客户端方面，就是一个可编程的v8 render,。web编程的协议本质是对request/respon编程，所以这是客户端和服务端共同的，客户端还主要负责DOM编程（以后扩展的webcliside技术还有很多~~），，有jquery,pwa,react,ajax这些客户端技术来与服务端的api交互。综合完成webapp的效果（会前后端开发的叫全栈）。我们讨论的就是这种前后端分离的web编程。区别于传统页面跳转无状态的方式。-—— 这种api和面向微服务的web开发特征是：将gui分离出去。web就有了类native的appstack和mvc,,前后端分离，仅靠一条api地址和参数交互，就分离了本地和分布式最根本的问题。各具体组成栈最大解偶，任何app，不要把gui作为框架和app栈的一部分。后端只需负责services。把写业务，和写界面的语言和生态分开。
>
> 对于可编程的web的要求，是为其装配某种web可用的语言件api供人们使用,这样才能接上语言和开发，而web是现在唯一的真正的跨端跨OS跨设备跨语言的编程模型和appstack，这也是它API方面的要求：历史上，这些web api,web service技术其实很早就出现，最初它们是在本地做的，为了达到分布式跨一切的效果，以前是利用各种rpc/rmi技术规范给本地语言导出的web函数建一个调用方的本地桩，使之分布式和组件化，这种rpc webservice越做越复杂就淘汰了，因为脚本语言出现了带来了源码即组件的概念：即通过虚拟机语言往往是跨平台的特性，一方面使得web服务本身并没有任何来自OS的载体webapi纯粹脱离本地平台化API，，另一方面分布式要求开发件和发布件一体，脚本语言刚好src api即web runtime一体，在编程上可作为虚拟机语言的脚本源码组件代替传统API桩的作用，而所谓API桩 ---- 其作用相当于一种语言或平台port给另一种语言或平台的代理和转接逻辑，为了API而设的wrapper api，一个类比例子是软件方面的桩，如x11server，它实质是部署在客户端的server。真正在服务端的是显示服务器,所以这种x11 remote app本质也是一种分布GUI的APP（跟webbrower一样），plan9的分布FS协议也是，而我们传统的WEB是另起灶炉设计的，这就是我们第一句话讲到的web的误导作用，误导了真正的在OS级直接提供原生分布式服务的架构发展，曾经为了web跟nativeapp接近搞出了ajax,websocket,wetbgl，好了不说它了，都是泪。

这就是说，web开发和nativedev天然断层，完全不搭。下面是业界对他们的融合之法：

为了达到使用同一语言能领域开发融合，需要针对appmodel栈中的平台，应用规范，整合具体语言针对的领域。或者作广泛意义上的appmodel融合。

（1） 作为融合能多领域开发的语言的例子，业界能做到这点的唯二的二种是go和rust：

比如对于跨web,native的开发，可以不改造既存语言和它们的生态，却综合它们的生态，也使语言建立层次。就如同c之于汇编一样，使新语言提升到高阶。比如，正如c是汇编的高级语言，rust是c/javascript的高级语言（视c/js为高级汇编,c,js都是基础语言代表二个平台。而go,rust这类都是主语言），rust/go是可以转译到低级语言javascript或c的。现在这种转译语言语言还非常流行，比如typescript可以转译到js,elixer可以转到haskell.这样就统一了语言针对的领域。

（2）为了进一步统一native/web，于是就有了emscripten和asm.js这种东西（在浏览器沙盒中甚至都可以运行linux，和qemu虚拟机，当然这些目前都是实验性质的）。再来看wasm，WebAssembly 不仅可以运行在浏览器上，也可以运行在 非web环境下。非Web环境可能包括JavaScript虚拟机（例如 node.js）。但是，WebAssembly 也被设计为能够在没有 JavaScript 虚拟机的情况下运行。这基本是浏览器中搞nativeapp，而且绕一层js，它还有一个意义：让前端支持cpp等语言，使js不再是前端唯一语言。只不过目前wasm仅是用来辅助js的一个扩展，而业界甚至也搞出了纯wasm的前端.

以上这2点，类似语言的DSL融合合，但又是关乎appmodel融合的。其实都是最终appmodel对语言要求的侧影。因为开头就说过，语言会提供什么技法，都可以被归结为“语言技法与问题域结合处的那些东西，要求语言提供某些特定的习惯用法与设计模式：如frp,pme能很好GUI和异步，mvc能很好处理WEB框架”，语言技法和领域都会让语言呈现变化一样。

下一文《盘点语言技法与问题域结合处的那些习惯用法与设计模式1，2，3》

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

qtcling - 一种更好的C++和标准库

作为一个程序员或编程技术爱好者，你是不是开始厌倦了各种虚拟机语言和脚本语言??no vm scripting

它们要么不是C系的。需要你重新学习一套语法。如python,c#,java,js之类….

可这是多大的资源浪费啊，要知道，C是这世上唯一的通用基础语言的教学典范啊（pascal也算吧。。。），计算机专业的学生和非专业的人士都是靠它入门的。学习曲线上自然希望以后学的高级语言也是基于它的为佳。相信很多人都在为适应开发而不断学习新语言的需要而苦恼，而且，大量第三方模块需要binding to c才能使用，而且即使转化后也只是运行在某个托管的环境下，还需要带一个宠大的执行环境才能完成发布或运行。是的，就是那个可恶的虚拟机。这种用内存和CPU资源模拟软件计算机的方式，简直就是在OS上再造了一层执行时。这种过度封装对有技术洁避的极客来说还能怎么忍受呢？

要么，像这类语言，语法上个个宣传它们是通用脚本语言，可还是专用性很强，如PHP大多都用于WEB as dsl,这造成的结果是不通用且怪异，而另一些语言，它们还可能变得越来越宠大和复杂，这是因为为了迎合日益复杂的应用的开发需要，它们需要不断集成，纳入各种新元素到语言生态中去，像java,其生态链已经进入到很完善的地步了，它们还可能无节制地膨胀。没有一个干净的典范开发方式和语言核心存在。这与CPP强调的保持一个干净核心的原则相左。

所以说，虽然它们有组件特性，可能有REPL环境，有脚本特性，可是在以上这二个巨大的缺点面前。那些好的方面也背上了不好的光环。有没有一种基于C系的解释型或带REPL的语言环境，既有传统CPP的好处，又可以直接在这种语法上无改地，或尽量少改地作脚本编程或解释编程呢？可喜的是，这并非技术的桎梏。可以有：

1，第一种是tcc，据说它的编译速度之快已达到了解释语言的级别。可实际上，它只是编译速度足够快而已，称不上，也不可能称得上是解释语言。略过。

2，然后就是我们的cling了。cling/clang是cern代替cint而开发的，基于jit,jit是一种能模拟REPL的技术，当然cling一个光吐吐的编译器还不够，cling/clang可以直接调用C系模块(call into libraries)，用户使用其内置的宏语法就可以测试。这使得在cling下组建自定义的CPP开发环境尤为现实，大多脚本语言都是先出来编译器，然后其它是binding C的，cling天然有纳入各种库的能力，所以有条件建设成为一个完善的语言系统，cern rootsys就是这样。

cling需要整合各种第三方库，原始的cling支持的库和扩展十分有限，一个在windows上不支持#include 的cling编译器语言是没意义的。一个具体的第三库如QT的整合，因此也可能需要面临各种问题，

等等，亲，你不是说cling是基于标准CPP实现，可以直接调用c系模块的吗，是的，但是局限也是有的：

1，可能模块有特殊的扩展。如qt的源码不是标准的clang能理解的，是受moc转化过的，带pme字典信息的。这种肯定需要转化过来。

2，有一些受clang特殊限制的，如内联汇编在clang中还不能工作得很好，使用了这种技术的QT库自然需要动点小手术修正，因此对qt源码的改造是需要的。

话说，克服了整合qt到cling，这足以成为一个十分实用的qtcling语言了，有了qtcling,从此我们的Cer就得福入门了，只需要学习一门语言，一种典范 – QT式基于PME的OO，我们就可以做系统编程和应用编程了，甚至是脚本编程。

（演示视频请通过点击右下原文链接查看）

下载地址（本站下载）：

qtcling

http://www.shaolonglee.com/owncloud/index.php/s/wh3yzswLrtNhwaa/download?path=%2F&files=qtcling.rar

---

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

clingrootsys原理剖析(1):JIT到底是怎么回事

所有的高级语言技术，都是由前端的翻译转化，源码理解，和后端的运行技术和语义实现的: 即编译-链接-运行循环这个标准过程组成的（真正了解这个三段式过程，无论是多复杂或复合了的语言系统，给其定性将不再是难事），而且其编译器实现一开始都是以静态过程式、函数为实现机制的。都是C语言和标准编译原理教程那套。而高级和复杂语言实现，都是先过程元素，然后再在编译器前端实现语法增强，或封装到class和库级增强实现的。（而真正分清这个，可以分步理清很多错综复杂的编译原理过程。 特别是cling这样的复杂语言系统的定性和实现原理。包括其实现，如JIT和库级pme都大有帮助。下面细述。

什么是解释系统

解释系统与编译系统最大的区别是在一个前后端配合循环(标准编译原理上的compile-link-run，实际上这里的compile更适合称为translate才能与其它语言共享同样的编译子过程称代而显得无歧义)中，它每次只取一条代码（最小生成和执行代码的单元）来运行，且纯粹依赖语言系统本身的后端—往往是一个软件系统 来运行，往往无需link，因为它是运行期完成query的，类似组件（接下来本文第三部会说到这个组件有什么魔力）。 这里的软件系统运行设施往往还负责其它方面的工作—比如一个虚拟机要完成的那些当然也有可能不像虚拟机那么复杂，就是一个简单的if-case流程（这是最初脚本语言的原型），它属于整个语言系统的一部分，所以说是在线解释。由于它在翻译和链接过程中是运行期实时完成的，运行期的那个大VM天然维护着这样一个软件从翻译到运行所需的全部设施的宠大环境 — 它就是整个软件化的语言系统连执行都是软件托管的，所以在前端它不耗时（较编译系统而言），耗时的是运行期的代码效率。 而编译系统是事先将整个应用一次全翻译好（这里耗时巨大），将每个模块链接起来（这里又一次耗时巨大），依赖后端-往往是一层薄薄的支持层仅起传手传递作用 – 传递喂给机器offline运行，此时的运行不干语言系统的事所以说是offline。由于机器执行机器码是2的8次方这个效率（这个数字还在不断增长），所以自然就快了。 解释系统每次只执行一个代码的本质无非是在编译器实现中在后端提供一个execframe对象就能达到，使语言系统具备生成动态最小可执行单元，比如一个调用路径中的函数栈，然后将代表其的ast node喂给后端执行时-那个execframe就行了。解释语言最鲜明的技术特点就是这个动态执行后端。可见它跟是否必有一个虚拟机没有关系，跟是否必生成中间码或平台码，或以其为最终目标运行没有关系。它就是一个普通后端会“emit function->feed it to execframe”的东西，免compiler免link，后端直接运行代码（一般运行后端可直接识别的中间码）的东西。 我们称以上这种解释系统称为普通解释系统。 令我们迷或的往往是JIT式的解释系统。而实际上，它是普通解释系统加了“以平台码为翻译目标和运行目标”之后的解释系统。这里详述：

Jit到底什么东西

JIT有什么特别，你依然得联系到标准的三段编译原理过程来解释它，即translate->link->run ,jit也被称为jit compiler,jit interpter，那么jit顾名思议，jitcompiler=jit translator，是语言都有的，所以这不是它的特征，而我们立马发现其有jit interpter，如果说到现在为止提到了解释，编译系统和这个JIT三大语言系统,这个新事物名称就证明它属于偏向类似解释系统。也许这个名称可以再加一条，即jit=jit to native。 上面解释了普通interpter是什么原理的，而基于jit的interpter除了它模拟了普通解释系统后端的最小单元编译和执行机制(或者它类似TCC编译过程十分快显得像解释器)，它还显得有那么一点特殊之处，即它针对平台码，无论是RAW CPP源码，还是二进制DLL符号，它都能实现compile和连接不费时，和运行期直接查找到符号， 这二者都齐了所以jit才表现得像个解释器。 可见，JIT它也并没有带来新的东西，原理上Jit只是接通了生成到native code同时为目标翻译码和执行码的东西，别无它。是传统解释套装并列的部件。实际上也并没有破坏编译原理的外观。它依然是一整套语言系统，只不过在流程上它可以与解释语言并列，通过开启JIT的选项转到使用JIT套件（再重复一次，jit它代表整个JIT语言系统，同时包含前后端，需在三段式环境下理解它）。

Cling中的jit

Cling基于clang+llvm,最主要的意义是其jit interpter机制，即解释语言那套+针对平台码平台符号，作为一个“特殊了一点的”解释器存在：即Cling jit后端可以jit to native,emit native function，其AST可以按NODE喂给运行器。 而其实JIT不仅限这个功能，cling依赖于jit能call into native libs不需binding，是JIT本身的功能，（当然，这事先需要在编译成二进制时保留JIT所需的符号，rdynamic causes symbols to be exported even though this is not a lib ,it Keep symbols for JIT resolution 这些DLL都是符号解析级的动态可载入组件，受操作系统DLL实现支持）。这种组件免除了link。因为组件都是在运行期link的。而从DLL中获得符号，就能省去compile->tranlate的需要。所以这也不脱离cling jit的作为解释系统的产品外观范围。所以，这个面向DLL的特性，一定意义上可当成，cling jit视DLL为raw cpp code组件（暂时你可以把这里和接下来的组件当成脚本源码文件一类的东西来理解），为“源码”（而普通解释器面向解释单元，解释模块，是真实可视的源码组件，JIT只工作在二进制导出符号层，能作CPP源码和二进制混合编程）。因为它视平台DLL为组件，因此能做到动态持续从“DLL源码”(DLL其中源码实际并不可见，这里说的是其中符号类似llvm jit眼中的“源码”，被它当成了组件）加载符号和运行. 这点意义上，宠统来说，JIT就是一个更高级的”DLL“机制而已，使其还可以直接视库为开发件，具备组件的特性。这是后话更多组件的情况将在第三部分介绍。

clingrootsys原理剖析(2):the pme

本文关键字：**cern root,rint,root6 cling,clang cling**

动态语言中的动态类型语言

一般会误以为动态语言就是解释语言。因为解释系统能动态执行代码也往往意味着其被归为动态语言。但实际上动态语言现在最常见的技术形式反而是一种称为“动态类型的动态语言”，它往往依赖前端而不是后端。这造成的结果是：静态语言系统和经典的编译->运行系统也能产生“动态语言”。 比如在编译器实现中，实际上类型系统可以提出元类型，封装有类型的基本信息，然后喂给后端的是元类型/对象产生的子类型/子对象树的形式就可以 – 一个较原来复杂一点的数据结构，然后其它过程保持不变喂给后端。运行期的类型信息照样在运行期可保留甚至动态演变。这难道不是动态语言吗？ （这种逻辑也可以工作在库级和工具链级，即语言系统实现的外部，比如pme，它的实现只要binding就可以了—而binding实际上是另一种编译器意义上的前端翻译，就行了，而执行时是现成的，比如qtmoc为qtcpp源码模式生成的字典，这就是为什么binding也能生成一种动态语言系统，后端执行时可以是静态的，但主要喂给它的是如PEM这样的业已包含类型系统–元类型系统，会将类型系统保持到运行期就可以了） 你可能会为编译过程的这些种种感到迷惑，但实际上这里面所有的技术，跟传统静态编译语言系统 – 你学到的最简单的编译原理实现，是一个外观的。而编译前端，解释前端，binding，这三个词都包含了转译。目标码可以是平台码也可以是中间码，供运行。所有这些，都不能改变所有用编译原理实现的语言系统共享同样的产品外观（都有该有的部分，只是呈现出了不同的形式）。回到系列文章第一篇的文头那些话，用这些通读所有复杂语言系统的定性你才能不致迷糊。

Pme为静态语言模拟了动态语言特征

Pme, poperty,method,event，是对反射机制的一种实现，加了反射机制实际上在静态类型之上加了一门新的语言，和库级运行时，可在运行时查询到整个活动对象树，及每个类的场景图，成员属性。PME是组件的一种通用实现方式。 而且，这种兼具object io特性的pme机制，可为运行时通过外在的编辑器改变objectlvl的程序逻辑提供了可能。借助PME组件的并持久化将成员属性什么的持久到XML等载体。下一次需要时又可加载进来（仅限代码中的成员数据）。 而只有在cling/rootsys这种大环境中，pme与JIT合作，这种动态性才得到最佳发挥，DLL加载终于通过JIT，变成了语言系统的功能。而不再停留在作为操作系统的一种机制，而pme模块可以动态加载，这在开发上体现为，pme DLL体内的逻辑是固定的。可改变的程序逻辑是DLL外的那部分。那个定制脚本部分和你的APP逻辑部分，可以是JIT CPP源码（这里除了PME支持的代码中的成员数据外，整个代码都可持久，The interpreted and JITted C++ shares the same virtual memory space as the app itsel。）。有没有感觉有脚本的样子？直到这里，cling/rootsys开始有了同时能模拟了脚本语言式的解释效果和动态加载效果，可谓叹绝。

Cling/rootsys中的pme字典生成

如果说cling call into raw dll靠的是符号，受JIT和操作系统DLL机制支持，而call into PME模块靠字典信息非符号，动态加载pme组件和发现组件里的OBJ树需要PME支持，因此需要自实现。这是为何呢 这实际上最重要的还是因为jit call into native libs只是使符合变得可见而已。而加载DLL中的资源，是普通的native langsys的功能，于是作为仅仅是执行引擎向OS的传手，llvm也可以而已。但其rootsys libs的pme是库级的，cling代码可以直接call into native libs，但不能call into rootsys libs，因为它们是有pme dicts as bindings的（不能直接通过加载的方式使其为cling可见必须通过对cling的封装变成rootcling才可以）。因此，cling除了jit，和pme，还需要一个手动或自动添加字典binding信息使pme module和普通raw c dll(那种业已解析为简单符号可直接加载的模块)变得一样。的方式，比如一个手动/自动DICT生成器。生成到raw cpp code传给LLVM后端。 带着这些观点，继续来看看cling/rootsys中的对应物，即其对pme模块的支持-aclic。 ACliC只是将pme模块形成加了pme字典的dll的工具。cling is faster building compiled code, but ACLiC can reuse it. Cling产生jit码是高速编译器产生的类解释器效果，而aclic可以在库级反射层面利用它。前面提过，将raw cpp改造成类似qtcpp的新语言系统，所有模块必须经过字典封装，这个过程也称binding。Rootsys即是这样的一门新语言系统。 在实现上，aliac是以patch cling的方式加上去到rootcling的。因此.L ++的方式产生so文件，只用于为pme模块产生dict 模块并链接好。

附：对于qtcling，有mocng，是基于clang的qt moc.exe重实现，这也可以作为cling的patch组件，类似aliac的方式加到qtcling,使之具备发现源码中有pme逻辑即自动生成dict模块的功能，to give it the ability to produce “automic dict generator for qt extending cpp syntaxs” 来完成对整个qt libs的从源码级的重新编译封装，最终完成整个qtcling语言系统的构建。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

terracling前端metalangsys后端uniform backend免编程binding生成式语言系统设想

本文关键字： 用**terra**打造更科学的**js,cpp,**用**lua+c**分离式模拟**JS**。**terra**真正的终身语言，**terra**最接近编译原理的元语言**,cling based terra**：前后端都可免编程**binding**生成的元语言体系

在前面《语言终极选型》《实践终极选型》系列中我们谈过"one for all",即一体化，终身语言的概念，联系到在《编程新手真言》第一部分我们一直在寻找某种lddlang，，比如在整书第二部分我们谈到过最熟悉的CPP，它本身就是一门多范式语言。甚至针对于那些要求更具动态性的类型系统，qt也通过扩展库和工具moc的方式组成了一门qtcpp langsys的小语言。在《JS完全》中我们那里我们谈到过js一门可用于web栈全栈开发的语言甚至进化到H5和mobile,desktop native，通常被称为某种一体化web,mobile,native语言的代表，而且它用函数模拟过程式和OO的方式也是某种“语法”一体化的表现，这此都是语言内部层次的极大化。

而后来我们跳出了单语言单生态的考虑着眼于一些综合语言系统，又有了新的发现，比如在《发布qtcling》时我们提到cling和rootsys，它是llvm based，整合了cpp,c scripting且免binding的一支，是真正实现全C系中一体化的，，在《发布monosys》中我们提到过java,net等统一后端语言，顾名思议它带有一体化语言后端的特征，还有一些利用translator compiler而非独立编译器实现的统一后端往往是针对具体语言的，，像vala这种，还有综合类型像zephir,rust这些，动静态结合带let等，他们都带有超越它们固有领域的极大化整合和改造倾向。

可是细细分析就会发现任何语言体系的极大化（通用化）其实正是它们企图在其内包含各种DSL的过程，在bcxszy part2中提到，发明各种DSL是软件模式之一，自古以来，DSL就是如上提出各种语言内机制或各种脚本语言、新语言/语言体系来完成的，即它们都是DSL技术的子集。

且它们统统都有局限。

比如，CPP是语言内的范型整合，且面向C系单生态的。而QT面向CPP也未免太单生态，其利用pme和type reflection扩展类型系统也隐喻着对它其它的扩展是secondary的事情。而js虽然在语法和问题域都有不俗的整合度，然而它终究是构筑于ECMA单标准和单语言实现上的，qtcling非免没有包括非C系，而直接rootsys也是单生态的，它binding库组成新cling语言体系的能力是巨大的，因为它是先库后binding出来的pyroot等，llvm也有免后端特点，然而cling/rootsys前端只有clang系，monosys它不是免虚拟机的，C#只能统一后端不能有真正的免binding前端生成器。C#有语言内编译服务然而缺少真正的语言内支持本语言开发生成器的特点。转换器往往固定而混合动静态类型语言往往扩展不到其它前端和后端的组合。

总之，他们共同的特征：离一门更合理的语言构造和使用方法的跨度始终跨越太大，或缺陷太明显了，这种“更合理，更整合”的设想就是接下来我希望得到的，我希望有一种 : 不致于破坏现有事实语言多生态的既有事实，又能巧妙地整合这些，还要能以传统发明语言的方式(而普通的像语言内提供类型修饰的机制终究有点捉襟见肘，比如py饰符)能在这个原始层次加以扩展的接口，且能在本语言内完成，形成一对多的，最好一主多宿（相对于主，宿可以动态拔插以扩展）的解决方案。

而以上所有这些语言，这些所有的特点，不能按常规方法，支持真正的元编程和代码自动生成。那么，用现有的方案改造/整合行不行？如果单语言的缺陷总是那么明显，那么或许至少二门语言组成的混合语言是另外一种出路（当然它也要以合理的方式支持尽可能多的扩展支持我上面讲到的合理，最小免修正整合)。

从1ddlang到anylang,从single lang到DSL mixed langsys

归纳一下：一种更为颇为科学的设想要求 --- 我们需要一种真正纳入到支持用户DSL创造的一体化语言体系。。最科学的，我们要保留现有的各种不同运行时，再促成一个真正的可用的统一后端，如colinux as xaas的东西，这里是onelang as langsys。

它至少要是某种统一后端或前端的东西，用户可以以优雅自然的方式来产生新语言，新语言作为这个新语言体系的可拔插部件， 真正允许用户用这二门元语言(as host)整合自己需要的语言作为guest language as language comopent or lib plugin

比如我们的目标至少要是：能用这种语言开发任意zend php等的逻辑，使得一种语言，任意既有无修改后端。能粘起来工作，比如我可使用cling写php的wp程序。

目前最大的整合方案如monosys和llvm based langsys like cling/rootsys是最接近我想法的东西，然而它们往往足够强大没有太多围绕它们的项目，最后我找到了terra：

可以说,在terra下，llvm回归了底层虚拟机的原来意味。它是这些语言的统一后端。

它的3个类比物：用function发明DSL，类js用function创造OO体系，用codegen生成代码，类CPP的模板。vala等等

在我强化过后的terra设想中，利用cling作统一metalang替换lua，负责生成各种具体前端语言。可以使得，lua是host,terra是guest，guest可以扩展的方式meta programming变身多种语言或某语言的复合体。，存在一主一guest二门体系，主可用来metaprogramming，客用来兼容后端，就如colinux一样。下面详述：

terra:a multiple stage langsys that can micic js,cpp,etc..

terra的基本描述：

Terra is a low-level system programming language that is embedded in and meta-programmed by the Lua programming language: We use LLVM to compile Terra code since it can JIT-compile its intermediate representation directly to machine code. To implement backwards compatibility with C, we use Clang,a C front-end that is part of LLVM. Clang is used to compile the C code into LLVM and generate Terra function wrappers that will invoke the C code when called.

最基本的考究，就是lua作为转换器前端，将代码转成terra表示，然后运行terra，因为terra是llvm based的，而转换器是lua based的，所以前后端一个主转换一个主运行，兼有写法上的高效和运行时的效率，

理解路径1)：a dynamic language for controlling the LLVM

整个langys，它利用动态语言的头，本地语言的尾，组成一个混合前后端(初看它比较像c preprocesser+vala translator这样的东西)，其实像llvm这种带了jit又带了中间码，又带了native code gen的东西，可以做到混合前后端部件，这样可以免VM且达到本地码的效率，借且llvm，达到与cling与C模块abi linking的效果(Terra code is JIT compiled to machine code when the function is first called)。terra其实是另外一种cling+clang

理解路径2)：a dynamic language for controlling the LLVM -> using a dynamic language to control code generation of static one

multiple stage programming，它是metaprogramming中code generate中的技术。它在一些数值编程领域非常流行。其本质：

Multi-stage programming (MSP) is a variety of metaprogramming in which compilation is divided into a series of intermediate phases, allowing typesafe run-time code generation.Statically defined types are used to verify that dynamically constructed types are valid and do not violate the type system.

A multi-stage program is one that involves the generation, compilation, and execution of code, all inside the same process

The staged programming of Terra from Lua,,,注意是从terra到lua的staging，这二者的相互欠入性来说，分清二种语言，terra core和full terra langsys，一份具体的用该语言写的代码是terra-lua代码。

因为事实上lua跟C是完全不同的二种语言，它们的interportable终究只是他们的外在属性，内在它们是不可交流的，那么这二者是如何联接起来的呢？技术本质和过程到底如何？这二门语言有共同词法作用域所以就保证了这二门语言无缝交互性（interpreter），极力使得他们像一门语言（中的变量作用域处理部分），除此之外，其它二门语言不同的部分，依然是原本二门独立语言该有的(c/terra和lua有着极广泛的互融合性interopable)。基本上平时你用lua编程(lua)，涉及到control terra to codegen的那部分用c(terra)/lua

理解路径3)：a dynamic language for controlling the LLVM -> using a dynamic language to control code generation of static one -> a low-level system programming language embedded in and meta-programmed by Lua

统一后的terra langsys其实本质只是：它们在metaprogramming这个层次上是结合且统一的。

an important application for MSP is the implementation of domain-specific languages,languages can be implemented in a variety of ways,for examples,a language can be implemented by an interpreter or by a compiler.

we think that having DSLs share the same lexical environment during compilation will open up more opportunities for interoperability between different languages

那么，terra是如何利用lua+c作为元，来生成其它任意中高级语言支持的呢？这是因为lua的数据结构恰好支持重造一门语言所需的那些metaprogramming特性，比如一级类型有function支持，有table支持AST表示，等等，在前面说到js是一种直接可在AST上写程序的语言。

最好的举例是先说js再说CPP

js:

在以前介绍js的时候，我们就谈过functional language就是AST语言，因为它可以直接在语法树上写程序，现在terra，进一步把它清希化了，结合type reflection这一切做到了极限。它可以用函数推导产生各种过程式和OO，从lua模拟C/cpp

cpp:

其实，它也是某种预处理器的极大化，如针对CPP的。完全可以用lua本身来模拟生成更好更统一的预处理，它很像用C写编译器时，这个C是动态的而已。用本语言在本语言的一个实现内写扩展，且加载为库。当然在terra中是lua代码。

还比如，用来实现类CPP的类型系统。

Objectoriented frameworks usually offer a type introspection or reflection capability to discover information about types at run-time. Metaphor allows this reflection system to be incorporated into the language’s staging constructs, thus allowing the generation of code based on the structure of types – a common application for code generation in these environments.

这也是为什么仅需c+lua，而不是需要是c/cpp+lua，因为CPP整个都可以是被扩展出来的。这比直接在llvm上构筑clang++好，因为我们可以用c+lua的terra来打造架构更科学的terra版cpp

terracling，架构更科学，前端改造为CLING based，后端保持llvm based的terra

那么能不能将terra改造成cling based 呢？即用cling+c替换lua+terra，因为C是支持函数指针为一级类型的。这样做的好处是：直接用C系作metalang控制语言，生成扩展的cpp,py,php等等。

加了metaprogramming特性的cling+llvm，它的前后端都可以免额外编程工作自动生成。比如语言前端的parse等可以binding c dll生成，再对接到后端，库也可以C模块方式集进来，可以直接用zend php或是llvm上的php实现如roadsend php等等

意义：

cling作为脚本语言对生成C代码自动化生成过程非常好，且扩展出来的CPP同属C系，因此metaprogramming可以分散 CPP式将所有范式集中一门语言的特点(比如把c++ template弄成简单的一种语言特性Terra’s type reflection allows the creation of class-systems as libraries.)，这样可以避免QT将PME支持聚集到另外一个QTcpp中去。也可以将CPP预处理以更科学的架构导入，而且可以通过编程和程序内的方法引入，而不是预作为库服务如reflection，也不是作为基础件如编译前端等，而不是像CPP一样杂合到一门复合语言内。

可以直接binding已有程序语言实现，无论是llvm based或llvm non based都可以，只要以c dll存在即可。

还有，其实lua与openresty,gbc这些我前面提出的东西结合紧。整个lua+c揭示了几乎二门必学语言的事实，terra像极了linux的架构，可以类linux一样产生各种封装的变体/新语言系统。且易定制/易自然定制。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

terra++ - 一种中心稳定，可扩展的devops可编程语言系统

本文关键字：**devops**可编程的语言系统。**programmable language**，可编程容器和可编程语言系统,**c++ as terra++**

在前面《Terracling:前端metalangsys后端uniformbackend的免binding语言》，我们简单聚焦其语言性质讨论了terralang，主要说到其几个区别性本质：

1,它里面有三种语言c,lua/terra，为什么把后二者放一起？因为使用整个terralang，顶层上还是使用lua来作开发的，terra是配合写被lua调用的函数区块的（我们用terralang指代整个terra语言系统，terra指代三种语言中的一种），这种terra里有可jitted编译运行的c，c通过terra被lua用于meta programmed。新语言terra实际上是multistage中的中间层，即stage1->lua,stage2->terra,stage3->c，terralang能做到这一层主要是因为terra用了llvm+clang，terra用他们构建了terra实现，用他们jitted本地代码，无须binding。所以实际上是clang实现的lowlevel c family语言，且它能lua混编和元编，，主要你还是使用lua，这就像C混编汇编主要使用c只在某些地方需嵌入汇编一样。

2，由于上述机理，它能用lua+terra的方式模拟C++的好多模板语法和复杂语法如预处理，将这些用语言套语言的方式来实现，分散到各种DSL支持文件中terra++，语言用库来扩展的思想在这里得到真正的具现（而实际上C++之父的这个思想在现今的C++实现上越做越复杂），且解决问题的方法使用的是更集成更传统的编译原理方式。不单C++，它甚至能模拟出各种其它基于C的DSL，并将这些lang tech,class libs持久为文件，作为关键字使用。它比cern cling这种更有扩展性，后者只是专注C++，而追赶C++核心的多次变化的cling实际上加大了对C系语言的学习成本，而lua和C都很稳定且语言特性十分接近相通。-----terralang实际只是DSL工具，混编工具+这个工具+这个工具下的二种语言：lua+c,语言还是极其简单和稳定的lua,c核心-----平常并不需要写terra，语言的发明者才需要写terra，我们只使用lua,或c,在发布涉及到terra实现的东西的时候，我们要么在C中内嵌lua，要么在lua中直接调用terra，要么发布纯粹的terra .o,.lib文件，无须binding也不需要嵌入这个几十M的llvm+clang实现不像cling scripting一样只能发布llvmclang本身。。你可以用lua+C写无关terra的直接应用，也可以用lua+terra写可编程的语言扩展，始终围绕着C核心作扩展却用的另外一种语言lua来写应用。

PS：围绕着C核心作扩展却用的另外一种语言lua来写应用这句话揭示了terralang如果能用tinyc代替lua这种就最好了，选择lua是因为lua与c最接近，lua就是c的脚本的精确对应化，tinyc没有类terra的扩展，提供不了metaprogramming c的那些功能，所以改成terracling会更好。

好了，我们来说terralang的另外一个巨大优点，在前面，我们见识过多种复合语言系统，无论它们以什么形式出现，像typescript,一些带let关键字动静态语言结合的语言系统，《elm liveeditor》这些devops语言，一些生成器为核心的multistage语言。他们都少了一些至关重要的特性，工具全集成特性和可编程特性，。terralang刚好都可拿来补足这些。

工具视角看terralang为IDE all in one

在我们讨论过的《elm liveeditor》这种语言中，我们想得到一种基础功能集成化（visual debugger,lang lib, auto build,ide all in one，甚至集成instant demo deploy and play）的东西，这其实就是devops适用语言的那些需求，那么terralang可以是这种语言么?我们说，它实际上是比live elm editor还要集成的工具和语言系统。

这是因为terra用代码形式来控制语言来发明和强化语言。terra允许以更灵活的方式控制语言本身，实现自我控制和强化，，这种可编程的特性允许它在部署过程中生成新语言或提供新的语言设施，那么这有什么好处呢。

如我们见过的语言系统通常都会带一个或复杂或强大或简单的IDE，提供可视为编辑和调试的功能，但这些外围实现始终是工具，terralang本身可被编程，它就可用语言本身作为ide（比如发明一门DSL实现IDE）,这些在terralang中直接有支持。VS elm显式IDE化为工具其发挥不了极佳的灵活性,terra这种非常统一和强大。又由于terra语言本身可被编程，在语言内调试在线调试这样的东西可以统一化成语言内编程手法实现，所以我们完全可用工具视角可以treate terralang as a ide。

而发布上，如上所述，由于我们发布的时候可以按lua或c的方式来任何一者standalone式发布，terralang as a langsys只需作为一个开发时的工具不必发布出去。负责这种功能的是运行时。一些虚拟机语言和面向对象语言更是需要发布巨大的运行时和类库，terra都可以分开发布他们或集成发布都可以，自由度更高。

而整个IDE加运行时的集成，这对强化terralang为devops又有帮助。

视terralang为可持续集成的CI工具,devops可编程的语言系统

可编程的工具或语言体系一体，可语言内通过语言扩展，这些上面都说了，在强调devops的云时代，集成化，可编程的langsys有什么意义呢？，利于CI这怎么说呢？

我们知道docker是一种可编程容器，我们知道docker之所以支持devops，是因为它的编排文件和构建文件可以用shell脚本和yml这种来书写，是programmable的，为什么本地沙盒容器成为不了CI容器。因为它不可编程，不可代码在线构建，作为数据打包和作为程序生成始终是二个不同的过程。所以我们同时需要一种可编程的语言系统，可编程=DEVOPS。

terra即是这样的语言系统。因为它的每个DSL都可以是一个langtechs,一个类，一个问题抽象，一份练习codesnippter，这种可持续集成的性质，可以让任何规模大小的工程都得到持续集成。

在教学上,terra做到了二门稳定的最小语言为核心集，它的DSL特性还能允许你轻易能展开《通过terralang学C++》的课题。langtechs和domainproblems lib,practise codesnippter可以归类到细化的dsl中,是天然的可扩展语言系统的典型，用应用plugin的方式扩展语言。而现在的语言系统，没有一种能达到terra的这种效果(而很多其它用语言发明语言的方式始终停留在库级，或一些有限的关键字和语法级，如python语法糖,js函数直接在语法树上写程序，cpp的预处理和模板元编程特性等。。)这些都太中心化。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

利用**terralang**实现**terrapp(1):**深刻理解其工作原理和方法论

本文关键字：发明自己的语言，可**lua**扩展的语言系统，用库发明语言**vs**用语言发明语言**,**是**toolchain**语言也是**app**生产语言，全生态语言

整个xaas系列文章中，我们编译组装了自己的dbcolinux，现在我们要发明自己的语言，如果说linux生态允许我们很大自由地定制一个完善的OS，那么terralang类似地，就是一个给我们定制语言系统的工具,它的设计目的之一就是这个，而且经过了专门的去复杂化。我们可以在少量知识和实践储备的前提下，copy-paste式地发明自己的简单语言系统

我们要定制的语言系统是cpp ,在前面关于terralang的文章中《terracling》《terra++》，我们多次提到了这个目的，我们现在要利用terra最小核心实现一个真正可用的cpp —— 其实terra已经是个cpp了，但是它离真正可用的cpp还是有点差距，因为它只定义了最小语言扩展所需用到的核心(稍后谈到它的正交类型和语句设计)，所以严格来说它只是个cplus，而现在我们要在这个基础上造一个接近标准CPP的dsl for terralang，terralang.org的网站中，也有大量相关资料专门提到这个专门课题，这成为我们的所有just enough起点。

与**lua**接通**,**更像**c**，设计语言的语言

在这之前，我们要了解terralang作为元语言工具的这种能力的深层原因，我们按前述文章的做法，称呼 terralang为整个terralangsys，而terra是单个语言：

首先，terra应该被称为terrac（Terra是c系的，terra is c extender towards Lua），但是它却更适合被称为terra/lua的，这是为什么呢，因为terralangsys中，terra被设计成与lua混编，那么terra究竟是发明语言的语言，还是用来被日常使用的语言。都可以，甚至terra当然也可以作为独立语言（A scripting-language with high-performance extensions.....An embedded JIT-compiler for building languages.....A stand-alone low-level language....），但是大部分情况下，它以terra/lua方式被使用。lua中欠入terra---terralang主要用来metaprogramming或写dsl。严格来说，terralang中只有terra,lua二种语言。

> terralang中有二种独立的语言，它们的runtime是并存的，也是可以离线分别使用的。Terra compiler is part of the Lua runtime，Terra executes in a separate environment: Terra code runs independently of the Lua runtime.在lua运行期对应terra的编译期，而terra的运行期是OS的,并不绑定整个terralangsys
>
> 但是在语法上和使用上，terra/lua却是一体的。甚至是紧密联系的，就像一种语言——即terralang。
>
> terra是用来代替和改进C的，它有自己的类型系统（Terra’s types are similar to C’s），代码系统，和所有一门语言应该有的那些，本来如果C有terra需要的与lua交互的一切，那么C直接就是terra了，但是因为没有，所以对C的增强就演化成了terra，——我们把它称为cplus,在terra中日常编程是lua/terra混编，所以实际上可以视为lua/terra+cplus混编，另一方面，terra也是对lua的扩展，Terra expressions are an extension of the Lua language. The Terra C API extends Lua’s API with a set of Terra-specific functions,一句话，terra使lua像c,使c像lua
>
> 那么，为什么要这么做呢？因为terralang被设计成将C与lua良好交互，inter-operate，它必须要及其简单，一开始就考虑进免binding交互，luajit有ffi可以以简单的方式与C交互，其实luajit ffi对于C的封装已经接近terra了，terra也是采用它的作法，既然Lua/c已经可以，为什么还要出一个terra/lua，这是因为其远远不够。比如它缺少元编程和multiple stage programming那些，所以terra增进和增加了这些，它有哪些方面使得它一面像C，一面又接通lua
>
> 其实现原理和准则是什么呢？诚然多语言体系有极可能优秀的多的方案，但terralang只选取了适合自己的部分。
>
> 它实现了一个预处理器，使得设计期和执行期分离，Separating compile-time and runtime-time environments，这二个期，它们交互的单元非常小 —— 一个terra function或type这使得语言系统高效清希，二个期都有完备的语言系统，却能承接，服务于一体化结果。这个过程是这样的：
>
> 在设计期和编译期，The preprocessor parses the text, building an AST for each Terra function. It then replaces the Terra function text with a call to specialize the Terra function in the local environment. This constructor takes as arguments the parsed AST, as well as a Lua closure that captures the local lexical environment. 使得在语法上，terra/lua是一体的。写出来的.t程序将包含多语言源码，但在编译期shared lexical scope and different execution environments因为词法作用域上，双方的types彼此可见。随时相互转化。这种简化是刻意而为之的：基于lua ffi，它使来自terra的所有语言元素作为lua的值存在，简化了这二门语言在各个期的所有conversation，稍后会谈到。
>
> 在运行期，it will call into an internal library that actually constructs and returns the specialized Terra function. The preprocessed code is then passed to the Lua interpreter to load. Terra code is compiled when a Terra function is typechecked the first time it is run. We use LLVM to compile Terra code since it can JIT-compile its intermediate representation directly to machine code. Clang is used to compile the C code into LLVM and generate Terra function wrappers that will invoke the C code when called.—— 最终在底层，combined terra/lua会是一堆lua和本地码混合的东西，没有terra的任何成份。compiler, generated code, and runtime of a DSL. 最终这三者都被打通形成一体语言。
>
> 这样的努力带来的效果就是——比起其它多语言系统，作为多语言免binding交互的典范，Terra and Lua represent one possible design for a two-language system.而其它元语言，如py的meta object,采用的并非语言控制语言，而是将这一切集成在单语言内，这样不够灵活也不够强大-只有同时一门编译语言一门脚本语言才能自然而然地兼有多语言的优点，比如llvm可以将性能关键部分生成为machine code保证性能。还有.net的clr封装的unmanaged 模块等。无一不带天然缺陷。

下面详细说下其中都有些东西，刚好的可互操作的类型正交系统，元编程能力，和multiple stage->DSL是一条因果路径的。首先是其类型系统。

类型和语句的转化，正交化类型转化设计,type-directed staging approach，和共享词法域

二门语言的交互，要看具体语言类型的是否typechecking属性，作用域，生命期，内存布局这些。

在luajit的ffi时代就有这种思想，这也是lua的设计初衷———lua被设计成与c正交,user defined struct可以是一段C,既然有table，为什么不直接用它表达对象呢，有函数，为什么不能用它表达其它呢，这就是正交。terra对于c类型设计也是正交的，——— 所以，它是一种刚好设计语言的语言。不多不少，刚好正交。比如它的类型与C的那些很对应，比如它还有pointer，struct—它还加入了更多的面向metaprogramming stage programming的方面。比如，built-in tables that make it easy to manage structures like ASTs and graphs, which are frequently used in DSL transformations.

这种类型是用来作元编程的（至于要不要写DSL那是另外一回事），Terra’s type system is a form of meta-object protocol，与其直接把类型设计为面向app生产，terralang的类型被设计成天然在某个staged compilation流程中被使用，这就是ecotype，它允许程序员—— 这里主要是元编程者或DSL发明者，在类型被具化之前，干预类型的内存布局行为等操作，

首先来看二门语言和各个期的typechecking。我们知道terra是需要的，lua并不需要，为了保证语言的一体化。就需要协调 - between compiler, generated code, and runtime of a DSL关于类型可能带来的问题。In Terra (and reflected in Terra Core), we perform specialization eagerly (as soon as a Terra function or quotation is defined), while we perform typechecking and linking lazily (only when a function is called, or is referred to by another function being called). 这样就可以规避矛盾的发生。

所以，terra的编译期运行期分离，正交化设计的类型系统，都是为了简化，变态简化，只是为了最终使DSL的发明更简单一点。最后的重头戏来了，这就是共享词法域，它是服务于让terra写dsl变态简化的另外一个方面。

下面来说这个共享词法作用域，它最终使分离的东西在语法层形成一门叫terralang的东西。去除了在传统stage programming放置操作符（quotation,escape,etc..）的需求。这里需要结合terralang的生成代码机制和元编程机制讲，我们还没有谈到terralang的生成代码的机制，上面只是粗略提到，总而言之，从语法到最终语言形态上，二种共存的语言要处理的问题不可绕过（between compiler, generated code, and runtime of a DSL）作用域就是一方面，The shared lexical environment makes it possible to organize Terra functions in the Lua environment, and refer to them directly from Terra code without explicit escape expressions. 甚至去除了namespace的调用需要。To further reduce the need for escape expressions, we also treat lookups into nested Lua tables of the form x.id1.id2...idn (where id1...idn are valid entries in nested Lua tables) as if they were escaped. This syntactic sugar allows Terra code to refer to functions organized into Lua tables (e.g., std.malloc), removing the need for an explicit namespace mechanism in Terra.

对于作用域，在staging的各个阶段和形态上并同时透明地跨越lua,terra，都保持了正交。这样本节开头提到的关于类型的矛盾，就都解决了。而且我们始终要记住：变态简化是设计一开始就maintain的原则。

下面来综合讲述metaprogramming到DSL的原理。

生成式元编程，和DSL发明

其实元编程不一定DSL，我们可以仅使用terralang作meta programming。这也是大部分情况下我们使用terralang的情景。这节标题中的元编程可以放到前一标是，只是terralang的元编程更适合写DSL而已，与后者结合更为增益。

在前面说过，其实无论那种语言，写代码都是扩展语言写“DSL”，库也是语言的扩展。----- 要么面向问题要么面向扩展语言本身要么面向APP，只是terralang使得这种DSL具现化。

多种语言有多种元化代码生成的手段，CPP是基于模板的— 它就是一种编译期的类型特化过程，，还有跨语言元化的，这就是stage programming,在stage programming的范筹里，有一些生成操作符，上面宠统描述terralang的元编程技术中也有提到过。

为动态运行期生成代码的能力。这相当于cpp template programming的多语言版本。虽然terralang是真真实实地使用llvm，而不是同样图灵完备的基于template技术的CPP实现———。但这二者很类似，这2个过程可以类比。

前面的类型正交设计和共享词法域已经为terra做了大部分工作。ease这里的复杂度也许只是关于DSL本身的。我们知道，如果一旦涉及到编译原理，就很复杂，这里的复杂度几乎减无可减，因为涉及到数据结构，词法分析，递交等系统编程，但terralang也有自己的方案，它用了一种叫Pratt Parsers的方法,What makes this parsing technique interesting is that the syntax is defined in terms of tokens instead of grammar rules，再加上terralang也会专门性地提供一些造语言的API，极大简化了发明DSL的难度。这些发明出来的dsl作为terra的模块存在。并非欠入terra，而是欠入到lua。

这就是说，为terralang扩展DSL，做法上也很简单，完全去除了需要专门发明词法解析器等语言设施的需求。虽然它使用了编译原理，但是它使用的是更有限更简单的有限集，因为所有语言的区别只是一些对于terralang的入口table。

Terralang的DSL能力尤为珍贵，因为它允许任何人发明语言，terra扩展语言本来就设计成要开放成给任何人使用。

———————

Terra是为c系增加一门scripting的最佳实践。它把c改造为Lua like，也把lua/terra，改造为c like scripting，当然它的主要作用在于发明DSL，——以后，我们可以在这个c/terra上发展cpp，除了拥有一个全态全包的语言体系，甚至还能有一个高性能的库Terra’s type reflection allows the creation of class-systems as libraries。

Terra,，the programmable cp with Lua,,just like programmable nginx with Lua,有了terralang，从此什么语言都有了，terralang既是app生产语言，又是toolchain语言。还是shell语言，makefile语言等

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一种新的DSL生成和通用语言框架：pypy

本文关键字：**DSL框架和自动化生成工具,pypy as dsl framework and jit framework**

在《bcxszy》 part2中提到，发明各种DSL一直是软件工程模式之一，在那里，我们还一直在找寻某种1ddlang和1dddev方案 --- 更多更好的DSL和统一的语言系统并不矛盾，如《编程语言选型通史》《编程实践选型通史》所讲，问题的根源是不断出现新的问题域要求语言系统足够领域通用，最终要导向到语言选型问题，语言选型其实是一个涉及到编程所有的领域的活动(不光问题还要平台还有考虑人的入阶曲线且能将现有的codebase轻易迁移过来)，而理想的状态是提出一种语言或混合系统language for all，它可以集成一种强大简便的DSL方案，能胜任其它语言能做的事而不带有任何先天缺陷,保持同一生态不断层。关于DSL和这种lang for all的设想，有语言内(langtechs,lib,qtmoc工具链+pme)和语言外(方法论,toolset lvl，各种语言标准emscirpt)的一系列手段。

这些在我以前langsys系列文章中都不断涉及：

> 在《发布odoo8》时我们谈到主从语言,lua+c,or py+cpp----这也是传统语言选型的经典标准---也是初级标准，注意到因为大凡脚本语言系统，为了兼顾效率和考虑进通用目的，都是binding c extensions--这也是为新语言快速建库的方法，不过当这类语言这样做的时候，它实际上也在承认它是靠补丁工作的，如果满足于同时使用二门语言，其实这是完全可以的，不过这像极了学会了使用C还要学会汇编一样，这样的转换始终带有历史遗痕和存在断层，仅支持从库级和语言技法级，扩展级去扩展DSL支持，这种语言通常用cffi这样的库支持，这样的语言代表是py,php,etc..
>
> 而.net,java这样的语言系统，它提出了统一后端,语言服务也是运行时和库，可以作为API调用,有DSL支持，即使所有语言可以无缝interspect，且它提倡将原生扩展做进纯粹managed runtime，如c#重写forms而不依赖native forms， 这实际上是要统一clr上的生态使之脱离对本地的依赖，只是将一切放在虚拟机的运行时里，明显效率是个问题。
>
> 在上述二种语言生态充分饱和之后，效率和大语言思维终于被提上日程，业界要做的，只能在上述这二个基础上改造，这首先是给常见的语言如php,py增加JIT，近年来，JIT发展迅速，为了将上述php,py语言发展为真正的通用语言奋斗（再在这上面扩展DSL，这就是上面说的langone+dsl化）。就是jit系统+混合语言系统：
>
> 在前面《发布terracling》时，我们提到从CPP到混合语言系统选型出现的历史现象回顾，且做了一个小型的关于混合语言的综合。联系到更早在《发布qtcling》时我们谈到llvm的jit原理和它独立于传统编译器的事实，这里我们看到LLVM作为一个DSL和JIT工具框架，它的强大实用性，要理解它，可拿它与clr,jvm这样的东西类比，因为它们都支持多语言前端和统一后端且都有JIT，有强大的可比性，然而他们的区别却微小不易发现而至关重要，足以影响到他们归类在不同的流派：
>
> clr,jvm是虚拟机流派，llvm是运行时流派，llvm后端就是一个to native os的运行时没有VM。它没有为任何语言设置一个解释部件只是一路翻译，最后仅执行jit后的代码，这样的代码已是jitted to native的,这使得它的效率是很高的。一句话，llvm的统一后端和其运行时就是免虚拟机且JIT的没有虚拟机和解释部件，它允许从C系开始制造前端这是它与clr,jvm不一样的地方（后者如果要写C扩展是用虚拟机routing原生代码），它产生的新DSL和语言，可以与原生C语言系统的模块在IR级交互可直接调用这类模块无须binding，且由于jit是类解释系统的在线执行机制，因此可以支持产生qtcling as c++ script这样的语言。 而jvm,clr无非就是虚拟机+解释，而jvm,clr同样有jit，对于中间表示（字节码或AST）和执行结果，他们都提供了一个可写多语言前端为任一语言集成jit的框架，JIT和虚拟机都是黑盒（或者半JIT半解释，或者纯JIT），有没有黑盒这个是没有差异的，---------- 产生差异的，恰恰是这个黑盒内部是如何运作的：llvm是分析字节码然后以jit方式快速编译且执行.新语言不需要VM运行只须带LLVM运行时，而clr,jvm的jit默认是解释系统加jit协同工作的，任何语言结果必须带虚拟机。llvm是all code defaultly routed to llvm,clr 是一切routed to vm first,then selectively to jit
>
> 纯走JIT，完全可以使二门语言需要混合的部分走统一的工具链流程和开发发布。不必涉及到专门的binding过程。这就维护了统一生态不必断层。而统一后端加统一都走jit，可以使得多语言天然可交互。
>
> 除了qtcling，在前面《发布terracling》时，并提到一种原型terracling as toolkit，qtcling和terralang都是典型的llvm based jit mixable langsys，而terracling更先进，因为它提出了用lua metaprogramming terra(c)产生新语言。使得选型二门中心语言，其它DSL都可以以库的方式被plugin进来，然而其方法主要还是用lua结合编译原理编程产生新的语言parser..
>
> 最后，我们来归纳一下开发界对DSL逻辑和dsl语言发明的所有招数：有interlanguage interopt,binding interface tool,Preprocess,template,meta programming,partical evaluation,src2src translator（甚至到支持全部语言的haxe）,common runtime,anonation,js functional metaprogramming,regluar compiler和编译原理，还有jit+mixable mutiple langsys

如果LLVM是这么好的框架，那么不出所料，在LLVM上直接做PY，PHP的JIT应该会收到好的效果，然而，事实上llvm被尝试用于将很多传统语言如php,py装配新的jit，然而收到的实际效果却不好。google的unshadow和dropbox的pyston都反响不佳， 据说它对前端语言的要求最好是非动态类型的，这次我们碰到了pypy, 它不光是更好的LLVM，且它也面向多语言走JIT没断层，vs terracling它也有metaprogramming+编译原理出新语言系统的能力且以语言内机制自动完成jit部分，没错，它其实是另外一种更强大的langone+DSL框架，单PYPY是语言实现，整个PyPy语言系统就是一个编译器框架,完成可以拿来跟llvm+terracling结合效果相比，与llvm这种忠实地从0开始再造轮子的方法相比,pypy似乎更聪明一点，它重用轮子，它极力促成的结果是：使py真正变得通用化且集成DSL开发机制而能使产生的语言巧妙免除binding c的那些场景，因为它走的是更聪明的jit：

pypy:更合理的metaprogramming和auto jitted backend,像terracling一样装配了一个语言产生器

在制造DSL和混合语言的手段当中，有一种是语言转换器，就是src2src translator，pypy的原理:1）The RPython translator takes RPython code and converts it to a chosen lower-level language, most commonly C. 2）Unlike Python, RPython is a statically-typed language,,,,

> 也即，pypy使用了rpython(rescricted)这门python的子集来自实现，但是要注意rpython其实离python十W八千里了，与其说它是python，其实不如说它是另外一门语言，它就是实现产生DSL的元语言。类似llvm的前端部分，terralang的lua metaprogramming部分。如支持clang实现的那部分。pypy就是用rpython实现的python语言的前端部分和解析部分，虽然rPython不是完整的Python，但用rPython写的这个Python实现却是可以解释完整的Python语言。，，这句话亮了，作为用户我们面向的，始终仅是最后一部分，即可以解释的完整的PY。
>
> 这里的特点在哪里呢？它有三层，即使有这么多层，且全程用py或rpy实现，也丝毫不影响性能。注意这里的1）pypy代码，2）rpython表示，和3）jit运行代码。用户写的是pypy代码，不运行，rpython仅用于工具链表示，产生的c代码才进入到运行，而jit过后才实际运行。离线的始终是那些预置化过程和面向用户代码的部分。运行的始终是jit过后和优化过程的C代码部分和原生代码部分。
>
> 然后其它的事就交给rpython强大的工具链，这是rpython第二部分，这就是我们说到的DSL产生工具框架和JIT产生器，类似LLVM统一后端。亮点是它是一个src2src 转换器，目前Pypy只实现了Python到C的编译，也就是说编译器的后端实现了直接转成了机器码。然后对这部分代码作JIT,它的JIT是jitted to c相当于免VM的native code了，逻辑是：PyPy 的編譯工具鏈可以靜態地對 RPython 代碼做類型推導。類型推導是編譯的步驟中相當重要的一步，做了類型推導之後，就可以靜態地把 RPython 的代碼直接翻譯成 C 代碼然後靜態編譯了。 用户不需要发明JIT，并不需要自己实现JIT编译器，工具链支持自动生成ＪＩＴ，只要按照PyPy框架的指引，用RPython实现一个带有足够annotation的解释器，就自动得到了高性能的带JIT编译器的实现。
>
> 这里的特点又在哪里呢？不可忽视的地方在于， 按需执行的JIT - 对特定的函数做修饰，然后动态的把它们编译成机器码并切换到使用c扩展。这种做法的好处是，重要的事情说三遍，写解释器，得到JIT编译器。写解释器，得到JIT编译器。写解释器，得到JIT编译器。当有人想写一个新的编程语言的实现时，只要在PyPy框架下用RPython编写一个对应上面(2)的语言解释器，就可以借助作为meta-compiler的(3)的部分，得到一个能支持把(1)JIT编译到机器码的高性能实现。
>
> 且它还实现了运行时优化器。
>
> PyPy's powerful abstractions make it the most flexible Python implementation. It has nearly 200 configuration options, which vary from selecting different garbage collector implementations to altering parameters of various translation optimizations. ----- 这有点类似kernel的busybox配置过程了。
>
> 最后说它的缺点，由于pypy实际上不面向混合来自C语言的扩展，PyPy有很弱的C语言扩展性。它支持C语言扩展，但是比Python本身的速度还慢。但是要说这是PYPY的缺点分明是无理取闹嘛，直接将逻辑写在纯PYPY上，开启JIT就好了。

pypy:更合理的断层,jitted2c使其跨系统和应用二栈，jitted2js使其跨任意全栈

最后，pypy jitted to c的特点使得pypy可以跨系统和应用二栈，因为传统C系开发发布的那些领域就代表了整个系统开发栈。而其实rpython可以编译到js的，这使得py码代码迁移到web是一个巨大的帮助，可以将整个pypy编译为pypy.js放在浏览器中，如js有asm.js产品，可以将浏览器中的js+css+html通过模板编程控制手段化为py+css+html，这样py就web全栈了。

这是我以前在《发布jupyter》中提出的设想，结合它作为在线IDE，可以完全将所有前后端编码+逻辑调试放在浏览器端。

拟下文是《web开发发布的极大化：一套以浏览器和paas为中心技术的可视全栈开发调试工具，支持自动适配任何领域demo》，增补《编程实践选型通史》part2 web部分，那文说得不够透彻，预计深入讲解一下，可能自圆其说的往往都会有点玄，抱歉抱歉。。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在tinycolinux上编译pypy和hippyvm

本文关键字：在**tinycolinux**上编译**pypy**和**hippyvm,pypy**上的**php,hippyvm on rpython, hippyvm vs phalanger**

在《发布wordpress on .net》时我们谈到clr上的php实现，即phalanger，在《pypy:一种新的DSL框架》中我们说到pypy才是真正的vmlangsys allinone，因为它走JIT，使来自原生c语言的扩展变得不再必要。在PYPY上就能实现效率和生态全包，这才是不拖泥带水最正统的VM编程语言体系，比CLR，JVM正统多了：就如同汇编之后进入os编程的时代C是作为高一阶语言生成机器码汇编的一样，在新时代VM和脚本时代的混合语言中py与c即是这样的关系，把这个自动化过程做进语言系统的pypy即是这样的大语言思维方案。

> 在那里我们还提到，比起clr,jvm，它也具有多语言前端和统一后端，实际上这个统一后端是统一工具（这里并没有一个像CLR一样的统一后端），把rpy当工具set，把其它语言当前端，我们可以在rpy工具链上实现多种语言，且带来更多更好的新功效：比如在《pypy:一种新的DSL框架》末尾我们提到它可以促成py与js的混编，在后端使用PY生成浏览器中中的JS。
>
> 实际上该如何理解py和rpy的关系？rpy是工具，也是语言(静态py子集)，它与py共同作用，py+rpy是作为元语言系统来生成其它语言系统的，py又是这个关系中rpy的metaprogramming lang（实际上就是rpy受py调用而已,相当于terralang中的lua+terra，只不过它们是非C的且兼容的PY语法版本。），因为这二者使用基本一样的语法。所以使发明新语言的过程变得简单，可以使用PY+RPY生成多种前端（虽然多种语言其实地位是平等的，但用于产生新语言时，还是用倾向于用PY，因为它是RPY上的主语言，类CLR上的主C#）。而用它们来生成PYPY时，就等同于说，PY生成了自己（假设我们用cpy+rpython生成pypy，这个pypy跟cpy是兼容的）。整个过程rpython只是工具，并不影响我们得到一个原生的pypy。即生成得到的pypy是最终jitted to c的，其实跟cpy是一样的c based python实现性能上一点不差还较Cpy快。一般说pypy就是pypy实现+rpy工具链。源码和生成结果都是这样。接下来会看到。

而pypy上也是有php实现的，作为例子，我们来介绍pypy的编译，顺便介绍其上多语言 - 一个PHP实现hippyvm。hippyvm也是PyHyp的一部分，PyHyp is a composition of PyPy and HippyVM., a single file can contain multiple fragments of PHP and Python code，当然我们本文主要讲编译，并不会过多涉及到混编的内容。

我的环境是tinycolinux+cpy2.7.14+gcc481+php561

准备工作

由于编译过程会使用到大量内存，官方说大约2.5G内存时间上大约总是会用1.5个小时以上，我使用的是1G云主机，只能时间换空间了，先开启3G交换文件内存，但实测在使用交换文件1.5G左右，编译进程会很慢，形似卡住，实际上也卡住了。换成4G内存的云主机照样开启3G交换内存，才最终通过编译，/tmp下生成的临时文件倒是不大，毕竟，预处理多久都可以，但是会因为内存少而卡住，这个就不能接受了.

按如下在tinycolinux上开启交换内存：

```
sudo dd if=/dev/zero of=/swapfile bs=1024k count=3072 创建大小为3g交换文件
sudo mkswap /swapfile
```

临时开启:sudo swapon /swapfile

或者做到/etc/fstab中:/swapfile none swap defaults 0 0

除了bootstrap py,编译过程中会用到php-cli，我们分别用这样的参数来编译，记得下载对应缺失的4.x tcz pkgs然后重启生效：

cd Python-2.7.14 && sudo ./configure && sudo make && sudo make install

(以上需expat2,bzip2,libffi,ssl,curses这几个事先安好重启)

cd php-5.6.31 && sudo ./configure --enable-fpm --enable-zip --enable-mbstring --with-mysql=mysqlnd --with-mysqli=mysqlnd --with-pdo-mysql=mysqlnd --with-zlib --with-gd --with-curl --with-jpeg-dir=/usr/local CFLAGS=-D_FILE_OFFSET_BITS=64 CXXFLAGS=-D_FILE_OFFSET_BITS=64 --enable-opcache --with-openssl -with-openssl-dir=/usr/local/include/openssl && sudo make && sudo make install

(jpeg6在4.x tcz mirror中无对应tcz，需要自行下载jpeg-6b源码以--enable-static --enable-shared configure并编译出，因为hippy编译中会用到php,py的bin和lib，默认在/usr/local下，图方便所以不需加--prefix参数)

添加py支持：cpython:get-pip.py,pycparse,hippyvm src/requires.txt中的东西

然后准备hippy的源码,github/hippyvm/hippyvm，按readme.md检出https://bitbucket.org/pypy/pypy/，形成可用的源码结构，我这里是2018.2.15左右都是最新的源码。注意这里都选取默认branch，不要检出我们上面提到的PyHyp相关的brands，即https://github.com/hippyvm/hippyvm/pypy_bridge，按其readme.md,它对应的pypy在bitbuket的bitbucket.org/softdevteam/pypy-hippy-bridge/，它使用的是它修改了的pypy源码,这个修改的pypybridge也需要修改的bridge的hippyvm/pypy_bridge.

