**云计算架构实战**

作者：刘彬

联系：[heidsoftg@gmail.com](mailto:heidsoftg@gmail.com)

Blog:https://github.com/heidsoft

# 大数据

**数据库技术、并行计算、分布式计算**

数据挖掘学科之间的交叉

### 术语解释

1. R、SAS、SPSS、S-PLUS、MATLAB中的一种数据分析环境
2. OLAP技术，包括关系型OLAP技术，以及立方、切片等多维OLAP技术
3. 流式处理的相关技术原理，了解CEP的基本原理
4. 数据呈现技术（特别是报表设计）
5. 运用回归、分类、聚类、关联规则、图计算的常见算法
6. 数学模型和统计模型，熟练掌握常用数学建模方法和统计分析方法
7. 静态页面生成技术、Google Web Toolkit
8. 分布式列存数据库、ETL、数据质量、商业智能应用构建工具、大规模数据云计算
9. 数据库基础、具有数据库索引优化、查询优化和存储优化经验
10. 大规模系统的负载均衡、网络存储、网络安全、数据库高可用设计及性能评估机制
11. 掌握主流应用服务器框架、分布式数据库、缓存、文件系统、消息系统等技术
12. OOA,OOP,OOD的软件编程设计
13. android系统的Custom Building System

### 什么是数据挖掘

1. 【数据挖掘是在大型数据存储库中，自动地发现有用信息的过程】
2. 【数据挖掘技术用来探查大型数据库，发现先前未知的用用模式】
3. 【数据挖掘又是数据库知识发现（KDD）的重要组成部分】

### 数据挖掘要解决什么问题

伸缩性

高纬性

异种数据和复杂数据

数据的所有权与分布

非全统分析

### 数据挖掘的任务

预测任务：任务的目标是根据其他属性的值，预测特定属性的值。被预测的属性一般称目标变量或者因变量，因而用来做预测的属性称为明变量或自变量。

描述任务：其目标是导出概括数据中潜在联系的模式(相关、趋势、聚类、轨迹和异常)

## NoSql

Nosql简介<http://www.nosql-database.org/>

## leveldb

<https://code.google.com/p/leveldb/>

## mongodb

<http://www.mongodb.org/>

## redis

<http://www.redis.io/>

## cassandra

<https://cassandra.apache.org/>

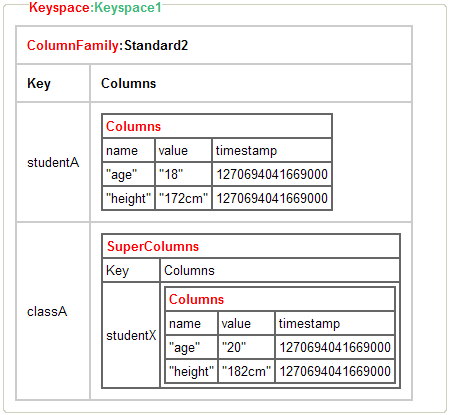
<http://www.datastax.com/documentation/gettingstarted/index.html?pagename=docs&version=quick_start&file=quickstart>

分布式 Key-Value 存储系统：Cassandra 入门

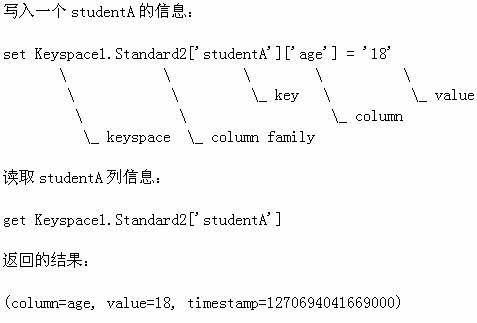
Cassandra 的数据存储结构

Cassandra 的数据模型是基于列族（Column Family）的四维或五维模型。它借鉴了 Amazon 的 Dynamo 和 Google's BigTable 的数据结构和功能特点，采用 Memtable 和 SSTable 的方式进行存储。在 Cassandra 写入数据之前，需要先记录日志 ( CommitLog )，然后数据开始写入到 Column Family 对应的 Memtable 中，Memtable 是一种按照 key 排序数据的内存结构，在满足一定条件时，再把 Memtable 的数据批量的刷新到磁盘上，存储为 SSTable 。

**图 1. Cassandra 的数据模型图：**

1. 
2. Cassandra 的数据模型的基本概念：
3. 1. Cluster : Cassandra 的节点实例，它可以包含多个 Keyspace  
   2. Keyspace : 用于存放 ColumnFamily 的容器，相当于关系数据库中的 Schema 或 database3. ColumnFamily : 用于存放 Column 的容器，类似关系数据库中的 table 的概念 4. SuperColumn ：它是一个特列殊的 Column, 它的 Value 值可以包函多个 Column5. Columns：Cassandra 的最基本单位。由 name , value , timestamp 组成

下面是关于数据模型实例分析 :

**图 2. 数据模型实例分析**  


[**回页首**](http://www.ibm.com/developerworks/cn/opensource/os-cn-cassandra/index.html#ibm-pcon)

**Cassandra 节点的安装和配置**

**获取 Cassandra**

|  |
| --- |
| # wget http://labs.renren.com/apache-mirror/cassandra/0.6.0/apache-  cassandra-0.6.0-rc1-bin.tar.gz  # tar -zxvf apache-cassandra-0.6.0-rc1-bin.tar.gz  # mv apache-cassandra-0.6.0-rc1 cassandra  # ls Cassandra |

**Cassandra 的目录说明**

|  |  |
| --- | --- |
| bin | 存放与 Cassandra 操作的相关脚本 |
| conf | 存放配置文件的目录 |
| interface | Cassandra 的 Thrift 接口定义文件，可以用于生成各种编程语言的接口代码 |
| Javadoc | 源代码的 javadoc |
| lib | Cassandra 运行时所需的 jar 包 |

**配制 Cassandra 节点的数据存储目录和日志目录**

修改配制文件 storage-conf.xml：

**默认的内容**

|  |
| --- |
| <CommitLogDirectory>/var/lib/cassandra/commitlog</CommitLogDirectory>  <DataFileDirectories>  <DataFileDirectory>/var/lib/cassandra/data</DataFileDirectory>  </DataFileDirectories> |

**配置后的内容**

|  |
| --- |
| <CommitLogDirectory>/data3/db/lib/cassandra/commitlog</CommitLogDirectory>  <DataFileDirectories>  <DataFileDirectory>/data3/db/lib/cassandra/data</DataFileDirectory>  </DataFileDirectories> |

修改日志配制文件 log4j.properties：

**log4j.properties 配置**

|  |
| --- |
| # 日志路径  #log4j.appender.R.File=/var/log/cassandra/system.log  # 配置后的日志路径 :  log4j.appender.R.File=/data3/db/log/cassandra/system.log |

创建文件存放数据和日志的目录

|  |
| --- |
| # mkdir – p /data3/db/lib/cassandra  # mkdir – p /data3/db/log/Cassandra |

**配制完成后，启动 Cassandra**

|  |
| --- |
| # bin/Cassandra |

显示信息

|  |
| --- |
| INFO 09:29:12,888 Starting up server gossip  INFO 09:29:12,992 Binding thrift service to localhost/127.0.0.1:9160 |

看到这两行启动回显信息时，说明 Cassandra 已启动成功。

**连接到 Cassandra 并添加、获取数据**

Cassandra 的 bin 目录已自带了命令行连接工具 cassandra-cli，可使用它连接到 Cassandra，并添加、读取数据。

**连接到 Cassandra，并添加、读取数据**

|  |
| --- |
| # bin/cassandra-cli --host localhost --port 9160  Connected to: "Test Cluster" on localhost/9160  Welcome to cassandra CLI.  Type 'help' or '?' for help. Type 'quit' or 'exit' to quit.  cassandra>  cassandra> set Keyspace1.Standard2['studentA']['age'] = '18'  Value inserted  cassandra> get Keyspace1.Standard2['studentA']  => (column=age, value=18, timestamp=1272357045192000)  Returned 1 results |

**停止 Cassandra 服务**

**查出 Cassandra 的 pid:16328**

|  |
| --- |
| # ps -ef | grep cassandra  # kill 16328 |

**Cassandra 配制文件 storage-conf.xml 相关配制介绍**

**清单 1. storage-conf.xml 节点配制说明清单**

|  |
| --- |
| <!-- 集群时显示的节点名称 -->  <ClusterName>Test Cluster</ClusterName>  <!-- 节点启动时，是否自动加入到集群中，默认为 false -->  <AutoBootstrap>false</AutoBootstrap>  <!-- 集群的节点配制 -->  <Seeds>  <Seed>127.0.0.1</Seed>  </Seeds>  <!-- 节点之间通迅的监听地址 -->  <ListenAddress>localhost</ListenAddress>  <!--  基于 Thrift 的 cassandra 客户端监听地址，  集群时设为：0.0.0.0 表示侦听所有客户端 , 默认为：localhost  -->  <ThriftAddress>localhost</ThriftAddress>  <!-- 客户端连接的端口 -->  <ThriftPort>9160</ThriftPort>  <!--  FlushDataBufferSizeInMB 将 memtables 上的数据写入在 Disk 上，  超过设定好的限制大小时 ( 默认 32M)，则将数据写入磁盘，  FlushIndexBufferSizeInMB 超过设定的时长（默认 8 分钟）后，  将 memtables 由的数据写入磁盘中  -->  <FlushDataBufferSizeInMB>32</FlushDataBufferSizeInMB>  <FlushIndexBufferSizeInMB>8</FlushIndexBufferSizeInMB>  <!--  节点之间的日志记录同步模式。  默认：periodic, 对应配制 CommitLogSyncPeriodInMS  启动 batch 时，则对应的配制 CommitLogSyncBatchWindowInMS  -->  <CommitLogSync>periodic</CommitLogSync>  <!-- 默认为每 10 秒同步一次日志记录 -->  <CommitLogSyncPeriodInMS>10000</CommitLogSyncPeriodInMS>  <!--  <CommitLogSyncBatchWindowInMS>1</CommitLogSyncBatchWindowInMS> --> |

[**回页首**](http://www.ibm.com/developerworks/cn/opensource/os-cn-cassandra/index.html#ibm-pcon)

**常用编程语言使用 Cassandra 来存储数据**

在使用 Cassandra 时，通常情况下都需要使用第三方插件 Thrift 来生成与 Cassandra 相关的库文件 , 您可以在 http://incubator.apache.org/thrift 下载此插件，并学习它的使用方法。以下是分别在 Java、PHP、Python、C#、Ruby 五种常用编程语言中使用 Cassandra：

**Java 程序使用 Cassandra**

把 libthrift-r917130.jar，apache-cassandra-0.6.0-rc1.jar 加入到 Eclipse 的编译路径中。

建立数据库连接：使用 libthrift-r917130.jar 的 TTransport 的 open 方法建立起与 Cassandra 服务端 (IP:192.168.10.2 端口：9160) 的连接。

数据库操作：使用 Cassandra.Client 创建一个客户端实例。调用 Client 实例的 insert 方法写入数据，通过 get 方法获取数据。

关闭数据库连接：使用 TTransport 的 close 方法断开与 Cassandra 服务端的连接。

**清单 2. Java 连接 Cassandra，写入并读取数据。**

|  |
| --- |
| package com.test.cassandra;|  import java.io.UnsupportedEncodingException;  import org.apache.thrift.transport.TTransport;  import org.apache.thrift.transport.TSocket;  import org.apache.thrift.protocol.TProtocol;  import org.apache.thrift.protocol.TBinaryProtocol;  import org.apache.thrift.TException;  import org.apache.cassandra.thrift.Cassandra;  import org.apache.cassandra.thrift.Column;  import org.apache.cassandra.thrift.ColumnOrSuperColumn;  import org.apache.cassandra.thrift.ColumnPath;  import org.apache.cassandra.thrift.ConsistencyLevel;  import org.apache.cassandra.thrift.InvalidRequestException;  import org.apache.cassandra.thrift.NotFoundException;  import org.apache.cassandra.thrift.TimedOutException;  import org.apache.cassandra.thrift.UnavailableException;  /\*  \* 使 Java 客户端连接 Cassandra 并进行读写操作  \* @author jimmy  \* @date 2010-04-10  \*/  public class JCassandraClient{  public static void main(String[] args) throws InvalidRequestException,  NotFoundException, UnavailableException, TimedOutException,  TException, UnsupportedEncodingException {  // 建立数据库连接  TTransport tr = new TSocket("192.168.10.2", 9160);  TProtocol proto = new TBinaryProtocol(tr);  Cassandra.Client client = new Cassandra.Client(proto);  tr.open();  String keyspace = "Keyspace1";  String cf = "Standard2";  String key = "studentA";  // 插入数据  long timestamp = System.currentTimeMillis();  ColumnPath path = new ColumnPath(cf);  path.setColumn("age".getBytes("UTF-8"));  client.insert(keyspace,key,path,"18".getBytes("UTF-8"),  timestamp,ConsistencyLevel.ONE);  path.setColumn("height".getBytes("UTF-8"));  client.insert(keyspace,key,path,"172cm".getBytes("UTF-8"),  timestamp,ConsistencyLevel.ONE);  // 读取数据  path.setColumn("height".getBytes("UTF-8"));  ColumnOrSuperColumn cc = client.get(keyspace, key, path, ConsistencyLevel.ONE);  Column c = cc.getColumn();  String v = new String(c.value, "UTF-8");  // 关闭数据库连接  tr.close();  }  } |

**PHP 程序使用 Cassandra**

在 PHP 代码中使用 Cassandra，需要借助 Thrift 来生成需要的 PHP 文件，通过使用 thrift --gen php interface/cassandra.thrift 生成所需要的 PHP 文件，生成的 PHP 文件中提供了与 Cassandra 建立连接、读写数据时所需要的函数。

**清单 3. PHP 连接 Cassandra，写入并读取数据。**

|  |
| --- |
| <?php  $GLOBALS['THRIFT\_ROOT'] = '/usr/share/php/Thrift';  require\_once  $GLOBALS['THRIFT\_ROOT'].'/packages/cassandra/Cassandra.php';  require\_once  $GLOBALS['THRIFT\_ROOT'].'/packages/cassandra/cassandra\_types.php';  require\_once $GLOBALS['THRIFT\_ROOT'].'/transport/TSocket.php';  require\_once $GLOBALS['THRIFT\_ROOT'].'/protocol/TBinaryProtocol.php';  require\_once  $GLOBALS['THRIFT\_ROOT'].'/transport/TFramedTransport.php';  require\_once  $GLOBALS['THRIFT\_ROOT'].'/transport/TBufferedTransport.php';  try {  // 建立 Cassandra 连接  $socket = new TSocket('192.168.10.2', 9160);  $transport = new TBufferedTransport($socket, 1024, 1024);  $protocol = new TBinaryProtocolAccelerated($transport);  $client = new CassandraClient($protocol);  $transport->open();  $keyspace = 'Keyspace1';  $keyUser = "studentA";  $columnPath = new cassandra\_ColumnPath();  $columnPath->column\_family = 'Standard1';  $columnPath->super\_column = null;  $columnPath->column = 'age';  $consistency\_level = cassandra\_ConsistencyLevel::ZERO;  $timestamp = time();  $value = "18";  // 写入数据  $client->insert($keyspace, $keyUser, $columnPath, $value,  $timestamp, $consistency\_level);  $columnParent = new cassandra\_ColumnParent();  $columnParent->column\_family = "Standard1";  $columnParent->super\_column = NULL;  $sliceRange = new cassandra\_SliceRange();  $sliceRange->start = "";  $sliceRange->finish = "";  $predicate = new cassandra\_SlicePredicate();  list() = $predicate->column\_names;  $predicate->slice\_range = $sliceRange;  $consistency\_level = cassandra\_ConsistencyLevel::ONE;  $keyUser = studentA;  // 查询数据  $result = $client->get\_slice($keyspace, $keyUser, $columnParent,  $predicate, $consistency\_level);  // 关闭连接  $transport->close();  } catch (TException $tx) {  }?> |

**Python 程序使用 Cassandra**

在 Python 中使用 Cassandra 需要 Thrift 来生成第三方 Python 库，生成方式： thrift --gen py interface/cassandra.thrift, 然后在 Python 代码中引入所需的 Python 库，生成的 Python 库提供了与 Cassandra 建立连接、读写数据时所需要的方法。

**清单 4. Python 连接 Cassandra，写入并读取数据。**

|  |
| --- |
| from thrift import Thrift  from thrift.transport import TTransport  from thrift.transport import TSocket  from thrift.protocol.TBinaryProtocol import  TBinaryProtocolAccelerated  from cassandra import Cassandra  from cassandra.ttypes import \*  import time  import pprint  def main():  socket = TSocket.TSocket("192.168.10.2", 9160)  transport = TTransport.TBufferedTransport(socket)  protocol = TBinaryProtocol.TBinaryProtocolAccelerated(transport)  client = Cassandra.Client(protocol)  pp = pprint.PrettyPrinter(indent=2)  keyspace = "Keyspace1"  column\_path = ColumnPath(column\_family="Standard1", column="age")  key = "studentA"  value = "18 "  timestamp = time.time()  try:  # 打开数据库连接  transport.open()  # 写入数据  client.insert(keyspace,key,column\_path,  value,timestamp,ConsistencyLevel.ZERO)  # 查询数据  column\_parent = ColumnParent(column\_family="Standard1")  slice\_range = SliceRange(start="", finish="")  predicate = SlicePredicate(slice\_range=slice\_range)  result = client.get\_slice(keyspace,key,column\_parent,  predicate,ConsistencyLevel.ONE)  pp.pprint(result)  except Thrift.TException, tx:  print 'Thrift: %s' % tx.message  finally:  # 关闭连接  transport.close()  if \_\_name\_\_ == '\_\_main\_\_':  main() |

**C# 使用 Cassandra**

在 C# 中使用 Cassandra 需要 Thrift.exe 来生成动态链接库，使用 ./thrift.exe --gen csharp interface/cassandra.thrift 生成所需要的 DLL 文件，生成的 DLL 提供了与 Cassandra 建立连接，读写数据等所需的类和方法，在编程环境中引入生成的 DLL，即可使用。

**清单 5. C# 连接 Cassandra，写入并读取数据。**

|  |
| --- |
| namespace CshareCassandra{  using System;  using System.Collections.Generic;  using System.Diagnostics;  using Apache.Cassandra;  using Thrift.Protocol;  using Thrift.Transport;  class CassandraClient{  static void Main(string[] args){  // 建立数据库连接  TTransport transport = new TSocket("192.168.10.2", 9160);  TProtocol protocol = new TBinaryProtocol(transport);  Cassandra.Client client = new Cassandra.Client(protocol);  transport.Open();  System.Text.Encoding utf8Encoding = System.Text.Encoding.UTF8;  long timeStamp = DateTime.Now.Millisecond;  ColumnPath nameColumnPath = new ColumnPath(){  Column\_family = "Standard1",  Column = utf8Encoding.GetBytes("age")};  // 写入数据  client.insert("Keyspace1","studentA",nameColumnPath,  utf8Encoding.GetBytes("18"),timeStamp, ConsistencyLevel.ONE);  // 读取数据  ColumnOrSuperColumn returnedColumn = client.get("Keyspace1",  "studentA", nameColumnPath, ConsistencyLevel.ONE);  Console.WriteLine("Keyspace1/Standard1: age: {0}, value: {1}",  utf8Encoding.GetString(returnedColumn.Column.Name),  utf8Encoding.GetString(returnedColumn.Column.Value));  // 关闭连接  transport.Close();  }  }} |

**Ruby 使用 Cassandra**

在 Ruby 中使用 Cassandra 需要先安装 gem，安装命令：gem install cassandra

安装完成后，打开 Ruby 的 irb 开始使用 Cassandra。

**清单 6. Ruby 连接 Cassandra，写入并读取数据**

|  |
| --- |
| > require 'rubygems'  > require 'cassandra'  # 建立数据库连接  > cdb = Cassandra.new('Keyspace1',"192.168.10.1:9160", :retries => 3)  # 写入数据  > cdb.insert(:Standard1, 'studentA', {'age' => '18'})  # 读取数据  > cdb.get(:Standard1, :studentA)  # 关闭连接  > cdb.disconnect |

[**回页首**](http://www.ibm.com/developerworks/cn/opensource/os-cn-cassandra/index.html#ibm-pcon)

**搭建 Cassandra 集群环境**

Cassandra 的集群是没有中心节点的，各个节点的地位完全相同，节点之间是通过 gossip 的协议来维护集群的状态。

**以下是两台安装了 Linux 系统的服务器，且初步设置了 Cassandra 环境和启用了端口 7000,9160:**

|  |  |  |
| --- | --- | --- |
| 服务器名 | 端口 | IP 地址 |
| ServiceA | 7000,9160 | 192.168.10.3 |
| ServiceB | 7000,9160 | 192.168.10.2 |

**配制服务器 ServiceA、ServiceB 的 storage-conf.xml 文件**

**ServiceA 的配置**

|  |
| --- |
| <Seeds>  <Seed>192.168.10.3</Seed>  </Seeds>  <ListenAddress>192.168.10.2</ListenAddress>  <ThriftAddress>0.0.0.0</ThriftAddress> |

**ServiceB 的配置**

|  |
| --- |
| <Seeds>  <Seed>192.168.10.3</Seed>  <Seed>192.168.10.2</Seed>  </Seeds>  <ListenAddress>192.168.10.2</ListenAddress>  <ThriftAddress>0.0.0.0</ThriftAddress> |

配制完成后，分别启动 ServiceA 和 ServiceB 上的 Cassandra 服务。

查看 ServiceA 和 ServiceB 是否集群成功，可使用 Cassandra 自带的客户端命令

|  |
| --- |
| bin/nodetool --host 192.168.10.2 ring |

**集群成功则会返回以下类似信息：**

|  |
| --- |
| Address Status Load Range Ring  106218876142754404016344802054916108445  192.168.10.2 Up 2.55 KB 31730917190839729088079827277059909532 |<--|  192.168.10.3 Up 3.26 KB 106218876142754404016344802054916108445 |-->| |

**使用 Cassandra 命令行工具进行集群测试**

从 ServiceB 连接到 ServiceA，可使用命令：

|  |
| --- |
| cassandra-cli -host 192.168.10.3 -port 9160 |

**集群测试一**

|  |
| --- |
| 写入集群数据  ServiceA 连接到 ServiceA:  # set Keyspace1.Standard2['studentAA']['A2A'] = 'a2a'  ServiceB 连接到 ServiceA:  # set Keyspace1.Standard2['studentBA']['B2A'] = 'b2a'  ServiceA 连接到 ServiceB:  # set Keyspace1.Standard2['studentAB']['A2B'] = 'a2b' |

获取集群数据：

|  |
| --- |
| ServiceA 连接到 ServiceA :  # get Keyspace1.Standard2['studentAA'],  get Keyspace1.Standard2['studentBA'],  get Keyspace1.Standard2['studentAB']  ServiceB 连接到 ServiceA :  # get Keyspace1.Standard2['studentAA'],  get Keyspace1.Standard2['studentBA'],  get Keyspace1.Standard2['studentAB']  ServiceA 连接到 ServiceB :  # get Keyspace1.Standard2['studentAA'],  get Keyspace1.Standard2['studentBA'],  get Keyspace1.Standard2['studentAB'] |

**清单 8. 集群测试清单二**

ServiceA 停止 Cassandra 服务，ServiceA 连接到 ServiceB 并写入数据

|  |
| --- |
| # set Keyspace1.Standard2['studentAR']['A2R'] = 'a2R' |

启动 ServiceA，并链接到 ServiceA 本身，读取刚才在 ServiceB 写入的数据

|  |
| --- |
| # bin/cassandra-cli -host 192.168.10.3 -port 9160  # get Keyspace1.Standard2['studentAR'] |

[**回页首**](http://www.ibm.com/developerworks/cn/opensource/os-cn-cassandra/index.html#ibm-pcon)

**总结**

以上我们介绍了 Cassandra 的数据模型、节点安装和配置、常用编程语言中使用 Cassandra 以及 Cassandra 的集群和测试。Cassandra 是一个高性能的 P2P 去中心化的非关系型数据库，可以分布式进行读写操作。在系统运行时可以随意的添加或删降字段，是 SNS 应用的理想数据库。

cassandra-1.2.6-bin.tar.gz在redhat5.4上安装部署

Java >= 1.6

\* tar -zxvf apache-cassandra-$VERSION.tar.gz

\* cd apache-cassandra-$VERSION

\* sudo mkdir -p /var/log/cassandra

\* sudo chown -R `whoami` /var/log/cassandra

\* sudo mkdir -p /var/lib/cassandra

\* sudo chown -R `whoami` /var/lib/cassandra

\* bin/cassandra -f

## Pig

<http://pig.apache.org/>

## hive

<http://hive.apache.org/>

## Hadoop

<http://hadoop.apache.org/>

### Hadoop 2.2.0 安装

下载hadoop2.2.0

配置Jdk

|  |  |
| --- | --- |
| **Daemon** | **Environment Variable** |
| NameNode | HADOOP\_NAMENODE\_OPTS |
| DataNode | HADOOP\_DATANODE\_OPTS |
| Secondary NameNode | HADOOP\_SECONDARYNAMENODE\_OPTS |
| ResourceManager | YARN\_RESOURCEMANAGER\_OPTS |
| NodeManager | YARN\_NODEMANAGER\_OPTS |
| WebAppProxy | YARN\_PROXYSERVER\_OPTS |
| Map Reduce Job History Server | HADOOP\_JOB\_HISTORYSERVER\_OPTS |

### 在ubuntu下单机版安装配置

apt-get install python-software-properties

add-apt-repository ppa:ferramroberto/java

apt-get update

apt-get install ppa-purge

update-alternatives --install

update-java-alternatives --list /usr/lib/mozilla/plugins/mozilla-javaplugin.so mozilla-javaplugin.so /usr/lib/jvm/java-6-sun/jre/lib/i386/libnpjp2.so 1

addgroup hadoop

adduser --ingroup hadoop hduser

su – hduser

ssh-keygen -t rsa -P ""

cat $HOME/.ssh/id\_rsa.pub >> $HOME/.ssh/authorized\_keys

<http://mirror.bit.edu.cn/apache/hadoop/core/hadoop-1.1.2/hadoop-1.1.2-bin.tar.gz>

启动

/usr/local/hadoop/bin/start-all.sh

查看进程

Jps

/usr/local/hadoop/bin/stop-all.sh

### Ubuntu10.10的网络配置

一、配置大概分三类：通过配置文件配置、通过命令配置、通过图形化的网络连接菜单配置。

拨号无线等的没条件实验，不涉及。

主要文件：/etc/network/interfaces，这里是IP、网关、掩码等的一些配置；/etc/resolv.conf这个文件保存DNS的有关信息

主要命令：sudo /etc/init.d/networking restart重启网络，使配置文件的配置失效；sudo route add default gw <IP地址>,设置网关。

二、配置方法

1、通过命令配置

如果你只是想暂时性的修改IP地址，可以选择这种方法。重启网络后会丢失。

1）sudo ifconfig eth0 <IP地址> netmask <子网掩码> up

eth0是网卡。最后的up表示开启网卡，可以不加。

2） sudo route add default gw <网关IP>

设置网关  
3） 设置DNS，这个必须去配置文件修改，同第二种

4）配置主机名称

命令：sudo /bin/hostname <主机名>  
但是系统启动时,会从/etc/hostname来读取主机的名称。

2、通过配置文件配置

1）配置IP网关：sudo gedit /etc/network/interfaces 里面添加

dhcp的：

auto lo  
iface lo inet loopback  
  
auto eth0  
iface eth0 inet dhcp

静态IP的：

[复制代码](javascript:void(0);)

auto lo  
iface lo inet loopback  
#上面的是回环  
#网卡eth0的配置  
auto eth0  
#staic 静态IP  
iface eth0 inet static  
address 192.168.1.5  
netmask 255.255.255.0  
gateway 192.168.1.1  
#network 192.168.3.0   
#broadcast 192.168.3.255   
#后面两条是网络号和广播号，这个可以由其它信息计算，因此无需设置

[复制代码](javascript:void(0);)

这里还可以配置第二IP（虚拟IP）

[复制代码](javascript:void(0);)

auto eth0:1  
iface eth0:1 inet static  
address 192.168.1.60  
netmask 255.255.255.0  
#network x.x.x.x  
#broadcast x.x.x.x  
gateway x.x.x.x

[复制代码](javascript:void(0);)

2）配置DNS： sudo gedit /etc/resolv.conf

nameserver 202.107.117.11

3）主机名：sudo gedit /etc/hostname 里面输入主机名。默认有个主机名，因此这个不是必须的。

4）重启网络：sudo /etc/init.d/networking restart

也可以重启网卡：

sudo ifconfig eth0 down   
sudo ifconfig eth0 up

重启网卡对别的网卡无影响，更推荐一些。

3、通过图形化的网络连接菜单配置

这个在系统-首选项里，也叫做network manager，通过右上角面板里的网络管理小程序可以启用它的配置。

这个有点问题：它和第二种方法共用配置文件，所以会导致种种冲突，使用时要注意。

不如，这两种方法都做了配置，sudo /etc/init.d/networking restart启用的是第二种方法的配置，而点击桌面面板上的网络管理小程序中的auto eth0启用的是第三种的配置。

### HDFS

**HDFS是以流式数据访问模式存储超大文件而设计的文件系统**

**CHD4下载**

<http://www.cloudera.com/content/cloudera-content/cloudera-docs/CDHTarballs/3.25.2013/CDH4-Downloadable-Tarballs/CDH4-Downloadable-Tarballs.html>

CHD4文档

<http://www.cloudera.com/content/cloudera-content/cloudera-docs/CDH4/latest/CDH4-Installation-Guide/CDH4-Installation-Guide.html>

Apache Hadoop NextGen MapReduce (YARN)

ClouderaMailList

* CDH Users: [**https://groups.google.com/a/cloudera.org/forum/#!forum/cdh-user**](https://groups.google.com/a/cloudera.org/forum/#!forum/cdh-user)
* Impala Users: [**https://groups.google.com/a/cloudera.org/forum/#!forum/impala-user**](https://groups.google.com/a/cloudera.org/forum/#!forum/impala-user)
* Hue Users: [**https://groups.google.com/a/cloudera.org/forum/#!forum/hue-user**](https://groups.google.com/a/cloudera.org/forum/#!forum/hue-user)
* Search Users: [**https://groups.google.com/a/cloudera.org/forum/search-user**](https://groups.google.com/a/cloudera.org/forum/#!members/search-user)
* Cloudera Manager Users: [**https://groups.google.com/a/cloudera.org/forum/#!forum/scm-users**](https://groups.google.com/a/cloudera.org/forum/#!forum/scm-users)

### CDH4开发

<http://archive.cloudera.com/cdh4/cdh/4/hadoop/hadoop-yarn/> 软件下载

<http://archive.cloudera.com/cdh4/cdh/4/hadoop/index.html>

     > [**Collection（采集） & Ingestion**](http://www.cloudera.com/content/cloudera/en/developer-community/developer-admin-resources/data-processing-analytics.html#collect)(摄入) (Data Ingest(数据采集), RDBMS Connectivity(关系数据库管理系统连接))

     > [**Storage（存储） & Persistence**](http://www.cloudera.com/content/cloudera/en/developer-community/developer-admin-resources/data-processing-analytics.html#store)（持久） (Filesystem, Distributed Big Data Store, Serialization)

     > [**Transformation（转型）& Enrichment**](http://www.cloudera.com/content/cloudera/en/developer-community/developer-admin-resources/data-processing-analytics.html#transform)（富集） (MapReduce APIs & Frameworks, Workflow Coordination（协调）, Web UI)

     > [**Analysis**](http://www.cloudera.com/content/cloudera/en/developer-community/developer-admin-resources/data-processing-analytics.html#analyze)(分析) (Batch/Real-Time SQL Query, Advanced Analytics)

### Open-Source Projects Inside, or Related to, CDH

|  |  |
| --- | --- |
| * [**Apache Hadoop Core**](http://www.cloudera.com/content/dam/cloudera/Resources/PDF/Ask_Bigger_Questions_Whitepaper.pdf) The foundation for Hadoop: MapReduce, HDFS, and Common * [**Apache Avro**](http://avro.apache.org/) Data serialization system * [**Apache Crunch**](http://incubator.apache.org/projects/crunch.html) (not in CDH) Java library for writing, testing, and running pipelines of MapReduce jobs * [**DataFu**](https://github.com/linkedin/datafu) (CDH4.1 and later) Library of Pig UDFs for data mining and statistics * [**Apache Flume**](http://flume.apache.org/) Aggregator for log and event data * [**Apache HBase**](http://hbase.apache.org/) Scalable record and table store with real-time read/write access * [**Apache Hive**](http://hive.apache.org/) SQL-like language and metadata repository | * [**Hue**](https://github.com/cloudera/hue) Browser-based UI for interacting with Hadoop * [**Impala**](https://github.com/cloudera/impala) (not in CDH) Real-time SQL query engine for HDFS and HBase data * [**Apache Mahout**](https://cwiki.apache.org/confluence/display/MAHOUT/Mahout+Wiki) Library of machine-learning algorithms * [**Apache Oozie**](http://oozie.apache.org/) Workflow engine * [**Apache Pig**](http://pig.apache.org/) High-level data flow language * [**Apache Sqoop**](http://sqoop.apache.org/) Tool for bulk data ingest of relational data * [**Apache Whirr**](http://whirr.apache.org/) Library for running Hadoop in the cloud * [**Apache ZooKeeper**](http://zookeeper.apache.org/) Distributed synchronization service |

