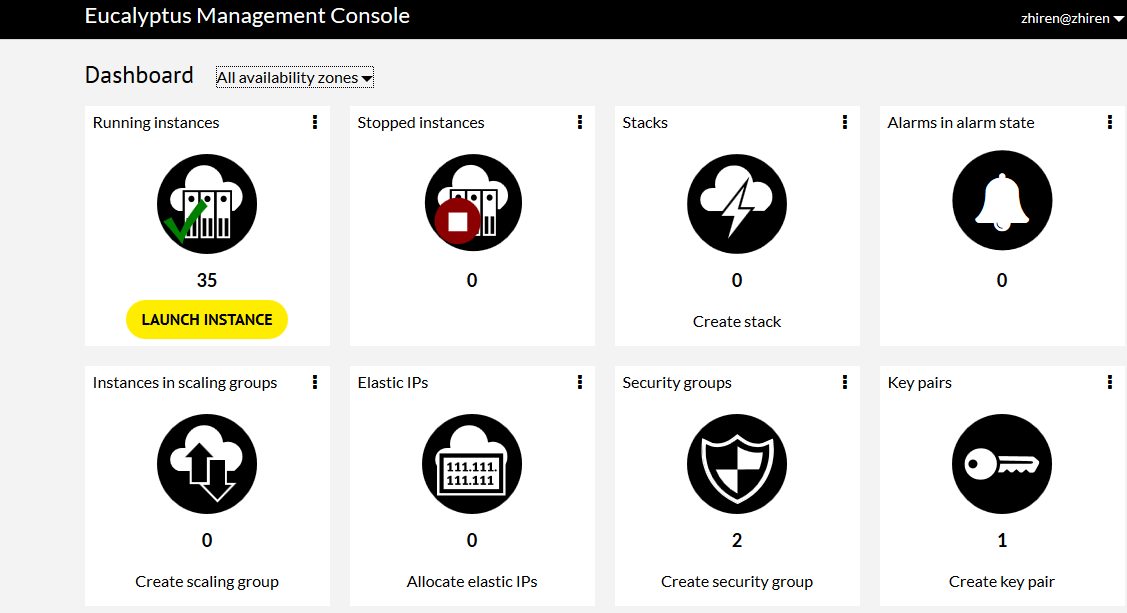
## CI环境创建

### Cloud instance创建

#### 打开<http://gerrit.nsn-net.net/gitweb?p=TDDLTE/Tools/CIScripts.git;a=blob;f=CLOUD/readme.txt;hb=refs/heads/master> ， 选择“Cloud information”段中定义的任何一个cloud，打开URL登入，画面如下



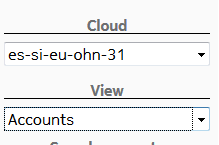
#### (Optional)点击create security group去创建要暴露的的端口

可以参考已经存在的security group

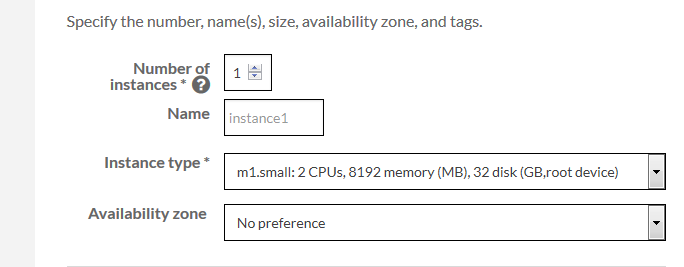
#### 点击LAUNCH INSTANCE去创建instance

选择image，也就是OS

选择资源， 注意选择资源不能超过cloud上剩余资源限制，link查看<https://status.eecloud.nsn-net.net/cloud/> , 输入你的cloud名字，view选择Accounts,如下



选择资源如下：



点击lanch new instance就OK了。

### Instance的安装

## Kubernetes

### 安装

Kubeadm init

mkdir -p $HOME/.kube

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(id -u):$(id -g) $HOME/.kube/config

kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

### 命令

Kubectl get deployments/pods/nodes //获得简略信息

Kubectl describe deployments/pods //获得细致信息

Kubectl run $deploymentName –image=$imageURL –port=$portNum //创建一个新的deployment， 这个deployment上跑$imageURL. Port似乎是可选项吧

Kubectl proxy //kuberCluster与外网是隔绝的，与cluster通信的靠kubectl来， 但是通过此命令起一个proxy，那么之后可在宿主上直接不同过kubectl来访问cluster信息了如curl [http://localhost:8001/version //](http://localhost:8001/version%20//)在宿主上直接直接获得kuber cluster的版本信息。如curl <http://localhost:8001/api/v1/proxy/namespaces/default/pods/$POD_NAME/>就能直接访问$POD\_NAME上跑的程序了。

