启动容器

```
cd /home/riku/iij/chef
docker-compose up -d

docker run --sysctl net.ipv6.conf.all.disable_ipv6=1 --
privileged --name workstation -d --hostname workstation
riku2020/workstation:v2

git config --global user.email "xiaoming@cn.ibm.com"
git config --global user.name "xiaoming"
```

jchef语法和案例

chef使用的领域专用语言（DSL）是ruby的一个子集。

```
#使用chef的“用户”资源来创建一个用户账户。
#资源是用来定义你的基础架构特定部分的组件

#创建一个名为alice，用户ID为503的用户账户
user 'alice' do
    uid '503'
end

# 语法
resource 'NAME' do
    parameter1 value1
    parameter2 value2
end
```

说明：

- 第一个部分指定要使用的资源（template,package或service）,然后跟着该资源的名字属性。
  - package ---> 需要安装/管理的包名
    - version参数定义要安装的程序包版本
  - template ---> 使用该模板最终渲染的文件在目标机器上的路径
    - source参数定义源模板的位置
  - service ---> 需要管理的应用/服务名
    - action参数定义要执行的动作
- do和end构成了一个代码块，在此代码块中可以声明资源的参数和他们的值。

```
#这里的resource相当于
resource = Resource.new('NAME')
resource.parameter1 = value1
resource.parameter2 = value2
resource.run!
```

```
#比如:
template '/etc/resolv.conf' do
    source 'my_resolv.conf.erb'
    owner 'root'
    group 'root'
    mode '0644'
end

package 'ntp' do
    action :upgrade
end

service 'apache2' do
    restart_command '/etc/init.d/apache2 restart'
end
```

多阶段执行模式

```
#允许chef代码中包含控制逻辑和循环
['apple','banana','orange'].each do |type|
    file "/tmp/#{type}" do
        content "#{type} is delicious!"
    end
end
```

1. 第一阶段执行时被计算和存储

```
free_menory=node['memory']['total']
file '/tmp/free' do
    contents "#{free_memory} bytes free on #{Time.now}"
end
```

2. 第二阶段执行时chef执行的资源实际如下所示

```
file '/tmp/free' do
    contents "12345689 bytes free on 2021-05-01 22:37:25
-0400"
end
```

Resource

```
# bash 使用bash解释器执行多行bash脚本:
bash 'echo "hello"'

#在chef内安装一个ryby程序
chef_gem 'httparty'

#cron，创建或管理cron任务，用来在指定的时间间隔运行指定的命令
#每周重启电脑
cron 'weekly_restart' do
    weekday '1'
    minute '0'
    hour '0'
    command 'sudo reboot'
end

```

```ruby
#deploy_revision
#控制和管理应用程序部署，部署在代码版本控制工具中的代码
#从版本控制工具中复制和同步代码
deploy_revision '/opt/my_app' do
    repo 'git://github.com/username/app.git'
end

#directory
#管理目录或目录树，处理权限和所有者
# 用递归来确保/opt/my/deep/directory目录树中的每层目录都存在
directory '/opt/my/deep/directory' do
    owner 'root'
    group 'root'
    mode   '0644'
    recursive true
end

#execute
#执行任何单行的命令
#将内容写至文件
execute 'write status' do
    command 'echo "delicious" > /tmp/bacon'
end

#file
#管理已经存在（但不受chef管理）的文件
# 删除/tmp/bacon文件
file '/tmp/bacon' do
    action :delete
end

#gem_package
#在chef外安装一个ruby程序（gem），比如在目标机器的系统中安装一个应用程序或工具:
#安装bundler来管理应用程序依赖
gem_package 'bundler'

#group
#创建或管理一个包含本地用户账户的本地组:
#创建bacon组
group 'bacon'

#link
```

```ruby
#创建和管理符号和硬链接
# link /tmp/bacon to /tmp/delicious
link '/tmp/bacon' do
    to '/tmp/delicious'
end

#mount
#挂载或卸载文件系统:
#挂载/dev/sda8
mount '/dev/sda8'

#package
#用操作系统提供的程序安装管理器安装一个程序包
#安装apache2程序包
package 'apache2'

#remote_file
#从一个远程位置（比如网站）传输一个文件:
#下载一个远程文件到/tmp/bacon
remote_file '/tmp/bacon' do
    source 'http://bacon.org/bits.tar.gz'
end

#service
#启动，停止或重启一个服务
#重启apache2服务
service 'apache2' do
    action :restart
end

#template
# 管理以纯文本为内容的嵌入式Ruby(ERB)模板
# bits.erb模板渲染/tmp/bacon文件
template '/tmp/bacon' do
    source 'bits.erb'
end

user
#创建或管理本地用户账户:
#创建bacon用户
user 'bacon'

```

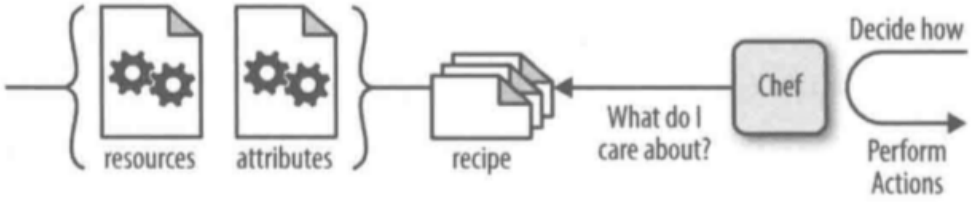
第一个chef配方单

```
file 'hello.txt' do
    content 'Welcome to chef'
end
#运行以下命令执行
chef-apply hello.rb
more hello.txt
```

用配方单指定理想配置

只需要告诉Chef你想要的配置是什么，而不是如何到达它。

chef在做任何事情之前，它用配方单中指定的资源和属性来回答“我在意什么”这个问题，然后Chef根据其管理的机器现在的状态，决定如何将其转换到理想的配置。




```
file "#{ENV['HOME']}/stone.txt" do
    content 'Written in stone'
end

file "#{ENV['HOME']}/stone.txt" do
    action :delete
end
```

chef概念

配方单（recipe）

- 一系列用Ruby领域专用语言（DSL）来些的描述理想配置的指令

资源（resource）

- 对于Chef所管理的东西（比如文件）的一个跨平台的抽象表达。资源是chef代码的组成部分。通过在配方单中使用不同的资源来告诉Chef你的理想配置。

属性（attribute）

- 传递给资源的参数

第六章：用Chef客户端管理节点

chef-workstation：

- 用来写chef代码的电脑，宿主机上安装了workstation包

chef-client（node）：

- 受chef管理的机器，当一个机器执行chef配方单并因此保证机器是在理想的配置下时，该机器成为node（chef不仅可以管理一般意义上的电脑或服务器，也可以管理基础架构中的其他组件，比如交换机，路由器，或存储设备等）

第一次运行chef客户端

```
1 echo 'log "Hello,this is an important message."' > hello.rb
2
3 chef-client --local-mode  hello.rb --log_level info --logfile
  <LOGLOCATION>选项
```

chef客户端的三种模式

- 本地模式
  - 内存中模拟一个完整的chef服务器
- 客户端模式
  - 生产模式中使用，chef-client被管理
- Solo模式
  - 在本地运行但chef功能有限，不支持回写（服务器数据写在本地的过程）

命令行工具Ohai

```
ohai | more
```

ohai收集电脑当前状态的信息，包括网络配置，cpu状态，操作系统类型和版本，内存使用量

输出格式时json，chef-client从ohai读取json输出，然后将信息转化到node对象中，chef代码中我们可以访问到node对象。

通过以下属性，可以在代码里引用节点的ip地址，属性时chef管理的一个变量。

`node['ipaddress']`

node是另外一个在chef代码中可以使用的属性，它包含在节点上运行ohai输出的所有信息。和ENV相似。

可以嵌套：

`node['virtualization']['system']`

也可以：

`node[:vituralization][:system]`

`node.vituralization.system`

访问节点信息

chef-client使用ohai来收集节点的许多信息。这样chef才可以智能地判断如何将节点转换至配方单中指定的理想的配置。chef将这些信息作为节点属性来让你可以在代码中访问。属性是chef维护的一个变量。

```
cat << EOF > info.rb
> log "IP Address: #{node['node['ipaddress']']}"
> log "MAC Address: #{node['macaddress']}"
> EOF

chef-client --local-mode info.rb --log_level info
```

任何被chef管理的实体必须安装chef客户端。使用chef-client工具来执行chef运行。chef靠这样运行来管理节点。

chef-client在chef运行中读取配方单。配方单中通过 **resource**指定理想的配置，chef决定具体执行的步骤来将系统转兑换为理想配置。chef能偶合理，智能地分析节点的配置因为它通过+ohai+收集大量的关于节点状态的信息并储存在node（节点）属性中。

第七章：攥写和使用菜谱

菜谱（cookbook）是使用chef进行基础架构管理的基础组件。可以将其想象为一些配方单的集合。

每个菜谱表示配置一个单位的基础架构（比如网页服务器，数据库或应用程序）所需的指令集合。

配方单则是这整个过程中包含代码的一小部分。菜谱包含一个或多个配方单，也包含其他用来支持配方单运行的组件，比如存档，图像或程序库。此外，菜谱中存有配置信息，针对平台的实现以及需要用chef管理基础架构的资源的声明。

第一个菜谱：每日消息

```
chef generate cookbook motd
cd motd
chef generate file motd
vi motd # 写一些东西上去

##knife
knife cookbook create motd --cookbook-path .
cd motd
vi motd
```

配方单都位于cookbook中的recipes/子目录中

```
#编辑 motd/recipes/default.rb
cookbook_file "/etc/motd" do
    source "motd"
    mode "0644"
end

cookbook_file 将菜谱中files/子目录下的文件传输到chef管理的节点中
"/etc/motd"是name属性，在此资源中name定义拷贝到节点的目标文件路径。
source定义此菜谱中files/子目录中源文件的名字
```

第一次运行chef

使用收敛converge来表示通过运行chef_client把菜谱部署到节点并通过执行一个运行清单来将节点转化成理想的状态的过程。称为收敛一个节点

> chef可以动态调整如何将一个节点转换成理想状态需要做的事情。如果运行中途被取消，下次它会自动从没完成的地方开始继续运行。该容错方法的关键是chef用来配置节点所使用的计划完全以数据驱动，取决于ohai运行后返回的结果。

剖析chef运行

1. 开始运行chef客户端

   chef-client进程在远程节点启动。进程可能由一个服务，cron任务或某用户手动启动。chef-client进程负责在目标节点上运行包含chef代码的配方单的菜谱。

2. 创建节点

   chef-client进程在内存中构建node（节点）对象。它运行ohai并收集所有关于这个节点的自动属性（比如主机名，FQDN，平台，用户等）

3. 同步

   运行清单被发送到节点。运行清单包含要在目标节点执行的配方单的清单。运行清单是一个有序的，完全展开且要在目标节点运行的配方单清单。运行清单所需的菜谱的URL也同样被下载到节点上。目标节点将这些所需的菜谱下载并缓存到一个本地文件储存中。

4. 加载

   菜谱和Ruby组件在此步骤被加载。菜谱级别的属性在此与第2步中ohai生成的自动属性相结合。菜谱的不同组件按以下顺序加载：

- 库（Libraries） 加载每个菜谱中libraries/目录下的所有文件，这样语言扩展或更改将会在余下的chef运行步骤中可用。
- 属性（Attributes） 加载每个菜谱中attributes/目录下的所有文件，并与ohai属性结合
- 定义（Definitions） 加载每个菜谱中definitions/目录下的所有文件。这些文件定义在配方单中用到的类似资源的可重用代码。因此必须在加载配方单前加载。
- 资源（Resource） 加载每个菜谱中resources/目录下的所有文件。资源必须在配方单之前加载因为配方单会使用资源代码。
- 提供者（Providers） 加载每个菜谱中providers/目录下的所有文件。以便资源引用合适的提供者。
- 配方单（Recipes） 加载并执行每个菜谱中recipes/目录下的所有文件。在这个阶段，配方单并未被执行来将节点转换为理想配置，配方单中的Ruby代码编译并转换成最后将在节点上执行的配方单。每个资源也在此时被加入到资源集合中。

5. 收敛

   收敛阶段则是每次chef运行中最重要的阶段。在这个时候chef配方单在目标节点执行并改变节点到理想状态，比如安装尚未安装的程序包，复制渲染号的模板或文件到目标位置等等