## Kettle

<http://www.weka.nl/>

## Ganglia

## mahout

数据切分是从逻辑上或物理上将数据分割成数个更好维护或访问的部分，同时很多流行的开源关系型数据库都支持分片（比如MySQL Partitioning及Postgres Partitionging）。

## Lustre

<http://wiki.lustre.org/index.php/Main_Page>

## OCFS2

<https://oss.oracle.com/projects/ocfs2/>

## mpfs

<http://www.mpfs.de/>

## Gluster

# 虚拟化

## Exsi

## Xen

Xen(发音为Zen)是由XenSource所管理的一个开源GPL项目。Xen是openSuse和Novell主要支持的虚拟化技术。

Xen服务器上的虚拟机以两种模式运行：全虚拟化或半虚拟化。

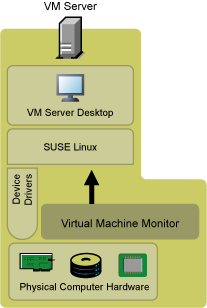
**全虚拟化：**是一种完全模拟所有硬件设备的虚拟化模式。由于VMM（虚拟机监控系统）模拟了所有的计算机设备使得操作系统好像独自占用了所有计算机的资源，所以尽管需要特定的计算机硬件，大多数操作系统都可以以全虚拟化模式运行。完全模拟计算机硬件需要消耗虚拟机服务器更多地CPU资源。所以全虚拟化模式操作系统运行速度会比较慢。

**半虚拟化：**是一种选择性的模拟硬件设备的虚拟化模式。以半虚拟化模式运行的半虚拟化操作系统并不需要完全模拟硬件，因此只需付出更少的管理代价。例如，VMware系统运行时不需要模拟显卡，因此虚拟机服务器就不用模拟视频数据了。所以，以半虚拟化模式运行的操作系统消耗更少的CPU资源并且执行地更好。它也并不需要特定的计算机硬件。

虚拟机监控系统（VMM）在服务器硬件和Suse linux系统内核之间运行。计算机启动时会首先加载VMM,然后以特权模式启动虚拟机服务器。特权模式指的是虚拟机服务器可以创建并控制虚拟机，而且还可对计算机硬件进行直接存取。你需要设置虚拟机服务器与本地设备驱动程序来匹配实际的计算机设备。例如，如果计算机有一个物理网卡e1000，就要设置虚拟机服务器去加载运行e1000的Suse linux驱动程序。

SUSE®Linux Enterprise Server 11 Service Pack 3 (SP3) 随附最新的开放源代码虚拟化技术 Xen 和 KVM。有了这些超级管理程序，SUSE Linux Enterprise Server 就可用于在单个物理系统上安装、监视和管理多个虚拟机 (VM)。

目前Xen开源社区开发和维护着涉及软件层的虚拟机监控程序(VMM) 和 hypervisor，同时它也提供功能性的虚拟机服务器。

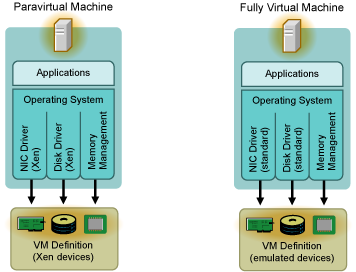
[](https://cn.opensuse.org/File:Xen_arch_10_a.png)

***Figure 2-1*** *虚拟机与设备驱动*

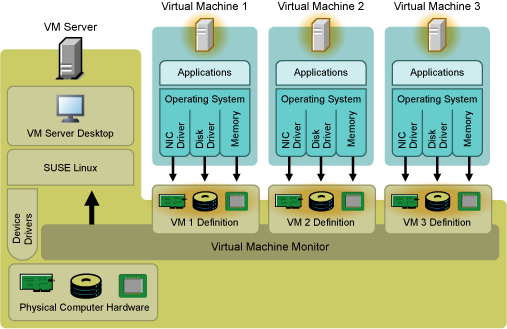
如何定义一个xen虚拟机服务器(Defining a VM),你可以在存放虚拟机镜像的虚拟机服务器上定义一个虚拟机，在配置文件/etc/xen/vm/vm\_name里面去定义。配置文件里面定义了虚拟资源，比如在虚拟机上安装和启动操作系统时你所看到的CPU、内存、网卡及块设备。例如如下表格中定义xen虚拟机服务器的配置。

|  |  |
| --- | --- |
| name="sles11"  description="None"  uuid="a3d58b25-825c-0aa0-235e-2908d65bbcee"  memory=2048  maxmem=2048  vcpus=4  on\_poweroff="destroy"  on\_reboot="restart"  on\_crash="destroy"  localtime=0  keymap="en-us"  builder="hvm"  device\_model="/usr/lib/xen/bin/qemu-dm"  kernel="/usr/lib/xen/boot/hvmloader"  boot="c"  disk=[ 'file:/var/lib/xen/images/sles11/disk0.raw,hda,w', ]  vif=[ 'mac=00:16:3e:21:69:d0,bridge=br0,type=netfront', ]  stdvga=0  vnc=1  vncunused=1  viridian=0  acpi=1  pae=1  serial="pty" | [Xen arch 04 a.png](https://cn.opensuse.org/File:Xen_arch_04_a.png) |

在全虚拟化和半虚拟化两种模式下，虚拟机操作系统都是通过设备驱动与VMM相结合。在全虚拟化模式下，操作系统通过本地标准模拟设备的设备驱动来实现，例如AMD PCNet或者NE2000网卡，IDE磁盘驱动器和VGA显卡。在半虚拟化模式下，含有专有设备驱动（叫Xen驱动)的VM-aware操作系统通过VMM和VM服务器与计算机的物理设备通信。

[](https://cn.opensuse.org/File:Xen_arch_03_a.png)

比如说，如果一个以全虚拟化模式运行的虚拟机操作系统想要保存一个文件到它的20G虚拟磁盘上，操作系统就要通过设备驱动给VMM传送请求。VMM知道了它的500G物理磁盘的哪部分可以给虚拟机进行存取以后，它就会给虚拟机服务器下达指令。虚拟机服务器回去访问磁盘启动器并把文件写入预先确定好的磁盘区域里。 只要有足够的计算能力和可用的计算机资源，你可以再一台虚拟机服务器上创建并同时运行任意多个虚拟机。每一个虚拟机都相互独立地通过VMM和虚拟机服务器平台去使用虚拟或模拟的CPU、内存、块设备和网络资源。

[](https://cn.opensuse.org/File:Xen_arch_11_a.png)

## KVM

KVM（名称来自英语：Kernel-based Virtual Machine的缩写，即基于内核的虚拟机），是一种用于Linux内核中的虚拟化基础设施，可以将Linux内核转化为一个hypervisor。KVM在2007年2月被导入Linux 2.6.20核心中。

### KVM环境构建

我们知道，KVM从诞生第一天就是需要硬件虚拟化扩展的支持，所以这里需要特别讲解一下硬件系统的配置。

KVM最初始的开发是基于**x86**和**x86-64**处理器架构上Linux系统进行的，目前，KVM也被移植到多种不同处理器架构之上，包括：Intel和HP的IA64（安腾）架构、AIM联盟（Apple–IBM–Motorola）的PowerPC架构、IBM的S/390架构、ARM架构（2012年开始[注1]）。其中，在x86-64上面的功能的支持是最完善（主要原因是由于Intel/AMD的x86-64架构在桌面和服务器市场上的主导地位及其架构的开放性，以及它的开发者众多），本书也是采用基于Intel x86-64架构的处理器作为基本的硬件环境[注2]。

在x86-64架构的处理器中，KVM必须的硬件虚拟化扩展分别为：英特尔的虚拟化技术（Intel VT）和AMD的AMD-V技术。其中，英特尔在2005年11月发布的奔腾四处理器（型号：662 和672）第一次正式支持VT技术，而在2006年5月AMD也发布了支持AMD-V的处理器。现在比较流行的针对服务器和桌面的Intel处理器多数都是支持VT技术的，下面主要解释英特尔的VT（Virtualization Technology）技术相关的硬件设置。

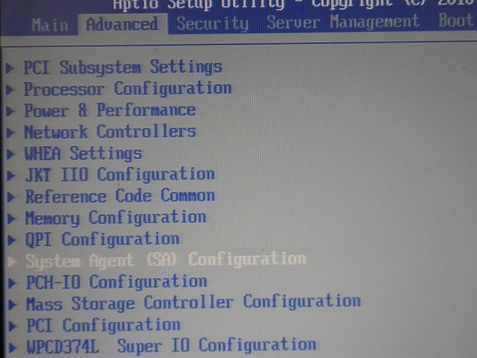
首先处理器（CPU）在硬件上支持了VT技术，还需要在BIOS中将其功能打开，KVM才能使用到。目前，多数流行的服务器和部分桌面处理器的BIOS都是默认将VT打开了的。

VT的选项，一般在BIOS中的“Advanced à Processor Configuration”这里来查看和设置，它的标识一般为“Intel(R) Virtualization Technology”或者“Intel VT”等类似的文字说明。

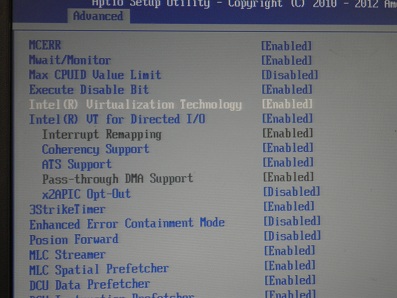
除了必须的处理器虚拟化扩展支持以外，如果服务器芯片还支持VT-d，也建议在BIOS中将其打开，因为后面的一些相对高级的设备直接分配功能会需要硬件VT-d技术的支持。VT-d（Virtualization Technology for Directed I/O）是对设备I/O的虚拟化硬件支持，在BIOS的位置可能为“Advanced à Processor Configuration”或者“Advanced à System Agent (SA) Configuration”，它一般在BIOS中的标志一般为“Intel(R) VT for Directed I/O”或者“Intel VT-d”等。

下面以一台Intel Romley-EP平台的服务器为例来说明BIOS中的设置。

1. BIOS中的Advanced选项：



2. BIOS中的VT和VT-d选项，如图3-2所示：



对于不同平台或不同厂商的BIOS，VT和VT-d等设置位置可能是不一样的，需要根据实际的硬件情况和BIOS中选项来灵活设置。

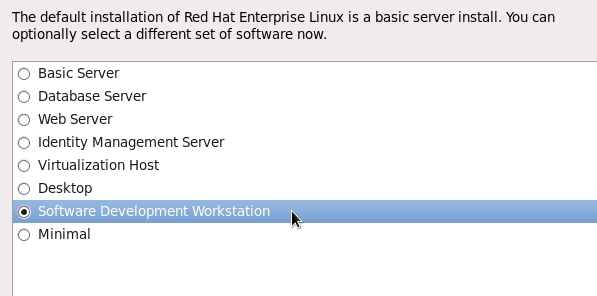
设置好了VT和VT-d的相关选项，保存BIOS的设置并退出，系统重启后会生效。

KVM是基于内核的虚拟化技术，要运行KVM虚拟化环境，安装一个Linux操作系统的宿主机（Host）是必需的。由于Redhat公司是目前对KVM项目投入最多的企业之一，从RHEL6开始，其系统自带的虚拟化方案就采用了KVM，而且RHEL（Red Hat Enterprise Linux）也是最流行的企业级Linux发行版之一，所以本节选用RHEL来讲解Linux系统的安装步骤和过程，并且本章后面的编译和运行都是在这个系统之上的。

当然，KVM作为流行的开源虚拟机之一，它可以在绝大多数流行的Linux系统上编译和运行，所以你依然可以选择RHEL之外的其他Linux发行版，比如CentOS、Fedora、Ubuntu、Debian、OpenSuse等系统都是不错的选择。

本节内容基于目前最新的RHEL版本——RHEL6.3 Server版的系统来简单介绍，普通Linux安装的基本过程这里就不再详细描述，本节主要说明一下安装过程中一些值得注意的地方。

在选择哪种类型的服务器时，选择“Software Development Workstation”即可（如图3-3所示），然后选中当前页面的“Customize now”，点击“Next”按钮进入下一步去选择具体需要安装的组件和设置所需要的各个RPM包。

[](http://smilejay.com/wp-content/uploads/2012/06/3-3.jpg)

RHEL6.3安装中选择服务器类型

在选择了“Software Development Workstation”之后，在具体组件的选择界面，就可以看到已经默认选择好了很多的组件（如图3-4所示），这里主要需要检查一下Development这个选项这里已默认勾选了很多的开发组件。其中，Development选项中的Development tools和Additional Development这两个组件最好被选中，因为在本书中的编译以及其他实验中可能会用到，其中包括一些比较重要的软件包，比如：gcc、git、make等（它们一般被默认选中）。另外可以点击下方的“Optional packages”按钮根据需要选择一些可选的软件包。图3-4中的Virtualization选项中，我们可以先不选中其中的任何组件，因为本章会自己编译KVM和qemu-kvm，而在后面第7章讲发行版中的KVM时，一般会安装Virtualization组件并使用发行版中自带的KVM Virtualization功能。



Software Development Workstation 的具体组件的选择

然后，继续进行后面安装流程，可以安装相应的软件包，安装过程的一个快照如图3-5所示。



RHEL6.3安装过程快照

在安装完所有软件包后，系统会提示安装完成需要重启，重启系统之后，即可进入到RHEL6.3系统中。至此，Linux系统就安装完毕了，这就是在本书中作为宿主机（Host）的操作系统，后面的编译和做实验都是在这个宿主机上进行的（当然，我们会使用本章中讲述的自己编译的kernel和qemu-kvm来做实验）。

### KVM源代码

KVM作为Linux kernel中的一个module而存在，是从Linux 2.6.20版本开始被完全正式加入到内核的主干开发和正式发布代码中。所以，只需要下载2.6.20版本之后Linux kernel的代码即可。当然，如果是学习KVM，推荐使用最新正式发布或者开发中的kernel版本，如果是实际部署到生产环境中，还需要自己选择适合的稳定版本进行足够的功能和性能测试。如果你想使用最新的处于开发中KVM代码，你需要自己下载KVM的代码仓库，本节中就是这样来讲解的。

总的来说，下载最新KVM源代码，主要有如下三种方式：

* 下载KVM项目开发中的代码仓库kvm.git
* 下载Linux内核的代码仓库linux.git
* 打包下载Linux内核的源代码（Tarball[注3]格式）

1.  下载kvm.git

KVM项目的代码是托管在Linux内核官方源码网站[http://git.kernel.org](http://git.kernel.org/)上的，可以到上面去查看和下载。该网页上virt/kvm/kvm.git 即是KVM项目的代码，它是最新的功能最丰富的KVM源代码库（尽管并非最稳定的）。目前，kvm.git的最主要维护者（maintainer）是来自Redhat公司的Avi Kivity和Marcelo Tosatti。从<http://git.kernel.org/?p=virt/kvm/kvm.git>网页可以看到，kvm.git下载链接有如下3个URL可用于下载最新的kvm的开发代码仓库。

git://git.kernel.org/pub/scm/virt/kvm/kvm.git

http://git.kernel.org/pub/scm/virt/kvm/kvm.git

https://git.kernel.org/pub/scm/virt/kvm/kvm.git

这三个URL下载的内容都是完全一致的，根据自己实际情况选择一个下载即可。Linux内核相关的项目一般都使用Git[注4]作为源代码管理工具，KVM当然也是用Git管理源码的。可以使用git clone命令来下载kvm的源代码，也可以使用Git工具的其他命令对源码进行各种管理，这里不详述Git的各种命令，有兴趣的读者可以参考后面的附录“git 常用命令”。

kvm.git的下载方式和过程，为如下命令行所示：

[root@jay-linux ~] cd kvm\_demo

[root@jay-linux kvm\_demo]# git clone git://git.kernel.org/pub/scm/virt/kvm/kvm.git kvm.git

[root@jay-linux kvm\_demo]# cd kvm.git

[root@jay-linux kvm.git]# pwd

2. 下载linux.git

Linux内核的官方网站为[http://kernel.org](http://kernel.org/)，其中源代码管理网为[http://git.kernel.org](http://git.kernel.org/)，可以在那里找到最新的linux.git代码。在源码管理网站上，我们看到有多个linux.git，我们选择Linus Torvalds[注5]的源码库（也即是Linux内核的主干）。在源码的网页<http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git> 中可以看到，其源码仓库可以也有如下3个链接可用：

git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

http://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

这三个URL中源码内容是完全相同的，可以用用git clone命令复制到本地，其具体操作方式与前一种（kvm.git）的下载方式完全一样。

3. 下载Linux的Tarball

在Linux官方网站（<http://kernel.org/>）上，也提供Linux内核的Tarball文件下载；除了在其首页上点击一些Tarball之外，也可以到如下网址下载Linux内核的各个版本的Tarball。

<ftp://ftp.kernel.org/pub/linux/kernel/>

<http://www.kernel.org/pub/linux/kernel/>

kernel.org还提供一种rsync的方式下载，此处不详细叙述，请参见其官网首页的提示。

以用wget下载linux-3.4.1.tar.gz 为例，有如下的命令行代码：

[root@jay-linuxkvm\_demo]# wget http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.4.1.tar.gz

[root@jay-linux kvm\_demo]# ls -l

[root@jay-linux kvm\_demo]# tar -zxf linux-3.4.1.tar.gz

4. 通过kernel.org的镜像站点下载：

由于Linux的源代码量比较大，如果只有美国一个站点可供下载，那么可能会速度较慢，服务器压力也较大。所以，kernel.org在世界上多个国家和地区都有一些镜像站点，而且，一些Linux开源社区的爱好者们也自发建立了不少kernel.org的一些镜像，在中国的镜像站点中，如下两个镜像站点推荐给大家下载Linux相关的代码及其他源码（本书写作之时，它们并没有提供git形式的代码仓库kvm.git或linux.git的下载），访问速度比较快。

其中一个是清华大学开源镜像站：<http://mirror.tuna.tsinghua.edu.cn/>；其中的如下链接地址与<http://www.kernel.org/pub/linux/kernel/>就是同步的，用起来比较方便。

<http://mirror.tuna.tsinghua.edu.cn/kernel/linux/kernel/>

另外一个推荐镜像是北京交通大学的一个开源镜像站，链接如下：

<http://mirror.bjtu.edu.cn/kernel/linux/kernel/>

另外，还有如下两个镜像站可以推荐给大家参考：

网易开源镜像站：<http://mirrors.163.com/>

搜狐开源镜像站：<http://mirrors.sohu.com/>

上面三种方式下载的源代码都是可以同样地进行配置和编译，本章中以开发中的最新源代码仓库kvm.git来讲解KVM的配置和编译等。KVM是作为Linux内核中的一个module存在的，而kvm.git是一个包含了最新的KVM模块开发中代码的完整的Linux内核源码仓库。它的配置方式，与普通的Linux内核配置完全一样，只是需要注意将KVM相关的配置选择为编译进内核或者编译为模块。

在kvm.git（Linux kernel）代码目录下，运行“make help”命令可以得到一些关于如何配置和编译kernel的帮助手册，如下面命令行所示：

[root@jay-linux kvm.git]# make help

Cleaning targets:

clean           – Remove most generated files but keep the config and

enough build support to build external modules

mrproper        – Remove all generated files + config + various backup files

distclean       – mrproper + remove editor backup and patch files

Configuration targets:

config          – Update current config utilising a line-oriented program

nconfig         – Update current config utilising a ncurses menu based program

menuconfig      – Update current config utilising a menu based program

xconfig         – Update current config utilising a QT based front-end

<!- 此处省略数十行帮助信息 ->

make W=n   [targets] Enable extra gcc checks, n=1,2,3 where

1: warnings which may be relevant and do not occur too often

2: warnings which occur quite often but may still be relevant

3: more obscure warnings, can most likely be ignored

Multiple levels can be combined with W=12 or W=123

Execute “make” or “make all” to build all targets marked with [\*]

For further info see the ./README file

对KVM或者Linux内核配置时常用的配置命令，对其如下的一些解释：

q   **make config**

基于文本的最为传统的也是最为枯燥的一种配置方式，但是它可以适用于任何情况之下，这种方式会为每一个内核支持的特性向用户提问，如果用户回答“y”，则把特性编译进内核；回答“m”，则它特性作为模块进行编译；回答“n”，则表示不对该特性提供支持。输入“？”则显示该选项的帮助信息，在了解之后再做决定处理该选项的方式；在回答每个问题前，必须考虑清楚，如果在配置过程中因为失误而给了错误的回答，就只能按“ctcl+c”强行退出然后重新配置了。

q   **make oldconfig**

make oldconfig和make config类似，但是它的作用是在现有的内核设置文件基础上建立一个新的设置文件，只会向用户提供有关新内核特性的问题，在新内核升级的过程中，make oldconfig非常有用，用户将现有的配置文件.config复制到新内核的源码中，执行make oldconfig，此时，用户只需要回答那些针对新增特性的问题。

q   **make silentoldconfig**

和上面make oldconfig一样，但在屏幕上不再出现已在.config中配置好的选项。

q   **make menuconfig**

基于终端的一种配置方式，提供了文本模式的图形用户界面，用户可以通过光标移动来浏览所支持的各种特性。使用这用配置方式时，系统中必须安装有ncurses库，否则会显示“Unable to find the ncurses libraries”的错误提示。其中“Y”、“N”、“M”、“？”键的选择功能与前面make config中介绍的一致。

q   **make xconfig**

基于X Window的一种配置方式，提供了漂亮的配置窗口，不过只有能够在X Server上使运行X桌面应用程序时才能够使用，它依赖于QT，如果系统中没有安装QT库，则会出现“Unable to find any QT installation”的错误提示。

q   **make gconfig**

与make xconfig类似，不同的是make gconfig依赖于GTK库。

q   **make defconfig**

按照内核代码中提供的默认配置文件对内核进行配置（在Intel x86\_64平台上，默认配置为arch/x86/configs/x86\_64\_defconfig），生成.config文件可以用作初始化配置，然后再使用make menuconfig进行定制化配置。

q   **make allyesconfig**

尽可能多地使用“y”设置内核选项值，生成的配置中包含了全部的内核特性。

q   **make allnoconfig**

除必须的选项外，其它选项一律不选 (常用于嵌入式Linux系统的编译)。

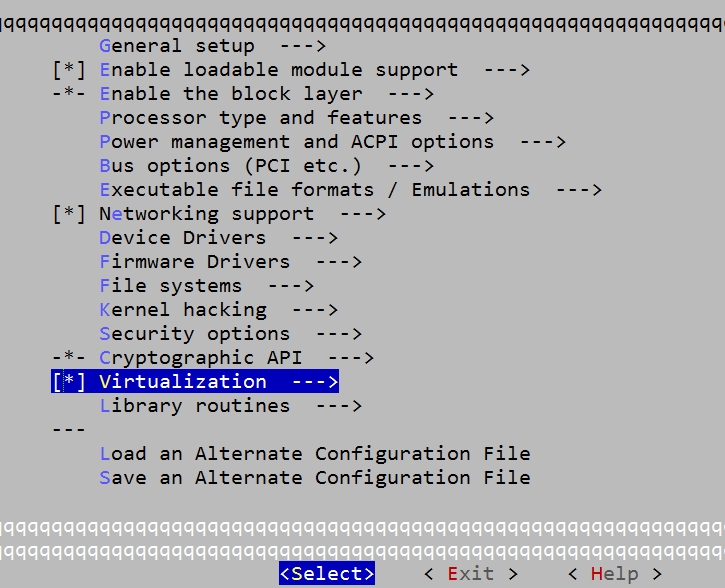
q   **make allmodconfig**

尽可能多的使用“m”设置内核选项值来生成配置文件。

q   **make localmodconfig**

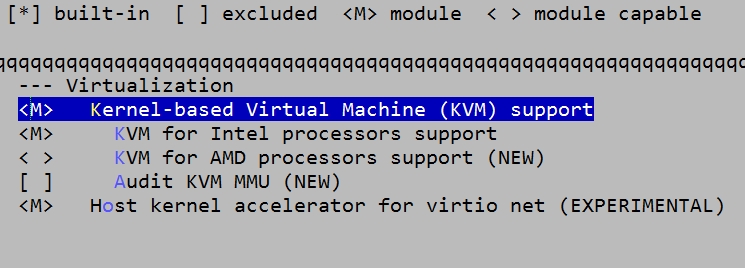
会执行 lsmod 命令查看当前系统中加载了哪些模块 (Modules)，并最后将原来的 .config 中不需要的模块去掉，仅保留前面 lsmod 命令查出来的这些模块，从而简化了内核的配置过程。 这样做确实方便了很多，但是也有个缺点：该方法仅能使编译出的内核支持当前内核已经加载的模块。因为该方法使用的是 lsmod 查询得到的结果，如果有的模块当前没有加载，那么就不会编到新的内核中。

下面以make menuconfig为例讲述一下如何选择KVM相关的配置；运行make menuconfig后显示的界面如图3-6所示：



make menuconfig 命令的选择界面

选择了Virtualization之后，进入里面进行详细的配置，包括选中KVM，选中对处理器的支持（比如：KVM for Intel processors support，KVM for AMD processors support），如图3-7所示：



Virtualization中的配置选项

在配置完成之后，就会在kvm.git的目录下面生成一个.config文件，最好检查一下KVM相关的配置是否正确，与KVM直接相关的配置项主要有如下几个，本次配置的情况如下：

CONFIG\_HAVE\_KVM=y

CONFIG\_HAVE\_KVM\_IRQCHIP=y

CONFIG\_HAVE\_KVM\_EVENTFD=y

CONFIG\_KVM\_APIC\_ARCHITECTURE=y

CONFIG\_KVM\_MMIO=y

CONFIG\_KVM\_ASYNC\_PF=y

CONFIG\_HAVE\_KVM\_MSI=y

CONFIG\_VIRTUALIZATION=y

CONFIG\_KVM=m

CONFIG\_KVM\_INTEL=m

# CONFIG\_KVM\_AMD is not set

CONFIG\_KVM\_MMU\_AUDIT=y

### Qemu-kvm

## libvirt

libvirt 提供一种虚拟机监控程序不可知的 API 来安全管理运行于主机上的来宾操作系统。libvirt 本身不是一种工具， 它是一种可以建立工具来管理来宾操作系统的 API。libvirt 本身构建于一种抽象的概念之上。它为受支持的虚拟机监控程序实现的常用功能提供通用的 API。libvirt 起初是专门为 Xen 设计的一种管理 API，后来被扩展为可支持多个虚拟机监控程序。

### kvm与qemu的关系

qemu：it is a emulator, 用来仿真cpu，设备，总线等设备

kvm： it's a kenerl module, 用来处理qemu向内核发起的请求

qemu是kvm的客户端展现，kvm是内核为qemu提供的服务代理，用来处理qemu的请求

### kvm 存储栈分析

## VirtualBox

## Hyper-V

## Cgroup

# 数据分析

## WEB爬虫

## Heritrix

<http://crawler.archive.org/index.html>

## 搜索引擎

## teradata

## sybase

## greenplum

## hana

# 云存储

## SAN存储

## NAS存储

iSCSI initiator和target，对CDP、snapshot、云备份

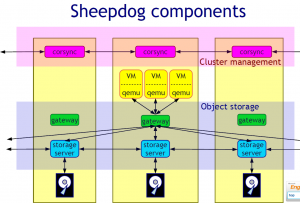
## Sheepdog

sheepdog原理介绍

1. sheepdog是作为虚拟机kvm的volume使用的，是非普通的文件系统，这点和Amazon的EBS(Elastic Block Store)比较类似

2. sheepdog是一种对称(symmetric)的结构，各个节点的地位相同，没有中心节点，没有meta-server，使用[Corosync](http://www.corosync.org/) 对物理节点进行管理

3. sheepdog中的对象存储分为两类，其一是One reader One Writer 其二是No writer multiple reader 而且对象是4M大小分片的，使用“一致性哈希”算法来确定对象存储位置，多副本存储

[](http://www.yaronspace.cn/blog/wp-content/uploads/2011/03/sheepdog1.png)

sheepdog的代码结构

./collie/treeview.c

./collie/collie.c

./sheep/vdi.c

./sheep/store.c

./sheep/sdnet.c

./sheep/work.c

./sheep/sheep.c

./sheep/group.c

./lib/logger.c

./lib/event.c

./lib/net.c

sheep目录下是有关sheepdog的大部分逻辑的处理，部署在各个节点上  
collie目录是作为管理管理sheep的代码  
lib目录下是关于网络、日志和事件等处理模型

sheepdog(牧羊犬) 官方网站：<http://www.osrg.net/sheepdog/>

Sheepdog is a distributed storage system for [KVM](http://www.yaronspace.cn/blog/?s=KVM). It provides highly available block level storage volumes that can be attached to KVM virtual machines. Sheepdog scales to several hundreds nodes, and supports advanced volume management features such as snapshot, cloning, and thin provisioning.