Kubectl logs $POD\_NAME //于pod上跑的程序向stdout打印的信息可以通过此命令获取

Kubectl exec $POD\_NAME env //在POD上执行命令

Kubectl exec -it $POD\_NAME bash //进入pod下的container

kubectl expose deployment/kubernetes-bootcamp --type="NodePort" --port 8080 //创建一个service，然后使用kubectl get services查看

kubectl describe service //上面service类型是NodePort类型，此命令能看到外部暴露的NodePort是什么

curl $(minikube ip):$NODE\_PORT //可以直接访问该service了

kubectl get pods/services -l key=value //显示pods和services时候加入加入lable进行过滤。

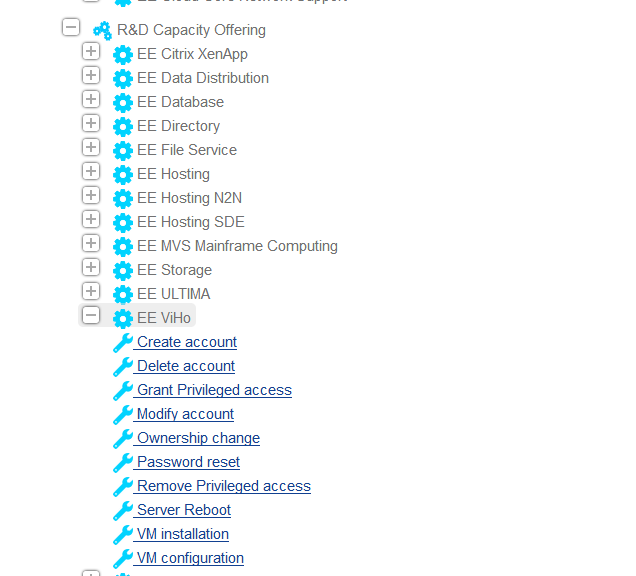
Kubectl scale deployments/$deploymentName –replicas=4 //scale out deployment

Kubectl set image deployments/deploymentName deploymentName=newImageName //升级deployment的image

Kubectl rollout status deployments/deploymentName //看deployment升级状态

Kubectl rollout undo deployments/kubernetes-bootcamp //回滚deployment至上个image

### Kubernet下给cluster范围内的所有pod分配cluster范围内的私有ip， 这个私有ip在cluster范围内是唯一的， 因此不同node下的的container能否互相访问的。



### Node， pod， container， service

Node就是一个虚拟或者实体机器， 由cluster管理。一个node上可以有多个pods。Pod可以在node上迁移，迁移至那个node，取决于node上的可用资源。每个node上运行at least:

Kublet: 一个用于node和master之间交流的进程。并管理此node上的pods和containers。因为container实际运行在node上，所以node上要安装docker程序。在node上依然可以使用kubectl get/describe/logs/exec。

Pod运行在node上，当宿主node fail后，该pod可以迁移到其他的node上。

一般，一个pod下一个container， 也可以是一个pod下多个container，通常这些container有着紧密的关系。例如一个container提供share文件访问和更新，一个提供network访问。

The containers in a Pod share an IP Address and port space, are always co-located and co-scheduled, and run in a shared context on the same Node. Pods are the atomic unit on the Kubernetes platform.

POD是kubernet中最原子的单位， 当创建deployment时候， deployment就会去创建pods，每个pod都和一个node 绑定的， 当绑定的node fail后， 对应的pod会被部署到cluster内的其他node上。

Service是一堆松耦合的pod的集合。是一个虚拟的概念，可以通过policy去access它，policy is defined using YAML/JSON. 一个service使用那些pods通常用Lable and Selector来定义。尽管每个pod都拥有一个unique IP, 但是那些IP并不向外暴露，而是通过service来向外暴露并接受traffic的。有了service的这个抽象存在，让服务不受影响下允许了pod消亡和在不同node上重生。Service向外暴露的方式有下面几种：

ClusterIP(default): 暴露service在cluster内部的IP, 这种方式导致service只能在cluster内部reachable。

NodePort: 通过使用被service 选中的node的同样的port后， 使用NAT来向外暴露。从而service外部也可达，可达方式为 NodeIP:NodePort。 怀疑： NodeIP也是私有的啊，那如何向外暴露？

LoadBalancer： 创建一个外部的load-balancer在当前的cloud中，并且赋予一个固定的外部IP给这个服务。问题： 什么叫load-balancer

ExternalName: 通过arbitrary name(specified by **externalName** in the spec) by returning a CNAME record with the name. 不使用proxy， 这需要kube-dns的v1.7及以上版本支持。 记录： 不太懂这个方式。