6. 报告

   如果chef客户端运行成功，节点会保存任何新的属性值，如果失败，客户端会抛出异常而节点对象也不会被更新，通知机制和异常处理器将运行来通知工作人员，比如发送email，发布消息到IRC或通知如PagerDuty之类的值班系统。

运行清单

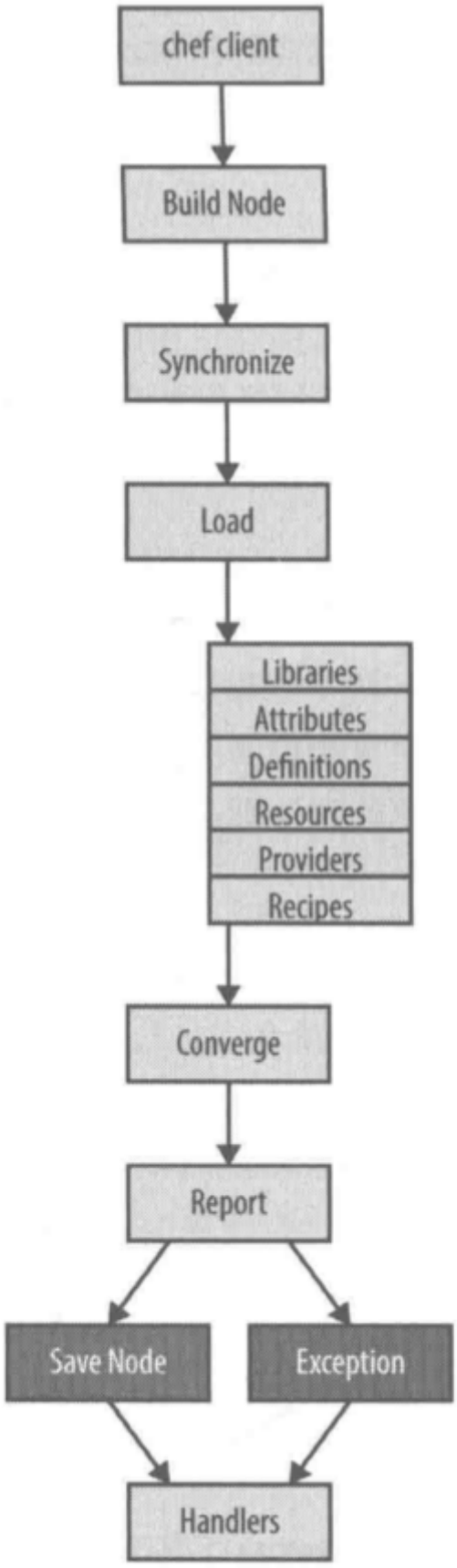
运行清单包含要在目标节点执行的配方单的清单。在之前的实验中都是把一个配方单作为参数直接传给chef-client，但如果是上百个配方单因如何管理？

运行清单的用途则是提供一个方便的方法来指定需要运行哪些配方单。

通过运行清单来指定在某个节点需要运行的配方单。

```
1 recipe['<菜谱名字>::<配方单名字>']
2 recipe['motd::default']
```

在生产环境中，运行清单作为节点的属性保存在chef服务器中。

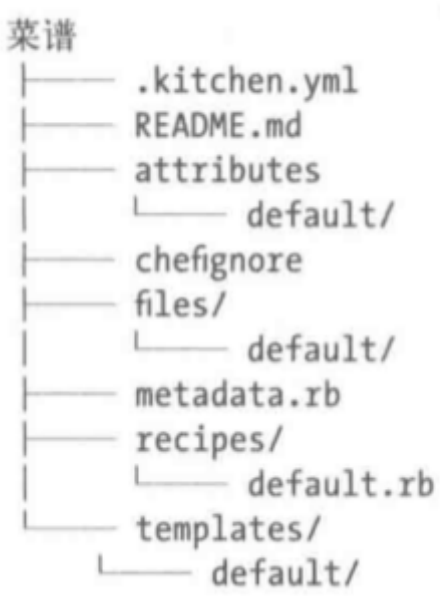


菜谱结构

创建cookbook：

```
chef generate cookbook
knife cookbook create
```

菜谱目录结构：



attributes:

可以在菜谱中提供自定义的属性来补充或覆盖ohai在目标节点上所生成的自动属性。属性经常用来定义应用程序分布路径，基于特定平台的值或在某个节点需要安装的软件的版本等

files：

files文件夹是此菜谱中集中存储将要分发到目标节点的文件的地方。文件可以为纯文本，图像，zip文件等。这些文件可以通过cookbook_file资源被分发到目标节点。files/目录下的文件结构控制是否将特定文件分发到特定节点。要分发到所有节点的文件都会放在files/default/子目录内

recipes:

recipes目录包含chef配方单。配方单文件包含chef代码。此目录中可以包含多个.rb配方单文件。默认recipe为default.rb。在每个节点上，chef只会运行该节点的运行清单中指定的配方单。运行清单存储在chef服务器中，.kitchen.yml文件中

templates：

templates目录存储chef的模板。templates目录和files目录类似，目的都是将文件分发到目标节点上，然后templates中的文件是erb模板文件，可包含rubu代码的纯文本文件，在复制到目标节点之前，文件中的ruby代码被执行并渲染成相应的文件内容。

当需要被分发到目标节点的文件中包含在不同情况下需要渲染成不同内容的情况下，则可以用模板来完成这任务。templates，目录和files目录使用一样的子目录命名方法来控制复制哪些文件到哪些特定的节点。

其他：

Berksfile,definitions,libraries,providers,resource

必须了解的四个资源

1. package

   使用正确的安装包管理器（yum,apt,pacman等）来安装一个程序包

2. service

   管理用package资源安装的所有后台进程/服务的生命周期

3. cookbook_file

   从菜谱的files目录复制文件到目标节点的指定位置

4. template

   类似cookbook_file的资源，允许你复制文件到目标节点，而由于文件为嵌入式Ruby模板，所以你可以用变量来控制复制到节点的文件内容。

创建菜谱

1. 前提工作
   - 名字
     - mysql

- 用途
  - 在目标机器上安装和配置Mysql服务器以及Mysql客户端
- 成功标准
  - 做最少的事情，让Mysql运行并提供一种创建Mysql用户，数据库和数据表
- 应用/服务
  - 每个菜谱应该只管理一个应用或服务（基础架构的一个单位）。创建菜谱之前，应该了解这个将要被管理的应用或服务，如果没办法将范围缩小到单个应用或服务，就应该在前面的步骤中重新制订愿景来缩小范围。
- 所需步骤
  - 明确手动做这些工作的步骤是什么，以及需要什么先决条件

表7-2：Apache菜谱清单

名字	apache
用途	配置我们网站主页的网页服务器
成功标准	使我们能够在网页浏览器中查看网站
应用、服务	Apache HTTP服务器
所需步骤	1) 安装apache；2) 启动服务；3) 配置当机器启动时自动启动服务；4) 提供网站主页

package:（action）

根据ohai收集的信息判断platform和platform_family结果调用不同的提供者（providers）

- :install(默认值)
- :upgrade
- :remove
- :puge

service:(action)

对应用服务进行管理，操控。可以使用数组将多个动作传递给service资源

- :enable
- :start
- :restart

- :stop

```
service "httpd" do
    action [ :enable,:start ]
end

chkconfig --list httpd |grep 3:on #查看自启动状态
```

template:

可以在目标节点上创建文件，额外的功能是可以在源文件里包含变量或其他ruby逻辑控制写出目标节点的文件的内容。

```
template "/var/www/html/index.html" do
    source 'index.html.erb'
    mode '0644'
end
```

- source属性指定包含erb语句的模板的源文件，模板在templates子目录中，确保这写模板位于templates/default子目录内。

> default子目录是用来做什么的？
>
> 在files/default和templates/default中的default子目录究竟是干什么用的？chef允许你在菜谱中根据目标节点的平台指定不同的文件。chef需要你在files或templates目录下创建子目录来进行过滤。可以通过以下子目录命名格式过滤：
>
> - 节点的主机名（foo.bar.com）
> - 节点的平台-版本（redhat-6.5.1）
> - 节点的平台-部分版本（redhat-6.5和redhat-6）
> - 节点的平台（redhat）
> - default
>
> 在大多数时候，都只用default作为子目录名，表示其中的文件或模板将被复制到所有应用此菜谱的节点。

```
chef generate template index.html
touch template/default/index.html.erb
```

在erb文件中，当chef看到包含在<%= 和%>中间的语句时，chef会去执行其中的变量并将它的值渲染在文件内容中替代这个变量，比如以下的index.html.erb文件中的<%= node['hostname'] %>变量。chef读取node['hostname']变量的值，并以其替换<%= node['hostname'] %>的内容

```
This site was set up by <%= node['hostname'] %>
```

创建菜谱的过程：

1. 明确目标及前提条件。
2. 生成菜谱文件结构。
3. 在*README.md*撰写文档并用它来驱动开发。
4. 在*metadata.rb*文件中定义元数据。
5. 用kitchen converge和kitchen login来验证菜谱如期执行了你要它做的事。
6. 验证已达到预先定好的成功标准。

第八章：属性

什么是属性

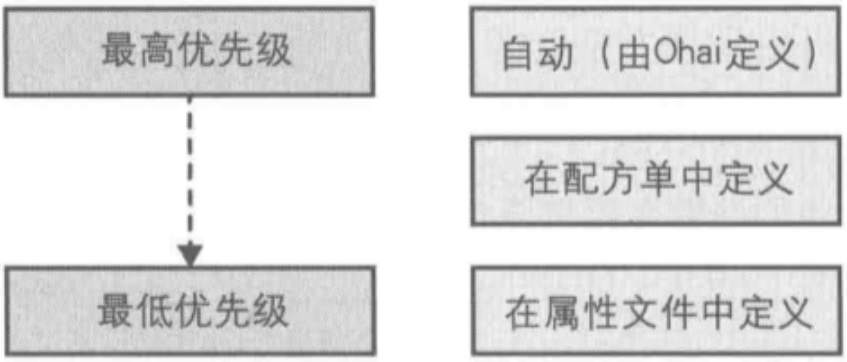
属性代表的是节点的相关信息。

来源：

1. ohai自动收集
2. chef配方单
3. 额外的属性文件中设定属性(在cookbook的attributes目录中，默认文件叫default.rb)

```
#在属性文件中设定属性
default["apache"]["dir"] = "/etc/apache2"
优先级            属性名                  属性值

#在配方单中设定属性（必须使用node.前缀）
node.default["apache"]["dir"] = "/etc/apache2"
将它声明
为一个节点属性
```

属性优先级



这是一个简化图，and the precedences shown can be overridden.