安装步骤

1，环境准备

[corosync-1.3.0.tar.gz](ftp://ftp:downloads@corosync.org/downloads/corosync-1.3.0/corosync-1.3.0.tar.gz) [sheepdog-0.2.0.tar.gz](http://sourceforge.net/projects/sheepdog/files/sheepdog/0.2.0/sheepdog-0.2.0.tar.gz/download)

2，安装corosync

corosync 是linux 集群管理的引擎，具体请参看[官网](http://www.corosync.org/doku.php?id=welcome)

$ tar -xzvf corosync-1.3.0.tar.gz

$ cd corosync-1.3.0

$ ./autogen.sh

$ ./configure

$ sudo make install

3，qemu 0.13的安装 （略）

4，sheepdog安装

$ tar -xzvf sheepdog-0.2.0.tar.gz

$ cd sheepdog-0.2.0

$ ./autogen.sh

$ ./configure

$ sudo make install

这时可能会出现linux/signalfd.h 这个文件找不到，以及后续会出现signalfd undefined reference 的错误，我的系统是CentOS ，出现了这种错误，出现这种情况的话，我简单的解决方法是对signalfd进行模拟，这点我参考了qemu的有关实现，  
具体是在sheep/work.c中加入以下代码：  
代码太长，我贴到最后了，请参看附件

5，sheepdog的具体使用

1> 配置/etc/corosync/corosync.conf  
mv /etc/corosync/corosync.conf.example /etc/corosync/corosync.conf  
同时修改，bindnetaddr字段为自己的ip地址，mcastaddr（怎么修改暂时不清楚，应该是广播的地址，保持默认值即可）  
2> 启动corosync  
# corosync {注：需要是root账户}  
3> 如果是ext3文件系统，需要加入user\_xattr

mount -o remount,user\_xattr /

4> 启动Sheep

# sheep ~/store\_dir <span style="color: #ff6600;">#不能是相对路径，相对路径会出错</span>

# collie cluster format --copies=3

5> 查看状态

# collie node list

# collie cluster info

# colli vdi list

6> 创建镜像

# qemu-img create sheepdog:Alice 256G

# qemu-img convert ~/amd64.raw sheepdog:Bob

不过我在Convert时一直提示不成功，error while writing  
查找原因：通过查看sheep.log日志文件，可以发现时由于Too Many open files 错误提示，原来是打开太多文件所造成的，因为sheepdog对文件进行4M进行分片的，这样会导致大量的文件打开，而一般的系统进程最多的文件数是1024 所以就这个错误！  
解决方法：ulimit -n 4096 {或者更大的值} 这样修改只是在目前会话中有效，需要长久生效，请参看[这篇文章](http://hi.baidu.com/ajin111/blog/item/7754e1620a8373d3e6113a3f.html)

总结

sheepdog作为一个新的项目，感觉还是挺不错的，刚好弥补了开源虚拟化中有关镜像存储的问题，为后续的虚拟机迁移做准备，目前sheepdog暂时还未支持migrate ，不过看它的TODO LIST 下一步已经开始做了，系统能够尽快做出来，下一步主要是看下它的源码，因为很多地方会出现诡异的错误的，还需要解决呀！

## KFS

## CEPH

<http://ceph.com/>

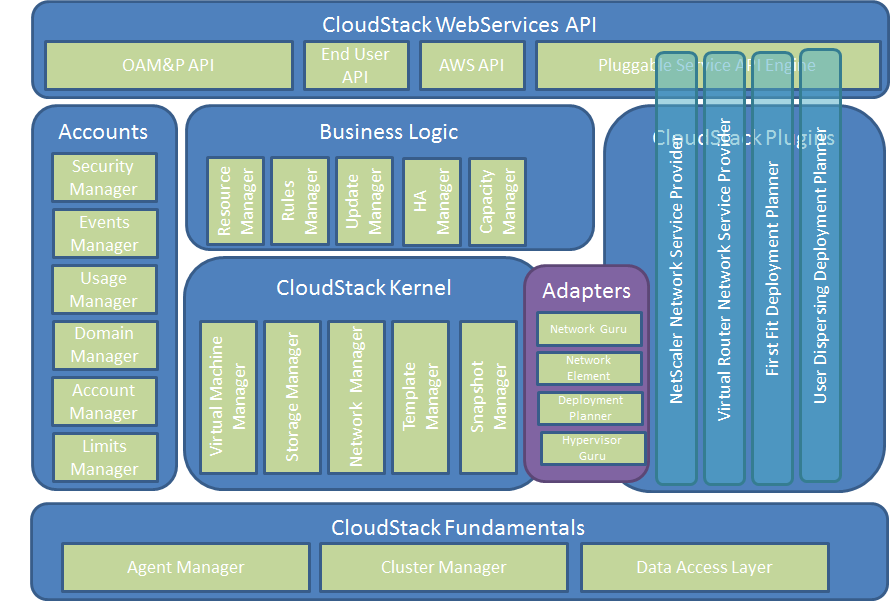
# 云平台

## cloudfoudry

## opennebula

## cloudstack

CloudStack是一个开源云计算解决方案。可以加速高伸缩性的公共和私有云（IaaS）的部署、管理、配置。使用CloudStack作为基础，数据中心操作者可以快速方便的通过现存基础架构创建云服务。

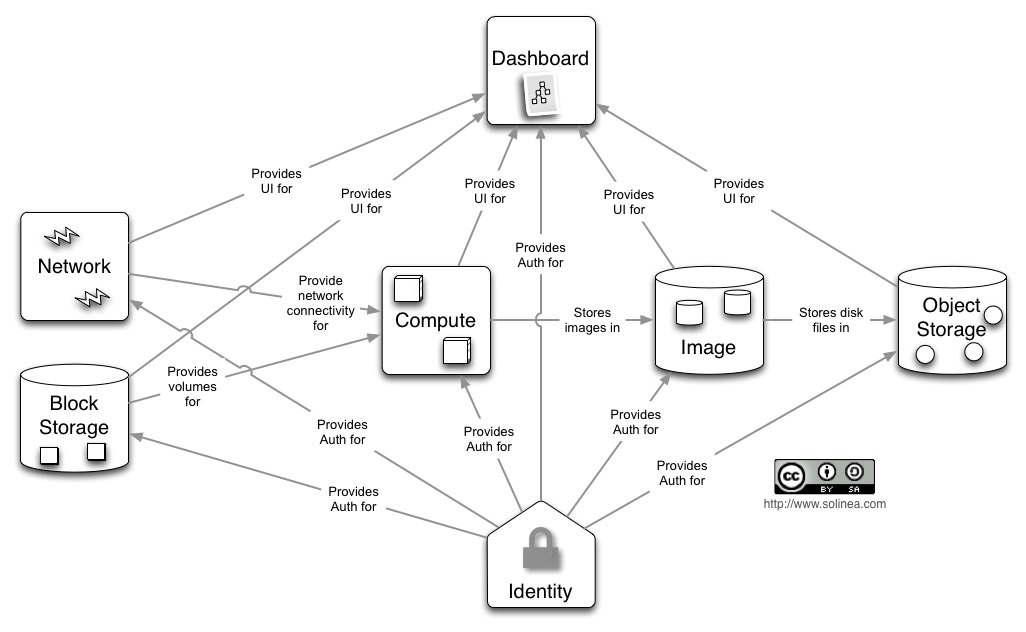


## openstack

供OpenStack 是一个旨在为公共及私有云的建设与管理提软件的开源项目。它的社区拥有超过130家企业及1350位开发者，这些机构与个人都将OpenStack作为基础设施即服务（简称IaaS）资源的通用前端。OpenStack项目的首要任务是简化云的部署过程并为其带来良好的可扩展性。本文希望通过提供必要的指导信息，帮助大家利用OpenStack前端来设置及管理自己的公共云或私有云。

OpenStack是IaaS(基础设施即服务)组件，让任何人都可以自行建立和提供云端运算服务。

### Conceptual Diagram（概念图）



### http://docs.openstack.org/training-guides/content/figures/5/figures/image31.jpgLogical Diagram（逻辑图）

### Openstack Swift

## Docker

Docker的主要优点

* 轻量级资源使用：容器在进程级别隔离并使用宿主机的内核，而不需要虚拟化整个操作系统。
* 可移植性：一个容器应用所需要的依赖都在容器中，这就让它可以在任意一台Docker主机上运行。
* 可预测性：宿主机不需要关心容器内运行的是什么，同样，容器也不需要关心是在哪个宿主机上运行。所需要的接口都是标准化的，并且交互也都是可预测的。

### 常用组件介绍

### 容器化的综述

### 服务发现和分布式配置存储

### 网络与通信

### 集群

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 项目名称 | 中文描述 | 英文描述 | 开源证书 | 开发语言 | Like |
| fleet | 调度器和集群管理工具 | Base coreos ,  A Distributed init System | Apache 2.0 license | Go | / |
| marathon | 调度器和集群管理工具 | Deploy and manage containers (including Docker) on top of Apache Mesos at scale. | Apache 2.0 license | scale | 建议  使用 |
| Swarm | 调度器和集群管理工具 | Swarm: a Docker-native clustering system | Apache 2.0 license | Go | / |
| mesos | 宿主机抽象服务，用于为调度器联合宿主机资源 | Apache Mesos is a cluster manager that provides efficient resource isolation and sharing across distributed applications, or frameworks. | Apache 2.0 license | C++ |  |
| kubernetes | 一个管理容器组的工具，具有先进的调度能力 | Kubernetes is an open source system for managing containerized applications across multiple hosts, providing basic mechanisms for deployment, maintenance, and scaling of applications. | Apache 2.0 license | Go | 太复杂 |
| compose | 一个用于创建容器组的容器编排工具 | Define and run multi-container applications with Docker | Apache 2.0 license | python | / |

Marathon 优点

* HA -- run any number of Marathon schedulers, but only one gets elected as leader; if you access a non-leader, your request gets proxied to the current leader
* Constraints - e.g., only one instance of an application per rack, node, etc.
* Service Discovery & Load Balancing via HAProxy or the events API (see below).
* Health Checks: check your application's health via HTTP or TCP checks.
* Event Subscription lets you supply an HTTP endpoint to receive notifications, for example to integrate with an external load balancer.
* Web UI
* JSON/REST API for easy integration and scriptability
* Basic Auth and SSL
* Metrics: available at /metrics in JSON format

### Flynn

Flynn是一个使用Go语言编写的开源PaaS平台，Flynn使用模块化的设计，任何一个模块都可以独立的进行修改、升级和替换。Flynn的目标是简化分布式环境中应用的部署和维护，通过使用git push命令，Flynn就可以将应用部署到Docker，从而省去了复杂的配置和操作。Flynn的架构大致分为两层，Layer 0是底层的资源层，提供分布式配置、任务调度、服务发现、主机隔离等基础功能；Layer 1基于Layer 0构建了一个用于集群中管理、部署、扩展服务的系统，主要包括管理API/客户端、Git接收器、数据存储、路由。Flynn目前仍在开发中，尚未发布稳定版，但已经获得了很多公司的资助，它被称为是下一代的开源PaaS平台。

### Deis

Deis也是一个支持共有云和私有云的开源PaaS系统，Deis基于Docker和CentOS构建了一个类Heroku的PaaS系统。Deis主要设计用来和不同的云提供商进行交互，目前支持 Rackspace、EC2、 DigitalOcean、Google Compute Engine、Bare-Metal。Deis使用out-of-the-box的方式支持Ruby、Python、Node.js、Java、Clojure、Scala、Play、PHP、Perl、Dart和Go语言，同样支持git push部署。Flynn和Deis都是两个基于Docker的云计算微PaaS技术，关于它们的区别，可以参考这篇文章，作者从架构、实现方式等多方面对二者进行了比较，Deis目前也尚未发布1.0版本，但在GitHub上已经有2000+的star量。

### Dokku

Dokku是一个迷你版的Heroku，基于Docker使用100行左右的Bash代码编写，简单的安装和配置后，即可使用Git命令将应用部署到本地的Dokku平台（当使用git push命令的时候，Dokku会使用buildpack检测应用，然后再部署）。Dokku实际上相当于一个单机版的Heroku，它包含4个组件，分别是Docker、Buildstep、pluginhook、sshcommand。Dokku目前支持Node.js、Ruby、Python。

### CoreOS

CoreOS是一种新的、架构体系重新设计的Linux发行版，可以运行在既有的硬件或者云上。CoreOS不提供类似yum或者apt类似的包管理工具，用户不需要在CoreOS中安装软件，而是让程序都在Docker容器中去运行。CoreOS使用systemd和fleet来对容器进行管理，通过etcd进行服务发现和配置信息共享。CoreOS目前风头正劲，目前已经获得融资并于上周宣布收购私有Docker仓库服务商Quay.io，进军企业级的Registry。另外，CoreOS的etcd等组件也获得了社区的认可，并得到了大规模使用。CoreOS已经发布首个稳定版本，目前主流的云服务商都提供了对CoreOS的支持。

### Fig

Fig是一个基于Docker的用于快速搭建开发环境的工具，目前Fig团队已经加入Docker公司。Fig通过一个配置文件来管理多个Docker容器，非常适合组合使用多个容器进行开发的场景。Fig可以和Docker一起来构建基于Docker的复杂应用，CoreOS的功能强大但是配置比较复杂，而Fig相对而言比较简单，但是很难在多台服务器上做扩展，如何使用Fig构建多个容器的复杂应用并且把这些应用部署到基于CoreOS 的生产环境可以参考这篇文章。从Docker官方收购Fig也可以看到该项目的重要性，Fig目前尚未发布1.0版本。

### Atomic

Atomic是一个基于Fedora的用于运行Docker容器的原型系统，它从Fedora（在将来可能会是从另一个发行版）获取软件包，并扮演了一个“操作系统二进制文件的Git”的角色，允许对不同集合的软件包进行操作安装，并在它们之间进行原子的整体切换。因为Atomic项目与它的上游的Red Hat产品几乎承载着一样的二进制文件和内核，所以它同样会从上游的工程工作和支持中受益，这成为了它发展过程中的一项绝对优势。Atomic项目目前还没有达到用于生产环境的要求，仍在开发中。

### Drone

Drone是一个使用Go语言编写的基于Docker的持续集成系统。Drone可以快速提供隔离的虚拟环境编译测试，而且根据需要保留结果，比使用VM更加简洁有效。如何使用 Drone和Docker搭建全功能的CI服务器可以参考此文。使用Drone搭建CI服务器后，代码可以不离开公司网络即可测试，这非常适合大公司的保密原则，另外，由于Drone基于Docker使用，所以部署到生产环境也非常容易。

### Kubernetes

Kubernetes是来自Google的容器集群管理工具。Kubernetes支持跨平台，它可以在除Google以外的其它云平台中运行，比如AWS。基于Docker之上的Kubernetes可以构建一个容器的调度服务，其目的是让用户通过Kubernetes集群来管理云端容器的集群，而无需用户进行复杂的设置工作。系统会自动选取合适的工作节点来执行具体的容器集群调度处理工作。目前Kubernetes已经得到得到微软、IBM、红帽、CoreOS等公司的支持。

### Panamax

Panamax是一个CenturyLink开源的Docker管理工具，用户可以把多个Docker容器组合为模板并分享到GitHub。Panamax中的应用是由基于Docker镜像的独立服务组合而成，这些Docker镜像来自Docker Hub或者其它的Docker registry。Web的用户界面允许每个服务可以连接到其他服务，并可以配置环境变量、端口绑定、卷。另外也可以添加自定义的Docker运行命令。当这些服务组合在一起成为一个具备完整功能的应用后就可以作为一个模板保存到GitHub。Panamax的最初版本运行在由Vagrant管理的VirtualBox上，由于Vagrant的限制，目前Panamax仅可运行在Mac和Linux的VirtualBox上，并不支持其他虚拟化平台。CenturyLink的云平台也将会支持Panamax。

### Swarm

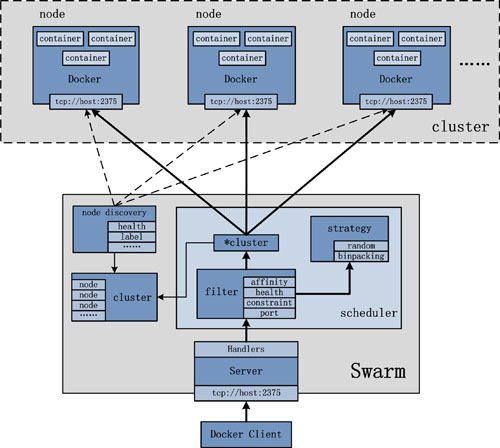
Docker自诞生以来，其容器特性以及镜像特性给DevOps爱好者带来了诸多方便。然而在很长的一段时间内，Docker只能在单host上运行，其跨host的部署、运行与管理能力颇受外界诟病。跨host能力的薄弱，直接导致Docker容器与host的紧耦合，这种情况下，Docker容器的灵活性很难令人满意，容器的迁移、分组等都成为很难实现的功能点。

Swarm是Docker公司在2014年12月初新发布的容器管理工具。和Swarm一起发布的Docker管理工具还有Machine以及Compose。

Swarm是一套较为简单的工具，用以管理Docker集群，使得Docker集群暴露给用户时相当于一个虚拟的整体。Swarm使用标准的Docker API接口作为其前端访问入口，换言之，各种形式的Docker Client(dockerclient in go, docker\_py, docker等)均可以直接与Swarm通信。Swarm几乎全部用Go语言来完成开发，并且还处于Alpha版本，目前在github上发布的版本仅有v0.1.0-rc1。然而Swarm的发展十分快速，功能和特性的变更迭代还非常频繁。因此，可以说Swarm还不推荐被用于生产环境中，但可以肯定的是Swarm是一项很有前途的技术。Swarm的设计和其他Docker项目一样，遵循“batteries included but removable”原则。笔者对该原则的理解是：batteries included代表设计Swarm时，为了完全体现分布式容器集群部署、运行与管理功能的完整性，Swarm和Docker协同工作，以达到效果；“but removable”意味着Swarm与Docker并非紧耦合，这套管理引擎并未侵入Docker的使用，同时这套机制也可以用以其他容器技术的集群部署、运行与管理。

#### Swarm架构

Swarm作为一个管理Docker集群的工具，首先需要将其部署起来，可以单独将Swarm部署于一个节点。另外，自然需要一个Docker集群，集群上每一个节点均安装有Docker。具体的Swarm架构图可以参照下图：



**图5.5-1**

Swarm架构中最主要的处理部分自然是Swarm节点，Swarm管理的对象自然是Docker Cluster，Docker Cluster由多个Docker Node组成，而负责给Swarm发送请求的是Docker Client。

#### Swarm命令

Swarm架构图可以让大家对Swarm有一个初步的认识，比如Swarm的具体工作流程：Docker Client发送请求给Swarm；Swarm处理请求并发送至相应的Docker Node；Docker Node执行相应的操作并返回响应。除此之外，Swarm的工作原理依然还不够明了。

深入理解Swarm的工作原理，可以先从Swarm提供的命令入手。Swarm支持的命令主要有4个：

* swarm create
* swarm manage
* swarm join
* swarm list

##### swarm create

Swarm中swarm create命令用于创建一个集群标志，用于Swarm管理Docker集群时，Docker Node的节点发现功能。发起该命令之后，Swarm会前往Docker Hub上内建的发现服务中获取一个全球唯一的token，用以唯一的标识Swarm管理的Docker集群。

**注：**Swarm的运行需要使用服务发现，目前该服务内建与Docker Hub，该服务发现机制目前还在alpha版本，站点为：http://discovery-stage.hub/docker.com 。

##### swarm manage

Swarm中swarm manage是最为重要的管理命令。一旦swarm manage命令在Swarm节点上被触发，则说明用户需要swarm开始管理Docker集群。从运行流程的角度来讲，swarm经历的阶段主要有两点：启动swarm、接收并处理Docker集群管理请求。

Swarm启动的过程包含三个步骤：

发现Docker集群中的各个节点，收集节点状态、角色信息，并监视节点状态的变化；

初始化内部调度（scheduler）模块；

创建并启动API监听服务模块；

**第一个步骤**，Swarm发现Docker集群中的节点。发现（discovery）是Swarm中用于维护Docker集群状态的机制。既然涉及到发现（discovery），那在这之前必须先有注册（register）。Swarm中有专门负责发现（discovery）的模块，而关于注册（register）部分，不同的discovery模式下，注册（register）也会有不同的形式。

目前，Swarm中提供了5种不同的发现（discovery）机制：Node Discovery、File Discovery、Consul Discovery、EtcD Discovery和Zookeeper Discovery。

**第二个步骤，**Swarm内部的调度（scheduler）模块被初始化。swarm通过发现机制发现所有注册的Docker Node，并收集到所有Docker Node的状态以及具体信息。此后，一旦Swarm接收到具体的Docker管理请求，Swarm需要对请求进行处理，并通过所有Docker Node的状态以及具体信息，来筛选（filter）决策到底哪些Docker Node满足要求，并通过一定的策略（strategy）将请求转发至具体的一个Docker Node。

**第三个步骤，**Swarm创建并初始化API监听服务模块。从功能的角度来讲，可以将该模块抽象为Swarm Server。需要说明的是：虽然Swarm Server完全兼容Docker的API，但是有不少Docker的命令目前是不支持的，毕竟管理Docker集群与管理单独的Docker会有一些区别。当Swarm Server被初始化并完成监听之后，用户即可以通过Docker Client向Swarm发送Docker集群的管理请求。

Swarm的swarm manage接收并处理Docker集群的管理请求，即是Swarm内部多个模块协同合作的结果。请求入口为Swarm Server，处理引擎为Scheduler，节点信息依靠Disocovery。

##### swarm join

Swarm的swarm join命令用于将Docker Node添加至Swarm管理的Docker集群中。从这点也可以看出swarm join命令的执行位于Docker Node，因此在Docker Node上运行该命令，首先需要在Docker Node上安装Swarm，由于该Swarm只会执行swarm join命令，故可以将其当成Docker Node上用于注册的agent模块。

功能而言，swarm join可以认为是完成Docker Node在Swarm节点处的注册（register）工作，以便Swarm在执行swarm manage时可以发现该Docker Node。然而，上文提及的5种discovery模式中，并非每种模式都支持swarm join命令。不支持的discovery的模式有Node Discovery与File Discovery。

Docker Node上swarm join执行之后，标志着Docker Node向Swarm注册，请求加入Swarm管理的Docker集群中。Swarm通过注册信息，发现Docker Node，并获取Docker Node的状态以及具体信息，以便处理Docker请求时作为调度依据。

##### swarm list

Swarm中的swarm list命令用以列举Docker集群中的Docker Node。

Docker Node的信息均来源于Swarm节点上注册的Docker Node。而一个Docker Node在Swarm节点上注册，仅仅是注册了Docker Node的IP地址以及Docker监听的端口号。

使用swarm list命令时，需要指定discovery的类型，类型包括：token、etcd、file、zk以及<ip>。而swarm list并未罗列Docker集群的动态信息，比如Docker Node真实的运行状态，或者Docker Node在Docker集群中扮演的角色信息。

Swarm的架构以及命令并没有很复杂，同时也为希望管理Docker集群的Docker爱好者降低了学习和使用门槛。

俗话说得好，没有一种一劳永逸的工具，有效的管理Docker集群同样也是如此。**缺乏场景来谈论Swarm的价值，意义并不会很大。相反，探索和挖掘Swarm的特点与功能，并为Docker集群的管理提供一种可选的方案，是Docker爱好者更应该参与的事。**

[Swarm: a Docker-native clustering system](http://github.com/docker/swarm)

[Intro to Docker Swarm: Part 1 - Overview](http://technolo-g.com/intro-to-docker-swarm-pt1-overview/)

[Intro to Docker Swarm: Part 2 - Configuration Options and Requirements](http://technolo-g.com/intro-to-docker-swarm-pt2-config-options-requirements/)

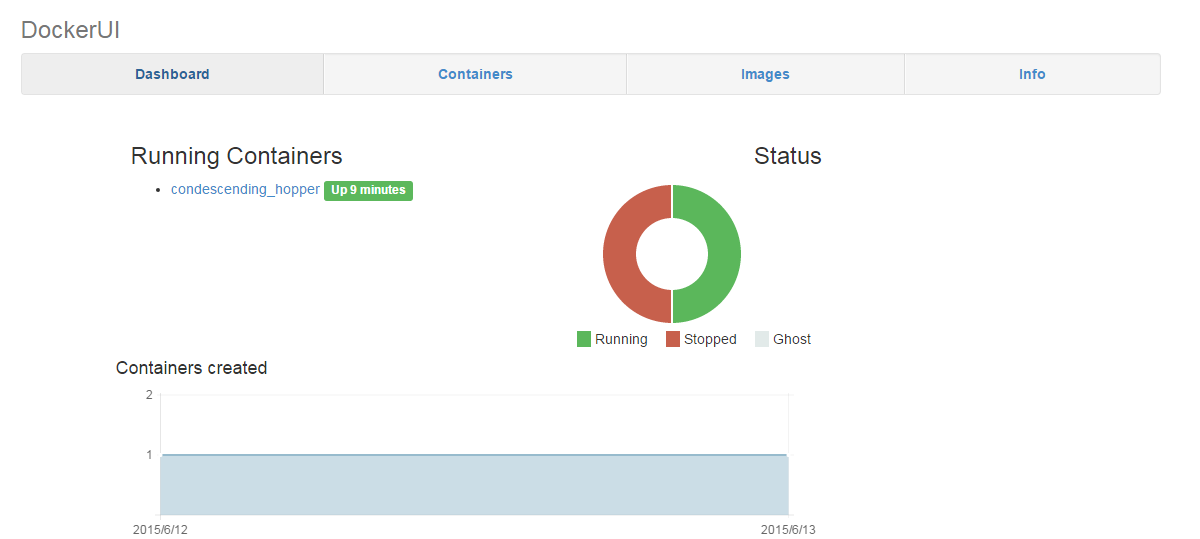
### Boot2Docker

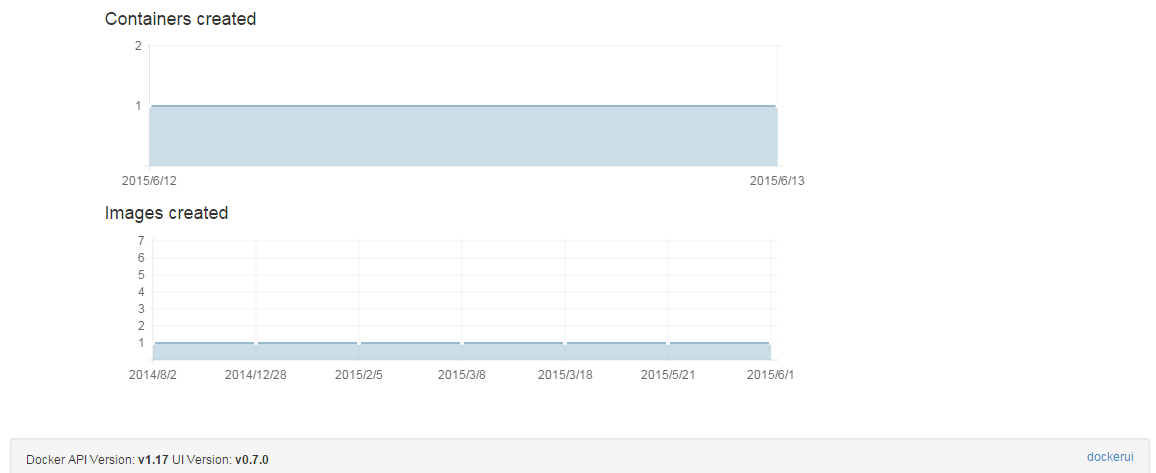
Boot2Docker是一个专为Docker而设计的轻量级Linux发型包，解决Windows或者OS X用户不能安装Docker的问题。Boot2Docker完全运行于内存中，24M大小，启动仅5-6秒。Boot2Docker需要运行在VirtualBox中，具体的安装可以参考官方文档。

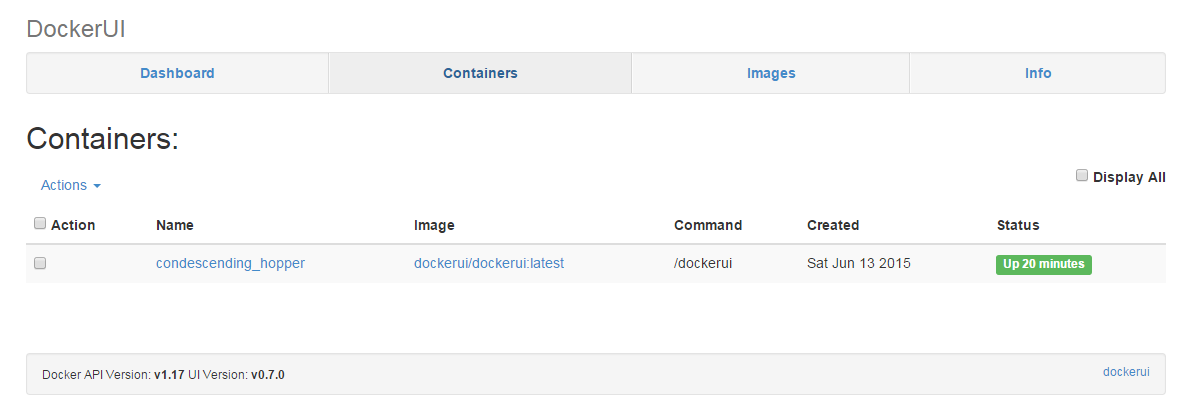
### DockerUI

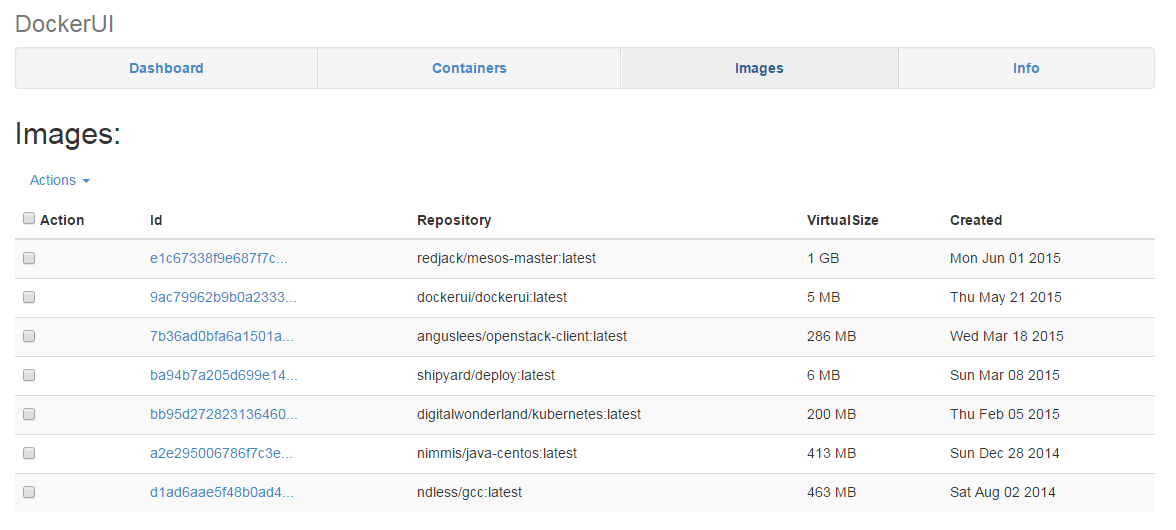
docker pull dockerui/dockerui

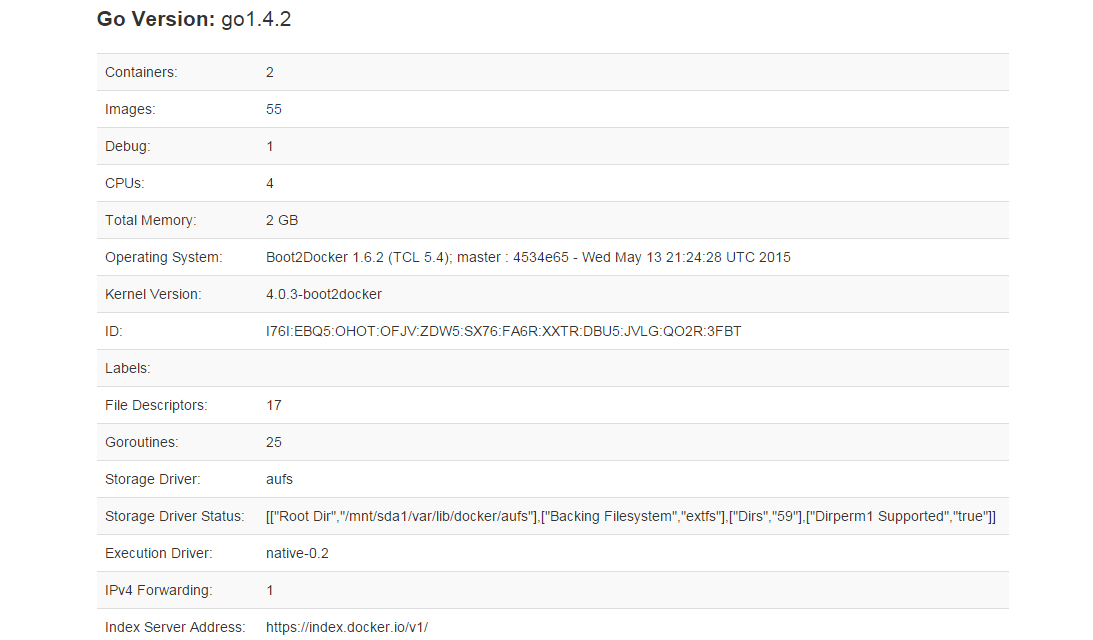
docker run -d -p 9000:9000 --privileged -v /var/run/docker.sock:/var/run/docker.sock dockerui/dockerui











### Cgroups

cgroups 实现了对资源的配额和度量。 cgroups 的使用非常简单，提供类似文件的接口，在 /cgroup目录下新建一个文件夹即可新建一个group，在此文件夹中新建task文件，并将pid写入该文件，即可实现对该进程的资源控制。groups可以限制blkio、cpu、cpuacct、cpuset、devices、freezer、memory、net\_cls、ns九大子系统的资源。

1. blkio 这个子系统设置限制每个块设备的输入输出控制。例如:磁盘，光盘以及usb等。
2. cpu 这个子系统使用调度程序为cgroup任务提供cpu的访问。
3. cpuacct 产生cgroup任务的cpu资源报告。
4. cpuset 如果是多核心的cpu，这个子系统会为cgroup任务分配单独的cpu和内存。
5. devices 允许或拒绝cgroup任务对设备的访问。
6. freezer 暂停和恢复cgroup任务。
7. memory 设置每个cgroup的内存限制以及产生内存资源报告。
8. net\_cls 标记每个网络包以供cgroup方便使用。
9. ns 名称空间子系统。

### LXC

### AUFS

AUFS (AnotherUnionFS) 是一种 Union FS, 简单来说就是支持将不同目录挂载到同一个虚拟文件系统下(unite several directories into a single virtual filesystem)的文件系统, 更进一步的理解, AUFS支持为每一个成员目录(类似Git Branch)设定readonly、readwrite 和 whiteout-able 权限, 同时 AUFS 里有一个类似分层的概念, 对 readonly 权限的 branch 可以逻辑上进行修改(增量地, 不影响 readonly 部分的)。通常 Union FS 有两个用途, 一方面可以实现不借助 LVM、RAID 将多个disk挂到同一个目录下, 另一个更常用的就是将一个 readonly 的 branch 和一个 writeable 的 branch 联合在一起，Live CD正是基于此方法可以允许在 OS image 不变的基础上允许用户在其上进行一些写操作。Docker 在 AUFS 上构建的 container image 也正是如此，接下来我们从启动 container 中的 linux 为例来介绍 docker 对AUFS特性的运用。

典型的启动Linux运行需要两个FS: bootfs + rootfs:典型的启动Linux运行需要两个FS: bootfs + rootfs:



图5-1

bootfs (boot file system) 主要包含 bootloader 和 kernel, bootloader主要是引导加载kernel, 当boot成功后 kernel 被加载到内存中后 bootfs就被umount了. rootfs (root file system) 包含的就是典型 Linux 系统中的 /dev, /proc,/bin, /etc 等标准目录和文件。

对于不同的linux发行版, bootfs基本是一致的, 但rootfs会有差别, 因此不同的发行版可公用bootfs 如下图:

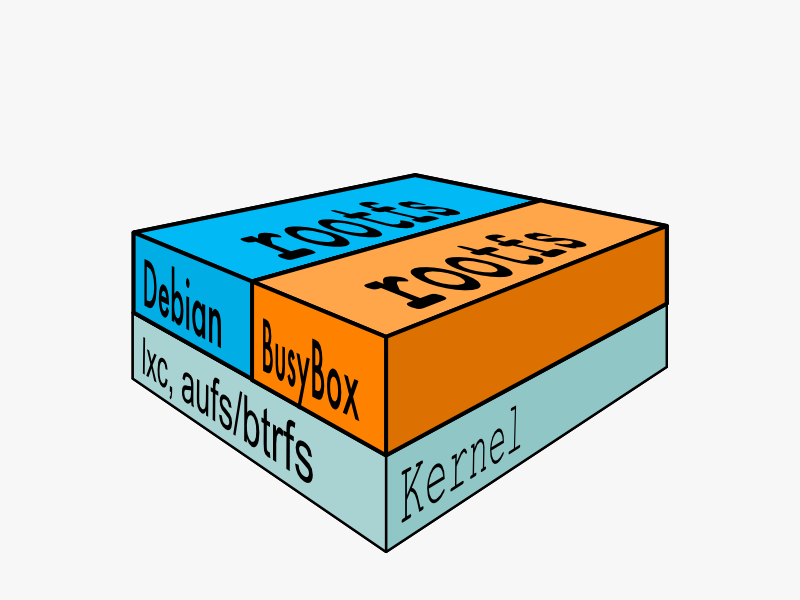


图5-2

典型的Linux在启动后，首先将 rootfs 设置为 readonly, 进行一系列检查, 然后将其切换为 "readwrite" 供用户使用。在Docker中，初始化时也是将 rootfs 以readonly方式加载并检查，然而接下来利用 union mount 的方式将一个 readwrite 文件系统挂载在 readonly 的rootfs之上，并且允许再次将下层的 FS(file system) 设定为readonly 并且向上叠加, 这样一组readonly和一个writeable的结构构成一个container的运行时态, 每一个FS被称作一个FS层。如下图:

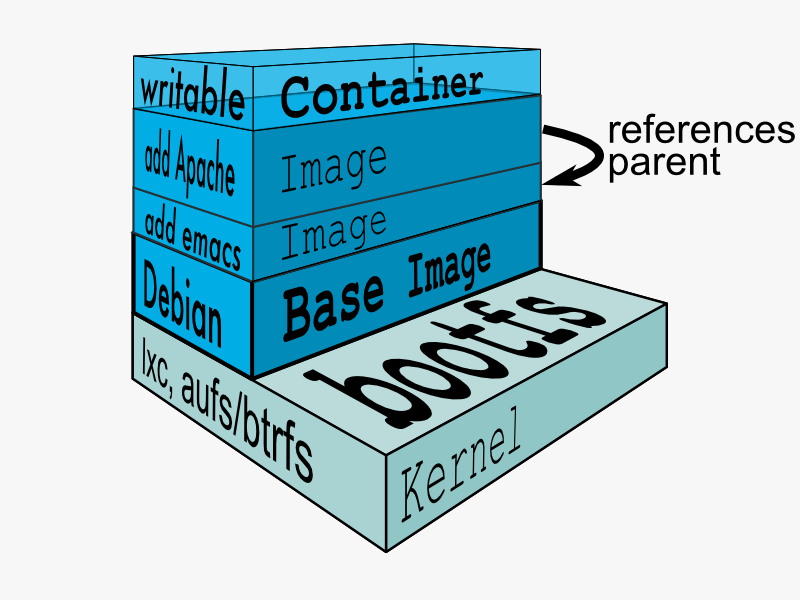


图5-3

得益于AUFS的特性, 每一个对readonly层文件/目录的修改都只会存在于上层的writeable层中。这样由于不存在竞争, 多个container可以共享readonly的FS层。 所以Docker将readonly的FS层称作 "image" - 对于container而言整个rootfs都是read-write的，但事实上所有的修改都写入最上层的writeable层中, image不保存用户状态，只用于模板、新建和复制使用。

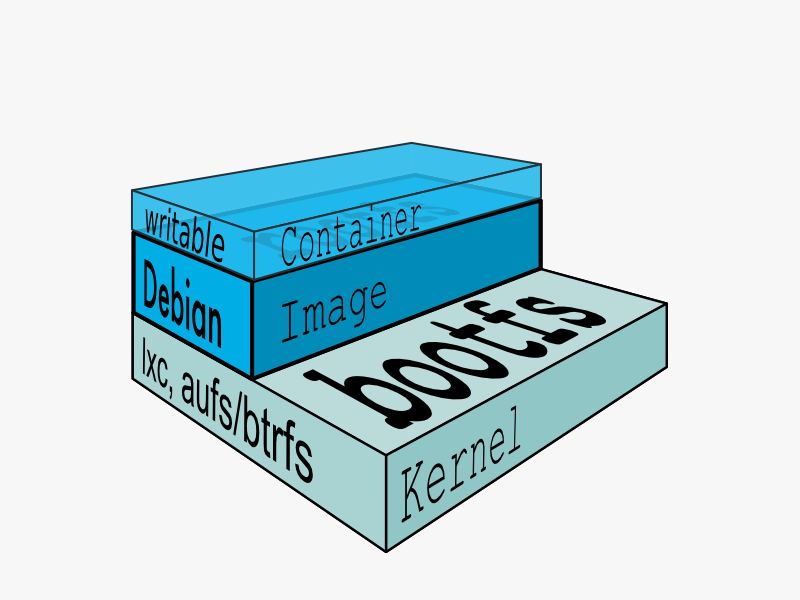


图5-5

上层的image依赖下层的image，因此Docker中把下层的image称作父image，没有父image的image称作base image。因此想要从一个image启动一个container，Docker会先加载这个image和依赖的父images以及base image，用户的进程运行在writeable的layer中。所有parent image中的数据信息以及 ID、网络和lxc管理的资源限制等具体container的配置，构成一个Docker概念上的container。如下图:

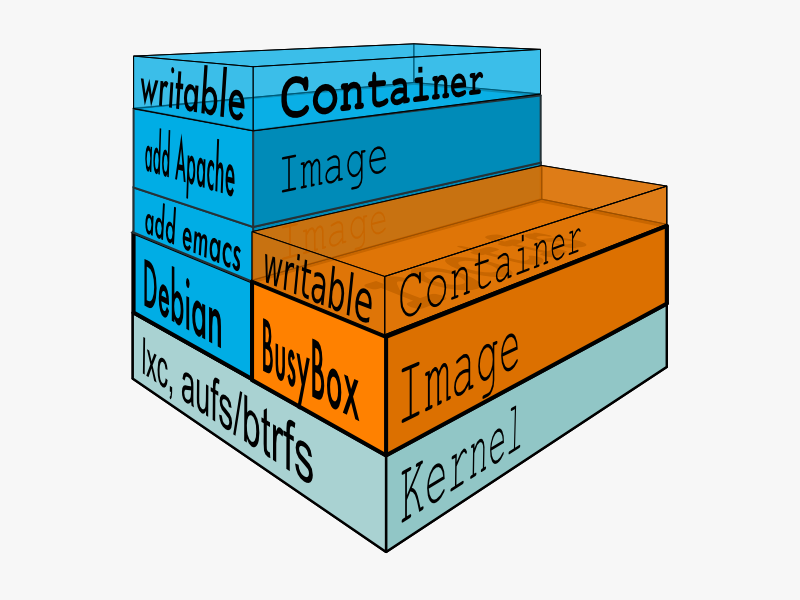


图5-6

安全永远是相对的，这里有三个方面可以考虑Docker的安全特性:

1. 由kernel namespaces和cgroups实现的Linux系统固有的安全标准;
2. Docker Deamon的安全接口;
3. Linux本身的安全加固解决方案,类如AppArmor, SELinux;

由于安全属于非常具体的技术，这里不在赘述，请直接参阅[Docker官方文档](http://blog.dataman-inc.com/untitled-2/docs.docker.com/articles/security/)。

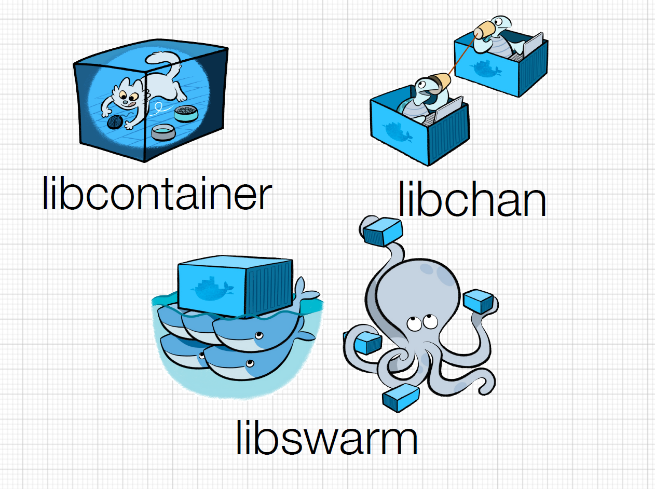


图5-7

#### Libswarm

Libswarm，是Solomon Hykes (Docker的CTO) 在DockerCon 2014峰会上向社区介绍的新“乐高积木”工具: 它是用来统一分布式系统的网络接口的API。Libswarm要解决的问题是，基于Docker构建的分布式应用已经催生了多个基于Docker的服务发现(Serivce Discovery)项目，例如etcd, fleet, geard, mesos, shipyard, serf等等，每一套解决方案都有自己的通讯协议和使用方法，使用其中的任意一款都会局限在某一个特定的技术范围內。所以Docker的CTO就想用libswarm暴露出通用的API接口给分布式系统使用，打破既定的协议限制。目前项目还在早期发展阶段，值得参与。

#### Libchan

Libchan，是一个底层的网络库，为上层 Libswarm 提供支持。相当于给Docker加上了ZeroMQ或RabbitMQ，这里自己实现网络库的好处是对Docker做了特别优化，更加轻量级。一般开发者不会直接用到它，大家更多的还是使用Libswarm来和容器交互。喜欢底层实现的网络工程师可能对此感兴趣，不妨一看。

#### Libcontainer

Libcontainer，Docker技术的核心部分，单独列出来也是因为这一块的功能相对独立，功能代码的迭代升级非常快。想了解Docker最新的支持特性应该多关注这个模块。

## Mesos

### Marathon

### Bamboo

### Chronos

### AWS

<http://aws.amazon.com/cn/>

## Aliyun

<http://www.aliyun.com/>

## 云平台架构

#### overview

云推送系统，云消息系统，底层协议

高性能服务器软件开发

OSGI、AXIS2

penshift/cloudfoundry/Appscale/stratos

Portal、BI、EAI

计算机网络协议及大并发量、网络服务器端开发

虚拟化、操作系统内核

数据中心资源管理、监控、调度等系统研

1、负责云系统的架构安全设计

2、云系统的均衡负载设计

3、云的优化存储设计

Linux/Unix操作系统，PBS、LSF等排队系统

Fortran、C/C++、Java

Python、Perl、TK/TCL

VASP、Gaussian、CP2K、CPMD、ADF、Amber、CHARMM、Gromacs、Molpro、Vitual Docker、Schrodinger、Sybyl、Materials Studio、Discover Studio

快速部署、资源调度、多租户技术、海量数据处理技术、大规模消息通信技术、大规模分布式存储技术、许可证管理及收费；

算法、数据架构和存储技术（SAN、scale-out NAS、iscsi、FC）

网络交换、路由协议开发或访问控制、AAA开发’

多租户技术

## 混合云

#### 混合云问题定义

# 资源监控

## zabbix

<https://www.zabbix.org/wiki/Main_Page>

## nagios

<http://sourceforge.net/projects/nagios/?source=dlp>

<http://www.ibm.com/developerworks/cn/linux/l-ganglia-nagios-1/index.html>

[Linux下Nagios的安装与配置](http://www.cnblogs.com/mchina/archive/2013/02/20/2883404.html)

一、Nagios简介

[Nagios logo](http://images.cnitblog.com/blog/370046/201301/30172010-1a4517cdb03747cc8d37837d6f550d8a.png)

　　Nagios是一款开源的电脑系统和网络监视工具，能有效监控Windows、Linux和Unix的主机状态，交换机路由器等网络设置，打印机等。在系统或服务状态异常时发出邮件或短信报警第一时间通知网站运维人员，在状态恢复后发出正常的邮件或短信通知。

　　Nagios原名为NetSaint，由Ethan Galstad开发并维护至今。NAGIOS是一个缩写形式: "Nagios Ain't Gonna Insist On Sainthood" Sainthood 翻译为圣徒，而"Agios"是"saint"的希腊表示方法。Nagios被开发在Linux下使用，但在Unix下也工作得非常好。

主要功能

网络服务监控（SMTP、POP3、HTTP、NNTP、ICMP、SNMP、FTP、SSH）

主机资源监控（CPU load、disk usage、system logs），也包括Windows主机（使用NSClient++ plugin）

可以指定自己编写的Plugin通过网络收集数据来监控任何情况（温度、警告……）

可以通过配置Nagios远程执行插件远程执行脚本

远程监控支持SSH或SSL加通道方式进行监控

简单的plugin设计允许用户很容易的开发自己需要的检查服务，支持很多开发语言（shell scripts、C++、Perl、ruby、Python、PHP、C#等）

包含很多图形化数据Plugins（Nagiosgraph、Nagiosgrapher、PNP4Nagios等）

可并行服务检查

能够定义网络主机的层次，允许逐级检查，就是从父主机开始向下检查

当服务或主机出现问题时发出通告，可通过email, pager, sms 或任意用户自定义的plugin进行通知

能够自定义事件处理机制重新激活出问题的服务或主机

自动日志循环

支持冗余监控

包括Web界面可以查看当前网络状态，通知，问题历史，日志文件等

二、Nagios工作原理

　　Nagios的功能是监控服务和主机，但是他自身并不包括这部分功能，所有的监控、检测功能都是通过各种插件来完成的。

　　启动Nagios后，它会周期性的自动调用插件去检测服务器状态，同时Nagios会维持一个队列，所有插件返回来的状态信息都进入队列，Nagios每次都从队首开始读取信息，并进行处理后，把状态结果通过web显示出来。

　　Nagios提供了许多插件，利用这些插件可以方便的监控很多服务状态。安装完成后，在nagios主目录下的/libexec里放有nagios自带的可以使用的所有插件，如，check\_disk是检查磁盘空间的插件，check\_load是检查CPU负载的，等等。每一个插件可以通过运行./check\_xxx –h 来查看其使用方法和功能。

　　Nagios可以识别4种状态返回信息，即 0(OK)表示状态正常/绿色、1(WARNING)表示出现警告/黄色、2(CRITICAL)表示出现非常严重的错误/红色、3(UNKNOWN)表示未知错误/深黄色。Nagios根据插件返回来的值，来判断监控对象的状态，并通过web显示出来，以供管理员及时发现故障。

四种监控状态

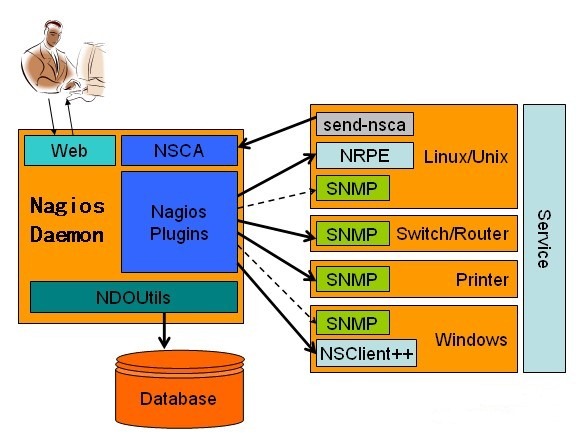
[](http://images.cnitblog.com/blog/370046/201301/30172012-5e48d4a0cbc542b88faf0cdaab257bef.jpg)

　　再说报警功能，如果监控系统发现问题不能报警那就没有意义了，所以报警也是nagios很重要的功能之一。但是，同样的，Nagios 自身也没有报警部分的代码，甚至没有插件，而是交给用户或者其他相关开源项目组去完成的。

　　Nagios 安装，是指基本平台，也就是Nagios软件包的安装。它是监控体系的框架，也是所有监控的基础。

　　打开Nagios官方的文档，会发现Nagios基本上没有什么依赖包，只要求系统是Linux或者其他Nagios支持的系统。不过如果你没有安装apache（http服务），那么你就没有那么直观的界面来查看监控信息了，所以apache姑且算是一个前提条件。关于apache的安装，网上有很多，照着安装就是了。安装之后要检查一下是否可以正常工作。

　　知道Nagios 是如何通过插件来管理服务器对象后，现在开始研究它是如何管理远端服务器对象的。Nagios 系统提供了一个插件NRPE。Nagios 通过周期性的运行它来获得远端服务器的各种状态信息。它们之间的关系如下图所示：

[](http://images.cnitblog.com/blog/370046/201301/30172017-a0cab434c3184ee48163b3c2f86d0299.jpg)

Nagios 通过NRPE 来远端管理服务

1. Nagios 执行安装在它里面的check\_nrpe 插件，并告诉check\_nrpe 去检测哪些服务。

2. 通过SSL，check\_nrpe 连接远端机子上的NRPE daemon

3. NRPE 运行本地的各种插件去检测本地的服务和状态(check\_disk,..etc)

4. 最后，NRPE 把检测的结果传给主机端的check\_nrpe，check\_nrpe 再把结果送到Nagios状态队列中。

5. Nagios 依次读取队列中的信息，再把结果显示出来。

三、实验环境

|  |  |  |  |
| --- | --- | --- | --- |
| Host Name | OS | IP | Software |
| Nagios-Server | CentOS release 6.3 (Final) | 192.168.1.108 | Apache、Php、Nagios、nagios-plugins |
| Nagios-Linux | CentOS release 5.8 (Final) | 192.168.1.111 | nagios-plugins、nrpe |
| Nagios-Windows | Windows XP | 192.168.1.113 | NSClient++ |

Server 安装了nagios软件，对监控的数据做处理，并且提供web界面查看和管理。当然也可以对本机自身的信息进行监控。

Client 安装了NRPE等客户端，根据监控机的请求执行监控，然后将结果回传给监控机。

防火墙已关闭/iptables: Firewall is not running.

SELINUX=disabled

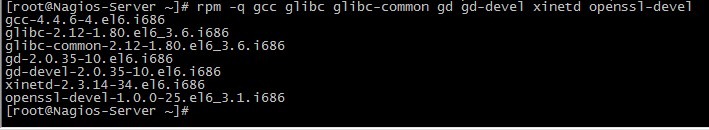
四、实验目标



五、Nagios服务端安装

5.1 基础支持套件：gcc glibc glibc-common gd gd-devel xinetd openssl-devel

# rpm -q gcc glibc glibc-common gd gd-devel xinetd openssl-devel

[](http://images.cnitblog.com/blog/370046/201301/30172023-cbd356b06f1c4fb796b99237b3554b2f.jpg)

如果系统中没有这些套件，使用yum 安装

# yum install -y gcc glibc glibc-common gd gd-devel xinetd openssl-devel

5.2 创建nagios用户和用户组

# useradd -s /sbin/nologin nagios

# mkdir /usr/local/nagios

# chown -R nagios.nagios /usr/local/nagios

[](http://images.cnitblog.com/blog/370046/201301/30172026-dd983461e7e24d3fa12c39d213616afc.jpg)

查看nagios 目录的权限

# ll -d /usr/local/nagios/

[7](http://images.cnitblog.com/blog/370046/201301/30172027-2efbcf5f62f34eae857b7c48c0f94d02.jpg)

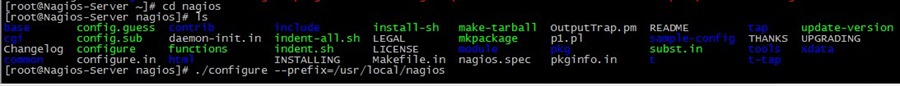
5.3 编译安装Nagios

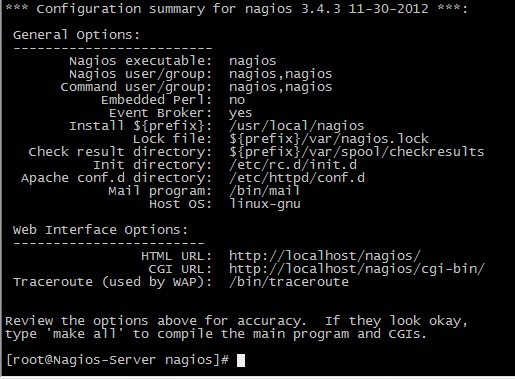
# wget <http://prdownloads.sourceforge.net/sourceforge/nagios/nagios-3.4.3.tar.gz>

# tar zxvf nagios-3.4.3.tar.gz

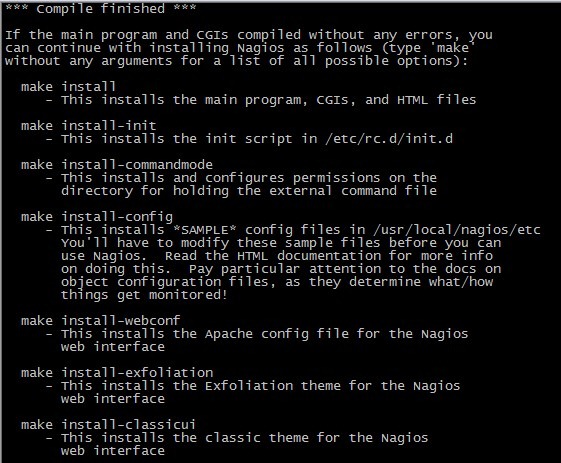
# cd nagios

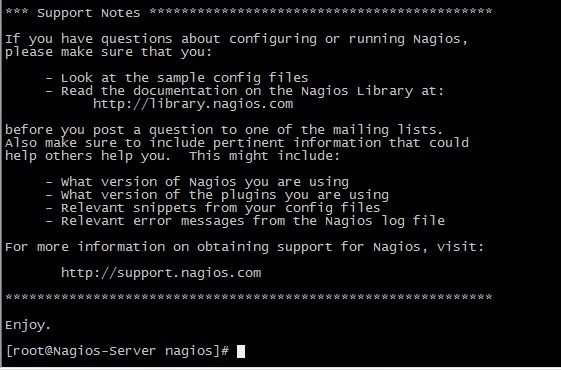
# ./configure --prefix=/usr/local/nagios

[](http://images.cnitblog.com/blog/370046/201301/30172029-51aac459f3a34106af7db6aaaa403d9e.jpg)

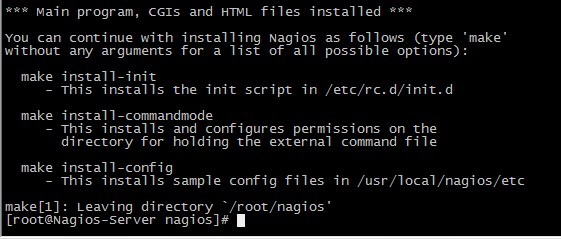
[](http://images.cnitblog.com/blog/370046/201301/30172030-18fdb7961c474b75a2b033e2a4171519.jpg)

# make all

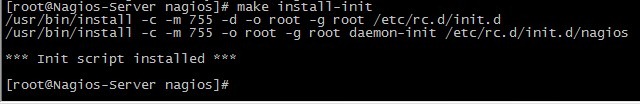
[](http://images.cnitblog.com/blog/370046/201301/30172031-5c6a7463c11d4544bee8f2eed12d7647.jpg)

[](http://images.cnitblog.com/blog/370046/201301/30172034-ebe3fcafffe94564b8161e6b78fe64c8.jpg)

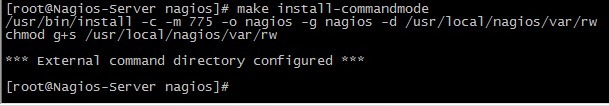
# make install

[](http://images.cnitblog.com/blog/370046/201301/30172035-b01026f0079c477c815351fb70635f40.jpg)

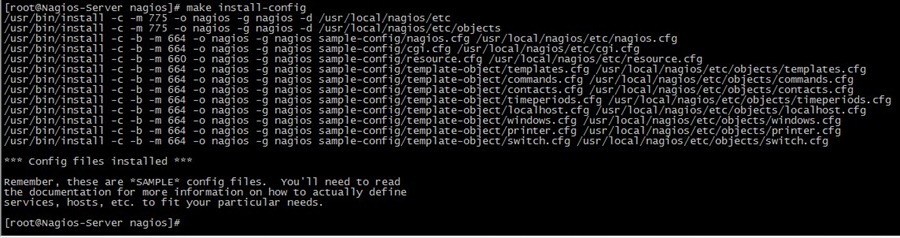
# make install-init

[](http://images.cnitblog.com/blog/370046/201301/30172038-629553af940645d3a1e1b34db39106e7.jpg)

# make install-commandmode



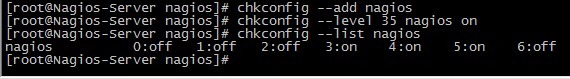
# make install-config



# chkconfig --add nagios

# chkconfig --level 35 nagios on

# chkconfig --list nagios



5.4 验证程序是否被正确安装

切换目录到安装路径（这里是/usr/local/nagios），看是否存在etc、bin、sbin、share、var 这五个目录，如果存在则可以表明程序被正确的安装到系统了。Nagios 各个目录用途说明如下：

|  |  |
| --- | --- |
| bin | Nagios 可执行程序所在目录 |
| etc | Nagios 配置文件所在目录 |
| sbin | Nagios CGI 文件所在目录，也就是执行外部命令所需文件所在的目录 |
| share | Nagios网页文件所在的目录 |
| libexec | Nagios 外部插件所在目录 |
| var | Nagios 日志文件、lock 等文件所在的目录 |
| var/archives | Nagios 日志自动归档目录 |
| var/rw | 用来存放外部命令文件的目录 |

5.5 安装Nagios 插件

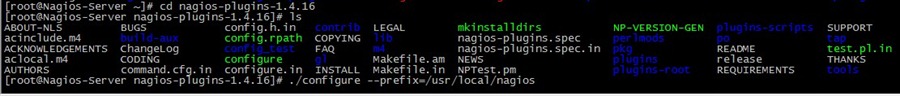
# wget <http://prdownloads.sourceforge.net/sourceforge/nagiosplug/nagios-plugins-1.4.16.tar.gz>

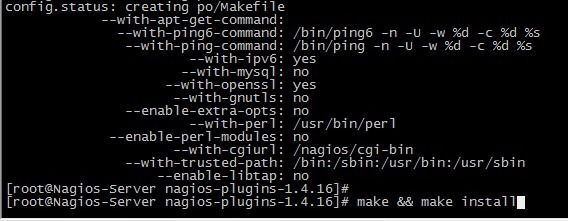
# tar zxvf nagios-plugins-1.4.16.tar.gz

# cd nagios-plugins-1.4.16

# ./configure --prefix=/usr/local/nagios

# make && make install





5.6 安装与配置Apache和Php

Apache 和Php 不是安装nagios 所必须的，但是nagios提供了web监控界面，通过web监控界面可以清晰的看到被监控主机、资源的运行状态，因此，安装一个web服务是很必要的。   
需要注意的是，nagios在nagios3.1.x版本以后，配置web监控界面时需要php的支持。这里我们下载的nagios版本为nagios-3.4.3，因此在编译安装完成apache后，还需要编译php模块，这里选取的php版本为php5.4.10。

a. 安装Apache

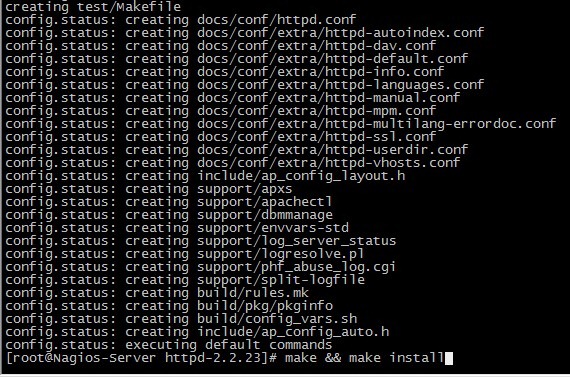
# wget [http://labs.mop.com/apache-mirror//httpd/httpd-2.2.23.tar.gz](http://labs.mop.com/apache-mirror/httpd/httpd-2.2.23.tar.gz)

# tar zxvf httpd-2.2.23.tar.gz

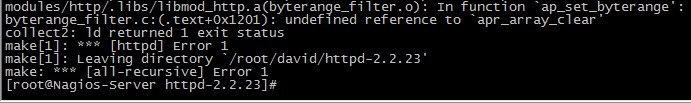
# cd httpd-2.2.23

# ./configure --prefix=/usr/local/apache2

# make && make install



若出现错误：



则在编译时入加 --with-included-apr 即可解决。

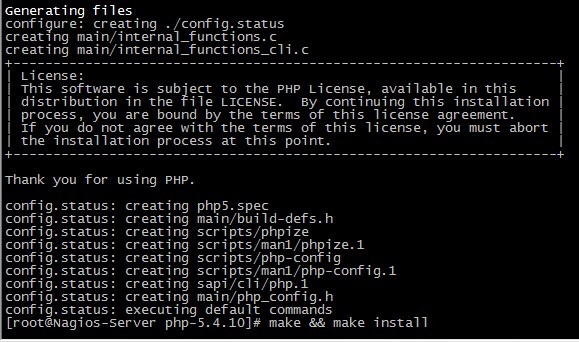
b. 安装Php

# wget <http://cn2.php.net/distributions/php-5.4.10.tar.gz>

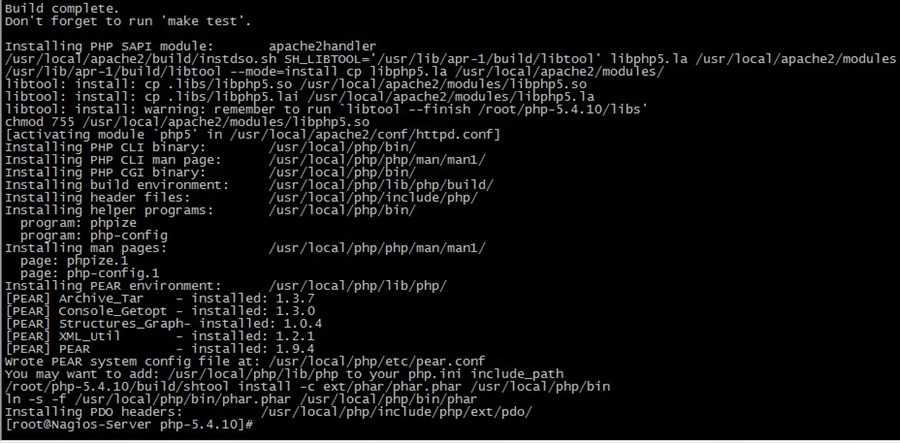
# tar zxvf php-5.4.10.tar.gz

# cd php-5.4.10

# ./configure --prefix=/usr/local/php --with-apxs2=/usr/local/apache2/bin/apxs

[](http://images.cnitblog.com/blog/370046/201301/30172050-6bda75716f614a6ab7f2acf8de422062.jpg)

# make && make install

[](http://images.cnitblog.com/blog/370046/201301/30172054-97af3f5b712041f78d9e44d94ac17bbd.jpg)

c. 配置apache   
找到apache 的配置文件/usr/local/apache2/conf/httpd.conf   
找到：

User daemon

Group daemon

修改为

User nagios

Group nagios

然后找到

<IfModule dir\_module>

　　DirectoryIndex index.html

</IfModule>

修改为

<IfModule dir\_module>

　　DirectoryIndex index.html index.php

</IfModule>

接着增加如下内容：

AddType application/x-httpd-php .php

为了安全起见，一般情况下要让nagios 的web 监控页面必须经过授权才能访问，这需要增加验证配置，即在httpd.conf 文件最后添加如下信息：

#setting for nagios

ScriptAlias /nagios/cgi-bin "/usr/local/nagios/sbin"

<Directory "/usr/local/nagios/sbin">

AuthType Basic

Options ExecCGI

AllowOverride None

Order allow,deny

Allow from all

AuthName "Nagios Access"

AuthUserFile /usr/local/nagios/etc/htpasswd //用于此目录访问身份验证的文件

Require valid-user

</Directory>

Alias /nagios "/usr/local/nagios/share"

<Directory "/usr/local/nagios/share">

AuthType Basic

Options None

AllowOverride None

Order allow,deny

Allow from all

AuthName "nagios Access"

AuthUserFile /usr/local/nagios/etc/htpasswd

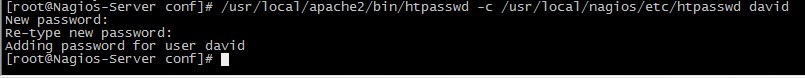
Require valid-user

</Directory>

d. 创建apache目录验证文件

在上面的配置中，指定了目录验证文件htpasswd，下面要创建这个文件：

# /usr/local/apache2/bin/htpasswd -c /usr/local/nagios/etc/htpasswd david

[](http://images.cnitblog.com/blog/370046/201301/30172056-36df7ebedd9d448a8b38a2ccd4633ebe.jpg)   
这样就在/usr/local/nagios/etc 目录下创建了一个htpasswd 验证文件，当通过[http://192.168.1.108/nagios/](http://172.16.1.124/nagios/) 访问时就需要输入用户名和密码了。

e. 查看认证文件的内容

# cat /usr/local/nagios/etc/htpasswd

[26](http://images.cnitblog.com/blog/370046/201301/30172058-a256928a00cf40769bc3a5815401732b.jpg)

f. 启动apache 服务

# /usr/local/apache2/bin/apachectl start

到这里nagios 的安装也就基本完成了，你可以通过web来访问了。



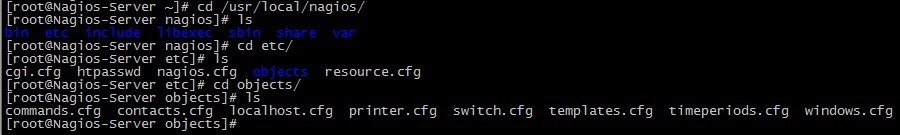


六、配置Nagios

Nagios 主要用于监控一台或者多台本地主机及远程的各种信息，包括本机资源及对外的服务等。默认的Nagios 配置没有任何监控内容，仅是一些模板文件。若要让Nagios 提供服务，就必须修改配置文件，增加要监控的主机和服务，下面将详细介绍。

6.1 默认配置文件介绍

Nagios 安装完毕后，默认的配置文件在/usr/local/nagios/etc目录下。



每个文件或目录含义如下表所示：

|  |  |
| --- | --- |
| 文件名或目录名 | 用途 |
| cgi.cfg | 控制CGI访问的配置文件 |
| nagios.cfg | Nagios 主配置文件 |
| resource.cfg | 变量定义文件，又称为资源文件，在些文件中定义变量，以便由其他配置文件引用，如$USER1$ |
| objects | objects 是一个目录，在此目录下有很多配置文件模板，用于定义Nagios 对象 |
| objects/commands.cfg | 命令定义配置文件，其中定义的命令可以被其他配置文件引用 |
| objects/contacts.cfg | 定义联系人和联系人组的配置文件 |
| objects/localhost.cfg | 定义监控本地主机的配置文件 |
| objects/printer.cfg | 定义监控打印机的一个配置文件模板，默认没有启用此文件 |
| objects/switch.cfg | 定义监控路由器的一个配置文件模板，默认没有启用此文件 |
| objects/templates.cfg | 定义主机和服务的一个模板配置文件，可以在其他配置文件中引用 |
| objects/timeperiods.cfg | 定义Nagios 监控时间段的配置文件 |
| objects/windows.cfg | 监控Windows 主机的一个配置文件模板，默认没有启用此文件 |

6.2 配置文件之间的关系

在nagios的配置过程中涉及到的几个定义有：主机、主机组，服务、服务组，联系人、联系人组，监控时间，监控命令等，从这些定义可以看出，nagios各个配置文件之间是互为关联，彼此引用的。

成功配置出一台nagios监控系统，必须要弄清楚每个配置文件之间依赖与被依赖的关系，最重要的有四点：

第一：定义监控哪些主机、主机组、服务和服务组；   
第二：定义这个监控要用什么命令实现；   
第三：定义监控的时间段；   
第四：定义主机或服务出现问题时要通知的联系人和联系人组。

6.3 配置Nagios

 为了能更清楚的说明问题，同时也为了维护方便，建议将nagios各个定义对象创建独立的配置文件：

 创建hosts.cfg文件来定义主机和主机组

 创建services.cfg文件来定义服务

 用默认的contacts.cfg文件来定义联系人和联系人组

 用默认的commands.cfg文件来定义命令

 用默认的timeperiods.cfg来定义监控时间段

 用默认的templates.cfg文件作为资源引用文件

a. templates.cfg文件

nagios主要用于监控主机资源以及服务，在nagios配置中称为对象，为了不必重复定义一些监控对象，Nagios引入了一个模板配置文件，将一些共性的属性定义成模板，以便于多次引用。这就是templates.cfg的作用。

下面详细介绍下templates.cfg文件中每个参数的含义：

[复制代码](javascript:void(0);)

define contact{

name generic-contact ; 联系人名称

service\_notification\_period 24x7 ; 当服务出现异常时，发送通知的时间段，这个时间段"24x7"在timeperiods.cfg文件中定义

host\_notification\_period 24x7 ; 当主机出现异常时，发送通知的时间段，这个时间段"24x7"在timeperiods.cfg文件中定义

service\_notification\_options w,u,c,r ; 这个定义的是“通知可以被发出的情况”。w即warn，表示警告状态，u即unknown，表示不明状态;  
 ; c即criticle，表示紧急状态，r即recover，表示恢复状态;  
 ; 也就是在服务出现警告状态、未知状态、紧急状态和重新恢复状态时都发送通知给使用者。

host\_notification\_options d,u,r ; 定义主机在什么状态下需要发送通知给使用者，d即down，表示宕机状态;  
 ; u即unreachable，表示不可到达状态，r即recovery，表示重新恢复状态。

service\_notification\_commands notify-service-by-email ; 服务故障时，发送通知的方式，可以是邮件和短信，这里发送的方式是邮件;  
 ; 其中“notify-service-by-email”在commands.cfg文件中定义。

host\_notification\_commands notify-host-by-email ; 主机故障时，发送通知的方式，可以是邮件和短信，这里发送的方式是邮件;  
 ; 其中“notify-host-by-email”在commands.cfg文件中定义。

register 0 ; DONT REGISTER THIS DEFINITION - ITS NOT A REAL CONTACT, JUST A TEMPLATE!