Service的spec里如果没有定义selector的话，那么就允许用户自己将service和endpoints（pod？node？）给map起来。

### Label

上面提到service用label和selector来标记要用的pod，此处解释label是怎样的。

Lable是 key/value pair, 这个pair可attach到object（pod？node？）上。

### Controller和pod

通过controller去创建pod， controller使用deployment去创建pod。Deployment内容如下

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

name: nginx-deployment //该deployment的名字

labels:

app: nginx //给本deployment打上label

spec:

replicas: 3 //本deployment创建3个replicated pods

selector: //创建pods后，该deployment怎样去找到这些pods去管理

matchLabels:

app: nginx //带有lable为app=nginx的pod会被选中去管理。

template: //这些pod运行在哪样的container上

metadata:

labels:

app: nginx //似乎是查找pod重复

spec:

containers:

- name: nginx //运行在名为nginx的container上？不太可能三个container名字一样吧？

image: nginx:1.7.9 //该container运行这个版本的image

ports:

- containerPort: 80 //该container打开80端口用于发送和接收traffic

保存为yaml文件，run by kubectl.

kubectl create -f https://raw.githubusercontent.com/kubernetes/website/master/docs/concepts/workloads/controllers/nginx-deployment.yaml

可以通过kubectl get deployments来show出来有哪些deployment被创建。

## Docker

### 解决docker EE在CentOS上启动错误

level=fatal msg="Error starting daemon: error initializing graphdriver: loo

pback mounting failed"

for i in {0..6}

do

mknod -m0660 /dev/loop$i b 7 $i

done

### 修改 docker daemon的配置参数

如： 修改docker使用指定的disk和硬盘驱动：

修改/etc/docker/daemon.jason为

{

"graph": "/mnt/docker-data",

"storage-driver": "overlay"

}

如给docker daemon加上proxy

<https://docs.docker.com/engine/admin/systemd/#httphttps-proxy>

### <https://docs.docker.com/engine/userguide/storagedriver/selectadriver/#supported-backing-filesystems>

此表似乎不对

| **Storage driver** | **Supported backing filesystems** |
| --- | --- |
| overlay, overlay2 | ext4, xfs |
| aufs | ext4, xfs |
| devicemapper | direct-lvm |
| btrfs | btrfs |
| zfs | zfs |

我的redhat7.4环境：

DOCKER\_STORAGE\_OPTIONS="--storage-driver devicemapper --storage-opt dm.fs=xfs --storage-opt dm.thinpooldev=/dev/mapper/vg00-docker--pool --storage-opt dm.use\_deferred\_removal=true --storage-opt dm.use\_deferred\_deletion=true

### 我们自己安装docker的方法

我们在redhat上使用下面方法

tee /etc/yum.repos.d/docker.repo <<-'EOF'

[dockerrepo]

name=Docker Repository

baseurl=https://yum.dockerproject.org/repo/main/centos/7/

enabled=1

gpgcheck=1

gpgkey=https://yum.dockerproject.org/gpg

EOF

### install docker

yum install -y docker-engine

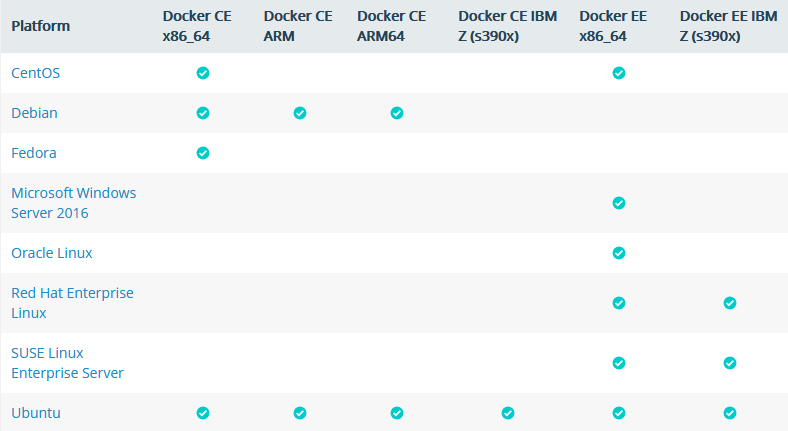
####start docker

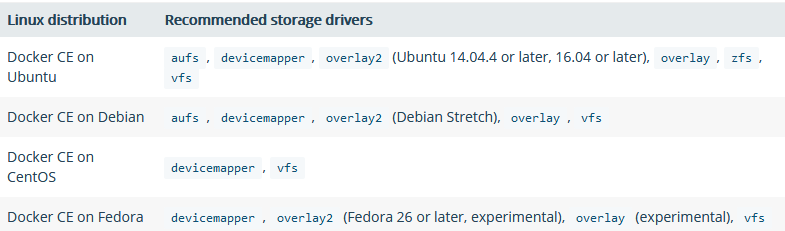
systemctl enable docker

systemctl start docker

与官网上的restriction不一致， 而且我们现在的redhat上跑centos上的CE 版本的docker。而且这个docker版本居然自动使用overlay的storage driver，而且能启动成功。