因为不支持prefix且默认是就地生成，所以把整个源码目录移到/usr/local/hippy，处理一下源码，把targetthispy.py移到hippy src根下,然后将hippy目录中的hippy也移到src root中。将goal/targetpypystandalone.py也移到src root下,这样就基本准备妥当了

编译

其实未编译就能运行，称为untranslated，非jit版本。是cpython逻辑，就跟rpy一样，这个比普通的cpy还慢。直接python ./bin或pypyinteractive.py就可以了，而我们要得到的是-Ojit的版本

源码目录中那个rpython就是工具链，在源码中rpy虽然是源码形式，但一直也是可立即待用的工具。，你可以把rpy想象成一堆py工具，用cpy或pypy执行它，会产生C的本地代码（translated），这跟C项目通过makefile产生exe是一个道理只不过这是py的构建系统。且这里是产生编译器和语言套件。

而lib_py,lib_pypy，就是pypy生成后支持的额外平台模块,lib_py是纯py的，lib_pypy是pypy支持的独有模块

好了，先构建pypy。

cd /usr/local/hippy

sudo python ./rpython/bin/rpython --continuation -Ojit targetpypystandalone.py

漫长编译过程结束后（期间因为经常会出错，重新编译不会续编，所以上面 --continuation），最后结束，看到可以分为几个步骤，

annotate,rtype,pyjitpl,backendopt,stackcheckinsertion,database,source,compile,build_cffi

2核4G内存+3G交换内存下，除了pyjitpl和stackcheckinsertion用了约半小时，其它都是十分钟之内，耗时最大的是stackcheckinsertion，

编译好的pypy可以删除rpy，但是最好还是保留，因为根本就不大，接下来会看到。因为更能清希化：pypy就是pypy实现+rpy的事实。

如果不开启jit即不带-Ojit，那么编译好后的pypy实际上就是一个普通pypy解释器，就跟上面untranlated的cpy直接运行一样（非C，且未带jit）甚至更慢。至于rpy，你是在开头和结尾都不必由用户涉及的，只在编译pypy的过程中出现（作为工具链控制产生过程和目标pypy解释器选型），只对采用rpy来发明新语言的用户有意义。

然后用高速的pypy还构建hippy,这个pypy-c就是translated版本且with jit的pypy

sudo pypy-c ./rpython/bin/rpython --continuation -Ojit targetthispy.py

完工，同样是jit的新语言-php！

当然目前这个hippyvm是很初级的，wordpress都运行不了，未来把OC移殖其上，当pypy源码中集成了php或其它语言前端，其实它也完全可当成语言的裁剪器如busybox

---

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

elmlang：一种编码和可视化调试支持内置的语言系统

本文关键字：编码和可视化调试支持内置的语言系统，以浏览器技术化的**IDE**和**WEB APP**为中心的可视化程序调试语言系统,让编程和调试装配到浏览器,为每个**APP**装配一个开发时高级可视**debugger**支持

不可否认的是，即使编程语言的技法再“抽象”，库再领域完善，工具再完备（况且工具也不完备，我们稍后会谈到），它们还是没做到尽量对每个人都像。且能适配任何应用，编程依然是专业人士的事。

更高层的“艺术化编程手段”是一种出路，在《bcxszy》part 2中，我们归纳了从工程和艺术层面使编程高级化的手段，比如提出更多语言，即语言DSL化脚本化（针对语言技法的改进或增强也是一种DSL化,pme,etc..），提出更多领域语义（针对领域和库，及适配人的建模过程），提出更多模式（超越语言但能被任何语言都实现的patterns,frameworks)来对接人和人对领域的统一理解，及提出更多工具（可视化编程,rad）这些手段，这些手段间又是相互联系的，比如rad可能跟pme有关以后者为技术实现基础。又都往往需要集中这些，使之能体现到一种高级综合语言系统实现中---因为我们总是最终依赖一门语言为中心的各种选型，开发总是与具体语言和它的生态绑定，因为没有人再倾向于发明轮子。

然而直到现在，编程还是尤为专业化。编程复杂度，专业程序依然如此之高源于一个基本的事实：这是因为业界注重于解决问题为先，怎么复杂怎么来，似乎走了一种过度抽象的道路，治标不治本来的历史遗留复杂度，甚至于上面提到的方方面面：

首先拿语言技法来讲，

> 降低编程复杂度需要涉及到众多因素，统一到以语言为中心这首先是为语言提供来自其它语言的优点--这就是DSL化和标准化，是把多种语言集成或打散的过程，（它要解决的矛盾是各种语言本身的技法生态带来的不够用或聚集过于复杂化，比如语言技法是针对写法的抽象，需要分散与集成写法）。
>
> 例子一是多典范语言设计，但它其实是很无力的设计-因为业务逻辑膨胀的速度远超语言系统对接起来要求提供的那些，比如硬件的原子操作都能形化为语言的关键字这只能使得语言似乎太聚集和膨胀，各种语言标准只能将语言实现堆得越来越复杂。将一切堆到库级，用库来设计，也避免不了语言技法级本来就存在的问题，这是因为库属于那个语言的生态，跳出这个生态除非在其它语言中有等价实现才有可能，这依然是分裂主义，我们需要共用一个生态的多种语言。
>
> 然后它们又提出了.net,java共后端多前端这样的东西，然而这不是真正的langinone，除了用了vm有效率问题与与nativedev断层之外。不是说.netfx的多前端不可以分散化各种langtech，而是 --- 它们本来就支离破碎，OO这个东西其实也有问题（它虽然免去了要求人们去理解底层的方式但是仅是复用层面如此---面向被使用者，但它是一种过程式范式的附加而不是替换，这就造出了新东西，要求人们同时理解过程和OO，而OOP三重机制比较繁复），各种DP advanced oop techs，framework只是越来越多，做的决不是减法。它建立在专业的底层上，却要求更专业的人来理解和应用它。那么，有没有一种统一的范式，可以类过程式又能可选地实现为OO呢（后面我们谈到函数式）
>
> 类似多语言系统的观点在我以前的文章中随处可见，针对它我们也提出过混合语言系统设想。
>
> 而工具上，语言的高级化和底层不变又形成了矛盾，因为debug的时候我们从来都是通过在某个编辑器和IDE中，追踪底层的执行frame的，所有现在能看到语言编译或解释实现都是这个套路的，而coding过程中DEBUG过程比重几乎甚至是超过coding的专注于讨论这个其实比讨论OO更有意义（后面我们谈到语言内置高级化的可视调试，且适配到per app）。
>
> 一句话，我们并没有处理好底层的简化工作--对应于我们需要的现实的映射，依然还在一方面极大地依赖计算机的方式来处理建模的事情一方面对抗没有一套统一方案真正可用的困难，这造成了与人的断层。编程是不是一种跟跳舞一样容易的事情，完全取决于编程者专业度需要专业人员处理各种细节。一句话，即使有语言作代理对接领域，即使这个代理不断亲切化，也有人难于越过这个学习门槛。

抽象永远是正确的，但关键是如何去统一和抽象，对于过度设计该尽力避免，决不应该乱统一和抽象，即业界总是造出新东西，而不知道造出可替代的东西。我们需要从多个方面去重新考虑语言和根本上与此为中心的开发设计，提出较《bcxszy》pt2中艺术化编程手段更先进的手段，甚至使编程变得不像编程反而正是唯一出路。elm-lang正是这样的语言系统设计。

下面结合elm-lang来一一说明，每条都对应elm的一个特性和其对于传统过度设计的修正性设计：

首先来看elm-lang是一种什么东西:

elm-lang A delightful language for reliable webapps.Generate JavaScript with great performance and no runtime exceptions.

it use FRP,Functional reactive programming (FRP) is a programming paradigm for reactive programming (asynchronous dataflow programming) using the building blocks of functional programming (e.g. map, reduce, filter).

compiled to js的DSL设计与jsintro特性

在《发布terralang》《发布pypy》这些文章中我们不断提到高级混合语言系统，和可裁剪的语言系统如linux kernel般可裁剪的思想，它们主要是从DSL化，和语言内技法这些方面去抽象语言---往更简单更统一更强大的语言系统方向抽象。

而elm-lang正是这种translier为技术的混合语言系统，它用函数式静态系统haskell生成标准的动态语言js，类pypy静态生成py这种动态语言（haskell,js vs rpy-pypy），只是前者少了JIT而且也不是作为DSL工具链框架存在，排除这些它们是相同的。

rpy最典型的用途是jitted一套语言工具的前端，但实际上，它可以jitted out任意的一段rpy写出来的codesnippter或项目,因为它是通用jitter，，所以在作与js的交互时，我们仅需要compiled codesnippter to js，不需要把整个pypy.js带过去(必须事先让js engine binding到pypy.dll)。而elm-lang也可compiled to js这就与web生态接轨了，统一了WEB全栈开发。elm-lang+它的各种库就是以webapp开发为中心的，因为它具有jsintero因此可用于在服务端生成eml后缀的服务端程序就如同php内嵌js一样,jupyter之于nb一样，所以elm就是一个服务端编程语言---生成html+js页面，而这对客户端也是一样的：在IDE选择上，jsintero特性也为它寄宿于chrome+nodejs提供了条件。

treate oo as paradism pattern but not explicit langtech

elm-lang被设计成用于替代js+各种库如react,redux全家桶，将web开发各种范式由JS+库的生态尽力整合到一门语言elm的langtech上。这主要是因为它的frp特性。

先不说FPR，单就函数式语言本身来说，函数式极其类似C过程式，这也就是为什么JS代码看起来很亲切的原因，是一种能兼容兼顾过程机器抽象和OO人类抽象的机制。且做到了像C一样能成就copywrite式编程for newbies.高层能直接对接webappstack的gui-view,db-model,io-msg，低层能像C过程一样copywrite又通过函数式天然保有OO的功效。而oo is evil 的地方在于它高于函数的class封装显式化了，而这是不需要的， 满布class的一份源码不能成就copywrite式编程 --- 而这是新手入阶的基本范式，而functional的js可以保留这种能力同样可以非常自然和容易地进行OO，不需要涉及到OO和OOP对传统过程式的侵入。elm对接copywrite式的demos修改式编程非常好因为它类似C过程式。

况且，它还有它的可视化的调试器inside支持，这又是一个极为重要的加成：

uniform coding and debuging support inside langsys design：为每个APP装配一个开发时高级可视debugger支持

为每个APP装配一个开发时高级debugger支持，elm-lang从工具的debug层面探求使webapp开发变得变成极简的艺术手段：

debug.elm-lang.org有一个online debug它为每一个APP写一段debug逻辑，这得益于elm-lang的frp+pme（pme来源OO，所谓的响应式编程很多是基于event bus或observer的，但在frp的elm中用了更强大更原生的函数范式界的机制来代替）,这就是所谓的debugging support inside languagesystem，类unittest，就是那种边写代码边额外写测试用测试驱动编程的过程，这里是用DEBUG辅助编码无错。

elm-lang采用的function reactive programming完全得益于语言本身的FPR属性，,,,这个编程范式能带来可交互特性。fpr编程全称为interactive fpr programming，这个名字对支持elm的coding,debuging一体化，即语言内可视调试是本质上的帮助。debugger，且是visual的---当然这是不同于traditional step,step into,step over,stack frame view的低级debug，，它是面向具体app的高级debug，具体来说是hot swap支持，它类似传统QT gui creator的pme。

可视化和与IDE DEBUGER的结合则是PME这种东西的功劳。谈到PME,这是VB和QT这些GUI工具的事，因为它们都是交互相关的(注意交互二字)，所以在这里是共用的，交互DEBUG。 这在debug.elm-lang.org中的《How Elm makes this possible:Purity,Immutability,Functional Reactive Programming》中有述。

这其实也是类war3 we的东西。如果说war3 we采用的是面向war3 game ,gui editor+jass自动生成，那么之于elm-lang per app debug可以面向任何app，它就是debug ide+elm生成的js。

uniform webiz client and ide app ecosystem:让编程和调试装配到浏览器

在《编程实践选型:part编程的最高境界是什么》中我们谈到WEB的极大化和浏览器对webapp开发和浏览的双重支持，但是那里我们并没有说完，WEB是一个在领域逻辑上把开发发布做到极简化的工程样例。W3C主导下的WEB，各种标准而不是工具，使得WEB处于设计泥坑不断提出设计和反设计，比如抛弃了如XHTML这样的东西。所以有时标准不如一套简洁有利工具的支持。

由于elm-lang的第1，3特性，它使得基于elm-lang开发一个webize ide成为现实。这使得WEBAPP的开发真正武装到了牙齿。

远程调试器使得一个编辑器不用跟本地语言绑定，所以使得一个简单的text编辑器也能成为语言开发工具。甚至于一个浏览器加一个插件的方式，如php xdebug+chrome插件。

与elm-lang关联的另一个项目-lighttable(nfw)就是这样做的。它可以实现在一个IDE中同时调试js和php，比如让chrome os中的chrome+nodejs核心同时成为WEB浏览客户端，和真正的多语言可视调试IDE --- 这统统都是web技术。这样WEB开发发布就真正极大化了。

这篇文章是我《web开发发布的极大化：一套以浏览器和paas为中心技术的可视全栈开发调试工具，支持自动适配任何领域demo》系列中的一篇，这文也是对，下文也许是在tinycolinux编译chrome一文后再源码编译出这样一个ide或《把owncloud当git空间管理项目codesnippters》，因为elm-lang类jupyter使得codesnippter is post，即用编程的方法直接服务于写博客，可谓真正的uniform code and debug，且oc可以当成一个极好的codesnippter project workspace存储基地，好了，结文。

关注我。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

why elmlang:最简最安全的full ola stack的终身webappdev语言选型

本文关键字：**react**，**stdlib inside lang**，前端开发的几种技术方向和潮流，**elm editor vs vscode+elmplugin,haskell vs elmlang**， 打造类**tcent cloudbase**的碎片化**cloudapp** 云原生**devdeploy**环境和私有小程序平台,把**vscode**当成**dsl trigger editor**生成器可配核心，一种可调调试难度与核心裁剪的专业语言与**ide**。使用**elm lang as vistual trigger lang:Time Traveling Debugger,**适合**webdev**的函数式语言或函数式命令式混合语言**,elmlang vs py,vs js**

我们知道，站在最高层来看，开发是一种综合考虑“语言，问题，人和设计”的工程行为，而软件作为产出，是一种抽象app栈架构，一般appstack包含有网络，IO，持久，用户界面这些子栈堆叠而成(最后是业务逻辑)，这其中ui是代表，它代表一种appmodel，如desktop appmodel使用本地图形渲染出来的gui,web appmodel使用page ui。

------ 在开发行为中，语言处在中心方案域位置（问题，设计是方案，人，语言是实现域），，而在开发中方案域和目标域的重要特征是它们都会发生变化(软件整个银弹讨论都是关于维护软件生命期变动的讨论)，语言也处在最易变动的中心位置，本身作为变化体串联起这种变化：比如，一般情况下，我们都是用通用语言+库扩展来解决问题的，局限于使用语言本身提供的映射手段来解决新出现的问题，但当语言本身的抽象手段不够用时，我们往往将其DSL化：提出一种新语言或对现存语言综合增强改造，这种语言多变适配开发可以任意方向发展，往往一个价值观，一个硬件特性，对方案和实现领的任何巨细修补行为，都可以成为主导发明一种新语言特性的理由，，不信去看网上的一份《Every Language Fixes Something》and only something，

------ 包括语言在内，我们可以把处理“语言技法问题，抽象问题，映射人的设计需求”这些综合问题在内的开发体做成中间件appframework，它属于解决问题的“一次大而全的设计和实现”，一般封装直到建立appstack为止（给开发者最大关注业务逻辑的时间而不是浪费在appstack本身上），比如qt就是一种框架（qt为cpp增加了moc和event,signal，和gui框架，已经把app变成了qtcpp，因为给cpp增加库级已经不能不能满足qt的需求，所以它干脆改造工具链加入moc处理，对应上面提到的“这是一种对问题人语言的综合处理反映到语言层的处理”，一般框架具体到产生一个APP结构兼实现业逻辑的功能。，qt的库包含了对桌面APP的实现，一起组成了整个qt框架），而高级的框架它超越语言和应用stack，甚至考虑到对OS改造。全程提供full ola（os,langsys,appstack） stack增强或改造，如plan9。（后者叫架构更合适）。

这其中，appstack,appmodel,applang,appframework都是有机联系的，定义APP的UI，即appstack中的ui部分尤为重要，如果说appstack决定一种APP类型，往往也决定开发APP选用的这门语言的选型和框架架构结构，拿上面的qtcpp桌面开发来说，面向对象+消息往往是QTcpp这种语言的ui方案,desktop ui天然就是一个widget组件树存在互动和管理这些互动的组件，这类OO语言如java能很好handle，，事情到了web前端领域，作为最能体现编程技术潮流的前沿地方,框架和语言的变化在这里变动更剧烈，也存在着关于UI开发延伸到语言的变化的类似进程：html页面不是desktop gui那套，而是一种js api视角的dom，js有部分函数特性函数是一级对象，functional特性的js本身能很好处理这种dom，OO反而不必（正如在函数语言中一等函数类型可以轻易表达很多设计模式问题一样，传统的OO命令式语言需要借助接口这类复杂的东西来达成），最初的人们满足于简单的js+html，然后，需求更新了：后来人们想得到更强大的js前端页面，提出了jquery.js扩展操作dom，实现了在一次bs交互中动态更新某节点的功能，这仅仅是往js语言扩展了一个库。到后来，他们想得到universal desktop and mobile/webapp(非universal platform app,非苹果发布了m1芯片和新的arm macbook和big sur 16，真正将ios做到了intel上的APP。)，让web页面做到跟桌面GUI一样强大，以及single page app这类东西的时候，实际上，对页面的动态性和性能优先已经到了一个很高的地步，各种xx.js扩展和不断的ecma标准化已经很难框定这类东西了，人们发现原来流行纯函数式语言时代的某些语言和机制可能才是处理这种逻辑(fullwebstackdev)的最佳方式（js跟纯函数式语言还是有点区别），比如函数语言的FRP可以适配更多 dom ui方面的新需求：因而越来越多的框架应运而生，angular，react ，vue 三大开源框架，他们最大限度解决了前端页面加载速度 及更新速度问题（稍后会谈到使用virtualdom和diff算法）。当然，也方便书写和开发。于是，当前端可以以统一的frp react方式被解决时，这种手段也被包含在后端形成mvc，形成full webstack级的框架方案。

> 纯函数语言与ml系函数语言： 我们知道，冯氏机代码的眼中只有指令和数据，这二者是平等的东西，存在时空可置换原则，冯氏机中的编程语言也有二派。一种是命令式，一种是声明式，函数语言一般属声明式语言，前者主要针对代码中的数据作数据方向的创新抽象，并整合到语言，后者主要是函数式主言这类针对代码结构进行语言发明创新方向的成就 同样的道理发生在软件抽象的任何地方正如脚本和数据一个是时间一个是空间，你可以用脚本编码数据本身，也可以把脚本作为数据文件供调用，，（还比如devops把部署自动化了，也是命令脚本化了，资源文件化了，这样就为开发上云准备了条件（因为做成了可被调用的“代码”形式在云上的异地容器运行完成，我们终端只需获得结果），再加上云ide我们就彻底不需要在本地维持一个开发环境。） 纯函数语言，偏函数，高阶函数是函数语言的语言级区别于命令式的概念，纯函数的特点是：无状态，无副作用，无关时序，幂等（无论调用多少次，结果相同），这些数学上的东西可以实践成函数语言，但完美表达数学概念的语言机制实际上作为实用工具来使用会显得很原始，这种语言稳健，精美的类型系统能自由安全处理，推导数据类型的正确性和安全性（不过这样也有drawback，函数调用深度debug比较麻烦打印出中间结果比较费劲）。写的时候能调通的代码基本没什么bug，自带并发你完全不用考虑那些诸如死锁，线程池等等的复杂概念，天然适合分布式和容错系统，但学习成本和适用范围比命令式语言要高很多（像Monads和Functors这样的东西很难理解，特别是没有数学背景），还有比如它的代码中不带数据和变量（传数也不便，monads用管道类似的操作来传数），没有循环，只能用作为超集的递归代替。而我们现在使用的命令式则走的是另一个相反的反面，比较亲和大众但是副作用和学习曲线也相当巨大，并发需要涉及到加解锁。 命令式这就是我们现在正在流行的C系派生出来的大支，而函数式语言曾经在学术界非常流行，近几年也有一些流行但都偏小众。如一些年前曾经流行的erlang也被选为最不推荐学习的语言之一 因此，现在的函数式语言，大部分是混合命令式和函数式，在纯函数语言中加入了一些命令式的成份来稀释它使它变得实用，同时保留函数式语言的大部分优点。保留了函数式针对代码结构而不是数据化方向创新的成果，使得函数语言中的函数变成传统过程式（子过程函数）的合理延伸，，比如haskell和Ocaml就是这样一种ml思想(基于ml 1975年metalanguage的派生)而elmlang就是haskell的子集。

在前面《elmlang：一种编码和可视化调试支持内置的语言系统》和《在群晖docker上装elmlang可视调试编码器ellie》中，我们重点讲到elmlang作为函数语言用于webappdev的一些FRP原理和基础及elmlang的visual debuggable特性，本文则是从elmlang用于fullwebstackdev语言选型方面的细节和合理性角度讲的,即以下几个方面介绍elmlang：

1）。elmlang是一种基于ml系的haskell派生的语言。本身是用正确的语言干正确的事的典范，elmlang在语言期就给webapp保证的很多webapp的专用开发特性，如pure views, referential transparency, immutable data and controlled side effects，所以在扩展级和用户级可以做很少工作就可以写出webapp。注：除了这些，elmlang还提供运行期无错和time可回溯的debug（这些都来自于母体haskell的函数语言特性）。

2）。elmlang是封装了从applang到appframework一条龙的东西。elmlang的核心和标准库包括了开发webapp的必要成份，elmlang现在整体库的体量达到了1.5k。一切都是all at hand，但elm与py这类batteryincluded不同，因为elm不定位于通用语言，所以它是webdev的终身语言。注：elmlang app本身却没有提出一个appstack的web框架之类的东西，它推荐使用elixer+phoinex。ellie正是使用的这一架构。

下面详细解析：

elmlang:用正确的语言干正确的事。webdev = elmlang as functional programming language，vs python,vs rust

我们这里在讨论作为方案的语言对应于解决问题的问题域的手段，要知道，语言能完成的事情都一样，因为它们都是图灵完备的，但成本和抽象手段都不一样，人们显然更追求用最简单的语言能提供的最直接的映射手段去表达一个问题的解决：

这种简便性体现之一是代码可以写得更少更直观更符合自然语言：比如py的成功，它用字母代替一些逻辑符合，它用很多语法糖generator造出很多方便的写法。这样下来，一个.py文件显得很清爽，甚至一个没有写过程序的新手，都能在他的意念里推导出这些代码的用意（前提是它熟悉一些C过程式的东西），这是提高语言流行度一个最得分的选项（保证更少的代码是一个重要方面，或者说，python可以用其它语言的经验来“推导”出其灵活的用法。这种灵活正是为了能在python中少写代码，使代码显得简单。主要集在中语句和写法上的创新上。python是一门从核心创新的语言（集中在过程式和单条语句上）。而不是像其它语言一样从外部和范型级别增加新东西扩充功能去创新。）当然，其实py本身是很复杂度的，python的简单是上帝视角下比较了多门语言的不便之后，发现了py其实是一门灵活胶水特性之后的认知开始的（弃它的缺陷不顾，如全局gil，无jit慢）新手学py，还是一样难的（甚至py越是灵活就越显蒙逼）。而且它作为胶水是batteryfullincluded不像lua只配了一个语言核心。。Guido开发Python的初衷：开发者需要一种高级脚本语言，在易用性和功能性之间取得平衡、在处理复杂逻辑时没有Unix shell的限制；能够像C语言那样，全面调用计算机的功能接口，又可以像shell那样，可以轻松的编程。但是python对于shell的包容实际上没有perl好。曾经一度perl在胶水，shell,web方面的表现都很出色，perl -pe "可以代替sed awk,grep等一系列用到re的地方，而且它的进程管理也能代代替ps等（至于shell的pipeline，perl语言中的pipeline是默认的行为，是以源码级函数或子过程为单位。当然它不是以组件级执行进程获得输入输出为单位，后者perl也有（当然没有shell的那种方便,这句话其实也说明perl的代码中好多符号，其实是优点而不是缺点））,perl是web开发原语内置的语言，因为它的自语言和文本处理和正则很强大这些刚好是web domain的逻辑。但当过程和命令式远远还有深意可挖时，perl追赶OO和webframework让它迷失了方向，让php和python分别蚕食了web和胶水，shell领域。2015 年发布了 Perl6，perl6相对perl5是一门全新的语言，，“Perl 6” 已经被现在称为 Raku 的产品所采用。现在，perl7准备在perl5的基础上原路继续发展。

elmlang它使用了haskell的子集。一开始就定位于简化代码而设计，跟py的目的有异曲同工之妙，elmlang的docs中，有很多函数语言haskell的成份和章节。还有天然的Debuggable属性：它维持了一个Predictable State Container for JS Apps 这使得it easy to trace when, where, why, and how your application's state changed.促成"time-travel debugging", and even send complete error reports to a server.

除此之外它是安全的，它与rust语言安全类型与并发的异曲同工之处：比如error处理和pattern matching(实际上，异常和其他数据结构一样是同质的，不是有特殊语法规则和能力的元素。因此也可以复用其他用于组合和处理数据结构的方法论和工具。elmlang运行期免错，是因为它有函数类型的maybetype，A Maybe type may or may not contain a value, we must handle the case where the value is Nothing)。当然这种脱离语言本质区别所属比较是没有意义的，但它们在维持安全的外观上是很相似的。

最后，当然还是那句话：对于没有任何基础的初学者来说，任何语言都一样绝对难，只是相对不难。elmlang和py就是相对不难的。

elmlang是封装了从applang到appframework一条龙的生态级东西。vs js

新语言直接提供了新问题所需要的元素，集成了更简便的表达问题域的方式，即通用语言核心就集成了专用问题域方案而尽量不借用扩展库来完成没有多少其它乱七八糟的东西。“通用语言干啥啥都行，干啥啥干不好”，而DSL或许才是出路（如上，这种DSL不是扩展一下通用语言的库就行了，而是重新设计整个生态或再度框架化一次最优方案）

拿js来说，js的出现和选型本身就是一种DSL化，它是异步IO语言，是web前端的良好选型，浏览器中的js只绑定dom，不是通用语言，后端的nodejs算是通用语言。这二种运行环境中,较其它语言js其实最本质的特点是它的IO。Ryan在发明nodejs时，他评估了很多种高级语言，发现很多语言虽然同时提供了同步IO和异步IO，但是开发人员一旦用了同步IO，他们就再也懒得写异步IO了，所以，最终，Ryan瞄向了JavaScript。因为JavaScript是单线程执行，根本不能进行同步IO操作，所以，JavaScript的这一“缺陷”导致了它只能使用异步IO。（注：不要把异步IO与并发弄混，在单核语言的前提下和限制下，实现并发。用尽单核语言的潜能。JavaScript是单线程执行的，不存在后台语言那种并发执行。一些编程语言/环境只有一个线程。这时如果需要并发，协程是唯一的选择。）nodejs和浏览器中的js加一起，使得js变成了通用语言，elm进一步以js为低阶语言建立stack，它可以mvc（一体webappstack）结构生成客户端可用的js。任何语言，具备能生成js这点的，都可以称为浏览器（上层）语言。"相当于"让浏览器解析这种语言。而且elm内置webfront模块控制html，它定义或模块支持有，相关css和div控制的逻辑。

对于elmlang，如果说js是一种focusing io的子集做进语言核心，那么elm进一步focusing on frp and react patten，elm对前端开发的支持和融入到语言本身是原生植入到语言的。elm架构和生态在js扩展端的对应物是redux全家桶。 redux参照了elm架构，而不是反过来。二者共享很多相同的成份和实现。

> Elm is reactive. Everything in Elm flows through signals. A signal in Elm carries messages over time. For example clicking on a button would send a message over a signal. You can think of signals to be similar to events in JavaScript, but unlike events, signals are first class citizens in Elm that can be passed around, transformed, filtered and combined. 在0.17版本之后(目前2020未是0.19.1) Elm已经用subscription取代了signal

下面简单梳理下React 的工作原理及虚拟dom 和 diff算法

> 与 jq等不同，React 会创建一个虚拟 DOM(virtual DOM)。虚拟dom 说白了 就是真实dom树的一个描述，用js的对象去表示真实dom ，当一个组件中的状态改变时，React 首先会通过 "diffing" 算法来标记虚拟 DOM 中的改变，第二步是调节(reconciliation)，会用 diff 的结果来更新 DOM。所有显示模型数据的 Views 会接收到该事件的通知，继而视图重新渲染。 你无需查找DOM来搜索指定id的元素去手动更新HTML。 — 当模型改变了，视图便会自动变化。------ 这又是一个解放程序员双手让他们关注业务逻辑的框架带来的功劳 将Virtual DOM（虚拟Dom）树转换成Actual DOM（真实Dom）树的最少操作的过程，叫作调和。diff算法是调和的具体实现，将O(n^3)复杂度 转化为 O(n)复杂度。diff算法原则：分层同级比较，不跨层比较；相同的组件生成的DOM结构类似；组内的同级节点通过唯一的id进行区分（key） 整个框架级别，通过Models进行key-value绑定及custom事件处理,通过Collections提供一套丰富的API用于枚举功能,通过Views来进行事件处理及与现有的Application通过RESTful JSON接口进行交互. 纯函数的柯理化可以直接在语言处理和编译期就可以合并多个纯函数（因为输出确定）降低算法复杂度和程序时间。

无论如何，elmlang可作为你的终身语言如果你是一个webdever。再也不要去追逐一门通用语言并寄希望于扩展它的库能达到终身适用其它未知未来领域了，因为这好似不可能。 all inside,all at hand : elmlang这种语言因为面向一种业务领域。很容易积累起知识群，经验群等社区。面向search engine也是一种，虽然它要搜索一次。也算某种面向搜索引擎编程。面向XX也罢，因为一切all at hand,searchable，左右逢源，无论什么问题都有参考能独立解决，总是最好的学习，，

未来我们进一步缩小这个webapp的范围将elmlang打造成webapp trigger editor and cloud app editor环境和workshop.

Golang，一门独立门户却又好好专注于解决过程式和纯粹app的语言

本文关键字：真正的**APP**语言。**GO**正确的设计。**GO**真正的分布式语言

以前，我总谈到编程是从xaas开始，到langsys到 domainstack到app的四栈叠加过程，语言因为平台也有本质上的二种：toolchain式和app式，历史上，人们总是企图从toolchain式语言上封装一次，在这上面发展app语言，这使得任何一种app都有了平台相关性，这种相关性或是CPU架构，OS的，或是toolchain libc。所以才会有那些移殖性的讨论和软件虚拟机语言（它们将平台重新发明了一次，以封装相异）和实现品，并在这上面长足发展了很多开发理念和实践。

在《bcxszy》的语言选型上，我们一直在寻找一种包含四栈却又不致于断层设计的语言体系，我们从.net,java,到qtcpp,llvm cling,terralang,再到lua,js,python，这些。我们认为但凡是编程，总免不了要从xaas开始。

但其实这是不对的，app和hosting居然是可以分离的！纯粹的toolchain编程和pure app编程是有区别的，基于前者的应用编程有四栈，但基于后者的只有三栈，因为，存在一种app和app开发生态，它是可以没有任何平台依托而存在的。无runtime设计，无backend to any hd, os,libc设计。

这就是go。它是真正的APP语言。

Go的设计级优点:真正脱离平台的APP语言

比起.net/java这种，c family based,llvm based语言，go对平台没有任何必须要依赖的逻辑，是完全问题域开始的。最通用的情况来讲，基本上在c family式语言中，为了对接上toolchain langsys的成果，x86上的for app高级语言(及它们产生的任何程序)，都会直接或间接引用到glibc这种运行时，因为程序设计，必须不是个重造轮子的过程。一些基于llvm的语言，封装c dlls形成自己的语言库，这种技术在llvm出现后更流行，因为新发明的语言往往可以直接call into dll libs。——— 但其实，C系更多的是一种toolchain。而非appdev langsys，但现实是是很多编译为native目标的本地语言都脱离不了它们是从toolchain生态作app programming的事实，这终究是错误的，而在.net/clr,jvm这类体系中，managed clr也是带虚拟机的，这类虚拟机后端植根于某OS中，引用众多，又巨大，虽然虚拟机上的APP是跨平台的，但虚拟机它本身不是跨平台的（实际也是多平台），虽然它出现的目的就是为了脱离平台，但它终究也是一种平台依赖，都免不了与toolchain及其hosting生态绑得很紧，作某层次的引用，—— 这造成的最大现实就是，来自语言和平台的依赖，使得分布式程序这个东西从源头上变得不存在，于是在分布式开发中处处掣肘。使得本地程序和分布式根本无法割舍。

而GO（非cgo）从0开始另起灶炉，它在各个平台的实现不包括运行时，而只是编辑器工具，支持面向各个平台能直接生成APP不需要发布任何runtime，好吧，它其实也发布运行时，只是它这个运行时非常小，且集成到每一个每一个由GO产生的APP中。不是像java,net等所有此类APP共享的，巨大，且与OS依赖严重的这类运行时。—— 你可以这样理解：它的每一个可运行目标，虽然再小，里面都有一个小机器，这种小机器不是x86，也不是arm,也不是任何其它一种cpu，运行时并不需要外置另外发布，所以它免去了从硬件开始就带来的平台相关性。它是将四栈中的平台集成到APP尾端的一种行为，而不是置于首端。

> golang它的前身limbo所属的os-plan9，甚至有从0开始，把kernel和系统调用都重做一遍的动作。它不基于x86。glibc都不用。因为glibc是从linux kernel来的,是一个rootfs最基础的部分。因为只要工作在x86上，你就得继承那里的成果，而GO不需要。---- 发明 plan9和limbo的那帮人实在是一帮眼光先到的家伙。

这就是真正的APP语言。是分布式开发的本质选择。

go的语言级优点：只做好过程式的分布式新C规范

曾经lua这样的语言也很流行，因为它直面了程序设计中的痛点：x86下的过程式，都不好用。而基于过程式之上的各种OO，又过于发展得太复杂。这类语言痛点劣迹斑斑人们却又视而不见，它们没有一个稳定的语言核心，它们语言技法虽显高级而难用，用于分布式开发要拖上平台。—— 所以，可见，对于这种语言的强化或简化需要是十分重要的。因为高级的语言技法，问题域的封装是开发和工业中的大难题。

Lua出现那时，并发处理也不是很流行，分布式也不，lua用不同于CPU绑定的方式实现并发。可见APP处理问题的方式与硬件，系统编程都不必相同。除非为了获得硬件加速效果 — 这也是GO提出平台与APP分离的依据所在。除此之外，LUA最主要的优点在于：它比起现今的多种语言，短小，只专注于过程式，核心稳定，技法紧凑，可以用较为省事的形式实现复杂的OO，比起那些严肃地从x86开始到OS到glibc的语言体系，它就是它自己，专门做lua支持下的那些有限集技术下的APP的！事到如今，类lua的语言一个个出现，如js,go，lua的思路也就不那么新鲜了

GO刚好也是lua一路的，只不过go vs Lua，有它从0开始的平台无关性，而lua依然是平台相关的那一类，所以除了那个线协程设计，它本质上也不是适合分布式的。

所以，还有什么理由不选择GO呢，即使单单是为了认真的做分布式开发。

我们把bcxszy的目录也明确分成了二部，第一部os/toolchain/xaas，第二部app langsys/middleware stack/devops tools，以示我们app编程不需扯上xaas和toolchain langsys，terralang c其实也不是良选。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

我为什么选择**rust**

本文关键字：**allinone**编程语言，个人是否真的可能学好多门编程语言

我们在前面《编程语言选型通史：快速整合产生的断层》提到,我们需要一门"简单，oneforall"显得不那么“断层”的语言来工作和学习，以积累自己的codebase和开发经验而无需推倒重来或切换，---- 这个问题之所以重要和紧迫，如上文所讲，是因为编程语言一直处在开发和学习的中心，占据一个程序员的大部分时间和心智精力，语言选型必须先于其它进行。而现实情况是，技术总在演变，而融合正处在初级阶段，人们在学习和工作中总涉及到使用多门语言的情景，产生的断层和学习成本巨大(见《qtcling - 一种更好的C++和标准库》和《编程语言选型之技法融合，与领域融合的那些套路》)

> "程序员总是需要不停学习"，。这个时候，我们希望获得一种简单，但allinone的语言和一劳永逸的开发领域研究对象。对于出现的新开发，也可以以自然不虐心的方式融入，共享同一套生态，---- 俗语，用一种语言解决所有问题。这个问题放在10年前是不可能讨论的，但现在可以了。
>
> “任何编程语言都一样麻烦”,软件无银弹，排除这些同质部分。来设想一下，对这种语言最基本的要求，首先是要保证简单，避免CPP那种越做越复杂带来的坑。不要求它有对低阶语言有很好的源码兼容性，但至少要能接上现今二大生态（naitve/web）。语言写法上的限制和自由是一对矛盾，这个时候我们宁愿入门前曲线陡峭，入门后一马平川，除此之外，它还应绝对安全（非必要不写unsafe），是运行期免错的语言。

对于符合上述要求，现今为止能很好作到“allinone”写法的语言和跨embeded/native/web/mobile(甚至web前端)开发的语言。Dlang/go/rust是唯三的选型对象，这其中，rust是最全面的。从语言演变历史中那些坑中综合排查，这点上，因为rust出现得晚，取长补短，rust没有任何包袱。它是最不虐心的。rust源于cyclone的部分概念，虽然说这些cpp的新版本也能做到，但cpp应该改无可改面目全非了，不如一门新语言来得更让人易接受。

具体来看rust的优缺点：

它有：

1，它涵盖toolchain即appdev领域,是真正的通用语言,着眼现在面向未来

> 它支持跨embeded/native/web/mobile(甚至web前端)开发，而其它不少语言也宣称它们通用，但总是存在大量的坑（比如人们不常用php-cli，仅用php-cgi）。rust没有这些包袱，是真正的通用语言,即又具备动态和脚本语言的写法和效率，而且又以一种自然科学不虐心的方式进行。而且它的微运行时能嵌入browser，着眼未来wasm技术。它也能被很好用于未来的AI，因为安全和效率比py高。

2，它涵盖现在的主要语言体系，聪明地定位于安全的编译型语言。

> 如在静态语言中谋求动态，如let,又将以前仅运行期能做的。用模板和泛型可以完全弄到编译期，如整套的编译期内省。达到了现在流行的静态/动态/语言脚本语言都能拥有的效果，支持脚本语言的逻辑热更。上面谈到，它是强编译期安全性的，能保证运行期免错（解决了c增强系如CPP的痛点：内存泄漏和数据竞态）。 其实运行期与编译期只有二个期的不同（对导致语言灵活和动态性差别没有本质联系）。不一定语言的动态性就一定要体现在运行期。编译期可以让语言达到足够的动态性，这就是动态编译。

3，它涵盖了所有语言技术,建立了所有语言的多范式超集，具备了流行的开发支持，

> 见《编程语言选型之技法融合，与领域融合的那些套路》，现在所有的语言，基本不是函数式（这类语言的代表就是lisp,haskell）就是过程和OO式(cpp,java)或者它们的综合(py,js,go,rust)，都可归为这二种流派。其实现都是对他们的简化与修正。而rust对这二种流派都有全面吸收和支持。

4，它不复杂，为了融合和发展，它另辟创新，解决了C增强系的痛点，而没有提出过多的新概念，甚至没有GC

> 拿cpp与rust这些“增强C系语言”对比，我们要忽略cpp对c的源码兼容是c的增强（严格来说，cpp应该向它的祖先cfront靠拢以增强C。）这层极具混淆性的意义来说，把cpp与c看成是一高阶一低阶的不同语言，这样我们就能方便地拿出两者都提出一门高于c的新语言的起点性工作开始对比：它们都保留指针和控制底层的能力，但rust有引用和借用,类型所有权，，极其聪明地（虽然有些强制和面向契约）规避了cpp的坑,同样没有GC，同样用raii却避免了CPP的内存不安全。
>
> 与dlang/go这样的“简化C系/cpp语言类”比较，达到同样的效果却而避免给语言增加gc这样的乱入式大部头。一门语言集不集成GC，这是影响到语言本质的问题，比如go集成GC，其运行时会比rust大好几个数量级,rust极小运行时。这对嵌入硬件平台和嵌入WEB浏览器极为关键。它对glibc这样的东西几乎无依赖。。
>
> 对语言的学习，从对比它们的类型系统，作用域，支持的高级数据结构，异常，。。。。。这些层面，基本一个认真学过几门流行语言的人都可以在10小时之内了解完。（引用，借用，其实并没有给rust新增加多少东西，其它都是向简集成）

5，它兼顾了现在，也能接入既有多种生态和未来扩展方向：

> 它能生成对c的转译结果（接上了系统实现和系统开发生态），以及对js的转译结果（接上了web前后端生态），而现在转译式开发是很流行的。place rust on top of c and js（比如 c,js作为低阶语言仅具备基本的过程和函数，oo和高级函数语法放在高阶语言rust层），这是很well arranged的。

> 对于未来，它也有考虑，如wasm，如virtual appliance，见《云APP，virtual appliance：unikernel与微运行时的绝配,统一本地/分布式语言与开发设想》，未来应该是libos,unikernel,Fuchsia os这类可编程可devops可嵌入可物联的内核的天下。

它没有下面这些，但不重要

（一个事物的优点的这个说法，正是它缺点的某些方面，在一定的可讨论范围内，任何选型都是一对矛盾）：

它没有对C的源码兼容支持它没有GC。这刚好成全4，5

它没有过于简单的语法。但其实这是个仁者见仁智者见智的问题。（引用，借用，这些是rust最本质的创新）

它没有显式的语言层DSL支持，但它内外部已集成足够可用的东西。见3，5

它没有统一后端，但有 C ABI 二进制兼容对外interoperability能力也好。，也没有类elm的interoperability to js，见5

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一种最小(限制规模)语言**kernel**配合极简（无语法）扩展系统的开发

本文关键字：可裁剪语言。**better c only rust, easier scripting programming,Easy ori programming,**非专业编程

在《terra++ - 一种中心稳定，可扩展的devops可编程语言系统》《terracling前端metalangsys后端uniform backend免编程binding生成式语言系统设想》《利用terralang实现terrapp(1):深刻理解其工作原理和方法论》中我们讲到对语言核心和扩展进行可裁剪的特性（类linux kernel），这种设计在app和kernel都存在，却没被用在语言设计上，。这种需求很明显，是因为现在围绕语言为中心作为代表的各种开发生态已经十分破碎，又变动频繁断层横生，提高了人们学习它们的曲线（而语言决定开发，不可能要求所有的程序员订立契约，倾向使用某种语言的统一良好特性，一门语言好的坏的都会被继承下来，又没有一种真正可裁剪的语言存在个人不可能进行改良），。一些整合现有语言类的统一后端多前端语言(c#,jvm languages)又做不到真正的前后或后端统一解决多语言带来的痛点。而一些更为新潮的语言希望彻底更新这类弊端，所以直接发明新语言，这类语言往往基于强化C简化CPP的目的出现。如go/rust/dlang。却也有新的问题产生，见《编程语言选型之技法融合，与领域融合的那些套路》。而terralang这类可裁剪的语言，可以避免二者带来的问题。

> 有一种事实已不可否认，我们的开发中涉及到多种语言，这些语言中，至少会有二种核心必要的类型：一种业务逻辑相关的（这种专业度高往往是系统编程类），一种是胶水脚本的（脚本类），当然一种语言也可以同时担负这二种职责，只是语言的这二种职责导致的复杂度界限很明显，一种可以无限复杂，一种可以无限简单，这种界限很容易被人们发觉，只是人们往往倾向滥用一门语言的设计。

go/rust/dlang

人们可能滥用一门语言的设计的例子有：

比如go，是对C简化而不是C++，它倔强地不加入任何多的东西。因为它面向制造一种for网络和分布式的新c规范（类lua的协程和轻运行时促成的分布式，lua的语法体量也极小和stable，我怀疑go当初基本就是它是一个翻创），它深深意识到，c成功的主要因素除了它是构建基本所有现代OS的基石（windows,linux,osx），还因为它本身保持了一个极小的标准，而几十年不变。即使这是一门专业人员使用的平台逻辑/系统编程类/业务类语言，也应该是为了简单而存在的（脚本类非业务逻辑其实也应该简单化）。应该将扩展放置在其它的合适的层次。----- go在这点上控制的确不错：只要它不将其它的特点加入语言核心，人们就不可能会滥用go本身。但是当go用户需要更强大的抽象时就只能寻求库级方案。

然后rust，是对c的增强对cpp的简化，它聪明地为c增加了内存安全和并发安全的特点，但是它没有像go一样考虑到以上特性，正在考虑将泛型这样的东西加入语言核心。关于语言功能是选择核心化还是选择库化是关乎语言设计生命的，放进语言核心可以使语言更一致地处理编程范式，但代价和另外一个问题是使得语言复杂化和臃肿化。

> 导致这一现象出现的根本是因为，即使在系统实现和业务逻辑层，语言也需要高阶抽象和工程设计支持来获得对开发的支持。而这是另外的职责层面 ----- 脚本和用户逻辑所在。而rust中没有显式的分开控制。这很危险，当rust那些现在看起来适合整合的特性在10年之后变得不那么合理了，它可能会变成另外一种 cpp。所以我们要规避使用这个职责部分，以下会说到。

d语言似乎比这二者都要好一点，因为它在设计中，加入了很多显式的范式规范，如betterc开关，gc开关，而它又是一种二进制与 C、C++ 兼容的c family语言，这使得它在实际上就是一种“二种职责分离了的语言”。

对于新手入阶的方案：

以上这些理论成为语言需要良好的可裁剪特性出现的重要论据。具体到现实做法上，首先是将一门语言负责编写核心业务的部分和用来给非专业用户使用的部分分不开，不让使用这门语言的人进行滥用。

> 语言的专业度和向简性是一对实实在在存在的矛盾，系统编程是任何编程开始的地方要求所用语言严密，用户编程可以使用非安全代码和动态类型追求简易和效率。如何将这二种需求融合在一门语言中，最重要的手段还是提供明确区分二者界限的方式。terralang这种就有支持,比如它继续了lua对外来类型就是user data type定义的属性，而语法扩展层面，则可以通过函数式元编程继续。

对于新手的学习，这种分离可以让人们迅速入阶，视各种新语言为仅拥有不同kernel（各种方言c）。但脚本层统一新的语言体系。大大降低学习成本。甚至可将所有语言的kernel植进一个新的统一 clr。

对于terralang这样的天生设计成支持裁剪的语言，我们只需各自使用好c,lua的原来部分即可(lua,c定义了二种极简正交的语言系统kernels对应上面说到的二种职责，而terra允许混用它们组成整个混合kernel)。语言的扩展全部实现为模块。因为它内置clang kernel可利用terralang实现基于大量c的方言系统。terralang是一种c的代替，类似go对c做的工作，它对c采用的简化方向是使其接近lua(luajit ffi本来就有极好的与c交互，terra进一步发展了这层交互不引入新的复杂性)，为什么是lua，因为它是一种类似bash的ultra simple语言。

对于rust这种，我们要时刻把它当成一门拥有最小“不可变核心”的语言+扩展构成的体系，平时大量使用其基础类型系统和过程结构，视它是一种“betterc only rust”，在寻求可复用和工程支持时，应该将rust的这些高层抽象语言用其它的语言代替（如python这种胶水语言，甚至bash这种不像严肃语言的语言，对用户来说其实更合适，见《一种云化busybox demolets的设想：免部署无语法编程》,它实际上是变通用编程为问题绑定的编程。而且尽力去掉语法，这是为入阶人士量身打造的实训环境）。能不用则不用。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

lua/js/py复杂度分析，及terralang:一种最容易和最小的“双核”应用开发语言

本文关键字：为语言学习划分一个核心工程。双核语言

说到最常见的编程语言，肯定是c，不光因为它是专业非专业的学编程人士首先接确到的第一门教程级语言，也是因为所有语言都是某种“c系派生”，，，可是它足够简单吗？如果把它放在跟现在的那些语言在一起，它肯定是最简单的（一门语言最重要的就是它支持并内建的简元类型和数据结构，运算符，流程语句，作用域，抽象类型/用户类型支持。其它的都是库级和应用级的。，，而c语言规则和标准库都可以在几小时看完并理解，内置的数据结构只有数组，内置的复合数据类型只有结构体/同位体/位域，唯一的高阶语法是指针和函数数组结合处的应用，排除c面向的问题域全是内存池，编译原理，数据库引擎实现不讲，c本身的语言级复杂性就仅来自在这，在于基于指针可以形成很强大多变和复杂的抽象和它一些其它部分的二义性，），换句话说，c建立了现代高级语言学习和应用的标准核心和最低入阶曲线标准。过程式足够图灵完备学好它已经足够启动编程学习，其它语言只是在c的核心上扩展出其它的机制并集成到自己的核心层，造就出的新的学习曲线，选择性学习即可。

但是考察现在新出现的编程语言，如果向大部分和绝大多数非专业的人打听学习它们的复杂度，十之八九得到的答案会是：c本来就很难了，新的编程和编程语言也越来越难了------ 这不光是因为当代编程始终停留在“c风格，用专业的手段干专业的事情”上，从几十年前的C时代开始也没什么本质面向降低难度的变化（见《一种云化busybox demolets的设想和一种根本降低编程实践难度的设想:免部署无语法编程》）而且还是因为，新出现语言的过程中对往往着重于对c增强cpp进行简化的工作，现代语言的核心往往反而在C过程上集成了越来越多复杂的东西，成为新的复杂度标配，但殊不知对c的简化或许更为必要。因为它决定一个最小学习核心。（见《一种最小(限制规模)语言kernel配合极简（无语法）扩展系统的开发》中的解决方案）----- 这一切都阻碍了很多人想学习编程和坚持下来的决心。

> 多语言学习为什么要强调一个极简语言核心和标准，和如何去做 因为从不专业到专业，一个人要学习的逻辑体量当然是从少到多，比如对于一个普通人，一般来说，一个类型系统，词法和语法，函数，运算符这些过程式对他们来说就足够复杂了，不能再多了，以后面临多语言学习的需求下，最好是所有的语言共享一个基本相同的类C基础和核心。 除此之外，逻辑的内部体量也应该小，人的属性和学习过程只能是举一反三，前后相续，逻辑不用断层，这样容易建立联系，否则会给学习曲线造成不小的断层无法继续。我们应该用已有的抽象去建立抽象，而不是不断提出新的抽象。用已有的道理去解释新道理。

lua/js/python的核心复杂度比较

最小的应用开发语言核心应该只提供什么？我们用相关的lua/js/py来分析其复杂性。

首先，c的过程式肯定是一门现代语言必须集成的。

1）事情要从最近的语言都加入了函数式（不要把这里的函数式想象成过程中的函数）开始。函数式在80，90年代非常流行，现在热度也不减，它本来是数学上专业的东西，冯氏机or函数机曾经也是选型圣战之一。它的典型特点和给类c语言新增的复杂度，就是提出了函数也是值，也能进入表达式和变量，那么为什么过程式能干所有的事，足够图灵完备，还要函数式呢，因为函数式有巨大的好处，是升级了的过程，1，函数能保存状态和携带数据（过程中的函数只是数据处理器，只能在参数等出入口放置数据），成为某些设计必要的东西和原来过程下的代替如callback，2，函数能给数据结构天然的迭代器抽象促成for each in，3，函数给协程和用户级多任务/异步机制有天然的支持。4，函数能表达高阶数学抽象就不说了。

lua只是有限地利用了函数。而像js这样的语言，起源于c+一种Scheme函数语言，整个编译器实现处，用函数可以作为”元“，解释和实现很多其它的语言机制，比如面向对象，甚至面向过程本身，允许类似直接在抽象树上写程序的方法（即用户用函数写程序实际上模拟了一种语言发明，库级的东西与语言级原理相当对应）。

所以对于深刻理解了函数机制的人，他们很容易学会lua或js的函数部分和非函数部分。对于初级入阶的人，也显得相对学习起来容易。这就是js函数的举一反三机制。

这样看来，拥有c过程和函数为kernels的语言其实也不复杂。比如lua，js，都在接受范围之内，但js干脆把class这种udt/adt和oo(有class不一定要oo)也归入语言核心，实际上就有点过于复杂了。lua和js还有一个历史共同点，都有过成功的jit实现。

2）另外一种极端是py，它也容易举一反三。因为它在数据类型上（它区别于js的地方在于用内置数据结构造扩展而不是提出一种作为代码结构的函数机制），像lua一样，统一为某种meta数据结构，lua是table+metable，py中一切即对象，扩展语言的方法是库级扩展这个宠大的对象系统（其本质是在传统运行期执行栈帧中增加新的数据结构-也是发明py本身的技术，相比之下js的更类似于某种写ast的动作-也是发明js自身的技术），不过py复杂度总体上跟lua和js没法比较，因为python它强调battery included in stdlib，而lua和js只有一个最小核心和最小标准库。

> 可以看出，py和js都是用语言自身的机制造语言并允许用户扩展语言自身(下面会谈到terralang借助metaprogramming允许你将库真正写成语言而且可import as lib到terralang)，而java,cpp这些，用rtti这种完全新提出的东西造class和OO。前者没有增加用户学习曲线的复杂性，而后者增加的不是一大坨。

这样，lua实际上在c的基础上，至少增加了二种高阶语法体系，如果说c只有指针，lua实际上有面向函数和面向对象的气质和血统了（lua的模式匹配用来抢php,perl的市场也是kernel级的？）而过程式+函数式，往往是一种现代语言核心复杂度的标配了。。不过lua不同的地方在于，它聪明地让该属于核心的部分放到核心，把该留给用户的部分还给用户和库，以及应用。

3）lua/luajit聪明的地方还在于，它自己不准备发展得越来越大而定位于与宿主语言(这其中它只全面拥抱c)协作，所以它所有的机制包括扩展都足够“简元”，luajit的ffi使得lua就是c的脚本版本（而且这种互操作非单向而是严密双向的），使得c实际派生出另一个cscript分支，而避免了像js和py这种语言一样将这一切(编程涉及到的二种职责，见《一种最小(限制规模)语言kernel配合极简（无语法）扩展系统的开发》)放到一门新设计的语言kernel中。显式化这二者的界限而不是集成到一起模糊化反而是重要的。

> 这种可分的结合是非常合理的：c需要lua扩展，因为它解决了“用c实现对象系统”，C不能构建高级抽象的传统难点，而lua也需要c，因为一些扩展只能用系统逻辑实现来写。这二者相互穿插，在他们的领域都互相需要。而且你放心，能saveobj到exe的terra代码也允许是纯 pure lua code。lua runtime/c runtime都可以分别发布,terralang只是整合它们形成的一个console editor性质的东西，在terralang中并没有runtime,savetoobj的terra source .t只需lua runtime便可运行，不必带terralang这个近100m的宠物大物。 对于cscript,我们在《qtcling》《terracling》中讲到选型一门真正的c脚本的历程，但似乎只有ffi+lua才是真正的优选之道。

这一切的作用，就是控制了良好的外部复杂度和留给用户入阶处能举一反三的余地。

terralang：一种terra/lua组成的混合的最简应用开发语言核心和扩展体系

再来详细说说terralang，它是代替lua/c（这二种语言本来就是极简的）作为语言standalone kernels复杂度控制的新招：混合kernels。它进一步发展了上述c和lua的关系（相互扩展和混合），更加科学和实用地将复杂度控制和各层用户入阶考虑做到了极致。避免了单kernel的根本缺陷。

怎么说呢？terralang基于clang，是一个极强的静态分析工具，可以让古老的c工作在一个现代的实现平台上(terra)，terrlang中的terra是用clang实现的编译型语言，它面向接近c，却力求与lua靠扰，追求terra/lua之间的混合编程(terra中可lua,lua中可terra，顶层是lua，terra只是embeded并被lua metaprogrammed，你可以将terralang视为一个lua,一个lua with terra extends, standalone mixed terra/lua:terralang)。继续了独立的lua/c/ffi时代这二种语言间的交互能力工作。将这一切发展到一门”双核语言“上,方法是：terra函数是一种lua value，就像函数是一种lua type/value一样，然后统一terra/lua上下文语境中的词法作用域。这样二门语言就混合了

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

编程实践选型通史：平坦无架构APP开发支持与充分batteryincluded的微实践设施

关键字：自带**paas**语言系统,自带组件,自带用户工具语言系统，一元下的多元，初学者第一门实践语言选型

在编程实践选型上，往往涉及到来自语言，应用，工具，人（又分多种角色）多个方面的影响，需要处理多个现有生态异化矛盾和处理已有实现之间的差距，甚至出现需要重新发明语言等基础设施间的轮子的情况都时有发生，是一个格局需要放大到软件工程方方面面的过程。——一言以概之：软件工程，——-其实，大凡考虑进所有事情并最终处理涉及到人端的那些方面的事情都能称为工程，可见其大和复杂，这也是为什么很多图省事的方案企图提出一个“one for all”终极方案的原由所在。这个“one”往往可能被实现为开发上来讲的一种/多种语言系统，一个包含语言修正/强化的framework（如qt），一个工具，一套问题域解法，一套敏捷方法，等等，当然更有可能是它们的综合 —- 公司或个人将他们做成自己的codebase用于持久复用以应付统一选型。。

在《1ddlang选型》一文中归纳的是以前1-4种旧的纯粹语言选型方向的“one for all”讨论—lddlang也就是某种1ddcodebase，在那里我们主要聚集语言内部特性。本文作为《1ddlang选型》，《语言选型简史：快速整合产生的断层》的系列之三，将继续着眼于大的工程方面的综合实践选型。

> 那么，有多少种oneforall？都有过哪些实现或案例呢？理想的oneforall呢,这样的终极方案始终存在吗？讨论它有意义么？
>
> 虽然被不断怀疑，然而终极编程和“one for all”真的存在,它并不需要是类似编程葵花宝典之类的东西，我们可以理解让编程体现为适可而止，有止境的境界，在工程上(编程上让事情变得越来越容易最后不需投入或极少投入再学习成本)，通往其的方法可以有很多种，但一种无疑是那种—比如，某种直到脚本和可视编辑器级的工具级开发封装。 如果编程方法可以归结为一门最终的哲学，学者可以利用它举一反三，完成自举学习，那么这种元性质的哲学，就是终极。 图形界面的出现和DLL API机制，VB可视化，在这个意义上都是伟大的铺垫作品，面向对象也是一种终极编程，它在语言内在抽象接近平民，各种OO范式，PME，再后来，框架容器，都是使编程变得终极的方法和基础工作。
>
> 这些只是整合而已，严格来说其主旨只有二点： 1）着眼于大的面向人的工程和实践方面，首先要保证这个一体化方案必须是方案全包和self contained的（没有断层，不用跨越，，不用四处找方案）。必须是“一元下的多元”。比如统一后端语言系统要包含多种前端做到no binding。就可视作是此。 2）然后是平坦化（“多元下的一元化”）。能围绕一门语言良性地建立起实践的方方面面工程与选型，且所有其中的东西都是one style的no steeps，故这样的方案必须要包含一门语言，一种部署方案，然后使这一切形成平坦化的。可能最终归于一门语言系统级的封装实现中。
>
> 故最大整合，全包，又要求良性集成 – 尽量多用onestyle的抽象线索来整合并突出统一易用能用。才是其主要特点，而如上所讲，一元下的多元，多元下的一元onestyle其实并不矛盾

也许最好的例子，就是我们熟知的WEB和web开发界中.net这样的东西，WEB是一个涉及到运用云计算这些最新软件技术的领域，又运用了上述所有这些—-特别是语言上的开发技术，的地方如.net,js（当然还有更多…），是封装最完善最统一的领域，可以很好拿来作为例子。

《编程的最高境界是什么》：以**web**举例，**onestyle WEB**开发与调试设局的编程实践

WEB开发是现今最简单的开发，没有最简，因为它就是最简。

WEB前端开发门槛很低，因为在大量框架下，最前端的样式效果开发（实际上不叫开发，叫界面配置）已经变成纯粹的艺术行为了，因为它变成了debug and visual driven coding,或称programming,而js用来编程前端html dom 和style elements，也可用同样的调试和开发方式被调试。

而web app开发主要是处理一些服务性的api,云时代不同应用通过service api交互，服务性API，js+xml（json,etc..）让任何领域逻辑的开发编程，都变成都是界面编程下的逻辑(前端也是xml->html,css也有xml化的scss)。xml是数据化代码转领域逻辑的终极手段。

这二者是统一的技术一个可视一个XML语义化而已，使用相同的语言。这个层上的web开发就是一个充分经过了onestyle风格整合过程的多工程开发领域。这主要指问题域还有使用的语言，见下：

什么是终极编程及最好的实践语言，以**js+.net**举例

在前文《语言选型通史中》中我们谈到.net和JS，谈到前者的统一后端特性和后者的跨三界开发。

1) js为什么流行

a, 前端是一个对领域逻辑封装得最完善的领域，包括桌面GUI，HTML，网站模板技术，MVC范式等。html标识系统，JS作为前端语言内部的适用性，源于web前端，也能应用成为mobile,native的前端。

b, 而js也可用于后端服务器甚至传统native开发—当然这是非c runtime的而是jsruntime backend的（这属web的“系统编程”了，实现编程，区别前面谈到的service api应用），js+web有鲜明的one lang,one problemdomain solution的特点（这里js是nodejs实现或emcs规范，它被广泛用于服务端编程），虽然它只是在目前慢慢成为事实…。

你的第一门实践语言，应该是类JS的，松语法描述，调试资源丰富，从界面到逻辑，一条龙，可渐进，先前端再后端再全栈，是实践的最佳路径和起点。这或许就是为什么JS是越来越多地成为最佳实践语言的原因根本

2) 再来看.net和web的后端，系统编程和实现方面：面向多种定制化的平台，包括工具在内的全生态self contained

因为clr平台提供了最基础的后端和组件环境，包括语言服务，平台后端，工具在内的东西都可以定制成self contained的，可以说，统一后端，将原来native开发上可用的东西全弄到可托管后端支持下:

a, 统一后端，也是统一了平台，跨mobile,web,native desktop,是第四代语言最鲜明的特征。

(与traditional c runtime backend nativedev分开,在这里，桌面开发分native上的桌面开发，和托管到.net clr上的桌面开发，.net上有自己的封装版本，所以这是另外一种全功能的跨desktop,web,mobile托管平台。不对native开发断层。)

b, .net 可以直接调用语言服务，甚至新的语言完全可以通过dropin语言前端的方式放置的方式安装。因为在.net统一后端下服务即组件所以接入了后端的语言前端只是普通组件可按普通API方式调用。

c, .net这样的框架为IDE提供了一个统一。使得sharp,mono,vs这样的东西不致于存在像本地编译器那样的鸿沟，比如不必安装巨大的本地sdk，大家可以共用一套runtime（由于.net runtime即组件，组件即源码标准库，不必发布巨大的sdk），sharp,mono可以仅面向托管码，调试，智能提示可以轻易做得跟vs一样强大。

这点使得用户维护一个属于他们自己的，且跨三平台统一的codebase,runtime,ide,甚至一切成为可能并可打包带走。

它是一种根本上的迁移。只要是在迁移之上的东西，都是self contained.taken alway的。

架构已定，用户入阶和工具支持方面：

如果说选型已定（包括OS支持，语言，语言强化，服务端引擎，具体应用体系 — 这些往往被做在了一体化的语言或统一后端中，或PAAS环境中），那么这里就是尾端APP开发。由于问题域实在太复杂太多变太广泛，很难有大全的编程套件能兼顾覆盖全部领域，而.net这样的统一后端能容易裂变成子集又有社区包管理，能最大程度地向无架构APP开发靠拢。

充分的复用支持和应用域实现及工具级支持 — 但这也是在当今时代，复用二字，是用户需要学会用他们解决问题的最基础方法和唯一手段，也是使用任何语言最基本的素养要求：.net开发只是设想你会任何一门它支持的语言，即使是vb.net或是某brainstorm。

1) engitor无架构APP开发：

全生态+无架构是微实践的基础。因为只有做到了无架构，才能避免在用户开发APP层级还进行复杂设计的必要，才能到达微实践的地步。架构和设计已在前面的语言整合中被解决了。

用户开发应该是二次开发，无架构，平坦式复用，，，而大部分时候我们都是在进行二次开发。以软件为体系进行该软件之下的扩展开发，形成的应用叫APP。那种带设计的大型开发，要么是重造轮子，要么是更复杂一点的二次开发而已。

> 比如，OO就是语法代码级的扩展。和二次开发。是语言级对开发是一层层堆栈或剥栈的事实反映，迭代是最安全的类OO之于demo继续演化的工程行为行为，也强调继承了以一点点叠加慢慢发展的思路。因为从抽象上说，软件本质就是栈的添加修正行为。所以，结合《编程实践养成过程反推式》：demo演化，也是编程。只不过它处在工程明显的地方，即尾端。

所以，用户设计中，应该尽可能地避免复杂设计和规模过大的问题封装。（导致的语言应用过于复杂和问题封装问题）那种为了特定问题提出一套语言特性的做法更是不可取。语言核心应该稳定，不应该从下而上地变，虽然从上而下地变是可以的。更不应该透给用户编程层。

2) 微实践：

在以上架构下，实践者仅要求有的基本素养，1你不需要懂太多，在你能控制的复杂度内干正确的事；2假设在新API面前，每个人都是脚本工具小子，这样就够；3,懂得适当的逻辑归类，甚至免OO，比如，你不习惯用OO，完全可以用编辑器或者用统一后端支持下的仅支持过程函数在源码文件中摆放编程的免OO DESIGN语言来编程。

一个能打包打包，在一个小时，一篇文章中讲完教完的编程实践才是好的自包含的微实践课程。这对最初级的学习者也适用。

.net和js，它们都是onestyle的code and demobase(说demo是因为有组件，这个demobase是comopentbase)，js npm社区有几行代码组成一个库的micro service存在。开发只是组合服务。

demo as engine+microservice开发兴起了，所以1dddemobase > 1ddcodebase了，找类qt的那种开发库组织1ddcodebase过时了。恩直接找大量第三方小件，杂乱的也行，只要搭出的demo具备好的facades就行了。

可复用件选型+小微开发，这一切得盖于最良性的基建结构—语言选型部分。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

项目与领域选型通史 - 0基础，**demo as engine,not framework,but toolkit**，非侵入式省事编程选型

Xxx

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

用开发本地**tcpip**程序的思路开发**webapp**

本文关键字：**the headless cms,b/s web to c/s web, headless webapp backend.**

不可否认的是，webapp已经是与desktop, mobile app并列的主流appmodel之一，但是，web却是一种典型的Appstack as os，webapp是在native server apps上打出的一个b/s洞，再在这个洞里发展出来的一整个世界(跟移动端APP一样比如安卓)，，比如，它底下的appstack，分别属于native/desktop的范畴,负责gui的是nginx,db mysql,etc(代表webappstack的是服务端app，客户端只是“client side html,js resources”，它脱离了服务端，就没有自身存在的意义。不是一个独立可用的app)..

Vs nativeapp和nativedev，它从来没有自己的os，或任何标准的宿主定义。较之native app，它不算是一种有专门运行它的OS供它托管运行的“app”，你要说webapp的host是lamp，很明显，lnmp中的l并不属于web,是applicationserver?beanserver?,也不是，它是语言的组件服务器，换言之，只有nginx,mysql这样的东西属于web —— 只是说，技术上通过ln+np的组合，能搞出一种web程序。这跟移动开发类似，它们都是linux和一种虚拟机语言双重托管运行下的app，——— 本来嘛，web开发和移动开发是beyond native层面的，也只须这样。

web的设计与缺陷

在开发上,动态程序的web app是monolith的前后端整合的，叫page app，程序员在后端完成所有的程序开发，Webapp的框架逻辑无非是routing,template,orm，route，mvc这些框架逻辑。代表一种appmodel的，无非就是它的stack框架逻辑。因为它考虑进了浏览器是服务端和客户端一体app。web程序之间不用交互和复用，没有api机制，也没有web件，web as service(当然，这些后来也有。。。)，只有语言源码级的复用。

应用上，和后端运维上，也都是整合在web的。用户在一个web界面上完成所有的事，比如cms后端管理。用户和内容也是集成在远程的。没有线下只有线上。 这也就是说，程序靠后端，内容靠用户，全民线上分布式，没有线下分布式，它整个就是monolitch的（而且采用http，js这些具体选型没有二选，使得webapp是fixed的）。

以上这引起都不是问题。也不是web的问题。就像Web刚开始一出来其实就是分发静态文档只是后来有人用它来强行运行webapp而已(而且分布式应用开发本来在工业上的实现就很破碎，历史上并不存在一个真正的分布式OS，也不存在一个分布式appserver，见《plan9一个真正的分布式OS》) ——— 历史上，长链接，webgl,streaming content，这些，一直都是从各个维度去克服web monolitch page appmodel，使之多元的努力，

只是，只能有线上分布，不能有真正的线下分布，web这个缺点是显而易见的。有完全适合将web置于线上的现实需要，也就存在与现实的web应用现实相左的需求，比如，存不存在一种线上线下合作分布式的webapp呢？那些在本地可以处理的就让它在不必在远程，比如后端管理，使之跳出browser? 就像git的分布式那样，——— 在前面，我们也不断讲到此类思路，比如用静态网站思路来开发webapp，用tcpip来开发b/s。

新**webapp**

这样的方案是存在的，网上有wordpress headless cms这样的项目，这样努力的结果就是重新将web置于规范级，将webapp重设计，它仅需要是一个http协议，也不必是一个b/s app，web只需是一个gui，而不应成为full appstack的全部。这样就分离了线上线下。线上线下分别开发，这二者通过api链接。

重新分离的好处多多，最明显的就是，开发上：

1）Web的服务端可以真正作为headless backend，变身as service服务。有api机制和复用。客服分离开发，用c/s方式和类nativedev方式开发，客服不再拘泥彼此的技术规范和语言技术选型。

2）简化了服务端开发和选型，显示逻辑分离，服务端web框架再不用mvc这样的东西及其它同时考虑处理客户端routing等的逻辑,Lnmp中也不再需要php了。可以在服务端用任何一种语言来实现。也可以有gitstack这种多选择的选型结构。或仅是其它采采用了http的其它非lnmp的xxxstack,所以，webapp的后端可以是任何东西。

3）将客户端开发独立成线下，不再将webapp视为一个monolith的appmodel，类c/s web，可以用任何语言实现将html视为编辑器中的asserts，不仅是浏览器了。，可以是其它独立的客户端app,

4）变“瘦服务端开发”“强客户端开发”，比如Wordpress,插件可以在客端做，不用开源了。或者反之，那种复杂的线上交互网站，也是可以的(可是，那还有其它方法来解决不是？比如将动态部分也弄成一个headless “chat”,headless “comment”,headless “xxx”??)。

应用上：

1）cms可以发展为headless cms在本地管理，拥有真正独立的客户端app，可以fully Turn into functional in-browser application.