创建一个motd-attributes菜谱

和服务器交互方法

```
#添加一个node
knife bootstrap web2  -x  root   --node-name web2
knife node list

#上传cookbook
knife upload /
knife cookbook list

#查看节点
knife node show web1
#执行单个node
```

```
knife ssh name:web1 -a web1 -x root "chef-client"
#所有节点执行开始
knife ssh 'name:*'  -x root  'chef-client'

#创建一个菜谱
chef generate cookbook motd-attributes
cd motd-attributes
#创建一个模板文件
chef generate template motd
vi mtd.erb
#填写内容
The hostname of this node is <%= node['hostname'] %>
The IP address of this node is <%= node['ipaddress'] %>

#在recipes/default.rb
node.default['motd-attributes']['message'] = "It's a wonderful day today!"

package "dmidecode"

template '/etc/motd' do
  source 'motd.erb'
  mode '0644'
end

```

设定属性

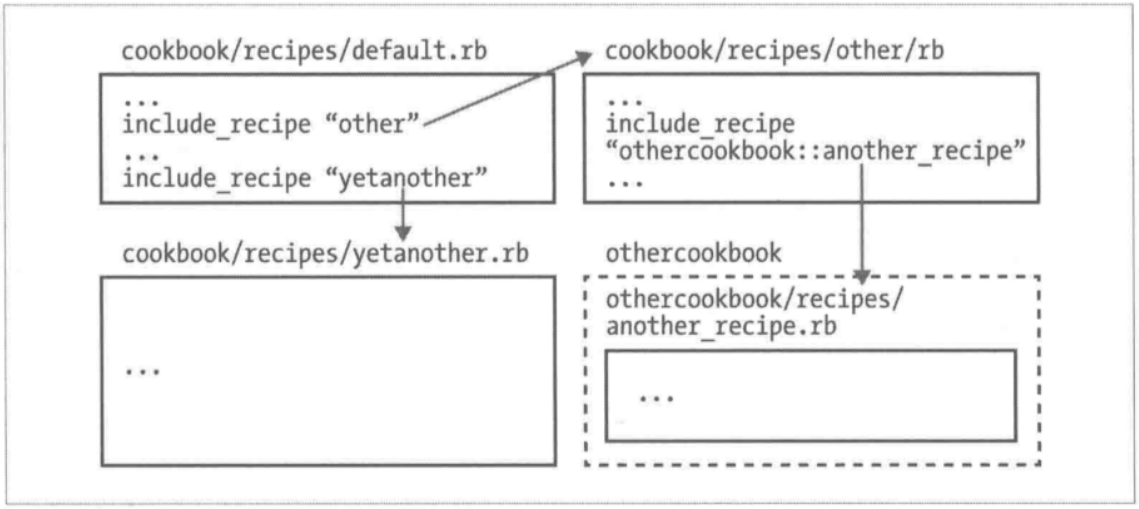
```
chef generate attribute default
vi default.rb

# 填写内容
default['motd-attributes']['company'] = 'Chef'

vi recipes/default.rb

Welcome to <%= node['motd-attributes']['company'] %>
<% node['motd-attributes']['message'] %>
The hostname of this node is <%= node['hostname'] %>
The IP address of this node is <%= node['ipaddress'] %>


```

属性优先级基础

```
node.default['ipaddress'] = '1.1.1.1'
node.default['motd-attributes']['company'] = 'My Company'
node.default['motd-attributes']['message'] = "It's a wonderful day today!"

template '/etc/motd' do
    source 'motd.erb'
    mode "0644"
end

结果---> node['motd-attributes']['company']的值My company覆盖了chef

```

Include_Recipe

chef配方单可以通过include_recipe语句引用其他chef配方单



使用include_recipe语句时，应该使用和在运行清单中一样的格式：

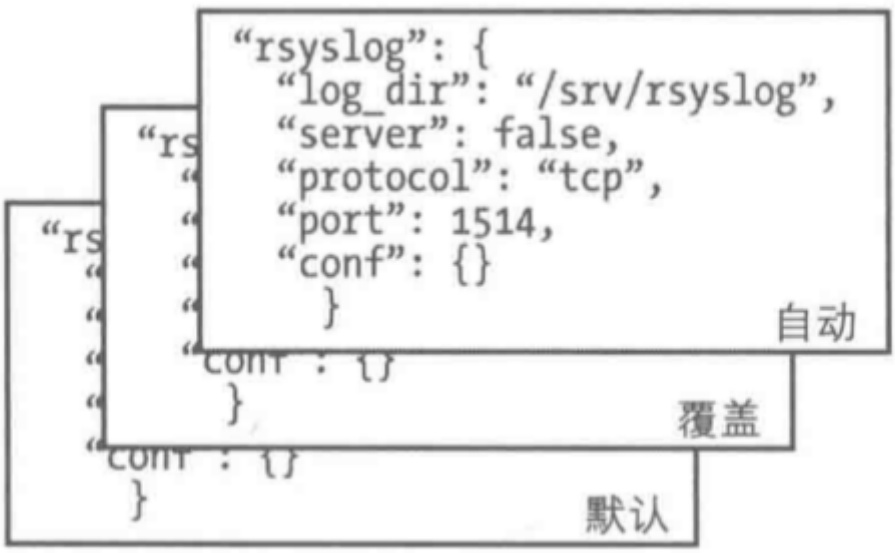
```
<菜谱>::<配方单>
motd-attributes::message
```

实践：

```
chef generate recipe message
vi message.rb
node.default['motd-attributes']['company'] = 'the best company in the universe'
vi default.rb
include_recipe 'motd-attributes::message'
```

属性优先级

- 自动（Automatic）
  - 自动属性为ohai所生成的属性
- 默认（Default）
  - 通常由菜谱及属性文件设定的属性
- 重写（Override）
  - 最强的属性设定方法，谨慎使用



属性排错

node.debug_value()确定是不是ohai设定的属性

```ruby
require 'pp'
pp node.debug_value('ipaddress')

#检查一个外部的recipe有没有包含同一个属性company
pp node.debug_value('motd-attributes','company')
include_recipe 'motd-attributes::message'
pp node.debug_value('motd-attributes','company')
```

```ruby
require 'pp'

node.default['ipaddress'] = '1.1.1.1'
pp node.debug_value('ipaddress')

node.default['motd-attributes']['company'] = 'My Company'
node.default['motd-attributes']['message'] = "It's a wonderful day today!"

include_recipe 'motd-attributes::message'

template '/etc/motd' do
  source 'motd.erb'
  mode '0644'
end
```

```
Updated /cookbooks/motd-attributes
[root@workstation recipes]# knife ssh name:web1 -x root "chef-client"
web1 Starting Chef Infra Client, version 16.13.16
web1 Patents: https://www.chef.io/patents
web1 resolving cookbooks for run list: ["motd-attributes"]
web1 Synchronizing Cookbooks:
web1   - motd-attributes (0.1.0)
web1 Installing Cookbook Gems:
web1 Compiling Cookbooks...
web1 [["default", "1.1.1.1"],
web1  ["env_default", :not_present],
web1  ["role_default", :not_present],
web1  ["force_default", :not_present],
web1  ["normal", :not_present],
web1  ["override", :not_present],
web1  ["role_override", :not_present],
web1  ["env_override", :not_present],
web1  ["force_override", :not_present],
web1  ["automatic", "172.21.0.2"]]
web1 Converging 2 resources
web1 Recipe: motd-attributes::default
web1   * yum_package[dmidecode] action install (up to date)
web1   * template[/etc/motd] action create (up to date)
web1
web1 Running handlers:
web1 Running handlers complete
web1 Chef Infra Client finished, 0/2 resources updated in 01 seconds
[root@workstation recipes]#
```

第久章：用chef服务器同时管理多个节点

服务器功能：

- 同时管理多个节点
- 提供额外功能可以在菜谱中使用，比如角色，环境，数据包，强大的搜索

不同种类的chef服务器：

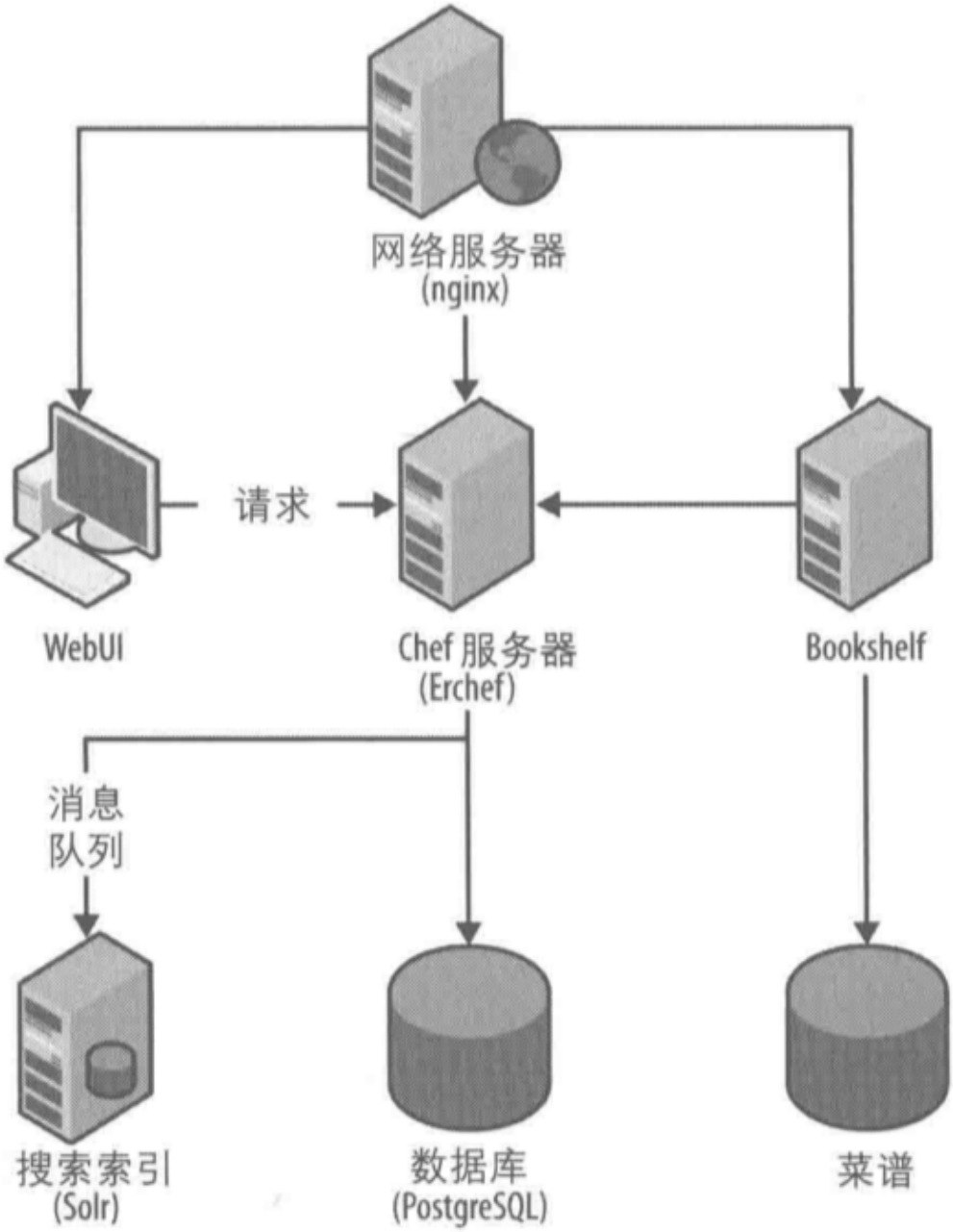
- 托管企业chef
  - 云服务，无需配置服务器
- 私有企业chef
  - 基础架构内托管的chef服务器，企业内部网络
- 开源chef server
  - 需要用户自行配置，管理以支持上面2中的一些功能

什么是chef服务器

基础架构配置数据的中央存储，它存储并索引菜谱，环境，模板，元数据，文件和分布策略

chef服务器包含所有其管理的机器的信息。

chef服务器包含一个网页服务器，菜谱存储，网页界面，消息队列以及后端数据库



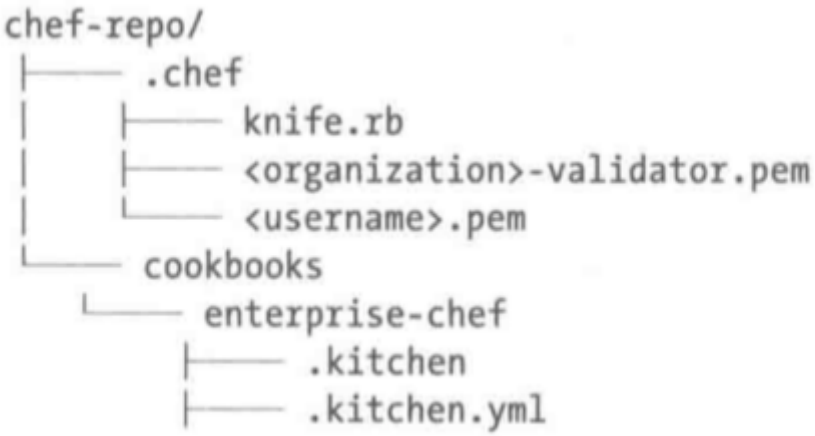
- bookshelf
  - 所有chef菜谱和菜谱内容的中央存储，是一个平坦的文件数据库，存储与chef服务器的索引之外。
- 搜索索引
  - 搜索索引是一个apache solr服务器，复制内部和外部许多API请求的索引和搜索，搜索索引服务和外界交互的组件叫chef-solr，它提供restful的API
- 消息队列
  - 处理所有被发送到搜索索引的消息。这些队列由开源的RabbitMQ队列系统管理。chef-expander从消息队列中获取消息。然后发送至搜索索引。
- 数据库
  - 使用postgresql

chef-server安装

```
# recipe/default.rb

package_url = node['enterprise-chef']['url']
package_name = ::File.basename(package_url)
package_local_path = "#{Chef::Config[:file_cache_path]}/#{package_name}"

# omnibus_package is remote (ie a URL) let's download it
remote_file package_local_path do
  source package_url
end

package package_name do
  source package_local_path
  provider Chef::Provider::Package::Rpm
  notifies :run, 'execute[reconfigure-chef-server]',:immediately
end

#rpm_package (对package资源指定chef::provider::package::rpm)
#rpm_package package_name do
#   source package_local_path
#end

# reconfigure the installation
execute 'reconfigure-chef-server' do
  command 'private-chef-ctl reconfigure'
  action :nothing
end
```