}

define host{

name generic-host ; 主机名称，这里的主机名，并不是直接对应到真正机器的主机名;  
 ; 乃是对应到在主机配置文件里所设定的主机名。

notifications\_enabled 1 ; Host notifications are enabled

event\_handler\_enabled 1 ; Host event handler is enabled

flap\_detection\_enabled 1 ; Flap detection is enabled

failure\_prediction\_enabled 1 ; Failure prediction is enabled

process\_perf\_data 1 ; 其值可以为0或1，其作用为是否启用Nagios的数据输出功能;  
 ; 如果将此项赋值为1，那么Nagios就会将收集的数据写入某个文件中，以备提取。

retain\_status\_information 1 ; Retain status information across program restarts

retain\_nonstatus\_information 1 ; Retain non-status information across program restarts

notification\_period 24x7 ; 指定“发送通知”的时间段，也就是可以在什么时候发送通知给使用者。

register 0 ; DONT REGISTER THIS DEFINITION - ITS NOT A REAL HOST, JUST A TEMPLATE!

}

define host{

name linux-server ; 主机名称

use generic-host ; use表示引用，也就是将主机generic-host的所有属性引用到linux-server中来;  
 ; 在nagios配置中，很多情况下会用到引用。

check\_period 24x7 ; 这里的check\_period告诉nagios检查主机的时间段

check\_interval 5 ; nagios对主机的检查时间间隔，这里是5分钟。

retry\_interval 1 ; 重试检查时间间隔，单位是分钟。

max\_check\_attempts 10 ; nagios对主机的最大检查次数，也就是nagios在检查发现某主机异常时，并不马上判断为异常状况;  
 ; 而是多试几次，因为有可能只是一时网络太拥挤，或是一些其他原因，让主机受到了一点影响;  
 ; 这里的10就是最多试10次的意思。

check\_command check-host-alive ; 指定检查主机状态的命令，其中“check-host-alive”在commands.cfg文件中定义。

notification\_period 24x7 ; 主机故障时，发送通知的时间范围，其中“workhours”在timeperiods.cfg中进行了定义;  
 ; 下面会陆续讲到。

notification\_interval 10 ; 在主机出现异常后，故障一直没有解决，nagios再次对使用者发出通知的时间。单位是分钟;  
 ; 如果你觉得，所有的事件只需要一次通知就够了，可以把这里的选项设为0

notification\_options d,u,r ; 定义主机在什么状态下可以发送通知给使用者，d即down，表示宕机状态;  
 ; u即unreachable，表示不可到达状态;  
 ; r即recovery，表示重新恢复状态。

contact\_groups ts ; 指定联系人组，这个“admins”在contacts.cfg文件中定义。

register 0 ; DONT REGISTER THIS DEFINITION - ITS NOT A REAL HOST, JUST A TEMPLATE!

}

define host{

name windows-server ; The name of this host template

use generic-host ; Inherit default values from the generic-host template

check\_period 24x7 ; By default, Windows servers are monitored round the clock

check\_interval 5 ; Actively check the server every 5 minutes

retry\_interval 1 ; Schedule host check retries at 1 minute intervals

max\_check\_attempts 10 ; Check each server 10 times (max)

check\_command check-host-alive ; Default command to check if servers are "alive"

notification\_period 24x7 ; Send notification out at any time - day or night

notification\_interval 10 ; Resend notifications every 30 minutes

notification\_options d,r ; Only send notifications for specific host states

contact\_groups ts ; Notifications get sent to the admins by default

hostgroups windows-servers ; Host groups that Windows servers should be a member of

register 0 ; DONT REGISTER THIS - ITS JUST A TEMPLATE

}

define service{

name generic-service ; 定义一个服务名称

active\_checks\_enabled 1 ; Active service checks are enabled

passive\_checks\_enabled 1 ; Passive service checks are enabled/accepted

parallelize\_check 1 ; Active service checks should be parallelized;  
 ; (disabling this can lead to major performance problems)

obsess\_over\_service 1 ; We should obsess over this service (if necessary)

check\_freshness 0 ; Default is to NOT check service 'freshness'

notifications\_enabled 1 ; Service notifications are enabled

event\_handler\_enabled 1 ; Service event handler is enabled

flap\_detection\_enabled 1 ; Flap detection is enabled

failure\_prediction\_enabled 1 ; Failure prediction is enabled

process\_perf\_data 1 ; Process performance data

retain\_status\_information 1 ; Retain status information across program restarts

retain\_nonstatus\_information 1 ; Retain non-status information across program restarts

is\_volatile 0 ; The service is not volatile

check\_period 24x7 ; 这里的check\_period告诉nagios检查服务的时间段。

max\_check\_attempts 3 ; nagios对服务的最大检查次数。

normal\_check\_interval 5 ; 此选项是用来设置服务检查时间间隔，也就是说，nagios这一次检查和下一次检查之间所隔的时间;  
 ; 这里是5分钟。

retry\_check\_interval 2 ; 重试检查时间间隔，单位是分钟。

contact\_groups ts ; 指定联系人组

notification\_options w,u,c,r ; 这个定义的是“通知可以被发出的情况”。w即warn，表示警告状态;  
 ; u即unknown，表示不明状态;  
 ; c即criticle，表示紧急状态，r即recover，表示恢复状态;  
 ; 也就是在服务出现警告状态、未知状态、紧急状态和重新恢复后都发送通知给使用者。

notification\_interval 10 ; Re-notify about service problems every hour

notification\_period 24x7 ; 指定“发送通知”的时间段，也就是可以在什么时候发送通知给使用者。

register 0 ; DONT REGISTER THIS DEFINITION - ITS NOT A REAL SERVICE, JUST A TEMPLATE!

}

define service{

name local-service ; The name of this service template

use generic-service ; Inherit default values from the generic-service definition

max\_check\_attempts 4 ; Re-check the service up to 4 times in order to determine its final (hard) state

normal\_check\_interval 5 ; Check the service every 5 minutes under normal conditions

retry\_check\_interval 1 ; Re-check the service every minute until a hard state can be determined

register 0 ; DONT REGISTER THIS DEFINITION - ITS NOT A REAL SERVICE, JUST A TEMPLATE!

}

b. resource.cfg文件

resource.cfg是nagios的变量定义文件，文件内容只有一行：

$USER1$=/usr/local/nagios/libexec

其中，变量$USER1$指定了安装nagios插件的路径，如果把插件安装在了其它路径，只需在这里进行修改即可。需要注意的是，变量必须先定义，然后才能在其它配置文件中进行引用。

c. commands.cfg文件

此文件默认是存在的，无需修改即可使用，当然如果有新的命令需要加入时，在此文件进行添加即可。

#notify-host-by-email命令的定义   
define command{

command\_name notify-host-by-email #命令名称，即定义了一个主机异常时发送邮件的命令。

command\_line /usr/bin/printf "%b" "\*\*\*\*\* Nagios \*\*\*\*\*\n\nNotification Type: $NOTIFICATIONTYPE$\nHost: $HOSTNAME$\nState: $HOSTSTATE$\nAddress: $HOSTADDRESS$\nInfo: $HOSTOUTPUT$\n\nDate/Time: $LONGDATETIME$\n" | /bin/mail -s "\*\* $NOTIFICATIONTYPE$ Host Alert: $HOSTNAME$ is $HOSTSTATE$ \*\*" $CONTACTEMAIL$ #命令具体的执行方式。

}

#notify-service-by-email命令的定义   
define command{

command\_name notify-service-by-email #命令名称，即定义了一个服务异常时发送邮件的命令

command\_line /usr/bin/printf "%b" "\*\*\*\*\* Nagios \*\*\*\*\*\n\nNotification Type: $NOTIFICATIONTYPE$\n\nService: $SERVICEDESC$\nHost: $HOSTALIAS$\nAddress: $HOSTADDRESS$\nState: $SERVICESTATE$\n\nDate/Time: $LONGDATETIME$\n\nAdditional Info:\n\n$SERVICEOUTPUT$\n" | /bin/mail -s "\*\* $NOTIFICATIONTYPE$ Service Alert: $HOSTALIAS$/$SERVICEDESC$ is $SERVICESTATE$ \*\*" $CONTACTEMAIL$

}  
#check-host-alive命令的定义

define command{

command\_name check-host-alive #命令名称，用来检测主机状态。

command\_line $USER1$/check\_ping -H $HOSTADDRESS$ -w 3000.0,80% -c 5000.0,100% -p 5   
 # 这里的变量$USER1$在resource.cfg文件中进行定义，即$USER1$=/usr/local/nagios/libexec;  
 # 那么check\_ping的完整路径为/usr/local/nagios/libexec/check\_ping;  
 # “-w 3000.0,80%”中“-w”说明后面的一对值对应的是“WARNING”状态，“80%”是其临界值。  
 # “-c 5000.0,100%”中“-c”说明后面的一对值对应的是“CRITICAL”，“100%”是其临界值。  
 # “-p 1”说明每次探测发送一个包。  
 }

define command{

command\_name check\_local\_disk

command\_line $USER1$/check\_disk -w $ARG1$ -c $ARG2$ -p $ARG3$ #$ARG1$是指在调用这个命令的时候，命令后面的第一个参数。

}

define command{

command\_name check\_local\_load

command\_line $USER1$/check\_load -w $ARG1$ -c $ARG2$

}

define command{

command\_name check\_local\_procs

command\_line $USER1$/check\_procs -w $ARG1$ -c $ARG2$ -s $ARG3$

}

define command{

command\_name check\_local\_users

command\_line $USER1$/check\_users -w $ARG1$ -c $ARG2$

}

define command{

command\_name check\_local\_swap

command\_line $USER1$/check\_swap -w $ARG1$ -c $ARG2$

}

define command{

command\_name check\_ftp

command\_line $USER1$/check\_ftp -H $HOSTADDRESS$ $ARG1$

}

define command{

command\_name check\_http

command\_line $USER1$/check\_http -I $HOSTADDRESS$ $ARG1$

}

define command{

command\_name check\_ssh

command\_line $USER1$/check\_ssh $ARG1$ $HOSTADDRESS$

}

define command{

command\_name check\_ping

command\_line $USER1$/check\_ping -H $HOSTADDRESS$ -w $ARG1$ -c $ARG2$ -p 5

}

define command{

command\_name check\_nt

command\_line $USER1$/check\_nt -H $HOSTADDRESS$ -p 12489 -v $ARG1$ $ARG2$

}

d. hosts.cfg文件

此文件默认不存在，需要手动创建，hosts.cfg主要用来指定被监控的主机地址以及相关属性信息，根据实验目标配置如下：

define host{

use linux-server #引用主机linux-server的属性信息，linux-server主机在templates.cfg文件中进行了定义。

host\_name Nagios-Linux #主机名

alias Nagios-Linux #主机别名

address 192.168.1.111 #被监控的主机地址，这个地址可以是ip，也可以是域名。

}

#定义一个主机组

define hostgroup{

hostgroup\_name bsmart-servers #主机组名称，可以随意指定。

alias bsmart servers #主机组别名

members Nagios-Linux #主机组成员，其中“Nagios-Linux”就是上面定义的主机。   
 }

注意：在/usr/local/nagios/etc/objects 下默认有localhost.cfg 和windows.cfg 这两个配置文件，localhost.cfg 文件是定义监控主机本身的，windows.cfg 文件是定义windows 主机的，其中包括了对host 和相关services 的定义。所以在本次实验中，将直接在localhost.cfg 中定义监控主机（Nagios-Server），在windows.cfg中定义windows 主机（Nagios-Windows）。根据自己的需要修改其中的相关配置，详细如下：

localhost.cfg

define host{

use linux-server ; Name of host template to use

; This host definition will inherit all variables that are defined

; in (or inherited by) the linux-server host template definition.

host\_name Nagios-Server

alias Nagios-Server

address 127.0.0.1

}

define hostgroup{

hostgroup\_name linux-servers ; The name of the hostgroup

alias Linux Servers ; Long name of the group

members Nagios-Server ; Comma separated list of hosts that belong to this group

}

define service{

use local-service ; Name of service template to use

host\_name Nagios-Server

service\_description PING

check\_command check\_ping!100.0,20%!500.0,60%

}

define service{

use local-service ; Name of service template to use

host\_name Nagios-Server

service\_description Root Partition

check\_command check\_local\_disk!20%!10%!/

}

define service{

use local-service ; Name of service template to use

host\_name Nagios-Server

service\_description Current Users

check\_command check\_local\_users!20!50

}

define service{

use local-service ; Name of service template to use

host\_name Nagios-Server

service\_description Total Processes

check\_command check\_local\_procs!250!400!RSZDT

}

define service{

use local-service ; Name of service template to use

host\_name Nagios-Server

service\_description Current Load

check\_command check\_local\_load!5.0,4.0,3.0!10.0,6.0,4.0

}

define service{

use local-service ; Name of service template to use

host\_name Nagios-Server

service\_description Swap Usage

check\_command check\_local\_swap!20!10

}

define service{

use local-service ; Name of service template to use

host\_name Nagios-Server

service\_description SSH

check\_command check\_ssh

notifications\_enabled 0

}

define service{

use local-service ; Name of service template to use

host\_name Nagios-Server

service\_description HTTP

check\_command check\_http

notifications\_enabled 0

}

windows.cfg

define host{

use windows-server ; Inherit default values from a template

host\_name Nagios-Windows ; The name we're giving to this host

alias My Windows Server ; A longer name associated with the host

address 192.168.1.113 ; IP address of the host

}

define hostgroup{

hostgroup\_name windows-servers ; The name of the hostgroup

alias Windows Servers ; Long name of the group

}

define service{

use generic-service

host\_name Nagios-Windows

service\_description NSClient++ Version

check\_command check\_nt!CLIENTVERSION

}

define service{

use generic-service

host\_name Nagios-Windows

service\_description Uptime

check\_command check\_nt!UPTIME

}

define service{

use generic-service

host\_name Nagios-Windows

service\_description CPU Load

check\_command check\_nt!CPULOAD!-l 5,80,90

}

define service{

use generic-service

host\_name Nagios-Windows

service\_description Memory Usage

check\_command check\_nt!MEMUSE!-w 80 -c 90

}

define service{

use generic-service

host\_name Nagios-Windows

service\_description C:\ Drive Space

check\_command check\_nt!USEDDISKSPACE!-l c -w 80 -c 90

}

define service{

use generic-service

host\_name Nagios-Windows

service\_description W3SVC

check\_command check\_nt!SERVICESTATE!-d SHOWALL -l W3SVC

}

define service{

use generic-service

host\_name Nagios-Windows

service\_description Explorer

check\_command check\_nt!PROCSTATE!-d SHOWALL -l Explorer.exe

}

e. services.cfg文件

此文件默认也不存在，需要手动创建，services.cfg文件主要用于定义监控的服务和主机资源，例如监控http服务、ftp服务、主机磁盘空间、主机系统负载等等。Nagios-Server 和Nagios-Windows 相关服务已在相应的配置文件中定义，所以这里只需要定义Nagios-Linux 相关服务即可，这里只定义一个检测是否存活的服务来验证配置文件的正确性，其他服务的定义将在后面讲到。

define service{

use local-service #引用local-service服务的属性值，local-service在templates.cfg文件中进行了定义。

host\_name Nagios-Linux #指定要监控哪个主机上的服务，“Nagios-Server”在hosts.cfg文件中进行了定义。

service\_description check-host-alive #对监控服务内容的描述，以供维护人员参考。

check\_command check-host-alive #指定检查的命令。  
 }

f. contacts.cfg文件

contacts.cfg是一个定义联系人和联系人组的配置文件，当监控的主机或者服务出现故障，nagios会通过指定的通知方式（邮件或者短信）将信息发给这里指定的联系人或者使用者。

define contact{

contact\_name David #联系人的名称,这个地方不要有空格

use                             generic-contact #引用generic-contact的属性信息，其中“generic-contact”在templates.cfg文件中进行定义

alias                           Nagios Admin

email david.tang@bsmart.cn

}

define contactgroup{

contactgroup\_name ts #联系人组的名称,同样不能空格

alias Technical Support #联系人组描述

members David #联系人组成员，其中“david”就是上面定义的联系人，如果有多个联系人则以逗号相隔

}

g. timeperiods.cfg文件

此文件只要用于定义监控的时间段，下面是一个配置好的实例：

#下面是定义一个名为24x7的时间段，即监控所有时间段

define timeperiod{

timeperiod\_name 24x7 #时间段的名称,这个地方不要有空格

alias 24 Hours A Day, 7 Days A Week

sunday 00:00-24:00

monday 00:00-24:00

tuesday 00:00-24:00

wednesday 00:00-24:00

thursday 00:00-24:00

friday 00:00-24:00

saturday 00:00-24:00

}

#下面是定义一个名为workhours的时间段，即工作时间段。

define timeperiod{

timeperiod\_name workhours

alias Normal Work Hours

monday 09:00-17:00

tuesday 09:00-17:00

wednesday 09:00-17:00

thursday 09:00-17:00

friday 09:00-17:00

}

h. cgi.cfg文件

此文件用来控制相关cgi脚本，如果想在nagios的web监控界面执行cgi脚本，例如重启nagios进程、关闭nagios通知、停止nagios主机检测等，这时就需要配置cgi.cfg文件了。  
由于nagios的web监控界面验证用户为david，所以只需在cgi.cfg文件中添加此用户的执行权限就可以了，需要修改的配置信息如下：

default\_user\_name=david

authorized\_for\_system\_information=nagiosadmin,david

authorized\_for\_configuration\_information=nagiosadmin,david

authorized\_for\_system\_commands=david

authorized\_for\_all\_services=nagiosadmin,david

authorized\_for\_all\_hosts=nagiosadmin,david

authorized\_for\_all\_service\_commands=nagiosadmin,david

authorized\_for\_all\_host\_commands=nagiosadmin,david

i. nagios.cfg文件

nagios.cfg默认的路径为/usr/local/nagios/etc/nagios.cfg，是nagios的核心配置文件，所有的对象配置文件都必须在这个文件中进行定义才能发挥其作用，这里只需将对象配置文件在Nagios.cfg文件中进行引用即可。

log\_file=/usr/local/nagios/var/nagios.log # 定义nagios日志文件的路径

cfg\_file=/usr/local/nagios/etc/objects/commands.cfg # “cfg\_file”变量用来引用对象配置文件，如果有更多的对象配置文件，在这里依次添加即可。  
cfg\_file=/usr/local/nagios/etc/objects/contacts.cfg  
cfg\_file=/usr/local/nagios/etc/objects/hosts.cfg  
cfg\_file=/usr/local/nagios/etc/objects/services.cfg  
cfg\_file=/usr/local/nagios/etc/objects/timeperiods.cfg

cfg\_file=/usr/local/nagios/etc/objects/templates.cfg  
cfg\_file=/usr/local/nagios/etc/objects/localhost.cfg # 本机配置文件  
cfg\_file=/usr/local/nagios/etc/objects/windows.cfg # windows 主机配置文件

object\_cache\_file=/usr/local/nagios/var/objects.cache # 该变量用于指定一个“所有对象配置文件”的副本文件，或者叫对象缓冲文件

precached\_object\_file=/usr/local/nagios/var/objects.precache  
resource\_file=/usr/local/nagios/etc/resource.cfg # 该变量用于指定nagios资源文件的路径，可以在nagios.cfg中定义多个资源文件。

status\_file=/usr/local/nagios/var/status.dat # 该变量用于定义一个状态文件，此文件用于保存nagios的当前状态、注释和宕机信息等。

status\_update\_interval=10 # 该变量用于定义状态文件（即status.dat）的更新时间间隔，单位是秒，最小更新间隔是1秒。

nagios\_user=nagios # 该变量指定了Nagios进程使用哪个用户运行。  
nagios\_group=nagios # 该变量用于指定Nagios使用哪个用户组运行。

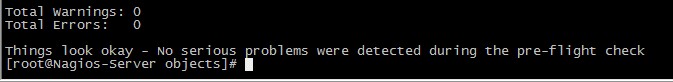
check\_external\_commands=1 # 该变量用于设置是否允许nagios在web监控界面运行cgi命令;  
 # 也就是是否允许nagios在web界面下执行重启nagios、停止主机/服务检查等操作;  
 # “1”为运行，“0”为不允许。

command\_check\_interval=10s # 该变量用于设置nagios对外部命令检测的时间间隔，如果指定了一个数字加一个"s"(如10s);  
 # 那么外部检测命令的间隔是这个数值以秒为单位的时间间隔;  
 # 如果没有用"s"，那么外部检测命令的间隔是以这个数值的“时间单位”的时间间隔。  
interval\_length=60 # 该变量指定了nagios的时间单位，默认值是60秒，也就是1分钟;  
 # 即在nagios配置中所有的时间单位都是分钟。

6.4 验证Nagios 配置文件的正确性

Nagios 在验证配置文件方面做的非常到位，只需通过一个命令即可完成：

# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg



Nagios提供的这个验证功能非常有用，在错误信息中通常会打印出错误的配置文件以及文件中的哪一行，这使得nagios的配置变得非常容易，报警信息通常是可以忽略的，因为一般那些只是建议性的。

看到上面这些信息就说明没问题了，然后启动Nagios 服务。

七、Nagios的启动与停止

7.1 启动Nagios

a. 通过初始化脚本启动nagios

# /etc/init.d/nagios start  
or  
# service nagios start

b. 手工方式启动nagios

通过nagios命令的“-d”参数来启动nagios守护进程：

# /usr/local/nagios/bin/nagios -d /usr/local/nagios/etc/nagios.cfg

7.2 重启Nagios

当修改了配置文件让其生效时，需要重启/重载Nagios服务。

a. 通过初始化脚本来重启nagios

# /etc/init.d/nagios reload  
or  
# /etc/init.d/nagios restart  
or  
# service nagios restart

b. 通过web监控页重启nagios

可以通过web监控页的 "Process Info" -> "Restart the Nagios process"来重启nagios



c. 手工方式平滑重启

# kill -HUP <nagios\_pid>

7.3 停止Nagios

a. 通过初始化脚本关闭nagios服务

# /etc/init.d/nagios stop  
or  
# service nagios stop

b. 通过web监控页停止nagios

可以通过web监控页的 "Process Info" -> "Shutdown the Nagios process"来停止nagios



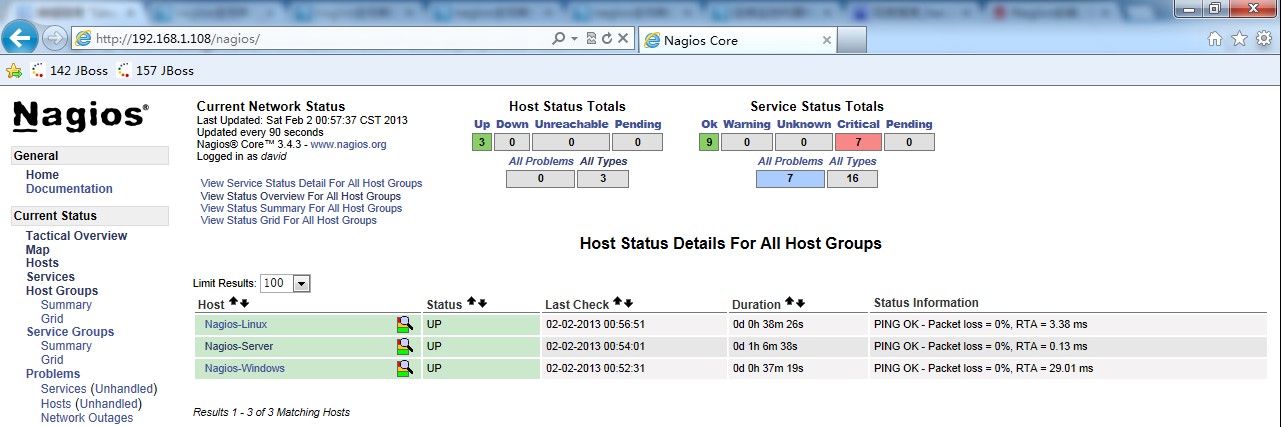
c. 手工方式停止Nagios

# kill <nagios\_pid>

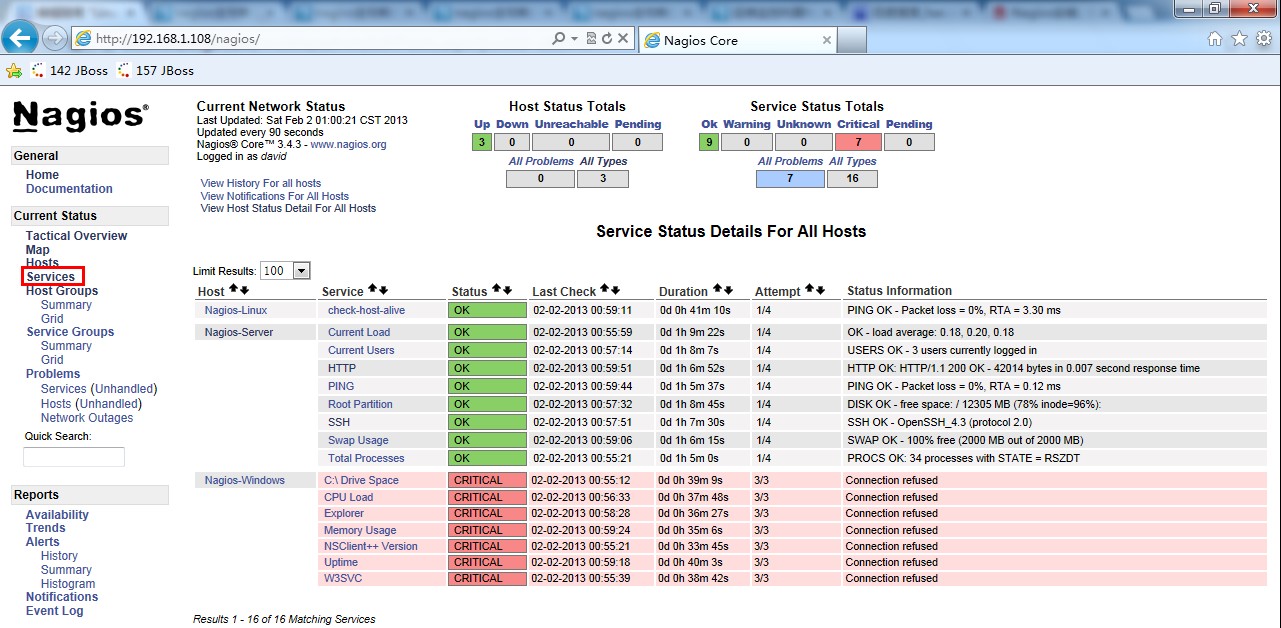
八、查看初步配置情况

8.1 启动完成之后，登录Nagios Web监控页[http://192.168.1.108/nagios/](http://172.16.1.124/nagios/) 查看相关信息。

8.2 点击左面的Current Status -> Hosts 可以看到所定义的三台主机已经全部UP了。



8.3 点击Current Status -> Services 查看服务监控情况。

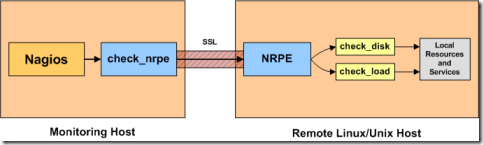


看到Nagios-Linux和Nagios-Server的服务状态已经OK了，但是Nagios-Windows的服务状态为CRITICAL，Status Information 提示Connection refused。因为Nagios-Windows上还未安装插件，内部服务还无法查看，所以出现这种情况。将在下面具体讲解。

九、利用NRPE监控远程Linux上的“本地信息”

上面已经对远程Linux 主机是否存活做了监控，而判断远程机器是否存活，我们可以使用ping 工具对其监测。还有一些远程主机服务，例如ftp、ssh、http，都是对外开放的服务，即使不用Nagios，我们也可以试的出来，随便找一台机器看能不能访问这些服务就行了。但是对于像磁盘容量，cpu负载这样的“本地信息”，Nagios只能监测自己所在的主机，而对其他的机器则显得有点无能为力。毕竟没得到被控主机的适当权限是不可能得到这些信息的。为了解决这个问题，nagios有这样一个附加组件--“NRPE”，用它就可以完成对Linux 类型主机"本地信息”的监控。

9.1 NRPE 工作原理

[](http://images.cnitblog.com/blog/370046/201301/30172018-ee257c664e12472590da4a3b503be33e.png)

NRPE 总共由两部分组成：

check\_nrpe 插件，位于监控主机上

NRPE daemon，运行在远程的Linux主机上(通常就是被监控机)