2)如果是展示型网站如cms，远端web程序也只需是个存储空间。比如阿里云OSS这种。这样，一个静态空间就可以解决cms托管的问题，而且更专业。

还有更多。。

其实，上面几点就“treating b/s webapp as c/s webapp in nativedev manner”一个意思。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

raw js下使用Electron，以webapp思路开发支持Electron小程序的移动APP(1):谈web前端的演变

本文关键字：**web**前端演变的历史**,vue,react,pwa,spa,mpa**

在《用开发本地tcpip程序的思路开发webapp》等许多文章中我们都讲过web/webstack的原型(那文主要讲前后端分离的web开发思路)，如果说一种ui代表一种app，(我们用app特指那种需要下载/安装/运行，区别于用浏览器打开作为的那种桌面和移动app，它们一般寄宿在os上用native rendered gui库发明,体验为无延迟的UI和APP，可以调用本地的服务)，那么或许在2015年前就从来没出现过什么严肃意义上的web app ui/web app，只有page ui和网站，虽然我们谈webapp已经很多年但其实它是个新事物，为了应付日渐庞大的 Web 页面，经过优化的 JavaScript 引擎已经可以和一些编译语言的速度相提并论已经不仅是用js实现交互就行了。因此用web打造app的思路出现了，从网站到webasapp而这二种需求看似共享同样的基础，但面向它们的设计都不相同，web一开始就被设计成静态，保有鲜明的内容属性而非可编程属性（语义，routerable,可退回/可前进，页面内容可索引可爬取可seo），因此它要变身“某种用html表达的app”，类似native render ui/native app(甚至整合它们形成多端合一的app:一云多端的OS+多端合一的APP,universal webui+siteui app)。这不但要求web本身规范变(w3c)，而且要求支撑现有web运行起来的那些东西也要变，甚至变化过后，会破坏现有的web本身。

从siteui page到webapp

首先是硬件和OS平台要为支持它们（而产生变化），webos有chromeos和html手机(基于 Firefox OS 的 KaiOS据称全球第三大移动os)这种，包括最近出来的fuschia。

然后是web本身的富网页技术(基于flash的内容创作型工具被面向程序员的js技术代替)要到位，用js写成用浏览器跑出来的ui，会跟真正的原生UI在体验上相差很大比如延迟高没有长链不能调用摄像头无法推送无法离线，这一般来自浏览器或应用的“webview”支持，（虽然说现在的浏览器有localstorage,websocket,webgl,etc.近二年又有service worker.但还是不够）因此要求增强浏览器本身的api(一般来说，正常的浏览器page运行在沙盒里没有权限调用本地api，而特制的浏览器如electron这种会面向“发明webapp”的需求保留一些特权API,，这也是它经常被apple mas禁止的原因，微信那个浏览器也是增强了私有 api的,实际上，在w3c规范之上定制浏览器这种行为就破坏了web的应用形式，故提高webapp体验的同时又不致于破坏现在的web这是一对矛盾)

> webapp的意义：（这个技术如果成立，web ui可以完全代替渲染原理的远程桌面形式的remote app，真正实现原生不光跨端甚至跨云的UI：我们知道，远程UI的原理大部分都是投屏技术这丝毫不原生，云没有自己的原生UI与APP技术:webapp与容器正在填补这个空隙，在前面一些文章中我们多次提到远程APP这种云原生APP的需求,曾经我追求这样的统一本地和远程的appstack《cloudwall：一种真正的mixed nativeapp与webapp的统一appstack》，但现在看来：正是桌面早已成一套的情况下原生云UI都做不到成为规范，，所以导致现在都没有“universal desktop/web app”）

最后也是最大的变化还是语言层上这种appstack本身和开发这种WEBAPP的范式上(因为前端始终是面向程序员的而不仅是内容制作)，事情发生在2015年之后，产生了将web ui作为app ui的技术潮流，并由此诞生了很多 js框架。

> 我们在前面说到框架，往往是基于对整个ola（os/langsys/applicationdomain）的修正增强补充。"js框架与浏览器/server环境"的关系跟"ola与框架"的关系一样，可以全方位去修补，因此，类似react,vue这样的东西其意义可大可小，可以是一个库也可是一个框架。虚拟DOM就是这类框架用来修补目前还跟不上webapp需要而作的框架内暂代方案。

webfront/webfront框架的前世今生

我们来谈下webfront/webfrontdev的前世今生（从网站前端到webapp）。首先是自适应、响应式设计，前后端用模版实现分离，DOM API最开始是一些JS框架要面向的全部东西（这些DOMAPI就是网站ui可编程的全部，注意从这里开始，我们严格区分siteui与webappui的说法），JQ这些能很好封装ajax和dom，然后出现了更高级一点的MVC，前后端通过json数据交换实现分离《用开发本地tcpip程序的思路开发webapp》，那种monolith websiteapp的局面(服务器脚本中套前端和模板)被打破,一切用到的展示数据都是后端经由过程异步接口(AJAX/JSONP)的方法供给的，前端尽管展示。。再后来就是MVVC，到这里开始，已有webapp的初级形式，一些网站前端开始谋求progssive(注意这里是网站用的Progressive)。与移动端UI结合，又有pwa,spa,mpa(这里才是webapp化之后的前后端分离技术),

对于webapp:spa,mpa基本上，它们的分离前后端处理方法是把原来后端做的事放到了纯前端，SPA 可以在客户端提供完整的路由、页面渲染、甚至一部分数据处理； 这往往需要一个比 jQuery 时代更重的 JavaScript 框架，来实现这些原本发生在后端的逻辑(在js下的web开发就是前后端分离又融合又融合又分离全栈技术方向会越来越统一，甚至环境都可以互换，ssr可以放到服务端做，而serviceworker也可以像cloudflare一样放到服务端做，被称为大前端，比如angular的出现，使得后端的mvc,和依赖注入等技术在前端也可用，但这只是例子之一)。而且使用虚拟DOM组件来代替raw DOM。----- 目前比较主流框架：vue、react、angular等框架。AngularJS 可以构建一个单一页面应用程序（SPAs：Single Page Applications）。Vue (pronounced `/vju:/`, like view) is a **progressive framework** for building user interfaces.我们在《elmlang：一种编码和可视化调试支持内置的语言系统》《why elmlang:最简最安全的full ola stack的终身webappdev语言选型》讲过react和fp和虚拟dom组件（这里的组件并非指脚本语言二进制构件级的概念，而是指前端和浏览器DOM层级节点的“组件”，代表前端ui的统一可管理单元，类似桌面的widget），vue就是较它们更为轻量一点的方案，也有虚拟dom和组件，Vue 除了是 React/Angular 这种“重型武器”的竞争对手外，其轻量与高性能的优点使得它

同样可以作为传统多页应用开发中流行的 “jQuery/Zepto/Kissy + 模板引擎” 技术栈的完美替代。angular,react,vue都属于纯粹用组件，用程序员的思路来管理前端，虽然这样可以使得html ui更像webapp ui，但这样会破坏web的原始语义（比如seo），所以普通网站不太适合用这套框架，这都是网站管理后端，工具化应用用的。

> 大前端，这就有点像游戏引擎下开发游戏APP的思路了（这里是逻辑渲染引擎）。virtualdom就是从chrome真实渲染引擎中再建立一套编程用的场景管理和渲染实体/节点。而游戏UI/传统桌面GUI都是组件管理和组件逻辑，这个组件可以是程序上的componet也可以是可视的widget，组件就是变web html模板元素为传统的gui方式进行开发(template based ui变成了带状态的可编程ui widgets)。

PWA 方案（pwa+轻度spa）更接近于 Web 的方式，它是 Web 的增强而不是替代。PWA 一词出自 Alex Russell 的 Progressive Web Apps: Escaping Tabs Without Losing Our Soul，构成 PWA 的标准都来自 Web 技术， 它们都是浏览器提供的、向下兼容的、没有额外运行时代价的技术。 因此可以把任何现有的框架开发的 Web 页面改造成 PWA，而且与 SPA 方案不同， 没有强组件化机制，因此不需要一把重构可以逐步地迁移和改善。

Progressive 是指 PWA 的构建过程。Service Worker 是一个特殊的 Web Worker，PWA 对性能的提升主要靠 Service Worker(cloudflare workers用的类serverless云函那套)，它是在传统的 Client 和 Server 之间新增的一层。性能提升程度取决于这一层的具体策略。例如：它可以提供缓存服务,通过service worker在后台更新下载实现页面离线也可以浏览(如material design)，加速应用程序渲染，并改善用户体验。

SPA:得益于ajax，我们可以实现无跳转刷新，又多亏了浏览器的histroy机制，我们用hash的变化从而可以实现推动界面变化，纯粹依赖原生浏览器环境，而无须用到vue,react组件里那些重度方案。

更多etc..

electron开始webapp

虽然web是本身跨平台跨语言的,但它并非严格跨端的，上面说到其存在多种浏览器以及内置浏览器的APP（如微信小程序以微信为宿主）的具体实现形式，这些依然需要在js框架层进行适配，web应用(包括webapp为前端的形式的整个web)都一般有明显的前后端分离做成“一云多端”结构，后端一般是传统意义上的普通网站，（你可以在后端托管具体端，比如小程序端的前端资源文件，和提供小程序框架性的服务API。当然如果是管理性后端你也完全可以可以做成webapp形式)，前端则对应PC/mobile上原生浏览器网页,pwa page(ES5起兼容的普通浏览器pwa+spa那套)，webapp spa/mpa,应用内的小程序，H5（这货也可以理解为原生浏览器），等多种形式作为UI端和客户端。----- 这些客户端共享webfront技术、w3c规范虽然不是严格意义上真正的跨端，但已经做得差不多了，而vue,react等框架用“AST,渲染引擎，bridge,runtime等抽象中间层”进一步实现了“框架和开发层跨端”，(这个跨端才是跨各种运行js的浏览器实现而不是传统意义上的跨OS) ---- 也干脆出现了一些“跨服务商的webapp引擎”，如小程序开发框架，开源小程序引擎，在中国主要是一些电商系和外卖系移动APP的前端，其本质无外乎内置了自己的浏览器，加上vue封装的小程序引擎，处理“提供一个小程序的宿主运行环境（如微信小程序的运行环境：分成渲染层和逻辑层，WXML和WXSS工作在渲染层，JS脚本工作在逻辑层），并植入自己业务的服务性API”的东西，及更多其它事情。

跟普通浏览器环境一样，微信内置浏览器一样，electron也是一种端,它提供了环境(整个v8服务,及它自己特有的一些api和扩展)，支持严肃和纯粹，追新的webapp创造，，electron可嵌入显示普通网页和webapp，你也可以用electron做pc小程序也可以做移动小程序，electron下的小程序也不是普通网页套个electron壳(比如我要做个blog客户端就是给个主页逻辑给electron)，而是网页意义上真正的app化。

跟它们不一样的是，electorn还提供了devstack(开发，语言支持，注意这并不是js而是更强大的nodejs/包结构和可开发生态)，框架方面你还需要vue等。vue和nodejs.还有vue-electron这样的封装，通过webpack生产webapp打包及流程自动化，产生支持普通浏览器js的vue引用。

下文我们就准备使用raw api,使用electron提供一个小程序的宿主运行环境，配合简单后端，打造我们的移动/PC小程序:一个类《打造小程序版本公号和自托管的公号》miniblog的客户端。利用它发明多端合一的“小程序”可免去依附于微信这样的环境(微信小程序审核很费事。一点社交性质，个人版的小程序博客带评论都不允许。为免责关注程序而不是内容)。

一种开发发布合一，语言问题合一的**shell programming**式应用开发设想

本文关键字：最简单的编程模式,具体**app**具体开发,将开发局限在具体**app**级,**demo as engine**

在我们前面，为了ease编程复杂度和去断层化，我们从xaas聊到langsys,devops tools,domainstack，涵盖众多，我们提到云化程序应该与云容器一起考虑，开发与部署一体，才能可能达到更自然的类本地API导出+api sync的分布式，我们还提到很多语言，比如terralang，还提到很多devops tools,我们其实都有一个错觉，以为编程的复杂性来自于语言，系统，这实际是对的，可又有失全面限制我们的认识面。—— 因我们始终没有考虑进对问题的理解，因为编程即抽象，是涉及语言，平台，问题，人全面四者的,平常我们总是把它们当成是语言的附属 —— 一些库级的语言附属品，问题总是被抽象成某语言的可复用库，业界从来都没有关于它的独立考量和成套的技术，但实际上跳出这些，做综合全观，往往会有意想不到的认识，比如我们发现： —— 具体问题域可以与具体app绑定。而app级，可以有具体语言绑定，作开发发布。而这是一种更为正宗的“应用编程”，有完整的xaas,langsys,app,people的生态，而且更强大。

较之目前，xaas,langsys这类基础工具往往做得不够亲和，它们对抬高编程入阶的影响也就越大的现实。。如果把这种开发独立出来成为一支，使之主动无缝对接前二者。另辟新径，整个编程会显得尤为更简单,比如，它免部署，甚至不需要用到通用语言这种难懂的东西。

这篇文章将综合描述这种可行的案例。

为什么**shell programming**是最简单强大的**programming**

在我们使用os的最初，我们使用C系作应用开发，同时作为一个开发者，自己还敲打一些命令行，进行一些运维级的操作和浅层shell开发。CUI和GUI的使用分开了运维和普通使用者。shell编程用类似配置的方法提出了第一种编程方式 —— 这些都是我们现在经常做的事。可是稍微一想就会觉得很分裂：为什么C语言用指针file* 来抽象文件，CPP可以用FILE file，它们抽象编程的方法都用了与它们的简元，语句这些直接的基础设施，而shell可以直接操作一个文件，不用类型指代，这源于一个傻子都明白的事实：任何编程，只要它的代码是要存为目标上的可执行文件的，编程都是为运行期作预编码的事实。在编程的时候，运行期是未知的，除非debug的时候，或devops调用模拟器的时候，这就造成了编程的脱节。

而shell编程恰恰相反，编程是在现在的上下文中作运维。开发和部署是天然一体的。而且最重要的，问题和要编码的事，都是天然一体的。

> 编程最难也是最难统一的： 第一是开发和发布的断层，它使开发可以依赖通用语言，而不是发布上下文相关的语言。这带来了强大性，但也使人们分裂，因为通用语言必须用最原始的机制先后去抽象平台，人，最后才到目标问题和目标appstack。难度徒增。 在分布式程序中，这种难度尤甚。 第二，无非是对问题域的抽象。就像VB，抽象了界面开发为触发/触发处理。VB的做法向现实生活抽象，但是对于其它域，它就没有抽象的可能。线程能抽象并发，但也仅此而已。没有一种抽象习惯用法能统管所有，还很好用。设计模式也不能。

较shell programming，用传统方法作应用编程，没有直接一行代码如shell programming所做到的那样，一行代码敲下去，就能立马出效果的东西存在。但这极有可能是我们需要的最终唯一的东西 — 因为任何程序都会是一种执行体。我们只需“具体APP在具体OS下只须具体开发”

这是因为，我们业界，应用层编程始终还在用着实现层的东西。而且我们一遍遍用这种方法来解决我们的上上层开发。而且一直天真地以为：程序必须要预先编制，先编写好，后运行。—— 而busybox规范下的，系统和系统shell编程的本质，就是一对天然可以融合的东西，是任何系统应用编程的所有。自成生态的所有。独立，闭环，好用。

但是，千W不要以为shell编程就很简单，它不能称为通用语言和通用开发，是因为它仅仅不那么做，高级shell编程只限于运维，不像系统开发者，涉及到对内存的使用（有指针，etc..）产出系统程序，但高级shell编程的复杂度往往不比py这类脚本语言低，其语句，类型也颇为接近一般语言。—— 它的强大是因为它足够自我。

而且，它可用于任何编程。准确来说。任何系统应用编程，都可以是这种shell编程。

任何**APP**编程都是**shell programming**

想象一下，比如要开发一个这样的普通应用和准备一套这样的langsys：最终的这个程序，应该是一个类busybox这类可裁剪的的exe系统，demo as engine,搭配一个bash，就能写batch调用，bat调用即app，busybox就相当于一个函数库，里面的各个exe其实汇总起来组成了一个模块，只不过语言本身也是这个exe之一，app即exe的调用。函数即exe，参数即io，再也不用理解py那些理念懂基本的C系流程即可，开发上，直接用OS的观点来对应语言简元。比如，类似9P，任何一个语言元素都可以形成为fs的一个对象，everythingis a file,the programmingdomain programming is about file io。

其本质，就是把语言抽象的机制与OS系统应用的机制（包括语言，当时执行的上下文，这就需要一个容器保存）结合起来。这样，计算机应用直接对应开发。编程即计算机应用。编程即系统功能配置。以及问题操作，开发即运维。

上面所涉及到的要素，重点介绍以下：应用即容器。exe is componet,os is container。

1，它与容器搭配使用。

套间，接口，这不就是组件的那些概念么，可是传统组件没有os container保存不了部署上下文和运行期的那些资源，只有部署包含了，那么就能称为demo,否则只是一个执行体。这种APP是一个容器镜像，在展开执行时会产生一个轻量级的全套OS。每一个APP自带OS，是被正确部署了作为整个发布包的,运行时可以展开为设计好的全面运行环境，而无须host上的此类信息,app直接在这个guest上shell programmly运行。这就需要涉及到egxbox，在极小粒度上整合容器和语言的思想。这是后话。

2，它有组件系统

demos as engine,它也是可以被开发的。 类busybox，一堆exe像busybox中的子体一样，一个接口就是一个可运行件中的某个小运行件，在文件系统中，它是一个可以cd进.ls显示的文件系统。或许与常规fs不同的是，它会是一个executiable archive。是运行件也是开发件。任何GUI程序将基于CUI之上，CUI都是接口。

3，它是一种天然的分布式

由于开发件与运行件天然对应。且基于OS级，为了模拟类本地的host/host程序交互，需要sync2个远程guest OS来处理程序的内外上下文逻辑。如果二端的可以sync同样的结果。包括API。那么它就是一种天然的分布式和云开发的基础。所有的分布式，就成了二个OS间的通讯，而不是程序级的。因此分布式服务就能基于虚拟API进行。这是一种同步，当然也可以进行数据内容的sync，5G下这很现实，就像物联网基于container的应用发布一样，每天会产生几G的数据。

4，它有全面的开发支持，低入阶属性

由于它是纯粹的exe调用bat，将开发局限在具体app级又自带OS。因此可以在host端作为模拟器guest container实时播放。方便调试和再开发学习。playable类packer语言。demo playable类似按键精灵脚本，又类似editor的效果。对初学者入阶简单。

开发不仅是编码件，而且是调试件，这是一种良好的预埋debug middleware件设件，与带问题域统一学习支持的四合一开发方案。

————————

关注我。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

rcore,zcore,兼谈fuchsia:一种快速编程教学系统和rust编程语言快速学习项目

本文关键字：一种快速编程教学系统和编程语言设想,把**devops**和**hypersior**集成到**os**和**app,learn rust:the hard way**笨方法学**rust**

在前面《云APP，virtual appliance：unikernel与微运行时的绝配,统一本地/分布式语言与开发设想》和《一种设想：为linux建立一个微内核,融合OS内核与语言runtime设想》中，我们都讲到“语言机制与OS机制一一对应的运行时”的思想，这些属于平台与语言连接处的联系对于编程学习至关重要，直接影响着你在这个OS上学LANG写APPDEV的曲线，（正如“业务逻辑与APP编程”处的连接一样（《一种开发发布合一，语言问题合一的shell programming式应用开发设想》一文，业务逻辑如果采用尽量接近平台的某种逻辑，也会变得相对容易，这是后话，目前，我们仅关注前半句），这种思想得以最大体现的二个地方，一是发明一个OS，二是学习一个语言。

rcore,zcore：

拿发明OS来说，实际上oskernel作为对裸机编程的典范，也需要极高的抽象，内核的发明是一个重写语言的过程（这种工作却没有被用户空间的语言继续下来），采用什么样的语言来写内核，决定了工程的难易度，甚至直接决定最终设计，(比如windows在kernel中实现了一个对象系统，它是用C模拟的是在发明reimplent c，，如果采用的语言抽象层次高，这类事情成本就低，）比如，采用rust，它抽象高,batteryincluded，写出的应用代码量就小。甚至可以拉高抽象决定设计，做出language based os:一种general kernel design using language-based protection instead of hardware protection，libos。

> rCore是c版本的清华大学教学操作系统uCore(u:micro)的Rust移植版本,它里面的syscalls是兼容linux的，可以运行userspace的linux用户程序，后来，他们参考rust-osdev.com织织的成果，将整个rcore又重写为zcore，以接近fuchsia使用的zircon微内核，（Fuchsia 是谷歌试图使用单一操作系统去统一整个生态圈的一种尝试，类似华为的多场景OS鸿蒙），这种微内核特征在《一种设想：为linux建立一个微内核,融合OS内核与语言runtime设想》已经说过：拉高抽象结构，将被兼容内核放在下层用户进程级，这其实就是wsl1等 os subsystem的套路，也是libos，“一个进程一个OS”的思想，以及language based os。
>
> 目前为止，它们的代码http://github.com/rcore-os/中，有很多子项目(rboot,rcore-user,etc..)，最重要的当然是http://github.com/rcore-os/zCore，它重写了rcore为微内核zircon-object(包含内存管理和任务管理二部分)，有兼容以前rcore的linux-object和linux-syscalls,linux-loader，还增加了对fuchsia的zircon-object（除内存管理和任务管理的其它部分)和zircon-syscalls,zircon-loader。作为最终结果的zcore，可在unix(linux/macos)下以libos方式运行linux(make rootfs,cargo run --release -p linux-loader /bin/busybox)和fuchsia的程序(cargo run --release -p zircon-loader prebuilt/zircon/x64)。也可bearmetal以独立qemu运行(make image,cd zCore && make run mode=release linux=1,cd zCore && make run mode=release)。如果不好理解，你可以把整个项目成果想象成拥有wsl1 subsystem和win子系统的win10(wsl2改为更按近vm的实现方式)。
>
> 编译时，rust源码编译跟cpp一样慢，这是真的（不过想象其运行期安全和抽象在运行期0成本就忍了）。下载大量crates时。记得把register设为ustc的源。不过，我编译上述项目时，在osx14下运行cargo run --release -p zircon-loader prebuilt/zircon/x64和时cd zCore && make run mode=release都有失败。

以上都不是重点，这里才是：在重写zcore的过程中，他们发现fuchsia的微内核中的系统跟rust语言的很多机制紧密对应（非精确一一对应）。------ 除了他们的V3 rev文档库https://github.com/rcore-os/rCore-Tutorial中将这些开发过程都详尽写出来了，在https://rucore.gitbook.io/rust-os-docs/也有一篇《Rust语言速成》还提到，快速学rust最好的办法是发明一个rust os，顺便与C的交互也学习，探索高级语言的局限（而这在以前的c语言+linux这类教育中是不太可能的，因为前者抽象不高涵盖不了语言学习的典型场景而后者也不能）。。。

fuchsia其它特点:

继《一种设想：为linux建立一个微内核,融合OS内核与语言runtime设想》我们还发现了fuchsia中的其它特色。

zircon内核实现里有个虚拟机管理器：virtual app有时称为guest app，这一再印证，从《hyperkit:一个full codeable,full dev support的devops及cloud appmodel》一直写到这的“碎片化app，microapp，微内核OS，碎片化os，virtualapp”也正是zircon要集成到内核中的方案，记得华为的鸿蒙是“全场景分布式”，而最终，各种沙盒各种层次的虚拟化不就是为了使开发系统和部署APP结合，而且能做到更分布更细粒吗。只有细化才能分布式。

如果说云计算无O也就无APP，或许最接近原生的分布式APP：要算这种micro virtual app吧。----- 而其实统一语言，和统一内核，都是为了让真正的applvl的virtualapp出现。不过业界早有docker这样的暂代方案出现。

第二： Zircon 上的一层名叫 Garnet。 Garnet 包含Xi Core，the Xi Core engine for the Xi text and code editor. 它是Xi文本和代码编辑器的底层引擎，在系统中集成IDE，是为了降低可视调试难度。这接上了我《jupyter:devops tool with visual debug》《elmlang》的“开发要接上系统，集成在系统内部以降低难度，devops与rad结合”那些观点。

第三：zircon的flutter层面。这种用web前端技术完成的app ui，如果有了wasm的加持，可以出现在系统的各个层面，不光是web browser内。如果browser都虚拟化了，碎片化了。那么整个cli都是可以web方式开发的，共享统一appstack开发结构(一种ui决定一种app)。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一种云化**busybox demolets**的设想和一种根本降低编程实践难度的设想:免部署无语法编程

本文关键字：**shell language,debuginbuilt+google oriented programming practise+drive**。**programming:dgv programming**，限制源码规模。，**Debug appliance inbuilt**

在《软件即抽象》中我们讲到编程的本质是用抽象来构建图灵完备的概念体系，语言本身提供有限抽象，还负责映射无限的问题域抽象到有限的平台抽象域（各种语言在平台已实现了的lib,runtime,api,binary interface可开发件,对于云，只有api/api stub,无平台依赖），然而当代编程还停留在传统阶段，其过程和方法，依然是编码和调试以及维护，这导致了现在的编程依然很原始：

1，来说点抽象和科学层面的，由于组成编程的抽象本身是巨大的，同时要整合进语言平台问题和设计，所以编程复杂性始终无法脱离对其对那个工程规模级别知识库的东西进行背书，虽然编程语言编程工具和各种中间件越来越抽象，极大谈化了一部分抽象，编译器作为语言实现，虽然有宽松的repl和越来越强大的各种动态语言能力和运行时设施解放编译期检查，但始终只接受那些编译通过了的每一条语言元素。虽然有标准化但编程的软硬平台生态一向很破碎各自为政变动频繁这导致其附属开发也破碎，而且编程中很多部件和高层概念依赖很专业的东西构建，这些依然无法做到彻底抛弃和替换，使得人们难入阶即使入阶后依然不能举一反三（提供工具级和外来辅助手段降低编程入阶的专业度难度只能做到有限的效果。如果仅仅这样，后面遇到的曲线照样会陡峭。并没有越过曲线就一马平穿的道理，抽象依然存在）。

2，来说点具体和艺术层面的，对于具体实践中的编程编码和调试的难度也来自于组成编程各部件的固有复杂性，调试是驱动编程循环的，但语言基于数学和形式理论和图灵完备结果，平台最终是冯氏的内存加指令（大部分调试需要进入内存反工程，这影响我们不可能在一个更高的阶层上调试），平台加APP加语言运行时构成的那个内存中的巨大复杂运行时的追踪，从问题域映射到方案域一条路径上的所有细节，从搭建环境开始这条路径从开始到能看到结果很长，才能完成一次调试编码循环（devops正是为了解决这类问题，使得部分前置调试过程自动化，使得能看到结果的路径超短,不过这也是权宜。），足于把普通水平和没有耐心的人排除在外。

> 编程文科属性还是理科属性强一点？答案显然是后者，虽然编程语言有词法，语法，有点像作文，但实际上，跨过这些文法属性，属于理科的实验部分的属性还是比较强占比比较大，在编程中，需要不断调试，设立各种调试环境，追踪各种结果，验证想法。编程看来起像跟编译器对话，但是这跟自然语言完全不是一个级别(但是有一个例外：对于同一种抽象也存在多种表达法，类似文科中的写作文，每个人用于实现的中间过程都不相同，即抽象的多义性，但这反而正是工业编程中极力要避免的)，实践的10000小时理论+"无它唯手熟尔"对于编程来说不是十分适用因为我们往往把时间浪费在重复的事情上。往往对于一门语言，学了很久的语言高级技法，说了这么久，那么高级语法可能你用不着，实践中主要碰到解决的还是几大流程结构的熟练度问题。我们有时甚至要跟{}是否配对大战一天一夜。

------ 换言之编程的专业度和难度还是十分巨大，一直到现在，我们都没从根本上改变编程的形式和本质，只是不断变换抽象和堆叠抽象。所以出现了各种其它从不堆积抽象的传路路子，试图另找出路从根本上降低编程难度的创新方法，我们一一道来。

一些旧路子,及一种云化**busybox demolets**的设想

在全面降低编程复杂度方面，在语言级有很多我们耳熟能详的编程流派和学说(模块化，OO，etc..)。它们只是修补解决了一部分问题，比如对于敏捷方法论：，现在的编程基础如此，“一次设计”要完成的事情要把握的抽象幅度依然很多很大更别说往后的生命期内维护，对于最基本的模块化：编程是否真的可以像堆积木那么容易。但这或许正加大了复杂度，如果说任何一个小程序都需要大量逻辑的组合，那么人类始终只有一个限制：不能一次把握的大量的规模，接口，完成一次设计，这是一对似乎不能解决的矛盾。。devopsp这类手段其实其作用并不是级极的，它属于普遍的一种降低入阶和中间层的方法，上面说到，要做到降低整个编程难度，也并非上述降低入口难度这么简单。。编程所有的问题的复杂性是来自它的规模本来就是工程级别的。。。于是人们说在形成现在编程的所有基础和编程体制中“软件无银弹”。这依旧是文章一开始提到的编程复杂固有度问题。

然后就是那些高于语言级别的：

旧一点的有visual programming+rad，trigger editor,面向搜索引擎的编程，敏捷方法，serverless编程(这只是在devops减轻了环境搭建)，面向二进制编程(这类似基于接口的源码/业务逻辑抛弃型编程)，我也提出过一些方法，如《一种shell programming的设想》。《engitor:demo+debug driven programming》。甚至有基于程序员的契约编程（还记得py排版空格规则都是强制的么），

> 我们对于编程实践的讨论和降低其难度的尝试在很多地方都涉及过，如《xaas,appstack及综合实践选型通史》，和《兼谈fuchsia:一种快速编程教学系统和rust编程语言快速学习项目》，以及上面谈到的二文，我们可以将上面所有旧的路子结合起来：打造一种分布式busybox和一组云化cli demolet based programming设想：bash等shell语言可以说是busybox的专用语言，它组成了cli shell tools的可“编程”环境（可调用环境更准确一点），使得运维人员和非专业人员可以以简单的命令组合搭建shell脚本程序，完成较gui环境下更自由更强大的console应用任务。其本质是类似一种在内可脚本写作的可视化编辑器环境,如游戏世界编辑器（只不过不用trigger editor，而单纯用脚本,业务逻辑已写好并提供为cli）。在《一种shell programming式编程》中我们讲到，这是一种开发与问题合一的环境，于是这里的效果几乎结合了上面“可视化，demobasedprogramming,二进制脱离业务实现级别的编程”三种效果了。而采用的bash是一种真正的非专业人员定制语言，真正的非专业人员就要使用这种去除了严肃语言特色的东西。bash甚至没有模块。但这正是它的优点，因为这对于那个调试，还可以像shell一样基于pipe based调试而不是传统语言statement和module based的单元调试。每一句返回状态才能继续，这样调试

> 方便。前后无语义联系。一句一写的程序，而不是预先写成为单元。bash无模块也做到了控制逻辑的最终规模大小的问题（当然在bash+demolet programming这个层次上是不追求复用的，由于这个层次跟编写业务逻辑的严肃编程层断出了新的层面，因此它去掉了传统编程的难度和抽象规模固有问题）。
>
> 这里重点谈下面向接口，面向接口编程非OO语言用来实现多继承的interface关键字用于避免基于类的多继承缺陷的那一套，它更偏向指面向二进制的编程（面向组件编程面向构件编程面向积木编程），它允许业务逻辑（实现部分）与脚本逻辑分开（隔离并“抛弃”实现部分的源码，建立abi stub封装为大量二进制接口，接口专门可在另一级别被以“已部署”编程，新提出一个可编程层次，这个层次可以”无传统语法“进行，极大简化编程难度，比如允许bash轻语法语言+html标记语言这类用户级编程发生），建立接口的过程可以用类似busybox的方法批量生成。也可以是来自toolchain的一种自动支持。按二进制接口导出,最后的效果是你在类似c# ide中能实现直接查看dll透出的服务的效果。它一度被称为软件银弹真正解决法后来被脚本"源码as组件"和“面向服务,soa”这类东西模糊化。

本书第二大部分《minstack os : openaas+vscode os : debugable appliance builtin os for dever and common user》也正是为了提出一种这样的抹掉中间层复杂性的努力(它是一个三层结构：1，terralang + vscodeonline + openfaas cliiz backend, 2, distribute busybox tools,3, bash+apps+businesslogic languages+ more businesslogic demolets)，并没有提出彻底的解决之法，开头就说了它只是一种降阶法并没有解决抽象规模过大的固有问题，只是现阶段它可能会工作良好，所以我们叫它暂代方案。

未来我们依然需要包括上述方案在内的其它寻求全面降低编程复杂度以降低实践复杂度的方法

一些新路子：终极AI辅助和轻理科属性编程,literate+natureprogramming

新点的有：如ai辅助编程，，codeless无代码编程，literate programming（注意我不是指现在的基于xml和描述性语言的文学编程。），这些是区别于上面不能降低编程固有复杂度的根本性创新方法。因为它不能解决了改变传统编程中二大对象人和机的根本定位，这个努力的方向只能是AI,是一开始就是人跟人之间那种智能理解辅助级别的，比如，将编程语言的形式理论根本，发展为真正自然语言识别层次。或者提供一个智能机器人，其智能能达到和你结对编程辅导的地步。

那么，编程，能否真的有一天，能达到人人对话这个级别，彻底走过人机人编译器对话的原始阶段呢？大约等到共享汽车真的能做到在人挤人的街道上正常工作的时候，这个梦想就能有实现了(我是不太相信智能汽车的，智能有些东西在任何学科包括哲学上都处在没被解决的阶段)。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

mineportal个人云帐号云资源利用好习惯及实现

本文关键字：**mineportal,owncloud,**个人云,私有云

云应用和大数据时代，我们要怎么强化我们现有的个人帐号方案？

现状：

opensocical id之类的东西还没有普及使用，google wave也没有成功，gravatar也只是内部帐号，所有的帐号类的sns产品都不足于一统江湖变成帐号标准，默认情况下，人们常常借助手机号，好吧，还有一件大件，即email系统来标识身份。也许手机号和邮箱号，是二种常见的社会ID标识了。 PS：当然，那种要求上传身份证和cer id的是强实名制网站特有的。 在云时代，除了这些传统应用（邮件，即时通讯），社交，不同的平台要求注册各种内部ID，帐号是ID，然后与邮箱与手机号绑定，手机号通常用来收取各种验证码以证明你的ID有效性。 通过这种验证，我们被各种平台假设为外部手机号和邮箱后面对应的那个人。 问题来了： 我们使用的邮箱，包括手机都是由第三方运营的，是极易发生变动的（换号，邮箱费弃）。一旦倒闭，你就失去了再进入这些帐号的重要通道。

方法：

其实我们可以买一个域名，加上一个免费的企业邮箱，这样就有享之不尽的以域名为ID，邮箱号为前缀的专用云帐号和ID了，而手机号可以只保持一个。这样就不必与别人争帐号，为同一个名字后面加什么前后缀方便记忆又能标识清楚而伤脑筋，不会因为这些平台倒闭而影响分散在各处的帐号的访问。因为我们的域名总能绑定一个企业邮箱，这样无论我们的邮箱托管在哪里，但是邮箱地址是永久性属于自己的。 其实，很多地方也都要一个域名，比如一个姓名域名。更别说为了以后做站，或考虑或绑定接下来要说的owncloud，而一个域名一年只要50元左右，便宜点的不到10元一年。这是最值得投入的支出了。 那么我们为什么还推荐需要一个个人云呢？

现状2：

传统的计算机日常使用会产生需要随手带走的数据而U盘之类的东西显得太繁琐。而云就像一个大计算机，服务端应用和客户端APP产生的各种应用都会产生数据，且都需要被统一整理。 在PC上，我们个人的产生网上数据主要是一些通讯录，照片，收藏的网页地址夹，一些个人文件，工作资料，这些都需要统一收集并备份。 在手机上，各种手机APP都会产生数据，如果是品牌手机或装了特定的andriod定制os，就会有备份服务，比如ios有icloud，aliyun os有aliyun数据服各，总而言之，其上的数据托管在第三方平台，而手机上的资料最容易丢失。一个这样的自动存储后端似乎是必要的。 网盘即是这种同步机制的核心。通俗上人们叫他个人云，实际上可以理解为云数据自动同步存储后端。这跟手机安装一个网盘客户端同步日常产生的照片等是一样的效果，当然，专门的个人云是附加了特定功能插件的，比如同步通讯录什么的。比一般的网盘也更强大。

问题又来了:

依然是因为这些云备份大都依赖第三方提供的服务。而各种第三方网盘，要么速度有限制，要么文件有限制，动不动给你删文件，最大的问题：不够安全。各种照片门XX流你还见得少吗？ 所以一个受控的，且数据保存在自己服务器上的个人云，如果能有，当然是最好的。

实现：

owncloud就是我们需要的一个像这样的个人云之一。来存储和同步我们使用云资源时产生的文件和档案。且充当个人云后端。 对于个人云，网上有很多种方案，但是综合起来，owncloud是门槛最低且符合我们利用百度云这种网盘习惯的，而且owncloud有各种插件可以模拟备份你的通讯录，工作文件，笔记这样的主流功能实现，而且对于需要开设网站的，它甚至还可以成为我们网站前端的存储后端。 PS：而且owncloud不妖气，不一定要用VPS或云主机来承载它，一个比较好的php虚拟主机就能用，但是要求空间大，否则作为网盘没有意义，最小5-10G吧最好大于等于10G 好吧，minecloud是一个整合了wordpress+owncloud的个人云，这样的云，不光owncloud可作为个人云后端，甚至wordpress产生的，调用的数据都可以它为文件存储后端，维护起来非常方便。 不防看看，我帮你打包好了: 下载地址（本站下载）：

http://www.shaolonglee.com/owncloud/index.php/s/6EWog19iy7OIXGF

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

mineportal – 一个开箱即用的wordpress+owncloud作为存储后端

本文关键字：**mineportal,wordpress+owncloud,owncloud file backend for website**

亲是不是在维护一个网站？ 普通的网页程序如基于mysql的网站程序，导致了越来越多的问题，比如。 将什么都放在服务端。界面放在逻辑中。 维护比较困难，数据库不容易备份。 管理在远端。内容在远端。用户也在远端。脱网了完全不能参与。 这其实也是所有远程程序的特点，只是WEB这种远程程序更为特殊，它将用户端也放在远程。 解决的方法在哪？有没有像本地程序一样发布部署的网站程序呢？ 来看一些例子，比如p2pforum,静态博客。

p2p forum

osiris，一个去中心化的p2p论坛，这是很久以前的方案了，它允许每个用户都保有独立的论坛备份。下面是关于它的一些转载介绍： Osiris 是一套去中心化的分布式论坛程序。GIT式的内容控制能力和信用的继承功能,让它十分有趣。程序使用 C++ 编写，采用Python作脚本引擎。

- P2P – 公钥加密系统配合P2P技术，让每个客户端都拥有(数据)完整的论坛内容拷贝，离线一样浏览，不怕论坛蒸发。
- GIT式的内容控制 – 允许客户端通过信用系统将不速之客拒之门外。客户端秉承“不信谣、不传谣”的宗旨决不储存或转发丢弃的数据。
- 信用继承 – 如果愿意，信用可以继承，朋友的朋友是朋友，朋友的敌人休想在你的拷贝中发言。
- 支持代理 – 但 p2p 和 bootstrap 过程的代理不是分开的。 不难想象，每个用户都可通过信用系统有选择的将本地论坛拷贝 fork 成自己喜欢的讨论版本。最后论坛的各个小帮派间看到的内容也许会大不相同。

静态博客

什么是静态博客？ 静态博客有着比普通动态博客绝对的优势，速度快，而且使用先进的GIT分布式文件管理系统。常见的静态博客例子和程序gitcafe pages,github pages,jekyll,peclian,Hexo 所有的程序全是静态页面生成器这样的传统本地软件逻辑，，业务逻辑也全是本地逻辑。不再负责产生动态页面。 将中心逻辑维持在git本身。本地程序，仅通过cgi出网。web仅作为界面，不再是一种程序模型。 无数据库。数据安全，可以实时随时备份。网上网下一致。本地和远程同步。 去中心化。即使脱网，用户可以在本地长时间操作，等有网同步即可。 有哪些优势？ 这类博客维护简单，书写便利，发布的过程是离线修改博客的备份，然后同步到远程GIT空间即可。不需在线操作，且本地可永远持有博客的一份离线备份，不怕数据丢失。而且由于GIT的分布式文件系统特性，可以直接当网盘用。同步/拉取即可，省事快捷。 特点：交互式欠缺，普通的Web程序因为有中心数据库所以交互性尚可，而基于GIT的只能外挂交互式程序。，论坛，wiki这些程序难做成类静态博客形式，这类程序也不可能慢慢取代动态程序架构。变成未来流行的web程序托管和部署方案。 就像P2P forum也失败了。

新的方案：wordpress+owncloud

可以统一备份文件。且后端可作为文件模式存储。不再是数据库模式。交互性也得于保持。 另外，后端也可独立作为一个网盘程序。 下面是保存有shaolonglee.com后端文件的owncloud网盘。

下载地址（本站下载）： http://www.shaolonglee.com/owncloud/index.php/s/6EWog19iy7OIXGF

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

mineportal2：基于mailinbox，一个基本功能完备的整合个人件

本文关键字：**mailserver backed new mineportal,**邮箱附件外链，**owncloud backend static web hosting,**阿里云省事建站，如何借助**mineportal**低成本把虚拟空间+**email**玩成个人**portal**，**one place webapps for personal use**

在《发布erpcmsone》一文中我曾谈到我缺少一种能发现我需求的网站程序 — 能把我日常要用的东西整合到one place且生态统一易扩展的东西,我以为odoo是这样的产品，但现在看来，它其实更像是一个groupware for team use，在《mineportal:个人云帐号/云资源利用好习惯及实现》和《发布mineportal – 一个开箱即用的wordpress+owncloud作为存储后端》我谈到适合个人用的网站程序选型 —- 比如那些个人用户对于互联网的起点性需求：比如邮件,IM，存储，网站和网站存储，等等一个人急迫需要而不用依赖第三方的东西，能整合起来kept in one place随身带走takenable away的那些，,基本上它们是一些网上能发面的破碎的实现品.

vs mineportal，我把它们称为个人portal2，其实所谓portal,它是一个联系线下到线下的入口。强调这个入口是尤为重要的因为它是portal所指，比如存储和邮件是这个入口以下的私密部分，当它用于网站时，存储可当图床，IM可当网站聊天框，还比如，它能真正当portal使用—类opensocial用一个ID作通往各大网站的social connector+自动注册以代替原始的电邮注册，手机号注册，—— 可以看出，这些不全是web方式也不全是线上的。—- 最重要的一点：因为还有很多功能要扩展，所以最后这些都要生态统一，且可被扩展。最好有清晰的appstack划分：

什么是appstacks:

一般来说，用中间件的视角，可以将语言，应用，基建服务都实现为中间件。应用那部分可以做成appstack. 应用层的appstack中间件，及这些中间件规范出来的应用级开发发布支持：这些程序使得面向它开发出来的东西直接就是app，使得开发和发布变得micro app/service的东西。它的存在，使得软件的开发件和发布件之间有了一个中间层。 appstack中间件，可以使得语言系统中间件，xaas层可以与appstack层分开。并且，独立出来的appstacks可以做成组件，所谓组件，就是可以被开发层内欠调用的运行件，并且可以做成原生实现，这样执行效率很高。比如游戏有gameappstack，这样开发游戏扩展，就是写一些有限的脚本。 比如nginx是一种xaas/iaas，mysql是一种common storage appstack,,kbe是一种game appstack，blaaa 在这种情况下，那二篇文章中谈到ocwp有些局限，它仅能提供网盘和网盘存储，终归它缺少一些重要的东西如邮件,ocwp只包含oc+wp自身。从整个ocwp产品体系看nginx不算其appstack组成只是视为xaas/iaas，，虽然围绕oc可以扩展更多，oc有插件可以外挂external webmail client进来，但那始终是外挂。没有至关重要的邮件产品，不如做成自包体，比如将mail servers包含进来做成appstack.

为什么集成了mailservers的ocwp不是完善的portal

那么，如果将 mail servers集成到ocwp呢？如果是内置的，我们可以直接在ocwp中加一套mail servers，如果是外置的，我们可以比如，让oc支持从php imap扩展中读取附件 — 比如，用fc_mail_attachments和mail attachments这样的owncloud插件将你的EMAIL空间变成网盘，我还看了一下如pydio imap也支持，这基于以下一种事实：imap协议可以允许文件夹里的邮件带附件，且邮件是天然的消息系统，把邮件当实时消息，我们就得到了另一种sns，反正，什么都能围绕消息体和附件展开 —- 就这二块就足够让email based成为一个personal portalware。在使用上，一些邮盘客户端如imapbox能做到同步（虽然并不是那么完善），基本上能用邮件收发模拟发贴。比如邮件列表可以做成内部论坛如googlegroup之类的东西，

很久以前，我曾很崇尚这类邮盘之类的结合体和imapbox之类的产品，可是它的缺点在于：它改变了应用方式。一切既有实现都颇为不完善。单纯以邮件为后端的模式也不能提供如网站托管这样的个人portal应用，比如没有www件的支持，它不能真正让附件变外链（上面的oc to imap插件只是将imap里的附件镜像到了其内），邮盘空间也不能hosting website。

其它例子和the last saver：

再来看一些其它例子

“SmarterMail: A revolutionary secure email server with file storage, group chat, team workspaces, sharing and more”

缺点：虽然是appstack以下的，且功能完备,15.x及以下的版本占用资源少，但是它附件共享支持有限，不能供网站当文件夹图床用。而且它是闭源的。

“Like Syncany, Cozy provides flexibility to user in terms of storage space. You can either use your own personal storage or trust Cozy team’s servers. It relies on some open source software’s for its complete functioning which are: CouchDB for Database storage and Whoosh for indexing. It is available for all platforms including smartphones.”

cozy有点类似sandstorm，它不是应用，是paas,对nginx,langsys等进行了集成，不好归纳出一套appstacks。且cozy bug太多占资源太多。不过其内部apps有很多social connectors，这是亮点。

而一些纯工具性质的也不行，它们太简单不足于组成一个appstack及其下的app扩展： 如一些聚合工具，如thinkup php，it should be a standalone portal but not just a tool that can collabrate all these infratures. 或单纯的webdav,webdav is just a sync proctol, oc has its own general storage cheanism,,oc is more a groupware framework than a file sync server,,,we can build games around it 或单纯的客户端实现imapbox,what i need is not barely a php/phalanger imap attachment outlinker/saver prog.. in the totoal effort to mimic what all imapbox does

最后我找到了这个：

mail in a box: Take back control of your email with this easy-to-deploy mail server in a box. … DNS configuration, spam filtering, greylisting, backups to Amazon S3, static website hosting, and … Mail-in-a-Box is basedon Ubuntu 14.04 LTS 64-bit and uses 如果以它为基础重新建构mineportal2，将基本满足一个人上网存储，简化登录，将所有资料和网上活动最大聚集到一处的基本需求。它的优点易于搭建虽然它本身只是一个工具，可是着眼于组成它的整个体系可分离出以mailservers为主的appstack和与oc为主的app，且功能合理扩展性到位：

1，默认是没有wp的，它支持从oc文件夹进行static web hosting，且支持在oc内部通过ownnote,note这样的app直接保存为.html通过static web hosting输出，有了oc支持的后端app支持，也这并不会失掉去除wp后减少的交互性(可以调用oc的聊天，评论系统等)。

2，而且，oc支持好多social connector自动登录，如果联接的平台越来越多，它最终会变得像个人版的passwords repos，谈到这里要谈到手机平台上的微信登录等，微信是手机APP方式的自动登录，跟web通过api跨网站登录的方式有点区别，然而，oc本身是比微信要强大得多的，要说有差异微信是以sns而oc是以存储形成生态，实际上微信小程序这样的东西，就是相当于oc的apps，而OC比微信要更有资格担起personal portals，这个”portals”的角色与任务的，因为微信缺少至关重要的一部分，就是它整个都是公开的。没有私有的存储部分。而且它一切都是线上。

————————

mailinbox在aliyun上用脚本安装会出现service postgrey start failed的bug，自己处理一下，还有每用户static web hosted的地址user-data/www/(username)可以做soft linking到owncloud具体用户下。还有sqlite是默认的数据存储方式改成mysql+redis可以大大提高体验。最后，整个程序是unix系的，或许以后我会把它搞成windows版本。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

基于openerp的erpcmsone：可当网站程序可当内部知识管理及ERP,及前后端合一的通用网站程序选型,设计与部署

本文关键字：**openerp website,erp backend website,erp based portal,netdisk storage based blog,uniform www and erp system**

作为一个自己不断尝试建站自用的vps和虚机重度用户，我需要的是一个外能当个人博客用内能当云存储使用的网站程序，这个云存储要能以不同逻辑方式存储我的不同工作生活资料比如内部可写日记可存文件可同步书签，内部网盘系统的文件可以外链供网站使用，你可能看出来了我需要的是不仅是一个云存储，还是一个个人知识管理型的内部系统及图床(可能还会有更多。。。),我对这个程序的要求是它最好是一套独立的程序或几套相似的程序（我可能会对这套程序作二次开发，所以它不要太拼凑太碎片化，不要涉及到和跨越太多语言环境）。

一台云主机几乎成了我的第二电脑。相比之下，手机，家里的电脑成了终端。我给这台主机装上了WINDOWS（与我本地机一样用的一体化0pe+nt5系列），装上了wamp(我的一体化开发生产msyscuione)，和相关程序。我有一个大约4G大小的个人资料库（按media,softs分），都是工作生活中积累起来的资料，这个第二电脑我把它打造成我的建站和云存储器。 可是于知识管理我没能找到过一个像样的实现。所以我至今还在不断换站和寻找。。我尝试过WP+OC的组合，GITSTACK,SEAFILE等，没有一个独立建站程序能让我认为迎合，发现，了解了我的需求，而我也不知道精确告诉自己需要一个什么样的网站实现原型，这很奇怪。

毕竟，建站，是一个被深入研究过的领域，各种开源商业的实现品也很齐全，一个个人网站系统，必须在其内部系统提供至少涵盖日常你需要用的那些方面比如当个人网盘或管理书签用的那些吧？一个公司站，后台就不应该只是管理前端公司资讯CMS的某些条目定义，最好，它还要提供实际业务管理接口什么的或职员在线聊天什么的，毕竟它不能只是企业网站或管理平台，最好是个ERP，吃惊？是的，你没听错我就是想它是个ERP因为我是个小部门经理我管理一个网站我还管理工作在这个网站的某些号人，而我一直也在找个能管理自己学习工作资料的面向内外的一体化ERP而不满足它仅是个资料同步器或企业QQ之类的聊天+文件传送IM充当ERP！

一直以来，我们接确到的网站程序通常都是作为门户展示portal，个人知识管理(blog,wiki,etc)和用户交互lobby(maybe some页游登录前端)或社区用的(bbs,etc..)，它们都是公开性质向外提供WWW服务，存在推广需求供搜索引擎抓取的，供内部网使用且同样采用web b/s系统有：基于WEB架构开发部署的ERP，内部交流系统（Mahara,etc..），企业或自用个人网盘系统等，它们往往不提供对外服务及推广，不开放80端口，当然按其它分法还有更多其它的网站系统。。。。。。可是往往，综合对一个网站系统的前后端考究，你会发现有强大后端的网站系统并非一个全能的在线系统，一个CMS可能只是一个壳，而太多其它同类CMS实现往往聚集着重复的功能，一个据说有众多PLUGIN的BLOG体系刚好缺少了某些。你听说过的所有建站程序没有一个基本涵盖不了你日常需要应用到的那些基础方面，你把分散在各种的程序集在一起勉强让他们一起工作却各自要求不同的环境显得格格不入，要么一个程序无限深入，却在主干上漏了点什么至关重要的东西，没有一个像样的综合建站系统可用。

无论我们有没有觉察到，所有稍微大点的，完备点的网站应用体系，都一定存在一个对内对外的边界部分，都定义存在如上功能上的明显的对内对外区分点，外部就是对外提供服务的那部分，部分业务逻辑加界面逻辑对外的那部分，就是WWW 80，而内部就是用网站程序实现的部分或全部业务逻辑,部署上它可以是分布在其它机器上的非80服务或需要认证的部分（lamp中的mysql，phpmyadmin都可理解为此列当然重点在接下来这句），——也可以是通过后台管理系统之类的东西隔开的不外露功能部分。

可为什么这些不能整合呢？你可能会说我的需求太奇怪但它真实存在。你可能会说ERP只是管理软件中的一支，但实际上它的外延可以理解成我需要的以ERP为backend的后端个人知识管理系统+前端博客网站系统（如果公司系统也能适用那么它就是通用应用场景下的内外端兼备的网站系统），为什么不可能呢？ 通用的网站是什么？有没有W用的网站程序?真正一体化的网站程序涵盖个人工作生活那些需求？公司站呢？

如果你到达我的需求场景抵达过odoo设计者们的到达过的境界，你就会发现odoo（原openerp）从主干上提供的这些刚好是个完美的设计，erp+cms的风格和用法让它一点也不显奇怪，在遇到它之后我迅速明白于我的需求erp+cms就刚好，并只能选择跪拜它的设计超前或刚好。

uniform frontend/backend website ecosystem:

网站的前后端可以放在一起有时甚至被提倡这么做，只是历史上被忽视，这是为什么呢？因为它们共享大部分极其相似的东西

这就是说，后端，其实一切网站的功能，它们本来就是一体的是同一个技术和应用下的东西，本来都可以不分前后端整合在一起，只是人为进行了简单地策略上的隔离（故意不故意地这么干或者是后端或前端选择性透露或前向后隐藏而已），所以这是一种本来现象的非本质区分，下面说下：

应用上，网站的内网环境和外网展示，往往是有联系的，比如，网盘系统可以共享外链的方式做成前端外网服务的贴子附件。近来基于GIT HOOK的网站系统，基于MYSQLFS，MONGODBFS，S3，OSS的存储后端建站系统，都表达了这个鲜明的潮流，这就是说它们和应用上都存在极大重合，故可整合

开发上，大凡WEB架上的程序，其主要开发范式就是MVC，总结起来，CMS是一种最普通的网站程序，BLOG，WIKI，FORUM都可以用它来解释，这就是在前面说到的，mvc考虑进了对前端的设计，把逻辑和界面作为开发端的全部工作。网站设计其实存在一个通用领域抽象，这跟传统桌面引擎开发一样，有“网站引擎”之说，而所谓后端，除去业务逻辑，与网站相关的都是MVC，所以，它们在技术上存在本来的重合，故可整合。

部署上，其实所谓后台，其外延是很广的，可以分开被部署的部分也很多。这个区分点隔开区的后台系统部分或者服务部分，都是可以被分开部署的部分，oc整个系统除了网站系统的后端其它都可分开部署。附件存储可用WEBDAV部署在其它机器，或者oc整个codebase可以做成websocket服务器，这样webclient前端可以分开部署，只要提供服端IP就可以静态文件部署在静态空间或CDN环境。当然这些是设想。

any erp any website:

在我以前的贴子中，《利用wp+oc打造个人全功能网站系统mineportal》，在这里提到了二种需求，个人BLOG加个人云存储，这二者基本涵盖了个人在网上自建必须的那些自托管网络服务，分析一下这里的前后端，前端无疑wp,后端主要是oc存储服务，个人日记/照片/书签/联系人同步服务，要是前端wp和后端oc存储可以结合，即oc作为前端贴子和日志文档化存储方案就更完美了。

在这里，oc纯粹是被我当成个人erp被使用的，那么对于不局限于集成以上这些服务的这类个人网站比如把它当ERP还需要提供什么呢，至少应该用它来管理工作中的资料吧，比如，个人可以模拟一个有限的ERP环境存储每天在公司工作中产生的资料，做到自备份的目的 — 就可以免除光靠网盘同步或公司企业QQ之类的东西了，wp+oc在这种需求下是无能为力的。openerp8.0即可。

openerp,在8.0之后，openerp提出了前端与后端一体。前端主要是后台网站模块包括博客论坛门户这些常见的支持。当然它还没有集中化的附件（这在openerp中称为文档）管理和类似Owncloud client的东西，但是我想这在以后的oe未来版本中或许能看到。它最重要的部分-oe对erp的支持可以满足模拟一个私人自管理工作的需要的那些场合。

oe8与通用网站系统设计

openerp8就是我一直在找的通用网站系统。这是我现今最想得到的网站应用的正确集成和打开方式。它自成一体不碎片。 一个“通用需求网站程序设计”它的某个实现应该首先提供基础部分+然后才是完善和扩展，这就是说，总算有一种网站设计，它能符合个人或公司绝大多数同时面向内外部网络的WEB应用。而网站应用的正确打开方式和使用姿式应该是这样的uniform frontend/backend website，至少也应该是某种backend backed frontend website.就像oc8这样就刚好。

下载地址：（我已把它做成了msyscuione下的一个可选包。将程序主体保留在msyscuione的app and appdata部分，将混合架构保留在appstack/wnp下）

下载地址见源站文章链接。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

免租用云主机将**mineportal2**做成**nas**，是个人件也可服务于网站系统是聚合工具也是独立**pod**的宿舍家用神器

本文关键字：利用包含**nsd**的**mineportal**将个人**pc**打造成**nas,apache**的**oc**透露**owncloud**静态网站服务,发布**portalbox**，**based on colinux and mailbox,**一个网站和个人件，**hostos**与**guestos**的最佳组合

在《发布基于openerp的erpcmsone》时我曾谈到我用一台ECS作为我的服务器，那里我同时说到我把PC当成了跟手机一样的终端，一个中心多个终端这种星型设想本来也符合很多现有IT布局的情形 --- 比如NAS使用的场景。但云主机空间存储不大，作为一个需要大量存储的用户，虽然阿里云几十G的默认空间当网站空间+网站存储空间也是可以的，但是要当成nas存储大量资源显然不够用。

那么，能否把PC转换成服务器而不是终端，当成NAS呢？比如:可以直接在个人PC上构建这样一个服务器中心(这台PC可能在你上班时租住的宿舍里，可能在你家里)，再买一些大硬盘（现在硬盘的价格也低了，个人购置一个2-4T的存储基地也不是什么不可能的事）进一步地，仅把pc富余磁盘空间当静态存储/同步服务器空间终究太无趣，我们还可以把它当网站空间和可安装各种动态扩展的服务器空间。比如我们可以在其中装一套nas管理OS，这样大空间和大存储都有了，俨然一个paas了。(当然，装在不常在线的家用NAS上始终跟ECS这样的环境不能类比，比如家用nas当网站服务器是不能持续24小时提供服务的，但我们可以做一些强化达到同样的目的，以下会谈到)。

首先是为这样的nas选一套OS，在《一个设想：基于colinux，是metaos for realhw and langsys，也是一体化user mode xaas环境》我们还说到hostos/guestos不宜选virtualbox这样的东西(why not virtualbox,why not sandstorm paas,why not docker,利用guestos as container无限开,colinux配置文件中有资源配额字段，etc..)。而其实每一台pc都有服务器和客户端的双重角色，一台服务器也是如此，比如你需要一个guestos，且在hostos中管理guestos的能力，基于colinux的mineportal刚好满足和紧密对应这些它以带GUI的windows为shell以monolith一块的linux整体为服务器，这样就省去了通过类似web管理lnmp件的方式来管理服务器。在开发机/服务器上，这种布局也很流行(vagrant)。基于maininabox的内置程序有lnmp等，还提供mail,cloud storage,static website hosting等等，可以为这个nas提供最基础的个人件设施,maininabox也带dns服务器，是建立nas的重要条件。

所以我们的总需求和设想是，不仅把PC当NAS，而是把PC当成集服务器和终端的双重体，借助colinux+mineportal2的方式，将其功能发挥出来，只不过这次不是装ECS上而是装装在PC上，当然我们不能做到像黑群晖那样完善，我们就现成的地用mineportal2，把这个portalbox打造成nas而已。

步骤和强化

1). 磁盘布局：

现在的笔记本一般有双盘位加一个固态SSD系统盘，三个盘，我们可以将作为hoster的windows系统安装在SSD上，其它二个2T的盘，一个格式为NTFS，为D盘，一个格式为EXT3，在windows下隐藏，往其中灌入ubuntu14.04.vdi那个文件中的所有内容(先启动现有的colinux，挂载上2t盘，格成ext3，然后cp -ax / /2t，这里/是现有colinux根cobd0，/2t是2T盘挂载的cobd1，注意拷完后在新的/2t中修改fstab调整成正确的cobd xx)，2T的盘格成ext3比较费时，记得加T largefile，这样就快多了，把colinux放在C盘与windows,user,program files等放在一起，安装colinux为服务(那个colinux-install-service.bat，然后服务管理器中由手动调为自动)，修改好ext3分区上/usr-data/owncloud/config中的oc地址，再安装个owncloud客户端连上https://localhost/cloud随机启动，将C盘做成hwindowsgcolinux.gho的备份放在别处。

不认4G内存的处理和解决方法：

windows2008r2之后没有32位，而这之前的所有32位都存在不开启pae识别不了4g内存的问题，经测，win2008 32位在4G内存机上会认出完整4G内存（显示在我的电脑属性页），但这会造成colinux启动蓝屏，这个时候，bcdedit /set {current} truncatememory 3221225472，把4G截断为3G就可，但win7 32就会认出标准的该截断3G左右的内存，在其中执行colinux也往往不会蓝屏，win2k3也可以。

在使用的时候，D盘的文件可随时备份到2t盘所在的nas内，供远程访问，这样整台PC就是同时终端和服务中心了，且避免了设置raid备份的需要，因为windows中的那个2T和服务器整理的文件是有二份的，只要你不删，是可以随时双向恢复的，且笔记本的电池可以当ups用。你可以用https://localhost or 局域网IP/cloud这样的方式访问这个nas,当然这仅限本地，和局域网环境里使用手机终端streaming video,pics的情况

2). 然后就是使内网nat中的colinux对外开放(家用adsl路由方案往往如此),和开启你PC上的远程桌面：

我们可以在家用路由器上用静态IP设置NAS所在的那台PC，然后将这个IP透露成dmz主机，再开放3389,80,443等转发端口。最后在路由器内部申请一个oray的花生壳 DDSN域名指定到这个IP挂上。这样就可以在异地远程桌面和登录/同步nas https://你的花生壳域名/cloud 中的文件了。

3). 一些强化：

为了不用常开机。你可以设置网卡远程唤醒，或买一个极路由带远程开机助手的，不过这往往有点麻烦，最省事的方案不如买一个智能WIFI开关插座，设置电脑通电就启动，然后远程控制通电/断电/开机。

你还可以自建DDNS服务器，代替花生壳在上述方案中的作用。

as pond，以促成可用的网站系统**:**

用以上这里的nas方案做偶尔远程开机的同步服务器是非常理想的，因为同步毕竟不需要一天24小时开机，但至于用nas做网站系统就不能保证持续服务了，所以我们的方案是：

不直接用这里的nas当网站托管体，而是用这里的nas当远程网站的同步体和辅助体。什么意思呢，远程网站可以是一个很小的php虚拟空间装上一个owncloud，并通过oc的external storage插件把它外挂入我们的这个家用owncloud center repo，这样，我们就把这个虚拟空间当成了类手机一样的终端，参照我的《在owncloud上hosting static website》，在远程虚拟空间有更新时，我们可以把新增/改动的那部分手动复制到家里那个oc center repo，进行二个repo之间的对拷，当然，手机oc客户端是支持auto upload的，这个需要我们手动拉一下文件。

这就是说，静态网站系统生成的那部分才是我们需要备份的，至此，我们可以对静态网站和动态那部分作个概括了：一个网站系统的动态部分不需要透露给用户使用、只供管理使用和功能使用，静态部分仅供展示的，就是我们这样的oc托管静态空间。至于在虚拟空间端，我们需要设置一下.htaccess（如果是apache）：

apache的oc透露owncloud静态网站服务

```
<IfModule mod_rewrite.c>
RewriteEngine On
 RewriteBase /

 #绑定域名到具体目录，事先将owncloud的data弄得跟owncloud目录并列，将data设为0755，去除oc的htaccess
 RewriteCond %{HTTP_HOST} ^www\.shaolonglee\.com$ [NC]
 RewriteCond $1 !^(owncloud|oc)
 RewriteCond %{REQUEST_URI} !^/data/minlearn/files/~www/posts/
 RewriteRule ^(.*)$ data/minlearn/files/~www/posts/$1?Rewrite [L,QSA]

 #静态html,记得在管理器中把index先htm第二php
 RewriteCond %{REQUEST_FILENAME} !-f
 RewriteCond %{REQUEST_FILENAME} !-d
 RewriteRule ^([^\.]+)/$ $1.htm [NC,L]
</IfModule>
```

终于，我们知道为什么把这套方案称为pond了（我们不知道为什么喜欢pond这个词，感觉它表达的意思是指去中心化之后形成的一个一个孤岛，本地数据中心，联网时可以在线，不联网时就不用对公透露，P2P的个体），去中心化免ECS投入是这个nas的中心价值。所以题目称，这是一个免租用云主机将mineportal2做成nas，是个人件也可服务于网站系统是聚合工具也是独立pod的宿舍家用神器。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在tinycolinux上编译odoo8

本文关键字：在**tinycolinux**上源码安装**odoo8,**动态模式**python+uswgi+nginx,**精简安装**odoo8**模块

在前面《发布基于openerp的erpcmsone》时，我们谈到openerp其实是一种后端erp前端CMS的东西，其网站模块部分是通用cms网站选型的技术楷模，有可视化拖拉建站支持，且可集成后端erp部分（在线聊天啊，联系表单，购物车模块,etc..），当然，谈到cms，不是说它就是前端，它反而正是属于odoo后端支持部分的，只不过其展示部分是前端技术：

什么是CMS呢？所谓CMS，隐藏在内容管理系统(CMS)之后的基本思想是分离内容的管理和设计。页面设计存储在模板里，而内容存储在数据库或独立的文件中。 当一个用户请求页面时，各部分联合生成一个标准的HTML（标准通用标记语言下的一个应用）页面。对于一个CMS，其后台admin系统就代表了它的技术全部（负责内容模型表示和前端展示）。所以我们这里说到前端就是指其生成到html的后台模块支持部分等等--- 这跟不生成前台静态页面的前台纯动态交互网站需求不一样后者不需要html化。----- 在前面《发布mineportal,ocwp》《让oc hosting static website》中我们都谈到静态网站，它其实全称是静态生成器网站，它其实是CMS工具化了的一种形式。

而且，odoo还采用了pgsql，从Postgres 9.x开始，Postgres又添加了激动人心的NoSQL的支持,,Postgres是通过添加一个json(jsonb)数据类型来实现文档型存储的。这迎合了采用统一存储后端的设计，可以使得odoo的document模块使用分块filestor文件系统，见《发布mongopress,基于统一的分布式数据库和文件系统mongodb》同类文章。

最后，odoo采用python，要谈到语言的优异对比足于掀起大论战了，我不重复那些聚焦语言内部如何pythonic的老话题，只讲几条外部特征：

1，C系和原生程序，是基本所有现实中可见系统实现的基石，但C系不一定就是最好的，都是先用起来的实用主义的产品，而python，就是所有linux发布版事实上的脚本语言环境。

2，在语言选型上，虽然工程层面是提出越来越多的脚本语言来支持各种domain，但其实历史上还是倾向直接一门丰富langtechs语言支持库级表达的DSL，这也是为什么历史上众多语言很好地完成了某领域部分的事现实上在其它领域不好用，但还是会宣称它自己是通用脚本语言一样。比如php不被用于作非WEB开发，其它语言不常用于自然语言处或科学计算等等，python虽然也不够通用，但事实上它的应用领域最通用。