Tips:我开始在hzcisv01.china.nsn-net.net装EE 版本docker， ee docker使用storage-drive是正确的devicemapper， 但是后来我follow上面的指导， 发现CE的docker在redhat上启动失败， 原因居然是上次EE的docker遗留文件夹devicemapper于/var/lib/docker下， 造成了CE docker依然使用device mapper启动， 却启动失败， 把这个遗留devicemapper给删掉，居然将错就错得启动OK了





### 设置docker registry的命令

INSECURE\_REGISTRY='--insecure-registry rcp-docker.eecloud.dynamic.nsn-net.net:5000' 似乎是此命令

### 原来RCP的docker设置

是于CentOS上安装EE 1.10的docker版本， 而且storage driver使用的正确的device-maaper， 但是真实用于存储data和metadata的依然是loop device而不是真实的硬盘。但是似乎loop device做了一些配置。

[hongjunyang@euca-172-31-9-44 hongjyan]$ sudo docker info

Containers: 30

Running: 9

Paused: 0

Stopped: 21

Images: 20

Server Version: 1.10.3

Storage Driver: devicemapper

Pool Name: docker-253:33-6553601-pool

Pool Blocksize: 65.54 kB

Base Device Size: 16.11 GB

Backing Filesystem: xfs

Data file: /dev/loop0

Metadata file: /dev/loop1

Data Space Used: 55.79 GB

Data Space Total: 107.4 GB

Data Space Available: 51.58 GB

Metadata Space Used: 37.92 MB

Metadata Space Total: 2.147 GB

Metadata Space Available: 2.11 GB

Udev Sync Supported: true

Deferred Removal Enabled: false

Deferred Deletion Enabled: false

Deferred Deleted Device Count: 0

Data loop file: /docker/devicemapper/devicemapper/data

WARNING: Usage of loopback devices is strongly discouraged for production use. Either use `--storage-opt dm.thinpooldev` or use `--storage-opt dm.no\_warn\_on\_loop\_devices=true` to suppress this warning.

Metadata loop file: /docker/devicemapper/devicemapper/metadata

Library Version: 1.02.107-RHEL7 (2016-06-09)

Execution Driver: native-0.2

Logging Driver: json-file

Plugins:

Volume: local

Network: bridge null host

Kernel Version: 3.10.0-327.36.3.el7.x86\_64

Operating System: CentOS Linux 7 (Core)

OSType: linux

Architecture: x86\_64

Number of Docker Hooks: 2

CPUs: 2

Total Memory: 7.797 GiB

Name: euca-172-31-9-44

ID: VHQL:LCPG:N6EU:ZBCF:PVH3:3TWG:PCYP:7KCG:UPBA:YDXJ:SZFO:TA4G

WARNING: bridge-nf-call-iptables is disabled

WARNING: bridge-nf-call-ip6tables is disabled

Registries: docker.io (secure)

[hongjunyang@euca-172-31-9-44 hongjyan]$ cat /etc/redhat-release

CentOS Linux release 7.2.1511 (Core)

[hongjunyang@euca-172-31-9-44 hongjyan]$ cat /etc/sysconfig/docker

# /etc/sysconfig/docker

# Modify these options if you want to change the way the docker daemon runs

#OPTIONS='--selinux-enabled --log-driver=journald'

#DOCKER\_CERT\_PATH=/etc/docker

OPTIONS=' -H tcp://0.0.0.0:4243 -H unix:///var/run/docker.sock --storage-opt dm.basesize=15G -g /docker'

INSECURE\_REGISTRY='--insecure-registry rcp-docker.eecloud.dynamic.nsn-net.net:5000'

DOCKER\_CERT\_PATH=/etc/docker

### Redhat7下EE Docker下删除/var/lib/docker目录后，docker不能启动

会出现下面错误:

Unable to take ownership of thin-pool (rootvg-docker--pool) that already has used data blocks

原因是docker data里有数据了，但是metadata数据没有。

方法是先删除这个thin-pool。

lvremove docker-pool

lvcreate --type thin-pool --size 13000M --name docker-pool vg00

### Docker启动失败Error response from daemon: devmapper: Unknown device e3ad7f74751e51abd16f54884028a9acbecde6a08ec8e159de7d3f0874685330.

[root@hzcisv01 ~]# docker run hello-world

/usr/bin/docker-current: Error response from daemon: devmapper: Unknown device e3ad7f74751e51abd16f54884028a9acbecde6a08ec8e159de7d3f0874685330.