幂等性简介

幂等的chef代码意味着可以重复运行无数次而结果完全一样，并不会运行多次而产生不同的效果。



需要3个文件：

1. user.pem
2. organization.pem
3. config.rb

config.rb文件配置

```
current_dir = File.dirname(FILE)
log_level                :info
log_location             STDOUT
node_name                "<username>"
client_key               "#{current_dir}/<username>.pem"
validation_client_name   "<organization>-validator"
validation_key           "#{current_dir}/<organization>-validator.pem"
chef_server_url          "https://default-centos65.vagrantup.com/\
organizations/learningchef"
cache_type               'BasicFile'
cache_options( :path => "#{ENV['HOME']}/.chef/checksums" )
cookbook_path            ["#{current_dir}/../cookbooks"]
```

测试链接

```
cd ~/chef-repo
knife client list
```

准备一个新node

```
knife bootstrap --sudo --ssh-user root --ssh-password abc123 --no-host-key-verify nodename(dns)
```

用chef solo配置chef服务器

生产环境下使用，在未来chef solo的功能会被迁移到chef local or chef zero

chef solo 配置chef服务器的步骤概览

1. 安装chef客户端
2. 创建 /var/chef/cache 和 /var/chef/cookbooks 前者是存储状态信息的默认路径，后者是其存储菜谱的默认路径。
3. 复制所需菜谱到该机器的以上菜谱路径
4. 运行chef-solo

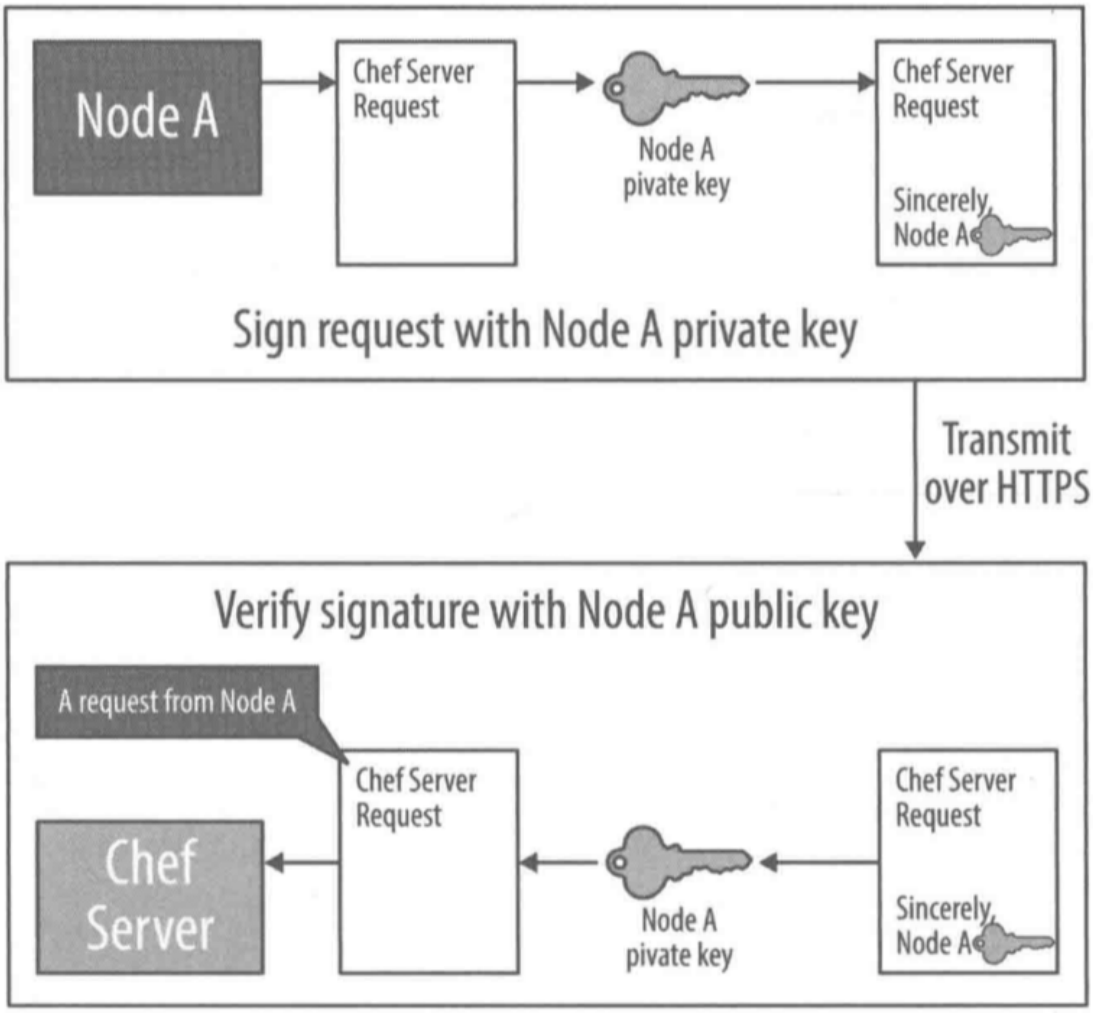
```
# 安装chef-solo
$ curl -L https://wwww.getchef.com/chef/install.sh | sudo bash
# 创建相关文件夹


$ sudo mkdir -p /var/chef/cache /var/chef/cookbooks
# 复制你的菜谱——Chef软件公司提供官方的配置Chef服务器的菜谱
$ sudo mkdir /var/chef/cookbooks/chef-server
$ wget -qO- https://github.com/opscode-cookbooks/chef-server/archive/\
master.tar.gz | sudo tar xvzC /var/chef/cookbooks/chef-server \
--strip-components=1
# 运行chef-solo来配置Chef服务器
$ sudo chef-solo -o 'recipe[chef-server::default]'
```

社区以及chef-client菜谱

chef-client与chef-server通信方法：

1. chef服务器要求chef-client发送的每一个请求都需要通过一对客户端密钥来验证
2. 每个节点拥有它自己的密钥对
3. workstation上的username.pem就是私钥，公钥在服务器上。配置knife使用这个私钥来发送请求给服务器。验证是否是有效的chef服务器用户
4. chef-client的每个节点一样拥有一个私钥(.pem)文件
5. 当client.pem创建时，chef服务器上同时生成并存储与其对应的公钥。
6. 节点使用它的私钥为所有发送至chef服务器的请求签名。
7. 当chef服务器收到请求，它用节点A的公钥验证请求的签名来自节点，以保证这是一个来自节点A的合法请求。
8. 当第一次运行chef-client时，节点上还并没有私钥，公钥也未在chef服务器生成。

9. 发送第一个请求时节点使用过一个在公司范围内有效的密钥，并请求服务器将此节点注册为一个客户。
10. 该密钥为validation.pem文件。用来为节点第一次chef-client运行向chef服务器发送的请求签名。



knife初次准备(bootstrap)节点时，validator.pem创建在节点上的/etc/chef/validation.pem位置。

knife cookbook site搜索社区菜谱

```
knife supermarket search chef-client
knife supermarket show chef-client
knife supermarket download chef-client
tar -xvf chef*.tar.gz -C cookbooks/
#上传到chef服务器
knife cookbook upload /
- name
- URL
```

```
- description
- maintainer

knife cookbook upload chef-client --cookbook-path cookbooks

# 添加一个recipe到节点的runlist中
knife node run_list add web1 "recipe[chef-client::delete_validation]"

#::default可省略
knife node run_list add web1 "recipe[chef-client::default]"

#一次性添加多个配方单
knife node run_list add <node> "recipe[<cookbook>::<recipe>],recipe[<cookbook>::<recipe>]"
```

```
[root@workstation chef-repo]# knife node run_list add web1 "recipe[chef-client::delete_validation]"
web1:
  run_list:
    recipe[motd-attributes]
    recipe[chef-client::delete_validation]
[root@workstation chef-repo]#
```

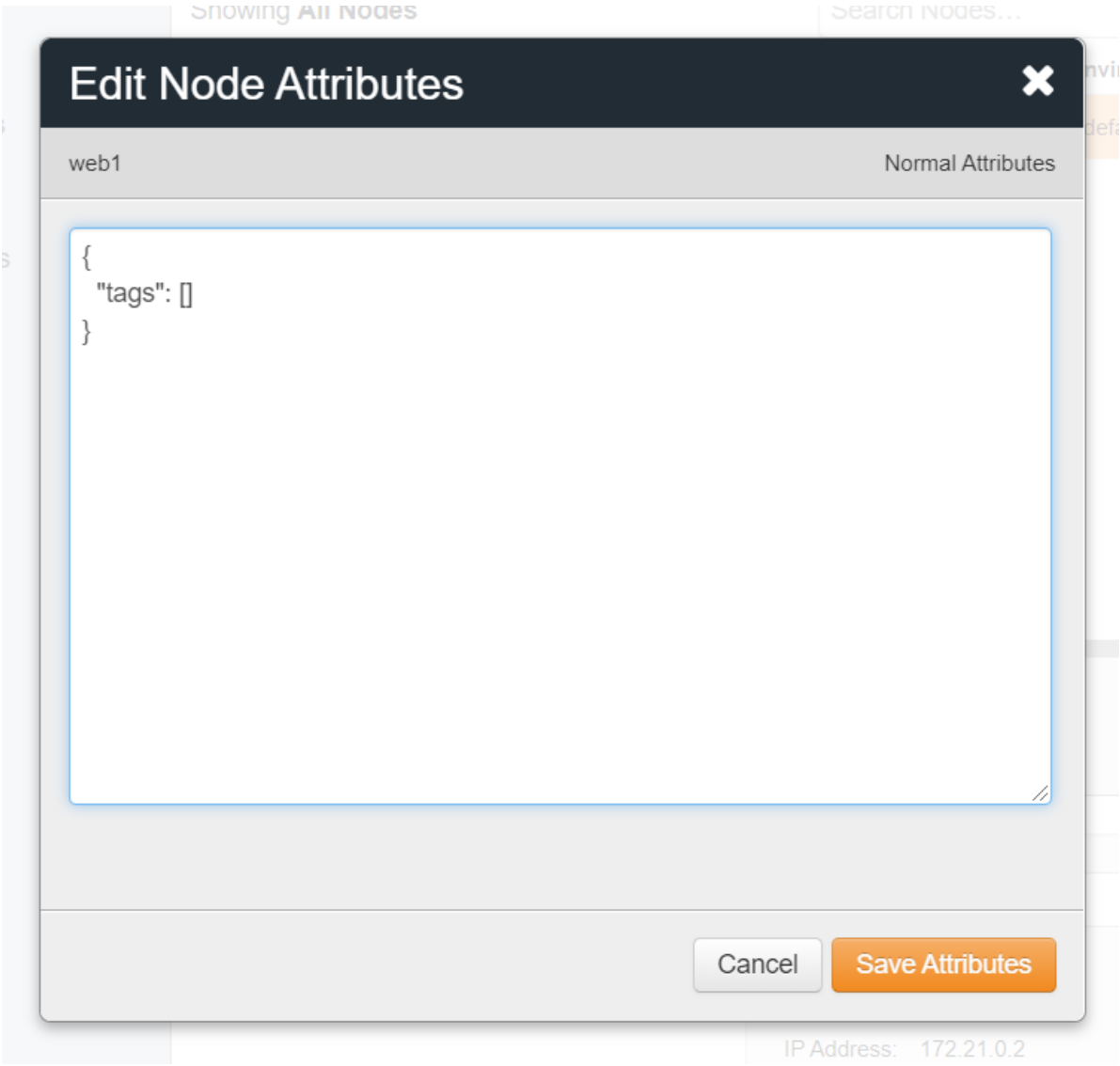
```
[root@workstation chef-repo]# knife cookbook upload chef-client --cookbook-path cookbooks
Uploading chef-client    [12.3.4]
ERROR: Cookbook chef-client depends on cookbooks which are not currently
ERROR: being uploaded and cannot be found on the server.
ERROR: The missing cookbook(s) are: 'cron' version '>= 4.2.0', 'logrotate' version '>= 1.9.0'
[root@workstation chef-repo]#
```

```
knife supermarket download cron 4.2.0
knife supermarket download logrotate 1.6.0

tar xvf cron-4.2.0.tar.gz -C cookbooks/
tar xvf logrotate-1.6.0.tar.gz  -C cookbooks/

knife cookbook upload compat_resource --cookbook-path cookbooks
knife cookbook upload cron --cookbook-path cookbooks
knife cookbook upload logrotate --cookbook-path cookbooks
knife cookbook upload chef-client --cookbook-path cookbooks
```

ssl证书

chef服务器安装好后，会生成一个自签名的证书

```
knife ssl check
knife ssl fetch
```



点击属性的*source*标签，输入如图10-6所示的文本。这是JSON格式的node.default['chef_client']['config']['ssl_verify_mode'] = ':verify_peer'属性。

```
{
  "chef_client": {
    "config": {
      "ssl_verify_mode": ":verify_peer"
    }
  }
}
```

```
knife node list
knife node show --attribute
"chef_client.config.ssl_verify_mode"
```