3，在语言选型上，工程上是提倡越来越多的语言，但具体到人和学习者，我们一般倾向于只学二门语言一门C系必学（C or c++），另一门应用脚本语言，且这二种语言形成one host one guest的only two选型特征，根据2中提到的二种语言要面向DSL包纳越来越多这些特征，lua虽然精微与C一样重正交设计易与c as hosting交互但依然需要出现c系的面向对象等CPP多范型里面的需求场景，所以除去lua,c这种较专用，重基础和偏门的，所以在应用上我们依然需要学习python和cpp这种多范型支持的，而python即是这种langtech level和liblevel都battery included语言。

python in onlytwo as guest for c series是种混合语言系统，业界已有混合语言的实作品，下面这些产品也有python界的比对物这里只是拿来作为例子：比如制造DSL支持领域逻辑+jit的terralang,比如compiled to lua的moonscript(它提出新语言免去了直接binding的需要)，还比如cython,zephir这种仅是生成C模块作为原语言模块的“混合语言”系统（它没有提出新语言）。

下面就让我们来打造tinycolinux上的lnpp appstack结构(linux+nginx+python+postgresql)，并安装odoo8，注意这里我们只精简安装odoo的必要模块和web相关模块。

编译lnpp的python+uswgi和postgresql

接《为tinycolinux创建应用和lnmp-源码和toolchain》文，我们这次是编译python,除了那文中gcc中需要的tinycorelinux的tcz，我们还需要openssl-1.0.0-dev.tcz(事实上python编译不要它但是接下来pip要用到它)，解压安装它，下载python src,我选择的是Python-2.7.14rc1.tgz，解压cd到src目录我们这里是/home/tc/Python-2.7.14rc1，sudo ./configure --prefix=/usr/local/python(你可以加条 --enable-threads未来用python启动uswgi多线程支持会用到),sudo make install

cd /usr/local/python/bin,下载pip，wget --no-check-certificate https://bootstrap.pypa.io/get-pip.py ,然后sudo python get-pip.py安装pip。接下来可以安装uswgi了sudo pip install uswgi（会用到与nginx编译时一样的pcre-dev.tcz），运行uswgi，显示安装后的uswgi版本是，ctl+c退出它，下面第二部分我们会谈到以正确详细的参数运行它。

对于pgsql我下载的是postgresql-10.1.tar.gz，按处理python src的方法处理它,会要求用到readline，在sudo ./configure --prefix=/usr/local/pgsql --disable-redline中禁用。sudo make install 编译完。然后在/usr/local/pgsql中创建一个data文件夹，右击权限设置为7777 组root,用户tc[1001]。这是因为pgsql默认实际上也不允许以root方式运行。

sudo -u tc /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data --encoding=UTF8创建默认系统数据库base，然后启动它sudo -u tc /usr/local/pgsql/bin/pg_ctl start -D /usr/local/pgsql/data(pg_ctl start也可是postgresql)，此时tc用户对于这个数据库的密码为空端口为5432， sudo -u tc /usr/local/pgsql/bin/psql base可连上管理,ctl+c退出管理,进入data目录。修改二个conf文件使得可本地用navcat等工具管理否则会出现server closed the connection unexpectedly postgresql错误,首先在postgresql.conf 打开listenadress="*",然后在pg_hba.conf中加一条: host all all 10.0.2.2/32 trust(10.0.2.2是tinycolinux slirp模式下的host windows地址，你也可以改成0.0.0.0)。

为什么加--encoding呢。因为不这样做稍后在安装完odoo在base中建立odoo数据库时会提示：new encoding (UTF8) is incompatible with the encoding of the template database (SQL_ASCII)

在lnpp中安装精简odoo，python模块和配置uswgi和nginx参数

我们先安装odoo再来处理python,这样运行它时可以逐个通过pip安装缺少的python模块，将odoo8释放到/usr/local/nginx/html,精简/usr/local/nginx/html/odoo/addons安装的所有模块，仅保留以下：

```
account
account_voucher
analytic
auth_crypt
auth_signup
base_action_rule
base_import
base_setup
board
bus
calendar
contacts
decimal_precision
document
edi
email_template
fetchmail
gamification
google_account
google_drive
im_chat
im_livechat
knowledge
mail
marketing
note
pad
pad_project
payment
payment_paypal
payment_transfer
procurement
product
project
report
resource
sale
sales_team
share
web
website
website_blog
website_forum
website_forum_doc
website_livechat
website_mail
website_partner
website_payment
website_report
website_sale
web_calendar
web_diagram
web_gantt
web_graph
web_kanban
web_kanban_gauge
web_kanban_sparkline
web_tests
web_view_editor
```

下面我们来联合配置启动uwsgi和python,nginx，我们还希望像lnmp一样，分别独立启动nginx,mysql和php-cgi(它就相当于python中的uwsgi)，先启动uswgi：

/usr/local/python/bin/uwsgi --socket :8000 --pythonpath /usr/local/nginx/html/odoo --wsgi-file /usr/local/nginx/html/odoo/openerp-wsgi.py

实际上它也有很多变体和缩略形式(你可以参照网上建立一个小例子代替openerp-wsgi.py中的内容来分别测试)：

--socket=:8000 --master --uid=tc --gid=root --wsgi-file /usr/local/nginx/html/odoo/openerp-wsgi.py --daemonize=/usr/local/python/bin/uwsgi.log

--socket=:8000 --chdir=/usr/local/nginx/html/odoo --wsgi-file openerp-wsgi.py (以上chdir也可用pythonpath代替，此pythonpath非python里面的应用模块寻找意义上的pythonmoudlepath)

--manage-script-name --mount /yourapplication=myapp:app

-s :8000 -w uwsgi-server:application -d somelogfile

（以上参数都可写进一个ini，然后以uswgi指定ini的方式进行，但上面我们倾向于不使用uwsgi+ini文件的方式）

可以看到上面总有静态配置的东西，要么地址要么模块名要么类名，而lnmp中的php-cgi后面的参数是不与任何静态地址挂钩的，它就是一个全局服务器将语言服务转化成cgi或uwsgi，所以我们得改动一下，这个改动叫“uswgi的动态模式”：

/usr/local/python/bin/uwsgi --socket=:8000 --master --daemonize=/usr/local/python/bin/uwsgi.log

nginx下正确配置以配合来自上面uwsgi的“动态模式”(可以看出与静态模式下配置条目的相对应性)：

```
include uwsgi_params;
uwsgi_param UWSGI_CHDIR /usr/local/nginx/html/odoo;
uwsgi_param UWSGI_MODULE uwsgi-server; (不需要.py)
uwsgi_param UWSGI_CALLABLE application;
uwsgi_pass 127.0.0.1:8000;
```

修改/usr/local/nginx/html/odoo下的swgi-openerp.py对应于下面的一些条目，(它相当于同cd目录下./openerp-server -c ./openerp-server.conf，openerp-server.conf中的内容即类似下面修改的得到的配置文件）：

```
db_host = 127.0.0.1
db_port = 5432
db_user = tc
db_password =
pg_path = /usr/local/pgsql/bin
addons_path = /usr/local/nginx/html/odoo/addons,/usr/local/pgsql/data/addons/8.0 (不设置这个，会导致 http://xxx:/web/static.... f
ull.css 404)
data_dir = /usr/local/pgsql/data
```

确定python所须模块在最后进行，注释掉uwsgi启动时的daemonize项，查看启动后的输出，并一一sudo pip install 模块名安装，其中pillow和pychart特殊处理如下：

```
.......
sudo pip install Pillow==3.4.2 (不安装这个版本会出现cant create space错误)
sudo pip install http://archive.ubuntu.com/ubuntu/pool/universe/p/python-pychart/python-pychart_1.39.orig.tar.gz
.....
```

上述lnpp全部成功启动会自动在/usr/local/pgsql/data下生成filestor,addons/8.0等目录，访问localhost，成功！！

总结起来，我们需要在tinycolinux启动时在/opt/bootlocal.sh中以如下命令分别启动nginx,uswgi和

```
/usr/local/nginx/sbin/nginx
/usr/local/python/bin/uwsgi --socket=:8000 --master --uid=tc --gid=root --daemonize=/usr/local/python/bin/uwsgi.log
sudo -u tc /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
```

好了，进入odoo怎么应用和操作又是一种境地了，odoo所有的操作中，数据都有固定的视图，一条博文和一个文件是一样的，一个产品和一个电脑是一样的，faint，我记得怎么进管理模式，忘了。

我特别喜欢python生态下的jupyter，如果说odoo的cms是一种带前端展示渲染后端控制渲染的综合应用逻辑体，且其可视化拖拉是一种visual editor and demo instant showing system，那么它其实可以结合jupyter，让jupyter直接支持这二种机制。见我的《发布engitor，一个paasone》。但其实paasone我更喜欢将其改成appstackx。一个同时带托管和编辑性质的传统简装paas(不喜欢sandstorm那种，它所处的抽象层是多余的不需要paas这样一个东西只需要appstackx)

下一篇或许是《oc上集成wordpress as cms app》和《oc上利用打造codesnippter note》etc，相比php v1版的mineportal，这是py v2版的mineportal了。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在**tinycolinux**上编译**seafile**

本文关键字：**tinycorelinux**上从**0**源码编译**seafile**，**uswgi**方式配置运行**seafile**

计算机科学和编程艺术起源于西方，在基础建设级很难发现中国人的建树，比如在C系相关的系统领域国内是没有什么作品广泛使用并让别人记住的，，但一个有趣的现象是，py域和应用域中国人异常活跃，且有不少佳品的，比如coco2dx，还比如我们要谈到的seafile，《在tinycolinux上编译odoo》一文中我们把曾odoo称为mineportalv2 - 它是groupware，vs odoo，seafile更接近personalware，其实更适宜用来打造mineportalv2，mineportalv1 oc只是一个复杂的图床加面向同步的webdav支持，而seafile有独立的fileserver，支持免文档数据库的切片文件系统，独立的standlone fileserver targetting storage domain logic implented as enginx appstack componet也有利于我们研究将其与enginx中的其它部件集成及深入《发布enginx》一文中的课题研究，且程序实现上鲜明的c+py混合编程特征和综合web+websocket的自然混合程序设计，更适合被用来作为教育目的，当然它不像OC一样仅需要虚拟空间就可以运行，只是这在云主机低价盛行的今天也不是什么大事了。因此接下来我们在tinycolinux上一步一步编译它：

编译**seafile**的五大件：

我们首先编译出GCC481和CMAKE,python+pip,nginx等，按《在tinycolinux中编译cling混合c和py在线学习系统》中说的一步一步完成，且准备gcc的autotools支持和git支持：

autogen.tcz，automake.tcz，autoconf.tcz，libtool.tcz，intltool.tcz，perl5.tcz，git.tcz，openssl-1.0.0.tcz

然后编译出五大件，我下载到的版本是:

jansson-2.10.tar.gz(一个json解析库,C项目，cmake或autotools构建)

libevhtp-1.1.6.tar.gz(一个强化libevent的http库,c项目,cmake构建)

ccnet-server-6.2.5-server.tar.gz(seafile 自己的rpc库，c和py混合项目as py lib，autotools构建)

libsearpc-3.0-latest.tar.gz(seafile rpc库，c+py混合项目as pylib,autotools构建)

seafile-6.1.1.tar.gz(seafile的,c+py混合项目as pylib,autotools构建。)

seafile-server-6.2.5-server.tar.gz(seafile负责文件传输的业务服务器,c+py混合项目as pylib,autotools构建)

seahub-6.2.5-server.tar.gz(纯py,django app，seafile的前端部分)

按依赖和先后顺后编译，使用到autotools一般都是先sudo autogen.sh，然后./configure，如果sudo autogen.sh之后产生不了makefile.in基本是libtool的问题，确认安好libtool.tcz解决它，一一./configure --prefix=/usr/local/seafile之后，基本都能完成，使用到cmake一般要shadowbuild，即sudo mkdir b到src下，然后cd b,sudo cmake .. && sudo cmake build ..，其中evhtp要sudo cmake -DEVHTP_DISABLE_SSL=ON ..，libevhtp-1.1.6.tar.gz中cmakelists.txt中取消三个test的编译需求。编译configure或link过程中的时候会调到下述tcz：

acl-dev.tcz,acl.tcz,bzip2-dev.tcz,bzip2-lib.tcz,bzip2.tcz,curl-dev.tcz,curl.tcz,expat2-dev.tcz,expat2.tcz,fuse.tcz,glib2-dev.tcz,glib2.tcz,guile-dev.tcz,guile.tcz

libarchive-dev.tcz,libarchive.tcz,libattr.tcz,libevent-dev.tcz,libevent.tcz,libffi-dev.tcz,libffi.tcz,libltdl.tcz,liblzma-dev.tcz,liblzma.tcz,libssh2-dev.tcz,libssh2.tcz,popt-dev.tcz,popt.tcz,vala.tcz

基本上，，都可以在4.x的tinycorelinux tcz repos中找到。自己整理一下对应关系，假设在第一步我们上述五个除seahub外都是安装到/usr/local/seafile的，所有成功结果会是这样：在/usr/local/bin下产生各种bin，在/usr/local/seafile/lib/产生ccnet,seafile,serpc的so,la，甚至在/usr/local/bin中也产生了seafile-admin：没有py后缀shebang为py，作为脚本使用)。这个脚本很重要，下面细说.

安装配置**seafile**并用**nginx+uwsgi**方式启动：

首先创建一个仓库（相当于odoo刚装完或重新配置时，要进入web/database/manager删减数据套件一样），seafile-admin就是用来产生这个套件的总工具，并负责调用seahub根下的manage.py来启动，下面我们用官方方法-即seafile-admin来产生套件并启动它：

在任意目录新建一个data文件夹，然后产生data/seafile-server/seahub的空文件结构，把五大件中的seahub改名替换/data/seafile-server/seahub中的seahub，四大件要么作为后端，要么sudo make install到并作为python lib，seahub中也有一部分要作为python lib，因此，export PYTHONPATH=/xxx/seafile-server/seahub/thirdpart一下，除去所有这些不可见部分，此后的seahub就相当于整个seafile website了。------- 现在，可以执行产生数据仓库(我们把它称为数据套件吧)的总脚本了，就是那个seafile-admin setup，回答所有问题后发现正确配置完成，pip install gnicore后即可访问，我们看到帮助文档中配合nginx是转发gnicore的数据，现在，我们将django的这种方式，换成nginx+uwsgi，去掉gunicore的必要。这实际属于django nginx uwsgi搭配问题。

首先，我们有如下发现：/usr/local/seafile/data/seafile-server/seahub/seahub下有一个wsgi.py和settings.py，这符合我们在《发布odoo》中用nginx+uwsgi将其启动的改造方式，也就是说，它可能天然支持纯uwsgi且seafile也保留了这种方式，那么究竟是不是呢？

进一步通过观看seafile-admin我们进一步明确了这种设想：它负责配置逻辑的产生（django app settings），且它调用的manage.py仅是一个wsgi.py的wrapper（为了seafile-admin中统一gunicore,fastcgi,etc..），所以，在seafile-admin->manage.py->wsgi.py的调用路径中，这样seafile-admin既是产生套件的工具，也用于统一启动，而原本这一切：用于seafile-admin中读取配置的部分settings.py+负责启动的部分wsgi，在无外头wrapper即seafile-admin情况下，它们是分离直接放进seahub根下的settings.py和wsgi.py中的:

现在既然有数据套件和套件配置了，所以尝试直接配置uwsgi和nginx启动这个套件下的seafile就够了，其它可按《odoo》一文中的来，成功！：

nginx配置逻辑：

```
include uwsgi_params;
uwsgi_param UWSGI_CHDIR /usr/local/seafile/data/seafile-server/seahub/seahub;
uwsgi_param UWSGI_MODULE uwsgi-server; (不需要.py)
uwsgi_param UWSGI_CALLABLE application;
uwsgi_pass 127.0.0.1:8000;
```

启动的逻辑：

```
/usr/local/seafile/sbin/nginx

/usr/local/seafile/bin/uwsgi --socket=:8000 --master --uid=tc --gid=root --daemonize=/usr/local/seafile/bin/uwsgi.log
```

好吧，自己DIY着去HIGH吧。恩恩

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在dbcolinux上安装cozy-light

本文关键字：**js**个人云存储**,cozy,node-legcay**和谐模式

在前面的《appstacks》,《apps》系列文章中，我们大力涉及到带存储支持的云程序，与语言选型放一起，我们写了py的2个(seafile,odoo)，php的2个（owncloud,mongopress），js的一个davros。并一直扩展它们的意义，认为它是一种小可视为与common storage based webapp合作(ocwp ownnote,mongpress,odoo),大可扩展为paas,webos的东西(sandstorm,cloudwall)，在《设想：cloudwall与树莓派》一文中，我们又把cloudwall与通用移动硬件的树莓派结合，提出了真正云硬件的概念。

这类OS应该仅是聚合？还是应是一个paas虚拟化结合的东西？

拿sandstorm来说

在前面《在tinycolinux上免sandstorm安装davros》时我们谈到了sandstorm和它与群晖OS等WEBOS的对比与意义:它提供了一套UI SHELL管理程序的安装,这是它的webapp聚合方面，davros就是sandstorm的file app,相当oc的file app,oc可以将一个external app加进它那个后台管理，而davros也是这样的方式被加进sandstorm的，因为它独立也能运行。

而它其实也是作为PAAS存在的它包装了一个node appstack(meteor)，却允许任何程序如php等安装入其中，它的PAAS还在于它的虚拟化，其实我之前一直很抵抗sandstorm的，它跟docker一样用到了分层文件系统这种虚拟方案，但其实sandstorm主体是没有分层文件系统的（它不管理虚拟机层的虚拟化iaas，离线的vagrant与它仅有SPK格式导入这层联系,也不属虚拟），它的grains才开始用到了一点虚拟化且仅仅是分层文件系统。(但其实仅是虚拟文件系统就能做好很好的PAAS了，比如docker,openvz）

换言之，sandstorm这样的东西才是一个全面工程，它涉及到了xaas,langsys,appstack,apps的改造和利用，即便它是一个轻量的xaas：跨xaas,langsys,appstack,apps的东西。，davros也需要sandstorm才变得合理，。但是因为sandstorm的编译涉及到一系列中心，我并不打算写文实践它。

值得一提的是，为了将这一切上提到OS和硬件层面，我们提出了dbcolinux慢慢将其打造成云OS,如将linux kernel作为共用的核心和装机中心，将/usr/local分给各种用户就可以打造openvz这样的东西。在《发布DISKBIOS》《/system,/usr分离式文件系统的linux发行版》中，让它直接管理虚拟机或实机装机,这种装机还考虑了运营对接到应用中的各种角色，后来我们的发布类文章都转到这个版本上,,我们甚至关注了对couchdb的使用甚至rapsian pi，让云OS寄托于专用可移动硬件。

好了，说了这么多，作为js personal cloud且采用cloudwall的另一个例子，现在来看我们的cozy:

cozy其实在《发布mineportalv2》中我们早提到过cozy,就像sandstorm也被我忽略一样cozy也是，不过cozy现在好像整个重构了。直到第一眼看到它的简化版：cozy-light我感到很惊艳，因为它支持的app中包括了html5 game app.

它也有paas成份,聚合的webos成份：

Cozy is a platform that brings all your web services in the same private space. With it, your web apps and your devices can share data easily, providing you with a new experience. You can install Cozy on your own hardware where no one profiles you.

most of the apps are runnable without Cozy Light

cozy也使用了pouchdb,couchdb的那种replicate协议是用来取代http的，，，默认加入同步网络的节点满足这类协议的，，，甚至都省了传统BS云同步中的同步终端，它们是满足协议即可当同步器/终端也可当同步中心。所以,cozy也有同样的功效，但是它在server端用的是leveldb。

好了，下面开始尝试在dbcolinux上安装它:

安装启动cozy-light

cozy-light好像2016年之后没人维护了，它的最新版本是0.4.9，相反它的APP在维护就够了，安装cozy-light分为安装cozy-light和各种支持APP支持,由于这二部分不是同步更新开发的，涉及到相同的东西有时会二处有不同的版本编译需求，比如pouchdb-4.0.3.tgz在app和cozy部都会被安装一次，都会用到leveldb，一个是120，一个是114,要找一个兼容这二者的js，我选择是的0.12.18带npm2.15.11，否则能编译完cozy-light是处处充满陷坑，稍后会提到为什么这么选.首先，node0.12.18安装https://nodejs.org/dist/latest-v0.12.x/,再装git,由于node 0.12.18属于老版本了，我们需要为/usr/bin/node建立一个shell wrapper开启它的和谐模式，否则会出错，把node重命名为nodejs，/usr/bin下新建以下内容文件并加起执行权限，引用nodejs：

```
#!/bin/sh
rdlkf() { [ -L "$1" ] && (local lk="$(readlink "$1")"; local d="$(dirname "$1")"; cd "$d"; local l="$(rdlkf "$lk")"; ([[ "$l"
= /* ]] && echo "$l" || echo "$d/$l")) || echo "$1"; }
DIR="$(dirname "$(rdlkf "$0")")"
exec /usr/bin/env nodejs --harmony "$@"
```

npm install cozy-light -g会自动从github下载0.4.9到/usr/lib/node_modules/cozy-light，我在香港主机装的，所以外网速度快,/cozy-light/node-modules有它引用到的submodules各个submodules有它subsubmodules，node的modules就是一个树形结构，没有ln这样的引用，同一个工程不同的部分引用相同的模块的不同版本会重复存在，这也就是如上为什么一个项目要选一个兼容node版本的另一原因。不指定 -g会安装到PWD，编译过程中

会调用node-gyp编译leveldb120,出了一些warnning:gyp WARN EACCES user "root" does not have permission to access the dev dir "/root/.node-gyp/0.12.28"，但是没关系,安装正确结束会输出一个cozy-light的模块树形表,直接启动它建立到/usr/bin/cozy-light的文件,cozy-light -p 80 start，启动失败,以下错误在设置了和谐模式后依然存在:

```
/usr/lib/node_modules/cozy-light/node_modules/pouchdb/node_modules/request/node_modules/hawk/lib/server.js:506
            host,
                ^
SyntaxError: Unexpected token ,
```

目测是request版本问题，查看其所在安装目录，发现安装的是最新的版本可能需要降级，我们用自定义位置的安装法：在具体模块树级层次中运行npm install。不依赖整体-g:打开/usr/lib/node_modules/cozy-light/node_modules/pouchdb/package.json，将"request": "^2.61.0",改为"request": "2.68.0"，为2016年1月的版本，删除pouchdb/node-modules下的request，进入/usr/lib/node_modules/cozy-light/node_modules/pouchdb/下执行npm install，再次执行cozy-light -p 80 start 成功。cozy-light再次启动会有bug，cozy-light stop后再start也不行，最好重启一下。

但是挑战不是这里，挑战和难度是安装app:

安装personal cloud distro

cozy-light install-distro personal-cloud

apps全被安装在于./root下，/root/.cozy-light levelDB的数据都在这里,这次node-gyp编译的是leveldb140，有出错，整个过程中，我先后尝试过4.x-latest,5.0-latest,6,0-latest，都有出错:nan_implementation_12_inl.h error: no matching function for call to ‘v8::Signature::New，追踪一下，依然是版本的问题:time@0.11.1'引用的nan 1.6.2,仅跟0.12适配,这也是为什么我选择0.12的原因，安装其它app或distros时，也会有其它的问题，app/distors安装跟cozy-light一样，受上面说的工程各层次级引用不同nodejs版本的原因导致出现node-gyp将库链接到不同node版本出现问题,在0.12下以上personal cloud distro全程通过。

还存在一个warning : An uncaught exception has been thrown:{ [Error: spawn ENOMEM] code: 'ENOMEM', errno: 'ENOMEM', syscall: 'spawn' },要打开swap参见我以前的《在tinycolinux xxx》文章增加swap部分

以上personal cloud distro只安装了tasky,contacts,simple-daskboard,,等几个app，安装一下files:cozy-light install cozy-labs/files,启动cozy-light后为其设置密码：cozy-light set-password,启动和进入files app时会现如下错误：

```
An error occurred while initializing notification module -- Error: connect ECONNREFUSED
[Error: No instance domain set]
Error: connect ECONNREFUSED
```

相信不难解决。自己解决。

---

files app是cozy,cozy-light共享的，所以也应该可以用同样客户端同步吧，另外cozy有一个cozy-fuse,都可值得试一下。

关注我。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在tinycolinux上安装sandstorm davros

本文关键字：**git**更新失败**tlsv1**，源码编译**nodejs**，提取**sandstorm**中的**davros**为免**sandstorm**版本

在《发布mineportalv1:ocwp》，《发布mineportalv2:odoo,seafile》中，我们不断提到“以中心存储为后端的webapp设计”，因为以存储为中心符合个人操作PC的习惯。对于服务器和运维人员也是一样，网站体APP也可以产生海量数据，对于迁移和备份是十分重要的，这种存储后端支持要么被集成在appstack中（像seafile使用专门的repo server,odoo使用postresgl），要么被app级自身提供如owncloud的图床(其实我更倾向于不使用专门的appstack组件的方式比如seafile的文件服务器，它破坏了logic server就是langserver的事实，复杂化了不应常变动的appstack单元，odoo用专门的文档数据库来做这个文件服务器倒是更顺眼一些)，也有程序从api交互级维护，我不知道buzz是不是这样的设计-忘了名了，一个社交聚合管理系统(维护一套应用协作)，更有一些程序从paas级促成这样的结果如sandstorm：

sandstorm就是一种以维护中心数据为一体的app管理程序设计，为了达到这个目的，它先提出一个paas，这就是sandstorm，它用类似docker的方式为每个应用准备一个沙盒环境，但在管理框架级---也就是sandstorm自身---它本身就是一个js应用，维护所有app产生的实例（grains）数据备份，打包，它的grains就是用来做集中化存储的，且各个grains可以共用用户验证机制这样可以进行API级的交互，还有它下面的一个app-davros:一个类oc personal cloud storage的东西，如果做起plugin来完成可以达到像oc一样的规模。

该如何看待sandstorm呢？它其实还是一种web os的东西。

继我们的《在tinycolinux安装chrome》，群晖就是这样的web os的典型 as nas os，而OC,sandstorm，也其实是类似群晖桌面的东西，它们都是用web ui做appstore和webapp管理的本质作用-类似native desktop shell，只是工作在不同的层次---而其实也差不多：oc纯php实现管理php apps,sandstorm纯js实现，但管理混合语言stack apps，群晖synology都是混合。如果说考虑与langsys绑定的关系，sandstorm和群晖一样可以管理应用混合语言架构的东西,可以装各种appstack和各种appstack下的apps，但它与oc一样也可以全是one lang webstack的:oc是php,sandstorm是js。

为了让它成为一个纯粹的类oc的只管理js web apps的项目，其实可以把sandstorm关于paas的部分截取去掉，（比如将其放在xaas层次的VM管理中做一个管理入口到sandstorm，这样sandstorm不光有apps,grains且有vms管理跨langsys,xaas,appstack,etc..），保证它的生态下全是类davros之类的js应用，共用一个类似mean的单appstack，且纯粹面向webapps

《elmlang》后，我们关注了大web的js语言，而且还将基于一体化存储后端的webapp系统的关注点从php,py移到js，那么这个davros，它就是我们的mineportal3。我们要做的，就是先提取sandstorm中的davros为免sandstorm版本先用起来因为它本身也是一个独立的单js app。分离掉与sandstorm的xaas管理层薄薄的联系就可以了。

在tinycolinux上编译安装nodejs和npm

tinycolinux上gcc481最高最能编译7.10.1 ,8.0.0和8.0.0以上会提示ArrayVector(v8::internal::StringStream::FmtElm [])相关的错误, 最新的894要求gcc494，

sandstorm自身用的是nodejs8.9.3,官方使用的davros 0.21.7 spk中使用的nodejs6.4.0，所以在这里我们使用6.4.0版本,首先装好git，然后装好py，下载nodejs640其源码,cd到其中，执行：

./configure --preifx=/usr/local/nodejs && sudo make install

cd到/usr/local/nodejs,export PATH=$PATH:/usr/local/nodejs/bin，执行nodejs发现需要libstdc++高版本，把libstdc++.so.6.0.18(这个是编译cmake时也需要的库，参见以前文章)换到/usr/lib下，接着执行npm install -g git://xxx，发现调用git时不能下载https里的git repos内容，提示SSL routines:SSL23_GET_SERVER_HELLO:tlsv1 alert protocol version Completed with errors

这是由于最近2018.2.1github不采用低级的加密方法了，git依赖cur,curl 命令行依赖 openssl 库才能使用 ssl 和 TLS。当前一般认为 TLSv1.1 及 TLSv1.2 才是安全的，很多 https 服务器仅支持这2个协议，不再支持 TLSv1.0 及 ssl。但是 openssl 是从 1.0.1 才支持 TLSv1.1 及 TLSv1.2。系统当前安装的openssl-1.00.tcz+curl不支持。查看已安装的ssl和curl，执行：curl -V(大写）发现openssl是1.0.0k,curl是7.30.0

我也不想去其它的5.x的tinycolinux中去找了，自己编译吧。好像5.x的是1.0.2的去掉了sslv2v3的？所以还是自己编译安装吧。

我下载的是openssl 1.0.1src和curl-7.15.0.tar.gz,首先安装perl5,openssl编译需要perl5,cd srcroot,./config --prefix=/usr/local shared && sudo make install就可以（注意不是./configure）一定要加/usr/local，否则安装到/usr/local/ssl中去了，加shared可免去下列错误:x86cpuid.s:(.text+0x2d0): multiple definition of `OPENSSL_cleanse' ../libcrypto.a(mem_clr.o):mem_clr.c:(.text+0x0): first defined here

接下来编译新的curl7.30.0，./configure --enable-shared --with-ssl=/usr/local

查看新的openssl版本

/sbin/ldconfig -v openssl version -a

查看curl是否引用了刚编译安装的1.0.1版本

curl -V(大写的），发现使用的是openssl1.0.1

现在git会自动使用ssl3,npm install -g git://xxx或https://可以用了。

准备davros代码并编译运行,失败

现在准备davros,我下载的是https://github.com/mnutt/davros中的davros-ca480aea708d0e9ae4b63342a4583660609f331f的0.21.7 release,将davros的根中的所有内容全选，上传到/usr/local/nodejs根目录，cd到此

我们看到js npm的包管理还是蛮好的，每一个包都维护一个package.json，申明它向前依赖的项。应用即包本身，各个包组成一个树形关联关系组成一个大应用，davros作为大应用，可以看到其根下有npm用的根package.json,bower用的根bower.json,etc..

根据https://github.com/mnutt/davros的说明，先sudo npm install，但是发现奇慢，加tb的mirror吧npm install -g cnpm --registry=https://registry.npm.taobao.org,再sudo cnpm install发现快多了（这是在安装src root下那个package.json的依赖关系包括bower）。它可以代替默认的npm，匹配不到的它会从默认从github下载。如果有紫红色的就是出错的

接下来，sudo bower install后会提示找不到bower,把生成的node_modules/bower/.bin中的那个链接文件移到/usr/local/nodejs/bin中，并修改指向对应位置

然后sudo npm install 和 sudo bower install --allow-root，发现git出错：error: SSL certificate problem: unable to get local issuer certificate while accessing

git config --global http.sslVerify false一下会在home/tc/下产生.gitconfig文件，再sudo bower install --allow-root这下能继续了。

我们也无法去追踪到底安装了多少东西了。

然后按照https://github.com/mnutt/davros的说明，sudo PORT=3009 ember serve，发现ember也没链接进/usr/local/nodejs/bin中（在src root package.json中它跟bower一样是要被安装的也一路并没有出错），直接执行吧，不做了：sudo PORT=3009 node_modules/.bin/ember server，发现ember的确在后台打开了守护，根据github的readme.md说明，这时本地桌面客户端可以连接了。

但其实我们根本不用这样做，因为这个后台守护会耗尽内存， top中会看到内存占用一直涨，最终命令行也显示heap out of memeory，尝试失败！！

按理说，这里要ember build一次，之后会将ember一系列东西，包括davros src root的app文件夹下面的东西全打包在生成的srcroot/dist下一个davros打头的随机文件名中。是不是这样呢，我们也没时间追究了，只能换个死方法了，我们直接从spk中取来所有ember build好的东西：

直接提取spk的已编译好的davros运行,成功

在另外一台机器上安装一个sandstorm，然后连上进入winscp,进那个spk的目录，我的是/opt/sandstorm/var/sandstorm/apps/e813a833d983fbc38d87da62ea461fa7/opt/app，全部打包下载，清空原来nodejs的根目录，重新/tce/nodejs460下make install一次，然后把新的spk中的包的内容全部上传到这里，./sandstorm中的launcher.sh弄出来到根，稍微修改下其中的路径，建立data,data-temp,samplefiles等目录，执行sudo ./launcher.sh(它其实就是nodejs执行根下的app.js)，注意8000端口不要被占据，成功！！不光oc的桌面客户端访问。网页端免sandstorm也可以进入。 可见它与sandstorm管理框架和ember build过程是没有太多导致运行失败上的关系的。

当然，这个免sandstorm是没有认证机制的，如果是自用的话，随便写个认证逻辑就可以，这个服务端比oc的服务端快多了。

下文《mineportal demobase总成：一个类sandstorm +sandstorm appsystem的东西》，这应该是在《tinycolinux上装chrome as 客户端》之大webstack对应于服务器的东西。服务于bcxszy教学和实用。

关注我。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

统一的分布式数据库和文件系统**mongodb**，及其用于解决**aliyun**上做站的存储成本方案

本文关键字：**mongopress**安装，**0**维护网站系统,**web**程序静态资源/媒体文件维护/备份,阿里云**mongodb**

从本地打包技术到分布式存储中的做站媒体维护

在大型本地程序的客户端发布中脱离不了文件系统/打包技术， — 这在游戏技术中很常见(看看大型3D网游DATA中的那些动则几G的资料片/资产库文件就知道了)。比如，zip包也是一种简单的文件系统。不过更复杂一些的场景，比如发布大量小文件还需要维护更新和版本化的时候，往往用打包+hash+加密技术(一个索引一个存储库)，像warcraft系列的mpq啊这些都是例子，就不细说了。

事情到了WEB和做站，同样地，WEB的静态资源往往有时变得很巨大，有图片这种小文件也有大文件存储这些需求，我们来细细分析 — 在其底下首先是分布式存储和其各种方案，从服务器上的raw filesystem，分布式文件系统。到简单的ftp，webdav(支持统一客户端同步etc..如owncloud和davros),到网盘的切片文件系统(支持秒传,如dropbox)，到对象存储技术（es2,aws,aliyun oss）,到传统数据库和它们的结合体如mysql+fuse文件(aw3fs,okasoft)，甚至像基于云计算渲染的游戏中免发布客户端资源做成stream,都可用于操作和存储网站文件。我们现在的web栈也往往都是一个httpd的page ui srv + 一个数据库srv + 具体WEB程序维护的程序数据库条目索引本地文件或分布式中的文件，可见，对存储的处理是通用web栈中一个基础环节。

然而，现在的以上这些，都涉及到不同的技术显得太碎片了，同是存储却用了不一样的多种不同质的技术，而且都不能使web做到像本地一样存储维护网站系统（除了OSS可以挂载到本地盘稍微有点接近这个之外）。这使得维护网站静态资源过程往往变得很痛苦，那么有没有一种像打包文件的本地方式一样维护/备份网站系统。统一备份网站数据/媒体，且避免像wordpress这种紧绑定媒体和数据库不好备份的缺点。更进一点，可以做到像dokuwiki那样免数据库一样，发贴即存储，打包复制带走即整个网站维护。

ps:整合的东西往往有一个干净，简单的门面。可以改造成各层开发者适用的版本，稳藏代码后面的复杂。这其实是现代开发，在源码处理（语法上）和发布上（组件技术上）的通用技术了。软件抽象和设计模式中的门面，都是这样的例子。整合的东西还可以提高应用的简便性。 而我们的工作，就是使web stacks中的存储自成一体，使之变得更整合更易维护。本站以前的发布/演示中，ocwp,cmserpone都是在一定程序上使媒体维护变得免维护化这样的例子,owcp用网盘作媒体后端，备份即同步网盘,cmserpone是导出一个公司的db。而接下来介绍的mongodb更像是一种突破。— 与oss不属数据库技术不同（它是干净的分布式存储），它是一种称为文档数据库，此文档二字即为网站资源，它具有gridfs本身就是一个文件系统,整合了mysql和fs。同时可用作mysql存储程序数据，你可以将它视为mysql+oss,或native storage backend for web apps(that shipped with webstack as ehanced lamp)，是web栈中一个很合理的整合存在。

mongodb与在阿里云中省事/低成本/免坑做站

阿里云是这样设计ECS的，在其产品计划中它把ECS作为计算产品出售，并不是一个带存储/计算的全能iaas产品(如果你要这样做，技术上是可以的，但是成本很高，ecs在安全/负载/流量方面没有优势不利直接建站，而其存储类产品有这方面的设施和价格优惠)，因此云存储中，有oss,ecs文件系统，块这样各层的东西，包括rds,mongodo这样的存储产品。比如，在iaas层，阿里云希望用户搭配使用ecs+rds。

甚至有负载ecs,更甚至，根据这些，阿里向用户提出游戏解决方案电商解决方案，搭配使用的方案。

but,这太乱了。而且其存储成本组成，不但包括存储本身还包括存储中的媒体产生的流量。

我希望有一种方案。像ECS一样全包但是不要涉及太多产品和计费方案。只涉及到最多二产品，且避免ecs的以上做站的短处，再来搭配ecs。aliyun Apsara for mongodb+ecs就是这样一种，内网中的这二个产品流量是免费的，mongodb中的出流量是免费的，ecs按量收费仅放web程序应该不会产生太多流量，这样流量带宽方面综合起来比较划算。ecs+rds for mysql+oss也可以，但是这处方案至少涉及到三个产品。

在mongodb这种方案中，你仅需要租用mongodo即可，外带一个支持程序的空间（比如阿里云的免费PHP空间），在程序支持方面，如果你是个人博客，可以选用比如mongopress,或成熟的nodejs+mongodb的mean stack系列下的app（不过这样你至少需要一个ecs，因为aliyun ews中的js下架了）。

安装方法：下载mongopress 0.2.3放到apps/wamdata/source下，默认密码admin/admin安装,然后在安装成功的后台点导入数据，下载plupload 1.5.1，解压2个文件plupload.js，plupload-html5.js覆盖mp-admin/js/plupload/中的2文件即可。

下面是我在msyscuione+ocwp使用的wam/wnm架构中测试的结果：

———————

当然它aliyun Apsara for mongodb也不是不贵，合租mongodb才划算，，体外话：对于降低个人网站最低成本和同类产品的自然整合考虑，我最初的想法是mailbox当网站空间的附件呢并把它发展为像owncloud这样的东西，不过还是偏向到web这块来了，那个想法有点疯狂有点像web1.0时代的google openwave设想。不过openwave现在也有成功的替代品了。叫什么来着我忘了。。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

捉虫与寻龙：从0打造wordpress插件wp2oc fileshare (1) – 将wp存储后端做进owncloud

关键字：**wp2oc fileshare**，**wordpress**媒体存进网盘,网盘作为**wordpress**图床,**owncloud wordpress backend storage**

其实用网盘做wordpress网站的图床一直是一个很流行的想法，业界存在oss,七牛网盘，百度盘wp2pcs,wp2pcs_sy方案，不过oss,七牛，百度盘pcs这三者始终是面向外接第三方服务，这些都不能得到服务保障，其中免费且最好用的百度盘pcs api需要申请权限，实用性大打折扣。

这里我们选择的用owncloud作为wordpress的存储后端，这二者生态相似，完成后的插件可以，1，基本（不能完全）代替wordpress原生图片媒体管理功能，2，网盘图床的操作/备份符合在文件夹操作文件习惯，且可以网盘特有的同步方式进行备份和打包，3，当媒体文件很大时，转移wordpress整个媒体也就是改一条外链。不再需要涉及到数据库备份。4，当然还有更多。。

1，确立需求：我们仅需要开发一个APP

我们需要的仅仅是将owncloud存储服务做进wordpress，owncloud有自己的rest api,可以将其服务以wordpress插件的方式做进wordpress形成其后端图床。

我们找到的是ocs filessharing api,为什么必须是fileshare而不是file呢，因为做图床的网盘必须是可以外链的。主要用到的是其get all share部分，所需的参数形式是http://www.xxx.com/ocs/path?=/dir&subfiles=true，首先对于使用到的参数部分我已经在后台加了设置接口了，主要就是四个：

接下来就是开发和调试了

PS：开发是一步一步确立调试的过程，如果说编码确定技术点然后一个一个攻克是寻龙过程，因为龙比较大还是比较容易发现的，而调试则是一个捉虫的过程，常指代开发过程中，这二者所花的时间和过程往往在开发软件和APP（APP指一些小软件只有几个）穿插。尤其是调试部分，需要频繁进行，一个负责的开发实践往往要体现这二者。 在下面的各个技点难点中，我们会同时谈到技术点和调试手段，即龙和虫：

2，技术难点：wordpress plugin开发

1，往wordpress媒体上传框新加选项卡,以下参阅了否子戈的部分代码。

```
// 在新媒体管理界面添加一个百度网盘的选项
function wp_storage_to_pcs_media_tab($tabs){
// if(!is_wp_to_pcs_active())return;
$newtab = array('tab_slug' => 'From Owncloud Fileshare');
return array_merge($tabs,$newtab);
}
add_filter('media_upload_tabs', 'wp_storage_to_pcs_media_tab');
// 这个地方需要增加一个中间介wp_iframe，这样就可以使用wordpress的脚本和样式
function media_upload_file_from_pcs_iframe(){
wp_iframe('wp_storage_to_pcs_media_tab_box');
}
add_action('media_upload_tab_slug','media_upload_file_from_pcs_iframe');
?>
```

2，改造owncloud files_sharing app，使之显示链接文件而不是外链共享文件。这是因为原文件中得到的结果是返回所有的共享而不是指定root share dir下的所有文件，而后者才是我们需要的，我使用的是8.0.16的相关文件，简单修改如下：

```
private static function getSharesFromFolder($params) {
$path = $params['path'];
$view = new \OC\Files\View('/'.\OCP\User::getUser().'/files');
if(!$view->is_dir($path)) {
return new \OC_OCS_Result(null, 400, "not a directory");
}
$content = $view->getDirectoryContent($path);
$result = array();
foreach ($content as $file) {
$result = array_merge($result, array('1'=>$file['name']) );
}
return new \OC_OCS_Result($result);
}
```

3，调试明确rest api一次request/response过程中的数据主要是什么形式的：

好像bookmark用的rest api是第一代，用的是json,而ocs api用的是owncloud api，那为什么二套可以共存呢，这是因为开源软件都是慢慢发展起来的，历史遗留中好的部分会存在很久。

注意，这里会出现不确定的复杂情况比如无限要求密码，此时记得要清空浏览器所有缓存重新粘贴完整url，调试一次就要清空一次才能保障调试结果顺利进行。

4，让owncloud ocs rest api免密码，这是因为上面的调视是可视化进行的，而owncloud ocs api是需要程序内编码验证的，而这些不能浏览器端以传递给URL的方式进行，只能通过CURL http basic auth方式进行，能传给URL的是以上几个提到的配置参数。

```
function wp_storage_to_pcs_media_list_files(){
$ch = curl_init(“get_option(‘oc2wpfs_oc_server’)”./ocs/v1.php/apps/files_sharing/api/v1/shares?path=get_option(‘oc2wpfs_oc_dir’).”&subfiles=true”);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($ch, CURLOPT_HTTPAUTH, CURLAUTH_BASIC);
curl_setopt($ch, CURLOPT_USERPWD, “get_option(‘oc2wpfs_oc_user’):get_option(‘oc2wpfs_oc_password’)”);
$output = curl_exec($ch);
curl_close($ch);
echo $output;
……
}
```

得出以下基本调试视图：

好了，接下来就是把获到的API response解析为上传媒体上的文件，让editor支持选择媒体等部分了。这样插件就基本完成了，留到以后做。。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

让**owncloud**成为微博式记事本

本文关键字：利用**owncloud**做网上记事本

相信不少人有把QQ空间之类的社交类，微博类程序当网上记事本用的经历，谈到记事本程序，有微博这种（加密或不公开），有mail based note俗称电子邮件便笈，也有利用emlog之类的自建碎语的，当然wordpress+插件或p2p wordpress theme也可以，还有人用wordpress页面+评论打造，当然，直接txt文档或excel表格也可以，单个mediawiki页面也可以(它支持页面内多条记事的归档，即一个页面一个归档)，所以说，网上记事这种需求是很普遍的，尤其是文字工作者（程序员，文学爱好者等等）。------- 程序员的能力几乎都在于速学和实践，大多的细节易忘，一段代码，一个DEBUG必知，所有这些记事在这个环节上必不可少。

当然，现在流行的方案是使用专门的网上的记事程序像有道笔记，印象笔记evernote etc..，还有leafnote，与作者一直说的web程序portal化和面向存储后端集中化比较接近，它们往往有完善的移动端支持，不过它们托管在别人主机上，备份和使用上都往往不尽人意。

加密的微博帐号往往最符合发便条的使用习惯，打开就是一个框，填入就可发布，反而现在的一些大而全的记事程序做得过于完善，比如不需要支持html内容，好了，适合自己用的就是最好的，我个人觉得，记事不需要像写博一样配备一个wywiws editor。

在前面《利用owncloud打造static web hosting空间》一文中，我们使用ownnote打造了一个静态网站托管空间，我们还说到，其实这种掩藏于portal之下的动态程序+呈现在www的静态展示，其实可以构成任何owncloud based cms程序，甚至强大的通用网站程序，这种网站程序的特点是：中心化存储，有OC storage backend支持，便于维护，打造和迁移。而在这篇文章中我们同样利用的是ownnote+OC的方案，且可以拥有前者所有的好处。

第一步是，在OC上打造多个ownnote app，我使用的是oc9+ownnote app 1.08。

在oc上安装二个**ownnote app**

由于oc上ownnote不可以直接改文件夹名来安装同一个APP产生不同的APP实例，因此我们需要修改ownnote的源码来使OC支持安装二个OWNNOTE，多个ownnote的存在，对于使用它来构建记事程序才是有意义的，否则，oc+ownnote只能使OC成为数量上仅限一个的记事程序，也限制了在这个OC上打造static web hosting的能力。

其实，总体的修改逻辑，就是把源码中ownnote字眼改成ownnote2，先上传一份同样的ownnote2与原ownnote并列，然后进行修改，下面只讲重点部分：

1)core部分，这部分必改，使得APP不与原ownnote混淆(程序逻辑，页面和显示正确)：

appinfo目录下app.php:

```
\OCP\App::registerAdmin('ownnote2', 'admin');

'id' => 'ownnote2',

'name' => \OCP\Util::getL10N('ownnote2')->t('Notes2') //这个使APP上显示notes2，如果是作staticwebhosting用，可改成posts，以与notes区别
```

appinfo目录下application.php:

```
namespace OCA\OwnNote2\AppInfo;

.....

use \OCA\OwnNote2\Controller\PageController;

use \OCA\OwnNote2\Controller\OwnnoteApiController;

use \OCA\OwnNote2\Controller\OwnnoteAjaxController;

(像以上关于命名空间的，在其它原码文件中也一律改为ownnote2，将从2)开始不再列出)

controller目录下pagecontroller.php: $response = new TemplateResponse('ownnote2', 'main', $params);
```

app根目录下admin.php:$tmpl = new OCP\Template('ownnote2', 'admin');

template下main.php：

```
\OCP\Util::addScript('ownnote2', 'script');

\OCP\Util::addScript('ownnote2','tinymce/tinymce.min');

\OCP\Util::addStyle('ownnote2', 'style');
```

```
$disableAnnouncement = \OCP\Config::getAppValue('ownnote2', 'disableAnnouncement', ''); \\这句应该移到3)数据库条目部分

$l = OCP\Util::getL10N('ownnote2');
```

template目录下admin.php

```
\OCP\Util::addScript('ownnote2', 'admin');

.....

$l = OCP\Util::getL10N('ownnote2'); //这句显示管理界面左边显示ownnote2
```

2)URL部分，这部分使得打开app时，是正确的http://oc/index.php/app/ownnote2：

appinfo目录下app.php : 'href' => \OCP\Util::linkToRoute('ownnote2.page.index'),

appinfo目录下application.php : parent::__construct('ownnote2', $urlParams);

js目录下admin.js : var newurl = OC.linkTo("ownnote2",url).replace("apps/ownnote2","index.php/apps/ownnote2");

js目录下scripts.js : var newurl = OC.generateUrl("/apps/ownnote2/") + url;

3)使ownnote2能正确读写DB条目:

appinfo目录下database.xml中:

```
<name>ownnote2</name> //这个出现在owncloud appconfig中

<name>*dbprefix*ownnote2</name>

<name>*dbprefix*ownnote2_parts</name>
```

appinfo目录下routes.php : array('name' => 'ownnote_ajax#ajaxsetval2', 'url' => '/ajax/v0.2/ajaxsetval2', 'verb' => 'POST')

js目录下admin.js : 所有$.post(ocOwnnoteUrl("ajax/v0.2/ajaxsetval2"), { field: 'folder', value: val }, function (data)

controller目录下ownnoteajaxcontroller.php:

```
public function ajaxsetval2($field, $value)

所有$FOLDER = \OCP\Config::getAppValue('ownnote2', 'folder', '');

controller目录下ownnoteapicontroller.php:所有$FOLDER = \OCP\Config::getAppValue('ownnote2', 'folder', '');
```

app根目录下admin.php:

```
OCP\Config::getAppValue('ownnote2', 'folder', ''));

OCP\Config::getAppValue('ownnote2', 'disableAnnouncement', ''));
```

lib/backend.php中所有关于db操作的逻辑也改成针对ownnote2。

可能上面的修改部分没有涉及到全部，但是你可以慢慢调试得出正确的ownnote2，这个尝试是必定存在的，请耐心完成。

比如如果发现产生不了数据库中的db条目，请删除原有的ownnote2条目重新反注册/注册app。

一切完成之后，使用方法为：如果同时存在ownnote和上面弄出来的ownnote2，在后台设置database and folder时的folder时，要禁掉ownnote，先修改ownnote2的，再启用ownnote修改ownnote的path.然后其它就都不用管了。可以像正常ownnote app一样使用app2了。

而且，我们还需要定制它，使得ownnote更好用，一句话，使之更像个人化的微博。

微博式记事定制

首先，我们使得那个新建note的new始终处于被点击状态，出现的name文本框会当成我们上面所说的微博记事的文本框。在js/scripts.js的function bindListing()未加一条：

$("#new").click();

这样一打开ownnote app，name框始终是待填入的。

因为这个文本框是设想用来直接填入记事内容的，所以未来往这个框中填入的东西也许是一些用户从网上复制来的东西，长度不定，内容有特殊字符，所以我们需要定制下save to folder时生成的文件名逻辑，使得填入的内容被置入记事内容体，而生成的文件名才被放置到这里，且经过了便于save to folder的过滤处理逻辑。

js去除特殊字符的函数是：

```
function stripscript(s)

{

var pattern = new RegExp("[`~!@#$^&*()=|{}':;',\\[\\].<>/?~！@#￥……&*（）——|{}【】‘；：\"”“'。，、？]")

var rs = "";

for (var i = 0; i < s.length; i++) {

rs = rs+s.substr(i, 1).replace(pattern, '');

}

return rs;

}
```

而去除空格的JS逻辑为这样：somestr.replace(/\s+/g,"")，故组合一下这二者，很容易得到过滤输入框内容生成文件名的逻辑：stripscript(somestr.substr(0, 50)).replace(/\s+/g,"");

下面我们将这条逻辑放在程序中的适当位置（主要是程序中产生note的地方）：

```
scripts.js中的createnote()中：

var name1 = $('#newfilename').val();

var name = stripscript(name1.substr(0, 50)).replace(/\s+/g,"");
```

文件名有了，我们还要把内容name1一同放进产生note的逻辑中传送到note内容区：

scripts.js中：

```
$.post(ocUrl("ajax/v0.2/ownnote/ajaxcreate"), { name: name, name1:name1, group: group }, function (data) {

loadListing();

});

controller/ownnoteajaxcontroller.php中：

public function ajaxcreate($name,$name1, $group) {

$FOLDER = \OCP\Config::getAppValue('ownnote2', 'folder', '');

if (isset($name) && isset($group))

return $this->backend->createNote($FOLDER, $name,$name1, $group);

}
```

lib/backend.php中：

```
public function createNote($FOLDER, $in_name,$in_name1, $in_group) {

$name = str_replace("\\", "-", str_replace("/", "-", $in_name));

$name1 = $in_name1;

.....

\OC\Files\Filesystem::touch($tmpfile);

\OC\Files\Filesystem::file_put_contents($tmpfile, $name1);

......

$query->execute(Array($uid,$name,$group,$mtime,$name1,''));
```

为了防止用户在编辑note后保存时破坏上述文件名过滤规则，实际上我们还需要在保存note的地方运用以上过滤逻辑：

script.js中的savenote()中：

```
$('#editfilename').val($('#editfilename').val().replace(/\\/g, '-').replace(/\//g, '-'));

// var editfilename = $('#editfilename').val();  //注释这条

.......

var content = tinymce.activeEditor.getContent({'format':'text'}); //使得后台那个可视编辑器只存取txt

var editfilename = stripscript(content.substr(0, 50)).replace(/\s+/g,""); //注释的那条改变一下逻辑放这里
```

这样就算完成了。新的ownnote清奇好用,恩恩！！

当然你也可以修改CSS加入name文本框加大的样式逻辑。这样编辑起来更方便。

与leafnote之类的相比，有OC作存储。个人觉得更方便，不过leafnote更强大，使用的mongodb也直接支持数据库引擎级的存储，不过OC有自己的特点就是它本身强调集中化APP数据的WEB APP架构设计，这个我更喜欢。

你其实还可以改htm存储为txt格式，这样新建txt，只要重命名txt（作为记事内容）丢到客户端同步的文件夹就能记事了。

官网andriod客户端支持的发贴逻辑需要定制才能。而且它只针对http://oc/index.php/apps/ownnote/，所以对于一个oc同时有二个ownnote的情况下，第一个要预留成记事而不是static web hosting。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

让owncloud hosting static web site

本文关键字：在**owncloud**存储中做站，**owncloud static website hosting, hosting website in owncloud,owncloud www service,mailinabox static website hosting**强化**,netdisk netstorage based blog system**，**netdisk based static website hosting and syncing**

在发布《mineportal》时我们谈到不使用静态网站的理由：它们不具有动态性。不能交互。不可编程扩展。它们只是一个从线下生成然后将结果同步/发布到线上的工具。所以静态网站，并不能真正代表现代意义上的“通用网站程序需求”，它们仅能负责展示，其它交互上的功能或扩展，需要靠那些传统网站意义上的东西外挂进来。

所以那篇文章中mineportal方案1中我们用到了oc+wp。因为我们终究需要的是“网站程序”，可是话说回来，虽然static web site不具备动态性，可是如果一个网站需要交互和动态场景的地方，比如评论，后台编辑，网页生成等处，如果这些都可以被动态程序解决，那么这时这些动态程序（后端如oc）+前端静态网站展示——由它们组成的整体其实也满足可交互被扩展，也可以被称为“网站程序”成为我们的典型需求。而它的好处：网站可以打包打走，天然静态化是显然易见的，在以上提到的二篇文章中都不断被提到过。

在《发布mineportal2》中我们谈到可在mailinabox的oc中hosting static website，甚至谈到利用比如ownnote就可以作为编辑工具。那么不防更进一步，我们下面我们来谈在maininabox中一步一步实现它：

记得事先在oc中开启ownnote(此ownnote非qownnote，也不是qownnote对应的官方原notes app)，并做好“同时保存到sql和文件夹”设置，并做好与nginx的指向配置。

第一步：设置nginx

我们首先要解决的问题是，owncloud ownnote默认生成htm，我们只能以http://xxx.com/*.htm的方式访问静态网站，以下设置能让nginx像wordpress一像把.htm的静态地址重写为/post/的形式：

```
add_header Content-Type: ‘text/html; charset=utf-8’;
location / { try_files $uri $uri/ @htmext; }
location ~ \.htm$ { try_files $uri =404; }
location @htmext { rewrite ^(.*)/$ $1.htm last; }
```

上面的add_header content-type不能省，因为oc ownnote默认保存的htm是gb2313，即使在网页中添加meta charset=gb2313也不能让浏览器认中文会显示乱码。

第二步:为owncloud ownnote建立static website template支持

第二个问题：ownnote保存贴子内容直接为htm，是没有模板化效果的。我们需要集成外来模板。

我们选择了jekyll的simple jekyll主题。我们通过修改/usr/local/lib/owncloud/apps/ownnote/lib中backend.php来进行，先找到savenote()函数:

以下http://xxx.com是你nginx指向的root目录里面存有jekyll-simple的css，这里先定义了一个pre:

```
$precontentpart='<html><head><link rel=”stylesheet” href=”http://xxx.com/jekyll-simple/main.css”></head><body>
<div class=”page-content”>
<div class=”container”>
<div class=”three columns”>
<header class=”site-header”><h2 class=”logo”><a href=”/”>site title</a></h2><div class=”nav”><div class=”site-nav”><nav><ul cl
ass=”page-link”><li><a href=””>Home</a></li><li><a href=”/archive”>Posts</a></li><li><a href=”/about”>About</a></li><li><a hre
f=”/feed.xml”>RSS</a></li></ul></nav></div></div></header>
</div><!– end three columns –>
<div class=”nine columns” style=”z-index:100;”>
<div class=”wrapper”>
<article class=”post” itemscope=”” itemtype=”http://schema.org/BlogPosting”>
<header class=”artilce_header”><h1 class=”artilce_title” itemprop=”name headline”>post title</h1><p class=”artilce_meta”><time
 datetime=”2016-07-03T00:00:00+00:00" itemprop=”datePublished”>Jul 3, 2016</time></p></header>
<div class=”article-content” itemprop=”articleBody”><!–内容开始–>’;
```

再定义post(after)部分：

```
$postcontentpart='<!–内容结束–></div>
<footer class=”article-footer”><section class=”share”><a class=”share-link” href=”” onclick=”window.open(this.href, &#39;twitt
er-share&#39;, &#39;width=550,height=235&#39;);return false;”>Twitter</a><a class=”share-link” href=”” onclick=”window.open(th
is.href, &#39;facebook-share&#39;,&#39;width=580,height=296&#39;);return false;”>Facebook</a><a class=”share-link” href=”” onc
lick=”window.open(this.href, &#39;google-plus-share&#39;, &#39;width=490,height=530&#39;); return false;”>Google+</a></section
><hr><section class=”author”><div class=”authorimage box” style=”background: url(/jekyll-simple/assets/img/Taffy.jpg)”></div><
div class=”authorinfo box”><p>Author | David Lin</p><p class=”bio”>Currently a Ph.D. student in Singapore University of Techno
logy and Design in the area of Human-Computer Interaction(HCI).</p></div></section></footer>
```

```
</article><!– end article –>
</div><!– end wrapper –>
</div><!– end nine columns –>
</div><!– end container –>
<footer class=”site-footer”><div class=”container”><div class=”footer left column one-half”><section class=”small-font”>Theme
<a href=”https://github.com/wild-flame/jekyll-simple”> Simple </a> by <a href=”http://wildflame.me/”>wildflame</a>? 2016 Power
ed by <a href=”https://github.com/jekyll/jekyll”>jekyll</a></section></div><div class=”footer right column one-half”><section
class=”small-font”><a href=”https://github.com/wild-flame”><span class=”icon icon–github”></span></a><a href=”https://twitter.
com/Taffyer”><span class=”icon icon–twitter”></span></a></section></div></div></footer>
</div><!– end page-content –>
</body></html>’;
```

一起写入，上面的内容开始内容结束中的内容是指的ownnote借助编辑器中插入到生成最终htm的文章内容：

\OC\Files\Filesystem::file_put_contents($tmpfile, $precontentpart.$content.$postcontentpart); 这样新写的ownnote贴子会自动插入这些，具备模板化后的css效果。其它主题可类推定制。以上基本是一个文章页的通用html前端技术之essential in a nut了。

最后。虽然我们在这里的方案有很多hacking的痕迹，可是如果考虑进一步把上面所有这些做成oc的插件，还比如更进一步完善ownnote编辑器，使之输出直接支持模板化的效果。它就会成为高可用的产品。至于图片附件什么的，你可以参照站内文章中的《将owncloud作为wordpress的图床》之类。

网上还有一些利用oc做静态网站的方案，不过它们基本上只是纯粹把oc当同步工具用。本质上借助的还是jekyll,jade,gitpage这样的方案，只不过存储由github repo换成了owncloud空间而已。并没有使用到本文提到的ownnote等后台编辑增强方案。后者更自然，更强大。

————

接下来我们会为mailinabox建立一个利用colinux封装的mailinabox box，称为mineportal2 box，其实不得不说把一切封装在黑盒中是好的方案，运维的一个基本素质就是不要去动黑盒中的东西，这对用户是好的而运维人员也可以避免少动黑盒内部对系统产生问题。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在tinycolinux上编译jupyter和rootcling组建混合cpp,python学习环境

本文关键字：升级/枚举**tinycorelinux**上的**gcc**，在**tinycorelinux**上安装**python jupyter**

在前面《tinycolinux上编译odoo》中我们谈到python在流行的“one host one guest”学习语言选型组合中是对应于cpp的，还谈到一些混合语言工具，如terralang,rootcling等，见《发布qtcling》和《发布terracling》，技术界二二相对的事物总有惊人的对应：cpp,py组合的cling就相当于lua,c组合的terralang：

事实上该如何评价cling和c++,py的关系呢：要把rootcling当工具而不是语言。它是搭建一个混合C++和PY的语言系统的REPL环境和学习平台的极好工具，但是我们要实际拿来用中心依然是分开了的，独立的二门语言，即C++和PY --- 毕竟C++历史上不是以REPL方式拿来用的，terralang之于lua+c也是一样的道理。

在更早一些的文章中我们提到和发布过《发布engitor》，jupyter只不过是IDE B/S化了，想象那个python idle ide，jupyter pythonkernel notebook本身就是这个IDE的在线化和极大化(它支持更多语言和可渲染HTML等)。只不过，在那里我们还以技术狂想的形式设想了它其它方面的用途：它还可能与服务器运行设施结合，给设计人员或开发人员提供在线支持开发的可能for both maintainer and developer（传统上都是线上运营线下开发），更进一步，它还可以发展成visual builder技术，以实现applevel同时在线运行和可视化编辑的平台，我们称它为appstackx。

可视化的基础概念是以拖拉方式就能使其在一起工作的可复用件，以前是lib reuse，组件就是一些二进制接口透露出来的服务就能成为可复用件的东西，是demo as reuseable software componets当然，脚本语言的组件天然是源码形式的。无论如何，这距我们的理想：tool as framework but not engine又进了一步：它使得中心可复用件的engine变得谈化，用随手能找到的工具来代替，由于工具不准备作复用件进入架构层，所以就谈化了架构的存在降低了学习成本使得软件开发真正意义上变成了组装测试----要知道，为庞大复杂的软件系统划模块定接口是一件多么可怕的事，而一个新手随便找到能工作起来的东西搭个系统可以给他多大的自信和帮助（以后深入学习组件内部）。这叫入阶平滑无欺。

下面，我们在tinycolinux上一步一步建立起这个REPL环境和其jupyter支持（root cling源码中有支持将这个c++ repl kernel为jupyter使用的模块clingkernel和kernel.json文件），这就需要同时在tinycolinux源码编译出rootcling,python等，又涉及到编译最新的cmake，所以不妨看下《在tinycolinux上创建应用》的开头我们为一个全新平台准备gcc toolchain支持的描述 --- 我们这里只升级GCC和GCC里面带的LIBSTDCXX，而会不是GLIBSTDC。

在tinycolinux上编译gcc 4.8.1和cmake

首先，cling会用到新的支持C++11的GCC来编译且会引用到GCC的头文件来运行，所以我们使用在前文一直使用的gcc4.6.1来bootstrap到4.8.1，下载4.8.1的源码http://ftp.tsukuba.wide.ad.jp/software/gcc/releases/gcc-4.8.1/gcc-4.8.1.tar.bz2,在/home/tc下解压它，cd gcc-4.8.1 && ./contrib/download_prerequisites 会把编译用到的库下载解压好，我们想直接覆盖原来的GCC461安装，所以直接 cd .. && mkdir gcc481build && ./configure --enable-checking=release --enable-languages=c,c++ --disable-multilib && sudo make install，由于不带prefix，它配置出来的configure和make之后的结果会默认安装并覆盖GCC461，也会升级libstdcxx.so，这样就完成了我们的目的：在本系统上枚举一个新的高版本的gcc,gcc -v 发现输出4.8.1。

由于编译GCC，PYTHON，和接下来的CLING，可能会产生大量中间文件，所以tinycolinuxhd镜像放大为4G，将新GCC产生的/usr/lib/libstdc++.so改动链接指到/usr/local/libstdc++.so.6.0.18，而非随系统自带的libstdc++.so.6.0.13，否则接下来的CMAKE在./configure过程中会提示找不到c++stdlib 4.3.15，而且，4.x的curl.tcz，expat2.tcz，git.tcz，libssh2.tcz,libssl-0.9.8.tcz,openssl-1.0.0.tcz,sqlite3-dev.tcz全部下好按以前安装tcz的方法安装好，未来都有用。

现在安装升级cmake(在lnmp src中有一个旧版本2.x的cmake)，以前的cling和llvm都是用标准./configure的现在都改用CMAKE了，依然配置安装到默认/usr/目录,我下载的源码是cmake-3.10.1.tar.gz,在/home/tc下解压./configure && sudo make install,cmake -v发现是3.10。

安装在前文《编译odoo》中的python，由于jupyter会用到sqlite3模块，所以安装完sqlite3-dev.tcz重新源码跑一次并安装，（最好重启一次）python的./configure会自动发现sqlite3开发库会生成_sqlite3.pyd之类的支持。

这三大件准备好了就差不多了。

在tinycolinux上编译root cling和配置jupyter支持

跟下载gcc481源码一样，用GIT工具（上面提到要安装tcz）以以下过程分别检出llvm,clang,cling的源码（编译llvm会统一编译clang,cling），我检出是20180115左右前后的版本，为了控制tinycolinuxhd大小，检出后删除根下.git和tools/clang,tools/cling下的.git：

```
git clone http://root.cern.ch/git/llvm.git src

cd src
git checkout cling-patches

cd tools
git clone http://root.cern.ch/git/cling.git
git clone http://root.cern.ch/git/clang.git
```

```
cd clang
git checkout cling-patches

cd ../..
```

建立与src并列的clingbuild，执行以下CMAKE配置过程：

cmake -DCMAKE_BUILD_TYPE=MinSizeRel -DCMAKE_INSTALL_PREFIX=/usr/local/cling -DPYTHON_EXECUTABLE=/usr/local/python/bin/python2.7 -DLLVM_TARGETS_TO_BUILD="XCore;X86" -DLLVM_BUILD_LLVM_DYLIB=true -DLLVM_LINK_LLVM_DYLIB=true -DLLVM_BUILD_TOOLS=false -DLLVM_BUILD_EXAMPLES=false -DLLVM_BUILD_TESTS=false -DLLVM_BUILD_DOCS=false ../src

以上cmake配置过程会显示cling未来会引用GCC481的哪些路径下的头文件，如果找不到就直接调用GCC动态调试路径。

编译并安装cmake --build . ，编译完整个cling会占用大约2G不到，sudo cmake --build . --target install安装，安装也才300多M。

测试一下/usr/local/cling/bin/cling发现是5.0.0版本，现在来开启它源码自带的jupyter支持。首先在python中开启juypter notebook：

sudo /usr/local/python/bin/pip install jupyter，安装完后运行：/usr/local/python/bin/jupyter notebook --ip=0.0.0.0,(有时默认只绑定127.0.0.1)，可以看到python2.7的kernel已在ip:8888下完全正确运行了。

当然，如果嫌老是打全路径太麻烦，可以export PATH=(注意等号左右无空格)$PATH:/usr/local/python/bin。

现在，将cling源码下附带的jupyter支持开启，到/usr/local/cling/lib/jupyter/下，会发现有setup.py和几个文件夹里有kernel.json文件。

首先为python安装clingkernel支持，就是setup.py能做到的：sudo /usr/local/python/bin/python /usr/local/cling/lib/jupyter/setup.py

然后将某一个文件夹里的对应的 kernel.json注入到jupyter，让它知道：sudo /usr/local/python/bin/jupyter kernelspec install /usr/local/cling/lib/jupyter/某kernel.json所在文件夹。

完工，重新开启jupyter notebook会发现可用的c++ repl !!