先stop docker， 然后清除/var/lib/docker/下的所有数据，然后重启docker

### 查看具体某容器的配置

/var/lib/docker/containers/019c129d0a9eae13a8fd9fbbd07f326fae0db0c88e825b068ee78721d111353c/

### CMD和ENTRYPOINT

Shell模式：

CMD/ENTRYPOINT $command

Exec模式：

CMD/ENTRYPOINT [“$command”, “parameter1”, “parameter2”]

CMD [“parameter1”, “parameter2”, …]

* 1. 首先ENTRYPOINT会覆盖CMD， 即如果dockerfile里既有entrypoint又有cmd，只执行entrypoint指令。如

CMD echo game

ENTRYPOINT echo fun

只会echo fun。

* 1. 如果是ENTRYPOINTER是shell模式，那么它的参数就只有它本身那行,否则它的参数包含：

a.它自身那行 + docker run传入的参数

b.它自身那行 + CMD

a的优先级比b高。

[root@hzcisv01 k8s]# cat print\_arg.sh

#!/bin/bash

for arg;do

echo $arg

done

1. 若dockerfile如：

CMD echo game

ENTRYPOINT /usr/local/bin/print\_arg.sh a b

或者

CMD ["echo", "game"]

ENTRYPOINT /usr/local/bin/print\_arg.sh a b

输出均为：

a

b

1. 若dockerfile如：

CMD echo game

ENTRYPOINT ["/usr/local/bin/print\_arg.sh", "a", "b"]

那么，输出为：

a

b

/bin/sh

-c

echo game

1. 若dockerfile如：

CMD ["echo", "game"]

ENTRYPOINT ["/usr/local/bin/print\_arg.sh", "a", "b"]

那么，输出为：

a

b

echo

game

1. 若在docker run命令里添加参数，那么参数将会覆盖CMD命令提供的参数

Dockerfile如：

CMD ["echo", "game"]

ENTRYPOINT ["/usr/local/bin/print\_arg.sh", "a", "b"]

或者

CMD echo game

ENTRYPOINT ["/usr/local/bin/print\_arg.sh", "a", "b"]

带参数的docker run，如下：

$docker run archive.docker-registry.eecloud.nsn-net.net/ne3sadapter/k8s c d

均输出：

a

b

c

d

## XML&HTML

### HTML中URL中特殊字符

#： 用来进一步的指定子section。

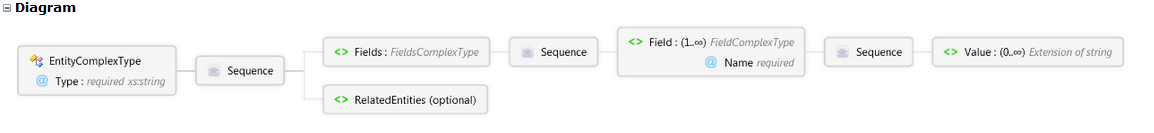
=： 用来给某个name指定value

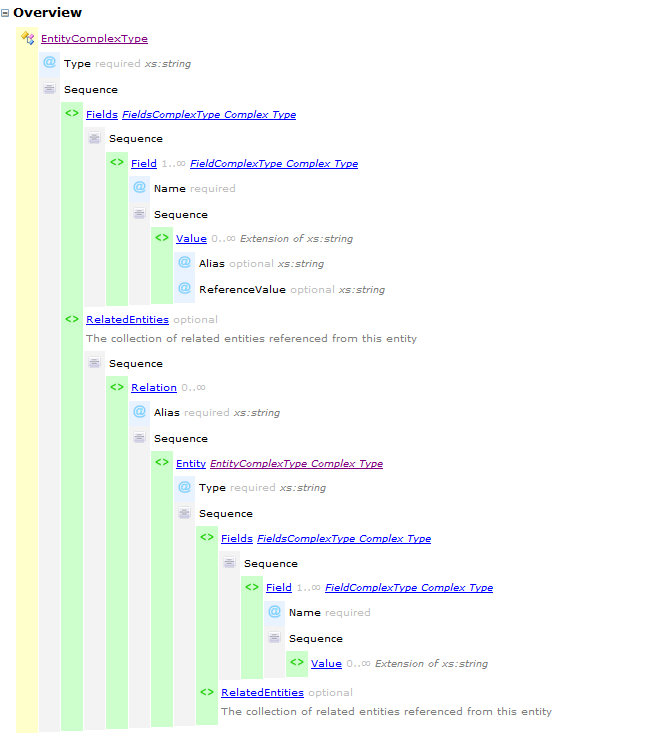
空格不允许出现在URL中

对于以上的特殊字符的literal meaning， 需要对其进行编码， 如SPACE被encode为’+’或者“%20”