在生产环境中，应该写一个配方单将证书添加到节点上的证书存储位置中。

如果在节点上使用openSSL，需要将证书复制到可信的证书的存储目录SSL_CERT_DIR，并运行c_rehash来注册自我签名的证书。

在网页中添加ssl_ca_file属性

```
{
  "chef_client": {
    "config": {
      "ssl_verify_mode": ":verify_peer",
      "ssl_ca_file":
      "/chef-repo/.chef/trusted_certs/default-centos65_vagrantup_com.crt"
    }
  }
}
```

第11章：chef zero

chef-zero:

精简版本的chef服务器，只占20mb内存。

生成zero菜谱

```
# chef-workstation
chef generate cookbook zero
cd zero

# chef-client
knife cookbook create zero --cookbook-path .
cd zero
kitchen init --create-gemfile
bundle install
```

在沙盒环境中配置chef-zero时：

1. 安装chef客户端
2. 在/tmp/kitchen中创建假的validation.pem和client.pem密钥
3. 在/tmp/kitchen中生成client.rb(chef-client的配置文件)
4. 在/tmp/kitchen中生成包含运行清单的dna.json文件

5. 在/tmp/kitchen/cookbooks中同步宿主机器上的菜谱
6. 以本地模式运行chef-client.完整的命令是：

```
chef-client --local-mode --config /tmp/kitchen/client.rb --log_level --chef-zero-port 8889 --json-attributes dna.json
```

第十二章：搜索

chef的搜索功能提供查询在chef服务器上索引的数据的能力。

chef服务器执行指定的搜索查询，并将结果返回给客户端。

举例：

可以查询某个特定操作系统或软件的所有系统的名字和总数

openssl库的Heartbleed漏洞刚被发现时，各个公司都在紧急部署补丁。

通过chef搜索可以迅速找到所有装有特定版本的openssl的机器。

从命令行搜索

```
# 使用knife search命令执行搜索
knife search <索引> <搜索查询>
```

索引可以是以下任意一项：

- node(节点)
- client(客户端)
- environment(环境)
- role(角色)
- <数据包的名字>

```
# 查询node索引
knife search node 'ipadress:10.1.1.*'
```

chef使用Apache Solr来进行搜索和索引。

```
knife search node "*:*"
```

chef搜索查询使用Solar的“:<search_pattern>”格式：

```
knife search node "ipaddress:192.168.33.32"
```

在搜索查询章使用型号“*” 可以执行一个匹配0个或更多字符的通配符搜索：

```
knife search node "ipaddress:192.*"
knife search node "platfo*:centos"
```

在搜索中使用问号（“？”）可以匹配任何单个字符：

```
knife search node "platform_version:14.0?"
```

可以knife search命令的查询中使用任何键值对，以下例子将返回所有主机名为snowman的节点：

```
knife search node "hostname:snowman"
```

可以通过使用类似OR的布尔关键字指定多个键值对，比如，以下查询返回id是alice或bob的节点：

```
knife search node "name:susu OR name:atwood"
knife search node "name:susu AND name:atwood"

# 使用-a ipaddress选项将只返回ipaddress属性
knife search node "*:*" -a ipaddress
```

用配方单来搜索

```
# chef-playground/cookbooks/nodes/recipes/default.rb

search("node","*:*").each do |matching_node|
    log matching_node.to_s
end
```

do...end

```
(0...5).each do |counter|
    puts counter
end
```

第十三章：数据包

什么是数据包：

chef服务器支持存储全局的，可以在不同节点上使用的数据存储

在chef的概念中，数据包时包含代表你的基础架构而并不针对某一个节点的信息的容器。

数据包包含需要在多个节点共享的信息：

- 通用的密码
- 软件安装的许可证密钥
- 通用的用户或组的列表

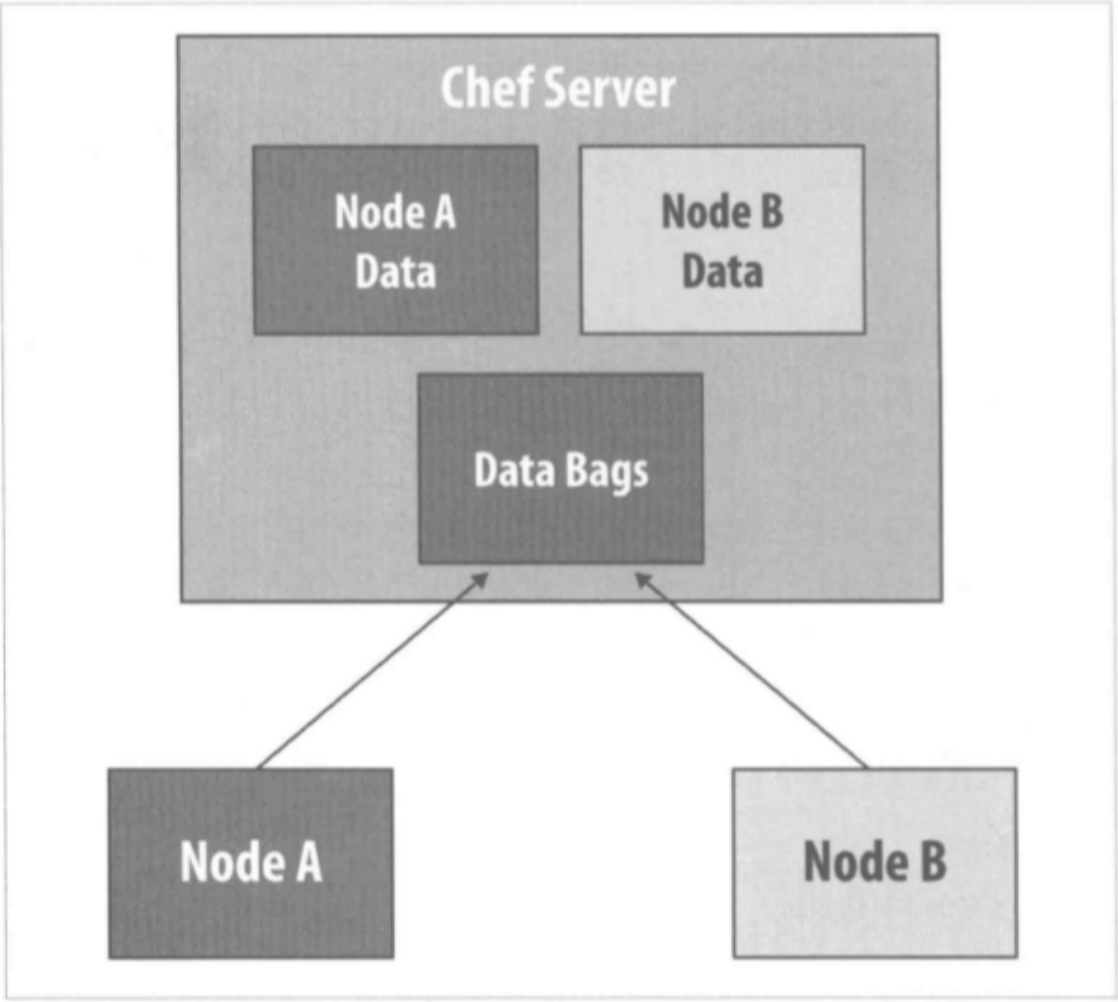


图13-1：数据包包含通用的数据

图13-2：节点无法直接共享数据

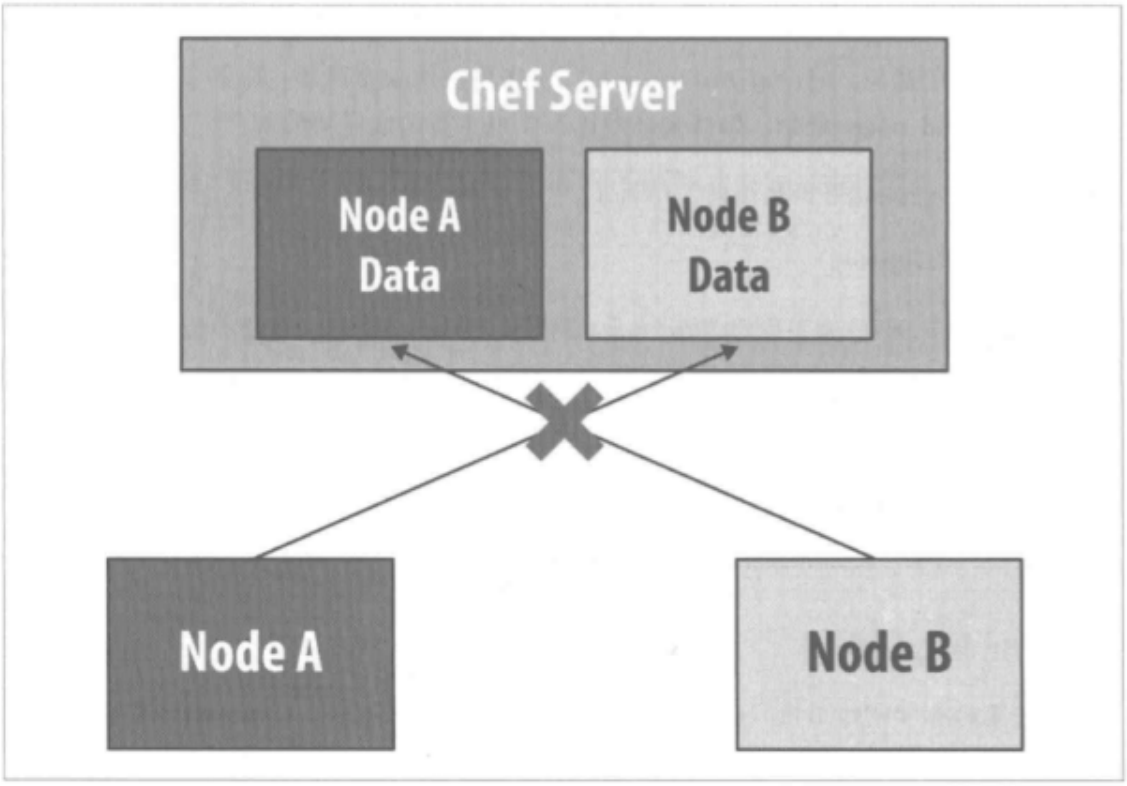


图13-3：Chef服务器不提供节点间直接共享数据的机制

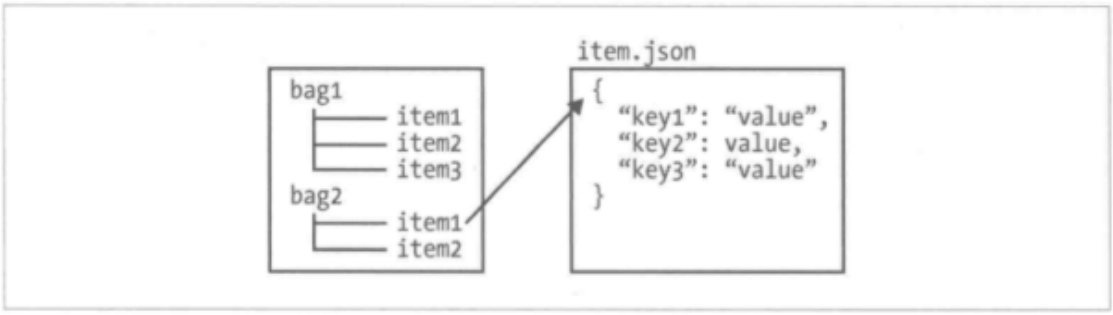


图13-4：数据包内容

使用knife在命令行进行数据包的基本操作

实际的例子：

任务：

现在需要用knife命令在命令行执行一个搜索查询开始。

分配：

确保员工alice和riku在所有节点上都有他们的用户账户。

方法：

用数据包来存储这些用户列表。

结果：

新员工被添加到这个列表中并自动创建他们的账户。用户列表可以在所有节点上被访问

```
cd chef-repo
mkdir -p data_bags/users

# 创建一个.json文件来表示数据包中的每个项目
# 每个数据包项目包含表示一个Unix用户的键值对
# che-repo/data_bugs/users/alice.json
{
    "id":"alice",
    "comment":"Alice Jones",
    "uid":2000,
    "gid":0,
    "home":"/home/alice",
    "shell":"/bin/bash"
}

# chef-repo/data_bugs/users/riku.json
{
    "id":"riku",
    "comment":"riku jinjin",
    "uid":2001,
    "gid":0,
    "home":"/home/riku",
    "shell":"/bin/bash"
}

# 在chef服务器上创建名为users的数据包，运行knife data_bag create命令
knife data_bag create users

#创建数据包项目，运行knife data_bag from file命令
#该命令表示数据包项目的.json文件在data_bags目录下的以数据包名字命名的子目录中:
knife data_bag from file users alice.json
knife data_bag from file users riku.json
```