始终要记得，这是一个混合了python和C++的repl学习环境和工具，缺一不可成就cpp,py这对one host one guest好CP。下面就介绍在tinycolinux上安装terralang吧。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在群晖docker上装elmlang可视调试编码器ellie

本文关键字：**elmlang live editor,docker**要注意的地方。

在前面发布《elmlang时》我们谈到elmlang的函数FRP和可视调试特征，使得为其装配一个live ide变得可能，elmlang提供的插件，已经使其它能很轻松地接入市面上几大IDE，如本地我们有atom，vscode这样的东西，在业界是推崇用vim的，他命令区和编辑区合一的ui方案使之成为通用ide，那么在远程呢，越来越流行的还有很多web IDE，elmlang for webapp的特性使得其天然就与web ide相生相融，与我的想法颇为迎合的是，elmlang的官方发布了一个ellie:el-li-e，elmlang live editor的意思，它模拟了atom这样的本地编辑器方案，该项目托管在https://github.com/ellie-app/ellie。

下面介绍如何将其安装到docker下。其实上述github repo已有docker支持了，且同时提供了for development和for production的二套方案，然而我测试时发现这二套直接利用生成的image和是存在很多问题的，container最终也运行不起来，所以我自己测试修正了一套。

我选用的测试环境是群晖下vmm出来的纯净ubuntu-16.04.5，安装好docker-ce和docker-compose后。git代码（我git pull到的是2018.8.22左右的cd242bea9114bf4b835cefeb228c77233a88ac07）。

> 基本上ellie源码就是混合erlang->elixir，nodejs->elmlang，haskell-elmlang五种语言组建出来的： elixir与nodejs都是语言，分别执行exs与js，其应用以语言库的源码形式发布。elixir又作为erlang的一个库与可执行服务正如elmlang是nodejs的一个库与可执行服务一样，erlang也是源码形式发布的，所以erlang->elixir是语言源码套源码形式发布的。 可nodejs->elmlang不一样，虽然elmlang本身以haskell开发，但是elmlang是以haskell compiled binary形式整合在nodejs生态中的，所以ellie中，nodejs是源码套二进制语言。可elmlang本身的库与应用又是源码发布的。 所以整个ellie源码的语言套语言架构中，源码形式逻辑发布的共有nodejs和elixir和elmlang，其中elmlang负责自身的执行，整个ellie app层次，nodejs源码是后端，负责elmlang代码的执行结果反馈(webpack框架)，而elixir负责的是前端（phoenix框架），负责你打开ellie时的那个界面，总之很绕。。。 所以它们被做进ellie这个docker编排逻辑中时，需要安排好几种语言的运行时和库支持 -- 在development版本的docker中可以看到清楚的逻辑，前后端各维持在一套dockerfile build中独立生成image和不同的entrypoint run中运行，而在prod中前后端整合到了elixir image下,它们最大的区别是，dev环境下的webpack需要附加express 8080持续运行(npm run watch)，而prod模式下，一次webpack build就行了(npm run build)，不要持续运行。

好了，在针对prod的dockerfile和docker-compose.yml作修改之前，先改几个源码中的文件：

配置文件config/prod.exs中的config :ellie, Ellie.Repo段

在adpter条目下增加：

```
  username: "postgres",
  password: "postgres",
  database: "ellie",
  hostname: "database",
  port: 5432,
  ssl: false,
```

以上是ellie container实例启动时连接postgresql实例的配置。

database是数据库所在主机的主机名，docker-compose.yml中数据库 postgresql9.5对应container的ID，一般是database，对于那个ssl，如果不加ssl，会在运行时出现ssl not avaliable

config :logger段也换成这个：config :logger, :console, format: "[$level] $message\n"，否则接下来被一个run.sh包含的phx.server控制台不出任何信息。

assets/package-lock.json中，找到natives，升级一下其版本

"natives": { "version": "1.1.6", "resolved": "https://registry.npmjs.org/natives/-/natives-1.1.6.tgz" },

以上是为了在防止nodejs在编译deps时出现natives有关的错误。

dockerfile中：

DEPS下加一段安装postgresql-client:

```
# Install postgres-client
RUN echo "deb http://apt.postgresql.org/pub/repos/apt/ stretch-pgdg main" >> /etc/apt/sources.list.d/pgdg.list \
    && wget -q https://www.postgresql.org/media/keys/ACCC4CF8.asc -O - | apt-key add - \
    && apt-get update \
    && apt-get -yq --no-install-recommends install inotify-tools postgresql-client-9.5
```

然后就是dockerfile主体部分了：

```
# Download Elm platform binaries
RUN mkdir -p /tmp/elm_bin/0.18.0 && mkdir -p /tmp/elm_bin/0.19.0 \
    # goon executable for Procelain Elixir library, to run executables in Elixir processes
    && wget -q https://github.com/alco/goon/releases/download/v1.1.1/goon_linux_386.tar.gz -O /tmp/goon.tar.gz \
    && tar -xvC /tmp/elm_bin -f /tmp/goon.tar.gz \
    && chmod +x /tmp/elm_bin/goon \
    && rm /tmp/goon.tar.gz \
    # Elm Platform 0.18
    && wget -q https://github.com/elm-lang/elm-platform/releases/download/0.18.0-exp/elm-platform-linux-64bit.tar.gz -O /tmp/p
latform-0.18.0.tar.gz \
    && tar -xvC /tmp/elm_bin/0.18.0 -f /tmp/platform-0.18.0.tar.gz \
    && rm /tmp/platform-0.18.0.tar.gz \
    # Elm Format 0.18
    && wget -q https://github.com/avh4/elm-format/releases/download/0.7.0-exp/elm-format-0.18-0.7.0-exp-linux-x64.tgz -O /tmp/
format-0.18.0.tar.gz \
    && tar -xvC /tmp/elm_bin/0.18.0 -f /tmp/format-0.18.0.tar.gz \
    && rm /tmp/format-0.18.0.tar.gz \
    && chmod +x /tmp/elm_bin/0.18.0/* \
    # Elm Platform 0.19
    && wget -q https://github.com/elm/compiler/releases/download/0.19.0/binaries-for-linux.tar.gz -O /tmp/platform-0.19.0.tar.
gz \
    && tar -xvC /tmp/elm_bin/0.19.0 -f /tmp/platform-0.19.0.tar.gz \
    && rm /tmp/platform-0.19.0.tar.gz \
    && chmod +x /tmp/elm_bin/0.19.0/* \
    # Elm Format 0.19
    && wget -q https://github.com/avh4/elm-format/releases/download/0.8.0-rc3/elm-format-0.19-0.8.0-rc3-linux-x64.tgz -O /tmp/
format-0.19.0.tar.gz \
    && tar -xvC /tmp/elm_bin/0.19.0 -f /tmp/format-0.19.0.tar.gz \
    && rm /tmp/format-0.19.0.tar.gz \
    && chmod +x /tmp/elm_bin/0.19.0/* \
    # 以上都是准备elmlang的binaries到tmp下的原逻辑，，以下准备整个app执行环境,命名为tmp2是为了将这二步骤以对应的方式列出。
    # 这里的tmp2，其实对应原版的dockerfile中是 add . /app，只是原版的构建出来在单机跑起来没事，在迁移安装到别的docker主机上跑起来，会提示找不到
文件(定位不到正确的app顶层。所以deps.get时会找不到package.json等,entrypoint也找不到run.sh)。你多构建几次原版dockerfile与这里对比就知道了。
    # 你可能已经注意到这条很长的RUN，它将所有关于生成app的逻辑都维持在一个RUN中，否则就超了docker构建时的分层文件系统了，会导致不意料的事情发生。猜
测原版 add . /app 就是没有维持在同一个文件系统中。docker-compose.yml中的volume也会不能生效。
    && git clone https://github.com/minlearn/ellie-corrected /tmp2 \
    && mkdir -p /tmp2/priv/bin \
    && cp -r /tmp/elm_bin/* /tmp2/priv/bin \
    && mkdir -p /tmp2/priv/elm_home \
    # 安装elixir相关的所有扩展并生成项目的数据库文件
    && cd /tmp2 \
    && mix deps.get \
    && mix compile \
    && mix do loadpaths, absinthe.schema.json /tmp2/priv/graphql/schema.json \
    ## 安装nodejs相关的所有扩展，并生成项目的webpack静态文件
    && cd /tmp2/assets \
    && npm install \
    && npm run graphql \
    && npm run build
```

至此，生成构建了所有项目运行时的资源。

已经差不多可以运行了。准备**ENV**预定义的参数，**docker run**时会欠入到实例：

```
ENV MIX_ENV=prod \
    NODE_ENV=production \
    PORT=4000 \
    ELM_HOME=/tmp2/priv/elm_home \
  SECRET_KEY_BASE="+ODF8PyQMpBDb5mxA117MqkLne/bGi0PZoTl5uIHAzck2hDAJ8uGJPzark0Aolyi"
```

定义运行

```
# WORKDIR的主要作用就是定义接下来所有指令，尤其是entrypoint等的路径
WORKDIR /tmp2
EXPOSE 4000
RUN chmod +x /tmp2/run.sh
ENTRYPOINT ["/tmp2/run.sh"]
```

这个run.sh是分离postgresql所在容器和ellie所在容器的entrypoint，所有连接数据库初始化的工作都要在这里完成，因为它继承了ENV关于prod的预埋参数所以运行时不会出错，否则比如在非docker构建的情况下，你把mix phx.server单独在命令行中执行，会出现如下错误：(EXIT) no process: the process is not alive or there's no process currently associated with the given name。你就需要在run.sh中export所有这些参数，这也是docker的联合文件系统在编译（dockerfile）/运行(run.sh)不同阶段需要做到逻辑同步的要求。

run.sh的内容（它是**git repos**中要新增的一个文件，需提交到新**git repos**中）：

```
#! /usr/bin/env bash
set -e
cd /tmp2
until PGPASSWORD=postgres psql -h "database" -U "postgres" -c '\q'; do
  >&2 echo "Postgres is unavailable - sleeping"
  sleep 5
done
mix ecto.create
mix ecto.migrate
mix phx.server
```

最后,**docker-compose.yml**也一目了然了。

```
ellie:
  image: minlearn/ellie-corrected
  links:
  - database:database
  ports:
  - 4000:4000
  restart: always
  environment:
  - SERVER_HOST=52.81.25.39
database:
  image: postgres:9.5
  environment:
  - POSTGRES_PASSWORD=postgres
  restart: always
```

minlearn/ellie-corrected是我在dockerhub上编译正确的ellie，实际上，上面的ellie的volumes同样是没有起作用的。留给其它人解决吧（这就是分层文件系统给人理解上带来的极大不便）。反正项目部署到任何支持docker的机器都可以启动并进入ellie所在IP:4000的界面了。

假设上面的没加SERVER_HOST,进去你会发现ip:4000/new显示ellie的动画，但一直hangout，控制台显示，[error] Could not check origin for Phoenix.Socket transport.

这就需要设置SERVER_HOST=ip变量了(这个ip是你部署ellie所在机器的外网IP或被访问IP：4000所在的IP)，这个变量不能放在dockerfile中，也不能放在run.sh中(因为这二个文件要做进docker image中的，而你无法预知要将这个docker image放哪个IP的主机上)，故要放在docker-compose.yml中ellie段下在实际开启ellie container时指定，比如我部署运行时的IP是52.81.25.39。

其实docker就是一个通用的应用和OS的虚拟容器，它可以同时虚拟出我在《DISKBIOS》系列设想中用openvz虚拟出的同时运行的，却又可应用可OS的通用虚拟环境。只是它使用的aus联合文件系统我一直都不太喜欢，因为会带来污染问题和以上说到的编排dockerfile时的理解不便，突然想到联合文件系统会不会是客户端的安卓应用缓存清理的技术，其存储中，系统/应用双清的技术会不会也与它有关，就不那么讨厌了。。

有了ellie和公网盒子的群晖，你就有了一个超级方便的js学习环境，软硬环境的超级联系，使组成一个高效的生产工具变得可能，或许你还需要一个小终端，比如，《利用7寸小本打造programming pad 的 segment box》？

关注我，关注“shaolonglee公号”

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

把群晖**mineportalbox**当**mydockerbox**和**snippter**空间用

本文关键字：把群晖当真正产品级的**mydockerbox**用,利用**docker hook dir as volume**在群晖上制造**snippter**空间

虽然在前面的《群晖docker上装ellie》和更前面一些文章中我诟病过docker aufs的诸多弊端，但不可否认的是，docker这种能虚拟user mode OS能虚拟application的机制能成为开发和部署的利器，比如结合git server webhook能构建出dockerhub这样的自动构建/发布平台（除了开发，部署，甚至于dbcolinux中我用openvz模拟其用于“装机”），windows 10以后也把windows2016中的docker技术应用到了桌面级-从此桌面沙盒程序将一切程序数据和程序文档数据隔离在container内，动态装卸，能在APP级允许程序实现类手机应用的“缓存清理”---当然windows docker也可用于服务性程序。bitnami打包的服务器程序已经基本全部有dockerized版本。。。越来越多的迹象表明，docker这种虚拟化机制还是十分流行和有用的。

其实群晖套件spk就是沙盒原理，群晖在技术上可以仅实现为docker only box，比如它的photostation等都可以实现为docker，变成纯dockerbox用。变成类上面win10+docker支持的hostos+guest all docker化架构，只是它要兼容没有docker支持的机型，所以不便全盘docker化。

好了，现在接《群晖docker上装ellie》，继续讲解未完成的东西，接《使用群晖作mineportalbox（2）：合理且不折腾地把webstation打造成snippter空间》,本文也会讲如何在群晖上把docker打造成又一个snippter空间:而且这个空间也是可以统一像home photostation,home cloudstation一样置入home中的。

在群晖上装**ellie docker-compose.yml**

群晖6.2的docker是不支持直接安装docker-compose.yml的，只有把minlearn/ellie-corrected和postgresql9.5的镜像先在群晖中下载下来安装，然后先用postgresql镜像开一个容器postgresql1，主机端口不必是5432，但容器端口要是5432，密码环境变量设置一下，volume可以新建一条挂载到主机的home/docker/postgresqldata中，对应docker的/var/lib/postgresql/data，定为rw。这个目录会屏蔽docker中的/var/lib/postgresql/data，因为下一次你使用不同的主机目录把它volume出来到新的主机目录，出来的数据是全新的postgresql给你的模板数据，所以，任何主机目录中的volume对应体，在重新链接或转移时，应该备份一次。

然后，用ellie-corrected镜像开一个容器，按上文加上4000:4000和自定义环境变量SERVER_HOST=your ip。link到新开的postgresql1容器，别名定义database。ellie会启动，等待一会ellie便会连上postgresql，mix ecto-create,migrate,phx.server。等启动完成，创建快捷方式到群晖桌面为群晖的公网IP:4000。点击就可以访问了。 ellie-corrected的volume 在上文讲到，ellie docker-compose.yml中为ellie-corrected设定的那条volume会不能生效。问题是：你把容器的/app映射到主机上的/apphost，启动ellie后会在日志中显示/app/run.sh执行permission denied，这是因为，1：通过volume postgresql,volume ellie这样的方式mount到主机目录的目录是容器实例运行时，自动屏蔽掉容器端的那个目录的，而使用主机目录的一切的。包括权限。2，像postgresql的volume是不包含任何代码和可执行体的数据路径，所以没有执行权限问题。而ellie中有一个/app/run.sh要执行，要知道，docker内部的权限体和host上执行这些代码的执行体是不一样的。启动时ellie自然会有权限问题。

> 其实群晖docker默认给docker volume划分的共享文件夹是根目录下的那个docker，在这里建立的docker volume依然会有权限问题。而群晖也没有为docker建立一个统一docker用户，因为各个docker内部的权限都是不一样的---分属各种各样的docker内部os template定义的用户。那么，有没有一种方法，将它转为主机上统一用户，使得docker出来的任何需要权限的目录在主机上都可以通行呢？
>
> 有一个workarounds可以解决，统一把/docker或home/docker中，volume出来的主机目录设为EVERYONE读写权限即可。这样会有安全隐患，但群晖一般都是一台独用的，admin,guest都是被禁用的。所以能保证你当前用户不被破解，安全是可以基本得到保证的。

那么，最终如何实现在主机端访问到ellie /app，像postgresql volume一样使用ellie volume "/apphostdir /app"的方式呢？

修正**ellie**所用的**/app volume**

VOLUME这个参数可以透出容器内部目录，当然不用VOLUME参数，在了解容器ROOTFS中存在/app的前提下（这需要用户看过ellie的dockerfile），直接volume出来/app到/corrspondinghostappdir也是一样的，使用VOLUME更能清晰化这种过程和目录(另一种bind mount)。

就如同WORKDIR定义了接下来在dockerfile中的当前目录环境一样，VOLUME也规约了一段dockerfile中上下文中，那些影响各层aufs构建打包过程中，向最终volume定义透露出来的最终aufs叠加层。一个volume指令后面的指令(主要是run影响aufs叠加）会对它上面的volume不产生作用。即，仅对调用volume指令前的那些aufs操作产生的结果有意义。

所以这个 VOLUME /app放在chmod+x /app/run.sh后也会是没有意义的。你可以尝试移动/app/run.sh使之放在docker rootfs的其它地方，自己尝试吧。

好了，通过上述的讲解，你可以通过docker把一切程序用到的data和app数据归纳到home/docker/xxx下。将群晖打造成一个真正产品级的mydockerbox,和各种volume制造成的snippter空间。

关注我。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在群晖**docker**上构建私有云**IDE**和**devops**构建链

本文关键字：云**IDE**。**docker as cloud ide**，在群晖上安装**docker gitlab,gitlab ci for docker**

在以前的文章中我们说到docker是一种，集云虚拟化，装机，开发机，user modeos,langvm,app runtime为一体的东西。(或者不严格地说，仅仅可以当所有这些东西来用)。而这，其实就是我们一直想集成达到的DISKBIOS方案。

在《docker as engitor及云构建devops选型》一文中我们还说到，docker可用于组建私有devops，模拟engitor的效果，在那文的文尾我们提到云IDE，git是这个云IDE收集工程源码文件的云化过程（git同时是实现为客户端也是服务端一体的，所以它是云IDE客户端负责收集工程文件，在服务端它返回给下一级CI过程），那么集成了CI的git服务器实现品(如gitlab version8+版本以上自带CI模块)，就是云IDE中定义如何自动化构建这个工程的过程。

可见在云开发中，docker生态是一个非常流行和强大的东西，云IDE的先进理念实际就是devops(实际上，像gitlab这样的实现品已有cloud ide这样的插件)。

下面我们就来讨论如何用docker的gitlab ci模拟云IDE中的自动化构建链效果。我们的环境是群晖docker上。用外置postgresql实例的方法，我们最终要实现的结果，就是实现gitlab以docker为executor的CI链，可以实现面向docker为开发机的构建，发布的自动化过程。VS 托管在远处的devops服务器，有一个私有devops的好处是，我们可以在本地即时快捷地观看和控制程序构建的过程。

群晖**docker**上搭建**gitlab**

跟《docker上安装ellie》一样，这同样是个复杂的过程，gitlab是ruby的，gitlab cl是nodejs的，跟ellie docker一样是涉及到多语言环境的。我们复用ellie中的postgresql9.5镜像。

我用的是2019.2.2号左右dockerhub上sameersbn/gitlab的GitLab Community Edition 11.7.0（在他的镜像中，7.4.3之前版本，镜像里包含所有组件，7.4.3版本镜像里只包含核心组件：nginx、sshd、ruby on rails、sidekiq），不要下载官方的gitlab/gitlab-ce，那个镜像里内内置了postgresql数据库。启动时占用内存过大。而且不正交。由于这个镜像很大，外网线路下载起来很费事，容易中断，我们可以利用上shadowsocks的方法，在windows上开一个允许局域网连接。然后在群晖控制面板->你当前使用的网络界面中配置一个代理服务器。之后下载就会快多了，下载完全后，同时下载redis:latest，这样postgresql9.5,redis,gitlab镜像都有了。先启动postgresql和redis的实例。

再开启一个sameersbn/gitlab的实例，link到postgresql9.5:别名postgresql，redis:别名redisio，80容器端口映到8001，因为主机群晖占用了80。增加几个环境变量，

```
GITLAB_SECRETS_DB_KEY_BASE=随便写
GITLAB_SECRETS_SECRET_KEY_BASE=随便写
GITLAB_SECRETS_OTP_KEY_BASE=随便写
```

启动，gitlab会自动连接postgresql，发现容器退出，查看日志后发现，FATAL: role "root" does not exist,数据库中没有root用户，这是因为gitlab实例对postgresql实例的数据库有root检查，及其它一些硬性配置上的要求。下面这些做：在群晖的web版进postgresql1实例的终端机界面(点新增会自动打开一个bash终端)新建一个root用户并赋于权限。

```
su - postgres
psql
create user root with password 'password';
ALTER ROLE root WITH SUPERUSER;
此时再尝试启动应该没有上述错误了。但又退出，且提示psql: FATAL: database "gitlabhq_production" does not exist
CREATE USER gitlab WITH PASSWORD 'password';
CREATE DATABASE gitlabhq_production OWNER gitlab;
GRANT ALL PRIVILEGES ON DATABASE gitlabhq_production TO gitlab;
\q
```

最终启动成功，发现内存维持在1G多比gitlab/gitlab-ce少很多，打开群晖ip:8001，会提示让你修改root的密码，这个root是gitlab用户的不是postgresql的。用root和这个密码登录，进群晖ip:8001/admin/。

现在可以在上面建立repo，clone的界面上显示的是localhost，你需要额外加二个启动环境参数来定制这里显示为localhost的部分,另外如果你想导出各种volumes，参照ellie关于权限的处理方法就行。

最后，然后进admin/runners查一个token，备用。

在群晖**docker**上安装**gitlab ci for docker**

这里的坑有点多。

首先不要下载sameersbn/gitlab-ci-multi-runner:latest(gitlab/gitlab-runner也是multi的)，这个版本太老，启动后link到一个别名为gitlab的第一步安装的gitlab实例，sameersbn的runner是可以定义环境变量注册的

```
RUNNER_TOKEN：上面的token
CI_SERVER_URL：http://link到的gitlab别名:80到主机的转发端口/ci
RUNNER_DESCRIPTION：随便填
RUNNER_EXECUTOR：这个暂时先填shell
```

虽然方便，然而我尝试了下这种方法在上述sameersbn/gitlab-ci-multi-runner版本中根本无法使用，一直提示404,PANIC: Failed to register this runner. 404，PANIC: Failed to register this runner. Perhaps you are having network problems

我们下载同gitlab版本的gitlab/gitlab-runner:v11.7.0，启动后link到第一步安装的gitlab别名gitlab，然后进终端机用命令行方式注册runner到CI：

像上一个方法一样新建一个bash，会进入/home/gitlab_runner中，打入gitlab-runner register会提示输入六个选项的参数。依次是：

```
url：这个填http://gitlab/ci
registration-token：这个填第一步获取备用到的那个token
executor这里填docker
docker-image这里我可以按需求填alpine:3，这个有什么讲究呢？这什么选这个呢？预置的有什么用呢？其实这是构建Docker image时填写的image名称，根据项目代码语言不同，指定不同的镜像。
description随便填
tag-list这里填v1170
```

所以你看出来了，以后devops的触发主要是由其中对应到这里的tags来触发，docker ci build的原理其实就是以某docker image为虚拟机，在里面一层一层构建fs，然后叠加成最终镜像，这里的docker-img即为那个虚拟机。

所以docker image加tag的组合可以根据很多不同目的来定义多个。多用。

以上我们注册的runner是全局的。也有per工程私有的runner，上述tag为v1170的docker runner就是工程全局共享的

至于各种参数具体有什么用，等以后讲吧。那个触发文件流程定义.gitlab-ci.yml更是复杂,反正runner是建立起来了，在项目的/settings/ci_cd，CI/CD Pipelines -> Runners activated for this project，会看到已激活的runners

---

还有，我们可以搞个for elmlang,下回吧，这样在我们的私人服务器上就可以即时持续集成了（以达到不断向其喂给碎片化项目内容，持续集成为大应用的目的，这也许就是微服务的由来）

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

docker as engitor及云构建devops选型

本文关键字:**docker as engitor,**云构建**devops**

在发布《engitor as demo show engine,applet container》时，我们谈到engitor是一种延续langsys，独立于开发，但对接开发增强开发，及负责整个发布和部署,甚至运行（除了xaas层次的那部分运行）的综合过程统称，解决开发完成后，增强开发环境，问题域demo组装及发布至上线的一系列后续工作。类似应用服务器，但不止这些。比如vs appserver,an engitor可以是强化语言系统之后可视化的开发增强支持(engitor=app engine as editor)甚至提供baas，paas这样的运行增强支持，而传统appserver仅负责特定的部分。

> 我们总把整个软件工程分为xaas,langsys,engitorx(appdomain),apps四个层次。越到最后规模越小，程序逻辑越片断化具体领域化，最终，它使源码形式的语言系统写出的源码片断程序，可以以可视化的形式开发，碎片化发布/累积的方式运行，使得练习和演示可以向最终应用逐步无痛地迭代。。我们一直在为这些领域选型。如上所述，appdomain的engitor承上启下是规模较大的一块，其选型也就越复杂。

在那里，我们主要选取了jupyter和openresty来说明engitor：

1)在用jupyter充当这个engitor时它同时是enginx，它的特点是.ipynb，技术上这实际上是一种web脚本和各种语言后端的服务环境"web engitor"(但是它支持cs app化engitor)，.pynb可以欠在webviewer中也可以欠在其它支持jupyter cs协议（类似cgi）的clientview中，由于engitor是支持多语言的。所以，它实际上是将多种语言源码片断(a note)统一发布成web应用的服务端脚本的形式并将执行结果返回。这其实就是动态web脚本的理念。但是它第一次实现了将不同的语言统一化为服务端脚本，且提供了一个在线IDE(以开发一段note测试一段note的行为)。而实际上足够复杂的ipynb是可以开发app的，也不限于用在线IDE的教育工具的形式去展现,其前后端一个是语言一个是应用就像普通WEB一样。

2)而openresty的enginx是纯运行方面的engitor选型，我们在《发布enginx中》，提到组件服务器环境，它使服务性程序的协议部分，变成组件的交互。将lamp,lnnp,lnmp结构扩展到可配置的通用组件化服务器程序的结构(实际上是利用lua为nginx写脚本)，而kbengine这样的结构就演示了如何用openresty来充当通用程序的enginx-game app

legacy engitor和devops云构建

以上选型都有几个共同的特点，1,在这种engitor是一个组装运行环境，这种语言环境“在线收集合成了”用户碎片化方式提交的源码逻辑，是个云构建化的开发环境类程序。2，且形成的engitor app要在这个engitor辅助下运行，因为它要面向源码片断输出这种源码下的应用。这此都符合我们对engitor选型的一惯要求和标准。

那么是否能构建一个engitor，它依然能够面向对一端是语言src逻辑输出另一端是应用输出而不局限仅用于要求输入端必须是源码，输出端必须是APP？(一言以蔽之通用化构建任意程序)，且不要求运行在以上具体engitor下？那么这还叫engitor吗？还有意义吗？

毕竟，我们想得到一个万用的engitor，将传统上从(linux的生态开始处,CUI处，那个时候仅有os kernel和toolchain)，将任何复杂应用的开发涉及到的多种语言源程序/二进制的编译过程，多种语言vm的打包过程自动化起来，将这些在传统上是构建脚本的编排技术，和OS的包管理技术考虑进来，甚至使构建本身云化和构建服务外部化云化，喂给远程构建-云构建，。形成自动化，云端脚本化编译的结果，并以此为运行目标，仅负责书写最终APP上的事。

这实际上就是输入端接受任意构建，输出端产生任意程序的单一要求而已。这样的engitor实际上以os为enginx运行，以能运行上其上的所有可能语言系统为engitor中的langsys。而engitor也不必是个jupyter+web执行环境式的“云构建”和中间件打包。比如，它可以是任何程序（非源码形式的某语言源码片断,二进制也可，非IDE类产出过程也可）构成的“云构建”和中间件打包。它可以没有任何关乎engitor意义上的输入输出。但是依然可以适用于engitor特例。

那么如何整合这些，这实际就是devops做的事。传统我们在PC上用各种开发用的虚拟机vagrant，那么我们现在有docker和devops

docker as engitor和云IDE。

在那文中我们讲到jupyter也有jupyter hub。实际上它相当于docker版本的github+dockerhub组成的devops。

docker as 通用构建技术和容器的情况下，实际上docker与docker-compose是二个独立的过程，docker只负责run，而github相当于ide中收集源码的工程环境，那么我们还可以得到什么呢？ 比如结合前面的ellie，我们可以在结合docker和gitlab cl for elmlang的情况下，把这个ellie ide放进去。做一个云IDE。

自由大开脑洞去吧。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

戒掉PC，免pc开发，cloud ide and debug设想

本文关键字：分布式**IDE**，**cloudide**，远程编码，远程调试**,jupyter with visual debugger support**

编程界有关于语言的圣战，OS之争，也甚至有代码编辑器是选择cui text ide还是gui IDE的选择的讨论。这次我们讲的是云IDE，其实，我们一直讲的是devops和云可视IDE这类服务端直接支持的开发部署，从《ellie可视化》到《docker as snippter空间》，从《一种设想：在网盘里coding,debuging，运行linux rootfs作全面devops》到《分布式IDE：osx一个完美的开发集中》，这些都是从不同层面去说的，毕竟这样一个工作要解决好多问题，而为什么要把开发做上云的理由也很明了：devops在云端，服务器环境入口网络好。我们也一直在寻求某种“免PC的开发”：如果把发布部署做上云了，如在《去windows去PC，打造for程序员的碎片化programming pad硬件选型》中所讲一样，我们甚至不需要一台PC仅需要一个平板就可以在云端构建，或者一个专用programmingpad，，实现戒掉PC把一切放到远端，免除本地重复部署，比如借助devops可以在一次性的虚拟沙盒环境中编程部署，干净，永远守护，开箱待用，，----- 它是云OS扩展一级的东西，与lnmp,openfaas云开发云部署同级，如果说后者解决的是部署前者解决的就是开发调试，当然也有将二者结合的，我们甚至谈到一种天然绑定debug设施的os和appstack结构(press esc切换运行态/开发态)。

云 IDE是新概念吗？不不不，早在 2010 年就有成熟的产品了，其实类似baota panel的文件管理器当cloud ide也可以，但是毕竟它缺乏真正的开发所需的一条路径上的东西。云调试与云编程支持。这里的技术是把IDE分布化，基础原理依然是设置协议（语言服务，调试服务API后端化）和前后端分开，这样前端可以ported到任何设备，《用开发本地tcpip程序的思路开发webapp》讲到，Api分离，实际上是从业务逻辑中分离了gui栈，这是一种经典的抽象，也是很久以前桌面界的编程方案了。其实web不是没有过前后分离，只不过很晚才将前后分离做到云函数和devops式建立api服务池的程度。

目前有很多实现，但我们接确到的大都就是jupyter和vscode online这二个，jupyter面向文档嵌入代码领域而vs code online面向真正的传统IDE在线化，它们二者都有明显的前后协议层支持。针对语言编程和调试。它们二者都可以与docker打通，而docker实际上是一个贯通开发部署综合的devops，所以与devops又有了联系。

vscode/vscode online

微软向云靠拢其营收已超过windows本身。github,azure,onedrive,vscode/vscodeonline都是例子，微软在 Build 2019 开发者5月份的大会上宣布了 Web 版本的 VS Code，即 Visual Studio Online Private Preview。在 2019 11 月 4 日发布公开预览版的 Visual Studio Online。

实现：

其实vscode本来就是一个分布式IDE和云化版本。(vscode本来一开始就是面向要被online的，它的插件设计规范中的Language Server Protocol，Debug Adapter Protocol就决定了,vscode的设计师一开始就是从分布式角度去做的,cdr的code-server就是这样的思路提出了web版的vscode，上面提到微软201911公布了 Visual Studio Code 1.40 版本，官方直接支持web搭建，只是没有code-server的完善比如它没有登入验证),前端上，从页面上直观地看，VS Online 就是一个 Web 版的 VS Code，vscode使用Electron(原来叫 Atom Shell),桌面vscode最初也是由atom而来，这使得前端很容易被ported到web，但这其实只是它的一个前端界面，而 VS Online 更强大的能力来自于vscode-server+remote development extensions，这里我们暂时把web或桌面那个editor界面称为前端，vscodeserver+remotedevelopment exs称为后端。

在vscode中，如果你使用了remote-devopment中的ssh组件（一般地，你用remote-ssh远程开发，用remote-container本地），会自动在远端下载vscode-server，这并不仅仅是打开远程文件这么简单（这仅属于工程组织文件托管范畴和代码托管），而是连IDE和插件服务都做在了远端了。------ 单纯的remote-developer是一个插件包，是用了同步/打开远程文件用的。在vscode的插件搜索栏中，搜remote插件出来的东西多得你理解不透。大部分是解决代码托管一类的，解决类似在wxsdk tool云函数同步到后端的功能(tencent-cloud-vscode-toolkit)，git可能是提交工具也可以是代码空间也可以是工程文件组织器。比如remote-github(注意到云函数和一些git平台，都有ide，只是git配ide白瞎了，因为没有调试的可能性。不过，虽然目前只是在测试阶段，微软已经实现了为github集成了基于vscodeonline的不离开页面的沉浸式webide,github人类的代码基因库结合全能可用的webide，这是极佳的整合和创新。),你要的那种网盘文件。codesnippter空间也有。类似google colab 网盘挂载。------ 真正发挥作用的是后端插件管理和其组成的那个IDE(ide是由大量plugins组成的嘛)服务，这正是vscode-server。只是架构上,，这个vscode-server，被没有被做成统一后端，vscode-server不能按版本单独部署，必须要用一个vscode-server-client来唤起部署，vscode-server按需部署必须至少绑定一个front。这二者不能随意组合,也没有配置项将IDE前端和vscode-server连接起来，所以，没用到远程插件服务的情况下，本地vscode开发并不需用到（比如，同为remote的remote-container并不需要用到vscodeserver?）。

所以，不满足桌面？不想使用浏览器？这也不是什么大问题，由于它的前后端分离，可以将vscode online接入后端云开发环境（或本地模拟出来的“远程”版本，不过这样失去了云存储和云开发的某些意义）把它变成vscode。同样可做到类本地vscode效果。。而vscode online是前后可分离,服务端可以是本地模拟的也可是真正远程的，更自由，允许桌面版连接服务端远程开发，不局限于web as front，当然如果可以你也可以定制出你自己的vscode online。如果你想任意组合这二者，实现"让web/桌面vscode统一remote后端"的功能：比如，你想在本地搭建一个GUI前端(仅把它当editor front)，却想调用后端的ide服务和插件，最终是为了编辑开发托管代码，或调用远程docker里面的应用呢，这当然也是可以做到的，1，自建codespace服务器(可以把它理解为桌面那个editor前端+vscodeserver)，在目标机器上安装 VS Code，搜索Visual Studio Codespaces (formerly Visual Studio Online)安装，并命令中注册，它的原理应该也是部署了一个vscodeserver。2，可以在远程直接安装github/cdr/code-server的那个web版，然后查看它带的codeserver版本，来决定本地桌面端可能会用上的版本对上，file->about->help可以看到它采用的code-oss版本和commitid。（2比1好，毕竟自带一个web前端是最基本的，桌面和移动端可以作为额外项添加）

这样你可远程同步代码 + 本地调试，远程托管代码+远程调试，也可本地代码+本地调试（vscode itself），也可以本地代码，远程调试。当然也有对接重量级docker和devops的。甚至上面谈到的“一次性的虚拟沙盒环境中编程部署”，代替程序员日常的维护多个虚拟机完成不同开发工作的需求(还记得程序员下班不关机这梗吗)。这就是remote Container。

> vscode/vscodeonline开源在github，叫Code - OSS,其源码倒是没有缺失和阉割也有完善的构建支持（https://github.com/Microsoft/vscode/wiki/How-to-Contribute#build-and-run），mit的开源许可也很友好，但是vscode二进制本身是this not-FLOSS license 的并带了telemetry/tracking，而且其remote-development却是不开源的，且规定二进制vscode-server不能被打包（见 https://code.visualstudio.com/docs/remote/faq，Can VS Code Server be installed or used on its own?Why aren't the Remote Development extensions or their components open source?）。所以，类似code-server,Che,VSCodium这样的公司和产品就只能自己编译源码。集成docker devops运营（https://opensource.com/article/20/6/open-source-alternatives-vs-code）。https://zhuanlan.zhihu.com/p/98184765,code-server是vscode的魔改版本(基本上，code-server采用了vscodeonline里面的vscode-server组件+web前端，由于code-server必须要绑定一个前端，这造成code-server是前后一体放在远端只能web界面)。微软也有托管计划Visual Studio Codespaces。code-server也有。

体验:

这就要说到体验了，追求一种存储也在远程，而且最好一条龙开发部署都在远端的IDE。一切硬件无线和软件云化的云计算时代，讲究没有任何随身携带的东西，体验就跟使用云笔记一样（云上存储空间和执行空间全包，作为后端，前端只需要要一个GUI，前后端通过协议跨架构交互）因此在PC和手机上都有支持。云IDE这样的东西最讲究体验，体验是第一位的，比如它比本地版本不要落后太多。web端的肯定跟本地的前端，移动的前端会有区别，体验和技术上的

移动端有ios的Serveditor（以前的vsapp），jupyter也有这类APP叫juno connect。macbook要arm化。也支持ipad开发，当然，功能可能稍微会有点受限。安卓端当然也有。这里的体验差别就更大了去了。

> 只是说实话我对ssh的稳定性很不放心。说实话它真不如一个网盘客户端（最好是一个类finder的集成在file explorer中）,不过市场中也有cloudsync的插件。web端的体验其实还可以，只要不刷新网页后台一直websocket连着。

jupyter

jupyter notebook最初来自于julia和科学计算，叫ipython,后来发展成了通用notebook工具。以前，我们曾介绍过它也是一个devops。因为它可以结合docker形成binder这样的平台和工具。jupyter只是个notebook，最初它用用来写文档中的代码块。后来被作为语言学习平台。不是面向生产和真正的IDE的。这类产品是cloudide，比如vscode/vscodeonline，VS Code 也有对它的支持：Jupyter Notebook Editor。

jupyter也是前后端分离开发的典型，它有一个协议系统，这种分离使得前端可以不限于webasfront，也可以是mobileapp。甚至小程序这种嵌入前端。比如第一步要将语言发展为repl(脚本语言天然有repl，而cpp这样的要借助一定手段或现有产品变通。如cling)，然后按jupyter的语言kernel协议写成插件接上jupyter。

它也有很多泛化产品：如接上docker的binder，如基于jupyter notebook的jupyter book(注意名字区别)，叫可执行的markdown（就是在md中嵌入ipynb片断，或者理解成在ipynb中写markdown），进一步可发展为利用云snippter笔记写blog,可导出整书pdf（我还没找到按toc能整书生成pdf的产品，那就成word了，gitbook算一个）。

还有如基于jupyter的可视调试：jupyterlab/ debugger。当然它也是分前后端和协议实现的。这就一步一步走向了真正的cloudide的段位了。

如果说上一代jupyter是面向文学编程和科学研究的，下一代是称为Jupyterlab的产品就有点ide的味道了，有Elyra这样的产品了。

下一文探索发明lang.sh:把jupyter with debugger supporter和code-server整合安装在minstackos

---

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在云主机上安装vscodeonline

本文关键字：**teamviewer**保持在线的替代品把**vscode terminal**当虚拟主机管理面板。**remote-openfaas,**打造**vscodeos:**用插件打通**openfaas**界面到**vscodeonline**，实现**ide.sh**与云函数容器对接

在前面《云主机上部署pai》,《云主机上部署openfaas》中，我们用同样风格的脚本写出了在云主机上部署的二个paas面板，pai类似虚拟主机管理器，而openfaas是paas->faas，综合这二者都是部署和devops面板，它们在透出的界面5523,8080处用web操作。然后我们在《戒掉PC，免pc开发，cloud ide and debug设想》又遇到了vscodeonline，这三者都是云主机构造paas APP以“在线开发和部署”的OS扩展，体验良好度又都有前后分离十分接近，因此，我们这次也把vscodeonline集成在这里，组成成为minstackos的主要部分。未来，我们把所有讲到的paas app用这些面板串联起来，使它们成为可一键在线开发和部署的APP，形成minstackos的appstore来源。

> 在《在openfaas面板上安装onemanager》中我们讲过云主机的ssh往往是很容易断的，如果是graphic linux下，需要tv这样的这样的远程桌面方案来保持长时间在线，但实际上，vscodeonline也有linux终端面板。remote-ssh连接的vscodeonline可以保持长时间ws在线而且可以点上面的一个图标扩展到整个IDE的编辑区。因此体验上，后者可成为前者的良好替代。

好了不废话了。下面依然在一台ubuntu 1h2g上进行。

基础

一些变量

```
MIRROR_PATH="http://default-8g95m46n2bd18f80.service.tcloudbase.com/d/demos"
# the code-server web ide
CODE_SERVER_PATH=${MIRROR_PATH}/codeserver
```

安装codeserver

脚本被做成融合成安装pai和openfaas的风格，按standalone方式安装,以root身份运行。你可以集成自己需要的语言和插件服务到这个IDE，以做到尽量开箱即用。

```
# install codeserver
installCodeserver() {

    echo "====================codeserver install progress======================="
    msg=$(mkdir -p ~/.local/lib/code-server-3.5.0
    wget --no-check-certificate -qO- ${CODE_SERVER_PATH}/v3.5.0/code-server-3.5.0-linux-amd64.tar.gz > /tmp/code-server-3.5.0-
linux-amd64.tar.gz && tar  -xvf /tmp/code-server-3.5.0-linux-amd64.tar.gz -C ~/.local/lib/code-server-3.5.0 --strip-components
=1
    rm -rf /tmp/code-server-3.5.0-linux-amd64.tar.gz
    ln -s ~/.local/lib/code-server-3.5.0/bin/code-server ~/.local/bin/code-server
    PATH="~/.local/bin:$PATH"

   # systemd service start
    rm -rf ~/.config/code-server/config.yaml
    cat << 'EOF' > ~/.config/code-server/config.yaml

bind-addr: 0.0.0.0:5000
auth: password
password: pleasecorrectme
cert: false
EOF

   # systemd service start
    rm -rf /etc/systemd/system/code-server.service
    cat << 'EOF' > /etc/systemd/system/code-server.service

[Unit]
Description=code-server
After=network.target

[Service]
Type=exec
ExecStart=~/.local/bin/code-server
Restart=always
User=root

[Install]
WantedBy=default.target
```

```
EOF

    systemctl daemon-reload && systemctl enable code-server
    systemctl start code-server 2>&1)
    status=$?
    updateProgress 95 "$msg" "$status" "code-server install"
}
```

安装完成后记得修改~/.config/code-server/config.yaml下的密码，端口为5000。如果你要用上证书，就最好搭配脚本中的nginx+certbot申请的那个。cert: false也可以用假的localhost的那个，但是基本没有什么用。

其实，利用那个remote-container，可以把openfaas-cli跟vscode连起来，利用工程源文件下的yml模板（.pai.yml,openfaas-cli.yml,etc...）文件打造一个带开发部署的工程资源组织文件，形成remote-openfaas效果：多环境多语言下，需要频繁切换环境，一次开发总是跟一次塔环境开始的，这也是vagrant和docker对于开发的意义（以前是vm，没有模板机制），而docker用于开发也用于部署。以后一套APP天然就有一个online webide守护，自带开发环境了。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

利用citrix xenapp and xendesktop建立你的云桌面

本文关键字：云办公。真正可用的个人云桌面,云下载云播放。**GPO dns**，**xendeskop storeweb**无法完成您的请求

其实云电脑桌面，虚拟桌面vdi,瘦客户端这样的东西出现很久了，只是它们从来没有像今天的疫情时代一样让人对他们更为关注。一般来说谈到云电脑人们都会想到最初的vps，正如谈到云桌面这个人们会想到远程桌面这个工具层面的东西一样，但其实云桌面与办公，可以是内涵和外延很广的东西，比如虚拟化这些东西结合起来，造就了一些专门做vdi的公司，如vmware,pd,citrix，（citrix很早就提出了BYOD，带着你的设备办公的理念。它的xenapp and xendesktop产品就是vdi产品，当然citrix不止这个产品，它还有虚拟化全套，和workspace云工作台这样的东西-类似钉钉,slack）同类产品是vmware的view。。越来越多的云服务商开通了它们的云桌面，云手机产品。，，还有很多。

只是云电脑用于生产力和异地办公，甚至发展普及到个人使用代替实体电脑的程度，需要解决很多问题。相信谁都尝试过用一般的云主机和一般的远程桌面办公，用云电脑，性能和速度，成本，都是大问题。gd5446显卡的云机在远程桌面下打开有flash的网页，鼠标寸步难移，对于多媒体应用，有点接近云游戏的实现难度了，可是我们知道，现在这个时代，5G才刚刚开始，云游戏还没有被普及。

citrix，是如何尽可能在现有条件下克服问题的，又达到什么样的成果？citrix中的app,xdesktop，使用一种ica的协议。这种协议可以工作在恶劣的带宽环境中，极大利用带宽，而且可以实时多媒体（类似云游戏的那种）,可以打视频电话等，体验接近普通本地办公电脑，简直可以类比虚拟机界中的pd(pd的远程桌面也做的不错可以发布remote app)，云存界的icloud+icloudriver(其算法十分稳健体验好。比如删除中同步，单用户范围内秒传)。协议加速是一方面，另一方面，云电脑的技术实际就是pve虚拟化那套，加spice+virtiogpu显卡虚拟加速化那套(当然还有其它跨网络的GPU虚拟化方案)即vdi技术。

反正我是怀疑这效果，所以我看了下阿里云的云桌面产品，阿里的云桌面称为图形工作台，按量测试，先开通集群，然后开通实例，只在杭州h区i区有机器，系统是windows 10 2019srv based on ltsc，最低2h2g的机器，看看显卡驱动,是qemu -vga std虚拟出来的机器，当然也有用vgpu显卡的。硬盘也是virtue scsi，非blk（我的装黑果文章中，用这种显卡会更兼容，装黑果失败了所以我来折腾citrix windows了），集群和实例都收费，集群更贵阿0.71元一个小时。先看效果。带宽给足的情况下，效果居然还可以，播放高清视频和本地差不了多少，也没有出现开flash网页鼠标飘飘的情况！可做小型云下载云播放系统。

这部分费用产生在哪？，citrix是一个分布式结构，讲究扩展和性能，部署在多机环境，基本大量使用windows的基础设施来构建自身（比如用windows的身份机制，用windows的iis，用windows的数据库，需要配合windows域控，iis,selserver一起工作，很吃内存），一般使用多台windows server来构建服务器集群（也有linux版的Citrix），服务器有二大件：desktop delivery controller（DDC，其中又分几个小件，其中有Delivery controller业务核心,Licensing Server认证服务器,studio,storefront这二配置工具,citrix director组件。）和virtual delivery agent（VDA），virtual delivery agent就是安装在按量机中的程序，服务器的主要部分delivery controller是安装在阿里云别的服务器上的 ，感情阿里云是在这里建立了各种delivery controller服务器，与citrix授权一起产生租费的（citrix收费的方式是订阅制），然后通过某台阿里云称为gws的域服务器（暂且这么命名吧）作认证透出服务，我们可在网卡dns处找到它172.16.0.2xx，

现在我们来考虑自建所有的服务器自测效果，如果自己建的也能达到差不多的效果，证明citrix是高度可用的,citrix的授权有30天的免费期可以拿来测试。我重新开了2台港区按量2H4G装的win2019 datacenter with ui，港区网络有时好有时坏，方便长时间看综合效果，，

1台2C4G测试机上安装通用服务器（域控，mssqlserver）和ddc业务服务器并配置，相当于gws

这部分服务器程序在新装机器上，包括os在内约占2G多启动内存,随着运行会占更大内存。因此需要一个4G左右的机器。且Delivery controller是（业务核心）组件，生产环境按接入的用户数，基本上4G为起步。如果内存不够且是ssd机，手动加大虚存尝试。我开的4G。

先处理下服务器，，计算机名修改为ddc，表示这里安装的是通用windows servers程序和ddc服务器。方便辨认，计算机名也有讲究，licensing server用它跟citrix注册，因为这台测试机器要作为许可服务器在内的服务器，许可的主机会使用主机名向ctirix注册。本机如果有域先脱域(系统属性，计算机名修改处，从当前域转为WORKSPACE工作组，这样就脱域了)，注意，如果ping localhost得到结果是你的windows用ipv6 ::1表示域名，那么添加REG_DWORD值16进制，HKLM\SYSTEM\CurrentControlSet\Services\Tcpip6\Parameters\DisabledComponents，设置成20，重启这样ipv4就好了。dns为自动获取。

把官方按量机中的用户目录/下载中的二个文件citrix_virtual_apps_and Desktops_7_1912.iso和citrixcqi.msi复制过来（citrix专属驱动就复制不过来了）。其内就是主要的服务端安装文件了(也有客户端receiver for Windows and osx，在客户端,我们要使用到的就是其workspace app中的->receiver中的->viewer,注意到receiver 和delivery 是一对相反词)。打开iso，从iso复制出来安装文件夹到桌面，安装期间几次要重启计算机再继续，如果直接使用iso安装会失去挂载。

——— PS：如果你这台机器性能不够，其实你可以在这里仅安装dc，而把licensing server,studio,和storefront。甚至域控，都放在域内另外一台机安装，或者把许可服务作为通用服务放到域内另外一台机中去，为了达到动态组装，扩展，性能，这也是生产环境中推荐的方式,ddc不能在一台域控上安装只能在一台域成员上安装，除非你先装DDC再装域控，因为安装程序的逻辑是：当它发现本机有域控，就不能安装Delivery controller 和virtual delivery agent。至于mssql，它是可以跟域控一起装的，我们这里也是这样做的，虽然install sqlserver on a domain controller is not recommended。 我们的测试是域控,mssql,ddc全装在这台机。 ———

执行autoinstall，安装DDC，组件统统选上除了Director（在安装程序的逻辑中，以上都可以部分安装，directer可选安装,），选上sqlserver，安装程序建议5g，要开的端口会提示你许可证服务器7279,27000,80828083,Dc:80,89,443,Storefront:80 443，统统在服务端防火墙处放行，最后发送珍断信息不要勾选，因为你是在本地用户下安装，非域下，安装完窗口末尾有一行提示”本机需要域才能配置DDC”。ddc安装完后，如果需要新添组件，可以通过programfiles/XendesktopServersetup/xendesktopserversetup.exe修改，需要保留安装程序文件夹。

现在安装域控，windows域控相当于osx server的部署描述符文件，服务器管理中添加角色和功能，,基于角色或基于功能的安装，选择ad目录域服务,接下来功能就不用选了，勾“如果需要，重新启动服务器。”部署后黄色叹号处需要配置，将此服务器提升为域控制器，添加新林，域名填ctx.srv。表示，这是一个citrix用处的srv的集群，这是域的内网根域名（整体很容易理解：这是一个命名为ddc，以ctx域林为根，自身为控域的域成员，使用的内网dns逻辑是xxx.xxx.srv，避免使用com那些结尾的域名，如果有人注册了这个com，就会登录到它的服务器，所以这里还是不用com的好），域名系统服务器全局编录，林功能级别保持为server 2016，netbios域名默认CTX。无法创建dns委派，略过，会自动重启，本地adminstrator会变成域用户主机会变成ddc.ctx.srv，下次登录不到本地administrator,只能用域逻辑登录。因为是域控，不可能再作为本地机器登录了。ctx/administrator自动也是本地管理员组。设置domain policy，让域内主机自动配置。比如如果你想让域内主机都不用按ctlaltdel登录，需要在计算机管理，组策略中找到域，default domain policy,计算机配置，策略，windows设置，安全设置，本地策略，安全选项，交互式登录，由没有定义改成已启用。

———— Ps:我们手动设置下mssqlserver，以免配置时等待过长。让sqlserver工作在域上使用ctx/administrator logon，默认安装方式下给你的logon帐号是nt认证built in network service,网络是tcpip enable, tcp all使用动态端口。这种方式下等待过长。

事实上mssql不推荐安装在域控上，如果你要接入另外的域机器上的mssql，需要那台机器上的实例选择了“混合模式”验证身份登陆，这样才可以顺利的进行接下来的远程连接。。Services account用CTX/administrator,密码填上。auto startup type。安装完后，开始菜单打开SQL Server configure manager, network configure, 找到实例for sqlexpress的protocols,tcpip enable,给于所有ip，enable,yes,端口1443，特别注意ipall也要1433,同样在这个窗口，去SQL Server services那里右击restart一下，windows防火墙里放行。打开防火墙→高级设置→入站规则→新建规则→选择C:\Program Files\Microsoft SQL Server\MSSQL14.MSSQLSERVER\MSSQL\Binn\sqlserver.exe，如果你的防火墙关的，不用去理这步。 ————

现在来配置，使ddc和vdc运行起来。现在开始在studio中配置（如果登录的不是域用户无法继续），打开studio，交付应用程序和桌面，创建站点的交付控制器开始，，使用30天免费订阅(也可以在这里选择授权文件安装，也可在localhost8082处进行，这时，主机名就会影响往citrix注册)，选择空配置，并输入站点名称default，studio,storefront等要用到sqlserver信息，如果是连接外部域主机，三个位置都填”mssql域主机名,1433“，提示没有权限，使用同域的ctx\administrator继续,连接，正在生成数据库架构。正在创建storefront群集,完成。如果你使用内存有限的机器，配置服务阶段会卡很久。

第一个站点建好了。可以在内网直接用浏览器加html5查看，但无法连接远程桌面，因为站点中没有远程桌面或app定义，在storeservice可以看到 http://ddc.ctx.srv/Citrix/StoreWeb/,,这称为receiver for web站点。进去但是里面是没有任何程序的。配置成公网访问，会在iis管理器default site处添加一个https443段在http段下，进去storeweb一样是空的(实际上添加一个citrix gateway)。我们来安装vda(同样会产生一个有定义的Citrix gateway).

再添加一个domain用户，为接下来加入域的vda计算机用，active directory用户和计算机处，找到ctx.srv,computers，添加minlearn@ctx.srv，不能改密码永不过期，默认就是domain users组。

第二台2C4G测试机上安装配置vda,桌面服务器，用客户端连接，运行

这次安装DC本身，如果你是同一个镜像恢复过来的系统，要去windows/system32/sysprep/下执行sysprep.exe。进入系统全新体验oobe确定重新生成一个sid.这部分仅是一个代理占内存不大。但却是桌面应用所在的地方。当然性能也是越高越好，仅上网使用浏览器的话2G吧。

———— ps:Ddc不推荐与vda所在服务器整合，我们的测试也采用分开机制，Delivery controller中的Delivery controller组件是桌面分发的上层服务器，VDA是桌面分发的下层服务器，这个也算服务器，只要receiver才是最终客户端。生产环境中这二者往往装在不同机器上。citrix本身就是面向licensing to 500+ 机器起步的，况且不好直接拿服务端让人家远程进来吧？ 况且如果你把ddc,vda安装到一台机去了。登录StoreFront后，会显示“无法完成您的请求” ————

dns设为前面win主机的内网ip，准备加入域ctx.srv。系统属性计算名中，先给这台机器主机名叫vda,表示这是vda所在服务器，将这个机器由组切换加入域，重启用第一个服务器上的ctx\minlearn用户登录，，整个机器域名会变成vda.ctx.srv，因为这不是域控服务器本地administrator会保留，minlearn用户在本地权限是受限制的,在域用户下，访问设置会出现没有权限或找不到，无法访问本地设备等情形，这要针对找到那个资源或设置条目，执行，通过开始菜单中的windows管理工具找，不要直接开始，设置。改dns，要控制面板，网络和internet，网络连接，解决这种问题需要在域控处一一设置权限。但是我们在本地提升它为管理员也可以，注销切换到本地administrator,在administrator下，本地计算机用户组管理，把ctx/minlearn升为本地administrator组，要先保证dns为域ip哦。

把安装文件拷过来，一样的方式安装VDA,receiver也要安装，因为我们呆会要本地测试一下，启用与服务器的中转连接，组件全不选，功能选一定选上实时音频，防火墙规则都是自动80,1494,2598,8008,1494udp,2598udp,16500-16509udp，，期间会提示远程桌面模式尚未配置（跟其它服务器一样，vda也包装了windows的会话主机当服务器使用），还有119天过期，略过，过程中需要注册到DDC，填ddc.ctx.srv，安装，发送珍断信息不要勾选。后期可以通过programfiles/Xendesktopvdasetup/xendesktopvdasetup.exe修改vda向ddc注册的参数。驱动管理器中发现新装了很多citrix相关的虚拟设备和驱动，主要是io设备。

———— PS：DDC与VDA也有一个高可用模式，允许recevier直接与ddc通讯： 虚拟桌面的代理VDA默认是与DDC之间每5分钟通信一次的啦，所以如果DDC都挂了情况下，VDA和DDC之间的通信就会出现问题。 如果 XenDesktop 站点中的所有 Delivery Controller 均出现故障，可以将 Virtual Delivery Agent (VDA)配置为在高可用性模式下运行，以便用户可以继续访问和使用他们的桌面。 在高可用性模式下，VDA 将接受来自用户的直接 ICA 连接，而不是由控制器代理的连接。这样就可以做到在DDC都挂了情况下依然继续使用虚拟桌面喔。就这是VDA的高可用模式。 不过我没有测试高可用模式。 ————

继续返回第一台机配置。Studio中创建计算机目录，多会话操作系统，未进行计算机管理的计算机，其它服务或技术，添加计算机，计算机目录是安装了vda的计算机的目录,这里填vda.ctx.srv，添加一个交付组，帐号使用ctx\minlearn。

再次访问https://你的公网ip/Citrix/StoreWeb/发现有内容了，点击会引导你下载windows activex控件，或workspace客户端(其实我们只是使用workspace中的view来远程桌面，真正的workspace是一个工作流聚合程序)。我们的目标是使用客户端连接而不仅是浏览器。打开workspace，添加服务器地址，账户为域内minlearn。

成功！效果嘛，暂时看还是可以的。。

———

我后来用2台1h2g的轻量代替上面的测试方案，1台代替方案中的1-dc，studio,storefront(也即，仅保留ddc中的许可),1台代替方案中的2+studio,storefront(把除许可外的其它ddc部件移到这)。把非核心和核心服务器分开，做一个个人最低费用的且可用的vdi 集群（二台1h2g港轻量68元/月，三年是接近2000,其实我理想中的终端价格应该是>500-800元/一年，因为一般手机电脑3年就坏了，3年投入1000-2000是我的心理价位。云主机虽然不是终端还有云带宽，但价格下完全可以拿来类比）可是第二台机应该2h4g才好用。

Studio用来连接ddc,配置完后可以删掉？需要改动时再次连接就是。

也尝试过上面扩展中的2,3做成按需，和尝试过把按需ecs打造成关机0收费无盘网启系统节省费用，但最终证明这个是做不到的。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

aliyun,godaddy云主机单双网卡双IP单双网关，同时上内外网新方案和总结

本文关键字：双网卡双**IP**同时出网，**aliyun**静态路由映射，云主机多网卡设置,**godaddy**单网卡单网关内外网同时出网

周末闲来无事，又折腾起云主机linux变windows了，具体参见我之前的文章《共享在阿里云ecs上安装自定义iso的方法》，在那里其中遇到的改静态路由部分可谓是十分具挑战，阿里云是使双网卡双网关机器能同时上内外网的教学例子典范，而这款godaddy的云主机实在是太可爱了SSD+共享带宽加5刀一月而且支持虚拟国际信用卡，勾起了我小试一下的兴趣，放心我不是打小广告的，所以链接不给了。以下是教程和技术部分（它与aliyun比较属另一种类型即单网卡多IP同时出内外网）：

关于为多网卡多IP段设置内外网同时出网的通用解决方法：

如果提供了双网关双IP，一般外网网卡指定网关，内网网卡就不指定了（因为同一时间只许一个网关出0.0.0.0，也就是互联网，我们一般用外网IP出），这些都可以在网卡的GUI设置界面（就是windows的网卡右击->tcpip属性（->或属性->高级））设定，然后在route print里会生成具体的active route map列表。在这个基础上，我们再加入控制“网内卡”的route -p add附加逻辑部分：使其加入内网网关部分，让其中的流量导向到内网IP。 基本上，单双网卡/单双网关/单双IP/的情形都是这样。 记住这个路子就没错关键是找准技术点比如我摸了一天才明白这是多网卡多IP同时上内外网的不同情形。然后就是解决方法：

aliyun，双网卡内网外网同时上

像aliyun的双实网卡，在文章开头提到的那文章中我提到过。为内网卡添加静态持久路由可以参照文中的末尾图。

godaddy，单网卡内网外网同时上

godaddy网卡比较特殊，godday的一实一虚双网卡（其中一张网卡是用loop back），实际上只有一个物理网卡(virtio网卡)可以操作，另一个可以删除或弃用，属于单网卡，也只有一个网关。

实际上godaddy云主机只有一张真正网卡我也是测试猜的，下面说一下我转到正常思路上的一波三折：

godaddy主机都是有linux脚本的，在机器安装好windows的CD盘中，可以看到有二网卡，正是这个极大误解了我。以为WINDOWS下也需要虚拟一个网卡出来，此时通过web vnc可以看到virtio网卡被设置为dhcp，所有的内卡配置都是对的，但是不能远程桌面，在web vnc中可以ping到其它主机的DNS，选择一些godaddy的内网主机完全可以ping通且浏览其中的网站，可以对于普通其它主机，ping测试可以解出DNS，但接下来4次丢包率100%（有些教科书说法丢包就是不可达，跟那个不可达提示是一样的效果），此时接着linux下设置的思路，我为windows添加microsoft loopback，静态指定其为外网地址，可以上远程桌面。但是在机器内部还是不能上外网的，接下来的工作就是使之上外网。

其linux配置脚本是这样的：

```
GATEWAY=`/bin/grep "gateway" /etc/network/interfaces | awk '{print $2}'`
/sbin/ip route del default
/sbin/ip route replace default via $GATEWAY dev eth0 src 192.169.164.234 proto static metric 1024
```

上面意为把唯一的那个网关，也是默认网关删掉，把它改写为windows下的对应逻辑，一开始我想到windows下并没有route replace，于是百度windows下配置多网卡源地址替代方案，找到dog250的《Windows配置路由时可以指定源地址啦》，但是几经测试不成功可能思路不对并不是人家指导有误。

后来我想到windows命令行netsh指定给ms tcp loopback interface静态外网地址也不行（这个网卡跟我新建的loop网卡ID不一样）。提示网卡不存在。

折腾一天后我终于意识到那个loopback永远不可能出网，因为它出网的数据量永远都是0.