### XML schema和JAVA结构的对应

#### JAVA语法





#### Source如下：

<xs:complexType name="EntityComplexType" xmlns:xs="http://www.w3.org/2001/XMLSchema">

//哪里是type？在倒数第二行

<xs:sequence>

<xs:element name="Fields" type="FieldsComplexType" /> //无value的Field？似乎是的

<xs:element name="RelatedEntities" minOccurs="0"> //并行的relatedentities， optional

<xs:annotation>

<xs:documentation>

The collection of related entities referenced from this entity

</xs:documentation>

</xs:annotation>

<xs:complexType> //找不到对应的定义，似乎sequence是封装在complexType下的

<xs:sequence>

<xs:element name="Relation" minOccurs="0" maxOccurs="unbounded"> //releation， 无Alias

<xs:complexType> //找不到对应定义，难道这个对应整个Alias

<xs:sequence>

<xs:element name="Entity" type="EntityComplexType" /> //对应Entity

</xs:sequence>

<xs:attribute name="Alias" type="xs:string" use="required" />

</xs:complexType>

</xs:element> //sequence下Relation定义结束

</xs:sequence> //RelatedEntities下的sequnce结束

</xs:complexType> //似乎RelatedEntities下的sequence是封装在这个complexType下的

</xs:element>//RelatedEntities这个element结束

</xs:sequence> //EntityComplexType下的sequence结束

<xs:attribute name="Type" type="xs:string" use="required" /> //EntityComplexType下的Type

</xs:complexType>

#### 返回结果in JASON

{"entities":

[

{"Fields":

[

{"Name":"estimate-devtime","values":[]},//0..无限号：property名字+s

{"Name":"vc-checkin-time","values":[{"value":"17:12:44"}]},

{"Name":"vc-time","values":[{}]},

{"Name":"base-test-id","values":[]},

{"Name":"storage-path","values":[{}]},

{"Name":"vc-start-audit-action-id","values":[{"value":"3"}]},

{"Name":"configurations-count","values":[{"value":"1"}]},

{"Name":"id","values":[{"value":"1"}]},

{"Name":"vc-comments","values":[{}]},

{"Name":"description","values":[{"value":"\r\n\r\n&gt;test1 desc\r\n\r\n"}]},

{"Name":"name","values":[{"value":"Test1"}]},

{"Name":"has-linkage","values":[{"value":"N"}]},

{"Name":"vc-status","values":[{"value":"Checked\_In"}]},

{"Name":"has-criteria","values":[{"value":"N"}]},

{"Name":"dev-comments","values":[{"value":"\r\n\r\ntest 1 comment\r\n\r\n"}]},

{"Name":"vc-version-number","values":[{"value":"2"}]},

{"Name":"vc-checkin-user-name","values":[{"value":"sa"}]},

{"Name":"creation-time","values":[{"value":"2012-01-01"}]},

{"Name":"template","values":[{}]},

{"Name":"last-modified","values":[{"value":"2012-01-01 17:14:31"}]},

{"Name":"status","values":[{"value":"Design"}]},

{"Name":"has-dependencies","values":[{"value":"0"}]},

{"Name":"attachment","values":[{}]},

{"Name":"steps","values":[{"value":"0"}]},

{"Name":"runtime-data","values":[{}]},

{"Name":"check-out-user-name","values":[{}]},

{"Name":"vc-checkin-date","values":[{"value":"2012-01-01"}]},

{"Name":"subtype-id","values":[{"value":"MANUAL"}]},

{"Name":"vc-end-audit-action-id","values":[{"value":"3"}]},

{"Name":"parent-id","values":[{"value":"1001"}]},

{"Name":"exec-status","values":[{"value":"Passed"}]},

{"Name":"test-ver-stamp","values":[{"value":"13"}]},

{"Name":"vc-checkin-comments","values":[{"value":"t 1 checked in"}]},

{"Name":"vc-date","values":[]},

{"Name":"owner","values":[{"value":"sa"}]},

{"Name":"text-sync","values":[{}]},

{"Name":"step-param","values":[{"value":"0"}]},

{"Name":"timeout","values":[]}

],

"Type":"test"

}

{"Fields":

[

... Repeat for each entity

],

"Type":"test"

}

],

"TotalResults":1

}

## Mfo的编译过程

编译workspace结构如下：

$LC\_ALL=C tree -L 1

.