要搜索服务器上的数据包，以数据包名字作为index参数使用knife search.

在本例中，我们的数据包名为users.

以下命令将搜索我们在users数据包中创建的所有数据包项目。

```
knife search users "*:*"
knife search users "id:alice"
```

在配方单中使用数据包项目的数据创建本地用户

```
# 当前目录时chef-repo/cookbooks

#chef-workstation
chef generate cookbook users
cd users

#chef-client
knife cookbook create users --cookbook-path .
cd users
kitchen init --create-gemfile
bundle install


# chef-repo/cookbooks/users/recipes/default.rb

search("users","*:*").each do |user_data|
    user user_data["id"] do
        comment user_data["comment"]
        uid user_data["uid"]
        gid user_data["gid"]
        home user_data["home"]
        shell user_data["shell"]
    end
end
```

遍历数据包的每个项目并将每个项目的内容存在user_data变量中。user_data是一个字典（哈希）包含数据包项目的键值对。

search()代码块中的user语句是一个chef资源。

user资源在节点上创建本地用户。

接受以下属性：

- comment
  - 关于要创建的用户的信息
- uid
  - 以数字表示的用户ID
- gid
  - 以数字表示的用户的组的ID
- home
  - 用户跟目录的位置
- shell
  - 用户登录的shell

配方单中的代码读取user_data哈希并将其传递进chef的user资源。

```
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
alice:x:2000:0:Alice Jones:/home/alice:/bin/bash
riku:x:2001:0:riku jinjin:/home/riku:/bin/bash
[root@web1 ~]#
```

加密数据包

数据包项目可以被共享密钥加密使其可以在chef服务器中存储高度安全的信息。

比如可以使用加密数据包存储：

- SSL证书
- SSH密钥
- 密码
- 许可证号码

```
# 创建一个数据包时，一个包含密钥的文件被传递进来。数据包的内容被这个密钥加密后存储。
# 当节点试图解密并访问数据包内容时，它需要使用同一个密钥。
knife data bag create
```

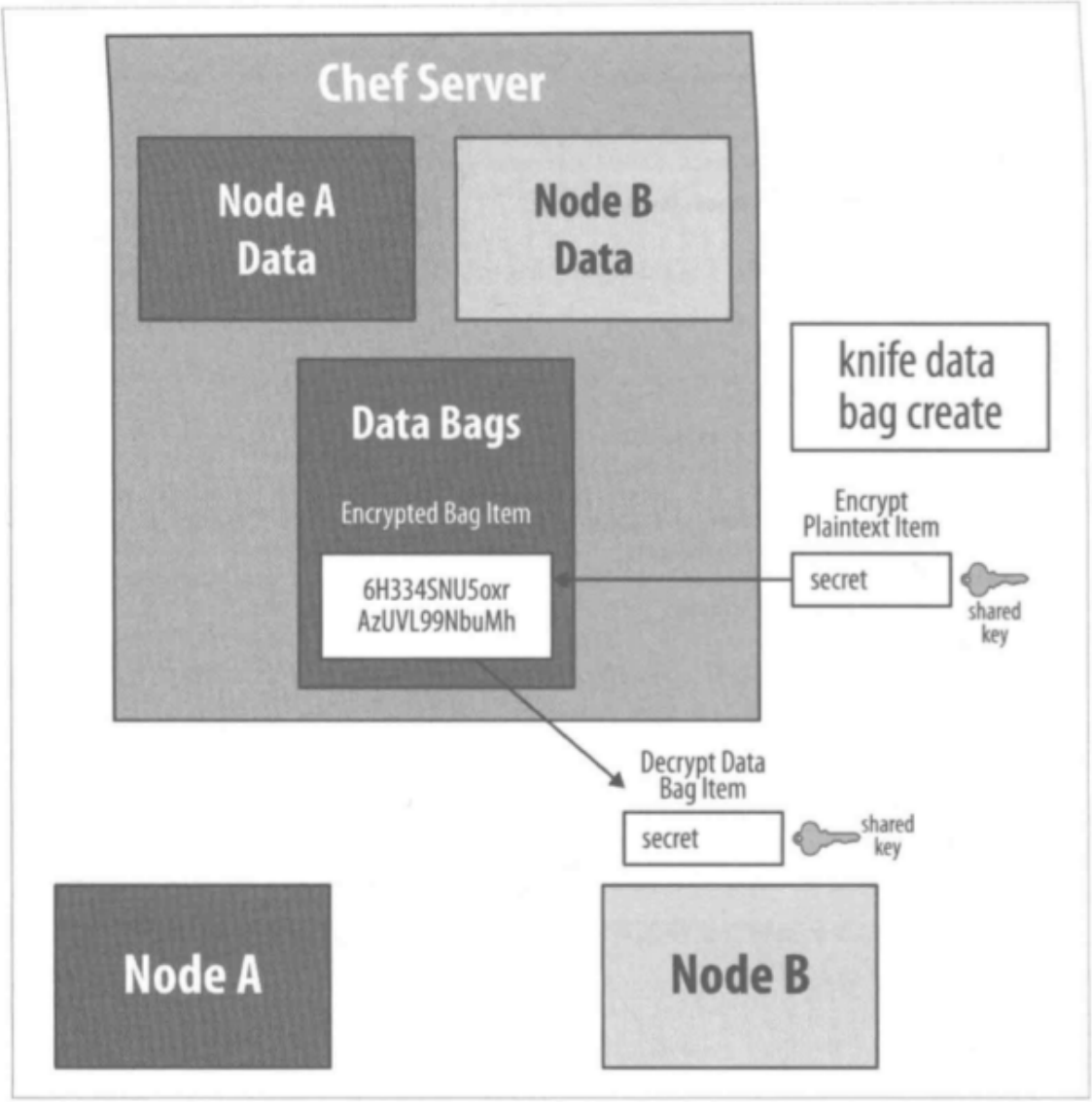


图13-5：加密数据包使用密钥

加密数据包实例：

```
# 当前在chef-repo下
# 生成一个作为密钥的密码。以下命令将生成一个512字节的随机密钥并保存至encrypted_data_bag_secret文件:
opensll rand -base64 512 | tr -d '\r\n' > encrypted_data_bag_secret

# 信用卡数据包需要加密
mkdir -p data_bags/api_keys
# chef-repo/data_bags/api_keys/payment_system.json
{
    "id": "payment",
    "api_key": "5224242-f324-e333-9j02-8c323csfken"
}
# 使用以下命令行创建数据包:
knife data bag create api_keys
```

```
14
15  # 数据包项目需要加密时，使用--secret-file参数来传递密钥
16  knife data bag from file api_keys payment.json --secret-file
    encrypted_data_bag_secret
17
18  #确认创建的数据包项目在服务器上是否被加密?
19  knife data bag show api_keys payment
20
21  #解密数据包项目的数据
22  knife data bag show api_keys payment \
23  --secret-file encrypted_data_bag_secret
```

chef-vault

解决分发密钥的问题。

由于chef本身对于节点的身份验证已经使用了公钥，私钥验证方法，而这个机制已经需要分发公钥到节点上。

利用这个机制，当数据包项目被创建时，在节点上创建一个共享密钥，然后，在需要访问这个数据包项目的节点上，用节点的公钥加密这共享密钥，然后将加密的共享密钥存储在chef服务器上。

```
1   mkdir -p data_bags/password
2   chef-repo/data_bags/passwords/mysql_root.json
3
4   {
5       "id": "mysql_root",
6       "password": "This is a very secure password"
7   }
8
9   knife vault create passwords mysql_root \
10  --json data_bags/passwords/mysql_root.json --search "*:*" \
11  --admins "admin" --mode client
12
13  # 通过--search或--admins参数指定拥有合法客户端密钥的用户或节点。
```

chef-vault只有在chef服务器拥有有效的客户端——密钥时才可以加密数据。

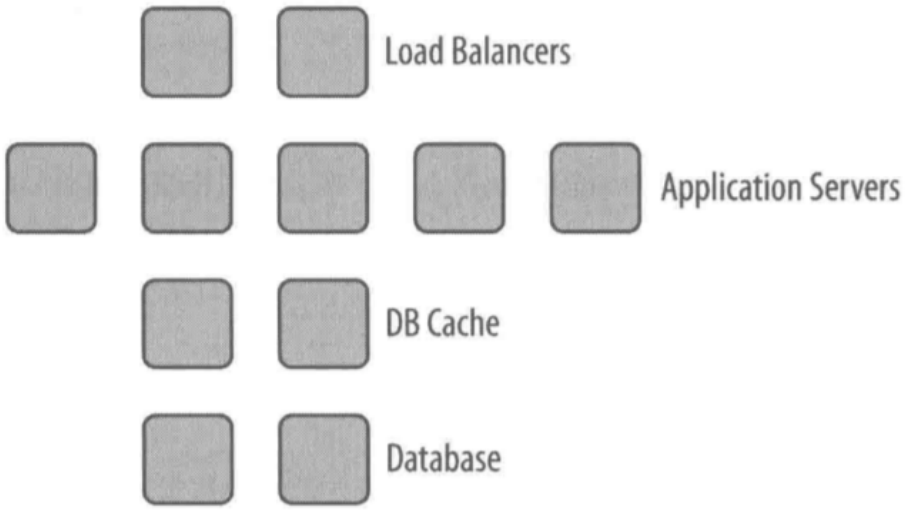
在chef-zero环境下这可能不容易配置。

```
#确认chef-vault创建的数据包项目是否真的加密?
knife data bag show passwords mysql_root

#解密数据包项目内容:
knife vault show passwords mysql_root --mode client
```

第十四章：角色

角色可以在逻辑上对执行同样工作的节点进行分类。

角色可以帮助我们将基础架构内的不同服务分类



角色可以用来表示基础架构内的不同服务器：

- 负载均衡服务器
- 应用程序服务器
- 数据库缓存
- 数据库
- 监视服务器

虽然可以直接将需要运行的配方单加入到节点的运行清单中，但那不是配置基础架构正确和有效的做法，我们通常描述某种服务器时：

- 它是一台网页服务器
- 它是一台数据库服务器
- 它是一台系统监视服务器

可以通过角色方便地指定配方单和属性，将某台服务器配置成理想的服务器。

通过使用角色可以同时将许多节点配置成某种服务器而无需重复工作。

将一些常用的功能集合在一起作为一个角色也是很常见的。最明显的例子是创建一个包含基础架构内每一个节点都需要运行的配方单基本角色，然后让每个其他角色包含这个基本角色。