后来知道windows单网卡可以指定多IP，那天试的最多的就是在virtio上静态绑定外内网IP和设置（注意这个顺序先绑）。试了很多次，机器上打开网页终于肯出外网了，像百度之类都可以，才最终意识到，这可能是个“单网卡多IP单网关主机，使其同时出内外网”之类的问题 — 这正是aliyun的相反或相对情形 。再测试，bingo!!成功。以下是结果图和解决方法：

这个贴图中的IP我也就不mark了(10.192打头的是内网IP，192.169打头的是外网IP，网关只有一个10.192.31.254),重点是绑定IP的顺序和针对内网的那二个持久路由。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在阿里云海外windows主机上开启VPN

阿里云ECS比较特殊，因为它是双网卡的，且其DNS走的却是内网卡，这就造成了一般的windows VPN搭建方案在阿里云ecs windows上不能用的情形，比如：

按网上的教程，第一步你允许外网卡出网了。但是此时你只能连上VPN但不能连外网，这是问题之一。因为DNS依赖的内网卡不能出网。

解决方法是，内外网连接全部出网并启用NAT，放行其访问公网就可以。。

第二，由于VPN走的NAT，开启VPN必须要开启NAT。在开启nat后，，VPN的确能连上外网，但仅限系统第一次开启后有效且一会就没了，运气坏的话一次也连不上，表现为这个时候你还会发现服务器在第一次重启15分钟后就会断掉你包括VPN远程桌面，PING的所有连接。

这是因为简单防火墙把你外网卡走NAT的所有对外端口都屏蔽了，，不止VPN的那些，其它都会断掉。所以此时离正常搭建VPN还是有问题的，这就需要NAT下开启相应服务端口。

解决方法是：这个时候在每个连接的NAT里服务简单防火墙那一栏开启相应的服务，包括远程桌面，WEB，PPTP，L2TP等等。。。其中L2TP是VPN用的，其它的，比如你想远程连接就开放3389（比如在开放了VPN的云主机上，远程桌面往往诡异地无法连接，此时你就需要想到是简单防火墙没给这个走NAT的3389放行）

所以配置路由和网关时应该这样进行：

只有在二张中的所有设置完成之后，你才能正常连上VPN，linux一键脚本修改原理如上。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

openvpn - the only access server we need,通用网络环境和访问控制虚拟器

本文关键字：**openvpn** 网络环境模拟，**openvpn** 访问控制，**openvpn** 虚拟局域网，**openvpn vps**

openvpn就是个人打洞器，privte tunnel（在你的机器和openvpn服务器之间做一个点对点 – 连接层，以促成在不同网络之间做一个通道 – 网络层），基本的用途是可用于访问不同网络，比如国际互联网，这点上，它也可用于一些通用环境下的访问点节点互通，甚至，做到APP对APP的通用网络连通控制（任意网络环境虚拟器将之做成节点，任意逻辑的访问控制器，比如在3389之前做一个nat，以使未登记过的IP不能访问3389）。

它甚至还可以维护一个节点管理器实现“多p2p”，将用户整合到你的专有虚拟网中，做自己的客户群入口把非必要用户掩映在一个入口之外（这绝对不像会员系统这属于在访问层就把不要的人给隔离了有绝对的保密性，甚至在公网环境下也不致暴露或可被找寻），比如，你可以把复杂网络环境下的多机集线进来做一个虚拟局域网（nat环境下-比如有路由器，此时出网需要在路由器作映射—此时称节点被路由过）。当然它与其它软件配合可以做流控之类的东西。像官方的一键包。

其实，很多网络操作系统如windows server,linux都有上述支持，只不过openvpn在单一个应用级别集成了这些，以在正确的层面做正确的事情的方式，其实openvpn是工作在实体路由器和实体网络环境之后的，然后它才能在它虚拟路由器级别干它的vpn事情。它并不对这些产生干扰。openvpn利用的是cs概念，开启s的节点有管理节点的功能，要达到另外一机参与到s机的局域网（更专业讲是VPN环境）环境，往往需要对作为服务节点的路由器设置有控制。 当然这是属于“加速”的范畴，与二机直接点对点如网线直连成VPN网络是不一样的，不过后者往往不可得。所以“加速”往往才是在正确的层面做正确的事情。和直观正确的策略。 即这种方案较传统p2p并不是过度虚拟出来的。是主流的。符合你为了学习找实现范例和过程。openvpn即是这么一个足够好的被研究对象。 总之，openvpn实现了一个“access server”，或称“connect server”，这个可以集成到portalserver.对应于“webserver as page guiserver”。做VPN等等，如果需要VPN翻墙则一个国外带宽的vps承载才有意义。

二机开启点对点局域网

开启需要打开路由器对openvpn的1194端口做局域网映射。基于上它是利用nat的原理 : 对路由器任何一个或一段协议端口的访问（从WAN口进来的访问），都可以重定位到局域网内某一台指定的网络服务器。开启过后，任意二机就有了虚拟局域网环境，在其中你可以共享浏览局域网文件，甚至可以玩对战游戏如war3。将越来越多的C端节点加入到S端节点你就有了多节点局域网环境。

搜索网上openvpn 虚拟局域网有大把教程，就不多讲了。如果你指定你的VPS只有某个局域网才能连接，那么指定openvpn建立的局域网的C端IP即可，这样就达到了IP级的访问控制。当然你需要流控和节点监控之类的逻辑需要借助其它软件打造，比如提取官方包中那个。

配置进阶

以上涉及到的只是表象，上面提到的局域网更适合被称为内部网，因为VPN的隐喻即是建立各种拓扑连通前提下的网络节点访问打通。而，更复杂的情形和其它针对技术在各种情景下都会涉及到，比如静态路由修复，包转发技术啊，这基本是高级网络测试了。

这一切，都是在理清背后那个拓扑结构的前提下，利用虚拟化和网络七层各层次的重定向技术来完成的，可能是软件级的，可能是路由器硬件级的。。而真相只有一个：对背后那个目标网络结构连通的认知，对网络基础知识的认知。

好了。更多的知识需要在其它应用性的贴子中涉及到的。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在colinux上装openvpn access server

本文关键字：**colinux modprobe tun,在colinux上编译colinux编译,openvpn access server cant open /dev/net/tun**

在前面的文章中我们分别介绍过colinux和openvpn(它是一个足于被称为可搭建access point server的东西)，在《发布mineportal2》中我们将mailinabox做到了colinux，并称为mineportalbox，这里我们准备把openvpn access server也做进mineportalbox，使之成为一个能构建集私密个人网络访问点，邮件消息，文件存储，静态网站托管空间的一体便易环境。

首先要解决的问题是重新编译colinux，为什么呢？因为官方发布的colinux0.7.9.exe中，安装到硬盘中的vmlinux-modules.tar.gz展开到colinux->lib->modules下没有modules.dep.bin，这造成在colinux中不能执行modprobe目录，而这会导致安装完openvpnas后出现"cant open /dev/net/tun"错误（通过943访问web管理界面会发现openvpn服务没启动，点启动会出现这个错误）

1,在colinux中编译colinux

我选择的是colinux2.6.33.7+ubuntu14.04 32位镜像（带gcc4.8）环境，下载源码后一般需要执行./configure,make download,make cross,make kernel这几步。（当然你可以make all不过那样不便于这里说明需要修改的地方），为了清晰起见，我按上面几步说明源码中要修改的地方，以使编译能顺利通过：

去掉configure中对depmod的检查或注释掉：# check_depmod "depmod"

bin/make-common.sh中，MINGW_URL修改为jaist的下载地址：MINGW_URL=http://jaist.dl.sourceforge.net/sourceforge/mingw

bin/make-cross.sh中，configure_binutils()，--disable-nls 后加--disable-nls --disable-werror

patch文件夹中，加2个补丁(在附件中下载，它们是ptrace-2.6.33.diff,vdso-2.6.33.diff,)，然后写在series-2.6.33.7中带入源码应用这二个更改

我在bin/make-cross.sh中还去掉了编译完binutils和gcc4.1.2将其临时文件夹删除的二条clean语句

好了，开始编译，如果你的colinux不是ubuntu14.04，也许上面的更改会有一些小的不同(比如还需要apt-get install zip textinfo flex etc..)。直到make kernel得到vmlinux文件和vmlinux-modules.tar.gz，删除现有colinux中的lib/modules/2.6.33.7-co-0.7.9目录，退出当前colinux，用新的2个文件覆盖colinux文件夹中旧文件，重新启动colinux会自动将新的vmlinux-modules.tar.gz注入lib/modules，在其中你会发现已有modules.dep.bin，执行modprobe tun，（tun是linux2.6x自带的，也是openvpnas需要的），进一步执行lsmod | grep tun，输出一行带tun的条目，说明tun运行成功。

2，安装openvpn

wget下载openvpn access server二进制包，dpkg -i 包名进行安装，passwd openvpn设定一个密码，因为我使用的阿里云专有网络IP主机+colinux slirp网络与host相连，会给出https://10.0.2.15:493/admin/这样的管理地址（事实上，整个openvpn这个时候是属于内部体系的，我们在这步先保证openvpn本身服务能运行，最终外部能不能连上一条逻辑上所有的东西以后再谈）。事先将493和1194转发/透露出来，访问https://主机外网ip:493/admin/管理，进去发现openvpn服务是OFF状态，启动依然显示“cant open /dev/net/tun”，可是在第一步的最后我们不是已经modprobe tun成功了吗？为什么Openvpn 加载了 tun 模块，仍然提示Cannot open TUN/TAP dev /dev/net/tun呢？

放心，这里只是一个小问题，只是因为ubuntu没有自动创建设备点文件而已，手动在colinux中执行：

```
mkdir /dev/net
mknod /dev/net/tun c 10 200
```

即可，现在试试开启openvpn服务，成功显示 ON.

3，远程DNS解析使得VPN可用

这个时候，通过https://公网IP:943/下载到的openvpn-connect-2.1.3.111_signed.exe是可以登录进VPN的，但是不能打开网站。这是由于openvpnas的安装脚本仅识别简单公网方式，不能识别复杂的aliyun专有网络和colinux slirp私有IP环境下内部地址的情况，使其最终变得可访问，这应是另外一个类似“将openvpn access server装在virtualbox或家庭内部nat设备”之类的问题，下面我们来尝试解决：

首先，进入https://公网IP:943/admin/管理界面，把server_network_settings页hostname由10.0.2.x之类的地址改为你的公网IP

然后进入vpn_settings页，由于默认安装方式下，三个主要问题（Should VPN clients have access to private subnets (non-public networks on the server side)？ Should client Internet traffic be routed through the VPN? Should clients be allowed to access network services on the VPN gateway IP address?）都是脚本选择yes的，这造成受DNS污染和劫持的客户端DNS解析都是不能工作的，我们需要接下来在DNS Settings单选区选择第三页指定DNS，主要DNS设为8.8.8.8，次要设为114.114.114.114，保存，update running server!!

重连客户端打开google等，发现可以打开了！

进行在这里，所有的工作只是保证openvpn服务正常运行，基本作为一个理论上可工作的VPN运行，而实际上，反VPN技术很强大，DNS只要从外入内就会有污染，这应该是更高级的VPN课题了，所以这里不讲，下回题目为《利用VPN打造游戏gamelobby client -- turn lan game into internet》。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

v2ray-利用看网站原理模拟链路达成VPN

本文关键字：**v2ray https tls, v2ray caddy**

在我们以前的文章中，我们一直谈到vpn相关课题,如《在阿里云海外windows主机上开启VPN》，《openvpn - the only access server we need,通用网络环境和访问控制虚拟器》等，在谈群晖的公网访问方式的时候我们也谈过到穿透盒子，花生壳蒲公英等，frp,cow等课题，这些与portalbox storage相对的accessbox vpn的课题其实也是属于vpn —— 一花一世界，vpn技术背后是一个代表access server的通用技术范畴，不只是用来翻墙的，它是用来打破现有复杂网络环境的限制，在这上面复合建造一层通用内网环境，来作本不可能的节点相通的节点之间互访或加速的。

以实现互访为基础目的网络活动，在各个层面都可能会达到目的，不必涉及到VPN。拿翻墙来说，远程桌面（remote app, vnc）,流量中转导向（反向代理,反向代理下的内网穿透），建立复合链路，都可以达到效果。不过真正的VPN和VPN下的翻墙，偏向指的是建立复合链路，建立网络上的网络：要从链路层面上开始去打造一个模拟网络环境去实现最终目的。这些在《openvpn - the only access server we need,通用网络环境和访问控制虚拟器》的开头处，我们说的很多。

VPN光提供好的通达性也不行，也要提供绝对的安全。抗封也是安全的一种，这些非安全因素可能来自恶意man in the middle，甚至也可能来自ISP，来自政府，其封VPN的技术手段多样，可能在VPN发生时的各个网络层面以不同技术存在，如dns过滤,封来路，封源头，封IP封端口，流量刺探，特征识别，协议分析。

那么，是否真的存在一种永远抗封、或尽可能99.99%抗封的VPN呢？使访问点之下成为绝对的公网中的暗网。甚至让来自提供网络服务的正当中间层的管理也变得无视。让安全的VPN成为任何云的绝对标配,走进accesspoint server的范畴。—— 我觉得，只要双方的网络没有去掉真正的物理层的互通，即还在同一个internet，就存在绝对的可能，因为抗封和反封本来就是一对此消彼长的矛盾，一种方式就是设置尽可能多的中间层和转发，中间层设置得越多，被破解和被识别的可能性就越低（当然效率就越低）。这些层面中，使用尽可能多的正常层面和服务（ISP不可能或不致于为了防VPN去封掉和干涉的那些业务），把风险高的层面层面降到只有1个甚至放到本地（比如shadowsocks就是这样，它把DNS放到本地不出网），需要越网的部分被封风险低的正常层面放到后面且高强度加密。

好了不重复了。我们接下来的要介绍的v2ray，是ss的增强，自带混淆和协议叠加，有流量中转导向。也可用于翻墙。它甚至可以与nginx或caddy等反向代理+流量中转+静态页面服务一起。产生更多有趣的功能组合。

V2ray：最基本的http混淆路径

v2ray的效果，它有点像利用nginx反向代理做一个谷歌镜像，都是采用了一定的网站原理达到访问被代理网站的手段，不过本质却完全不一样，nginx反代google是ngx转发http的基础上，仅代理一个网站的流量。这种方式有很大局限,所有的行为和流量在ISP眼中都是透明的，使用方式上也是用浏览器访问一个特定网站。而v2ray属于走链路的真正VPN。链接建立后使用方式不限应用类型和方式，是上述所讲协议层的层层叠加，xxx over yyy(yyy作为传输协议)，你可以把它想象成一堆流量处理工具：

以我们要讲的最基本的http混淆路径来说，它先模拟正常的https流量（这对于翻墙来说就是抗封1），再在这种流量上tls加密（对于翻墙这就是抗封2），这二者组成了流量混淆。这是ISP能看到的最终的那部分流量。是越网的那部分流量。也是正常的应用流量。

在远端不需要越网的流量则是正常的caddy，它将VPS上的ws流量作转发到上述的https，在本地不需要越网的流量则是正常的DNS。

风险最大的部分，是客户端连vps时采用的vmess协议。只有一小部分。这被封就没有办法了。所以CDN在这里发挥作用。如果再加以cdn，isp端往往都看不到vps的真实IP。此时cdn成为防封的又一强大叠加层。

如何搭建v2ray：with caddy or with/without cdn

这里讲解下基础的搭建方式，首先把域名A映射到VPS的IP。可以是二级域名。如果要使用CDN，去cloudflare申请个号，dns服务商要设为cloudflare中的相应条目，选择免费服务，在cloudflare中添加网站，即上面的二级域名的主域名，然后打开对应添加的网站的ssl/tls栏，设置为Your SSL/TLS encryption mode is Full（好像这个是自动设置好的？），然后点开网站的DNS页，进去添加A解析该二级域名，对应于这个二级域名的地方先把云朵点灰（不使用proxy，使用dns only）。跑完下面的脚本后再打开，否则脚本get不到ip。

再安装v2ray，source <(curl -sL https://git.io/fNgqx) –zh，这底下是一个multi-v2ray的自动化安装脚本。https://github.com/Jrohy/multi-v2ray,v3.7.3脚本中pip3挂掉需自己装，为什么选择这个呢，因为这是一个中性，不太复杂，也没有简单到什么都没有的脚本。脚本跑完后提示你可以使用v2ray命令。此时，它是按默认配置的—即第一条相对的inbound outbound配置项。

第三安装caddy，wget https://git.io/vra5C -O - -o /dev/null|sudo bash，这也是一个中性复杂度的脚本，然后caddy install安装caddy，中间会提示你启不启用php，按你需要选择。然后叫你填入默认域名-即第一条网站的域名，即前面映射的域名，填入，caddy会自带https，可以自动申请一年的免费证书，这时提示email时提供一个就好了,它就会帮你申请好。caddy start就能启动caddy了。

这二个脚本都基本你不需要再做什么，修改默认配置或添加配置即可，我们下面直接修改默认配置。

v2ray的配置：执行v2ray,5 global setting设为中文，重新执行v2ray,3更改设置，修改email为caddy中申请SSL提供那个，修改port为你喜欢的数定,修改传输方式，选择web socket,输入想伪装的域名，即caddy安装时的默认网站域名，重新执行v2ray，3更改配置，改变TLS，开启，选择生成证书，提示填入域名，这基本是caddy自动化申请ssl的v2ray版本，v2ray和caddy都需要来一次。要填入的域名就是那个映射的二级域名。如果没有

事先做好，这里会提示出错，输入的域名与本机IP不符。重新执行v2ray，4查看配置，我们可以有一条ws的/path/。一直都会出现的vmess协议和客户端配置都是当前v2ray配置用于客户端配置的那些当前参数，v2ray与ss一样，都是同一份客服。

现在来配置caddy。cd /etc,打开Caddyfile,vi Caddyfile,fastcgi下添加：

```
proxy /你看到的path/ 127.0.0.1:你选择的端口 {

                websocket

                header_upstream -Origin

        }
```

这样caddy和v2ray就对接起来了。重启生效。v2ray,caddy都会开机自启。

如果使用CDN，此时进CF，把云朵点成彩色proxy模式。

现在打开二级域名。会出现一个caddy的欢迎页(caddy默认占据80)，打开二级域名:你选择的端口,出现Client sent an HTTP request to an HTTPS server.显示这个才对。如果caddy的logs中出现v2ray.com/core/transport/internet/websocket: failed to serve http for WebSocket > accept tcp [::]:端口，就把上面过程重新跑一次。如果测试通过就可以开始下面的配置客户端了。

一个静态网站有很多流量即只有少量IP这个特征较明显，你可以这里设置一个简单的静态网站。以防人工鉴别，你可以放置一个下载超大文件的链接。文件放在这个服务器上。

v2ray连接方法:手动配置客户端

好了，下面我们讲解手动配置客户端参数。之所以选择手动，不使用v2mess和配置文件是为了学习目的。拿osx下的v2rayx来说。

点configure,增加一个服务器，adress填二级域名，port填你选择的端口，userid就是v2ray执行时显示的uuid,可以在v2ray上添加多个uuid，每一项uuid对应一个用户。alterid同理。security:auto, network:ws, transport点开，websocket页中，path填上面的/你选择的path/，headers，填{ "Host" : “你的二级域名“ }，tls页选择use tls,servername填那个二级域名。单击OK完成。

在v2rayx主菜单中选择刚配好的server条目，load core,模式选global模式。你可以稍后用APP代理设置。这里global可以使所有的APP都走v2ray的通道。

移动端可用Xcode连接手机或爱思助手连手机，安装shadowrocket ipa，正式版的App Store shadowrocket是下载不到实体文件的。你可以把ipa想象成开发版的安装包。安卓是v2rayng

---

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

为什么学js，a way to illustrate webfront dev and debug essentials in raw js/api，及用js获取azure AAD refresh token for onedrive

本文关键字:**why xxx lang series**，使用**pure js**获取微软**azure AAD refresh tokenfor onedrive**，安装**chrome**扩展禁用浏览器跨域保护

在前面我们讲到了《我为什么选择rust》，《why elmlang:最简最安全的full ola stack的终身webappdev语言选型》，这些都是标题中已经讲明的why xxx lang series，但其实，在本书所有提到语言选型，其它选型的相关章节中，都是某种意义上的"why xxx series vs yyy,zzz,blaaa..."，我们还有一些专门针对语言合理性和简单性对比的文章，如《lua/js/py复杂度分析，及terralang:一种最容易和最小的"双核"应用开发语言》和《一种最小(限制规模)语言kernel配合极简（无语法）扩展系统的开发》，在《why elmlang:最简最安全的full ola stack的终身webappdev语言选型》中我们进一步介绍了elm，并提到其与一些语言的简单合性理比较。甚至还有py,perl,ruby,php等。

本文要谈到的js，它是elmlang的直接近祖，在简单合理性方面，如果说py,perl,php,ruby这些通用无限领域语言是一派，那么elm,js,shell局限化一体环境语言就是醒目的另一派：

> js的最大优势是简单合理，JS的简单是是因为它绑定了应用环境和调试环境。与前端这种开发领域形成非常紧密的DS-L对应,整个js生态跟bash与shellprogramming一样，是环境与语言绑定的，所以调试方便。（它本身属于让js处在一个单进单线环境中的可调用环境,language in final app as lanuage runtime结构，类似游戏编辑器和编辑器脚本的组合,vs通用语言。因为它们绑定一个最终运行环境，可以避免因提供的类型和内置数据，都较泛化，调试环境依赖不断搭建，所带来的学习成本。
>
> 合理是，它基于c和函数式。这基本是现代编程流派的二大代表，每样都只取一点点又恰当好处，js非常精简，它混合一部分C和一部分函数式。使得不依赖OOP中过渡泛滥的观察者设计模式，用函数就实现了效果接近MVC。事件相当于一个主题(Subject)，而所有注册到这个事件上的处理函数相当于观察者(Observer)。要知道js函数比类更适合当XXX者XXX对象。nodejs并非通用语言没有多线程和胶水能力那些,但它的IO很快，多线程带来的效果可以不计，胶水能力需求倒显得其次，由于js本身就支持这种异步IO，所以库级的观察者订阅者模式，只是薄薄的一层就能实现。甚至不到10行。异步IO天然就是网络服务交互处理DSL的。它本身也是一种MVC绑定语言。框架绑定语言。专门用来写WEB，它对WEB的那些机制几乎是一比一原生的。虽然js及其生态，包生态有一些缺点，但这是语言发展过程中不能避免的。

其实，类似观点在《一种开发发布合一，语言问题合一的shell programming式应用开发设想》《elmlang：一种编码和可视化调试支持内置的语言系统》，《编程实践选型通史：平坦无架构APP开发支持与充分batteryincluded的微实践设施》,都涉及过）解决编程难度的真正途径只有一个，就是装配一个免IDE的调试环境，让程序与环境一起绑定和出现。让源码级的业务逻辑所指的环境天然出现。就像plan9一样，成为任何语言APP的寄体，OS作为云原生环境而不是k8s。(这也是本书选型shell和web开发作为实践部分的目的所在)

我们先从一个例子开始，这个例子是**js**的原生前端应用领域的典型,原生**web services api**交互，第一步

在《利用fodi给onemanager前后端分离(1)：将fodi py后端安装在腾讯免费cloudbase》的后面部分我们讲到将onemanager获得refresh token的过程用几个html搭建，使之脱离onemanager php后端的工作，在那里，我们的工作其实并没有达成，获得的refreshtoken要从url中复制，且放到onemanager的disk配置中并不生效，

这是因为我们只做了第一步：获得一个随机refresh token的过程，其它的都没有做，其实这里有二个过程，有二个html，除了前面的html和生成refreshtoken的工作，这个随机refresh token还必须要再次向azure交互一次，这里来完成它：

先给出第一步的html和逻辑解释：

```
<html><meta charset=utf-8><body><h1>get refresh token for onedrive (in static)</h1>
<body><p>allow javascript,and prepare the app reg before starting below:


<div>
    <br>
    <label><input type="radio" name="Drive_ver" value="CN" checked onclick="document.getElementById('morecustom').style.displa
y='';document.getElementById('inputshareurl').style.display='none';">CN: 世纪互联版</label><br>
    <label><input type="radio" name="Drive_ver" value="MS" onclick="document.getElementById('morecustom').style.display='';doc
ument.getElementById('inputshareurl').style.display='none';">MS: 国际版（商业版与个人版）</label><br>
    <label><input type="radio" name="Drive_ver" value="shareurl" onclick="document.getElementById('inputshareurl').style.displ
ay='';document.getElementById('morecustom').style.display='none';">ShareUrl: 共享链接</label><br>

    <br>
    <div id="morecustom">
        <label><input type="checkbox" name="Drive_custom" checked onclick="document.getElementById('secret').style.display=(th
is.checked?'':'none');">自己申请应用ID与机密，不用OneManager默认的</label><br>
        <div id="secret" style="margin:10px 35px">
            <a href="https://portal.azure.cn/#blade/Microsoft_AAD_IAM/ActiveDirectoryMenuBlade/Overview" target="_blank">申请应
用ID与机密</a><br>
            client_id:<input type="text" name="client_id" id="a1"><br>
            client_secret:<input type="text" name="client_secret" id="a2"><br>
```

```
        </div>
        <label><input type="checkbox" name="usesharepoint" onclick="document.getElementById('sharepoint').style.display=(this.
checked?'':'none');">使用Sharepoint网站的空间，不使用Onedrive</label><br>
        <div id="sharepoint" style="display:none;margin:10px 35px">
            登录office.com，点击Sharepoint，创建一个网站（或使用原有网站），然后将它的站点地址填在下方<br>
            <input type="text" name="sharepointSiteAddress" style="width:100%" placeholder="https://xxxxx.sharepoint.com/sites
(teams)/{name}"><br>
        </div>
    </div>
    <div id="inputshareurl" style="display:none;margin:10px 35px">
        对一个Onedrive文件夹共享，允许所有人编辑，然后将共享链接填在下方
        <input type="text" name="shareurl" style="width:100%" placeholder="https://xxxx.sharepoint.com/:f:/g/personal/xxxxxxxx
/mmmmmmmmm?e=XXXX"><br>
    </div>

    <br>
    <input type="submit" id="btn" value="点击获得一个refresh token">

</div>


<script>
......
</script>


</p></body></html>
```

在这个典型简化div-script结构的这样一个html页面中，我们看到了js内嵌于browser，用语言控制dom元素,形成前端效果的原始操作能力，

和div内的子元素，子div元素都存在上下层次关系，document.getElementByxxx('xxxx')就是这复杂关系下统一的选择符，js分散在可点击元素的onclick过程中控制这些效果。，它们也可写成对应的块，对于有ID的元素，它用document.getElementById('xxxx')，对于只有tag或name的，document.getElementByName('xxxx')返回的是数组，因为ID是页面上唯一的，而name是一种归类标识，一个页面上可以有好几个同样name的元素存在，（document.getElementByxxx('xxxx')这种完写法会比较累，当然jquery这种库会类似perl和bash一样定样一些专用符号简化这些逻辑，这毕竟是外来库，不讲。）

讲完了前端效果控制，下面是主要块中，干实事的逻辑了：

```
    //function notnull(t)
    //{
    //    if (t.Drive_ver.value=='shareurl') {
    //        if (t.shareurl.value=='') {
    //            alert('shareurl');
    //            return false;
    //        }
    //    } else {
    //        if (t.Drive_custom.checked==true) {
    //            if (t.client_secret.value==''||t.client_id.value=='') {
    //                alert('client_id & client_secret');
    //                return false;
    //            }
    //        }
    //        if (t.usesharepoint.checked==true) {
    //            if (t.sharepointSiteAddress.value=='') {
    //                alert(''.'InputSharepointSiteAddress'.'');
    //                return false;
    //            }
    //        }
    //    }
    //    document.cookie='disktag='+t.disktag_add.value+'; path=/';
    //      return true;
    //}


    clientid=document.getElementById('a1').value;
    clientsecret=document.getElementById('a2').value;
    document.cookie="clientsecret="+clientsecret;
    document.cookie="clientid="+clientid;

    url=window.location.href;
    if (url.substr(-1)!="/") url+="/";
    url="https://login.partner.microsoftonline.cn/common/oauth2/v2.0/authorize?scope=https%3A%2F%2Fmicrosoftgraph.chinacloudap
i.cn%2FFiles.ReadWrite.All+offline_access&response_type=code&client_id="+clientid+"&redirect_uri=http://localhost:8000/SCFOned
rive.github.io/CN/&state="+encodeURIComponent(url);


    document.getElementById('btn').onclick = function (e) { window.location=url;}
```

首先，（我们这里以互联版作参照）你应该在https://portal.azure.cn/#blade/Microsoft_AAD_IAM/ActiveDirectoryMenuBlade/Overview中按《利用fodi给onemanager前后端分离(1)：将fodi py后端安装在腾讯免费cloudbase》说的注册好应用，回调地址中填一条http://localhost:8000/SCFOnedrive.github.io/CN/，azure规定重定向 URI 必须以方案 https 开头。 但这类localhost 重定向 URI 例外，因为测试时我们都是本地测试的，因此azure允许这类例外存在。//注释的那一大段是检测你输入有效性的，图省事没有用起来，

这里的逻辑主要是获得clientid输入框a1,clientsecret输入框a2，存入cookie供接下来要转向到的那个长长的url对应的页面中（所以你的浏览器须允许js允许cookie）使用（跨页面传递变量要借助cookie而不再总是url分析,其中clientid在本页中直接放到url中应用了一次），然后这个url是点击button直接转向过去的window.location=url，表明这个长长的url是一次面向azure的get提交，跟直接在浏览器中粘贴地址访问得到什么样的返回，效果一致，redirect_uri=http://localhost:8000/SCFOnedrive.github.io/CN/，表明https://login.partner.microsoftonline.cn/common/oauth2/v2.0/authorize要向你设置的这个回调页面传递结果,它将转向到:http://localhost:8000/SCFOnedrive.github.io/CN/?code=xxxxxx&state=http%3a%2f%2flocalhost%3a8000%2fSCFOnedrive.github.io%2f&session_state=xxxxx#这样一个页面，传递的结果就是那个长长的生成的code，这是随机refresh token。开头说过，它并没有生效。需要在接下来的步骤中再跟azure交互一次。

> client id,access token，refresh token,这些其实都是在主帐号下分出子帐号，属于客户端调用专用验证机制，这样就不用涉及到主帐号。这是一次性的，用access secret验证完refreshtoken后，call back page其实用不着access secret，主程序也用不到callback page，以后的每次调用也不必维持这个callbackpage。甚至refresh token可以用别人的client id,access token得到。为什么本地要维护一个转向（至APP内部）处理页面呢，因为这是web services之间的交互逻辑。认证。多个不同来源的只能通过callback，转向这向逻辑来相互传递数据。

我们先从一个例子开始，这个例子是js的原生前端应用领域的典型,原生web services api交互，第二步

这是第二个页面，回调页面,/CN/index.html

```
<html><meta charset=utf-8><h1>azure callback page</h1><p>


<div>
    组装一条含随机refreshtoken的注册链接形式：<br><br>
    <a id="t1">No link here!</a><br>

    <br>
    调用注册链接并分析返回结果，请自行判断并复制！<br><br>
    <label id='t2' style="width: 95%">no result here!</label><br>
</div>

<script>
......
</script>

</p></body></html>
```

页面效果已尽量简化，重点是在中：

```
    function getCookie(name) {
        var arr;
        var reg = new RegExp("(^| )" + name + "=([^;]*)(;|$)");
        if (arr = document.cookie.match(reg)){return arr[2];}
        else return null;
    }


    function getContent(url2,url22) {
        var xhr = new XMLHttpRequest()
        // post请求方式，接口后面不能追加参数
        xhr.open('post', url2)
        // 如果使用post请求方式 而且参数是以key=value这种形式提交的
        // 那么需要设置请求头的类型

        xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded')
        //xhr.setRequestHeader('Origin', 'http://localhost:8000')

        xhr.send(url22)
        xhr.onreadystatechange = function () {
            // 数据全部返回的判断
            if (xhr.readyState == 4) {
                document.getElementById('t2').innerHTML=xhr.responseText
            }
        }
    }
```

```
    var q = new Array();
    var query = window.location.search.substring(1);
    var vars = query.split("&");
    for (var i=0;i<vars.length;i++) {
        var pair = vars[i].split("=");
    q[pair[0]]=pair[1];
    }

    var url = q['state'];
    var code = q['code'];
    if (!!url && !!code) {
        url = decodeURIComponent(url);
    if (url.substr(-1)!="/") url+="/";

        var url2 = "https://login.partner.microsoftonline.cn/common/oauth2/v2.0/"+"token";
        var url22 = "client_id="+getCookie("clientid")+"&scope=https://microsoftgraph.chinacloudapi.cn/Files.ReadWrite.All offline_access"+"&client_secret="+getCookie("clientsecret")+"&grant_type=authorization_code&requested_token_use=on_behalf_of&redirect_uri="+"http://localhost:8000/SCFOnedrive.github.io/CN/"+"&code="+code;
    document.getElementById('t1').innerText = url2+"?"+url22;


    getContent(url2,url22);
        //if ($tmp['stat']==200) $ret = json_decode($tmp['body'], true);
        //document.getElementById('t2').innerHTML=result;


    } else {
        var str='Error! 有误！';
        if (!url) str+='No url! url参数为空！';
        if (!code) str+='No code from MS! 微软code为空！';
        document.getElementById('t1').innerHTML=str+'<br>'+decodeURIComponent(query);
        alert(str);
    }
```

这里的流程正如页面效果中提到，先组装一条含随机refreshtoken的注册链接形式，然后调用注册链接并分析返回结果，请自行判断并复制，这一切都在pure js和单个页面中用ajax xhr完成，而类似onemanager这样的后端安装界面，用的是封装了服务器能力的curl_request，用的是post www-form-data，再次将得到的结果转到其它页面处理的，比如保存这个refresh token，而我们直接展示输出返回结果让用户判断是否成功决定复不复制了事。

在整个逻辑中，先定义二个函数，一个是取cookie，由于cookie只能先取得cookie数组，所以只能从数组中按名字正则方式取值，模拟关联数组，一个是取返回结果，这里的url和取得url的返回结果的方式和第一步有较大不同，因为这个callback和azure交互的方式是post，而且要以非url参数形式提交数据，普通情况下我们要准备一个form收集这些数据。且作为前端的js传递上并没有xhr这样的东西，无法做到curl这样的服务器函数效果。只不过我们现在的浏览器幸运地支持它，可以模拟curl。

getContent(url2,url22)就是上面提到的效果对应的函数，准备url2,url22请求接收方,请求发起方双方数据，url2即页面与azure交互的后端：https://login.partner.microsoftonline.cn/common/oauth2/v2.0/token(注意这个与前面一步不同)，要求的参数也不同（url22作为data发送，而不是url参数），写到t1中(由url2+"?"+"url22"形成的这个url不能直接在浏览器中访问，AADSTS900561: The endpoint only accepts POST, OPTIONS requests. Received a GET request)，这个交互url在函数中用POST提交方法，setheader提交头部信息进行，这样处理端就准备好了，接下来逻辑主流程会收集这个callback页面的发起方url，分析出发起请求参数用的code，即那个随机refresk token，在if else流程中的if部分，会调用getContent(url2,url22)会将异步调用得到的结果写进document.getElementById('t2').innerHTML代替初值（如果输出一个正常的json就对了那么refresh token就生效了，如果其它就是不成功的，可能是多次调用同一clientid,client sercet发生redeem了，返回第一页重来）。else退出逻辑。

> 这里的调试手段，是借助chrome的f12来调试。也可以console.log()输出变量在chrome中查看。也可以alert()，也可以用curl -X或postman,postman是一个类似cloudflare那个界面的东西(话说这个东西做在浏览器作调试器多好)。https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-client-creds-grant-flow，点"run postman"会自动将测试case导入到postman，我们要用的是“Token Request - Auth Code”，在postman中，要去body,xwww-form-urlencode中把参数弄进去，而不是最开头那个中Params，否则提示"AADSTS900144: The request body must contain the following parameter: 'grant_type'.，如果是curl -X方式，curl -X POST -H "Content-Type: application/x-www-form-urlencoded" -d 'xxx&scope=xxx&client_secret=xxxx&grant_type=xxxx' 'xxxx'
>
> 这里特别注意一个问题，curl和postman和browser（chrome）处理web service 交互的逻辑有点差别，chrome对跨域有限制，在本地利用localhost这样的地址测试时，会发生跨域错误。 //xhr.setRequestHeader('Origin', 'http://localhost:8000') 无效，请参照《3 Ways to Fix the CORS Error — and How the Access-Control-Allow-Origin Header Works》类似的文章，使用改变浏览器行为的笨办法，The quickest fix you can make is to install the moesif CORS extension，安装好了把它由OFF调到ON，从头开始。

在后端,js的开发，环境结合性和可调试性也是一样的。

利用大容量可编程网盘**onedrive**配合公有云做你的**nas**及做站

本文关键字：**onedrive**打造你的网站图床大文件外链床，**sync as network**，网盘附件床,**onemanager http**解发

在《免租用云主机将mineportal2做成nas，是个人件也可服务于网站系统》中我们提到将nas和个人网站合一的思路，，在《mineportal – 一个开箱即用的wordpress+owncloud作为存储后端》我们提到用网盘做网站图床甚至all webapp backend storage的思路，在《统一的分布式数据库和文件系统mongodb，及其用于解决aliyun上做站的存储成本方案》我们讲到过在阿里云上省事做站的成本考虑。

毕竟，我们应用软件的方式就是在一台机器上装个app，这种思路已经深入人心了，所以我们希望做站也一样（像安装应用一样简单）。碰巧网站应用的附件床是必不可少的，一个用户体验好的网站需要大流量和高速度，甚至大存储。这跟nas处理存储的方式和环境要求很相似(现在网盘能达到百M局域网速能代替nas)，所以我们希望它与nas和同步管理连接起来。（群晖有一个wordpress plugin）。这样服务端的网站数据也像手机端的照片，PC端的物料文件一样，参与到nas中心文件库的备份与维护工作中去。这样就大为省事省心了。

> PS：我们知道，以ecs为基础的建站成本很高。需要强大的带宽，这是个人承担不了的。于是运营商们提出了无服务器建站serveless，，这基本就是以前的弹性扩展，专门化的“网站云，云网站”。即利用oss做静态资源存储和展示，甚至包括cms中的前端静态网站资源，再利用云函数做应用托管，来弥补静态网站动态性不够的问题，所谓函数，也就是相当于一般网站/移动应用的plugin（这些相当wordpress plugin,,etc,,是函数集），它们帮你装好了各种语言backend，也提供了一个类似容器的主机环境，你免去了搭建服务器的工作，这应该也是我storage backend webapp的一种实现，最后利用cdn做分发流量租赁。你租用它们的oss(专门做站的网盘还有小鸟云这些??没用过),cdn,和函数托管能力，（业务逻辑）由你提供，当然你也可以安装他们提供的函数模块。，

但即使这样，其实综合成本上，一套下来也不会低。只是稍为省事。但是如果有了一个函数容器或ecs托管环境再搭配大容量网盘，情况或许就不一样了，网盘的线路是第三方的。不走服务器流量，网盘一般都是cdn加速的且网盘一切都是买断的。相当于为网站服务器准备了一个超大图床，甚至附件线路，sync as network线路,这实际上就是上面提到的更强大的oss替代。网盘如果能作为附件床再好不过，甚至可以当成个人nas用。

我们在前面提到在云上装黑果。本来我们的设想是利用云上osx能利用上icloud的能力，自建icloud且提供一个能运行icloud的托管服务的环境。Icloud是一个非常好的文件异备服务(多端同步防冲突算法非常鲁棒)，icloud是被设计供icloud产品系内部使用的，直接在icloudrive和国内环境下使用icloud速度很快托管在云贵，是一个非常好的小型nas（除了它不支持在线视频等一些局限）。但却不是一个好的附件床。它的外链分享（自2019起支持）是打开一张网页，点其中的按钮下载。目前还没有像oneindex一样的方案将它转成直链（能wget url大文件这种，能通过api展示markdown托管静态资源。）。况且受国内政策影响的,在中国，域名绑定一个国内空间就要备案。免备案的都是走国外路线的。 ，分享作为外链全部导到香港,速度大打折扣。况且即使能用，icloud的空间也有限，也没有强大的api支持。不支持其它os开发。我甚至用过用icloud pages+外链分享来做站可是不够省事。

Onedrive是另外一个选项，Windows这二年跟云和linux靠得越来越近了。这一切源于在微软工作的印度小哥的决策，onedrive在同步算法和client app支持方面跟icloud没法比，除了价格，（买断制不带onedrive的单office系列要便宜点，只有订阅制的带onedrive 1t/6t的office365系列618，1111，1212，1231去淘宝买或买可以差不多半价200左右得到，它也有国内网络互联运营版的onedrive。也有官方版走国际线路，国内速度肯定好点。价格也便宜点，但国内和官方版接下来提到的api支持可能不一样，购买时看产品描述问清楚）。，另外它还有对webapp和各种语言的api，这样并不局限于是什么托管环境（vps,虚拟主机，腾讯云函数，herku），借此你可以得到存储在它当中的文件直接外链，配合云主机/虚拟机/云函数环境可以将它做成个人nas和网站图床（不知道免费版5g有没有api可用），这样的程序主要是一些php的，有oneindex,olaindex(这个程序是php cli命令行版，还提供基于aria2 cmd实现的云下载功能),等等。前者可做成web界面的网盘。后者可以做成命令版。其它还有onemanager,onelist,pyone,cuteone,sharelist,fodi,cuteonep。

其中onemanger（https://github.com/qkqpttgf/OneManager-php）也是一个php网站程序，较oneindex,olaindex更强大，它也支持互联版。甚至支持安装在云主机和腾讯scf上。下面通过它来讲解试验。

使用云主机：

建好lamp环境，选择宝塔这样的会比较省事，上传https://github.com/qkqpttgf/OneManager-php代码，创好网站。添加onedrive盘，我们选择国际版，自己申请id和机密，会把你转向到这个链接:

```
https://apps.dev.microsoft.com/#/quickstart/graphIO?publicClientSupport=false&appName=OneManager&redirectUrl=https:%2F%2Fscfonedrive.github.io&allowImplicitFlow=false&ru=https:%2F%2Fdeveloper.microsoft.com%2Fzh-cn%2Fgraph%2Fquick-start%3FappID%3D_appId_%26appName%3D_appName_%26redirectUrl%3Dhttps:%2F%2Fscfonedrive.github.io%26platform%3Doption-php
```

程序提供的转向url中有OneManager&redirectUrl=https://scfonedrive.github.io&platform=option-php这样的一部分参数，可以快速创建，否则需要去https://portal.azure.com/#blade/Microsoft_AAD_RegisteredApps/ApplicationsListBlade手动创建，保存你的应用机密，网页转向，选择php(官方提供一个带php composer的足够环境来支持你的函数)，把注册成功处的应用id也保存下来。以后也可在https://account.live.com/consent/Manage管理你的所有应用。

将客户应用id和机密填入到onemanager处，应用跳转，蹦出一个错，不管，直接进入onemanager url，成功，可以看到用你的域名加文件夹加文件名就可以形成对应文件的直接外链。而你也有一个小\nas网盘了（可以上传，也可以下载，可以在线播放，但速度感人，最好是大量小文件，大文件分享获取有些吃力。不过像moeclub用于存镜像，然后港区ecs通过installnet装机不错）。至于如何加密，如何连接多个帐号用完6T(可以

把这么多用户想象成oss的buket)。自行研究。

使用腾讯云函数机：

腾讯云函数有一个免费额度的cloudbase套（包月免费自动续费），包括静态网站托管（如果要使用这个，cloudbase应用环境要转成按量，不过也照样享有一定免费额度）和scf，这就是我们上面的云网站，阿里云的弹性扩展一类的东西。

我们使用的是单独作为一个产品的云函数去安装。它会出现在cloudbase免费云函数列表中。去cloudbase中去创一个helloworld云函数扩展，进入https://console.cloud.tencent.com/scf/index会看到被分配到了上海区，下面就在这里安装onemanager。

跟安装在云主机上相比，除了不需要安装环境（云函数受自带language runtime backended），这里的区别，从下面步骤可以看出：

直接选择云函数服务区上海新建，命名空间default，另外一个是你的那个cloudbase云函空间，。我们看到php7.2函数模板里面就有onemanager。尝试选择它直接下一步部署，不进行任何设置直接点完成。 函数管理，解发管理，添加触发方式，选择API网关，勾选集成响应。访问API网关，其它就跟上面一样了。

现在尝试将其复制到cloudbase命名空间，在云函数在函数名列表中看到有个复制，但是无法复制到cloudbase云函空间，提示无法完成。只能在函数代码处先下载为zip包，在cloudbase新命名空间新建php7.2空白函数的函数代码处，删掉默认index.php上传zip包,（直接上传onemanager.git源码zip提示函数不正常，需要修正onemanager.git源码为tcf适用版本为https://github.com/tencentyun/scf-demo-repo/tree/master/Php7.2-SCFonedrive使用再cloudbase命令行上传）。代码中config.json中"memorySize": 改为256，选择http触发，触发路径/onemanager，但是执行失败，看函数日志，Object of class stdClass could not be converted to string in /var/user/function/common.php on line 90,原因不明，看源码像是路径处理错误。还可能是因为cloudbase的云函数只支持http而非上面api网关方式模拟http触发导致（而独立腾讯云函数产品 2019年12月起只支持api模拟http）。

不管了，反正都有免费额度，就使用单独云函数版吧。 注意：免费的azure add额度由于频繁调用API也有次数和额度限制，仅建议用于个人网盘性质用途不建议用于像对象存储之类的做站图床场景。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

利用onemanager配合公有云做站和nas（2）:在tcb上装om并使它变身实用做站版

本文关键字：**tcb**上安装**onemanager,onemanger**实用做站

在《利用大容量网盘onedrive配合公有云做你的nas及做站》中，我们初步谈到了使用onemanager配合云函数或云主机做站的投入和尝试，前文说到不能在腾讯cloudbase上安装onemanager，但实际上经过尝试发现是可以的：

注意：云函数产品建议使用包月免费或包月套餐尝试。按量如果控制不好，可能会因为代码问题或外部攻击造成高额费用，比如腾讯scf按量下如果免费额度用完，24小时才停机不会立马停，而且它的计费即使你帐号没钱也会往负了扣，极可能超支。包月则不会有预算超支。

所以我们接下来讲解在包月免费的tcb上安装它：

在tcb上安装onemanager

首先，从http://github.com/qkqpttgf/OneManager-php下载代码，先不上传到cloudbase空间，本地修改platform/tencentscf.php的GetGlobalVariable($event){...}函数体中的$_GET = $event['queryString']为$_GET = $event['queryStringParameters']，这样?admin等参数传递就正确了。然而程序还是得不到入口index.main_handler，直接使用cloudbase后台的新建函数只能用index.man作入口，手动修改入口可以执行，但程序会进一步得不到环境变量，我们可以统一使用cloudbase cli命令行工具全面定制：

cloudbase cli是一个nodejs程序。按腾讯产品文档在本地安装后tcb login --key登录，填入你的用户access keyid和keysecret，在本地做一个待上传目录，在此目录下写如下内容的cloudbaserc.json，同时准备子目录：functions/myonemanager/下放经过上面修改的onemanager代码，到待上传目录(你也可以建一个目录myonemanager，把om源码和cloudbaserc.json统统放进去不用建functions/myonemanager子目录,但是下面cloudbaserc.json中的functionroot要改为../)：

```
{
  "envId": "你的环境",
  "functionRoot": "functions",
  "functions":
  [{
    "name": "myonemanager",
    "timeout": 6,
    "runtime": "Php7",
    "installDependency": true,
    "handler": "index.main_handler有了这个就不用改入口了",
    "envVariables": {
      "Region":"ap-shanghai",
      "SecretId":"你的腾讯accesskeyid",
      "SecretKey":"你的腾讯accesskeysecret",
      "admin": "你要定义给后台的密码，明文",
      "sitename": "站点名，找一个在线base64转码后，将结果填这",
      "hideFunctionalityFile": "1",
      "disableChangeTheme": "1",
      "passfile": "密码文件名",
      "theme": "主题名",
      "timezone": "8",
      "disktag": "盘名1|盘名2",

      "盘名1": "{\"Drive_custom\": \"on\",\"Drive_ver\": \"CN\",\"client_id\": \"你的azure app portal for onemanager的client app
 id明文\",\"client_secret\": \"你的azure app portal for onemanager的client app secretbase64明文找一个base64转成结果填这\",\"disknam
e\": \"明文找一个base64转成结果填这\",\"domain_path\": \"明文找一个base64转成结果填这，形式是域名1:/目录1|域名2:/目录2......\",\"refresh_
token\": \"看接下来手动获取方法\",\"token_expires\": 9999999999}",

      "盘名2": "{同盘1生成方式}"

    }
  }]
}
```

可以看到盘名1后面的参数是一个字串，然而它本身也是个json，将json转成字串供cloudbase识别的方法是将所有"都\转义一下，如果你嫌麻烦实在想图方便，可以在正常非cloudbase区或vps上直接搭建一个onemanager,注册盘，然后将结果填到上面。 上面的clientid和secretid，正常方式安装od是自动的，但其实你也可以手动https://portal.azure.cn/#blade/Microsoft_AAD_IAM/ActiveDirectoryMenuBlade/RegisteredApps去生成，我这里是互联，新注册->任何组织目录中的帐户,多租户->重定向url:web,https://scfonedrive.github.io，这里可以直接看到client id了。看左边列，api权限不用设置，证书和密码->新客户端密码，期限永久，就看到secret了。 至于refresh token，也可以从https://service-36wivxsc-1256127833.ap-hongkong.apigateway.myqcloud.com/release/scf_onedrive_filelistor手动得到。token_expires填10位9。

cloudbaserc.json准备完毕，最后cd到这个目录，cloudbase functions:deploy，这样你就得到了一个完全手动和程序化的安装方式，后台改变256m到128m,触发路径/或/xx不能是/xx/结尾，以后deploy，提示覆盖直接确认即可。

更多让**onemanager**实用做站的考虑

我们知道，云函数主要是处理api结果的传送，在这里不能传递大量数据，保证一次http所有结果在最短的ms里完成，否则按调用次数和调用时长及内存占用的云函数会相应产生相对高的花费，查云函数后台，确保每次2ms内的调用是合理和正常的。故onedrive和托管onemanager等程序的空间（这二者最好是同一地域的，比如世纪互联配国内空间，国际版配港区空间）对提高调用速度至关重要，有些onedrive列表程序支持，前后端分离，云函数纯粹后端只返回api结果不包前端渲染。api速度快(列文件很快)如fodi.

处理静态资源问题和定制模板：

由于od是一个特殊的程序，它定位于网盘文件列表而非带资源的网站展示，它绑定的工作域名下，每一个路径，如果不是显式的?setup这种参数，就是文件调用，因此，它对所有js,css的引用，都是外部的(如果发现网页慢，将它换到快点的cdn地址)。这也是为了上面说的一次request/respon能尽快调用完成，所以od的templates都是不带静态asserts的。------ 所以并不推荐将静态template资源放在代码theme下，然后根据判断它是不是网盘文件进一步处理。

谈到od的templates,其实它也是网页模板技术的运用（本质就是定义一系列开头结尾组合形式的模板变量块，然后替换），你可以查看已有template自己写一个比如最简单的那个nexmoe1.html，，模板体逻辑通常是这样的：开头icon处理块，管理相关的style，前端样式style块,外部css和jss引用，渲染omf，md文件的逻辑块(require一个maked js然后根据md content在页面直接render)，。列文件和目录的逻辑（其实又包括div逻辑块，js逻辑块）,blaaaa.....。

加速和cdn：

我们知道网站速度至关重要。不光对用户对运营也是如此。要实用做站的话，必须要配cdn。对于cdn加速，比如要求文件静态化为各个url路径为目录名的目录下的index,html。onemanager有没有相关方面的支持呢

od是带缓存的。主要是存取到云函数backend空间的system temp目录中。这样列文件和目录的时间会相对变少。程序效果和体验会最佳。od内部对text文件(包括markdown)都是有1800秒缓存的。这个过程在common.php中，查看fetch files,render list主要函数，gethiddenpass()等类似函数。

对于md，上面说到它是在客户端通过client js来渲染从服务端拉取下来的内容的（如果发现大量md的网站慢，有可能这个js处在慢速cdn上，换个），，对于html则是跟md一样直接下载并output不经过主题渲染处理，相当于部署了一个静态页面。

本来它是在客户端生成的。其实在服务端也可完成md,比如下载一个php的渲染器mdparser.php，再在index.php中include 'mdparser.php';common.php中在对应headmd处理位置的地方作修改：

```
$parser = new HyperDown\Parser;
$headmd = str_replace('<!--HeadmdContent-->', $parser->makeHtml(fetch_files(spurlencode(path_format(urldecode($path) . '/head.
md'),'/'))['content']['body']), $tmp[0]);

$tmp = splitfirst($html, '<!--MdRequireStart-->');
$html = $tmp[0];
$tmp = splitfirst($tmp[1], '<!--MdRequireEnd-->');
```

在服务端生成html作为api结果返回会稍微增加api时间，但结果更合理。你可以进一步把渲染好的html结果保存在cache中对应md地址的子目录index.html（而不是原来的raw md content）中，然后下回fetch到这个md地址，直接取cache，按处理html的方式，直接render。这样的“全站伪静态”对cdn也是有用的。

你也可以修改refreshcache的逻辑，od有一个refresh cache，它是先切换到当前目录下就refresh哪个目录的cache。且只工作在手动管理模式下，其实你可以把它做成自动静态化的，浏览到对应md位置就生成对应index.html到cache/md命名子目录下。然后在后台做一个一键md全站生成html到cache静态化按钮和功能。或者生成到cloudbase的存储中。---- 已经有这样的程序了，静态网站生成器作为云函数，生成静态文件到oss，云存储。像极了自动化的github page action。

---

ps:。一般来说，根据我的“硬件使用3年即报废，不应超过500元/年，总共应投入1500，超过这个都是这个时代的奢侈品“的嘴强说法，如果做站用的虚拟资产也可归其中，则3年期的1t互联或淘宝买的个人/家庭office365官方版+3年最低配的1h1m云主机或虚拟主机也正好落入这个区间(office365已改为microsoft365,感觉国际个人/家庭版的没有国际商用版的快，优点是前者多了个文件vault库)。可惜bat等不支持终身制虚拟主机（onemanager需要php curl扩展）或终身制弹性云主机,否则这就是一种不虐心云主机的省事投资了。

在前面，我们为oc增加了wp功能和note功能，现在为od增加静态html功能，同样基于做站和静态化考虑，现在，我们对od开刀了。可能最终，我们要类群晖的dsm based on a netdisk,将od类似的理念打造一个mineportal os，而不仅仅停留在一个file lister，这是比云主机黑群还实用的方案，比如myportalnew: email是必须的+类似微信自托管的chat app+file sync+mateos app那些。。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

利用fodi给onemanager前后端分离(1)：将fodi py后端安装在腾讯免费cloudbase

本文关键字：将**fodi py**后端安装在腾讯**cloudbase**

在前面《利用onemanager配合公有云做站和nas》中，我们提到onemanager是个可以装在腾讯云函数API网关或cloudbase上作为云函数的产品，onemanager php backend它本身也是调用onedrive的api，在前面《owncloud微博式记事本》《wp2oc fileshare》中我们也提到这种web api的修改定制。最近的《打造小程序版本公号和自托管的公号:miniblog》也是这种例子,后端cloudfuntion化，前端miniprogram则被部署到微信，，前后端分离和web api化调用是早已流行的趋势。

在《利用onemanager配合公有云做站和nas》中我们也提到要继续为onemanager找提高浏览体验用的缓存方案静态化方案，onemanager拉取文件列表延迟高体验不佳（这种延迟一部分就是它本身前后端不分离，一次返回结果混合了渲染html的客端逻辑，客户端也没有 ajax,另一部分是onemanager这类程序本身也是使用api的，返回结果来自OD本身是外部的，其只有有限的doctrine缓存技术，没有像混合云那样的多样化混合本地和远程云结果构建高效网站体验的能力），这次我们找到了fodi。其前后分离和客户端ajax技术，体验比单纯的onemanger要好得多，fodi分离的好处是，相当大部分逻辑都被放在了客户端。如浏览文件，服务器仅负责一次返回一些json数据，现在的浏览器本身一般也有本地缓存存储。，整个可以带来类似pwa,js based react native app的效果，这也是fodi名字中的fast的理念技术基础。

我们的目标是使od像fodi那样前后端分离，并研究尝试让它接上fodi前端的方法。----- 等等，直接用fodi前后端一套流不香吗？fodi有二个后端，一个backend_cf for cloudflare,一个py for 云函数，fodi py后端输出的是加密的json对我们接下来的研究不够直观化（除非chrome f12），而nodejs端直接输出直观的json结果，但nodejs runtime(<=10.15)又不支持addeventlistener()，它是cloudflare的webwoker api用的(这是一种客户端API，然而CF把它ported到了服务端)，所以我们打算用改造onemanager的结果来适配fodi前端。使之成为fodi的php backend。

注意：Fodi有部分预拉取列表功能，所以产生的云函数调用会比较多（所以推荐将fodi装到vps）,但是om的缺点是云函数调用流量方面也相对较多。

这里还是介绍将fodi后端部署到cloudbase的方法

上传和修改后端逻辑:

用cloudbase-cli提交backend-py：

```
{
  "envId": "default-4gpm7vnrb911600e",
  "functionRoot": "functions",
  "functions":
  [{
    "name": "fodi",    //fodi就是backend-py
    "timeout": 6,
    "runtime": "Python3.6",
    "memorySize": 128,
    "installDependency": true,
    "handler": "index.main_handler"
  }]
}
```

以下保证要使用https://xxxx.service.tcloudbase.com/fodi形式的调用路径(后台的接入路径定义)：

```
def router(event):
    """对多个 api 路径分发
    """
    door = 'https://' + event['headers']['host']
    print('door:'+door)
    func_path = '' #event['requestContext']['path'] //置空
    print('func_path:'+func_path)

    api = event['path'].replace(func_path, '').strip('/')
    api_url = door + '/fodi' + event['path']  //加一个fodi串

    queryString = event['queryStringParameters']  //这里改
    body = None
    ......
```

弄好后，访问上面的接入路径，仅/fodi，输出path error和后面一长串东西就代表服务器搭建正常，

获取refresh token

然后就是那个refreshtoken的获取，http://scfonedrive.github.io已经挂掉了，我们可以自建，先在某网站下建一个get.html:

```
<html><meta charset=utf-8><body><h1>Error</h1><p>Please set the <code>refresh_token</code> in environments<br>
    <a href="" id="a1">Get a refresh_token</a>
    <br><code>allow javascript</code>
    <script>
        url=window.location.href;
        if (url.substr(-1)!="/") url+="/";
        url="https://login.partner.microsoftonline.cn/common/oauth2/v2.0/authorize?scope=https%3A%2F%2Fmicrosoftgraph.chinaclo
udapi.cn%2FFiles.ReadWrite.All+offline_access&response_type=code&client_id=04c3ca0b-8d07-4773-85ad-98b037d25631&redirect_uri=h
ttps://scfonedrive.github.io&state="+encodeURIComponent(url);
        document.getElementById('a1').href=url;
        //window.open(url,"_blank");
    </script>
    </p></body></html>

如果是国际版url换成:
 url="https://login.microsoftonline.com/common/oauth2/v2.0/authorize?scope=https%3A%2F%2Fgraph.microsoft.com%2FFiles.ReadWrite
.All+offline_access&response_type=code&client_id=4da3e7f2-bf6d-467c-aaf0-578078f0bf7c&redirect_uri=https://scfonedrive.github.
io&state="+encodeURIComponent(url);
```

再在根下建一个index.html

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
    <head>
        <title>OneManager jump page</title>
    </head>
    <body>
        <a id="direct">No link here!</a><br>
        If not auto jump, click the link to jump.<br>
        如果长时间未跳转，请点击上方链接继续安装。<br>
        <label id='test1'></label>
        <script>
                  var q = new Array();
            var query = window.location.search.substring(1);
            //document.getElementById('test1').innerHTML=query;
            var vars = query.split("&");
            for (var i=0;i<vars.length;i++) {
                var pair = vars[i].split("=");
                q[pair[0]]=pair[1];
            }
            var url = q['state'];
            var code = q['code'];
            if (!!url && !!code) {
                url = decodeURIComponent(url);
                if (url.substr(-1)!="/") url+="/";
                var lasturl = url+"?authorization_code&code="+code;
                document.getElementById('direct').innerText = lasturl;
                document.getElementById('direct').href = lasturl;
                window.location = lasturl;
            } else {
                var str='Error! 有误！';
                if (!url) str+='No url! url参数为空！';
                if (!code) str+='No code from MS! 微软code为空！';
                document.getElementById('test1').innerHTML=str+'<br>'+decodeURIComponent(query);
                alert(str);
            }
        </script>
    </body>
</html>
```

把get中scfonedrive.github.io换成你的index.html所在的网站地址，（之后保证把py后端中用于认证的地址和那个clientid,clientsecret替换用你自己新建的一个，具体方法见我前面的一些文章）

调用后结果显示在url中（整个页面显示404是没有处理结果的php后端，除非你把https://github.com/qkqpttgf/OneDrive_SCF部署在index.html所在的网站），分辨复制即可。

安排好后端和refreshtoken后，调用接入路径/fodi/fodi/，输出看到其输出的加密的json结果，就代表refreshtoken也正常了。 开始部署前端，可以另外一个网站，能托管html的就行。也可以在后端另起一函数，部署如下index.py:

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-


def main_handler(event, context):
    f = open("./front.html", encoding='utf-8')
```

```
    html = f.read()
    return {
        "isBase64Encoded": False,
        "statusCode": 200,
        "headers": {'Content-Type': 'text/html; charset=utf-8'},
        "body": html
    }
```

front.html当然是配置好的那个前端文件。

如果你fodi前端调用发生for each,length之类的提示错误，往往是refresh token没获取对。如果发生跨域错误（chrome f12可看到），则在后端面板中需要配置一条客户端网站的安全域名。

整个代码也较大，我们稍后将3rd依赖库精简掉，把那个加密逻辑去掉让输出常态化。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

利用fodi给onemanager前后端分离(2):测试json

本文关键字:利用**onemanager**给**fodi**做**php**后端

在前面《利用fodi给onemanager前后端分离1》中我们介绍了在cloudbase上安装fodi py后端的方法，这里继续尝试将om作为fodi的后端也尝试弄上。

这里要说个历史，fodi的作者也是参考了onemanager的,精简的onemanager流程就是fodi backend那些(https://github.com/qkqpttgf/OneDrive_SCF即是那个最简的php后端)。别看onemanager比fodi体量比fodi大，它其实后端就二主体逻辑文件。platform/TencentSCF.php和index.php(转common.php)，一个是处理scf的，一个实际上是将od api结果转换成om api结果并渲染的结果。我们的目的非常明确：om最终返回一个网页html，onemanger混合了前端在输出结果里。作为一次api response，所以我们需要给它前后分离一下。让它返回正常的类fodi后端的json结果，而不是post render过的html，然后考虑对接到fodi前端。

纯 **api** 服务器:基础工作

所幸虽然onemanager本身是客服输出合体的，但内部也是细分到不同函数的易区分，客服也进行了区分。这使得我们的工作较为容易进行：我的版本是https://github.com/qkqpttgf/OneManager-php/commite439ed8e68c4ae0d8d42bc5c293a3ba06aa1bc9c，20200607-1856.19，主线逻辑：main()->list_files->files_json(),or render_list()，分解重要的函数或逻辑有二支：（1）list_files($path)->fetch_files($path)，（2）render_list($path = '', $files = '')->fetch_files($path)->output，renderlist就是客户端输出逻辑。onemanger支持https://www.xxxx.com/?json的调用，这个函数就是common中的function files_json($files)，会输出json，相当于在fodi后端index.js中?path输出json那些逻辑。接下来的工作就是这二个函数的处理。

我们的测试环境是在一台VPS中(对应platform/normal)进行，我是在宝塔面板中完成的，分成二个网站部署好om代码和fodi frontend。然后fodi原来的后端我是在cloudflare中开免费worker搭建的。fodi front复制成二份html放在第二个站下，我们的思路是比对这二套系统的前后端，以促成我们的目标。事先在文件夹中放一些文件，还有一些准备工作：

om需要预处一下：服务端index.php顶层代码块（这里并没有处在一个函数中）中：

```
} else { include 'platform/Normal.php'; //从这句开始改
    //$path = getpath(); 注释掉
    $_GET = getGET()后面加一条：
    $path = $_GET['path']; //$_GET是收集到的用户输入的参数组成的数组，该getGet()在platform/normal.php下，这里是得到path项,这样，用户在浏览器中输入什么网址(http://apiurl/?path+参数)，就会展示该路径下的json结果
```

服务端common.php这个三合一平台通用接下来逻辑中，main($path)函数中：

```
if ( isset($files['folder']) || isset($files['file']) ) {
  return render_list($path, $files); //原来的带完整html输出的注释掉，换成接下来这句
  return files_json(/*$path, */$files);
```

function files_json($files)函数末尾：

```
return output(json_encode($tmp));
return output(json_encode($tmp), 200, ['Content-Type' => 'application/json']); //这条实际上在新版被修复了
```

客户端fodi那个frontend html中：要将/fodi/去掉或模拟出来

```
window.GLOBAL_CONFIG = {
SCF_GATEWAY: "https://apiurl/",    //这里你也可以用https://apiurl/?json如果上面你没有把render_list改成files_json
SITE_NAME: "FODI",
IS_CF: true
};
if (window.GLOBAL_CONFIG.SCF_GATEWAY.indexOf('workers') === -1) {
  window.GLOBAL_CONFIG.SCF_GATEWAY += '/';   //window.GLOBAL_CONFIG.SCF_GATEWAY += '/fodi/';改成/
  window.GLOBAL_CONFIG.IS_CF = false;
}
```

在宝塔服务端网站的设置->配置文件中，加好ssl，并把web安全域名设置一下，否则接下来chrome f12调试会产生跨域错误（cloudflare没有这个问题）。chrome这东西不光是web browser，也是webdever的IDE加调试器，其实这个也可以在程序逻辑中设置，但比较麻烦。

```
for nginx:
server块location下，直接放在伪静态里也可以
add_header 'Access-Control-Allow-Origin' '*';

for apache:
    </Directory>
```

```
    Header set Access-Control-Allow-Origin "*"
</VirtualHost>
```

然后就可以接下来调试了(注意不要用safari尽量用最新的google chrome，支持度足够，macos big sur之后的safari才能返回正确结果)，调试程序最难的是找到调试的方法，以上这些都可以在chrome的f12，network->request,respone中看到，否则还是chrome f12产生错误，对于后端，你加入的调试只能在云函数执后台看到，对于前端，在chrome中查看。要区分哪些?path是服务端测试调用的，哪些是客户端的。

纯 **api** 服务器:调试工作

在fodi构造https://apiurl/?path=%2Fd%2Fmirrors之类的url（%2F是/），比对cf和php后端观察到输出的结果。结果是都不会变的。喂给fodi的?path=返回结果不会变的，因为它是接受json请求的而不是GET形式的?path。喂给php端的需要再处理一下。

这是om输出的：

```
{"list":[
{"type":1,"id":"01AL5B7D25YGLLNRUY5FG3INXIYG5BDB5V","name":"d","time":"2020-07-23T14:14:52Z","size":14716769375,"mime":null},
{"type":1,"id":"01AL5B7D26ZI4V2QKN35HJSUUZPTHQ3KQN","name":"docs","time":"2020-08-02T05:59:38Z","size":5640494,"mime":null},
{"type":0,"url":"https:\/\/balala...","id":"01AL5B7DY3H337ZRZ3BNAIUSPVR5RXLU2M","name":"readme.md","time":"2020-08-05T12:25:06
Z","size":1311,"mime":"application\/octet-stream"}
]}
```

这是fodi的（在cloudflare那个界面可以调试到）：

```
{"parent":"/","files":[
{"name":"d","size":14716769375,"time":"2020-07-23T14:14:52Z"},
{"name":"docs","size":5640494,"time":"2020-08-02T05:59:38Z"},
{"name":"readme.md","size":1311,"time":"2020-08-05T12:25:06Z","url":"https://blalaa...."}
]}
```

我们的思路就是让其输出一致，而且由于fodi客户端那个html是用的ajax请求ajax结果，?path并不是提交用的。而是作为form data被返回的。这是后来的问题，先处理输出一致。

因为正确的json是parent,files,name,size,time,url，所以在common.php function files_json($files)中：

```
....
$tmp['list'] = [];改为        $tmp['files'] = [];
......
foreach ($files['children'] as $file) {
  ...
  $tmp1['name'] = $file['name']; //包括这句，以下三新加
  $tmp1['size'] = $file['size'];
  $tmp1['time'] = $file['lastModifiedDateTime'];
  ...
  // $tmp1['type'] = 0;   //这句注释
  // $tmp1['type'] = 1;  //这句注释
  //$tmp1['id'] = $file['id']; //包括这句。接下来5句注释，中间3句被移到上面了
  //$tmp1['name'] = $file['name'];
  //$tmp1['time'] = $file['lastModifiedDateTime'];
  //$tmp1['size'] = $file['size'];
  //$tmp1['mime'] = $file['file']['mimeType'];
  .....
  array_push($tmp['list'], $tmp1);改为array_push($tmp['files'], $tmp1);
  ......
```

加入parent,在main()合适的位置（对应于它在结果中输出parent字段在整个全部字段所在的位置，即$tmp['list'] = [];改为 $tmp['files'] = [];一句上面）加上

```
$a= str_replace($_SERVER['list_path'],"",array_pop(explode(":",$files['parentReference']['path'])));
if (isset($a)) {
    $tmp['parent'] = $a;
}
if ($a =='') {
    $tmp['parent'] = '/';
}
```

不断测试url，随着path参数的改变，json终于有fodi相同的结果。这样，fodi终于接受到初步正确的数据了(仅初始)，，但是到现在为止,点击任何条目包括文件夹都不会出来正确结果，如上所说，我们只是让后端返回了我们认为正确的结果，我们的程序靠喂?path这个工作。到现在为止这仅是一种手动工作。fodi frontend并不与之联动，这是第一步，fodi前后端它是自动联动api path参数变化经的。比如，查看chrome f12,客户端还

接受其它参数&encrypted=&plain=&passwd=undefined。（云开发会校验网页应用请求的来源域名，您需要将来源域名加入到WEB安全域名列表中。）