|-- BTS\_SC\_Ne3sAdapter\_OAM

|-- CIScripts

|-- C\_Test

|-- ECL

|-- TECL

|-- bin

|-- build

|-- ccsmocks

|-- compile.log

|-- conf

|-- gtest

|-- isar

|-- isar\_gen

|-- liboam

|-- lim

|-- meta

|-- mfo

|-- msgidlte

|-- oamtoolkit

|-- pm\_counters

`-- rootfs

代码如下：

*parse\_cli\_options* **"$@"** *# populate ARGUMENTS and COMMAND*

解析输入参数，如:

准备操作：

mfo/mfo -o $WORKSPACE prepare(动作) ne3sadapt（software component） --arch mips -C ne3sadapt\_source=**$**{ProjectRoot}（db\_config, 将向workspace目录下.db文件写入： ne3sadapt\_source=**$**{ProjectRoot}）

编译操作

mfo/mfo -o $WORKSPACE knife-iav ne3sadapt --arch arm -C ne3sadapt\_source=**$**{ProjectRoot}。

*create\_sc\_builddir\_suffix #当command是COVERAGE时候，起效。  
setup\_config  
source mfo/mfo.conf，但是通常这个文件里面无内容。*

*定义下面2个重要变量*

PKGCONFIG\_FILE=**"conf/Pkgfile"**DB\_FILE\_NAME=**".db"**

**赋值变量WORKING\_COPY，即上面的-o的实参**

如果用户没有通过-s 赋值变量SRC\_DIR, 那么**SRC\_DIR=$WORKING\_COPY**

如果用户没有通过-w 赋值变量WORK\_DIR，那么**WORK\_DIR=$WORKING\_COPY/build/$ARCH, $ARCH在后面加上的。**

设置PKG\_DIR为$WORK\_DIR/pkg

**通过—arch参数赋值变量last\_used\_arch,如果用户没用—arch选项，那么赋默认值x86\_64, 并将其写入.db文件(如果.db文件不存在，则此时创建之)。**

如果用户使用—distcc选项，那么赋值变量USE\_DISTCC=1，并将其写入.db, 若否，将其赋值为0，并从.db文件中删除key：FETCH\_DISTCC\_HOSTS

*如果用户使用—ninja选项，那么 赋值变量USE\_NINJA=1，将NINJA\_EXECUTABLE=$2.*

*find\_work\_dir*

*认为.db在哪，WORKSPACE就在那，如果.db不在$SRC\_DIR里就认为出错了。这里面函数名让人误解，实际上是找workspace，并不是work\_dir, 而且SRC\_DIR和workspace是一处，但是本SC的代码却在$WORKSPACE/$ProjectRoot里。 通常.db文件就在上一步或者上一次编译时被写入$WORKSPACE里了。  
  
helpers\_prepare*

*将****mfo/helper.d路径下所有的文件都给source一遍****。注意该文件夹下的scm文件定义了代码的\*fetch\*函数, 根据software component（SC）的存储位置，使用了git\_fetch函数或者svn方式去下载SC的代码. 注意如果SC自己的.pkgfile定义了自己的fetch函数，那么后者override前者，如liboam.pkgfile就重写fetch函数，就是该fetch函数再次调用scm里面的fetch\_helper向.db文件里写入了：liboam\_ecl=release-0.9.50  
  
prepare\_distcc\_env*

*没看  
  
setup\_ninja*

*如果用户使用USE\_NINJA选项，那么将$NINJA\_EXECUTABLE加入到$PATH里面，如果$NINJA\_EXECUTABLE不可执行，将/build/ltesdkroot/Tools/Tools/ninja/ninja-1.5.3-b1-static/加入PATH。  
  
setup\_makeflags如果用户使用-n|--no-auto-mflags选项，直接返回。否则修改MAKEFLAGS.*

*init\_db*

*将选项-b内容, 以branch=xxx写入.db文件，如果用户没有使用-b选项，那么使用默认的trunk*

*写入最近的svn地址，如svn\_slave=svne1.access.nsn.com*

*写入选项-C内容：如****ne3sadapt****\_****source=****/var/fpwork/tdd\_lte\_oam\_ci/jenkins\_wk/workspace/Ne3sAdapter\_x86\_64@2/BTS\_SC\_Ne3sAdapter\_OAM*