创建一个网页服务器角色

```
cd chef-repo
#添加一个节点
knife bootstrap web2  -x  root   --node-name web2

mkdir roles
# 最基本的角色包含name:(名字), description:(描述), 和run_list(运行清单)
# 一个角色可以用来把一个很长的配方单清单组织成一个单位。

# chef-repo/roles/webserver.json
{
    "name": "webserver",
    "description": "Web Server",
    "json_class": "Chef::Role",
    "run_list":[
        "recipe[motd]",
        "recipe[users]",
        "recipe[apache]"
    ]
}
# 在服务器上创建这个角色
knife role from file webserver.json

# 查看服务器上创建的角色
knife role show webserver

# 在服务器上设定节点的运行清单
knife node run_list set web2  "role[webserver]"
```

在chef运行时，运行清单中的网页服务器（webserver）角色将展开至该角色的运行清单：

- recipe[motd]

- recipe[users]
- recipe[apache]

角色是一个强大的抽象概念，它允许你按功能将基础架构加以分类，一个角色包含很多配方单时很常见的。

如果没有角色功能，将需要重复将几十个配方单一个一个添加到数百个节点的每个运行清单中。

属性和角色

角色同时可以包含属性

```
# chef-repo/roles/base.json
{
    "name": "base",
    "description": "common recipes for all nodes",
    "json_class": "Chef::Role",
    "chef_type": "role",
    "run_list": [
        "recipe[chef-client::delete_validation]",
        "recipe[chef-client]"
    ],
    "default_attributes": {
        "chef_client": {
            "init_style": "runit"
        }
    }
}

knife role from file base.json

# 可以看到运行清单中的项目以及此base角色的属性
knife role show base
```

角色中的属性也有优先级，它的属性设计为全局设定，比菜谱中设定的属性优先级更高。

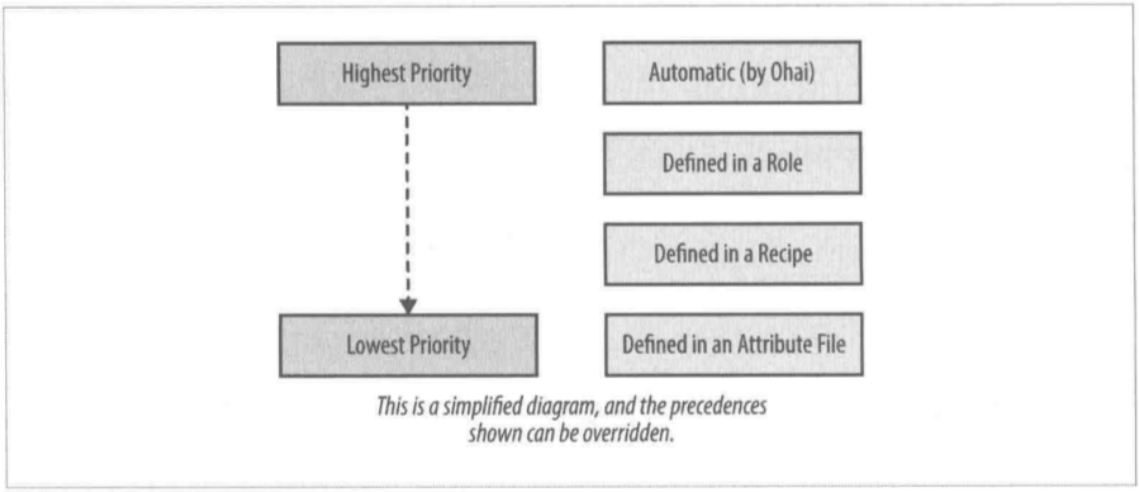


图14-2. 角色属性优先级结构

角色和搜索

```
knife search role "run_list::recipe\[apache\]"
```

角色菜谱

当一个角色被改变时，它会实时影响整个基础架构

因为菜谱时支持版本的，许多chef工程师使用角色菜谱替代角色中的运行清单。他们仍然使用角色中的属性，只是把运行清单移到了菜谱中，一个配方单可以通过使用include_recipe命令来轻松模拟运行清单。

```
include_recipe "motd"
inclued_recipe "user"
inclued_recipe "apache"
```

作为分类用途以及使用角色属性，节点仍然使用webserver角色，webserver角色的运行清单中仅包含webserver菜谱。

因此，我们仍然可以通过搜索找到所有网页服务器：

```
knife search node role:webserver
```

小节：

角色提供了将基础架构中的节点分类的功能，角色可以包含属性及包含配方单或其他角色的运行清单。角色允许你对一类的节点指定特定配置，而无需对每个单独节点做同样重复的工作。

第十五章：环境

环境提供另一种抽象，允许你针对特定环境指定属性以及菜谱的版本。

chef服务器支持“环境”功能为软件开发生命周期的每一个阶段建模

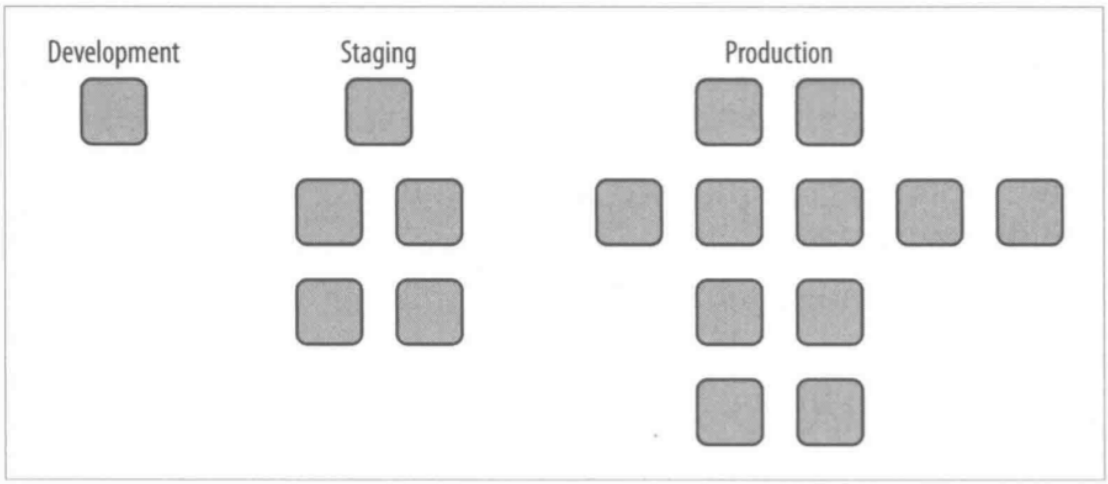


图15-1：环境概览

环境反映模式和工作流，并可以用来表示应用程序的各个生命阶段，比如：

- 开发
- 测试
- 模拟生成环境
- 生成环境

默认情况下，chef服务器只有一个名为_default的环境。

环境可以包含配置基础架构所需的属性，比如：

- 某付款服务API的URL
- 程序包存储源的位置
- 所需使用的chef配置文件的版本

和角色不一样，环境支持版本约束（指定在该环境下使用 的菜谱的版本），因此允许在chef服务器上对不同的环境提供不同的资源。

创建一个开发环境

```
cd chef-repo
mkdir environments
```

基本的环境需要拥有name:(名字)和description:(描述)。

环境也同时可以包含对菜谱的版本约束，能够在某个环境内指定特定的菜谱版本是环境最有用的功能。

```
# chef-repo/environments/dev.json
{
    "name": "dev",
    "description": "For developers!",
    "cookbook_versions": {
        "apache": "= 0.2.0"
    },
    "json_class": "Chef::Environment",
    "chef_type": "environment"
}
```

```
knife environment from file dev.json
knife environment show dev
```

属性和环境

环境可以包含属性。

创建一个表示生成（production）环境的.json文件。

这个环境将会约束apache菜谱的版本到0.1.0版本。

确保在生成环境下每日消息显示一个对于生产环境自己的特别消息。

```
# chef-repo/environments/production.json
{
    "name": "production",
    "description": "For prods!",
    "cookbook_versions": {
        "apache": "= 0.1.0"
```

```
    },
    "json_class": "Chef::Enviroment",
    "chef_type": "enviroment",
    "override_attributes": {
        "motd": {
            "message": "A production-worthy message of the day"
        }
    }
}
```

```
knife environment from file production.json
knife environment show production
```

环境属性的优先级

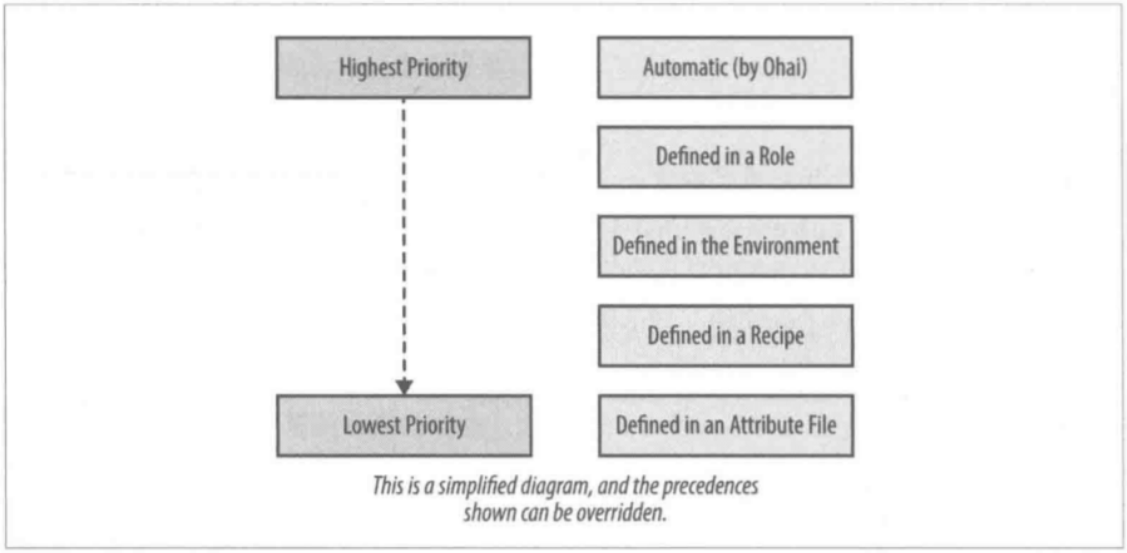


图15-2：环境属性优先级

实例：

```
template '/etc/httpd/conf.d/custom.conf' do
 ...
 variables(
    :document_root => node['apache']['document_root'],
    :port => node['apache']['port']
 )
 ...
 end

# 通过variables()属性传递一个映射来指定模板中变量的值，这样我们可以从配方单中传递变量给模板，使模板中也可以使用更短，易读的变量名
```

```
# chef-zero/apache/recipes/default.rb

package 'httpd'

service 'httpd' do
  action [ :enable, :start ]
end

# Add a template for Apache virtual host configuration
template '/etc/httpd/conf.d/custom.conf' do
  source 'custom.erb'
  mode '0644'
  variables(
    :document_root => node['apache']['document_root'],
    :port => node['apache']['port']
  )
  notifies :restart, 'service[httpd]'
end

document_root = node['apache']['document_root']

# Add a directory resource to create the document_root
directory document_root do
  mode '0755'
  recursive true
end

template "#{document_root}/index.html" do
  source 'index.html.erb'
  mode '0644'
```

```
  variables(
    :message => node['motd']['message'],
    :port => node['apache']['port']
  )
end

```

ruby的模板语言

```
<% %> 写条件逻辑
<%= %> 表示变量
-%> 表示这一行不会被渲染在输出的文件中
```

```
<% if @port != 80 -%>    #不会写入最终输出的文件
  Listen <%= @port %>    #将会写入
<% end -%>               #将会写入

<VirtualHost *:<%= @port %>>
  ServerAdmin webmaster@localhost

  DocumentRoot <%= @document_root %>
  <Directory />
    Options FollowSymLinks
    AllowOverride None
  </Directory>
  <Directory <%= @document_root %>>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Order allow,deny
    allow from all
  </Directory>
</VirtualHost>
```