这是由于fodi客户端那个html是用的ajax请求ajax结果，?path并不是提交用的。而是作为form data被返回的。一个是用户发动的url path，一个是xhr发动的url request path，这二者不是一回事。fodi客户端服务端交互有它自己的逻辑。这里面还有复杂的ajax参数交互（查看chrome f12 ajax类请求产生的xhr对象）：

比如，消息体中的数据起作用,对于URI字段中的参数不起作用，被封装在xmlhttprequest的formdata中。服务端要还原处理这类formdata，，，这一切是因为服务器端请求参数区分Get与Post。get 方法用Request.QueryString["strName"]接收，而post 方法用Request.Form["strName"] 接收，blala..ajax还有其它细节。下一步：与fodi frontend对接。可能需要更多研究。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在pai面板上devops部署static site

本文关键字：**blog**联合，一文多发，**blog**内容联合。**headless ghost cms**

前面《在云主机上手动安装腾讯PAI面板》中，我们发现PAI是一个利用git和devops，在仓库的根下放置.pai.yml来达到自动部署+运行APP目的的一种机制，除了没有容器和隔离，其它都这属于CD/CD的思路，下面来实际部署官方的hexo static blog例子，虽然我有点不承认它是serverless的hexo静态网站生成器（tx云函数官方也有一个staticsite版本，稍后会谈到真正的这类产品是headless ghost cms这种）但肯定比普通hexo要方便一点，完成之后的效果就是能作到类似利用git hook部署网站+自动更新（稍后也会谈到它的另外一个效果：可以与其它git仓库，如github,gitee作内容联合）。。

不废话了

安装hexo:

在管理面板中，我们要安装的是这个仓库，https://gitee.com/TencentCloudBase/pai-mate-hello-example-static，这个仓库是个hexo的example site项目，（正常安装后，会生成/data/pai_mate_workspaces/pai-mate-hello-example-static/pai-static-pages.js和/data/pai_mate_workspaces/pai-mate-hello-example-static/ecosystem.config.js，ecosystem.config.js是pm2用的守护脚本，它守护pai-static-pages.js开启3000端口上serving public dir的静态http服务，这个服务没开起来之前，访问与pai安装时的绑定域名会一直502，安装好后，可以访问域名），但是除了这些，它却没有安装好hexo本身（也许官方期待用户手动进服务器安装） 也就少了至关重要的hexo生成网页的作用，查看它的.pai.yml下，没有部署脚本deployScripts:，只有一行static: public（这正是上面二个生成文件生成的语句），---- 无论如何，这个仓库中的.pai.yml不完善。我们来完善补全这个仓库一下：

.pai.yml，注释掉 static: public，另：不知为什么，hexoauto加--watch会与上一条冲突，导致3000频频挂掉，故 --name hexoauto后不加--watch

```
# static: public
deployScripts:
  start:
    - npm install
    - npm install --unsafe-perm=true --allow-root -g hexo-cli
    - npm audit fix
    - pm2 start -s ecosystem.config.js
    - pm2 start "hexo generate --watch" --name hexoauto
  restart:
    - hexo clean
    - hexo generate
```

手动在仓库里添加以上二个js文件，pai-static-pages.js:

```
// This file is auto-generated by PAI-MATE
const handler = require('serve-handler')
const http = require('http')
const server = http.createServer((request, response) => { return handler(request, response, {public: 'public'}) })
const port = +process.env.PORT
server.listen(port, () => { console.log('Running at http://localhost:' + port) })
```

手动在仓库里添加以上二个js文件，ecosystem.config.js

```
// This file is auto-generated by PAI-MATE
module.exports = {
  apps : [{
    name: "pai-static-pages",
    script: "./pai-static-pages.js",
      watch: true,
      env: {
        "PORT": 3000,
        "NODE_ENV": "production",
        "NODE_PATH": "/usr/local/node/lib/node_modules",
      }
  }]
}
```

这样部署后仓库就运行起来了，点管理面板中重启应用能达到最基本的自动部署仓库中的内容和启动静态网页服务的目的。只是，它缺少一个pm2 git clone。依赖手动重启restart处的hexo generate。整个应用处，还是需要一道工序（而理想状态下，内容源git一下应该是唯一的工作）。

内容联合：

曾经我们有网盘联合，内容转存。blog和文章作为一种“内容”，有时也需要联合和一文多发。这类功能应该加到各大笔记和知识库核心功能中，当然也有这样的独立产品如artpub。

上面的git方式联合，只是让仓库和这个静态站之间作内容源联合。这种基于git devops的工具层的东西，在内容联合方面还是有局限的。

在多样内容源联合方案的选择上，还是基于API的好（因为可编程的东西不局限于工具，见《用开发tcpip的方式开发web》），比如那种headless ghost cms content api+JAMstack front-end like hexo的方案就好多了(这也使得hexo这类工具通用BLOG静态生成器上升为通用网站生成器的境界)，在“迁移内容”，和”换前端“方面都有很大的自由度（虽然费折腾但是做成工具和产品也一样）。

下文，由于pai是个类似baota panel基于pyenv的python项目管理器，下文探索它的py项目部分，未来讨论利用staticsite和markdown生成整站单页pdf book等课题

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在openfaas面板上安装onemanager

本文关键字：**openfaas onemanager**

在前面《在云主机上手动安装腾讯PAI面板》《利用openfaas faasd在你的云主机上部署function serverless面板》中，我们介绍了二种虚拟主机/容器后端，它们都可以搭配nginx作为前端，形成webstack，和通用服务器应用栈，而且都支持多语言，支持devops部署。其中，前者用git作devops，后者用容器（faasd中用containerd），后者的devops更全面：要知道，现在的devops构建基本是用docker完成的，无容器不devops。况且，openfaas是对我《最小编程学习集与开发栈选型：1，语言选型与开发融合，2，云系统，云服务器，虚拟boot，语言运行时和应用容器，及平台融合，3，云APPSTACK云调试云DEVOPS及开发融合》中的2，3的整个暂代实现（替代目前还无法做到的统一语言，统一内核）,它与我的cloverboot+黑群晖黑苹果等cloud os，组成了一个暂代可用的minstackos:一个企图将包括上面1，2，3的最小开发栈实现做入boot的方案。

其实老早就接确过docker和k8s,swarm这种容器管理面板，但是一直没怎么深入使用(有这功夫为何不研究coreos?)，一是对docker aufs不放心(二是docker本身虽然利用了内核功能，但是本体属于在应用级用虚拟化重新构建网络和文件系统，带来较高负担和延时)，二是swarm,k8s这种比较重，直到接确云函数，再到这里的轻量级faasd，才发现基于容器的面板也可以很轻量（当然，pure k8s和云函化的容器面板还是有点不一样的）。二是云函数本身就关注用它运行功能，而不用涉及到存放数据，因此docker aufs这种可能导致文件系统污染的缺陷也就不足于当回事了。最后这种容器用于生产环境的缺陷慢慢也正在基本被彻底修复。

> faasd用的是containerd，但使用docker构建应用，注意这里的docker-ce只是离线构建openfaas/faasd用的docker镜像之用，并不会加大faasd的运行时，这里提到openfaasd用docker image as app meta(dockfile vs pai.yml)，没错，openfaas用docker作devops。所以可以封装构建流程。而不仅仅像pai面板那样停留于用git拉取。

下面，接上文《利用openfaas faasd在你的云主机上部署function serverless面板》，我们将onemanager移过来。----- 为什么不是fodi,fodi毕竟没有url,不利于运营。

好了，不费话。

调试与基础

调试，前调试后调试，都是运维开发的必要过程，甚至是主要过程。因为它是搞清问题的必要实验过程。

在faasd所在主机上apt-get install docker-ce，我在faasd所在主机是一台装上了deepin20a的云主机，因为这样结合teamviewer可以远程桌面(linux上远程桌面大部分都是位图截取算法的不要用。tv效率高，支持本地远程剪贴板交互，而且tv还有一个图形scp功能。deepin+1h2g开2g swap完全可以驾驭，开机只占1G内存)可以长时间保持在线，而不必总是用本地的ssh(十分易断)，

还有，faasd构建应用涉及到大量与docker.io的交互，这个在国内云主机网络环境下速度是很弱的，虽然有一些方案支持自建docker registry给faasd用，但是太折腾，不如在云主机上装v工具，然后命令行下export http/https_proxy。而且，dokcer-cli是支持应用级proxy的，

```
sudo mkdir -p /etc/systemd/system/docker.service.d
sudo touch /etc/systemd/system/docker.service.d/http-proxy.conf
编辑http-proxy.conf
[Service]
Environment="HTTP_PROXY=http://127.0.0.1:8888/"
然后重启服务：
sudo systemctl daemon-reload
sudo systemctl restart docker
测试是否设置成功：
systemctl show --property=Environment docker
```

然后是登录docker login (registry name，对于官方dockerhub这里可省)，按提示输入用户名和密码。

当然市面上也有很多基于docker的免费或商用docker devops，如果上一步你选择了阿里开发中心的这些私有库，需要修改~/.docker/config.json到/var/lib/faasd/.docker/config.json供接下来使用。

解决这二个问题后。基本可以往下了：mkdir test && cd test && faas-cli new onemanagerforopenfaas --lang dockerfile

vi ./onemanagerforopenfaas.yml 在image路径前加上你的dockerhub用户名，或者把它改成完整地址（带registry。比如国内的阿里云docker仓库,记住版本最好要写上，不能用latest???）注意官方dockerpush支持private镜像，但是pull时会提示denied，所以不要把这个镜像设为private。(如果你使用官方dockerhub，且是private仓库，那么也需要~/.docker/config.json到 /var/lib/faasd/.docker/config.json，否则接下来通过faas-cli调用docker pull的时候会提示认证失败，push没问题，默认gateway127.0.0.1:8080需要一次faas-cli login --password xxx,自定义gateway的方法：dockerfile中gateway改且faas-cli login --password xxx --gateway配合改)

上面我们不用带语言的--lang php7（它是php72），是因为：虽然--lang dockerfile和--lang php7都会拉取一大堆template，但后者文件夹混乱。涉及到的修改需要在templates里进行，而前者涉及到的所有改动仅发生在一个文件夹，即生成的onemanagerforopenfaas这里。我们只需从template里复制一些关于php72的脚手架文件即可(index.php和functions文件夹)。不必修改template(也不要删掉，如果调用faas-cli xx -f ./xx.yml，它会再次自动下载)。

但我们这里不复制index.php和functions，把onemanager复制到生成的onemanagerforopenfaas根目录下，我这样做是为了不涉及太多的文件夹结构，也方便接下来的onemanager source（我使用的om版本依然是20200607-1856.19），因为它本身并没有使用类似如下的文件夹结构：

然后，对应修改onemanagerforopenfaas下的dockerfile（如果使用--lang php7要到template/php7下修改，十分混乱）：

```
# Import function
WORKDIR /home/app
COPY ./ ./
WORKDIR /home/app
```

dockerfile中有一处composer install由于被q也很费时。可修改dockerfile，禁掉composer install行，因为onemanager在vender中自带依赖，如果不禁掉，我们也可以在composer.json中自己手工添加下面chunks以提速(其实你也可以删掉composer.json，因为我们并没有使用到它)：

```
"repositories": {
    "packagist": {
        "type": "composer",
        "url": "https://packagist.phpcomposer.com"
    }
}
```

这样调试环境就建好了。调试过程就是修改源码（当然放在第二节中讲，这里是最主要的调试工作,这里仅关注流程），然后sudo faas-cli build -f ./onemanagerforopenfaas.yml && sudo faas-cli push -f ./onemanagerforopenfaas.yml && sudo faas-cli deploy -f ./onemanagerforopenfaas.yml。

下面仅给出结果，不给出具体的调试过程，但是要知道下面得出的结果是这里的不断调试过程得到的,所以这里的流程必不可少。输出信息你可以在源码index.php中写echo，可以观察8080 invoke按钮出来的信息，或网页直接展示。也可以journalctl查看。

源码修改：

faas的那个web环境类似虚拟主机空间，所以主文件index.php中(这也是程序的入口)，我们去掉for tencent的main_handler和for aliyun的handler，直接：

```
<?php

error_reporting(E_ALL & ~E_NOTICE);

include 'vendor/autoload.php';
include 'conststr.php';
include 'common.php';
include 'parsedown.php';


include 'platform/Normal.php';
$path = getpath();
$_GET = getGET();

$re = main($path);
$sendHeaders = array();
foreach ($re['headers'] as $headerName => $headerVal) {
        header($headerName . ': ' . $headerVal, true);
}
http_response_code($re['statusCode']);
echo $re['body'];
```

大部分需要调试的地方跟《利用onemanager配合公有云做站和nas（2）:在tcb上装om并使它变身实用做站版》一文一样，属于环境参数和path获取不到。

platform/normal.php中：

```
function getpath()
{
    $_SERVER['firstacceptlanguage'] = strtolower(splitfirst(splitfirst($_SERVER['HTTP_ACCEPT_LANGUAGE'],';')[0],',')[0]);
    if (isset($_SERVER['HTTP_X_FORWARDED_FOR'])) $_SERVER['REMOTE_ADDR'] = $_SERVER['HTTP_X_FORWARDED_FOR'];
    $_SERVER['base_path'] = path_format(substr($_SERVER['SCRIPT_NAME'], 0, -10) . '/');
    $p = strpos($_SERVER["Http_Path"],'?');
    if ($p>0) $path = substr($_SERVER["Http_Path"], 0, $p);
    else $path = $_SERVER["Http_Path"];
    $path = path_format( substr($path, strlen($_SERVER['base_path'])) );
    return substr($path, 1);
    //    return spurlencode($path, '/');

}
```

以上注意到，faas使用的php-cli index.php唤起的服务。php72的$_SERVER数组中，并没有request_uri，但是完全可以用base_path代替。否则index.php中获取到的$path永远为空。这是由不同的web环境导致的问题。

另外一个需要修改的地方：

```
function getConfig($str, $disktag = '')
{
    global $InnerEnv;
    global $Base64Env;
    //include 'config.php';
    //$s = file_get_contents('config.php');
    $configs = ''
    ......
}
```

以上，我们把原本存在于config.php中的'{}'放到上面代替''，因为正常逻辑下程序不能获取到config.php，我们需要手动用《利用onemanager配合公有云做站和nas（2）:在tcb上装om并使它变身实用做站版》获到的envVariables并把参数给它。这样getConfig('timezone')这样的调用才能发挥作用。具体为什么获取不到config.php。。我也没有深入调试。应该也是openfaas所属特殊web环境导致的问题。

你可能还需要禁掉getGET()中的这段:

```
if (!ConfigWriteable()) {
            $html .= getconstStr('MakesuerWriteable');
            $title = 'Error';
            return message($html, $title, 201);
        }
```

还有common.php中一些涉及到html输出的也给禁掉，以适应特殊的openfaas web环境。

突然想把git和docker弄成netdisk backend的静态仓库，不用做到git pull能git clone就好，因为它们的原理上，也是用curl来下载的。

发现一个v使用终端代理命令只在当前窗口生效的问题,如果执行./xx.sh是不能生效的此时开全局也无用，方法是要把该命令也写一次到./xx.sh中。

还有panel.sh缺少多租户环境的功能，对于openfaas是linkerd，我们知道，pai这种也是不支持多租户的，一次只能一个应用。多租对于个人用户也是有用的，那就是可以为每一个函数指定一个子域名并代理出去，目前openfaas使用/function/应用名的形式，可以寻求直接在主域上展示的方法。

还有那个顽固的dail tcp failed faasd-provider问题，日后也需要寻求解决

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

在openfaas面板上安装onemanager（2）

本文关键字：**openfaas onemanager,nginx仅监听ipv4，disabling IPv6 name/address support: Address family not supported by protocol**

在前面《在openfaas面板上安装onemanager》中我们讲到自建云函数将om放到自己服务器的方法，但是遗留了一些问题，1，经常会有“error finding function onemanagerforopenfaas.: Get http://faasd-provider:8081/system/function/onemanagerforopenfaas?namespace=: dial tcp: i/o timeout”，2，整个应用访问延时感比较明显（这些在腾讯scf也有,容器是多层虚拟化有抽象成本），但在faasd中，还有其它一些导致延时的问题。3，程序中有一些路径不对，readme.md可以显示，但压缩文件下载不了显示页面返回空白，4，/function/xxx/的形式也长。

第一个问题，我们得去了解faasd的机制，在安装步骤的cd faasd src,git checkout 0.9.2,faasd install时，它利用一个faasd bin启动了二个过程，faasd up和faasd provider，ps aux|grep faasd可看到二进程的路径，netstat -tlp | grep -i faasd或lsof -i:8080可看到端口（注意到它们都绑定在了ipv6），这里的逻辑如下：

> faasd deploys OpenFaaS along with its core logic faasd-provider, which uses containerd to start, stop, and manage functions.The faasd process will proxy any incoming requests on HTTP port 8080 to the OpenFaaS gateway container。
>
> 解析一下：云函数首先是个容器集群，faasd维护一个单机容器集群和一个内部网络(这种东西在《利用colinux制作tinycolinx，在ecs上打造server farm和vps iaas环境代替docker》一文中我们提到单机构造集群的例子，多个vm要重用80，再想象一下，虚拟机用得多吧，也要接确到与宿主的多种网络出入逻辑和子网逻辑。在《openaccess》一文中也讲到过,openaccess虚拟局域网，这些都是虚拟网络的例子。只不过这次是管理单机上（或集群）上的容器集。）多个容器pod也要network，这个用linux上的iptables转发就可以办到，但容器为了功能强大往大了做，它使用了cni。体现在openfaas端（/etc/cni/net.d），就是它在本地网卡和虚拟的cni openfaas网卡之间转发实体间占用虚拟网卡的IP空间，faasd自己维护一个这样带自我解析的机制从而做到不依赖于dns，它们都cat /var/lib/faasd/hosts和cat /var/lib/faasd/docker-composer.yml中可以看到定义。
>
> 我们发现，到最终应用，整个serverless机制，它就是faasd+cni+一堆容器堆出来的，gateway是容器，在10.62.0.1网关(这个由cni做出来的openfaas0网卡)，watchdog也是容器。app也是容器。只是hosts中没有app的10.62.0.xx的ip地址。ps aux | grep onemanager可以看到它是一个containerd-shim-runc-v2容器进程，每次重启后这个docker本身有一个ip 10.62的ip(journalctl -u faasd --lines 40，出现onemanagerforopenfaas has IP: 10.62.0.xxx获得)，每次deploy后也会换，你会发现每打开一个链接，lsof -i:8080都会增加几个进程，这就是openfaas的云函数机制，它是按需为访问活动中的函数（从后端容器进程中提取,我们知道，容器运行时即容器的实现方法，现在有很多种，不光docker。containerd也是）生成进程的，等到这些访问活动没了，它就自动消失，没有一个像httpd daemon的守护（这是延时的由来原因之一），faasd负责管理所有这一切。

无论如何，作为serverless，faasd这样做的好处是：（这样做的意义，在于最大利用单机上的资源，而且以可伸缩的方式，所以同样可以有集群方案），且你可以用为语言贡献库的方式去开发远程服务性APP。因为它们是直接php index.php或go index.go弄起来的 Any binary can become a function through the use of watchdog.，开发接上了部署都在API级（带容器集群开发/部署级别的faas给devops带来的功效相当于另一种带visual ide as devops的集成appstack环境）。而且这种非http方式提高了利用率。不过单机上的应用集群局限于单机的最大能力，而集群上的应用则不会。

解决问题1，2

来看第一个问题，我们猜它是发现不了docker-composer.yml中http://faasd-provider:8081中的faasd-provider对应的ip，于是我们把yml中的地址替换成10.62.01。问题1，解决。但随后经常出现“service not reachable for onemanagerforopenfaas”，（但这个其实访问很流畅：http://m.shalol.com:8081/system/function/onemanagerforopenfaas?namespace=-），于是我们查看日志：sudo journalctl -u faasd-provider --lines 40，发现error with proxy request to: http://10.62.0.xxx:8080/，这就是containerd的原始进程经过gateway代理的过程中出现的。我们要把它弄成ipv4的listen,如何在ipv4上启动faasd试试：

在ubuntu上关掉ipv6，我尝试过sysctl关掉所有网卡，禁掉/etc/hosts所有[::]:等方案都不行，最后etc/default/grub中修改GRUB_CMDLINE_LINUX_DEFAULT="quiet spalsh"为GRUB_CMDLINE_LINUX_DEFAULT="ipv6.disable=1 quiet splash"然后update-grub重启成功。再次netstat发现faasd进程都被forced to use ipv4了(不过听说faasd升到0.9.5，源码中有ip修改的地方，如果不是默认的空或者0或者0.0.0.0，那么都可以成功在IPv4下建立侦听。)。问题2延时减少了很多，（无论是直接访问原始容器还是代理到gateway的8080）“service not reachable for onemanagerforopenfaas”出现的机率都减少了很多(进入系统多等，或要访问应用至少一次，faasd-provider才会启动,这是40秒docker自定义网络延时？)。

但是nginx却启动不了了，考虑到禁用了ipv6,于是我把/etc/nginx/conf.d/default.conf中listen [::]:都注释掉了，但还是启动不了（网上说的那些ipv6 on的方法也不管用），我的是ubuntu18.04上的nginx 10.14.2，这是因为跟faasd一样，Nginx is doing DNS resolution at startup by default，在ubuntu上，With current default vhost listening on IPV6, nginx won't start on fully IPV6 disabled system, expecting to connect to a IPV6 socket on port 80.但是在其它发行版Other distributions does that (not including ipv6 listen by default) already (e.g. Centos and Upstream)，不过This bug was fixed in the package nginx - 1.17.5-0ubuntu1，于是

```
sudo touch /etc/apt/sources.list.d/nginx.list
deb [arch=amd64] http://nginx.org/packages/mainline/ubuntu/ bionic nginx
deb-src http://nginx.org/packages/mainline/ubuntu/ bionic nginx
wget http://nginx.org/keys/nginx_signing.key
sudo apt-key add nginx_signing.key
```

```
sudo apt update
sudo apt remove nginx nginx-common nginx-full nginx-core
如果apt包管理器询问你是否要安装新版本的/etc/nginx/nginx.conf文件，则可以回答否，以保留原来的注释掉了[::]:的config.
```

成功。

关于延时，watch-dog有一个特化版，可以将一般进程转为长驻留的http，这种方式下Keep function process warm for lower latency / caching / persistent connections through using HTTP，延时不会因为频繁经过网关开进程导致开销(否则你需要定时访问触发以触发网关warm up这个function)，而且它支持webframework，因为里面可以内嵌一个webserver(虚拟机管理面板中，为什么php cgi很容易跟nginx组合，配置成熟，很多细节可以配置，而python项目管理就相对少一些。因为php cgi自带了服务进程管理相当于这里的watchdog作为外部，框架放在内部源码部分仅涉及到业务逻辑不涉及到服务管理，内外分开，而python这样的没有cgi这样的东西，框架管理进程，直接在里面开服务。这种细节在外部与nginx组合的时候显得难于配置)，webframework可以允许你定制一次请求响应和url router，允许你用类似js的event,content方式开发。使用这种watch-dog，需要：

```
FROM openfaas/of-watchdog:0.7.7 as watchdog
最下面ENV fprocess="php index.php"之后加：
ENV mode="http"
ENV upstream_url="http://127.0.0.1:5000" (框架服务器透出的转发地址，转到watchdog:8080)
```

由于上述方案适合这种框架中自带服务器的，onemanager并不适用，所以我没尝试。它还支持一种mode=static，想到pai中那个static:public了吗？在支持statcisite方面，上次说到pai需要手动一次，因为它使用的是devops的一个部件git，而没有使用到devops的主要部件docker。这里用了云函数这种方案，就纯自动了。

解决问题3

你会发现地址不对了。这时在common.php中作以下替换： $_SERVER['base_disk_path'] . '/' . $path $_SERVER['base_disk_path'] . '/' . 'function/onemanagerforopenfaas/' . $path 有大约七处。

对于那个文件下载不了了的问题显示空白，curl_setopt是common.php中onemanager向上面的api要求结果的函数。影响着向下面的客户返回的结果。你可以在本地模拟curl -I 正常的om的下载返回结果，修改common.php以让整个程序也返回同样的结果。

另外发现一个可在common.php中markdown替换网址，以实现在域名子目录里放md网站效果的功能，(如果不这样，则你需要在md中定制带此目录的域名根路径)。这个其实跟domainforproxy一样可以做成后台配置项。

```
因为我使用了parsedown，你也可以在原始的$headmd输入中替换：
            $headmd = str_replace('<!--HeadmdContent-->', $parser->text(str_replace('/p/','/minlearnprogramming/p/',fetch_file
s(spurlencode(path_format(urldecode($path) . '/readme.md'),'/'))['content']['body'])), $tmp[0]);
```

被一个腾讯cvm的问题折腾了好久，别人能正常ssh我的服务器，我在本机却不能(没有任何响应)。web ssh下，别人和我都不能wget，提示no response。解决办法是去管理台切换一下172.16.0.xxx形式的私网地址。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一个**netdisk storage backend app webos**和增强的全功能网站云设想

本文关键字：利用网盘空间**,network filesystem**代替静态网站空间,做成静态网站的动态模块,利用**v2ray,nginx**给**onedrive+onemanager**做自动**cdn,**利用网盘代替函数计算

在前面《利用大容量网盘onedrive配合公有云做你的nas及做站》我们说到用网盘空间达成网站云和用网盘做附件床，一般这样的云网站方案中，静态html都是用oss做的空间。但其实配合oneindex等程序，网盘也可以存放md资源。直接in memeory servering md pages成html，这样网盘+云函数空间或ecs，就不仅可作为附件床了，甚至可以作为静态网页生成器空间。达到全功能网站云的效用（配合下面的内容加速甚至可以代替cdn）。

包括上文在内，我们所有的努力是让网站app和网站资源，聚合进一个网盘 as nas中，网盘网站通过客户端挂载后，可以本地文件形式一个一个编辑和同步，也可以直接在远程编辑md，类似mediawiki和gitpages在线编辑。网页和附件可以统一打包备份。服务于你的中心nas库建设(onedrive as nas)，一套下来可谓省事不少。

利用网盘直接展示和写作静态网站，自建静态展示空间

依然用前文的onemanger来举例，我们可以把静态网站的md文件按树形层次文件夹组织好，文件夹名即为文档标题（当然可以稍为处理一下，标题中不要出现特殊符号），然后一个文档文件夹中放一个head.md，它里面放主要文档内容因为它主要是在页面前面显示的。如果你的文档文件夹只有此内容部分，那么接下来显示的就是一栏只有一个head.md文件的附件列表，如果你的文档文件夹中还有其它下载资源和资源需要说明，可以在这里上传（或通过网盘上传）再放一个read.md，read.md效果会出现在页面下面，，如果是非文档文件夹和非文档子文件夹，就光放下载资源就可以了，一个页面除了文档部分就是附件列表，附件列表就相当于做成静态网站的动态模块（评论模块处）。整站主页嘛，就在网盘根下放一个head.md。

这样下来，一个利用网盘展示markdown网站文档的效果就达成了。

你可能需要定制一下把head.md和readme.md隐藏这样效果更好，在后台设置就好了。Pagenatior可能也需要处理一下更美观，—— 如果一文件夹下有太多下载资源或文档子文件夹的话。你当然还可以再定义主题风格。

网站可能还需要一些强化，因为纯粹依赖网盘没有本地部分，可能网盘一旦挂了，网站就空了，所以需要提供虚拟目录的功能，即网盘程序会挂载本地目录作为跟外来网盘同级的文件夹展示，你可以在这里放一个由markdown组成的备份网站（或者指定部分文件夹自动从网盘缓存到本地servering，如markdown网站所在文件夹然后挂载，后台有某个开关切换展示本地缓存的网站。）。如https://github.com/reruin/sharelist支持虚拟目录。可以本地和盘内共享一个域名。

内容加速，自备**cdn**

onedrive官方的外链速度由于在港区和新加坡，跟轻量机的带宽一样，也分闲时忙时，深夜凌晨快，白天很慢。那么有没有办法为网站加速吗，如果你托管onemanager的云主机是港区轻量再好不过，流量充足30M带宽。虽然转发是建立在轻量线路其实也分闲时忙时的提前下，但假设轻量线路比onedrive线路好，至少可以一试。

用二种方案，在服务器nginx加速，在客户端v2ray加速。服务端加速就是通过利用Nginx反代加速Oneindex，让Onedrive流量通过服务器中转，解决Onedrive在线播放视频下载慢等的问题。Nginx处和Onemanager处理处(支持sharepoint.com的反代)都要设置一下，

客户端v2ray嘛。不用介绍了吧。话说，如果v2ray也像网盘api一样。可以通过编程方式给内容加速就好了。提供api就接入了编程。

利用网盘代替函数计算,直接托管云程序**cloud demo**，自建脚本空间

我们知道网盘自身没有计算能力，必须要搭配函数计算或云主机调用其开放api才能调用其中数据为建站所有或用于nas目的，但如果在网盘中托管的是非静态数据，而是程序或脚本。然后考虑通过搭配云函数或云主机去运行它呢？？vs 渲染静态md成html，这里是运行脚本，呈现结果界面。比如，可以用于展示自己收藏的脚本并让用户点击launch it看到运行结果。形成自己的online demo repo —— 类似我们之前的折腾《使用群晖作mineportalbox（2）：把webstation打造成snippter空间》。脚本形式放网盘，还可以一同参与备份（网站数据和代码一起备份）。

这种方案有点接近让网盘具备云函数能力，与jupyter和herku容器这样的东西重复。这些都是利用脚本语言的runtime做程序执行机制的变体技术，比如，云函数计算实际上是将语言 Runtime 本质上是一个 HTTP Server, 再为web建立api机制而已（api,zeromq，消息件,restful也可以）。—— 当然，接近归接近，其本质并非如此，我们始终用的是网盘的存储能力它本身并不具备运行能力。只是可以作为一种折腾方向，不过值得一试，比如通过在onemanager中增加相关功能达成。

————

最近转到腾讯云了。发现其优惠力度和实用性比阿里云还大。尤其是618或1111，1212时的新购1到2折，所有主机还是100%CPU和5M带宽，企业用户更是优惠，这实在比抠抠嗖嗖的某些云服务商要好，很适合做云桌面(阿里云只有一个港区云轻量还算物美价廉其线路还是分忙闲的)。你可以赶在618新购，然后下一个3年用家人的再新购，腾讯函数云网站做成小程序可以接入大量微信用户。。3年1500左右是我的理想投入，因为

一台手机3年就刚好报费。视云主机为托管实体机的成本考虑就很容易理解吧。618的onedrive也几乎半价，仅官方版，

要购买互联onedrive参照https://docs.microsoft.com/zh-cn/microsoft-365/admin/services-in-china/buy-or-try-subscriptions?view=o365-21vianet，也是导向去买官方onedrive的网站购买，https://www.microsoft.com/zh-cn/microsoft-365/compare-china-global-versions-microsoft-365，互联只有Microsoft 365商业版，它是 Office 365 的升级版,(单用户43一个月，包年36，由于疫情听说包年有6月免费)没有官方细分的个人版和家庭版,每个用户都是1t加很多附属服务（商业基础只有web office），我们知道，windows是使用域的。世纪互联和windows官方用的是windows的基础设施搭建的onedrive，使用的登录域完全不同,二套产品是分别开发的，api支持和客户端支持情况不一，因此在接下来创建帐号时域名位置就可以区别看到。互联使用的是xxx@xxx.partner.onmschina.cn。这跟去互联官方azure.cn创建帐买通用云资源和主机创建帐号一样的，不要淘宝买互联帐号，那些全是各种公司的域控下分出来的子帐号（虽然单用户有5t的）而且有各种局限有的没有api权限。淘宝没有行货的互联。除了onedrive计划，互联还有https://www.microsoft.com/zh-cn/microsoft-365/sharepoint/compare-sharepoint-plans这个sharepoint专门计划。72一个月云盘无限容量(实际上就是1到 5到25自提)还可以绑定自定义域控，和享有一个网站。

本人使用的是官方office365+阿里轻量，明年转sharepoint+8元一月的cloud ecs得了。

————

关注我

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一种设想：在网盘里**coding,debuging**，运行**linux rootfs**作全面**devops**及一种基于分离服务为**api**的融合**appstack**新分布式开发设想

本文关键字：**Jupyter visual debug**，基于网盘**backend**的**ide**和**snippter**空间**,debug driven programming**，**make chrome like visual debug for every lanaguage**，**make every language a dsl,c**系语言学习最好的时代

2020版的macbook要迁移到arm了。这说明，云时代去终端的thin化努力在继续，未来我们的PC和手机会统一(请注意我指的并非是笔记本平板一体机，变形本这些玩意层面上的意思，而是架构和生态的变迁)，而云端始终配合终端变化，走的却是性能不断加强的fat化路子，不光硬件如此，《利用大容量网盘onedrive配合公有云做你的nas及做站》我们已经知道，云开发时代，本地的存储和带宽已经没有了任何优势，有一些资源是本地化不可能得到的，它们只存在于云端。群晖这样的nas跟onemanager+onedrive这样的组合比已经没有了太多优势，会越来越衰败。越来越多的本地的东西迁移到云上，成本也会越来越低，这是软硬一条龙上的变革，最终应用也会越来越倾向利用云计算，这所谓的一条龙，包括os,开发系统，也包括组成应用的各stacks本身，当然也就包括代码托管和开发方式的云化甚至软件的定义（跨多语言组件交互已成必要，中间件和service api原来分布式开发元素上面也有新变化），见下：

jupyter for od，让文学编程与网盘后端结合，,just another devops alternatives to cloudwall and github ci/di

首先是代码托管和云IDE方面：在《群晖打造snippter空间》等文章中，我们从云收藏APP的角度讨论了用网盘后端来保存snippter这种特殊内容的方法，就像收藏视频，收藏网页clipper一样，收藏code snippter其实也可以像跟收藏文本一样简单，但相应地，vs 视频之于视步转码云服务md文件之于markdown rendering，对于codesnippter也可以加入复杂的云上再处理（srvside or browser/client side格式化渲染显示,ssr），甚至，更进一步，将它与复杂的netdisk storage backended devops和工程文件组织环境，ci环境，云ide构建环境（云Ide和开发的极大化，就是devops)，。建立起关系，

在《群晖gitlab+docker打造devops》《jupyter设想：enginx,engitor,passone》《cloudwall》当中，我们都介绍过devops:一种将开发上云的手段，这三者都有某种程度的devops支持/都使用某种storage/都建立了某种appstack/都支持一种多种语言，比如，群晖上gitlab是用本地存储作storage,jupyter也是，不过它是一种文学编程工具(librette programming允许我们使用更丰富的文档与注释输助代码)，而且它的devops功能和云IDE很强（最近，jupyter又强化了它的visual debuger功能，使用xeus-python甚至可以可视化高级数据结构和内存对象树,比我们介绍过的《elmlang》这些云可视debug ide还要深层和完善,我们知道elmlang 等visual debug的作用主要是make chrome f12 like visual debug for every language，make every language a visual trigger/action dsl，就像chrome仅能作客端js调试的道理一样，它能使任何编程变得像chrome visual debug和接近web开发调试方式），cloudwall则用couchdb，couchdb不仅是对象数据库还是http应用服务器，由它组成cloudwall这种很内敛却selfly containing gui/http/storage的appstack，它虽然仅支持js(但却可以视js同时为数据和代码，天然可以在线coding和debuging同时mateapp方式客端即服端方式运作 --- 可以说cloudwall是我们见过的第一个完备的mateapp和mateos环境)，jupyter可以多语言，它的功能核心是各种protocols,appstack为类似普通lnmp的web page app(.ipynb),而gitlab，github这种，只有devops中中部分ci/cd功能(github actions,etc..gitpage generation)，离线客户端也是重要的部分，却没有鲜明的在线IDE支持这些方面，线上部分主要为在线hosting代码版本，appstack不限。

但如果能将这种jupyter ide和living coding,debuging做到网盘里，以网盘为存储后端，代替github等coding hosting空间，情况也计会是另一种美好，比如ipynb存在网盘里作为托管codesnipter和工程组织文件，可以直接在网盘里运行语言和部署应用，结合jupyter的visual debug，我们可以在chrome里调用装配了服务端jupyter支持的情况下，我们可以利用服务端rendering，哦不，服务端debuging。这样，网盘后端的jupyter实际承担了，后端language baas，存储，coding,debuging等在线开发全支持。---- 所以，它十分类似cloudwall和github的整体或部分devops相关作用。比如，它还可以用jupyter-book这样的应用达成gitpages,届时，我们甚至可以直接在前端做站,在前端写内容，在前端直接开发调试后端(我们可以将其做成，为每一个app装配一个livedebug，Jupyter backed debugger inside app,比如正常情况下显示程序本身，press esc 会消隐切换到一个visual debug环境)。markdown也直接写样式统一html，我们知道md可以代替html用，这样就有了三个统一，开发调试统一，前后端统一，内容样式又统一。。

利用云**booter**，在网盘里运行**os,**作全面**devops**

实际上，我们知道，除了语言后端和devops工具，更强大全面的devops需要docker或虚拟机来参与构建，虚拟机和docker始终显得太重，在《一键pebuilder安装deepin20中》我们提到一句后话：为了mate os下的开发，我们还准备了可视开发和网盘内devops支持，这就是整个文档集《minlearnprogrammingv2》part1提出的一种云booter：它更轻量，工作原理类似系统级的虚拟机，在boot和firmware级集成虚拟机支持，自带Linux kexec protocol，可以在启动booter时启动多个资源允许内的rootfs as os container content,使得app和os,subos一个性质都是rootfs。

ps:如果说在网盘里运行jupyter是对minlearnprogrammingv1 时代engitor概念体的强化，那么新的pebooter则是对v1时代diskbios的概念体强化（目前我们仅做到网盘云装机）。之前我们在文章中研究了 colinux,openvz,lxd等虚拟和多开方案并企图covering all硬件平台包括实机，但我们现在正式转到云boot上来并局限在仅云主机，天然多os和多开类colinux的rootfs级虚拟化。更适合云主机。

最后，当云开发，云devops,云代码托管都被做在了云上，再加上这里的云os，这所有加起来的好处是，我们可能并不需要一台真正的pc作开发和应用，不光内容同步可以整包打走。，甚至可以真正做到去PC戒电脑，这个clientless仅指去x86 pc化(比如我们可以选择更省能的arm平板加一台x86云上装有mateos和上面jupyter,云booter的服务器作mate pc。实现去pc化)。

甚至，当这一切发生变化的时候，云化开发方式和appstack也要经过云化：

我们知道，较之cloudwall，其单一性也是其自身的优势也同样是其明显局限，jupyter支持的多语言加网盘后端模拟的cloudwall更符合通用的情形，但是要达到用jupyter精确模拟cloudwall式mateapp的效果，需要更多额外的工作。

其中之一是需要统一的api定义和调用方式。

matecloud virtual appliance:一种统一web appdev和native appdev,基于api分离及服务组件化的融合分布式appstack

在《hyperkit:一个full codeable,full dev support的devops及cloud appmodel》中我们提到“一种可能行得通的appmodel设想：P2p 客服同体，只需sync api结果即可的app”，在《利用开发tcp思路来开发web》，我们提到利用wpcore as service，在《利用大容量网盘onedrive配合公有云做你的nas及做站》中我们见识到腾讯云函数serveless,这其实就是以前组件技术和远程方法调用，及webservices，serverless只不过倾向更多指免运维的极大化。我们还发现reactive page/pwa这种js库，缓存页面在客户端，反应很快。类桌面效果。因为它是使用web api的，前后端通过设置api分离。所以客户端html和静态资源可以缓存。

PS:甚至于我们上面提到的这种c++ xeus debug protocols，它实际上就是api（应用处）和语言接口（语言处）的一种类似体，我们知道，web天然就是服务化分布式的，web编程的实质在于service化，和service调用，面向产生和使用各种service api，这是一种与本地api开发和分发完全不一样的形式，是不同内存模式的多语言处理异构系统交互的方式，涉及到多语言和调用组件定义。然而我们现在的web是将html与后端逻辑形成一种组成app的stack，这样，实际上导致了界面和业务逻辑不可分，也导致了云服务应用中，客服cs/bs不可分，失去了streaming html as page gui的好处 ---- 真正正确的做法，应该是仅把html分离作为gui content only不集成到appstack as gui，采用js/reactive的做法，后端与前端完全通过service api分离，后端定义出业务逻辑api用一个rpc表达出各种外部服务接口，供前端（它只应是一个js/html reactive客户端的程序）使用，web api与web服务，使前后端真正分离,将界面渲染全面放到客户端reactive pages，没有任何服务端内嵌html渲染html的逻辑。整个应用用传统web的mvc来解决，比如onemanager的类比物：cloudreve就用了这种技术，如果onemanager来采用，展示速度又会提升很多。

这种分离客服reactive page纯客户端渲染方式，像极了native app，而本地开发其实也可以用这种方法。为前端调用定义出api。后端透出api，又可将本地开发和远程开发统一一个模型：比如实际上deepin的qt+go backend的deepin ui appstack也是这样的思路。除了界面，存储也可以以不掺入任何界面渲染的方式进行，仅是另外服务区导入进来的api。这就是各端分离。云函数纯api化，单独开发调用，通过rpc交互的思路，又迎合现在跨多语言开发的方式。

如果结合上面的mateos cloud boot技术，可以将每个这种app欠嵌一个vm，做成virtual appliance。只有web真正跨三端如果所有界面都用js/reactive呈现可以达到原生渲染的效率，这样就模拟了go，且不再局于Go和 go app 是最好的virtual applicane选型的说法.

本地app和远程app融合的最高境界是microservice app，即不光数据解决了sync，程序逻辑本身也一体化。这其实要跟mateos相协作。见打造一个全融合的云《一种追求高度融合，包容软硬方案的云主机集群，云OS和云APP的架构全设计》。

本文注定晦涩难懂，这是一种需要操作系统改革协作的工程，所以为此我们提出了mateos和直到mateapps的全设想，一个建立virtual applicance支持，一个建立appstack支持，相互绝配。这种分离也为编程教育划分了新的实践方向，我们会另写一篇《实践即工程与反工程》作序言放在《软件即抽象》下。因为实践所需要的工程规模选型需要同时涵盖本地开发和web开发域，搞懂这里的技术本质和经过几个大小这种工程，就可以由纯学习上升到参加工作所需能力的实践量了。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

一种设想：打造小程序版本公号和自托管的公号，将你的网站/**blog**做到微信/微信公号里且与**PC**端合一

本文关键字：打造小程序版本公号，打造微网站版本公号，**,**从私有云到私流，微信移动生态，微信开发者号注册，将你的网站做到微信，微信**blog**微网站/小程序**,**将认证后端做在网盘中**,**，将你的网站小程序化**weapp**化**,wxamp**化**,cloudfunctions**与**miniprogram**的绝配

在《mineportal：个人云帐号云资源利用好习惯及实现》中我们提到，id和用户身份是统一产品和运营生态的开始,而opensocial id之类的sns不争气统一不了标准。手机上也没有google id，appleid，移动手机号这样的入口标准，因此这项任务被拥有海量用户的app所占据，比如微信，微信有海量用户，,由于微信客端自带浏览器所以也可当微网站客户端用（只是从微信跳出或跳入），所以它将其分给能起运营作用的用户app或网站调用(除了用户接入服务还有其它，比如支付服务)，，让运营做到每个人的手机里，与以前的pc网站和pc网站运营是对等的。只不过这次这些转移到了手机上而已：移动端对应的是微网站和寄宿的自媒体号。比如微信公号。，这类产品还有开放平台。

微信/微信公号和开发者号：一个产品，一种身份

来看下这个“微信公号”，微信公号实际上是腾讯利用自己的资源自己运营的微信”内挂小程序/微网站”，用户只需提供内容，当然，单有一个发布内容的地方是不够的，公号下有很多默认功能扩展，还允许利用开放平台对公号进行功能扩展，微信公号自带很多扩展，除此之外，用户还能建立自己的微信小程序。用户开发的小程序是外挂式的(下拉显示在微信头部)，由于微信小程序/公号小程序/用户小程序/内欠到微信的微网站这四者共享同样的微信环境和微信运营资源。微信微网站/app/公号开发的基础和原理/技术实现/功能作用范围也是一样的，实际上都是一样的。比如你也可以发明一个更高级的留言程序创造相似的功能代替公号内官方那个留言，或一个微商城级别的程序或者租第三方微信服务商城里的服务仅需要一个ticket。而政策上，腾讯当然会限制或监管使用这些API的主体，比如用户程序或网站毕竟来自非微信的外部逻辑和安全上必需这样去做。面向开放的是个人还是企业,还要视api的重要度也要去设层保护，比如为了保护内部支付服务调用仅对企业开放这是必要的(而共享他们的用户身份oauth可以给任何个人)，(我们知道做站注册域名，工信部都需要监管一层)。

所以，腾讯还设立了微信开放平台和微信开发者号，辅助用来对微信小程序和微信公号开发和统一管理它们的API权限。个人可以注册订阅号和注册小程序号却不允许注册服务公号和企业订阅号也不能注册微信开发者号的。必须至少要用一个个体户或企业身份，PC网站移动应用也只能在开发者号里找到。在开放平台里你甚至可以将小程序授权给小程序开发商，只需获得一个票据就可以使用它们关于小程序的所有功能和托管无须开发纯买卖。（即使可以，用户小程序用户公号的权限和企业的也有区别，申完开发者号后，可以在里面申请开放每个小程序或微网站的api权限，按重要度，获取用户的基本信息这类基础API，对企业和个人的APP和公号来说都无须认证，不收费。涉及到支付等关键领域，还是会要求法人认证的，而法人有更高权限，如开发者版绑定的小程序较个人版本也有更高级权限支持js网页里跳转。），而且开放平台可以汇集帐号下注册的移动应用、网站应用、公众号及其他小程序，关联到开发者号后，将这些通过unionID机制统一用户账号。如果你不用到进阶API或统一uid，公号小程序和公号本身足于满足你对接海量微信用户的需求不必开开发者号。而为了对接独立微网站或PC网站，需要开发者号。 ------ 这样就有2号：微信号，开发者号，(这个还在不断扩展，如商户号)，三件可开发：小程序，移动应用接入/pc网站接入/微网站扩展/公众号、订阅号、服务号扩展，前者是号和主体后者是产品，我们最常用到小程序和pc网站，，注意区别。

> 注册开发者号的经验：如果是个人，不建议专门为了这个而去办个体户，因为维护成本较高且涉及到征信问题，在某宝上找个个体户挂靠，虽然也有风险，但至多损失的是短期临时的投资（找那种省事提供企业身份且无须年年交认证费和年审费的。现在虽然取消了开发者号年审但也会抽查，有些卖家会以腾讯会定期抽查要求你每年都交被认证方手续费和认证方年审手续费他们将维护一个个体的费用按人数分担到这些用户:腾讯会雇佣第三方来认证无论成功失败都收还很贵300一次不过别担心它有打回重填）如果你找到的比较有信誉，你至少可以用上一年+。分四步，第一步你可以填自己的管理微信（其中的信息只有你的微信认证后才能更改），第二步会涉及到法人主体信息电话回访人脸验证填写，第三（不开票），第四付300初次年审。以后审请网站应用认证的时候，也要用到法人身份和实名域名，还需要公章（个体户没有公章在公章处签名），和网址与主体保持一致,之后可改授权回调域域名无须再审，政策每年都在变，成功可获得appid和secret。审核小程序的时候也要注意，像博客这种小程序应该选择教育 > 教育信息服务

一个更自由的公号环境，小程序还是微网站好

只是由于公号实际上定位于媒体，所以它发表的每一篇都是不可修改的且每天一篇（个人订阅号）即使企业订阅号服务号也是有限制的（它有一个好处是可以开通流量主，再运营），---- 所以不断后台开公号插件和直接利用公号运营，是十分受限的。所以有没有方法打造能够存在于微信里的blog程序，产生和公号效果一样但无限制编辑发送的写文章后台，和打开文章不提醒外域跳转，能被微信用户关注，和能推送到文章和消息到关注用户微信的功能呢（为避免频繁消息可以设置一个可选推送开关）？（PS：以上这些功能缺一不可，都被做成了api，缺一则废。比如少了推送给用户。即使你的内容再精美，用户也不能被动成为你的受众。）最重要的：用户，我们能不能还像公号一样，多平台维护却始终能做到只维护一份共同的用户呢，

要打造自己的微网站托管公号，我们可以配合小程序和公号模拟现在公号已有的全部功能，那些在公号菜单里欠入微网站还能用微信一键登录的。就是服务号或公众号(个人订阅号只能把小程序当微网站用跳一次到小程序界面，或者文章里阅读原文达到)，而且可以实现一样的oauth无感登录注册。但是不能消息推送到微信（因为它来自外部网站）。这种情况下公共号只是一个传手，我们需要托管自己的微网站。这种公号实际是就是一个pc网站blog端的shell，而且微网站的用户和你后端对接到的用户是二份。不过它胜在体验自然。却输在以上方面，且需要法人认证。微网站往往不用开发。难度低。如果对公号开发，又维护了二份成本。

另外一种方法是用微信里的小程序。个人可用，小程序直接寄宿在微信头部，虽然可以欠入公号，但是需要跳出公号。这种体验终归有些不自然。但小程序扩展能力强，小程序可以基于任何目的比如它可以代替公号发布做成一个blog或cms，也可以是一个社区，也可以是其他各种需求的扩展，因为它是个app业务逻辑可定制，（当然，发布的时候，审核会让你选择一个用途。）但小程序往往需要自己开发，（小程序开发加入了太多新的不同于传统网页技术的东西，比如有自己的css和html标准），同样可以微信无感登录，且可与外面的网站后端共享后端，优点是可以维护同一份用户（在unionid的PC网站和小程序下）。

再来谈用户，上面二方案中，我们实际上都逃开了公号的限制而依然利用上了微信的运营资源：用户。我们都维护了一个共享后端，常见的微信扫码登录有几种,用户在微信内使用微信授权登录需公众号接口配置，用户在小程序使用微信授权登录需小程序配置，用户在PC的网页使用微信扫码登录需PC网站配置,用户还可以微信扫小程序码实现网页端登录，（实际上有了开放平台和unionid，我们实现三端合一的针对用户的运营。）------ 其实，用户微信登录后，我们依然不能真正获得并管理这些用户（因为它们都是微信里的，需要转发到微信运营），或者需要进一步（我们只获取了微信用户的名字和openid）再运营把它接入到自己平台并让用户在这里交互，转化。将用户转化成你的私域流量。

故，如果追求功能的自然化和全面化。小程序还是胜过微网站。关键的：后端一切托管在我方，所以我们有机会做更多事情。

在开发上，微信里的上述产品就是web开发的那些套路，比如api后端化，异步调用那些，得租用云主机或云函数来托管你的代码(cloudfunctions与miniprogram的绝配:一个托管前端，一个托管后端,且云函数：在云端运行的代码，微信私有协议天然鉴权，开发者只需编写业务逻辑代码)，涉及到websocket和端口的必须云主机，小程序虽小也是一个前后端完整的app，如web一样本质是一个web或ws程序

> 这里的技术点在哪呢？ 所谓微信小程序，只是提供前端给腾讯托管。我们需要提供一个纯api后端或服务端渲染网页的的后端（前者是前后端在发布层真正分离，后者是普通动态网站的那种前后端源码层可分离）,前者是前后端完全分离，可能是发布层完全另外部署的gui。如htmlui for aria2，后者是狭义开发上的分离，仅限开发层接口前后端的分离处理。 对现有程序对接入微信api和做成小程序，往往前端需要重新开发，后端只须准备一些对接层和api后端化重写。如果我们的程序本来就是前后端分离的，那么极易转化成小程序版本。最理想的情况下，后端纯api化。不要涉及到用任何服务端逻辑渲染前端的逻辑，前后端只通过欠入html的一条服务url路径来交互。后端仅headless core,前端是cli或html page only。 在《用开发本地tcpip程序的思路开发webapp》中我们谈到过那种前后完全分离，用开发spa的思路写网站的方法，这种方式方法下，实际上就是上面提到用js开发三端app的技术，涉及到vue、react、angular等这些前后端分离框架库所涉及到的文案，与传统jq操作dom的js库写网站界面相比，（我们知道一种gui代表一种appstack，界面就代表一种app方案方式），它更强调用本发native gui和tcpip程序的传统思路来开发web app，微信mobile miniprogram。 当然。这些库也有serverside rendering版本。但依然实现了spa和用协议交互前后端的传统APPDEV方式。 微信有专门的小程序开发IDE。注意其调试功能。也是web开发的那些。

---

discuz Q是一种微网站，我们也可以将前面的onemanager像fodi一样，后端纯api化，前端处理一切渲染。前后端只通过欠入html的一条服务url路径来交互。如fodi一样。

onemanager可把原来的用户系统加入扫码系统，用户扫码即登录，转化为一个中心用户库（即第二套帐密系统，用户名是自动的，密码可以自己修改）。选择某个微信为管理员。仅有这个管理员可以看到后台。

甚至把私域用户信息做进网盘后端。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

打造小程序版本公号和自托管的公号(2)：利用**mini-blog**将你的**blog**做到微信

本文关键字：合并多个**cloudfuncions**为一个

在《打造小程序版本公号和自托管的公号(1)》中，我们谈到了建立blog小程序的一些理论基础和必要条件，现在，我们选择一个这类小程序的源码，来实践一下：

我们选择的是https://github.com/CavinCao/mini-blog/commit/7921a126122f4a980e1270475aeb22bb2d50c3d0，积分功能v2版本后面的一个修复版，为什么选mini blog，因为它小。该有的都有，不该有的也没有。很容易看清这类实践的基本套路和这类程序的技术手法，下载源码后打开微信开发IDE，扫码IDE左上角小程序（开发社区号）绑定微信，，用IDE导入正确路径下源码，appid填你注册过的小程序appid，因为miniblog使用的腾讯云cloudbase托管的云函数后端（当然miniblog还用了cb的数据库），在点击云开发按钮新建第一个数据库或云函数之前，它会自动帮你注册或绑定已有的一个腾讯云帐号，为了达到在已有帐号里操作的效果，请保证预先处理你绑定IDE用的微信，它背后的某个腾讯云号已有绑定该小程序作为登录方式之一，这样新建云函或数据库会自动在你已有的腾讯云的cloudbase下进行，如果你使用了自动生成的腾讯云号，那么会得到一个新帐号已绑有小程序登录但未绑定微信，需要二次登录（先微信后小程序）才能进入帐号，解决方法是在那个新帐号里绑定一次微信。

IDE打开，我们发现源码有cloudfunctions云函和miniprogram小程序，点云开发，新建一个clodubase环境包月免费自动续费，登录web版会发现用IDE建的是只读的而且与网页端申请的那个免费可以共存，。接下来就是miniblog的安装：有12个库（access_token,mini_posts,mini_comments,mini_posts_related,mini_config,mini_logs,mini_formids,mini_member,mini_sign_detail,mini_point_detail,mini_subcribute,mini_share_detail），和8个函数，安装12库比较简单，记得把权限全设为所有只读仅管理员和创造者修改，如果稍后你发现编出的小程序有各种BUG，那么可能是12库少安装了某个或某些，或漏了权限没改过来。然后在IDE里定位到正确的cb部署环境，在IDE文件浏览器下点击各个函数对应的文件夹，右击部署（包括安装node modules方式），一一给函数的版本管理->配置处添加Env:cb名的环境变量项，其中syncservice还需要提供内容同步的公号的AppId和AppSecret，adminservice还需要author:你的管理者微信的openid串（可在稍后调出小程序界面后切到/index/index/从ide的console中直接输出的信息复制得到），miniprogram只需改下utils/config.js里的env id到正确的cb名，还需要在IDE文件管理器中右击index/posts/detail打到命令行安装一个客端模块：npm install wxa-plugin-canvas --production (设置，项目设置处勾上使用npm模块)，至于源码用的各种templateid，由于是公共ID，所以不用修改。小程序前后端基本能一起工作起来了，执行一次synservices拉取一些测试文章（根据错误添加IP白名单）。

下面，我们来深度定制它：

将**8**个云函数合并

由于部署8个云函数带来了管理上的困难，也显得零散，所以我们把它合并成一个叫mini的云函数中去，思路是先在cloudfunctions下新建mini，把各xxxservices中的config.json，package.json合并去除重复部分放到这，package-lock.json可删掉，如以下修改：

config.json(json不允许注释用时需要去掉//后的内容,下同)

```
{
  "triggers": [
    {
      "name": "myTrigger",
      "type": "timer",
      "config": "0 30 8 * * * *"
    }
  ],
  "permissions": {
    "openapi": [
      "wxacode.getUnlimited",
      "templateMessage.send",
      "templateMessage.addTemplate",
      "templateMessage.deleteTemplate",
      "templateMessage.getTemplateList",
      "templateMessage.getTemplateLibraryById",
      "templateMessage.getTemplateLibraryList",
      "subscribeMessage.send",
      "subscribeMessage.getTemplateList",

      "security.msgSecCheck"   //注意这个
    ]
  }
}
```

package.json：

```
{
  "name": "mini",
```

```
  "version": "1.0.0",
  "runtime": "Nodejs8.9",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "date-utils": "^1.2.21",
    "request-promise": "^4.2.4",
    "towxml": "^2.1.4",
    "hydrogen-js-sdk": "^2.2.0",
    "request": "^2.88.0",
    "wx-server-sdk": "^1.8.2"   //注意这里，提到了最高版
  }
}
```

然后是重点整合部分，把各个index.js整成/mini下总的index.js（login/index.js已被如下融合，除外）：

```
//////以下从各个index.js头部整合到的全局变量
const cloud = require('wx-server-sdk')
cloud.init({ env: process.env.Env })
const rp = require('request-promise');
const dateUtils = require('date-utils')
const db = cloud.database()
const _ = db.command
const RELEASE_LOG_KEY = 'releaseLogKey'
const APPLY_TEMPLATE_ID = 'DI_AuJDmFXnNuME1vpX_hY2yw1pR6kFXPZ7ZAQ0uLOY'
//收到评论通知
const template = 'cwYd6eGpQ8y7xcVsYWuTSC-FAsAyv5KOAVGvjJIdI9Q'
const Towxml = require('towxml');
const towxml = new Towxml();
const COMMENT_TEMPLATE_ID = 'BxVtrR681icGxgVJOfJ8xdze6TsZiXdSmmUUXnd_9Zg'

//////以下整合的main,用了自建的route分发，实际上就是折分二个return：event,openid

exports.main = (event, context) => {
  console.log(event)
  console.log(context)

  //注释掉
  //return {
  //  event,
  //  openid: wxContext.OPENID,
  //  appid: wxContext.APPID,
  //  unionid: wxContext.UNIONID,
  //}

  //这句重要
  return router(event)

}

async function router(event) {

  //从main中移到这里
  const wxContext = cloud.getWXContext()

  //不需要了
  //if (event.action !== 'checkAuthor' && event.action !== 'getLabelList' && event.action !== 'getClassifyList' && event.actio
n !== 'getAdvertConfig') {
  //  let result = await checkAuthor(event)
  //  if (!result) {
  //    return false;
  //  }
  //}

  switch (event.action) {

    //postsservices
    case 'getPostsDetail': {
      return getPostsDetail(event)
    }
    case 'addPostComment': {
      return addPostComment(event)
    }
    case 'addPostChildComment': {
```

```
      return addPostChildComment(event)
    }
    case 'addPostCollection': {
      return addPostCollection(event)
    }
    case 'deletePostCollectionOrZan': {
      return deletePostCollectionOrZan(event)
    }
    case 'addPostZan': {
      return addPostZan(event)
    }
    case 'addPostQrCode': {
      return addPostQrCode(event)
    }
    case 'checkPostComment': {
      return checkPostComment(event)
    }

    //按葫芦造样，把memberservices,messageservices,adminservices,scheduleservices,syncservices,syncTokenservices其它case部分也弄到
这

    //注意这句
    case default : return {openid: wxContext.OPENID}
  }
}

  .....
  //再接下来就是从各个原xxxservice/index.js中移过来不包括开头变量和main()的代码的部分,直接复制即可。
```

然后就可以删掉所有xxxservices文件夹了，客户端修求主要集中在utils/api.js和app.js中。将它们改为全部callfunction("mini")，完工！！

将主题集成到首页

由于主题被做成了一个与文章首页和我的（后台）并列的项。不爽。将其归到首页“最新，热门，标签”后的“主题”文章部分。

index.wxml，从topics.wxml中提取到这里，介于搜索栏和文章列表之间

```
<!-- 专题列表 -->
<view class="cu-list menu card-menu margin-top" wx:for="{{classifyList}}" wx:key="idx" wx:for-index="idx" wx:for-item="item"
id="{{item._id}}" data-tname="{{item.value.classifyName}}" bindtap='openTopicPosts'>
    <view class="cu-item">
        <view class="content padding-tb-sm">
            <view>
                <text class="cuIcon-title text-orange "></text>
                {{item.value.classifyName}}
            </view>
            <view class="text-gray text-sm">
                {{item.value.classifyDesc}}
            </view>
        </view>
    </view>
</view>
```

由于展示是依靠js的，所以index.js:

```
data: {
  classifyList: [],
  posts: [],
  .....
  navItems: [{ name: '最新', index: 1 }, { name: '热门', index: 2 }, { name: '标签', index: 3 }, { name: '专题', index: 4 }],
  .....
}
.....

// 如下设置内容，避免UI块重叠
tabSelect: async function (e) {
......
  case 1, 2, 3: {
    that.setData({
      .....
      showHot: false,
      classifyList: [],
      showLabels: false,  //case3时为true
      posts: [],
      ......
```

```
    })
  .....
  }

  case 4: {
    that.setData({
      ......
      showHot: false,
      showLabels: false,
      classifyList: [],
      posts: [],
      .....
    })

    let task = that.getClassifyList()
    await task

    break
  }
}
```

把函数getClassifyList: async function ()从/topics/topics.js中移到这，可把topiclist文件夹也移过来到index/下，修改相关路径,之后可以删掉topics文件夹了，

你还可以修改一些硬编码的东西。

```
app.json:    "navigationBarTitleText":
syncservice/index.js:          totalComments: 0,//总的点评数 totalVisits: 100,//总的访问数 totalZans: 50,//总的点赞数
pages/mine/mine.js，申请vip的提示语等
```

一些未来改进建议：

后台管理，留言可以加一个是否开启审核，“程序有一点点小异常，操作失败啦',这类消息提示可以更精确。因为一般都是数据库权限或templateid问题。

templateid位置：

```
adminservice/index.js : APPLY_TEMPLATE_ID
postservices/index.js评论：COMMENT_TEMPLATE_ID
utils/config.js：subcributeTemplateId
messageservice/index.js, template 评论被回复模板ID
scheduleservice/index.js签到提醒：SIGN_TEMPLATE_ID
mine/sign/sign.js:tempalteId
pages/mine/mine.js申请vip: tempalteId
```

整个过程都是在IDE模拟器中不断的修改，调试，再调试。之后就是提交体验了，提交体验版本然后微信浏览基本接近以发布后效果而IDE中的模拟器较之有一些效果体现不出来，还有，审核的时候有可能会说你动态网站不能通过很奇葩，那么以cloudbase为后端意义何在？

下文我们将用fodi分开前后端的方式来划分onemanager将它做成miniblog小程序的内容源或者给miniblog直接弄一个PC网站前端和pc网站后端写作模块。其实小程序和onemanger非常像。因为后者也是云函数为后端，只不过一个定位网站前端，一个定位小程序前端。后端都是api服务。mini blog for wechat是直接利用cloudbase sdk调用云函，天然鉴权请求，而onemanager的前端是利用普通网站提交，基于url route提交请求。fodi前后端利用更高级的ajax。这三者都有相似之处,其实cloudbase也可用于google flatter和mobile apps。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

打造小程序版本公号和自托管的公号(3):为miniblog接入markdown和增强的一物一码

本文关键字：微信一物一码，一文一码，小程序码转为普通二维码

在《打造小程序版本公号和自托管的公号2》中我们讲到了miniblog的初级搭建和定制，这里继续：

加入markdown编辑

前面说到miniblog的8个函数中，前6个函数可以整合，剩下synservice,synctoken不必整合。可以丢弃或保留（保留使用它的意义不甚大，这种同步结果和过程比较曲折，体验不好，要使用它，记得syncservice/index.js/function syncWechatPosts(isUpdate) 中应该 let data = { currentOffset: 0, maxSyncCount: 由10改为1000 }，同时提醒用户把该函数超时时间设置大一点。 否则很难有正确结果。我120数据用了8秒。不要太频繁调用这个函数，免费额度会减少。也不要搞按时自动触发。这底下是因为那个函数内，调用公号素材库未发布文章一次只能取到20条，循环获取就是maxsynccount指定的最大结果）。丢弃的情况下,我们则需要依赖内部的发布文章机制，由于手机实在不好发文作富文本修饰或输入（除非chromebook,apple arm macbook真的广泛用起来了，人们用它办公和文字生产了），粘贴复制还是可以的，所以最好是markdown的那套。目前miniblog的那个编辑器可以发markdown前端也能通过toxml渲染,但是测试了一下，不能修改和更新文章。

首先，加入miniblog的编辑文章功能。mini-blog-d02344fe6c6b347057983edd1ef6b7450f88fbcc这个rev删掉了发文功能（审核问题删了。），去它的上个版本把文件下载回来修改相关wxml,js调用加进去。然后是把富文本编辑器定制成markdown：

```
miniprogram/pages/admin/article/article.js中:
savePost: async function (e) 函数内：
contentType: "markdown",   //这里由html改为markdown
content: res.text,  //这里由res.html改为text
getContent: function () 函数内：
resolve(res.text)

miniprogram/pages/detail/detail.js中：let content = app.towxml(postDetail.result.content, 'markdown');
```

小程序首页和文章详情也都是调用toxml的地方，实际上，小程序首页如果文章比较多下拉后返回上面很费事，你可以顺便做个添加一个回到顶端，wx.pageScrollTo({scrollTop: 0,duration: 300})。由于源码中toxml是放到src根下的，你还可以把toxml放进miniprogram/component文件夹内，这里是放小程序组件的地方（由于本地的js版本不一，编译miniprogram最好是本地进行，提取dist下生成的用。这也是前面文章提倡就地npm install编译的道理），移动后修改相关路径(component json和miniprogram/app.js中)。接下来会谈到把海报生成poster组件也放进来。

加入增强的一物一码

小程序实际上它也是web原理的page app，有各种路径和调用参数，(IDE左下角可以看到)，而二维码是一种编码解码机制，的背后要么是某段文字要么是某个网址，工作原理是经过图像扫描，编码解码得出码后的结果。为app所用。微信支持三种为小程序生成二维码的API，可以为小程序本身和小程序各种页面和带参数的页面调用生成二维码。即所谓支持一物一码，这里是文章，所以是一文一码。

miniblog中有一个海报生成，里面就有二维码生成，它被放在了前端文章详情页，miniblog用的是api中的getunlimit()，可以为/page/detail/detail?id=blogid形式的带参页面生成对应小程序码，如果有100篇文章可以生成100个二维码。这种api生成的二维码是小程序样式的。较普通方块二维码有些缺陷（比如它难扫，而且这种小程序码中间一个大大的logo很占位置），最重要的问题是：它背后没有一个https打头的url：微信后台对其一物一码的绑定url，虽然它是页面但其实它是小程序页面不是https，仅限该miniblog小程序内绑定，我们需要把mini/index.js/getunlimit()换成createqrcode（较getunlimit和get，这个能得到普通二维码。get和createqrcode各有5W多的限额共10够用）：

```
async function addPostQrCode(event) {
  //let scene = 'timestamp=' + event.timestamp;
  let result = await cloud.openapi.wxacode.createQRCode({
    //scene: event.postId,
    path: 'pages/detail/detail?id=' + event.postId,
    width: 280   //结果得到的从280到1280，我这里只要最小的
  })
  .....
}
```