按照上图，整个的监控过程如下：

当Nagios 需要监控某个远程Linux 主机的服务或者资源情况时：

Nagios 会运行check\_nrpe 这个插件，告诉它要检查什么；

check\_nrpe 插件会连接到远程的NRPE daemon，所用的方式是SSL；

NRPE daemon 会运行相应的Nagios 插件来执行检查；

NRPE daemon 将检查的结果返回给check\_nrpe 插件，插件将其递交给nagios做处理。

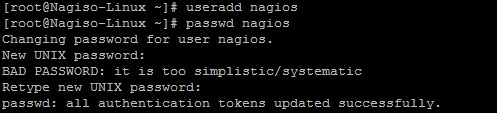
注意：NRPE daemon 需要Nagios 插件安装在远程的Linux主机上，否则，daemon不能做任何的监控。

9.2 在被监控机（Nagios-Linux）上

a. 增加用户&设定密码

# useradd nagios

# passwd nagios



b. 安装Nagios 插件

# tar zxvf nagios-plugins-1.4.16.tar.gz

# cd nagios-plugins-1.4.16

# ./configure --prefix=/usr/local/nagios

# make && make install

这一步完成后会在/usr/local/nagios/下生成三个目录include、libexec和share。

http://images.cnitblog.com/blog/370046/201302/02152814-8ad7b743e94d47e4a273d26f17fc039b.jpg

修改目录权限

# chown nagios.nagios /usr/local/nagios

# chown -R nagios.nagios /usr/local/nagios/libexec

http://images.cnitblog.com/blog/370046/201302/02152955-bba448f8c550421199e95f5abcd2a367.jpg

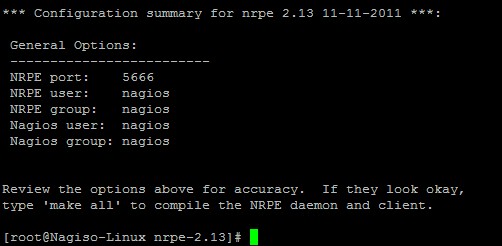
c. 安装NRPE

# wget http://prdownloads.sourceforge.net/sourceforge/nagios/nrpe-2.13.tar.gz

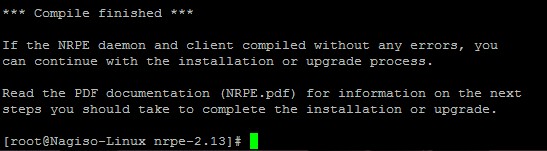
# tar zxvf nrpe-2.13.tar.gz

# cd nrpe-2.13

# ./configure



# make all



接下来安装NPRE插件，daemon和示例配置文件。

c.1 安装check\_nrpe 这个插件

# make install-plugin

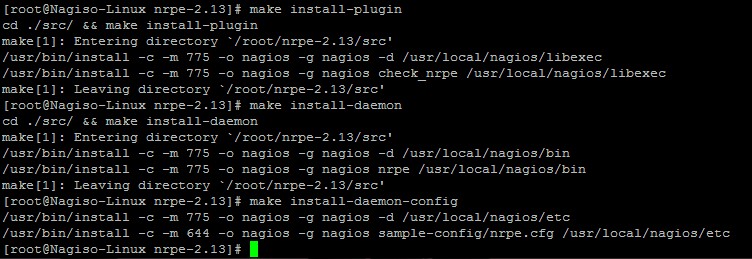
监控机需要安装check\_nrpe 这个插件，被监控机并不需要，我们在这里安装它只是为了测试目的。

c.2 安装deamon

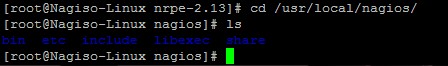
# make install-daemon

c.3 安装配置文件

# make install-daemon-config



现在再查看nagios 目录就会发现有5个目录了



按照安装文档的说明，是将NRPE deamon作为xinetd下的一个服务运行的。在这样的情况下xinetd就必须要先安装好，不过一般系统已经默认安装了。

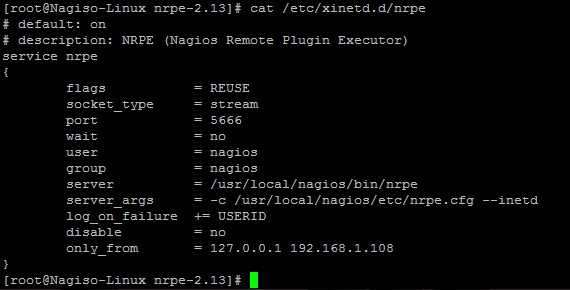
d. 安装xinted 脚本

# make install-xinetd

http://images.cnitblog.com/blog/370046/201302/02154307-bee4a49f8d87420a87899cd9c840c6d9.jpg

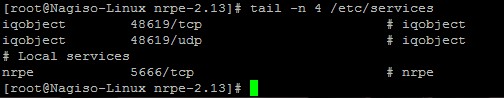
可以看到创建了这个文件/etc/xinetd.d/nrpe。

编辑这个脚本：



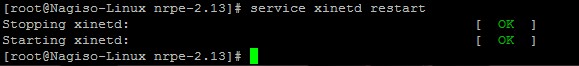
在only\_from 后增加监控主机的IP地址。

编辑/etc/services 文件，增加NRPE服务

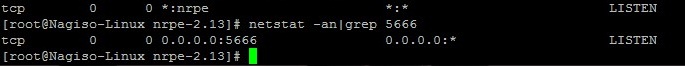


重启xinted 服务

# service xinetd restart



查看NRPE 是否已经启动



可以看到5666端口已经在监听了。

e. 测试NRPE是否则正常工作

使用上面在被监控机上安装的check\_nrpe 这个插件测试NRPE 是否工作正常。

# /usr/local/nagios/libexec/check\_nrpe -H localhost

会返回当前NRPE的版本

http://images.cnitblog.com/blog/370046/201302/02160042-6ae7c0101c264ff8b8076192ace36a23.jpg

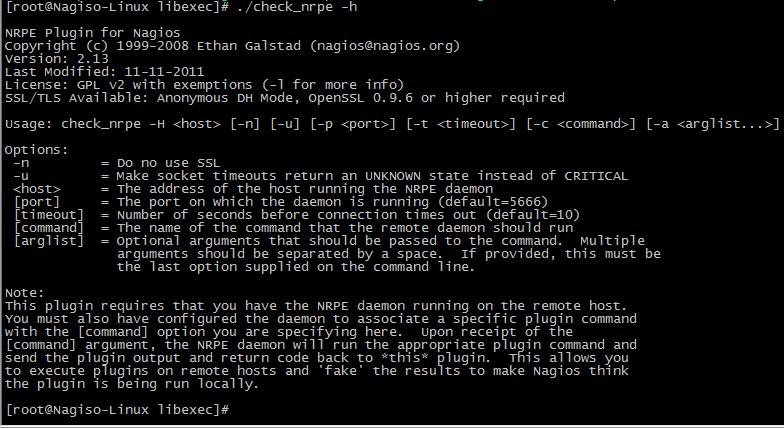
也就是在本地用check\_nrpe连接nrpe daemon是正常的。

注：为了后面工作的顺利进行，注意本地防火墙要打开5666能让外部的监控机访问。

f. check\_nrpe 命令用法

查看check\_nrpe 命令用法

# /usr/local/nagios/libexec/check\_nrpe –h



可以看到用法是：

check\_nrpe –H 被监控的主机 -c 要执行的监控命令

注意：-c 后面接的监控命令必须是nrpe.cfg 文件中定义的。也就是NRPE daemon只运行nrpe.cfg中所定义的命令。

g. 查看NRPE的监控命令

# cd /usr/local/nagios/etc

# cat nrpe.cfg |grep -v "^#"|grep -v "^$"

[root@Nagiso-Linux etc]# cat nrpe.cfg |grep -v "^#"|grep -v "^$"

log\_facility=daemon

pid\_file=/var/run/nrpe.pid

server\_port=5666

nrpe\_user=nagios

nrpe\_group=nagios

allowed\_hosts=127.0.0.1

dont\_blame\_nrpe=0

debug=0

command\_timeout=60

connection\_timeout=300

command[check\_users]=/usr/local/nagios/libexec/check\_users -w 5 -c 10

command[check\_load]=/usr/local/nagios/libexec/check\_load -w 15,10,5 -c 30,25,20

command[check\_sda1]=/usr/local/nagios/libexec/check\_disk -w 20% -c 10% -p /dev/sda1

command[check\_zombie\_procs]=/usr/local/nagios/libexec/check\_procs -w 5 -c 10 -s Z

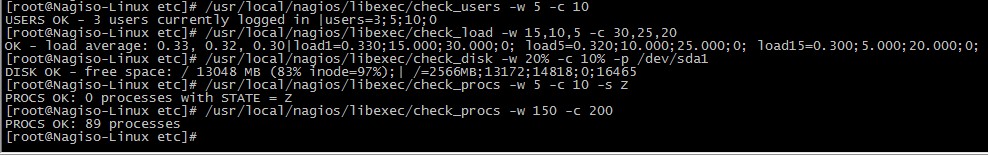
command[check\_total\_procs]=/usr/local/nagios/libexec/check\_procs -w 150 -c 200

[root@Nagiso-Linux etc]#

红色部分是命令名，也就是check\_nrpe 的-c 参数可以接的内容，等号 “=” 后面是实际执行的插件程序（这与commands.cfg 中定义命令的形式十分相似，只不过是写在了一行）。也就是说check\_users 就是等号后面/usr/local/nagios/libexec/check\_users -w 5 -c 10 的简称。

我们可以很容易知道上面这5行定义的命令分别是检测登陆用户数，cpu负载，sda1的容量，僵尸进程，总进程数。各条命令具体的含义见插件用法（执行“插件程序名 –h”）。

由于-c 后面只能接nrpe.cfg 中定义的命令，也就是说现在我们只能用上面定义的这五条命令。我们可以在本机实验一下。



9.3 在监控主机（Nagios-Server）上

之前已经将Nagios运行起来了，现在要做的事情是：

安装check\_nrpe 插件；

在commands.cfg 中创建check\_nrpe 的命令定义，因为只有在commands.cfg 中定义过的命令才能在services.cfg 中使用；

创建对被监控主机的监控项目；

9.3.1 安装check\_nrpe 插件

# tar zxvf nrpe-2.13.tar.gz

# cd nrpe-2.13

# ./configure

# make all

# make install-plugin

只运行这一步就行了，因为只需要check\_nrpe插件。

在Nagios-Linux 上我们已经装好了nrpe，现在我们测试一下监控机使用check\_nrpe 与被监控机运行的nrpe daemon之间的通信。

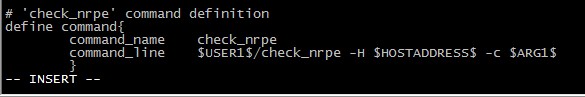
http://images.cnitblog.com/blog/370046/201302/02165718-6aa53e921dc2432a9738a5638aaa17e3.jpg

看到已经正确返回了NRPE的版本信息，说明一切正常。

9.3.2 在commands.cfg中增加对check\_nrpe的定义

# vi /usr/local/nagios/etc/objects/commands.cfg

在最后面增加如下内容：



意义如下：

# 'check\_nrpe' command definition

define command{

command\_name check\_nrpe # 定义命令名称为check\_nrpe,在services.cfg中要使用这个名称.

command\_line $USER1$/check\_nrpe -H $HOSTADDRESS$ -c $ARG1$ #这是定义实际运行的插件程序.  
 # 这个命令行的书写要完全按照check\_nrpe这个命令的用法,不知道用法的就用check\_nrpe –h查看.

}

-c 后面带的$ARG1$ 参数是传给nrpe daemon 执行的检测命令，之前说过了它必须是nrpe.cfg 中所定义的那5条命令中的其中一条。在services.cfg 中使用check\_nrpe 的时候要用 “!” 带上这个参数。

9.3.3 定义对Nagios-Linux 主机的监控

下面就可以在services.cfg 中定义对Nagios-Linux 主机的监控了。

define service{

use local-service

host\_name Nagios-Linux

service\_description Current Load

check\_command check\_nrpe!check\_load

}

define service{

use local-service

host\_name Nagios-Linux

service\_description Check Disk sda1

check\_command check\_nrpe!check\_sda1

}

define service{

use local-service

host\_name Nagios-Linux

service\_description Total Processes

check\_command check\_nrpe!check\_total\_procs

}

define service{

use local-service

host\_name Nagios-Linux

service\_description Current Users

check\_command check\_nrpe!check\_users

}

define service{

use local-service

host\_name Nagios-Linux

service\_description Check Zombie Procs

check\_command check\_nrpe!check\_zombie\_procs

}

还有一个任务是要监控Nagios-Linux 的swap 使用情况。但是在nrpe.cfg 中默认没有定义这个监控功能的命令。怎么办？手动在nrpe.cfg 中添加，也就是自定义NRPE命令。

现在我们要监控swap 分区，如果空闲空间小于20%则为警告状态 -> warning；如果小于10%则为严重状态 -> critical。我们可以查得需要使用check\_swap插件，完整的命令行应该是下面这样。

# /usr/local/nagios/libexec/check\_swap -w 20% -c 10%  
  
在被监控机（Nagios-Linux）上增加check\_swap 命令的定义

# vi /usr/local/nagios/etc/nrpe.cfg

增加下面这一行

command[check\_swap]=/usr/local/nagios/libexec/check\_swap -w 20% -c 10%

我们知道check\_swap 现在就可以作为check\_nrpe 的-c 的参数使用了

修改了配置文件，当然要重启。

如果你是以独立的daemon运行的nrpe，那么需要手动重启；如果你是在xinetd 下面运行的，则不需要。

由于本实验中nrpe 是xinetd 下运行的，所以不需要重启服务。  
  
在监控机（Nagios-Server）上增加这个check\_swap 监控项目

define service{

use local-service

host\_name Nagios-Linux

service\_description Check Swap

check\_command check\_nrpe!check\_swap

}

同理，Nagios-Linux 上我还开启了http 服务，需要监控一下，按照上面的做法，在被监控机（Nagios-Linux）上增加check\_http 命令的定义

# vi /usr/local/nagios/etc/nrpe.cfg

增加下面这一行

command[check\_http]=/usr/local/nagios/libexec/check\_http -I 127.0.0.1

在监控机（Nagios-Server）上增加check\_http 监控项目

define service{

use local-service

host\_name Nagios-Linux

service\_description HTTP

check\_command check\_nrpe!check\_http

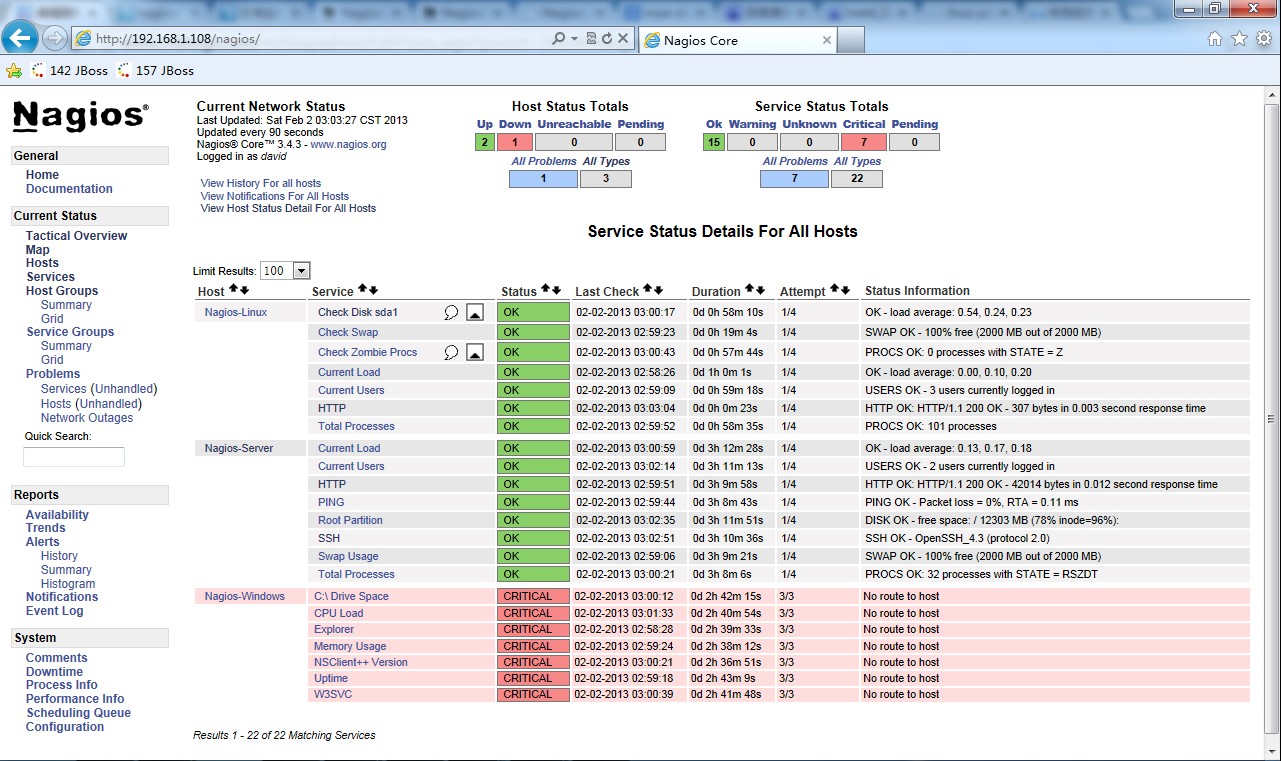
}

所有的配置文件已经修改好了，现在重启Nagios。

# service nagios restart

9.3.4 查看配置情况

登录Nagios Web监控页[http://192.168.1.108/nagios/](http://172.16.1.124/nagios/) 查看相关信息。

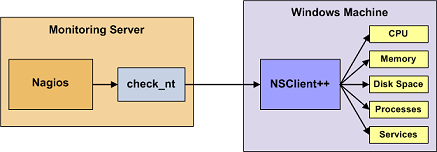


可以看到，对于Nagios-Server 和Nagios-Linux 上的相关服务的监控已经成功了，还有Nagios-Windows 上的服务还没有定义，下面讲到。

十、利用NSClient++监控远程Windows上的“本地信息”

在Nagios的libexec下有check\_nt这个插件，它就是用来检查windows机器的服务的。其功能类似于check\_nrpe。不过还需要搭配另外一个软件NSClient++，它则类似于NRPE。

NSClient++的原理如下图



可以看到NSClient与nrpe最大的区别就是：

被监控机上安装有nrpe，并且还有插件，最终的监控是由这些插件来进行的。当监控主机将监控请求发给nrpe后，nrpe调用插件来完成监控。

NSClient++则不同，被监控机上只安装NSClient，没有任何的插件。当监控主机将监控请求发给NSClient++后，NSClient直接完成监控，所有的监控是由NSClient完成的。

这也说明了NSClient++的一个很大的问题：不灵活、没有可扩展性。它只能完成自己本身包含的监控操作，不能由一些插件来扩展。好在NSClient++已经做的不错了，基本上可以完全满足我们的监控需求。

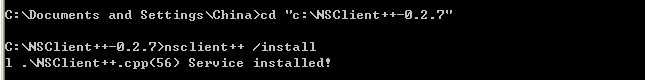
10.1 安装NSClient++

从http://www.nsclient.org/nscp/downloads 下载NSClient++-0.2.7.zip

解压到C盘根目录。

打开cmd 切换到c:\NSClient++-0.2.7

执行nsclient++ /install 进行安装



执行nsclient++ SysTray （注意大小写），这一步是安装系统托盘，时间稍微有点长。

http://images.cnitblog.com/blog/370046/201302/02222739-b0076ad25fbe40d99ff7094f43418657.jpg

在运行里面输入services.msc 打开“服务”



看到下图就说明NSClient服务已经安装上了

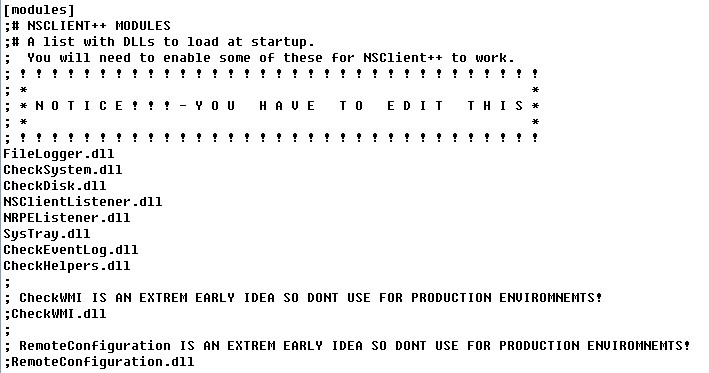
http://images.cnitblog.com/blog/370046/201302/02222842-718e9a7851a348809b4d00244e369544.jpg

双击打开，点“登录”标签，在“允许服务与桌面交互”前打勾。



编辑c:\NSClient++-0.2.7下的NSC.ini文件。

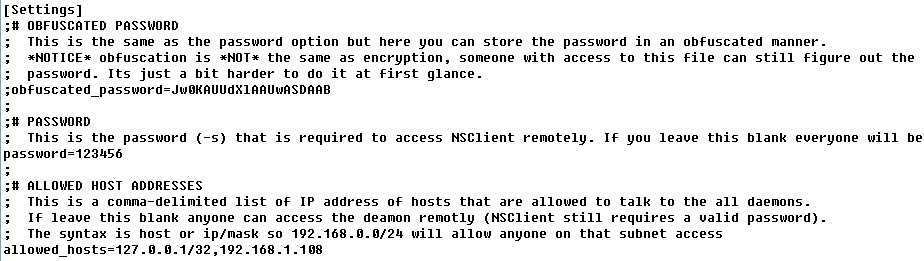
将 [modules]部分的所有模块前面的注释都去掉，除了CheckWMI.dll 和 RemoteConfiguration.dll 这两个。



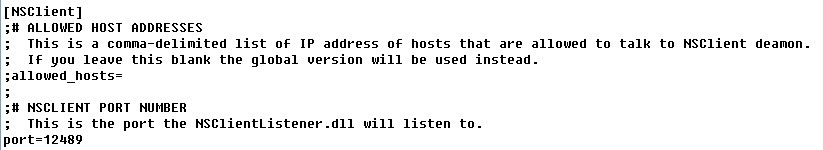
在[Settings]部分设置'password'选项来设置密码，作用是在nagios连接过来时要求提供密码。这一步是可选的，我这里设置为'123456'。

将[Settings]部分'allowed\_hosts'选项的注释去掉，并且加上运行nagios的监控主机的IP。各IP之间以逗号相隔。这个地方是支持子网的，如果写成192.168.1.0/24则表示该子网内的所有机器都可以访问。如果这个地方是空白则表示所有的主机都可以连接上来。

注意是[Settings]部分的，因为[NSClient]部分也有这个选项。



必须保证[NSClient]的'port'选项并没有被注释，并且它的值是'12489'，这是NSClient的默认监听端口。



在cmd 中执行nsclient++ /start启动服务，注意所在目录是c:\NSClient++-0.2.7

http://images.cnitblog.com/blog/370046/201302/02223713-a55666da26bb47338db68d06d04db0ff.jpg

这时在桌面右下角的系统托盘处会出现一个黄色的M字样的图标

http://images.cnitblog.com/blog/370046/201302/02223741-eea5809e0c0a4a75920308e0530a679e.jpg

查看服务

http://images.cnitblog.com/blog/370046/201302/02223800-aa5a5cf0f907417e804e1bf42e3ad95c.jpg

已经正常启动了。

注意服务默认设的是“自动”，也就是说是开机自动启动的。

在cmd 里面执行netstat –an 可以看到已经开始监听tcp的12489端口了。

http://images.cnitblog.com/blog/370046/201302/02223852-f70c360e03204553a28086040839174d.jpg

这样外部就可以访问了吗？

错！

防火墙也要打开tcp的12489端口，否则nagios 检查此服务的时候会报错。

这样被监控机的配置就搞定了，它就等待nagios 发出某个监控请求，然后它执行请求将监控的结果发回到nagios监控主机上。

之前已经在监控主机（Nagios-Server）上对Windows 主机的监控做了配置，但是commands.cfg 中默认没有设置密码项，所以要修改一下，增加"-s 123456"，如下：

# 'check\_nt' command definition

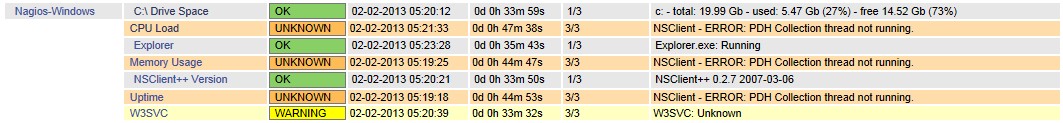
define command{

command\_name check\_nt

command\_line $USER1$/check\_nt -H $HOSTADDRESS$ -p 12489 -s 123456 -v $ARG1$ $ARG2$

}

现在打开Nagios Web监控页便可查看到相关信息了。



可以看到有错误：NSClient - ERROR: PDH Collection thread not running.

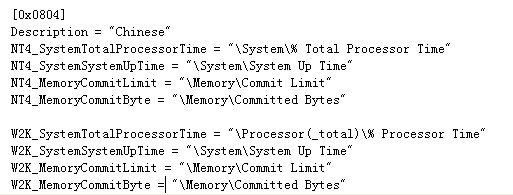
Google 一下，是由于操作系统语言的问题，好像NSClient 默认支持的语言并不多，具体可以百度一下。

查看NSClient的日志C:\NSClient++-0.2.7\nsclient.log，信息如下：

2013-02-02 22:05:30: error:.\PDHCollector.cpp:98: You need to manually configure performance counters!

需要手动配置performance counters。

打开C:\NSClient++-0.2.7\counters.defs文件，复制文件里面"English US"那部分内容，粘贴到counters.defs 文件的最后，修改Description = "Chinese"。



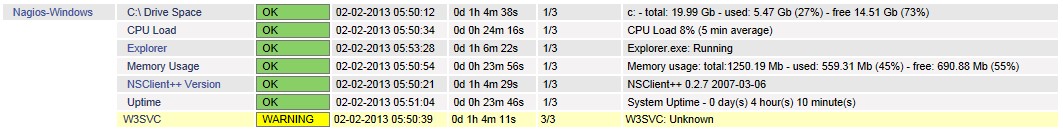
修改完之后，在mmc中重启NSClient 服务。

然后查看日志，内容如下：

http://images.cnitblog.com/blog/370046/201302/03004023-10d2dacf5c6a4d57bbb1c723972a67bd.jpg

在正常执行了。

打开Nagios Web监控页查看。

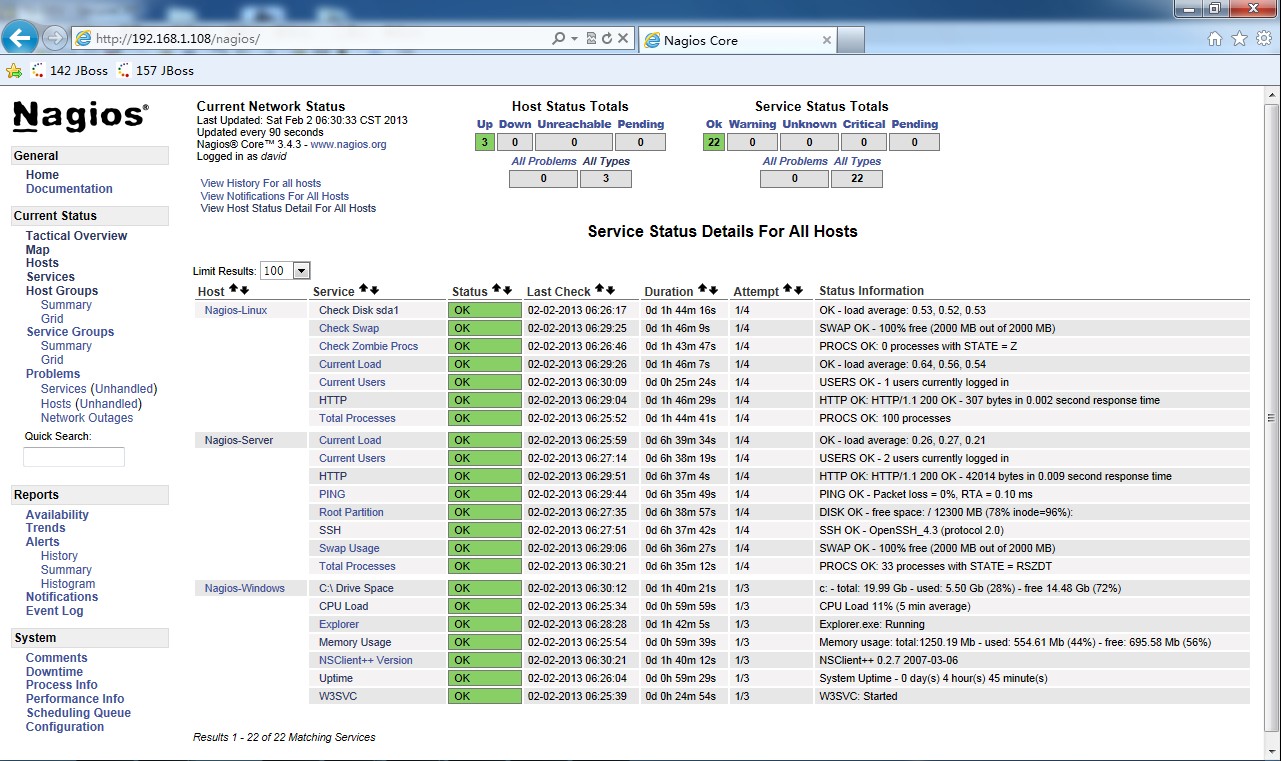


执行成功，但是W3SVC服务为Unknown 状态。查资料，需要开户Windows 的IIS服务。

打开“控制面板”进行安装。



安装完毕后，再到Nagios Web监控页查看，全部监控正常。



十一、Nagios邮件报警的配置

11.1 安装sendmail 组件

首先要确保sendmail 相关组件的完整安装，我们可以使用如下的命令来完成sendmail 的安装：

# yum install -y sendmail\*

然后重新启动sendmail服务：

# service sendmail restart

然后发送测试邮件，验证sendmail的可用性：

# echo "Hello World" | mail david.tang@bsmart.cn

11.2 邮件报警的配置

在上面我们已经简单配置过了/usr/local/nagios/etc/objects/contacts.cfg 文件，Nagios 会将报警邮件发送到配置文件里的E-mail 地址。

11.3 Nagios 通知

PROBLEM



RECOVERY



Linux下Nagios安装配置完毕。

**参考资料**

* Nagios官方网站：<http://www.nagios.org/>
* yahoon的小屋 《nagios全攻略》：http://yahoon.blog.51cto.com/
* 技术成就梦想 《运维监控利器Nagios》：http://ixdba.blog.51cto.com/

## cacti

## ganglia

# 自动化管理

## puppet

参考资料

<https://puppetlabs.com/>

<http://docs.puppetlabs.com/>

<http://dongxicheng.org/cluster-managemant/puppet/>

<http://puppet.wikidot.com/>

Puppet: 强大的中心化配置管理系统

不管是管理1台，或者是上万台机器，puppet都能轻松搞定。让你的所有配置工作整齐划一。所谓配置管理系统，就是管理机器里面诸如文件，用户，进程，软件包这些资源。传统的方法要管理1千台相同的机器，可以选择cfengine，或者是for循环，其实for循环是用的最多的方法。

用puppet，可以运行一个服务器端，然后每个客户端通过ssl证书连接服务器，得到本机器的配置列表，然后更加列表的来完成配置工作。

所以如果硬件配置好了，你在一天之内配置好上万台机器是很轻松的事情。前提是大部分机器配置类似。google用puppet管理了6000多台mac桌面机器。

这个是puppet的官方主页：<http://reductivelabs.com/projects/puppet>

个人用户也很有用，可以把你常用的配置记录成puppet格式的配置文件，下次重装系统，只需要运行这个配置就能配置到你指定的状态。比如我自己的机器，就是所有的软件包和各种各样的配置全都搞好了。

在 CentOS 6.2 上安装 Puppet 配置管理工具

Puppet 是一个客户端/服务器（C/S）架构的配置管理工具，在中央服务器上安装 puppet-server 服务器（puppet master），在需要被管理的目标服务器上安装 puppet 客户端软件（puppet client）。当客户端连接上服务器后，定义在服务器上的配置文件会被编译，然后在客户端上运行。客户端每隔半小时主动会和服务器通信一次，确认配置信息的更新情况，如果有新的配置信息（或者配置有变化），配置文件将会被重新编译并分发到客户端执行。当然，也可以在服务器上主动触发更新指令来强制各客户端进行配置更新。

以下安装采用两台服务器，一台是 master.vpsee.com 用来安装 puppet-server 服务；一台是 client.vpsee.com 用来安装 puppet 客户端。

Puppet 要求所有机器有完整的域名（FQDN），如果没有 DNS 服务器提供域名的话，可以在两台机器上设置主机名（注意要先设置主机名再安装 Puppet，因为安装 Puppet 时会把主机名写入证书，客户端和服务端通信需要这个证书）：

# vi /etc/hosts

192.168.2.10 master master.vpsee.com

192.168.2.11 client client.vpsee.com

Puppet 要求所有机器上的时钟保持同步，所以需要安装和启用 ntp 服务（如果采用 CentOS-6.2-x86\_64-minimal.iso 最小化安装，需要额外安装这个软件包）。

# yum install ntp

# chkconfig ntpd on

# ntpdate pool.ntp.org

29 Feb 15:22:47 ntpdate[15867]: step time server 196.25.1.1 offset 98.750417 sec

# service ntpd start

Starting ntpd: [ OK ]

安装 puppet 服务

Puppet 需要 Ruby 的支持，如果要查看命令行帮助的话需要额外 ruby-rdoc 这个软件包：

# yum install ruby ruby-libs ruby-rdoc

Puppet 不在 CentOS 的基本源中，需要加入 PuppetLabs 提供的官方源：

# yum -y install wget

# wget http://yum.puppetlabs.com/el/6/products/x86\_64/puppetlabs-release-6-1.noarch.rpm

# yum install puppetlabs-release-6-1.noarch.rpm

# yum update

在 master 上安装和启用 puppet 服务：

# yum install puppet-server

# chkconfig puppet on

# service puppetmaster start

Starting puppetmaster: [ OK ]

关闭 iptables：

# /etc/init.d/iptables stop

iptables: Flushing firewall rules: [ OK ]

iptables: Setting chains to policy ACCEPT: filter [ OK ]

iptables: Unloading modules: [ OK ]

安装 puppet 客户端

在 client 上安装 puppet 客户端：

# yum install puppet

Puppet 客户端使用 HTTPS 和服务端（master）通信，为了和服务器端通信必须有合法的 SSL 认证，第一次运行 puppet 客户端的时候会生成一个 SSL 证书并指定发给 Puppet 服务端。

# puppet agent --no-daemonize --onetime --verbose --debug --server=master.vpsee.com

Puppet 服务端接受到客户端的证书后必须签字（sign）才能允许客户端接入，sign 后用 puppet cert list –all 查看会发现 client.vpsee.com 前面多了一个 + 后，表示 “加入” 成功：

# puppet cert list --all

client.vpsee.com (65:3C:20:82:AE:F6:23:A8:0A:0B:09:EF:05:64:1D:BB)

+ master.vpsee.com (AF:A0:32:78:D4:EB:D3:EE:02:1C:62:1C:83:3C:46:EC) (alt names: DNS:master, DNS:master.vpsee.com)

# puppet cert --sign client.vpsee.com

notice: Signed certificate request for client.vpsee.com

notice: Removing file Puppet::SSL::CertificateRequest client.vpsee.com at '/var/lib/puppet/ssl/ca/requests/client.vpsee.com.pem'

# puppet cert list --all

+ client.vpsee.com (65:3C:20:82:AE:F6:23:A8:0A:0B:09:EF:05:64:1D:BB)

+ master.vpsee.com (AF:A0:32:78:D4:EB:D3:EE:02:1C:62:1C:83:3C:46:EC) (alt names: DNS:master, DNS:master.vpsee.com)

这样，客户端和服务端就配置好了，双方可以通信了。

Hello, world

现在可以在服务端写个小例子来测试一下。这个例子作用很简单，用来在客户端的 /tmp 目录下新建一个 helloworld.txt 文件，内容为 hello, world. 在服务端编写代码：

# vi /etc/puppet/manifests/site.pp

node default {

file {

"/tmp/helloworld.txt": content => "hello, world";

}

}

在客户端上执行 puppet，运行成功后会在 /tmp 看到新生成的 helloworld.txt：

$ puppet agent --test --server=master.vpsee.com

warning: peer certificate won't be verified in this SSL session

info: Caching certificate for client.vpsee.com

info: Caching certificate\_revocation\_list for ca

info: Caching catalog for client.vpsee.com

info: Applying configuration version '1330668451'

notice: /Stage[main]//Node[default]/File[/tmp/helloworld.txt]/ensure: defined content as '{md5}e4d7f1b4ed2e42d15898f4b27b019da4'

info: Creating state file /home/vpsee/.puppet/var/state/state.yaml

notice: Finished catalog run in 0.03 seconds

$ cat /tmp/helloworld.txt

hello, world

如何使用 Example42 的 Puppet 模块

现在我们差不多移植了100多台服务器到虚拟机，中间合并了很多功能类似的服务器（历史遗留问题）到同一虚拟机，并且整个配置过程完全用 Puppet 代码化，也就是说有一天我们实验室机房遭遇物理毁灭（刚看完 [Battleship](http://www.imdb.com/title/tt1440129/)，有点激动～），只要再给我们合适的硬件设备（异地有备份数据）我们应该能在6小时内恢复整个基础设施（OS, DNS, DHCP, Provision, Firewall, Squid, Web, Database, LDAP, Email, VPN, Kerberos, NFS, HPC, Cloud, Monitor, Backup, …）。自从 Puppet/Chef 之类的自动化配置工具流行以后，现在甚至有了一个新职位叫做 DevOps.