第十六章：测试

在部署chef代码到生产环境之前执行测试和验证确保它能产生预期的结果。

基础架构的改变最好是渐进式

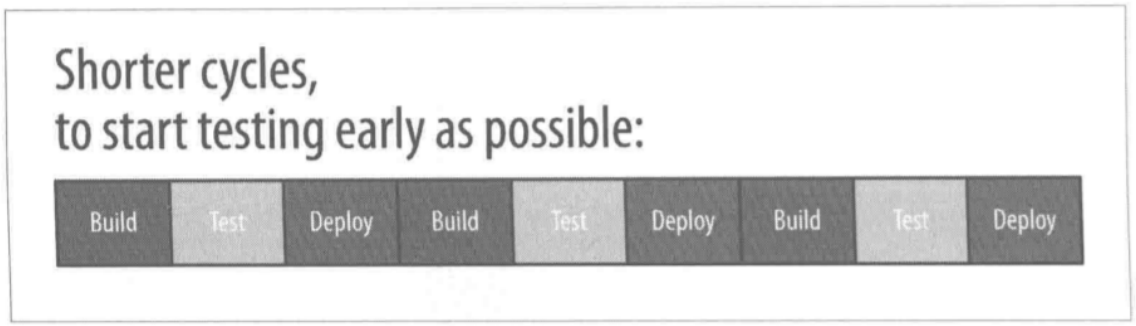


图16-1：使用渐进式的小规模开发->测试->部部署周期

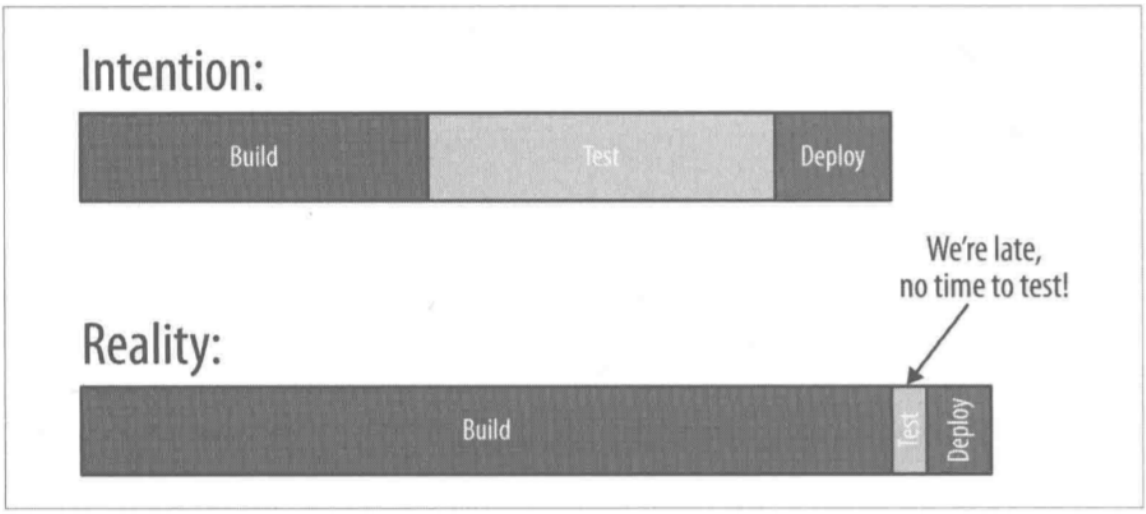


图16-2：在冗长的开发周期中测试时间经常被牺牲

寻找和修复软件和基础架构代码中的问题是需要成本的，而且发现问题的时间越接近周期结束修复这些问题的成本就越高。

- 需求
- 设计
- 开发
- 开发测试
- 整合测试
- 生产环境

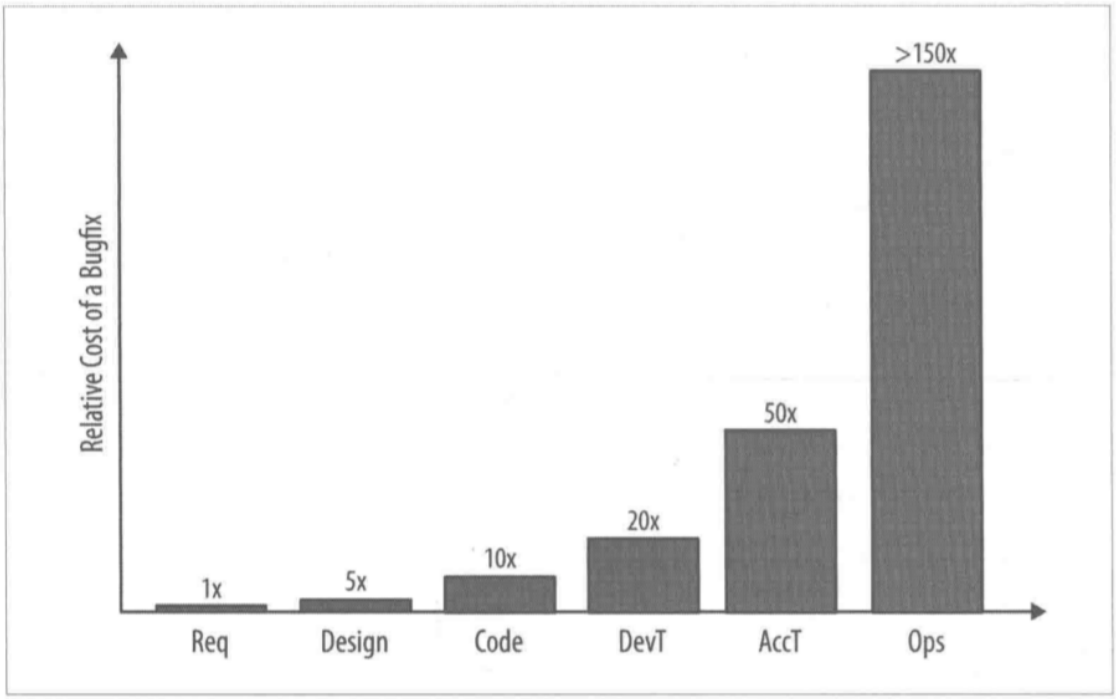


图16-3：开发流程中修复问题的成本

在做chef项目时应做到以下几点。

1. 第一次就做对以免未来再修复问题。
2. 越早发现问题越好，越接近写代码的时候就发现问题越好。
3. 尽可能做渐进式的小改变，测试小的改变要容易很多。

chef提供的测试工具在攥写菜谱时尽早对代码中的问题提供反馈

图16-4：每个开发阶段中的Chef测试工具

- 编辑器中打入chef代码的时候
  - 使用Foodcritic可以分析你的chef编程风格
- 在部署到测试节点之前
  - 使用ChefSpec可以帮助你保证代码的组织和文档符合标准
- 在部署到测试节点后
  - 使用Serverspec验证菜谱产生的行为与预期相符

> 在自动化输出中，术语_example_代表一个测试案例

重温Apache菜谱

生产一个apache-test菜谱

```
chef generate cookbook apache-test
cd apache-test

# chef-repo/apache-test/recipes/default.rb

package "httpd"
service "httpd" do
    action [:enable, :start ]
end

template "/var/www/html/index.html" do
    source 'index.html.erb'
    mode '0644'
end

# 创建一个ERB模板
chef generate template index.html

# 对index.html做一些改变
This site was set up by <%= node['hostname'] %>

# 如果标签中没有等号，它则作为脚本执行。
<% for @interface in node['network']['interfaces'].keys %>
    * <%= @interface %>
<% end %>
```

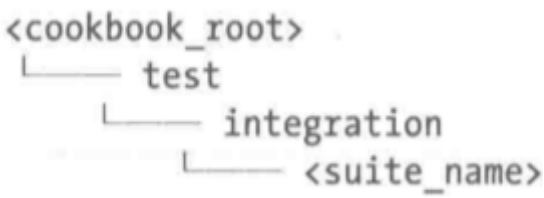
在测试节点上有三个网络界面lo,eth0和eth1,ERB中的脚本将渲染以下输出：

- lo
- eth0
- eth1

```
# chef-repo/apache-test/templates/default/index.html.erb
<html>
<body>
<pre><code>
This site was set up by <%= node['hostname'] %>
My network addresses are:
<% node['network']['interfaces'].keys.each do |iface_name| %>
  * <%= iface_name %>:
      <%= node['network']['interfaces'][iface_name]
['addresses'].keys[1] %>
<% end %>
</code></pre>
</body>
</html>
```

使用Serverspec进行自动化测试

默认情况下，Test kitchen会在test/integration子目录中寻找测试相关的文件。Serverspec在test/integration目录下的一些子目录中寻找它的测试文件。首先，在test/integration目录下需要针对每个测试套件的名字（suite name)有一个对应的测试目录。



套件（suite）：

默认是default，可以使用Test kitchen的套件功能对不同的运行清单和属性配置运行不同的自动化测试。

需要为apache-test菜谱创建一个测试目录

```
mkdir -p test/integration/default
```

在test/integration/default目录下创建一个子目录来告诉Test Kitchen我们希望使用Serverspec目录下的目录结构中获取此信息。

```
mkdir -p test/integration/default/serverspec
```

根据约定，Serverspec预期包含测试代码的文件以spec.rb结尾

```
require 'spec_helper'  # 1

set :os, :family => 'redhat', :release => 6  # 2
set :backend, :exec

describe 'web site' do  #3
  it 'responds on port 80' do
    expect(port 80).to be_listening 'tcp'
  end

  it 'returns eth1 in the HTML body' do
    expect(command('curl localhost:80').stdout).to match
/eth1/
  end

  it 'has the apache web server installed' do
    expect(package 'httpd').to be_installed
  end
end
```

1. require语句用来加载serverspec的gem库，这样我们可以引用Serverspec的类和方法，比如set方法
2. set语句让我们配置serverspec如何运行，在本例中，我们将：backend属性设定为：exec来告诉serverspec测试代码将会在同一台机器运行。
3. 测试用RSpec DSL（领域专用语言)写，使用describe和it语句。在本例中，我们使用RSpec DSL写检查网站是否使用TCP协议监听80端口（HTTP网站默认端口）的测试。

要运行这段代码，首先需要确保所有的gem文件在测试节点已经安装好，我们

RSpec DSL语法

RSpec DSL使用describe代码块包含一组测试，每个describe代码块由以下格式定义：

```
describe '<entity>' do  # entity是描述被测试的实体
    <tests here>
end
```

describe代码块的用途是将测试根据意图分组并描述被测试的实体。

每个分组的描述作为字符串传递给describe

这个字符串描述是给运行测试的人作为文档在测试输出中查看

实际的测试代码在describe代码块里面的it代码块中定义，格式如下：

```
describe '<entity>' do
    it '<description>'
        <examples here>
    end
end
```

it 代码块也接受一个字符串参数来作为该具体测试的文档。

比如，在示例16-4中，我们提供字符串‘responds on port 80’来说明我们的测试检查网站是否在80端口下运行：

```
describe 'web site' do
    it 'responds on port 80' do
        ...
     end
end
```

测试应该以expect格式写。

```
describe 'web site' do
    it 'responds on port 80' do
        expect(资源).to 匹配器 匹配器参数
     end
end
```

- 资源（resource）,也称为主题（subject）或命令（command）,是expect代码块的第一个参数，它代表被测试的实体。诸如Serverspec和CHefSpec的测试框架提供特定的资源来执行广泛的验证。
- 匹配器（matcher）用来定义对于资源的等同或相反的期待值。它的格式以expect（资源）.to来表示期待该资源等同于匹配器结果，以及以expect(资源).not_to表示期待该资源的结果相反于匹配器结果

```
describe 'web site' do
    it 'responsponds on port 80' do
        expect(port 80).to be_listening 'tcp'
    end
end
```

serverspec.org

Serverspec详解(p277)

供多个测试文件共享的代码可以移到spec_helper.rb文件中。

kitchen test命令可以按顺序执行以下全部命令：

- kitchen destroy（如果需要的话）
- kitchen create
- kitchen converge
- kitchen setup
- kitchen verify
- kitchen destroy

使用cookstyle来检查chef代码

```
cookstyle .
```

ChefSpec

```
describe '<recipe_name>' do
    <perform in-memory chef run>
    <example here>
 end
```

```
# 测试apache-test::default菜谱
require 'chefspec'

describe 'apache::default' do
    chef_run = ChefSpec::Runner.new.converge('apache-
test::default')
    <descriptions here>
end
```

```
#展示chefspec检查chef代码，查看是否已经安装httpd程序包
require 'chefspec'

describe 'apache::default' do
    chef_run = ChefSpec::Runner.new.converge('apache-
test::default')

    it 'installs apache2' do
         expect(chef_run).to install_package('httpd')
    end
end
```