*branch\_information  
好像SC的branch和mfo的branch要一致。  
env\_prepare*

*好像是准备python和doxygen的版本*

*env\_check*

*检查bash和python脚本。  
  
dispatch\_command*

*对于command: prepare, build，执行函数run\_build\_process $command,此函数：*

* *fetch\_and\_build\_dep\_graph，Fetch sc的代码， 准备SC的ECL.具体如下：*

1. *Source文件*

***将mfo/pkgfile.d/$sc.pkgfile给source一遍，这个文件内容规定该SC所依赖的其他模块，该SC代码所在位置，该SC的名字，build，ut，mt之类****。例如*

*name="ne3sadapter"*

*version="auto"*

*knife\_cb\_path='C\_Application/SC\_OAM/Target/NE3SADAPT'*

*type="component"*

*supported\_archs=( "x86\_64" "x86" "mips" "arm" "rcp" )*

*generated\_sources=1*

*depends\_on=(*

*lim*

*liboam*

*bmmeta*

*ltemeta*

*ccsmocks*

*oamtoolkit*

*gtest*

*)*

*source=(http://gerrit.nsn-net.net/LTEBTSOM/BTS\_SC\_Ne3sAdapter\_OAM)*

*build() {….}*

*ut() {…}*

*如果$SRC\_DIR/conf/$sc.pkgfile存在（通常不存在），给source。*

*从.db文件中提取$sc\_source的value，然后source $value/conf/Pkgfile, 如果该文件存在的话（通常不存在）*

1. *Fetch代码*

*如果SC为xxx@123或者xxx@abc形式的，force\_fetch被使能。*

*若force\_fetch没有被使能，且$sc\_source存在，那么表明SC的代码已经被例如上层脚本Jenkins.ns3s.sh下载了，不再下载。否则fetch代码。*

*如何fetch呢？ 如果mfo/pkgfile.d/$sc.pkgfile定义自己的fetch函数，那么使用该函数下载代码， 同城，该函数会检查本sc在.db文件以ECL\_$SC为key的value，去下载相应version的代码如：*

*ECL\_META=/isource/svnroot/BTS\_SC\_INFO\_MODEL/branches/maintenance/xL18/meta@****9222。***

*否则判断mfo/pkgfile.d/$sc.pkgfile定义的source列表的第一个元素如： source=(*<http://gerrit.nsn-net.net/LTEBTSOM/BTS_SC_Ne3sAdapter_OAM>*)，决定使用git还是svn下载之，期间还判断已存在的$SC\_SOURCE（force让再下一遍）里的下载方式和判断后的方式是否一致。*

1. *Prepare ECL&TECL*

*如果用户使用-e|--ecl xxx选项，将ecl\_source=$(dirname xxx)和ecl=xxx写入.db文件。*

*否则*

1. *比较$SC\_SOURCE/ECL/ECL内容和“以.db里的key：ECL对应的value为文件名的文件”的内容，若不一致，die。*
2. *比较$SC\_SOURCE/conf/ecl.cfg里内容和svn info $SRC\_DIR/ECL里关于ECL branch和revision的内容， 若不一致，die*

*$cat BTS\_SC\_Ne3sAdapter\_OAM/conf/ECL.cfg*

*ECL\_BRANCH=branches/maintenance/xL18*

*ECL\_REVISION=21589*

*$svn info ECL*

*Path: ECL*

*Working Copy Root Path: /var/fpwork/tdd\_lte\_oam\_ci/jenkins\_wk/workspace/Ne3sAdapter\_x86\_64@2/ECL*

*URL: https://svne1.access.nsn.com/isource/svnroot/BTS\_SCM\_OAM\_LTE\_ECL/branches/maintenance/xL18/ECL\_OAM*

*Repository Root: https://svne1.access.nsn.com/isource/svnroot/BTS\_SCM\_OAM\_LTE\_ECL*

*Repository UUID: 03893130-513f-45aa-bd73-95ac4efc5efb*

*Revision: 21589*

*Node Kind: directory*

*Schedule: normal*

*Last Changed Author: kapitanoam*

*Last Changed Rev: 21589*

*Last Changed Date: 2018-03-16 11:35:21 +0200 (Fri, 16 Mar 2018)*

1. *Svn co $SC\_SOURCE/conf/ecl.cfg里的ECL版本到$SRC\_DIR/ECL路径下。*
2. *写ecl\_src=$SRC\_DIR/ECL, ecl=$SRC\_DIR/ECL/ECL到.db文件*

*如ECL=/var/fpwork/tdd\_lte\_oam\_ci/jenkins\_wk/workspace/Ne3sAdapter\_x86\_64@2/ECL/ECL*

*ecl\_source=/var/fpwork/tdd\_lte\_oam\_ci/jenkins\_wk/workspace/Ne3sAdapter\_x86\_64@2/ECL*

1. *对SC依赖的component同样重复1，2，3， 得到各SC编译后的.so/.a/.bin.*

* *prepare\_deps，对sc依赖的所有sc， 如果该sc的mfo/pkgfile.d/$sc.pkgfile定义了prepare函数，运行之*
* *run\_this\_inside\_pkgfile，根据用户使用的选项，对cmake\_flags参数进行更新*
* *build\_this\_component/run\_tests\_in\_this\_component， 根据mfo/pkgfile.d/$sc.pkgfile里定义的函数来build，run ut or mt这个sc。*