这样就得到了普通二维码图片（中间LOGO，底下有一个提示语扫码使用小程序的提示）和带https的微信扫码绑定机制，我们把它叫作decodeqrcodeurl。得到这个，我们可以为其重新编码生成二维码，这样再处理之后的二维码有什么用呢，比如，用we-app-qrcode之类的组件再处理，不仅可以将提示语和LOGO去掉，而且可以用草料等工具定制美化。还可以可以压缩其体积，生成base64，供同一份内容的多次外发的文章脚部链接小程序二维码使用。真正让一物一码做到极致。

在mini/index.js中，我们加入重新解码的逻辑：

```
头部加入：
const decodeImage = require('jimp').read
const qrcodeReader = require('qrcode-reader')


async function addPostQrCode(event)前加入：
//解码二维码buffer
function qrDecode(data,callback){
  decodeImage(data,function(err,image){
      if(err){
          callback(false);
          return;
      }
      let decodeQR = new qrcodeReader();
      decodeQR.callback = function(errorWhenDecodeQR, result) {
          if (errorWhenDecodeQR) {
              callback(false);
              return;
          }
          if (!result){
              callback(false);
              return;
          }else{
              callback(result.result)
          }
      };
      decodeQR.decode(image.bitmap);
  });
}

async function addPostQrCode(event)中加入：
async function addPostQrCode(event) {
....
  if (result.errCode === 0) {

    qrDecode(result.buffer,function(urlcontent){
    console.log("decode:" + urlcontent);
  });
  .....
}

在mini下npm install -save jimp和qrcode-reader可以在package.json中加入：
 "dependencies": {
    .....
    "jimp": "^0.16.0",
    "qrcode-reader": "^1.0.4",
    .....
  }
```

miniprogram方面，文章详情页的分享小程序页面本来就存在，拉取（海报中的）二维码又涉及到服务端调用返回二维码费资源，干脆不交给用户，将它做成admin page内供管理员编辑文章处用更合适。也可以达到同样由用户来生成二维码的效果。还更可控，适用。比如，可以砍掉生成整个海报其它部分仅保留二维码生成部分，然后转移相关wxml,js函数调用，component定义（包括js中的初始data定义）到admin/article/articlelist页（不要放在article/article，批量操作不现实），按钮跟那些“展示”，“专题”，“标签”小红方块放一起，当还要定制主逻辑函数，具体如下：

```
//utils/api.js中， function getNewPostsList(page, filter, orderBy) {}和function getPostsList(page, filter, isShow, orderBy, lab
el) {}中，这二函数的_fields都加起：qrCode: true来，以便在articlelist.wxml中能：

<view class='cu-tag bg-red bg-{{item.qrCode == undefined?"red":"green"}} light' data-postqrcode="{{item.qrCode}}" data-postid=
"{{item._id}}" catchtap='onCreatePoster'>
  {{item.qrCode == undefined?"未生码":"已生码"}}
</view>

//然后是主函数：

onCreatePoster: async function (e) {
  let that = this;

  let postid = e.currentTarget.dataset.postid
  console.info(postid)
  let postqrcode = e.currentTarget.dataset.postqrcode
  console.info(postqrcode)
  let qrCode = await api.getReportQrCodeUrl(postqrcode);
  let qrCodeUrl = qrCode.fileList[0].tempFileURL
  console.info(qrCodeUrl)
  that.data.posterImageUrl = qrCodeUrl

  if (postqrcode == undefined || that.data.posterImageUrl == "") {
```

```
    let addReult = await api.addPostQrCode(postid)
    qrCodeUrl = addReult.result[0].tempFileURL
  } else {
    that.setData({
      posterImageUrl: qrCodeUrl,
      isShowPosterModal: true    //model是模态对话框意思
    })
    return;
  }

  let posterConfig = {
    width: 600,
    height: 600,
    backgroundColor: '#fff',
    debug: false
  }
  var blocks = []
  var texts = [];
  texts = [];
  var images = [
    {
      width: 560,
      height: 560,
      x: 20,
      y: 20,
      url: qrCodeUrl,//二维码的图
    }
  ];

  posterConfig.blocks = blocks;//海报内图片的外框
  posterConfig.texts = texts; //海报的文字
  posterConfig.images = images;

  that.setData({ posterConfig: posterConfig }, () => {
    Poster.create(true);    //生成海报图片
  });
  await that.onPullDownRefresh()

},

//你会发现列表是按createtime排的，utils/api.js中要改成timestamp，

if (orderBy == undefined || orderBy == "") {
  orderBy = "timestamp"
}

//然后utils/api.js中function getNewPostsList的检索条件和index.js中的检签逻辑(记得把总体文字缩短否则idx6之后显不出来，我是把已xxx全删了仅保留未xxx)也加起来。

if (filter.qrcoded == 1) {
  where.qrCode = _.nin(["", 0, undefined])
}

if (filter.qrcoded == 2) {
  where.qrCode = _.in(["", 0, undefined])
}

//如果你还想把生成的一物一码自动欠入各文章详情页内。，代替原来的onpostcreater()位置，那么可以
showQrcode2: async function (e) {
  let that = this;
  let qrCode = await api.getReportQrCodeUrl(that.data.post.qrCode);
  let qrCodeUrl = qrCode.fileList[0].tempFileURL
  wx.previewImage({
    urls:  [qrCodeUrl],
    current: "一文一码"
  })
},
```

整个过程依然都是在IDE模拟器中不断的修改，调试，再调试（提一下，我用的IDE是deepin20商店中https://github.com/cytle/wechat_web_devtools编译的，这个IDE腾讯官方没有列出为支持项）。要注意服务端的在函数后台看日志，IDE只能看出客户端的输出，由于上面是加入到服务端函数后端的，因此调试结果console.log("decode:" + urlcontent);在IDE端是看不到的（除非你打开IDE中的函数调用日志）。突然发现小程序用于实践很微粒化。一天一个微程序，像jupyter一样每天练习，持之以恒，可以积累下大量的语言和项目开发经验。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

除了**LINUX**，我们真的有可选的第二开源操作系统吗？

当你面临需要重新选择一门新操作系统的问题时，可能你是个不常使用PC常用手机的业外人士，即使正有一种力量正在劝导你弃用现在的windows或linux时，你也会对此不屑一顾，异或你只是一个对这个问题没有太多想法和爱憎的trickyer比如PC游戏轻度玩家你只是觉得所有的OS都一样好用所以不想发表太多看法，这在linux变得越来越好用的今天其实无可厚非，也或许之前的你曾经年轻过，在某个大学校园通往爱好hacking的路上无数次捣鼓过OS的原理和实现，那时你青春年少时间激情俱备一心只想发明自己的OS，但人到中年为了各种生计，你早已变得不屑为OS这种基础轮子进行重新选择的动力，或是浪费宝贵的生命？更何况你深知OS是一门极其复杂的产品和科学，W事用为上，你也觉得那并不十分值得。

而世界上就有这么一小撮人还在为最基础的事奋斗，伟大的战斗民族国民，俄罗斯的OS爱好者们，发明了reactos。曾经有一幅漫画疯刺他们（就是如下这个办公室讨论OS选型建议ROS的那个被踢下楼的那个）。但说实话，深思下来，这种漫画的恶毒之处在于他没有带来任何意义，反而细思极恐，如果reactos的发明者们的热情被他煽灭，世界上也许真少了一种十分伟大的作品，reactos的未来也许不会成真。。

而这，是一种莫大的无视和恶毒。。。当一种OS占据了世界百分比极大比例的时候，形成非一即二的垄断，那么它的存在，在战争年代的信息战和平时代的商业机密窃取国与国权谋中，如果这种力量被控制，无论这种力量背后代表的角色是谁，它对使用这种OS的人都是一种毁灭和伤害。唯一的办法，就是用另外一种OS。reactos意为反抗，它唱出了一种反垄断的正确声音，而这是尤其宝贵的，即使今天也如此。

政府大力发展工业和军用OS，同时也是民用即由于此，它们往往基于linux。未来的信息战，OS作为基建就像空气，当战争变得让空气充满了不安，那么种空气的失去就像战争期间士兵被断粮，这样的战争胜败不需10天。言重了。

———-

如果你以为这个世界为了不让OS成为垄断，只需要一到二种替代品，至少二种 – 那反而是对的。

那么对于WINDOWS我们现在即有linux系列，那么，除了linux，我们真的还有第二选择吗？没有！！一些不知名的OS基本没有份额，因为软件应有没有像windows和linux那样形成生态。

退一次讲，即使为了那个实用主义，选择对这个世界最基础的软件匮乏的最基本事实视而不见，linux真的就够用吗？真的就W用吗？linux能被称为好用吗？

linux基本在很多理论和基础上，都不及windows做得好。更不要说那些其它的OS了。只要没集成图形界面到kernel，其设计就是古怪的，内核没本地渲染层或，GUI反映就不够快捷。windows从初始版本开始的这些方面的设计理念和原则至少领先了业界20年。所以直到现在还在产生可供学习，实用和模仿的意义。

不要跟我说linux的安全，云计算的强大性，我承认的是linux的这些方面的强大，我遗憾的是我们缺少继续继承windows这些优点的新OS，这并不矛盾，这些linux没有做——只要这些没让最终用户形成策略上可用的东西并直观表达出来，那么它对大多数用户都是高冷的，大多用户并不需要灵活的命令行配置，不要说面向的用户不同，即使在服务器方面，图形化的产品外观facades依然是亲和的工具可以同时面对程序员配置管理者普通用户，说命令行可以更灵活配置更多更强大的逻辑，只是没有将图形的方便和内在发挥到最大。—- 我就是受不了linux配置生产部件需要一次次重复参数的调整而转到熟悉的windows的，虽然我理解它的每个细节但是我就是受不了它的散乱，它对用户没有经过好的打包。

好了我应该不要再说了，再说要变成类似编程语言优劣大战之类的宗教长论了，os之争曾经跟语言之争一样流行，但拜托，我需要的只是够用，省理的东西。这样就够，no more,but less will do。

而windows将这一切整合到恰当好处，对外给用户足够简单的使用接口，对内对配置用户和开发用户足够多的工具。——- 仅对用户提供策略，只在需要的时候向他们提供如何实现的机制，而这，永远不要过渡到最终用户。

reactos即是这样。它紧随WINDOWS的最初的OS理想，开源地模仿了WINDOWS珍贵的设计理念，将之公布于众 — 像windows的做法那样保留到现在。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

十年,最后一个**alpha,0.4.1**版的**reactos**终于变得可赏可玩了

从0.3.0到0.4.0，reactos花了十年。下一个版本据说就是beta了，这标志着reactos从此不再是个玩具了。

无论如何，reactos终于变得可赏可玩了，react提供live和boot完美地对应了需要一个类似winpe的维护环境，和需要一个实际安装到硬盘的真实环境的需要。

当然它要做终端，渲染图形或高精度稳定工业生产，是不行的。但是做一个爱好者定制的服务器系统，它还是十分有价值的，在这方面，linux都不是我的菜，毕竟它有开源的优势和另一个windows的美名，而不像linux那样天生设计成没有整合图形的版本和依赖高度配置对普通人显高冷的特性。

现在的它，完全可以像一个普通的OS产品外观那样工作，安装到实机硬盘啊，虚拟机测试啊，安装到云服务器（集成virtio）等等，联网啊，一般应用啊，安装各种软件和应用作扩展啊，我测试了下，利用常见图形分区工具这些给reactos分区，这些工具运行得并不好。所以将livereactos发展为winpe这样的东西需要找另外对应的能工作的版本和工具。开源，意味着，你以后不会再需要像利用winbuilder,nlite这样的工具slipstream驱动了，以前这些是外科手术式的，现在你完全可以在源码级进行。

但是主流可喜的现象是，大部分原win下的程序，像服务器程序，如xampp中的amp都可以无改地运行，第一次就成功。当然肯定表面之下掩盖bug，我猜也只是少量的.

但这样已经非常不错了。毕竟这是一个无工业投入的系统，而且服务器一般只运行有限的几个程序，够用就可以了。毕竟它还是真正属于得源者自己的。

也许像它未来beta版本和更以后版本说的，reactos版本推出会越来越快，因为里程碑式的难题已解决，接下来需要的是BUG的修复，和更多尾端配置工作的加入而已。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

能装机，能在无光驱的实机稳定启动的**reactos**版本

本文关键字**: reactos** 实机安装**,reactos** 实机测试

reactos 0.4.x发布并提供了livecd和bootcd二个版本，，可是都会出现不能在无光驱，实机启动/安装的问题（其实即使有光驱也会出现跟启动介质有关问题导致不能启动），由于livecd和bootcd中的setupldr.sys都支持从grldr 本机LLB bootstrap，所以对于无光驱启动，网上的解决方法一般是在linux下通过grub2来map 或map –mem到仿真光驱进行的（在windows下可结合winvblk这样的东西将光驱带入启动期）。

ps:grub支持file-backed仿真盘，和加了mem到内存的ramdisk盘，测试可用hd32之后的符号代表CDrom。如果不以mem方式加载cd，那么要求cd一定要连续。基本过程是：map –mem (hdr0)xx (hdv0)后，root(hdv0,0)显示虚拟后的C盘信息，再chainloader上面的启动文件（如果这个盘是直接可启动的，就直接chainloader整个盘符,比如(hdv0,0),,,这里r0代表虚拟前的镜像文件所在第一个盘符，v0代表虚拟后的第一个）。

且winvblk模拟的盘+grubdos模拟的光驱，能仿真到真实光驱的同时把它带给winpe。即把firadisk.ima或winvblk用map –mem (pd)firadisk.ima (fd0)这一句加载到虚拟软驱后，其实质是在PE中就可以看到用grub4dos创建的所有虚拟磁盘,我们可理解为这个friadisk给了PE系统一个启动时的注入的一个驱动，它将虚拟盘在启动时全部列出来，所以它与PE内启动机制中加载boot driver过程是绑定的。 可是，即使这样也并不意味着问题已解决，上面的方案看似能工作实际却不行。本文正是探讨一种能让能这二个iso在实机稳定启动，达到正常测试和安装的目的。

注意：以下涉及到的讲解和测试，以下探讨和测试在windows下完成，且均要求freeldr.ini所在的盘是FAT/FAT32.

livecd

livecd通过自带的菜单（grldr直接chainloader freeldr），或通过(grldr map –mem)，或grldr结合winvblk通过它带入，都会出现如问题：

这是目前的一个bug，见图：

——

IopCreateArcNames failed，大意是指不能分配一个盘符。当然这只是表象，那么导致问题发生的本质是什么呢？我们当然无时间从源码去归结原因，大致几种原因猜测如下：

启动介质光盘冲突？不能发现光驱？这应该是livecd不认识winvblk驱动？

网上有禁用floppy.sys，禁用/拔掉usb，置换uniata.sys，但都不行。

——-

可是所幸freeldr.sys原生支持ramdisk cd，它能在通过freeldr.sys启动时通过option将iso exportascd，加载到ramdisk到固定X：的方式。类似winpe。我测试了下，外置freeldr.sys+freeloadr.ini到fat32根，里面用ramdisk /exportascd选项，全部成功。

ps:原理是因为ramdisk能文件解压到了X盘。然后直接从x盘展开的文件系统中通过setupldr启动。ramdisk是与OS绑定的，用的是Os的loader+启动配置文件中的option选项，看起来与grldr模拟的ramdisk效果相同但实质不同，（能模拟ramdisk效果但不同比如不是X盘）.

bootcd

对于bootcd，通过实机或grub4dos仿真盘，bootcd会黑屏。

那么通过1类ramroslivecd的ramdisk方式行不行呢，不行，通过freeldr（指定freeldr.ini中参数到ramdisk(0)，注意freeldr.ini也支持systempath,kernel和hal参数），加载bootdriver过程黑屏，通过grldr+winvblk通常也不行。

ps:live系统和bootcd系统对于winvblk仿真盘的需求也不一样，这也是本文分开说这二块的原因，但其原理都是一样，就是类似0pe的那一套，可在kernel和boot driver和hal后，启动一个系统的子集。但是livecd明显将所有文件都在第一次启动时展开，bootcd就展开最小的，然后其它的靠后期安装到硬盘比里CD里面那个大cab文件。这导致二者处理方式不一样。

网上解决方法一般是在硬盘上开辟二个fat32,一个放安装镜像（即bootcd iso中提取出来的全部文件和另设的一个那个setupldr），一个放安装到的fat32 目标reactos系统分区。另外一个是生成img，再改造llb+loader，需要重新从外部做硬盘的安装镜像。这二种方法都有缺点，也破坏了源码生成iso的事实,特别后一种方法。依然是制造虚拟硬盘镜像，还要改造二段式boot过程。还是不够方便。

我还测试了下其它方案，比如利用grub+map –mem .img虚拟硬盘文件加载到hd31，但都有各种问题（路径是对的但不能加载hive，etc..）。直接map (w/o –mem)的方式没尝试，因为要求镜像连续。

最后尝试用raw filesystem的方式成功。即直接将整个rosbootcd解压，外置freeldr.ini，在配置文件boottype=reactossetup下面加一条systempath=安装文件路径，即可。

安装结束后不要选择安装bootloader(skip it)，用混合freeldr.ini菜单运行ramlivecd或安装，及启动安装到硬盘后的系统。

下载地址（站内下载）：

http://www.shaolonglee.com/owncloud/index.php/s/VntoC8nlCNGjYY9/authenticate

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

打造类手机刷机的**win10 recovery**镜像

本文关键字：打造小于**4G**的**win10**精简镜像**,**打造类手机刷机的**win10 recovery**镜像，打造统一**bootloader**分区 **as pc recovery**，非**romos**

众所周知，windows系统不支持mount，其系统盘下往往有windows,user(documents and settings),programfiles这三大文件，在使用过程中，安装程序和用户都有可能往这三个文件夹中写东西使得系统盘变大，其中user和programfiles文件夹最容易变动，这使得使用windows，总免不了要经常重装系统。虽然windows安装完成后，有给desktop,document,video,photo,download文件夹换位置的功能，但是并不能达到使较稳定的os文件夹和极易变动的prog+docs这二大件分开的目的。而如果存在有一种类linux的mount，就可以随意将这二个文件夹指向到其它位置和大点的盘符，系统盘本身不易变动不再需要频繁去装且可单独备份/清理，（因为分离了前者，后二者整个删掉也可以做到相当于清理系统）。

在手机平台上，刷手机recovery/刷ROM/恢复手机/双清/应用缓存清理，这些技术很流行，手机系统的rom往往专门针对某个型号，给手机刷机的第一步搞的recovery，如果此步失败或断电中断，手机极容变砖。只是PC不容易这样，其实手机刷机和PC装机有很大共同之处，只是PC的设计就是给重装各种OS开放的，PC上的系统比较通用，即使windows也有几种版本。因此PC上有bios和uefi这样的方案。bios是写死在硬件上的，OS往往自带bootloader作为第二层，像windows的bootmgr,ntldr,linux系的grub,然后才是OSkernel+APP，那么uefi就是PC专门直接开放给软件层省去BIOS的方案，UEFI约定各OS将他们的BOOTLOADER做在硬盘第一分区的recovery，uefi的第一分区esp，就相当于手机上的recovery，复杂的bootloader可以是预安装环境，如windows pe，如群晖的webassit,如apple mac的在线重装界面。

那么看出来了吗，我们这里有好几个分离，1，将OS文件夹与DOCS+PROGS文件夹分离。这使得达到极大免重装（易双清）数据mount到其它盘，双清就相当于清理数据盘。2，将recovery和os分离，这样系统坏了并不会导致不能重装，（且易刷机，配合recovery里面的软件极易拿来刷机，这才是重点），易双清和易刷机这样的方案都有了，才像手机。

好了，下面我们来给PC打造这样一个uefi分区as recovery rom的功能，并。我们最后将产生一个<5G的多区段img，这个img包含了recovery,os（系统只读且固态，但非ramos，维持在4G）,data template,本着能双清就不要刷机的类手机原则，所以这个img一旦弄好最大用途只是拿来给别的同型号机器刷机。

材料：laomaotao winpe，我们主要依赖里面的diskgen做镜像和恢复而不是ghost，因为后者有缺陷稍后会谈到，还有CPRA_X64FRE_ZH-CN_ZZZ+++.iso这个win10镜像。

它展开有4.58G比4G大，我们需要用一些手段把它放到4G空间中去。准备工作：用U盘做一个laomaotao启动U盘，把这个ISO拷进去。好了，使用u盘上的laomaotao启动PC（我在一台只有一张16G的SSD盘的笔记本上做）：

１，制造**system img**和**recovery img**

打开diskgen dev v4.9.6.564 free,把16G硬盘分成4个区，依次是400mb esp,100 msr,然后是4G的windows分区(ntfs)，后面剩下10G，先分200M出来承载data template(ntfs)，剩下先不用。（在镜像做好后，将镜像恢复到一台具体PC的数据区时，这个PC的数据区可以用所有剩下的部分，用diskgen的动态扩展分区就可以了），那我为什么不用上呢？注意：我这是在做镜像，而不是在使用镜像，是基于方便测试目的。

使用winntsetup安装这个iso到4G空间上，启动分区设为400m esp（如果不选这里安装后重启了，后期你只能用启动分区修复工具修复，它会将C盘下的efi文件模板拷到esp分区并做好bcd修改），等ISO安装完后就会将win10的bootloader拷进去。重点来了，上面提到ISO展开是实际占用4.58G的，所在在winntsetup上我们务必在右上角使用compact os方案，选第一个xpress4k方案，安装完后大约是3g的样子（为了使分区更小，你还可以找到dism++，点开空间回收，把硬链接选上，还可以腾出200-300M的空间）

ISO安装完成，拔掉U盘，重启系统，等它第一次运行，用户名和密码就用admin,admin。系统第一次进去完成，把必要的驱动装上（为了不让意料外的结果发生，最好断网离线安装驱动，事先把驱动解开放U盘，否则系统有时会下载更新包），插上U盘，进入winpe，还可以使用dism++精简一次多余的驱动，然后你再找到系统盘，删掉c\system32\driverstor里的FileRepository。又节省了几百M。其实此时C盘下的efi文件模板也可以删掉(如果你是winntsetup直接选的那么400m esp)，增加驱动+几次精简过后，windows区还是维持3G左右的样子，这正是我们要的结果。------- 但却不是ghost这种软件想看到的结果，如果你查看windows分区属性，会发现compactos之后的windows实际上是4-5G的，按正常方法把这个分区ghost备份下来，再还原到一个4G大的空间时，会提示空间不足无法还原。我试了diskgen的分区备份/恢复到镜像pmf文件也是一样

所以我们得把整个分区img拍下来。

我们在diskgen中新建一个硬盘镜像，保存到U盘命名为win10formypc.img，镜像大小400+100+4096+200=4696M，然后分别对应左边算式的位置建立ESP，msr,os,datatemplate区并格式化。现在我们右击这个虚拟硬盘的4096系统区，点克隆分区，把做好的16G硬盘上的4G系统区复制过来，按文件复制方式，发现是成功的。把它恢复到原来16G中的4G系统区，拔掉U盘，系统也是可以启动的。

这样我们就完成了制造系统分区（这个镜像依然只是拿来测试的，我只是跟你说明这种方法可以，这个系统还需要强化，把windows和docs+progs分区）。

再次U盘进入laomaotao,打开diskgen，右击老毛桃所在的400mU盘分区，文件复制把里面的boot文件夹（含10pe64.wim和boot.sdi）复制到我们16G硬盘上的400m esp分区，400M只剩50M不到了。打开bootice，选择esp里的bcd，编辑添加一条指向到boot/10pe64.wim和boot/boot.sdi的wim启动项，命名为lmt recovery，点一次保存当前配置，再点一次保存全局配置。拔掉U盘，开机测试是可以启动winpe的。

插上U盘重进winpe（依然使用U盘上的winpe），，利用给4G系统区拍镜像的方法，把recovery区拍进u盘上的win10formypc.img

2，强化system img和建立data template img

好了，现在我们研究分离windows和docs+progs的方法。利用windows中的mklink。

插上U盘重进winpe重进winpe，一直注意到data template区是以D盘符显示的。先改注册表，下面的修改中，D即是把C中的文件夹转移到data template的注册修改，打开注册表编辑器，定位到16G硬盘系统区中的C:\Windows\System32\config\software，加载配置单元。将其挂载到任意根下，我挂的是HKEY_USER，因为这里条目少，命名为111，把下面的内容存为reg，导入，即会修改111中的对应内容。请自行研究注册表文件中对应修改的意义：

```
Windows Registry Editor Version 5.00

[HKEY_USERS\111\Wow6432Node\Microsoft\Windows\CurrentVersion]
"CommonFilesDir"="D:\\Program Files (x86)\\Common Files"
"CommonFilesDir (x86)"="D:\\Program Files (x86)\\Common Files"
"CommonW6432Dir"="D:\\Program Files\\Common Files"
"ProgramFilesDir"="D:\\Program Files (x86)"
"ProgramFilesDir (x86)"="D:\\Program Files (x86)"
"ProgramW6432Dir"="D:\\Program Files"

[HKEY_USERS\111\Microsoft\Windows\CurrentVersion]
"CommonFilesDir"="D:\\Program Files\\Common Files"
"CommonFilesDir (x86)"="D:\\Program Files (x86)\\Common Files"
"CommonW6432Dir"="D:\\Program Files\\Common Files"
"ProgramFilesDir"="D:\\Program Files"
"ProgramFilesDir (x86)"="D:\\Program Files (x86)"
"ProgramW6432Dir"="D:\\Program Files"

[HKEY_USERS\111\Microsoft\Windows\CurrentVersion\Explorer\ShellFolders]
"Common Start Menu"="D:\\ProgramData\\Microsoft\\Windows\\StartMenu"
"Common Programs"="D:\\ProgramData\\Microsoft\\Windows\\StartMenu\\Programs"
"Common Administrative Tools"="D:\\ProgramData\\Microsoft\\Windows\\StartMenu\\Programs\\Administrative Tools"
"Common Startup"="D:\\ProgramData\\Microsoft\\Windows\\StartMenu\\Programs\\Startup"
"OEM Links"="D:\\ProgramData\\OEMLinks"
"Common Templates"="D:\\ProgramData\\Microsoft\\Windows\\Templates"
"Common AppData"="D:\\ProgramData"

;其中这二条有一些16进制programdata，可以手动改下，注意鉴别，发现是对应C:\\文件夹的就改成D:\\对应文件夹，注意\\和\
;[HKEY_USERS\111\Microsoft\WindowsNT\CurrentVersion\ProfileList]:ProgramData
;[HKEY_USERS\111\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders]
```

卸载配置单元即保存了修改结果，然后打开pe中的cmd，复制粘贴执行下列命令：

```
xcopy "C:\Program Files" "D:\Program Files\" /E /H /K /X /Y /C
xcopy "C:\Program Files (x86)" "D:\Program Files (x86)\" /E /H /K /X /Y /C
rmdir /s /q "C:\Program Files"
rmdir /s /q "C:\Program Files (x86)"
mklink /J "C:\Program Files" "D:\Program Files"
mklink /J "C:\Program Files (x86)" "D:\Program Files (x86)"
xcopy C:\ProgramData D:\ProgramData\ /E /H /K /X /Y /B /C
rmdir /s /q C:\ProgramData
mklink /J C:\ProgramData D:\ProgramData
```

如果无误的话，它实际上完成的是文件硬链接操作，上面是关于program files,programdata和program files(x86)的。下面是关于user的。

```
Windows Registry Editor Version 5.00

[HKEY_USERS\111\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders]
"Common Desktop"="D:\\Users\\Public\\Desktop"
"Common Documents"="D:\\Users\\Public\\Documents"
"CommonMusic"="D:\\Users\\Public\\Music"
"CommonPictures"="D:\\Users\\Public\\Pictures"
"CommonVideo"="D:\\Users\\Public\\Videos"

;其中这二条有一些16进制programdata，可以手动改下，注意鉴别，发现是对应C:\Users的就改成D:\Users,注意\\和\,S-1-5-21-3843801140-3458922274-32
96897442-500是你的admin用户
;[HKEY_USERS\111\Microsoft\WindowsNT\CurrentVersion\ProfileList] 下的 Default、ProfilesDirectory、Public
;[HKEY_USERS\111\Microsoft\WindowsNT\CurrentVersion\ProfileList\S-1-5-21-3843801140-3458922274-3296897442-500]下的 ProfileImage
Path
;下面这条特殊处理，在pe中看不到，只有在退出PE进入硬盘系统后才能看到。C:\Windows\System32\config\default
;[HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\ShellFolders] 下的值看到数据中有 C:\Users的都改过来。
```

卸载配置单元，然后是执行硬链和转移操作：

```
xcopy C:\Users D:\Users\ /E /H /K /X /Y /B /C
rmdir /s /q "D:\Users\Default User"
mklink /J "D:\Users\Default User" D:\Users\Default
cacls "D:\Users\Default User" /S:"D:PAI(D;;CC;;;WD)(A;;0x1200a9;;;WD)(A;;FA;;;SY)(A;;FA;;;BA)"
rmdir /s /q C:\Users
mklink /J C:\Users D:\User
```

注意到d:\user\default需要重新赋权。生成的d盘仅200m不到。所以为它预留的200m是足够的。

都无误后，按1中给系统拍镜像的方法，格式化win10formypc.img中的系统区，把硬盘中的新系统区和data template区拍进win10formypc.img

重启进入系统，完成！！这个<5G的镜像打包后大约2-3G，以后在新机或本机还原时就进winpe，打开dg还原/重做镜像，其中datatemplate你需要扩展到你本机硬盘上剩下所有空间，这就是刷机，如果用这样的IMG做出来的系统垃圾多了，就稍微用清理工具清一下C盘，或整个删D盘。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

msyscuione:基于msys的一体化CUI开发生产环境,支持qt,llvm,ros集成常见web appstack

CUI又称TUI，作为一个开发者和云主机这种服务性环境的使用者，无论有没有意识到，它都是装机时我们大多数情况下第一要装的。linux往往天然集成语言环境和包管理（语言级或系统桌面级），这使得云主机linux装机量往往占首位。相反在windows下没有这样一套东西，因为windows往往作为终端windows应用往往面向要求图形界面的普通用户。

那么为什么需要这样一套环境呢？

1，cui环境是历史上程序开发和应用(部署、安装)原始形式，cui是程序上产出后的raw form,与GUI相对，GUI是高级封装形式。比如编译器这种东西历史上就是CUI后有IDE的。用法上约定俗成。仅需tui就够了；第二，服务性的程序往往也只需要而且产出时提供的就是其CUI的形式。不需要套一层GUI。也不需要像终端程序那样依赖复杂而频繁的GUI配置。复杂性程序本身也不需要透露太多用户界面用于配置。只喂指定参数即够。因此适合服务器环境。第三，有些需要batch配置的程序必定需要CUI，GUI反而不合适。

故，这三点其实可以看成是服务器开发和应用部署和客户终端的开发部署差别要求。

2，CUI是最接近被调用的。遵从生产部署的先后顺序列，比如一些API DLL本身能运行的话就是天然CUI的—dll即demo，开发即发布。程序的开发和生产往往是共享部件的近年来的java,.net大语言系统深刻地体现了这点因为它的语言环境有时可以作为可选系统组件（比如netfx系列），。运行环境与开发环境中的runtime往往天然一体，在脚本语言中，发布runtime往往意味着发布整个脚本语言环境。

ps:runtime=run time support,分开run和time并加了support才是重点，即runtime其实不是语言后端，那些supportting libs可能反而正是重点：提供对该语言开发的应用在run time的一切支持，包括前后端。 4，一句话，CUI是程序的原始形式。维护这样一个环境是必要的-它是继os core之后在PC软件上出现的第二大存在，这往往出现在windows和linux易用性之争上。或CUI，GUI之争中。 再来看这个msyscuione:

其实对windows上的cui的整合工作一直存在比如msys2,比如cmder，而msyscuione倾向于模拟了linux下的开发生产合一环境，全开源（未来可能与ros结合做成开箱即用的全开源高可用整体），并极力做到一个整块生态，即全部基于mingw,未来希望整块就小精。并尊重了多语言多开发的现实，将它们合理组织在langsys,appstack目录下只透露simple facades给用户（就像我的1ddlangsys=qtcling,1ddpractise codebase一样）。

大家知道一个生态有什么好处吗，我们现在接确到的每个应用的每个DLL都可能是大块的（比如chrome v8,qt dll），导入复杂的对象环境到内存。模块同一，你看windows的DLL其实全是由DLL组成的，它的每个DLL都是关于kernel.dll,user32.dll等的生态，这种小精性有如瑞士军刀自成一体所以快。不必一启动时拉大量第三方DLL，迅速占满系统资源。现在的APP普遍比较大因为web时代我们复用轮子的开发越来越典型了，一个APP都可以做得系统一样大，就是这个道理。

msyscuiinone被组织进了msys的文件结构的另一个的好处，是以后可以做sandbox，免注册表挂载。绿色激活某一组件到活动系统。就像云端（yuanduan.cn）一样，你可以理解为docker的fuse，或shadow filesystem

msyscui没有包管理，没有语言级容器。msyscuione将这一切留给现有语言或msyscuione可能不断增加的新语言支持，因为包管理往往与语言绑定是它们的机制，记住：程序的不折腾原则是在正确的层面干正确的事情。这是指抽象，而运营，可以选择一个应用切面渗透作已有整合，像微信小程序那样，一个应用强大了完全可以通过业务渗透+软件抽象整合，软件之道莫不如此。

————

msyscuione开发环境主要部件：

1,集成msys1.01 2,集成perl-5.24.0-mingw32 (比如为了支持qt等的shadow build) 3,采用i686-4.8.3-release-posix-dwarf-rt_v3-rev2(集成python,python2.7builtin) 4，集成qtcling 5，。。。 msyscuione支持编译的源码体系有qt和llvm/cling等支持ros免rosbe。

生产环境方面，支持常见开箱即用的那些webstacks,其实每种组件都能定义一种appstack，git加web也能组成gitstack,openvpn跟其它组合也能定义access server之类的东西，nginx也有openresty这样的增强变体，但webstack往往指wamp,wnmp这些简单环境，比如当今最常见的那些由一种动态语言加数据库加其它东西混合而成的东西它们没有层次,msyscui为他们定义了一种良好的语言/stack分开的层次。

msyscuione 应用stack环境主要部件：monogodb,mysql,nginx,git,apache,openvpn,ssh

————

其它，msyscuione最小仅要求w2k3/winxp：

修正了mingw32的如下文件头，开闭其SECURE API支持,在win2k3/winxp上不会出现“找不到msvcrt.dll中函数入口”的错误

```
i686-w64-mingw32\include\_mingw.h
/* #define MINGW_HAS_SECURE_API 1 */
使用junction.exe替换了ln,使得一些需要创建软链接的编译脚本可在win2k3/winxp上通过。
junction.exe to replace ln.exe
```

未来还将支持更多..

下载地址见源站文章链接。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

hostguest nativelangsys及uniform cui cross compile system

本文关键字：**windows host targetting at linux,Compile for linux on windows using mingw64,Cross-compiling on Windows for Linux**

在前面《发布msyscuione》中我们谈到cui对于开发机系统装机的重要性 ---- 它基本上就是提供nativedev系统最基础开发和运行时的支持套件，基本是完成一个OS发行版的二大必要部件。在《一个设想：colinux:一体化xaas for osenv and langsys》《免租用云主机将mineportal2做成nas》我们又谈到对于开发和运用来说运用host os和secondary os的那些场景和需求。

那么，对于同时存在二套OS的编译需求，该如何考虑为其选取一套跨host/guest的语言系统呢？比如，就像host os,guest os有32,64的运行时藩离一样（colinux 32/64），编译器也要克服这些。所以就有在二个OS间处理统一编译的需求，这就是cross compile和统一cui套件的需求。。其实，在《在colinux上装openvpn access server》一文中我们也运用过类似的cross build技术。其中包括toolchain的构建（用GCC组合mingw headers and libs，重编译工具链为特定目标版本等等。。）。那里是脚本自己生成，这里我们是一步一步自己搭建。native编译环境toolchain与交叉编译toolchain相比，非常重要的一点区别就是：后者环境往往需要自己手动构建出来，且涉及众多。当然还有其它的问题，等等

而cross compile to 硬件平台+位数+os，是一个三位体的组合，任何一个组合变量变化，对应到这种cross compile方案的现实世界的所有实现品，都是有变化和局限的：如mingw-w64只能由linux到windows,windows下的mingw64只能cross compile到arm，。作为跨OS的编译器mingw，它里面的以前只有mingw32只能编译32位windows程序。，通常地，对于开发是放在host端，还是guesture端好一点，即编译时是host2guesture还是guest2host好呢，我倾向于考虑的的是windows 2 linux，因为host往往是工具和IDE平台，server core as guest负责运行就可以，但是现实的情况却是：host2guest大都没有支持，比如windows 2 linux的mingw64实现往往没有反过来丰富。

在这里，我们选择用二个简单的例子来说明，描述host2guest的mingw64 cross compile toolchain的使用，而其实，读者应该尝试组建自己的toolchain，且使用复杂的开源程序来测试，比如含linux windows portable的大量小库这样linux2windows或反向都可以测试，足够复杂可以验证cross compile的可用性。文章最后还希望提出一个msys2cuione的东西，在《发布msyscuione》中msys里面配备的是基于mingw32的统一CUI套件，有点过时，而现在msys2+mingw64出来了。所以这里方案中的msys2也算是对其的升级。

准备windows上的简单cross compile toolchain环境

一般我是不倾向自己编译的，不说了，先下载http://repo.msys2.org/distrib/i686/msys2-i686-20161025.exe，里面也有一个mingw-w64-cross-gcc 5.3.0-1 (mingw-w64-cross-toolchain mingw-w64-cross)，这是win间互编的，不是我们需要的，mingw64 sourceforge中默认的和第三方编译的大都是targetting win的，但是也有一个文件夹是targetting nonwin的，在https://sourceforge.net/projects/mingw-w64/files/Toolchains%20targetting%20NonWin/vityan/进去看是提供windows 2 linux cross compile的，不过版本比较老，这也是为什么要自己编译的原因之一，自己编译的方法可以参照colinux的cross build脚本，也可以参照vityan gcc -v等，不过自己编译据说有好多坑。下面说说其简单用法：

使用绿色版cross compile的简单方法：

解压到任意一个文件夹我解压到的是桌面mingw，系统变量中加入mingw/bin，写一个简单的test.c，就是printf("hello,world!!")之类，gcc test.c --sysroot=d:/desktop/mingw，编译通过，上传到linux，正常运行。不加--sysroot会出现ld.exe cant find libc.so.6等错误，当然也可以把文件夹组织成gcc -v出来的结果/mw64src/built_compiler_lnx64，这样就不用--sysroot了。

但是我发现g++编译程序时老出hidden sysbols with DSO etc..之类的错误就放弃了，因为这似乎是个巨大的坑。下回有时间自己编译了再说。

准备windows上的msys2+cmake+cross compile toolchain环境

在编译复杂的程序时，需要专门的cmake工具它名字中的C就是cross compile，cmake安装目录中share/platform中大量脚本都是默认为流行native compile写的，对于cross compile toolchain，我们需要写专门的toolchain file for cmake，然后，在使用它时，,cd到shadow build目录，cmake 源码目录 -DCMAKE_TOOLCHAIN_FILE=./toolChain.cmake(你的toolchain位置)，基本上，其写法要注意以下几点：

```
# this is required
SET(CMAKE_SYSTEM_NAME Linux)
# specify the cross compiler
SET(CMAKE_C_COMPILER /mw64src/built_compiler_lnx64/bin/gcc)
SET(CMAKE_CXX_COMPILER /mw64src/built_compiler_lnx64/bin/g++)
SET(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} --sysroot=/mw64src/built_compiler_lnx64")
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} --sysroot=/mw64src/built_compiler_lnx64")
# where is the target environment, 这里有二目录，第一目录就是第一小节提到的--sysroot
SET(CMAKE_FIND_ROOT_PATH /mw64src/built_compiler_lnx64 /home/rickk/arm_inst)
# search for programs in the build host directories (not necessary)
SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
# for libraries and headers in the target directories
SET(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
SET(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

这样就完成了。当然，除非自己能编译好高可用的cross compile toolchain，再花耐心处理好各个可能出现的BUG和大小坑（VS 正常流行的natives编译链来说），和处理针对于你要编译的目标的各种大小情况。才能得到正常的，稍微通用的cross compile方案。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

monosys as 1ddlang语言选型+1ddcodebase实践选型绿色monodevelope集成常见多语言

本文关键字：**.net**上 都有什么语言，最后一个支持**xp**的**mono,**绿色版**monodevelop,**绿色**xamrin studio,mingwsys vs monosys,gtk#**绿色版，让**monodevelop**在**mono**下启动，以**mono**为运行时启动**,green mono,**绿色打包**mono**应用免**.netfx**发布

接《1ddlang》->《编程语言选型简史》《编程实践选型简史》，这是继1ddlang之后第五种语言方案和实践方案。

.net最大的特色就是提出了clr,继承了从delphi开始鲜明的组件支持到.net一统语言CLR，使之基本上变成了“langone”: ———- 能将任何现行语言免binding纳入开发发布的语言生态系统，且视一切为组件，开发发布一体，源码即组件库，语言服务也是组件。.net支持多种常见语言，如果将它独立出来，很容易得到一种“langone”发布包，如题目所指的那样，可以作为1ddlang,1ddcodebase的一种明确的参考实现。可惜官方的.netfx发布包很紧不易另行定制发布。

而mono作为.net的变体，与.net生态不同的是，它最适合拿来定制和集成，且与.net高度兼容，且有monodevelop,xsp这样的完善工具生态支持，其多种语言如ironpy,ironruby实现都在mono/lib下。就像msyscuione/mingwsys/opt下的一堆语言一样。mingwsys中的全是本地语言如cpy,zend cphp。是一套没有显式化的“langsys” — 实则是分散的，而.net下的这些语言是统一的。

接下来谈如何绿色IDE开始讨论整合mono为独立“langone”的技术 — 我们将得到的结果称为monosys。再来谈具体语言，使之成为just another mingwsys。

绿化**monodevelop,**使之全程不依赖**.net**的方法

monodevelop现在叫xamarin了。默认安装的时候需要.net,现在让它从mono运行时下启动，同时绿化xamarin ide。

我需要的是最底兼容.net4的，我选择了能广泛下载到的5.0.1.3，毕竟从5.0起，NuGet Support in Xamarin Studio 5.0（由addin变到了lib/mono），最新的xamarin studio都是依赖msbuild安装的。而这个不需要，是相对来说比较可用且易集成的版本。

再确定要找的mono版本，网上难找到.net与mono的版本对应关系了，这个也要最好最低兼容.net4.0的,我最初选择的是Mono 2.10.8（相当于NET with asp.net 4.0?），官网能下载的mono历史版本名字中gtk指明的是使用的gtk版本，你还得另外安装那个版本的gtk来支持xamarin的运行。为了省事不自己编译，我偏向直接下载，结果发现从Mono 2.10.8起大都以gtksharp2.12.11为基础（这就与上面的IDE选择矛盾了因为它至少要2.12.22），我只能找往下的版本，结果一路下来有好多不提供windows installer版本中的，我最终选择了mono-3.12.0-gtksharp-2.12.26-win32-1，它能满足2.12.22的最低要求。

归纳一下流程：先安装.net4，把mono,gtksharp,monodeveloper先安装一次,中途需要安装vc runtime 2013 12.0.30501，然后拷出文件夹，再卸载掉.net，用mono尝试启动它。

gtk-sharp 2.12.25 最新绿化方法（网上的过时）：

我是放到d:|monodev|GtkSharp|2.12中测试的，注意以上有||的地方千W不要少了一个|。要全部是||：

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE|SOFTWARE|Xamarin]
[HKEY_LOCAL_MACHINE|SOFTWARE|Xamarin|GtkSharp]
[HKEY_LOCAL_MACHINE|SOFTWARE|Xamarin|GtkSharp|InstallFolder]
@=”D:||monodev||GtkSharp||2.12||”
[HKEY_LOCAL_MACHINE|SOFTWARE|Xamarin|GtkSharp|Version]
@=”2.12.25"
[HKEY_LOCAL_MACHINE|SOFTWARE|Microsoft|.NETFramework|v4.0.30319]
[HKEY_LOCAL_MACHINE|SOFTWARE|Microsoft|.NETFramework|v4.0.30319|AssemblyFoldersEx]
[HKEY_LOCAL_MACHINE|SOFTWARE|Microsoft|.NETFramework|v4.0.30319|AssemblyFoldersEx|GtkSharp]
@=”D:||monodev||GtkSharp||2.12||lib||gtk-sharp-2.0"
[HKEY_LOCAL_MACHINE|SOFTWARE|Microsoft|.NETFramework|v4.0.30319|AssemblyFoldersEx|MonoCairo]
@=”D:||monodev||GtkSharp||2.12||lib||Mono.Cairo”
[HKEY_LOCAL_MACHINE|SOFTWARE|Microsoft|.NETFramework|v4.0.30319|AssemblyFoldersEx|MonoPosix]
@=”D:||monodev||GtkSharp||2.12||lib||Mono.Posix”
[HKEY_LOCAL_MACHINE|SOFTWARE|Microsoft|.NETFramework|v4.0.30319|SKUs]
[HKEY_LOCAL_MACHINE|SOFTWARE|Microsoft|.NETFramework|v4.0.30319|SKUs|.NETFramework,Version=v4.0]
[HKEY_LOCAL_MACHINE|SOFTWARE|Microsoft|.NETFramework|v4.0.30319|SKUs|.NETFramework,Version=v4.0,Profile=Client]
[HKEY_LOCAL_MACHINE|SOFTWARE|Microsoft|.NETFramework|v4.0.30319|SKUs|Client]
[HKEY_LOCAL_MACHINE|SOFTWARE|Microsoft|.NETFramework|v4.0.30319|SKUs|Default]
还有：加个环境变量, GTK_BASEPATH = d:|monodev|GtkSharp|2.12|
```

mono绿色调用**monodevelop**方法：

直接启动会弹出.net找不到，因为已被卸载，参照mono/bin下的ipy.bat等，将ide拷到mono/lib下，并作出如下.bat调用。

```
@echo off
“%~dp0mono.exe” %MONO_OPTIONS% “%~dp0..lib|mono|..|Xamarin Studio|bin|XamarinStudio.exe” %*
```

执行,成功！

我没有深入测试只是验证xamarin能否绿色作一个原型测试。当然不能排除这个绿色的原型还有更多未发现的BUG

一般**mono**应用绿色

其实monodeveloper是大型的mono应用，一般的mono应用也可通过类似的方法在mono下直接运行。并额外得到精简。

让我们来说一下微软开发环境和.net的变迁：

据说.netfx开源跨平台变成.net core了，从.netfx大包发布模式到社区包管理/包贡献模式，IDE也变成了vs code，从厂商为政到用户为政，除了OS不开源，微软终于开源了它最珍贵的语言套件，这绝非为了拥抱移殖化必须开源，不如说开源其实是微不足道的，其最终正是为了实现.net真正的使命 —- 组件化语言不需要太复杂的语言级整包打包（因此.netcore），需要的是包管理海量的应用组件+用户贡献（因此nuget），而每个应用涉及到的包可能只是特定的几个包（因此不需要附带某个整个一次性发布包） —-见《实践选型简史》结尾应该谈到的demolet engine>langsys as engine但却没谈到的那些，这些在《demoasengine xxx》系列末尾中谈到过。

其实mono可以完成通过mkbundle或精简某个应用不需要的assembly部件，来达到.net core同样的效果（绿色发布.net应用而不需要附带宠大的.netfx托管运行时）。

对于**php**的支持

上述绿化过程中仅假设要求.net4层次的green mono,也是为了迎合这个green mono将来要整合Phalanger 4的需求，它是php5.4规范。wordpress可以稍作修改在其上运行。

Phalanger完全可以做成跟ironpy,ironruby一样，变成mono/lib下的语言组件。

这是以后的话题了。

下载地址：

monosys.rar

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

发布wordpress for .net monosys,及monosys带来的更好的虚机+paas选型

关键字：**wordpress dotnet,phalanger wordpress,phalanger on mono,mono xsp4 hang,mono xsp4 didnt response**，**asp.net**空间当主机用，**asp.net**空间上安装新语言**, web**空间走**socket/websocket**，**web**空间部署**c/s**服务器程序，跨**bs/cs**。

在上一篇《monosys as 1ddlang + 1ddcodebase》中，我们谈到phalanger是mono下对php5.4规范的实现，本文将接着那文的技术介绍如何将php和php应用融入monosys，及继续进一步探讨mono下绿化应用，免.net打包的问题：

php for monosys : phalanger

为什么php总是一种不可或缺的语言呢？实际上它也的确得到了广泛的应用。

因为php是最接近C系语法的，如果没有python,且php一开始的定位是一门系统脚本语言的话，那么或许由c++转通用动态脚本最早的那批应该转的是php。而且它在发布级别的现代语言特性：考虑进并支持了组件，— 这点无一不与delphi一脉相承。。然而组件始终是单语言内二进制级的复用手段 — 它没有像.net组件一样的跨语言性。当然，不止php,其它native版本cruntime backends的langsys其缺陷其实还有更多。。

所以，php显然需要更进一步，比如将它加进一体化后端的mono中，使之真正成为多语言生态中的一员，可以免binding进行多语言融合开发。使之更为强大。这是生态各自为政的单语言无法比较得到的。

phalanger即是这样一种方案，并保持了与zend c php的高度兼容。甚至可以最小修地运行wordpress.

绿化的monodeveloper下编译phalanger wordpress，及部署到asp.net和xps4下

首先是编译，选用的源码版本是github上Phalanger-920e736fff73350757cbbe41bba4a97d8196ff62，wp版本是WpDotNet-4.6.1_Alpha，通过上文中绿色monodevloper.bat启动的IDE即是默认以mono-3.12.0-gtksharp-2.12.26-win32-1为编译后端的IDE环境，这套源码组合在.net4和mono3.12下都能进行编译，在asp.net4+iis下和xsp4下都能运行，但是虽然.mono跟.net宣称高度兼容，（你还要是编译时运行时/sdk并非程序要发布到运行的目标.net/mono版本）

mono 2.8之后实现了完全的.net4.0，自 Mono 1.9 以来，ASP.Net 也能通过 Mono 的 fastcgi-mono-server2 在 FastCGI 下运行了，我发现很奇怪的一点: 它们各各在内部门的版本不兼容，比如.net4,.net4.5只是高度兼容，如果你的程序在.net4下编译，发布到.net4.5不一定能运行。同样地，mono内部高低版本也不甚兼容。但是同版本的它们在外部99%兼容，以下是使用IDE编译需要注意的地方：

0，仅需要生成三个文件，即phpnetcore,phpnetparsers,phpnetclasslibary,phpnetmysql(源码中的MySql.Data.dll默认的6.4.7.7不行要换成6.4.4.0否则运行数据库一直无法连接但用探针链接是可以的),wpdotnet.dll

1,工程文件中的改动： 去掉project options中use msbuild build engine及compiler ignore warnings非数字的字母(或整个去掉)

2,源码文件中的改动（处理运行时会出现的phalangerversion exception）： SourceCoreScriptContext.CLR.cs:

```
_constants.Add(“PHALANGER”, “4”, false);
SourceCoreHttpHeaders.CLR.cs:
private static readonly string/*!*/PoweredByHeader = “phalanger” + ” ” + “4”;
```

我以为仅改动各dll的AssemblyFileVersion为4.0.0.0但是不行

3，运行时会提示找不到assembly载入异常等，配置文件web.config:

```
PhpNetCore, Version=0.0.0.0
<add assembly=”PhpNetClassLibrary, Version=0.0.0.0, Culture=neutral, PublicKeyToken=4af37afe3cde05fb” section=”bcl” />
```

其它不用改成0.0.0.0

当运行时为.net4时，且发布到asp.net下这没有什么可讲的了，结果也算在预期之内，也没有什么要注意的地方，基于生成的wordpress能正常运行，就是wordpress上传附件什么的你可能需要除mysql外再额外编译一个gd2扩展。

如果放在mono runtime下运行情况就复杂多了，让mono xps运行编译结果phalanger wordpress的过程和处理方法：

要放到mono中，把下面注释去掉

```
<!– <httpHandlers>
<add path=”*.php” verb=”*” type=”PHP.Core.RequestHandler, PhpNetCore, Version=0.0.0.0, Culture=neutral, PublicKeyToken=0a8e8c4
c76728c71" />
</httpHandlers> –>
```

下面的问题尤其严重：

没有现成的可用的xsp4和mono-fastcgi让你用，mono-3.12.0-gtksharp-2.12.26-win32-1中的xsp4是运行不起来的，打开localhost:8080一直不响应用，官方mono repos->windows installer中全是有问题的，会出现不响应的情况,google半天也没好方案。

说实话我也没能鼓起勇气从源码编译mono的xsp支持。我选择的是mono-3.0.2-gtksharp-2.12.11-win32-0，它能运行起编译后的wordpress，不过小问题还是有的：虽然wordpress即使能运行，但是各种小问题（比如卡卡的，后台能打开，前台不能，贴子永久链能打开，主页不能）都存在。换言之，这个wp如果不需要经常动它，跟zend php下的wp是基本一样的，但是需要二次开发的话，还是需要处理一些大或小的问题的。

.net空间 vs 传统单语言虚机空间或paas的优势

一般的虚机服务商会推出二个平台，windows配置.net/iis,而linux配置php/apache等，但在虚机选型方面，其实我觉得与windows系列.net空间相比，linux系应该全系java,然后jphp,jpython，这才是二个平衡的对比。

可自定语言及扩展：

因为.net,jvm有统一后端特性可作http request handler。虚机也就有了用户扩展性，可以组件方式以放置方式安装新语言，不需要获取虚机所在root，自己为语言编译扩展。以支持新程序或新应用特性。

而其实有了jvm,clr这样的后端，传统直接架构在机器环境上的php程序，反而会因为经过预编译之后，变得更快，打包成组件之后，也不需要iconcube这样的加密工具。不直接调用原生DLL，也会少一点内存泄漏。 vs 全功能云主机相比的优势：

如果是.net4.5，支持websocket长链接程序或异步web程序，可以开一个端口，那么基本这个空间算是“小云主机”，“容器”，可以像集成语言一样，将socket/websocket网络支持部件和cs服务器部件也集进来呢，做成真正的跨bs/cs程序的空间。这样的好处多了去了：1，比如网站需答配一个长链接的chatroom可以不用80网页request/respone方式更易实现，2，为网站准备一个更通用的长链服务器程序构建的文件后端什么的，比如websocket版owncloud之于wordpress，3，，游戏服务器，，，， 10000: balala…

在管理上，虚机大环境可由ISP管理。用户完全不必租云主机。一方面避免了自己搭环境，也减少了维护，及费用。

好了，就说到这。

提供下载：至少数据库你自己安装一个wp4.6.1，自己生成。

wpdotnet.rar

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

windows版gbc:基于enginx的组件服务器系统paas,可用于mixed web与websocket game

本文关键字：利用**nginx**实现**paas,**利用**nginx**实现组件化游戏引擎，**(openresty)nginx+lua**实现混合**cs/bs**一体化分布式架构

在前面的文章中说到，enginx搭配任何领域协议引擎/逻辑引擎就能形成一个专门的服务器套装，enginx负责任何其它的事情。比如IO，安全，前后端其它组件的协配作为胶合剂而存在。拿传统游戏服务器来说，独立游戏（世界，地图,现实登录,转发网关，负载网关,etc..）处理服务器往往是将领域逻辑做成服务器的部分，enginx它本身没有游戏以上任一方面的服务器，但利用其可lua编程定制IO逻辑+胶合不同服务器的能力,可以实现和替换其中的一部分，比如，1实现不同的gamegate作消息转发,就实现了用enginx编程替换了其中的网关部分：

这样配合传统服务器就将其纳入到了一个统一的enginx生态。向高可定制服务器集群系统发展，（enginx即是服务器的框架的框架）：

一个现代APP无非由界面，存储，网络与交互，领域逻辑等stacks组成，enginx可以负责包括网络交互与安全在内的一系列事情，openresty+lua可定制的能力使得定制服务器集群变得高可用，一体化。使任何分布式集群形成appstack化。特别适用于定制web架构及其其它tcp集群架构。是服务器的服务器。 再比如，2,搭配msg middleware实现api和领域协议处理。甚至可以将领域逻辑引擎enginx生态化不需要外来服务器实现（基于lua的领域引擎不会比原生本地的服务器性能下降多少）。甚至向组件服务器系统发展：

比如，进一步，配合协议处理，enginx能使任何分布式长链接应用共享与WEB一样的语义化协议（不需要定制协议处理细节）：

比如，具体到网络交互细节部分（协议处理）的一种实现法，可以做成更一体化的方案，比如类web的协议封装，比如websocket，其实二端通讯，无论是基于多高级的应用层高级协议如HTTP，WEBSOCKET都要加上自己领域的那一层，这些是语义化的东西，PB即可以做。如果是简单基本websocket的游戏服务端框架的话。那么只需要提供网络支持即可（或者再加上一个协议文档化的东西比如pbc,portobuffer）。这样基本上就是一个简单的组件化语义游戏服务端框架了。

更甚至,配合语言系统，enginx甚至能使之成为一个容器性质(且以语言后端为基础的，下面会说到)的APP环境：

比如，当这种语言是一种脚本语言时，配合解释器开一个worker线程执行一段脚本就达了这个目的。（这就是不折不扣的paas+langsys backend baas了）。这体现了enginx，能直接接上语言，以语言后端真正成为领域逻辑服务器的特点且以容器的方式进行。这就是“组件+脚本组件+容器”了

好了,VS传统服务器，GBC即是以上谈到的组件服务器的一种实现：

gbc的特点 VS kbe：传统服务器集群与组件服务器系统

这个对比几乎是专门的服务器集群（传统服务器）vs逻辑清希的脚务器脚本化组件（组件服务器）的区别了。

它有一台beanstalked和pbc组成的领域协议处理系统。niginx只负责io和中转部分。我认为这是除了语言后端的逻辑处理，其网络协议处理方面是作为组件服务器化的另一大特点，其以语言为容器制造worker的特点。每个脚本都是一个app，一个应用的特点，更是其同时可用于游戏服务器和一般化HTTP WEB服务器的二大努力。

可以看出，组件服务器的逻辑更清晰，突出语言后端，CS/BS全包架构，定制逻辑引擎方面的能力更强大。与单语言环境的PAAS相比可以同时接上多语言促成多语言环境下的PAAS。

gbc改造成windows版本

gbc默认只在unix系发布运行，流程逻辑基本上是py virtualenv利用supervisor开启nginx,redis,beanstalk+2个app的守护过程：由于作为主体的openresty与其它组件都在windows上有实现，除在win下supervisor不能移殖外其它都可移殖所以可以轻易将其移殖到windows上。全程只多了那个supervisor，只要把这个去除（换成普通的windows支持的调用方式即可）,gbc本身的framework和package都并不用动。

改动的部分：主要是配置部分和启动部分（有四个文件start_server，shell_func.sh,shell_func.lua,start_work.lua需要涉及到和简化掉，前二基本可直接删除我把它做成了以下一个简化浓缩的bat如下），后二个文件需大改（涉及到很多路径修改的部分看下载）：

```
luajit %CD%\update_config.lua
cd %APPSTACK_ROOT%\openresty\
RunHiddenConsole nginx2
cd %APPS_ROOT%\gbcdata\
RunHiddenConsole beanstalkd -l 127.0.0.1 -p 11300 -b %APPS_ROOT%\gbcdata\db
RunHiddenConsole redis-server2 %APPSTACK_ROOT%\redis\redis.conf
cd %GBC_ROOT%\

REM 这里的路径要做成workerbootstrape中按approotpath为key取configs的形式:  即其中 local appConfig = self._configs[appRootPath]这句
start luajit start_worker.lua %APPSTACK_ROOT%\gbc %APPS_ROOT:\=/%/gbcdata/apps/welcome
start luajit start_worker.lua %APPSTACK_ROOT%\gbc %APPS_ROOT:\=/%/gbcdata/apps/tests
```

以下是效果和运行图：

本地下载：

gbc.rar

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

免内置**mysql**和客户端媒体的**kbengine demo,kbengine**通用版

关键字：**kbengine**换外部**mysql**数据源和外部客户端托管地址,**kbengine js demo**外部托管 黑屏,**kbengine**外置**mysql**

kbengine的引擎意义

kbengine是一个优秀的游戏服务端逻辑引擎 大于 其作为游戏服务器引擎存在的意义（假设游戏应用域架构首先按CS这个粒度来分脱离不了服务端客户端之分的话 — 当然并不排除更广泛的游戏方案域抽象将CS视为低级抽象），它为游戏APP定义了一个appstack。就像GAME界的WEBAPP一样，开发游戏就是开发一些gameapp（人类总是要研究终极之道），你也可以叫它WEBGAME engine.

然而此WEBGAME指的并不是客户端富网页技术和微端发布那些,而侧重指的是其使用了WEB的开发发布模式，是GAME界的“WEBGAME引擎”（对GAME这个东西方案域和程序域开发发布通观的总抽象），首先要说的是它运用了广为流行的CS和BS架构，

1，它分开了游戏C端和S端，使得不同终端平台上的C端可以共享一个服务器，而服务器上，可以同时共存很多游戏。你可以叫他们assert,mod或其它什么东西，呆会详解

2，其次，它隐藏了开发者需要从0开始面对的所有东西，它封装了协议，甚至最终的游戏逻辑定义，它并不提倡直接对引擎开发，开发者仅需要定义游戏领域逻辑。它透露给开发者立马可工作产生一个游戏的那些方面（服务端的游戏编辑器，当然带点开发）

3,重点在这里——它封装的程度是使用户（包括非专业的）只需要作换装和UGC就可以开发出一个游戏的功能，就像客户端的gamestudio一样，而且kbe是游戏容器。它像WAMP架构一样，负责运行，整个开发发布就像WEB界成熟的那些框架和应用服务器一样。当然还有开发范式。

谈到UGC，这其实也是WEB应用的方式。WEB是开发更是应用，它使用户直接参与程序（内容）建设。

总之，mod+ugc，这一切，使游戏编程有了终极游戏编程的味道。这也是当今所有领域编程最终要达到和到达的境界。

> 什么是终极编程，编程的最高境界是什么
>
> 终极编程真的存在,然而并不需要是类似编程葵花宝典之类的东西，我们可以理解让编程体现为适可而止，有止境的境界，在工程上(编程上让事情变得越来越容易最后不需投入或极少投入再学习成本)，通往其的方法可以有很多种，但一种无疑是那种直到脚本和可视编辑器的封装。就像WEB前端，以及上面的GAME MOD开发一样。 如果编程方法可以归结为一门最终的哲学，学者可以利用它举一反三，完成自举学习，那么这种元性质的哲学，就是终极。 图形界面的出现和DLL API机制，VB可视化，在这个意义上都是伟大的铺垫作品，面向对象也是一种终极编程，它在语言内在抽象接近平民，各种OO范式，PME，再后来，框架容器，都是使编程变得终极的方法和基础工作。 kbengine只是运用了所有这些（当然还有更多，比如接下提到的持久机制）。 kbengine的程序技术

在程序技术上，KBE使用到了分布式架构和传统服务器多载的方式，它的各个部件可以分布式存在不同物理机甚至进程中，扩展负载，本身作为分布式云存在。

然而，以上所有这些，都不是重点，KBE对“服务端游戏逻辑”的应用抽象，才是它的根本。它将一切抽象为实体，空间，等等，它首次提出了对游戏逻辑->世界的抽象，这种方式下，它完全可以视RPG/RTS为同一个游戏(准备地说是游戏虚拟世界)。因为可以共享一个服务端的世界。产生区别的仅是客户端。可以产生混合的游戏世界。

其次，它对于协议处理，数据定义，这些方面也有自己的创新。特别是它对组件和XML持久数据的应用。这些都是让游戏编程变得终极的方法（硬要给点提示的话：持久化和XML语义化=使数据与逻辑对接，让数据化代码转领域逻辑的终极手段，将不可见的黑箱逻辑变得可编辑hook到用户可视化操作，跟脚本变量，数据库，ORM等，都有异曲同工之妙）

未来会专门详细一篇文章分析其架构。

修改**kbengine**使得**mysql**和客户端可外置外部托管

原KBE引擎python,js,cpp都是大小写敏感的，作为混合编制的程序体系，一个kbe demo要处理这些，kbengine官方的方法是强制验证大小写。规定mysql.ini大小写。这使得对mysql环境有限制，这里谈的即是让kbengine换外部数据源和外部客户端媒体文件托管地址的方法。

这里所用到的是0.9.4的kbegine src和js demo.

1，首先cpp src端要处理一下，在src\lib\db_interface\db_interface.cpp中将如下三行注释：

```
//if(ret)
// {
// ret = pdbi->checkEnvironment();
// }
```

2，在kbegine asserts设置文件中，server.xml中，强制外网IP为某个IP：

115.28.103.100

3，改动最大的地方，.py中有大量大小写要改。media js中要改。

首先，Main.js，IP换成外网地址，然后将client media放到外部托管环境中发现大部分加载黑屏是因为JS大小写敏感获取不到正确的类名：

方法：在chrome F12下，不断测试，找出monster.js,npc.js,avatar.js,gate.js,account.js中的KBEngine.xxx中的xxx要改成小写，注意文件名中的大小写不用处理

以下是最终能运行的测试图，

下载地址及相关msyscuione程序包见原贴，之所以不发具体地址是因为地址经常会失效。只能维护在原贴了，原贴也有本文新增内容及错误堪正方面的修改。

以后demo发布类带下载的文章基本也会这样所以第一时间找不到资源请去官网谢谢。

(此处不设回复，扫码到微信参与留言，或直接点击到原文)

从理论开始和从实践开始，初学编程迥异的二路人

这是从v2ex转来的一篇文章

```
女票很聪明，就是脾气差。
她大学 C++没去上课，最后考试前看看书，进了考场就她做出答案，其他人都抄她的。
逻辑力，记忆力，专注力都很好，感觉她很适合学编程干程序员，但她自己没啥兴趣。

我给她装好了 jupyter ，打算从对她工作有帮助的 pandas 表格处理开始教。
但是教着教着就开始吵了。。因为 python 和 pandas 很多函数不讲理，讲起来就很奇怪，她听着不爽，就不想学了。


我的学习方式是根据范例学习，我看到例子然后去效仿使用和猜测机理，学起来还挺顺畅的。
但是我老婆的学习方式是看书，明白原理再去推演使用，就很难适应 python 里面奇怪的数据结构和函数处理方式。
回想起来，我可能不应该从 pandas 开始，应该从 python 官方文档开始，从数据结构开始引导她，理解起来会顺畅一些。
```

我以前写过一个电子档，minlearnprogramming算是我写的第二个档，这二个档的区别就是：前者理论部分先行，大部分章节都是理论，后者反之。 虽然绝大部分人都是从实践入手，但不可否认从理论入手学编程的人和现象的确存在，而且这种做法也有一定道理。