[安装好 Puppet 后](http://www.vpsee.com/2012/03/install-puppet-on-centos-6-2/)，就可以开始自己写配置代码了，当然也可以在网上找到别人的配置代码直接拿过来用，更妙的是一些通用的 Puppet 配置代码，比如配置一台 Nginx/PHP/MySQL、配置 NFS、配置 DHCP、配置 DNS、配置 OpenNebula 等等已经有人写成模块了，直接拷过来就可以用了。Example42 就是这样的开源 Puppet 模块大集合，包含了众多常用的服务器配置。

首先去 Example42 下载 Puppet 模块代码：

$ git clone http://github.com/example42/puppet-modules.git

看看 Example42 包含了哪些模块吧，几乎常用的都有，不用自己亲自去写配置：

$ cd puppet-modules/

$ ls

DOCS example42 mysql puppi synbak

Example42-tools exim nagios rails sysctl

LICENSE firewall network redis sysklogd

README.rdoc foo nfs repo syslog-ng

Rakefile foreman nginx resolver tftp

activemq git nrpe rootmail timezone

apache haproxy ntp rpmbuild tomcat

apt hardening openldap rsync trac

autofs hosts openntpd rsyslog users

backup iptables openssh samba vagrant

bind jboss openvpn sarg varnish

clamav jenkins oracle selinux virtualbox

cobbler lighttpd pam sendmail vmware

collectd link php snmpd vsftpd

common logrotate phpsyslogng spamassassin wordpress

concat lsb portmap splunk xinetd

controltier mailscanner postfix sqlgrey yum

cron mailx postgresql squid zip

dashboard mcollective powerdns squirrelmail

dhcpd monit psad ssh

dovecot monitor psick ssmtp

drupal munin puppet stdlib42

把上面的 Example42 代码移到 /etc/puppet 目录，然后修改 Puppet 服务器的配置，加上 puppet-modules 模块路径，别忘了重启 puppetmaster 服务：

# mv puppet-modules /etc/puppet

# vi /etc/puppet/puppet.conf

...

[master]

modulepath = /etc/puppet/puppet-modules

# /etc/init.d/puppetmaster

写个配置文件测试一下，自动配置 PHP + Apache + MySQL：

# vi /etc/puppet/manifests/node.pp

node 'web.vpsee.com' {

include apache

include php

include php::pear

include php::apc

php::module { mysql: }

php::module { curl: }

php::module { gd: }

php::module { idn: }

php::module { imagick: }

php::module { imap: }

php::module { mcrypt: }

php::module { ming: }

php::module { ps: }

php::module { pspell: }

php::module { recode: }

php::module { snmp: }

php::module { tidy: }

php::module { xmlrpc: }

php::module { xsl: }

php::module { ldap: }

include mysql

}

在要配置的机器上执行下面一条命令就自动装上 Apache/PHP/MySQL 了，再不用自己手动去安装配置了：

# puppet agent --test --server=master.vpsee.com

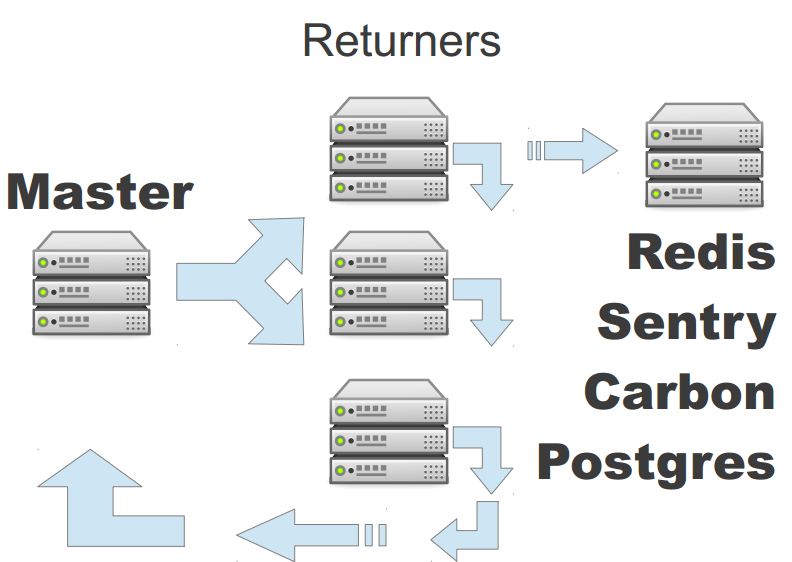
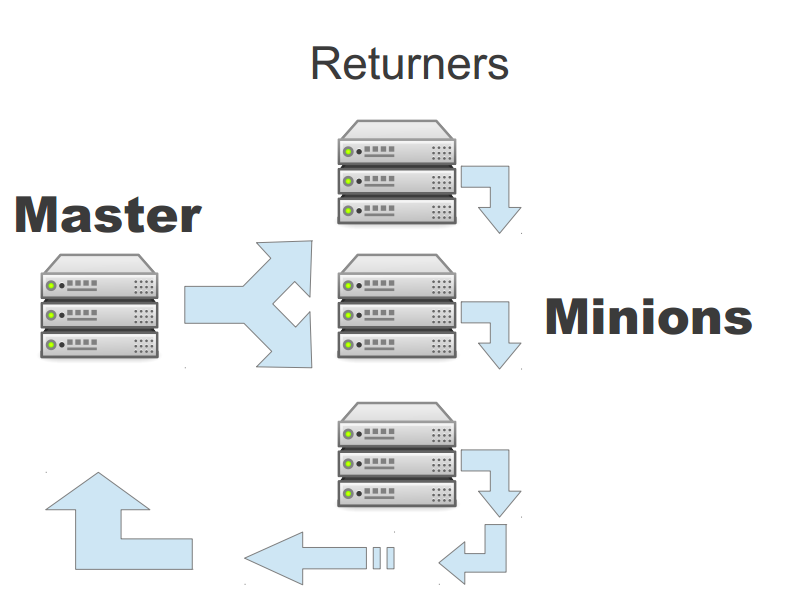
## chef

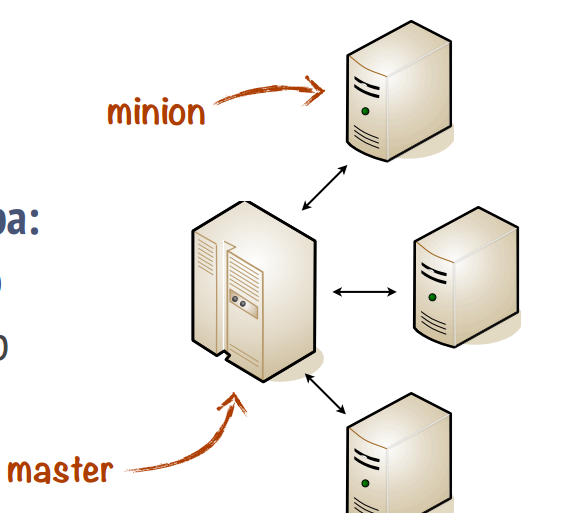
<http://www.opscode.com/chef/>

## Cobller

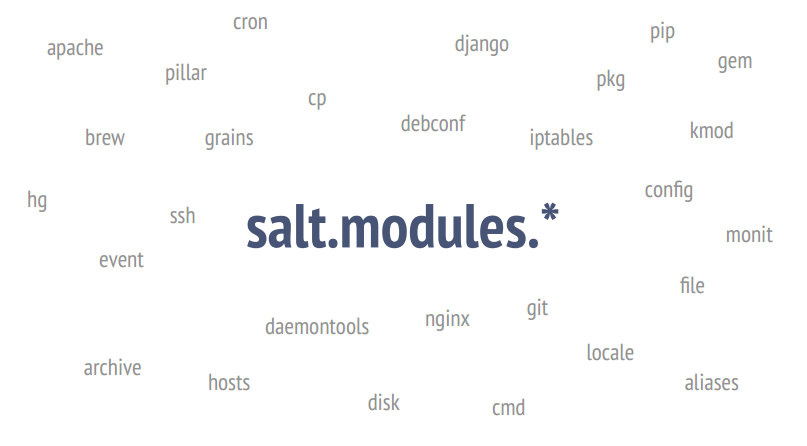
## Saltstack

### Salt的架构





提供丰富的模块



### 基础概念

|  |  |
| --- | --- |
| 名称 | 功能 |
| Master | 控制中心，salt命令运行和资源状态管理端 |
| Minions | 需要管理的客户端机器，会主动去连接Master端，并从Master端得到资源状态信息，同步资源管理信息 |
| States | 配置管理的指令集 |
| Modules | 包含命令行下运行的指令，和在配置文件里面使用的指令  模块可以的函数可以在命令行下运行 |
| Grains | minion端的变量，静态的 |
| pillar | minion端的变量，动态的，比较私密的变量，可以通过配置文件实现不同minions自定义 |
| highstate | 给minion永久下添加状态，从sls配置文件读取到的，即同步状态配置 |
| salt schedule | 自动保持客户端配置 |

### configuration rpm environment

安装epel源，

rpm -ivh http://mirrors.sohu.com/fedora-epel/6/x86\_64/epel-release-6-8.noarch.rpm

epel中关于salt的包:

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

salt-api.noarch : A web api for to access salt the parallel remote execution system

salt-master.noarch : Management component for salt, a parallel remote execution system

salt-minion.noarch : Client component for salt, a parallel remote execution system

salt.noarch : A parallel remote execution system

salt-cloud.noarch : Generic cloud provisioning tool

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

除了rpm安装，还可以用源码安装方法

具体方法参考官方文档<http://docs.saltstack.com/topics/installation/index.html>

hosts设置

server和client的/etc/hosts配置如下

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

192.168.1.253 test.itnihao.com

192.168.1.254 salt.itnihao.com

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

需要注意的是，client必须可以解析服务端的dns，即主机名

### master install

yum install salt-master

vim /etc/salt/master

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

interface: 服务端监听IP #非必须修改，默认是0.0.0.0

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

启动服务

/etc/init.d/salt-master start

调试模式:

salt-master -l debug

后台运行:

salt-master -d

监听端口

4505(publish\_port):salt的消息发布系统

4506(ret\_port):salt客户端与服务端通信的端口

要保证这2个端口能通信正常

如果开启iptables 需要添加以下2条规则

-A INPUT -m state --state new -m tcp -p tcp --dport 4505 -j ACCEPT

-A INPUT -m state --state new -m tcp -p tcp --dport 4506 -j ACCEPT

### minion install

#yum install salt-minion

#vim /etc/salt/minion

master: 服务端主机名

id: 客户端主机名(在服务端看到的客户端的名字)

启动服务

#/etc/init.d/salt-minion start

### key的颁发

由于salt的master和minions是通过证书通信的，故存在证书的信任颁发问题。

#salt-key -L

Accepted Keys:

Unaccepted Keys:

test.itnihao.com

Rejected Keys:

#salt-key -a test.itnihao.com

Accepted Keys:

test.itnihao.com

Unaccepted Keys:

Rejected Keys:

当运行salt-key -A时候，会接受所有客户端请求的证书

### 5.运行测试

# salt "test.itnihao.com" test.ping

test.itnihao.com:

True

到以上步骤，说明服务端到客户端通信正常，基础环境搭建成功。

那么接下来，应该是配置个性化的服务。例如软件包的安装，用户的管理，文件的同步等，这些需要对配置文件进行修改才能实现：

### 5.1 入口文件top.sls

top.sls文件的作用：**确定了minions属于哪个环境，有哪些minions，可以加载哪些sls状态配置。**

运行环境是由file\_roots配置决定的，在，master配置文件上面/etc/salt/master

**file\_roots:**

**base:**

**- /srv/salt**

**dev:**

**- /srv/salt/dev**

**prod:**

**- /srv/salt/prod**

这里定义了三个环境base，dev，prod，需要注意的是默认只有base环境，如果自定义环境，则按照上面的格式修改/etc/salt/master配置文件。

那么在top.sls文件中，可以对不同的minions定义不同的环境

#mkdir /srv/salt/ #由于salt安装好之后是没有/srv/salt目录的，所以建立此文件夹

#vim /srv/salt/top.sls

base:

'test.itnihao.com':

- git

- myapp

- user

base是环境

test.itnihao.com是minions

git myapp user 是3个sls模块文件，也就是当前目录下面应该存在git.sls myapp.sls user.sls三个状态配置文件，文件内容继续看下文

### 5.2 同步文件的例子

#vim /srv/salt/myapp.sls

/etc/myapp.conf: # ID declaration 注意这里的名字将和实际位置有关

file.managed: # state declaration 申明哪一种state被运用

- source: salt://files/myapp.conf # function declaration 哪个function将会被file.managed调用

- mode: 644

- owner: root

- group: root

- template: jinja

注意，这里的salt://files/myapp.conf是相对于/srv/salt/目录的路径，也就是说，salt的根路径是/srv/salt,而用salt://files/myapp.conf访问文件，在master上面的路径就是/srv/salt/files/myapp.conf，状态同步文件里面写salt://files/myapp.conf，这点和apache中的文件路径是一样的。（由于之前有人看不明白，所以啰嗦了，见谅）。

#vim /srv/salt/files/myapp.conf

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

A test file

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

### 5.3 用户管理

#vim /srv/salt/user.sls

admin:

user.present:

- fullname: administrator

- password: '$6$Lr.pTdgs7xxjXH7C$p9rm/IGEiBNFGOx8tCgajXsU74OBvjcgLmO4VNlTdBdnHMvaTQVshBiQXwM8Ua4961oLm/3wre97L7.9dTWJs.'

- shell: /bin/bash

- home: /home/adminsvn

注意这里的密码是hash过的，可以用grub-crypt 工具生成hash密码

### 5.4推送配置到minion客户端

#salt 'test.itnihao.com' state.highstate -v

当执行以上命令的时候，就把状态配置，同步到minions了

### 5.5客户端minion同步服务端

#salt-call state.highstate

当然，从minions也可以去主动同步master状态配置，使用salt-call即可

### 5.6变量

在salt中，有2个组件提供变量的功能，一个是grains，一个是pillar

|  |  |  |
| --- | --- | --- |
| Grains | minion端的变量，静态的 | salt "\*" grains.items  salt "\*" grains.item os |
| pillar | minion端的变量，动态的，比较私密的变量，可以通过配置文件实现不同minions自定义 | salt "\*" pillar.data |

关于这2个组件提供的功能，后面会详细介绍

### 5.7 关于salt执行命令带的参数说明

targeting

salt '\*' test.ping

引号中以实现很强大的minion的过滤与匹配技术

文档:http://docs.saltstack.com/topics/targeting/compound.html

salt 'shell正则' 命令

salt -E 'prel 正则'

salt -N $group 命令

salt -L 'server\_id1,server\_id2,server\_id3' 命令

G Grains glob match G@os:Ubuntu

E PCRE Minion id match E@web\d+\.(dev|qa|prod)\.loc

P Grains PCRE match P@os:(RedHat|Fedora|CentOS)

L List of minions L@minion1.example.com,minion3.domain.com or bl\*.domain.com

I Pillar glob match I@pdata:foobar

S Subnet/IP addr match S@192.168.1.0/24 or S@192.168.1.100

R Range cluster match R@%foo.bar

D Minion Data match D@key:value

示例:

salt -C 'webserv\* and G@os:Debian or E@web-dc1-srv.\*' test.ping

salt '\*' test.ping

Match all minions in the example.net domain or any of the example domains:

salt '\*.example.net' test.ping

salt '\*.example.\*' test.ping

Match all the webN minions in the example.net domain (web1.example.net, web2.example.net . . .

webN.example.net):

salt 'web?.example.net' test.ping

Match the web1 through web5 minions:

salt 'web[1-5]' test.ping

Match the web-x, web-y, and web-z minions:

salt 'web-[x-z]' test.ping

salt -E 'web1-(prod|devel)' test.ping

salt -L 'web1,web2,web3' test.ping

top.sls

base:

'web1-(prod|devel)':

- match: pcre

- webserver

### 6.分组

### nodegroup

对minion进行分组

文档: http://docs.saltstack.com/topics/targeting/nodegroups.html

在master的配置文件中按如下格式定义（/etc/salt/master）:

nodegroups:

group1: 'L@foo.domain.com,bar.domain.com,baz.domain.com or bl\*.domain.com'

group2: 'G@os:Debian and foo.domain.com'

在state或者pillar中引用的时候如下:

base:

group1:

- match: nodegroup

- webserver

### 7：minion

即为salt的客户端

### 8:状态管理

### [Salt States概览](http://www.ituring.com.cn/article/42238)

这部分的文档来自<http://www.ituring.com.cn/article/42238>，请参考原文

Salt States

参考链接：[官方文档http://docs.saltstack.com/topics/tutorials/starting\_states.html](http://docs.saltstack.com/topics/tutorials/starting_states.html)

简洁，简洁，简洁

众多强大而有力的设计都建立在简单的原则之上。Salt SLS系统也努力向K.I.S.S看齐。（Keep It Stupidly Simple）

SLS（代表SaLt State文件）是Salt State系统的核心。SLS描述了系统的目标状态，由格式简单的数据构成。这经常被称作配置管理。

Note

这篇文章从整体上介绍了Salt States，以后还会增加对各组件的深入介绍。

只是数据而已

深入学习之前，明白SLS文件只是结构化的数据而已是很有用的。看懂和编写SLS文件不需要理解这一点，但会让你体会到SLS系统的强大。

SLS文件本质上只是一些dictionaries，lists，strings和numbers。这种设计让SLS文件非常灵活，可以满足开发者的各种需求，而且可读性很高。写得越多，就越清楚到底写得是什么。

默认的数据 - YAML

Salt默认使用能找到的最简单的序列化数据格式——YAML，来表达SLS数据。典型的SLS文件如下：

apache:

pkg:

- installed

service:

- running

- require:

- pkg: apache

这些数据确保名为apache的软件包处于已安装状态（如果不是，那么就安装apache），服务进程apache处于运行状态。这些数据简洁，易于理解。下面简单解释一下：  
第1行是这段数据的ID，被称作ID声明。这个ID是将要执行的这些命令的名字。  
第2行和第4行表示State声明开始，使用了pkg和service这两个states。pkg使用系统本地的软件包管理器管理将要安装的软件，service管理系统守护进程。  
第3行和第5行是要执行的function。这些function定义了名字为ID的软件包和服务的目标状态。此例中，软件包应当处于已安装状态，服务必须运行。  
最后，第6行是关键字require。这被称为必要语句（Requisite），它确保了apache服务只有在成功安装软件包后才会启动。

添加配置文件和用户

部署像apache这样的web服务器时，还需要添加其他的内容。需要管理apache的配置文件，需要添加运行apache服务的用户和组。

apache:

pkg:

- installed

service:

- running

- watch:

- pkg: apache

- file: /etc/httpd/conf/httpd.conf

- user: apache

user.present:

- uid: 87

- gid: 87

- home: /var/www/html

- shell: /bin/nologin

- require:

- group: apache

group.present:

- gid: 87

- require:

- pkg: apache

/etc/httpd/conf/httpd.conf:

file.managed:

- source: salt://apache/httpd.conf

- user: root

- group: root

- mode: 644

这个SLS大大扩展了上面的例子，增加了配置、用户、组，还有一个新的必要语句：watch。

user和group这两个state添加在apache这个ID下，所以增加的user和group名字都是apache。require语句确保了只有在apache这个group存在时才建立user，只有在apache这个package成功安装后才会建立group。

service中的require语句换成了watch，从需要1个软件包改为监视3个state（分别是pkg、file和user）。watch语句和require很相似，都能保证被监视或需要的state在自己之前执行，但是watch还有其他作用。在被监视的state发生变化时，定义watch语句的state会执行自己的watcher函数。也就是说，更新软件包，修改配置文件，修改apache用户的uid都会触发service state的watcher函数。在这个例子中，service state的watcher会重启apache服务。

Note

Salt的watcher概念非常有意思。Puppet中功能类似的是notify，也可以触发服务重启。Salt的watcher非常灵活，watcher本质上是在state的代码中定义的名为mod\_watch()的函数，在这个函数中想做什么事情完全就看你的需求了。我没有仔细看Puppet的notify如何实现，不知道是否有这么灵活。

多个SLS文件

在更有扩展性的部署Salt State时，需要用到不只一个SLS。上面的例子中只使用1个SLS文件，2个或多个SLS文件可以结合形成State Tree。上面的例子还使用了一个奇怪的文件来源 —salt://apache/httpd.conf，这个文件究竟在什么位置呢？

SLS文件以一定的目录结构分布在master上；SLS和要下发到minion上的文件都只是普通文件。

上面的例子中的文件在Salt的根目录（见《[SaltStack中的文件服务器](http://www.ituring.com.cn/article/41632)》）分布如下：

apache/init.sls

apache/httpd.conf

httpd.conf只是apache目录下的一个普通文件，可以直接引用。 使用多个SLS文件可以更加灵活方便，以SSH为例：

ssh/init.sls:

openssh-client:

pkg.installed

/etc/ssh/ssh\_config:

file.managed:

- user: root

- group: root

- mode: 644

- source: salt://ssh/ssh\_config

- require:

- pkg: openssh-client

ssh/server.sls:

include:

- ssh

openssh-server:

pkg.installed

sshd:

service.running:

- require:

- pkg: openssh-client

- pkg: openssh-server

- file: /etc/ssh/banner

- file: /etc/ssh/sshd\_config

/etc/ssh/sshd\_config:

file.managed:

- user: root

- group: root

- mode: 644

- source: salt://ssh/sshd\_config

- require:

- pkg: openssh-server

/etc/ssh/banner:

file:

- managed

- user: root

- group: root

- mode: 644

- source: salt://ssh/banner

- require:

- pkg: openssh-server

Note

在ssh/server.sls中，用了两种不同的方式来表示用Salt管理一个文件。在ID为/etc/ssh/sshd\_config段中，直接使用file.managed作为state声明，而在ID为/etc/ssh/banner段中，使用file作为state声明，附加一个managed属性。两种表示方法的含义与结果完全一样，只是写法不同。

现在State Tree如下（有些被引用的文件没有给出内容，不影响理解）：

apache/init.sls

apache/httpd.conf

ssh/init.sls

ssh/server.sls

ssh/banner

ssh/ssh\_config

ssh/sshd\_config

ssh/server.sls中使用了include语句。include将别的SLS添加到当前文件中，所以可以require或watch被引用的SLS中定义的内容，还可以extend其内容（马上讲到）。include语句使得state可以跨文件引用。使用include相当于把被引用的内容文件添加到自身。

扩展被引用的SLS数据 Extend

扩展是什么意思呢？比如在ssh/server.sls中定义了一个apache通用的服务器，现在要增加一个带mod\_python模块的apache，不需要重头写新的SLS，直接include原来的server.sls，然后增加安装mode\_python的state，再在apache service的watch列表中增加mod\_python即可。python/mod\_python.sls内容如下：

include:

- apache

extend:

apache:

service:

- watch:

- pkg: mod\_python

mod\_python:

pkg.installed

这个例子中，先将apache目录下的init.sls文件包含进来（在include一个目录时，Salt会自动查找init.sls文件），然后扩展了ID为apache下的service state中的watch列表。

也可以在Extending中修改文件的下载位置。ssh/custom-server.sls:

include:

- ssh.server

extend:

/etc/ssh/banner:

file:

- source: salt://ssh/custom-banner

Extend使得Salt的SLS更加灵活。为什么SLS能够做Extend呢？文章一开始最强调了，SLS中的文件仅仅是结构化的data而已，在处理SLS时，会将其中的内容解析成Python中的dict（当然这个dict中会嵌套dict和list）。修改apache watch的内容，相当于往list里面添加一个元素；修改banner文件的下载路径相当于修改dict中的某个key对应的值。在extending时，会附加加require/watch的内容，而不是覆盖。

理解渲染系统 Render System

因为SLS仅仅是data，所以不是非得用YAML来表达。Salt默认使用YAML，只是因为易学易用。只要有对应的renderer，SLS文件可以用任何方式表达（Salt关心的是最终解析出来的数据结构，只要你的renderer能够按要求返回这个数据结构，Salt干嘛关心你如何书写源文件呢？）。

Salt默认使用yaml\_jinja渲染器。yaml\_jinjia先用jinja2模板引擎处理SLS，然后再调用YAML解析器。这种设计的好处是，可以在SLS文件使用所有的编程结构（jinja2能怎么用，这里就能怎么用。条件，循环，Python代码，什么都可以）。

其他可用的渲染器还包括：yaml\_mako，使用Mako模板引擎；yaml\_wempy，使用Wempy模板引擎；py，直接使用Python写SLS文件；pydsl，建立在Python语法基础上的描述语言。

简单介绍默认的渲染器 —— yaml\_jinja

关于jinja模板引擎的使用请参考其[官方文档](http://jinja.pocoo.org/docs)

在基于模板引擎的渲染器里，可以从3个组件中获取需要的数据：salt，grains和pilla。在模板文件中，可以用salt对象执行任意的Salt function，使用grains访问Grains数据。示例如下：  
apache/init.sls:

apache:

pkg.installed:

{% if grains['os'] == 'RedHat'%}

- name: httpd

{% endif %}

service.running:

{% if grains['os'] == 'RedHat'%}

- name: httpd

{% endif %}

- watch:

- pkg: apache

- file: /etc/httpd/conf/httpd.conf

- user: apache

user.present:

- uid: 87

- gid: 87

- home: /var/www/html

- shell: /bin/nologin

- require:

- group: apache

group.present:

- gid: 87

- require:

- pkg: apache

/etc/httpd/conf/httpd.conf:

file.managed:

- source: salt://apache/httpd.conf

- user: root

- group: root

- mode: 644

这个例子很容易理解，用到了jinja中的条件结构，如果grains中的os表明minion的操作系统是Red Hat，那么Apache的软件包名和服务名应当是httpd。

再来一个更NB的例子，用到了jinja的循环结构，在设置MooseFs分布式chunkserver的模块中：  
moosefs/chunk.sls:

include: - moosefs

{% for mnt in salt['cmd.run']('ls /dev/data/moose\*').split() %} /mnt/moose{{ mnt[-1] }}: mount.mounted: - device: {{ mnt }} - fstype: xfs

- mkmnt: True file.directory: - user: mfs

- group: mfs

- require: - user: mfs

- group: mfs

{% endfor %}

'/etc/mfshdd.cfg': file.managed: - source: salt://moosefs/mfshdd.cfg - user: root

- group: root

- mode: 644 - template: jinja

- require: - pkg: mfs-chunkserver

'/etc/mfschunkserver.cfg': file.managed: - source: salt://moosefs/mfschunkserver.cfg - user: root

- group: root

- mode: 644 - template: jinja

- require: - pkg: mfs-chunkserver

mfs-chunkserver: pkg: - installed

mfschunkserver: service: - running

- require: {% for mnt in salt['cmd.run']('ls /dev/data/moose\*') %} - mount: /mnt/moose{{ mnt[-1] }} - file: /mnt/moose{{ mnt[-1] }} {% endfor %} - file: /etc/mfschunkserver.cfg

- file: /etc/mfshdd.cfg

- file: /var/lib/mfs

这个例子展示了jinja的强大，多个for循环用来动态地检测并挂载磁盘，多次使用salt对象（这里使用了cmd.run这个执行模块）执行shell命令来收集数据。

简单介绍Python和PyDSL渲染器

在任务逻辑非常复杂时，默认的yaml\_jinja渲染器不一定满足要求，这时可以使用Python渲染器。如何在State tree中添加使用py渲染器的SLS文件呢？简单。 一个非常简单的基本Python SLS文件：  
python/django.sls:

#!py

def run(): '''

Install the django package

''' return {'include': ['python'], 'django': {'pkg': ['installed']}}

这个例子也很好理解，第1行告诉Salt不使用默认的渲染器，而是用py。接着定义了函数run，这个函数的返回值必须符合Salt的要求，即HighState数据结构（我接下来就写关于HighState的文章，现在不必关心其细节，反正就是一个dict，key和value都有规定好的含义）。 如果换用pydsl渲染器，上面的例子会更简洁：  
python/django.sls:

#!pydsl

include('python', delayed=True)state('django').pkg.installed()

如果用YAML，会是下面这个样子：

include: - python

django: pkg.installed

这也可以看出，正常情况下使用YAML是非常合适的，但如果有需要时，使用纯粹的Python SLS可以非常NB。

运行和调试Salt States

写好的SLS如何才能应用到minion呢？在SaltStack中，远程执行是一切的基础。执行命令salt '\*' state.highstate会让所有的minion到master上来取走自己的SLS定义，然后在本地调用对应的state module（user，pkg，service等）来达到SLS描述的状态。如果这条命令只返回minion的主机名加一个':'，多半是哪一个SLS文件有错。如果minion是以服务进程启动，执行命令salt-call state.highstate -l debug可以看到错误信息，便于调试。minion还可以直接在前台以debug模式运行：salt-minion -l debug。

接下来是什么？

接下来是关于Pillar的内容，官方文档[在此](http://docs.saltstack.com/topics/tutorials/pillar.html)，关于Pillar的文章发表[在此](http://www.ituring.com.cn/article/42398)。

原文地址<http://www.ituring.com.cn/article/42238>

### 1: state

salt基于minion进行状态的管理

1:state语法

文档:http://docs.saltstack.com/ref/states/all/index.html

结构:

$ID: #state的名字

$state: #要管理的模块类型

- $State states #该模块的状态

示例：

vim:

pkg:

{% if grains['os\_family'] == 'RedHat' %}

- name: vim-enhanced

{% elif grains['os'] == 'Debian' %}

- name: vim-nox

{% elif grains['os'] == 'Ubuntu' %}

- name: vim-nox

{% endif %}

- installed

如果是redhard系列的就安装 vim-enhanced，如果系统是Debian或者Ubuntu就安装vim-nox

附：state默认使用jinja（http://jinja.pocoo.org/）的模板语法，

文档地址： <http://jinja.pocoo.org/docs/templates/>

### 2：state的逻辑关系：

文档：http://docs.saltstack.com/ref/states/ordering.html

require：依赖某个state，在运行此state前，先运行依赖的state，依赖可以有多个

httpd:

pkg:

- installed

file.managed:

- name: /etc/httpd/conf/httpd.conf

- source: salt://httpd/httpd.conf

- require:

- pkg: httpd

watch：在某个state变化时运行此模块

redis:

pkg:

- latest

file.managed:

- source: salt://redis/redis.conf

- name: /etc/redis.conf

- require:

- pkg: redis

service.running:

- enable: True

- watch:

- file: /etc/redis.conf

- pkg: redis

附：watch除具备require功能外，还增了关注状态的功能

order：

优先级比require和watch低

有order指定的state比没有order指定的优先级高

vim:

pkg.installed:

- order: 1

想让某个state最后一个运行，可以用last

### 3：state与minion

将临时给minoin加个state

salt 'minion1' state.sls 'vim' #给minion1加一个vim的state

执行该命令后可以立即看到输出结果

### 9 pillar

Pillar是Salt非常重要的一个组件，它用于给特定的minion定义任何你需要的数据，这些数据可以被Salt的其他组件使用。Salt在0.9.8版本中引入了Pillar。Pillar在解析完成后，是一个嵌套的dict结构；最上层的key是minion ID，其value是该minion所拥有的Pillar数据；每一个value也都是key/value。

这里可以看出Pillar的一个特点，Pillar数据是与特定minion关联的，也就是说每一个minion都只能看到自己的数据，所以Pillar可以用来传递敏感数据（在Salt的设计中，Pillar使用独立的加密session，也是为了保证敏感数据的安全性）。

Pillar可以用在哪些地方？

敏感数据

例如ssh key，加密证书等，由于Pillar使用独立的加密session，可以确保这些敏感数据不被其他minion看到。

变量

可以在Pillar中处理平台差异性，比如针对不同的操作系统设置软件包的名字，然后在State中引用。

其他任何数据

可以在Pillar中添加任何需要用到的数据。比如定义用户和UID的对应关系，mnion的角色等。

用在Targetting中

Pillar可以用来选择minion，使用-I选项。

怎样定义Pillar数据？

master配置文件中定义

默认情况下，master配置文件中的所有数据都添加到Pillar中，且对所有minion可用。如果要禁用这一默认值，可以在master配置文件中添加如下数据，重启服务后生效：

pillar\_opts: False

使用SLS文件定义Pillar

Pillar使用与State相似的SLS文件。Pillar文件放在master配置文件中pillar\_roots定义的目录下。示例如下：

pillar\_roots:

base:

- /srv/pillar

这段代码定义了base环境下的Pillar文件保存在/srv/pillar/目录下。与State相似，Pillar也有top file，也使用相同的匹配方式将数据应用到minion上。示例如下：

/srv/pillar/top.sls:

base:

'\*':

- data

- packages

/srv/pillar/packages.sls:

{% if grains['os'] == 'RedHat' %}

apache: httpd

git: git

{% elif grains['os'] == 'Debian' %}

apache: apache2

git: git-core

{% endif %}

/srv/pillar/data/init.sls:

role: DB\_master

这段代码表示，base环境中所有的minion都具有packages和data中定义的数据。Pillar采用与file server相同的文件映射方式，在本例中，packages映射到文件/srv/pillar/packages.sls，

data映射到/srv/pillar/data/init.sls。注意key与value要用冒号加空格分隔，没有空格的话将解析失败。

Pillar还可以使用其他的匹配方式来选择minion，下面的例子中，servers只应用到操作系统是Debain的机器：

dev:

'os:Debian':

- match: grain

- servers

如何知道minion拥有哪些Pillar数据？

使用执行模块pillar。pillar模块有两个funtion：pillar.data和pillar.raw。示例如下：

# salt '\*' pillar.data

在master上修改Pilla文件后，需要用以下命令刷新minion上的数据：

salt '\*' saltutil.refresh\_pillar

如果定义好的pillar不生效，建议刷新一下试试。

Pillar中数据如何使用？

Pillar解析后是dict对象，直接使用Python语法，可以用索引（pillar['pkgs']['apache']）或get方法（pillar.get('users', {})）。详见下面的例子。

示例

targeting

使用-I选项表示使用Pillar来匹配minion.

salt -I 'role:DB\*' test.ping

在Pillar中使用列表

Pillar的key/value结构中的value可以是string，也可以是一个list。Pillar文件定义如下：

/srv/pillar/users/init.sls：

users:

thatch: 1000

shouse: 1001

utahdave: 1002

redbeard: 1003

在top.sls中引用Pillar文件，对所有的minion应用users中的内容：

/srv/pillar/top.sls：

base:

'\*':

- data

- users

现在所有的minion都具有了users数据，可以在state文件中使用：

/srv/salt/users/init.sls：

{% for user, uid in pillar.get('users', {}).items() %}

{{user}}:

user.present:

- uid: {{uid}}

{% endfor %}

利用Pillar处理平台差异

不同的操作系统不仅管理资源的方式不同，软件包的名字、配置文件的路径也有有可能不一样。Salt的执行模块屏蔽了系统管理资源的差异。其他的差异可以根据grains中的的os、cpuarch等

信息来处理，这些条件判断可以写在State文件中，但会使得State文件的逻辑不清晰。Pillar可以很好地解决这个问题。下面的例子中，在不同的os上安装对应的软件包，但state file完全

一样，不需要针对os作修改，灵活方便。

/srv/pillar/pkg/init.sls：

pkgs:

{% if grains['os\_family'] == 'RedHat' %}

apache: httpd

vim: vim-enhanced

{% elif grains['os\_family'] == 'Debian' %}

apache: apache2

vim: vim

{% elif grains['os'] == 'Arch' %}

apache: apache

vim: vim

{% endif %}

/srv/pillar/top.sls：

base:

'\*':

- data

- users

- pkg

/srv/salt/apache/init.sls：

apache:

pkg.installed:

- name: {{ pillar['pkgs']['apache'] }}

还可以在state file中设置默认值： srv/salt/apache/init.sls：

apache:

pkg.installed:

- name: {{ salt['pillar.get']('pkgs:apache', 'httpd') }}

salt敏感信息的管理,只有匹配到的节点才能看到和使用

文档:http://docs.saltstack.com/topics/tutorials/pillar.html

默认:pillar数据定义文件存储路径:/srv/pillar

入口文件:/srv/pillar/top.sls

格式:

base:

"targeting":

- $pillar #名字为pillar.sls的文件来存放对匹配到的minion的变量

$pillar.sls

基本:

$key: $value

state引用方式:

{{ pillar['$key'] }}

复杂:

users:

thatch: 1000

shouse: 1001

utahdave: 1002

redbeard: 1003

state引用方式:

{% for user, uid in pillar.get('users', {}).items() %}

{{user}}:

user.present:

- uid: {{uid}}

{% endfor %}

查看节点的pillar数据：

salt 'client2' pillar.data

同步pillar：

salt '\*' saltutil.refresh\_pillar

附：这里我们可以看到，pallar中也可以使用jinja（后面会提到）做一些处理

一个例子

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

定义pillar

/srv/pillar/top.sls

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

base:

'test.itnihao.com':

- test

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

/srv/pillar/test.sls

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

num1: aa

num2: bb

name: user1

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

/srv/salt/top.sls

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

base:

'test.itnihao.com':

- myapp

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

/srv/salt/myapp.sls

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

/etc/myapp.conf:

file.managed:

- source: salt://files/myapp.conf

- mode: 644

- owner: root

- group: root

- template: jinja

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

/srv/salt/files/myapp.conf

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

test

hostname {{grains['fqdn']}}

{{pillar['num1']}}

{{pillar['num2']}}

{{pillar['name']}}

\*\*\*\*\*\*\*\*\*\*{{pillar['num1']}}\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

/srv/salt/myapp.sls

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

/etc/myapp.conf:

file.managed:

- source: salt://files/myapp.conf

- mode: 644

- owner: root

- group: root

- template: jinja

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

master端运行命令

shell#salt '\*' state.highstate -v

minion端查看

#cat /etc/myapp.conf

test

hostname localhost.localdomain

aa

bb

user1

\*\*\*\*\*\*\*\*\*\*aa\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

### 10.grains

3：grains 第一次启动minion收集的系统信息常量，这意味着此类信息是静态的。

grains接口可以为salt模块和组建提供一些信息，例如，使用适当的salt minion命令可以自动汇报一些系统信息

The grains interface is made available to Salt modules and components so that the right salt minion commands are automatically available on the right systems.

记住重要的一点，grains是在salt minion启动的时候加载的信息，所以这个信息是静态的。也就是意味这grains不能改变。所以，grains比较适合的数据场合如系统内核版本，操作

系统类型等。

minion基本信息的管理

文档:http://docs.saltstack.com/topics/targeting/grains.html

基本使用:

salt '\*' grains.ls 查看grains分类

salt '\*' grains.items 查看grains所有信息

salt '\*' grains.item osrelease 查看grains某个信息

示例：

salt '\*' grains.item osrelease

minoin1:

osrelease: 6.2

在用salt进行管理客户端的时候或者写state的时候都可以引用grains的变量

自己写grains

#mkdir /srv/salt/\_grains && cd /srv/salt/\_grains

/srv/salt/\_grains/mes.py

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

#!/usr/bin/env python

def info():

message={}

message['m1']='my salt grains 1'

message['m2']='my salt grains 2'

return message

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

同步grains到minion

# salt "test.itnihao.com" saltutil.sync\_all

# salt "test.itnihao.com" saltutil.sync\_grains

注意，这里的同步命令有2个

查看自定义的grains

# salt 'test.itnihao.com' grains.item m2

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

test.itnihao.com:

m2: my salt grains 2

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

# salt 'test.itnihao.com' grains.item m1

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

test.itnihao.com:

m1: my salt grains 1

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

### 11:highstate

参考链接：[官方文档](http://docs.saltstack.com/ref/states/highstate.html)

作者言

这篇文档详细解释了SLS文件中，每个部分的名称与含义，以及SLS中的数据处理后的数据结构。

我只是SaltStack的初学者，如果文中有错误的地方，请不吝赐教。  
在学习的过程，我做了一些实验，犯了不少错，积累了一些经验，对SaltStack的运行也有一定了解，如果有什么问题，或是不太理解的地方，非常欢迎留言交流！

Salt State Tree

Top file

Salt State系统的入口文件，其中定义了minion处于哪个环境，加载哪些SLS模块。

State tree

存放在file\_roots目录下的一系列SLS文件。使用SLS模块的形式来组织State tree。

Include声明

一个list，其元素是要引用到本SLS文件的其他SLS模块。 只能用在highstate结构的顶层。

示例：

include: - edit.vim

- http.server

Module引用

SLS模块的名字，以在Salt master上的文件结构命名。名为edit.vim的模块指向salt://edit/vim.sls。

ID声明

定义一个独立的highstate数据段。ID在highstate dict中作为key，其对应的value是包含state声明和requisit声明的另一个dict。  
用在highstate结构的顶层或extend声明的下一层。  
 ID在整个State tree中必须是唯一的。如果同一个ID用了两次，只有最先匹配到的生效，其他所有的同名ID声明被忽略。

Extend声明

扩展被引用的SLS模块中的name声明。extend声明也是一个dict，其key必须是在被引用的SLS模块中定义的ID只能用在highstate结构的顶层。

在需要增加或修改另一个SLS文件中定义的state声明时，Extend声明非常有用。

下面的代码来自mywebsite.sls文件，其中include并且extend了apache.sls模块（增加了apache监视的对象），使得Apache服务在配置文件mywebsite发生改变时自动重启。

include:

- apache

extend:

apache:

service:

- watch:

- file: mywebsite

mywebsite:

file:

- managed

State声明

一个list，至少包含一个定义function声明的string，0个或多个function arg声明的dict。还有一些可选的成员，比如名字覆盖部分（name和names声明），requistie声明。  
只能用在ID声明的下一级。

Requisite声明

一个list，其成员是requisite引用。用来生成动作依赖树。Salt states被设计成按确定的顺序执行，require或watch其他Salt state可以调整执行的顺序。做为list组件用在state声明下一级，或是作为key用在ID声明下一级。

Requisite引用

只有一个key的dict。key是被引用的state声明的名字，value是被引用的ID声明的名字。 只能用作requisite声明的成员。

Function声明

state中要要执行的function。1个state声明中只能有1个function声明。

下面的例子中，state声明调用了state模块pkg模块中的installed功能。

httpd: pkg.installed

可以用行内缩写方式声明function（上面的例子中就是），使用完整写法使得数据结构更清晰：

httpd:

pkg:

- installed

需要注意的是连续的两个简写形式是无效的，为了避免疑惑，建议全部采用完整写法。  
INVALID:

httpd:

pkg:

- installed

service:

- running

只能用作state声明的成员。

Function arg声明

只有1个key的dict，作为参数传递给function声明，其值为有效的Python类型。其类型必须满足function的需要。 用在function声明下一级。

下面的例子中，state声明是file，function声明是managed，user、group和mode是传递给managed的参数：

/etc/http/conf/http.conf:

file.managed:

- user: root

- group: root

- mode: 644

Name声明

覆盖state声明中的name参数。name参数的默认值是ID声明。 name总是1个单key字典，其值类型是string。

在有的场景下，修改默认的name参数非常有用。比如说，可以避免ID冲突。下面例子中的两个state不能同时使用/etc/motd作为ID：

motd\_perms:

file.managed:

- name: /etc/motd

- mode: 644

motd\_quote:

file.append:

- name: /etc/motd

- text: "Of all smells, bread; of all tastes, salt."

另外一个使用name声明的场景是，ID声明非常长，又需要在多次引用这个ID。在下面的例子，使用mywebsite比/etc/apache2/sites-available/mywebsite.com方便多了：

mywebsite:

file.managed:

- name: /etc/apache2/sites-available/mywebsite.com

- source: salt://mywebsite.com

a2ensite mywebsite.com:

cmd.wait:

- unless: test -L /etc/apache2/sites-enabled/mywebsite.com

- watch:

- file: mywebsite

apache2:

service:

- running

- watch:

- file: mywebsite

names声明

将1个state声明扩展为多个不同名的state声明。

看下面的例子：

python-pkgs:

pkg.installed:

- names:

- python-django

- python-crypto

- python-yaml

转换成lowstate后的结果是：

python-django:

pkg.installed

python-crypto:

pkg.installed

python-yaml:

pkg.installed

完整的例子

下面的YAML是一个完整的例子，其中的名字部分使用的是hightstate组件名。

<Include Declaration>:

- <Module Reference>

- <Module Reference>

<Extend Declaration>:

<ID Declaration>:

[<overrides>]

# standard declaration

<ID Declaration>:

<State Declaration>:

- <Function>

- <Function Arg>

- <Function Arg>

- <Function Arg>

- <Name>: <name>

- <Requisite Declaration>:

- <Requisite Reference>

- <Requisite Reference>

# inline function and names

<ID Declaration>:

<State Declaration>.<Function>:

- <Function Arg>

- <Function Arg>

- <Function Arg>

- <Names>:

- <name>

- <name>

- <name>

- <Requisite Declaration>:

- <Requisite Reference>

- <Requisite Reference>

# multiple states for single id

<ID Declaration>:

<State Declaration>:

- <Function>

- <Function Arg>

- <Name>: <name>

- <Requisite Declaration>:

- <Requisite Reference>

<State Declaration>:

- <Function>

- <Function Arg>

- <Names>:

- <name>

- <name>

- <Requisite Declaration>:

- <Requisite Reference>

给minion永久下添加状态

文档： http://docs.saltstack.com/ref/states/highstate.html

默认配置文件：/srv/salt/top.sls

语法：

'\*':

- core

- wsproxy

/srv/salt/目录结构：

.

├── core.sls

├── top.sls

└── wsproxy

├── init.sls

├── websocket.py

└── websockify

应用：

salt "minion1" state.highstate

测试模式：

salt "minion1" state.highstate -v test=True

### 12.salt schedule

默认的state只有在服务端调用的时候才执行，很多时候我们希望minon自觉的去保持在某个状态

文档：http://docs.saltstack.com/topics/jobs/schedule.html

cat /srv/pillar/top.sls

base:

"\*":

- schedule

cat /srv/pillar/schedule.sls

schedule:

highstate:

function: state.highstate

minutes: 30

如上配置：

minion会每30分钟从master拉取一次配置，进行自我配置

### 13:实时管理

有时候我们需要临时的查看一台或多台机器上的某个文件，或者执行某个命令

1:cmd.run

用法 salt '$targeting' cmd.run '$cmd'

示例：salt '\*' cmd.run 'hostname'

执行下这样的命令，马上就感受到效果了，速度还贼快

2:module

同时，salt也将一些常用的命令做了集成

文档：http://docs.saltstack.com/ref/modules/all/index.html

这里几乎涵盖了我们所有的常用命令

比如：

查看所有节点磁盘使用情况

salt '\*' disk.usage

4：无master

文档：http://docs.saltstack.com/topics/tutorials/quickstart.html

主要应该在测试和salt单机管理的时候

### 14关于top文件的书写

查看/etc/salt/master，可以看到默认的top文件位置是/srv/salt/top.sls

#file\_roots:

# base:

# - /srv/salt

由于默认是没有/srv/salt的，所以需要手动建立此文件

mkdir /srv/salt

vim /srv/salt/top.sls

base:

'test.itnihao.com':

- myapp

- httpd

- user

- nginx

语法格式说明

sls文件用的是yaml数据格式，方便易用，书写简单

base是file\_roots里面定义的，可以代表分组

test.itnihao.com是主机名

:后面是资源内容

- 后面必须有空格，后面是模块名称

/srv/salt/myapp.sls

/etc/myapp.conf:

file.managed:

- source: salt://files/myapp.conf

- mode: 644

- owner: root

- group: root

- template: jinja

/srv/salt/httpd.sls

httpd:

pkg:

- installed

service.running:

- enable: True

/srv/salt/user.sls

admin:

user.present:

- fullname: administrator

- password: '$6$Lr.pTdgs7xxjXH7C$p9rm/IGEiBNFGOx8tCgajXsU74OBvjcgLmO4VNlTdBdnHMvaTQVshBiQXwM8Ua4961oLm/3wre97L7.9dTWJs.'

- shell: /bin/bash

- home: /home/adminsvn

- group: nginx

jdoe:

user.absent

nginx:

pkg:

- installed

service:

- running

- enable: True

- require:

- pkg: nginx

- watch:

- file: /etc/nginx/nginx.conf

- file: /etc/nginx/vhost

/etc/nginx/nginx.conf:

file.managed:

- source: salt://nginx/nginx.conf

/etc/nginx/sites.d:

file.recurse:

- source: salt://nginx/vhost

### 15写自己的模块

在master端

关于自定义模块的路径可以为下面2种:

/srv/salt/modules

/srv/salt/\_modules

当路径在此时候，执行state.highstate或者是saltutil.sync\_modules，saltutil.sync\_all函数功能，会自动同步到minions

mkdir /srv/salt/\_modules && cd /srv/salt/\_modules

/srv/salt/\_modules/os\_info.py

#!/usr/bin/evn python

import os

def echo\_os\_type():

'''

echo os type infomation

CLI Example:

slat '\*' os\_info.echo\_os\_type

'''

return os.uname()

#注意里面的注释风格

shell#salt "\*" saltutil.sync\_all

test.itnihao.com:

|\_

- modules.os\_info

|\_

|\_

|\_

|\_

|\_

shell#salt "\*" os\_info.echo\_os\_type

test.itnihao.com:

- Linux

- test.itnihao.com

- 2.6.32-220.23.1.el6.x86\_64

- #1 SMP Mon Jun 18 18:58:52 BST 2012

- x86\_64

前面写了自己的模块，下面再写一个稍微复杂点的模块，写一个tcp连接数的模块

还是在master端写

/srv/salt/\_modules/get\_tcp\_connect.py

#!/usr/bin/env python

import subprocess

def tcpconnect():

'''

get tcp connect status

CLI Example:

salt '\*' get\_tcp\_connect.tcpconnect

'''

tcp\_status=subprocess.Popen("netstat -an|awk '/^tcp/ {++S[$NF]} END {for (a in S) print a,S[a]}'",stdout=subprocess.PIPE,shell=True)

return tcp\_status.stdout.read()

shell#salt "\*" saltutil.sync\_all

shell#salt "\*" sys.doc|grep tcp

get\_tcp\_connect.tcpconnect:

get tcp connect status

salt '\*' get\_tcp\_connect.tcpconnect

shell#salt '\*' get\_tcp\_connect.tcpconnect

本文参考引用以下资料，在此谢谢大家的共享：

http://wiki.saltstack.cn/docs

http://intothesaltmine.org/blog/html/index.html

http://blog.halfss.com/blog/2013/05/22/yun-wei-zi-dong-hua-zhi-saltxue-xi-bi-ji/

http://www.shencan.net/index.php/category/%E8%87%AA%E5%8A%A/saltstack/

### 16 salt文件管理

文件，目录，链接的操作管理

文件管理

/etc/http/conf/http.conf:

file.managed:

- source: salt://apache/http.conf

- user: root

- group: root

- mode: 644

- template: jinja

- context:

custom\_var: "override"

- defaults:

custom\_var: "default value"

other\_var: 123

- require:

- file: /opt/nginx1

- replace: True

/opt/nginx1:

file.recurse:

- source: salt://files/nginx

注意 一下，这里定义了2个变量custom\_var other\_var

如果需要对这2个变量进行引用，则可以在salt://apache/http.conf里面进行变量引用

{{custom\_var}}

{{other\_var}}

context下面可以定义变量内容

defauls 如果context里面没有定义的变量，就会用defauls里面定义的变量

require 是依赖关系，表示在建立此文件前，必须存在的文件，当然这个文件出现在sls文件里面，不是本地文件

关键字

name 文件的名称

source 将从master上面下载的文件，路径为salt://path/file

source\_hash 文件的hash值 例如 /etc/rc.conf md5=ef6e82e4006dee563d98ada2a2a80a27

/etc/resolv.conf sha256=c8525aee419eb649f0233be91c151178b30f0dff8ebbdcc8de71b1d5c8bc06a

user 文件的属主

group 文件的属组

mode 权限

template 从master上面下载的文件模板

makedirs 创建文件夹

replace 是否替换文件，默认是True，如果设置为False，则不会替换同名文件，不改变文件里面的内容

context 定义变量可以在模板中使用的。

defaults 默认的变量在模板中使用

backup 是否备份

### [SaltStack中的文件服务器](http://www.ituring.com.cn/article/41632)

参考链接： 原文[Salt File Server](http://docs.saltstack.com/ref/file_server/index.html), 官网文档[cp模块](http://docs.saltstack.com/ref/modules/all/salt.modules.cp.html#module-salt.modules.cp)

在配置管理系统中，从中心服务器想客户端推送文件是很基本的需求。SaltStack使用内建的ZeroMQ服务器做为文件服务器。 文件服务器主要用来在state系统中推送文件到客户端，也可以用于其他文件的传输。

cp模块

客户端对文件服务器的操作都在cp模块中。Salt State系统，Salt-cp都用到了cp模块。

环境

因为文件服务器是为Salt state系统服务，所以也支持环境的概念。master的配置文件中定义各环境的路径，文件的路径基于指定环境的根目录（参见下面的例子）。

常见用法

get\_file

cp.get\_file用来从master下载文件到客户端，语法如下：

# salt '\*' cp.get\_file salt://vimrc /etc/vimrc

其中vimrc在master上的实际路径要看环境中定义的根目录，假设在master中有如下定义：

file\_roots: base: - /srv/salt/

那么vimrc的实际路径就是/srv/salt/vimrc，这样做的好处是，可以满足state系统中环境的概念。

源路径和目标路径中都可以使用模板，如下：

# salt '\*' cp.get\_file "salt://{{grains.os}}/vimrc" /etc/vimrc template=jinja

这个例子中，将下载与客户端操作系统名字相同的目录下的文件。

对于大文件，cp.get\_file支持gzip压缩，在参数中指定gzip的压缩级别，如下:

# salt '\*' cp.get\_file salt://vimrc /etc/vimrc gzip=5

其中，1代表作小压缩，9代表最大压缩。

cp.get\_file默认不在客户端上建立目录，如果客户端上没有这个目录了，文件拷贝将失败，可以指定makedirs=True来创建目录：

# salt '\*' cp.get\_file salt://vimrc /etc/vim/vimrc makedirs=True

在这个例子中，如果``/etc/vim不存在，将会建立此目录。

get\_dir

cp.get\_dir可以从master下载整个目录，语法如下：

# salt '\*' cp.get\_dir salt://etc/apache2 /etc

cp.get\_dir也支持模板和压缩：

# salt '\*' cp.get\_dir salt://etc/{{pillar.webserver}} /etc gzip=5 template=jinja

get\_url

cp.get\_url可以从一个URL地址下载文件，URL可以是msater上的路径（salt://），也可以是http网址。

salt '\*' cp.get\_url salt://my/file /tmp/minesalt '\*' cp.get\_url http://www.slashdot.org /tmp/index.html

get\_template

cp.get\_template可以在文件下载之前用模板引擎处理。

salt '\*' cp.get\_template salt://path/to/template /minion/dest

push

cp.push可以从客户端传文件到master上，处于很明显的安全考虑，默认没有启用此功能，如果你需要的话可以参考[文档](http://docs.saltstack.com/ref/modules/all/salt.modules.cp.html#module-salt.modules.cp)，很方便的开启。

### 17目录管理

/srv/stuff/substuf:

file.directory:

- user: fred

- group: users

- mode: 755

- makedirs: True

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

### 关键字

name 目录名称

user 用户

group 用户组

recurse 用户权限是否继承到下级

mode 权限

makedirs 设置为True，则目录不存在会创建

clean 目录下所有的文件或者文件夹将被清除

require 依赖的资源，如软件包或者文件

exclude\_pat 当clean设置为True，此值可以设置需要排除的资源

### 目录权限递归

/srv/stuff/substuf:

file.directory:

- uer: fred

- group: users

- mode: 755

- makedirs: True

- recurse:

- user

- group

关键字

name 目录名称

source 源目录salt://protocol

clean

require

user

group

dir\_mode

file-mode

template

context

defaults

include\_empty

include\_pat

exclude\_pat

创建链接

/etc/grub.conf:

file.symlink:

- target: /boot/grub/grub.conf

递归同步目录

/opt/nginx:

file.recurse:

- source: salt://files/nginx

这样会把salt://files/nginx所有的文件夹和文件递归同步到客户端

追加文件内容

/etc/motd:

file.append:

- text: |

Thou hadst better eat salt with the Philosophers of Greece,

than sugar with the Courtiers of Italy.

- Benjamin Franklin

file.rename(name, source, force=False, makedirs=False)

file.sed(name, before, after, limit=’‘, backup=’.bak’, options=’-r -e’, flags=’g’)

# Disable the epel repo by default

/etc/yum.repos.d/epel.repo:

file.sed:

- before: 1

- after: 0

- limit: ^enabled=

# Remove ldap from nsswitch

/etc/nsswitch.conf:

file.sed:

- before: ’ldap’

- after: ’’

- limit: ’^passwd:’

### 18 软件包管理

pkg.installed关键字

name 名称

fromrepo 从哪个源安装

skip\_verify 跳过GPG签名验证

version 软件包版本

httpd:

pkg.installed:

- fromrepo: mycustomrepo

- skip\_verify: True

- version: 2.0.6~ubuntu3

pkgs 一组数据包安装

mypkgs：

pkg.installed:

- pkgs:

- foo

- baz

mypkgs:

pkg.installed:

- sources:

- foo: salt://rpms/foo.rpm

- bar: http://somesite.org/bar.rpm

- baz: ftp://someothersite.org/baz.rpm

- qux: /minion/path/to/qux.rpm

pkg.latest

关键字

name

fromrepo

skip\_verify

pkgs

mypkgs:

pkg.latest:

- pkgs:

- foo

- bar

- baz

删除软件包

pkg.purged(name,\*\*kwargs)

pkg.removed(name, \*\*kwargs)

### 19 服务状态的管理

redis:

service:

- running

- enable: True

- reload: True

- watch:

- pkg: redis

如果redis服务已经存在，则pkg发生改变会reload服务

service.dead(name, enable=None, sig=None, \*\*kwargs)

service.disabled(name, \*\*kwargs)

service.enabled(name, \*\*kwargs)

service.mod\_watch(name, sig=None, reload=False, full\_restart=False)

service.running(name, enable=None, sig=None, \*\*kwargs)

### 20 用户管理

fred:

user.present:

- fullname: Fred Jones

- shell: /bin/zsh

- home: /home/fred

- uid: 4000

- gid: 4000

- groups:

- wheel

- storage

- games

testuser:

user.absent

user.absent(name, purge=False, force=False)

user.present(name, uid=None, gid=None, gid\_from\_name=False, groups=None,

optional\_groups=None, home=True, password=None, en-force\_password=True, shell=None, unique=True, system=False,fullname=’‘, roomnumber=’‘, workphone=’‘,

homephone=’‘)

### 21 hosts文件管理

salt-master:

host.present:

- ip: 192.168.0.42

host.absent(name, ip)

host.present(name, ip)

### 22 模板文件

#git clone <https://github.com/saltstack/salt-states>

#git clone <https://github.com/AppThemes/salt-config-example>

Salt官方文档

http://media.readthedocs.org/**pdf**/salt/latest/salt.**pdf**

关于puppet和salt的对比

<https://mywushublog.com/2013/03/configuration-management-with-salt-stack/>

23 动态变量

<http://imil.net/wp/2013/06/06/saltstack-dynamic-sls/>

I’ve been learning and diving into [SaltStack](http://saltstack.com/community.html) for about a month now, for both work and personal interest, that thing simply rocks. In the meantime, I’ve contributed a couple of modules, like [bridging](https://github.com/saltstack/salt/blob/develop/salt/modules/bridge.py) and [Xen](https://github.com/saltstack/salt/blob/develop/salt/modules/xapi.py) support, plus a couple of [grains](http://docs.saltstack.com/topics/targeting/grains.html) improvements for NetBSD.

But most of all, I’ve been preparing my *${DAYJOB}* infrastructure for *Salt*, and I must say this has been much easier than I thought, thanks to this beautifully designed piece of code.  
One aspect I’d like to share is the simple way I found to make a [minion](http://docs.saltstack.com/ref/configuration/minion.html) dynamically configured, through custom-made grains.

Like many companies, we have an Information System, which has a bunch of informations about everything on our network, and among those, the roles of the virtual machines which are started. Those informations are made available to a *minion*through an HTTP server, which will recognize the *minion’s* REMOTE\_ADDR and tell him what are its «roles».

The webserver is our always-favorite [nginx](http://wiki.nginx.org/Main), which interfaces a [WSGI](http://en.wikipedia.org/wiki/Web_Server_Gateway_Interface) server, [uWSGI](http://uwsgi-docs.readthedocs.org/en/latest/).

*nginx*‘s configuration is simple:

location / {

include uwsgi\_params;

uwsgi\_pass unix:///var/run/uwsgi/app/is/socket;

}

*uwsgi* application’s *.ini* is pretty trivial too:

[uwsgi]

workers = 2

log-date = true

plugins = python

chdir = /var/www/is

module = is

A very simple *python* script using [web.py](http://webpy.org/) resides in /var/www/is and gives the *minion*which will query him a [YAML](http://www.yaml.org/) structured output containing, among other data, the*minion*‘s roles.

I wrote a basic *grain module* which retrieves those informations and make them available as *grains* for the *minions*:

$ cat \_grains/is\_http.py

import requests

import yaml

def more\_infos():

'''

Returns the minion's roles and stuff.

'''

r = requests.get('http://private.fqdn/')

return yaml.load(r.text)

After deploying that *grain module* via salt '\*' saltutil.sync\_grains, the *minions*are now aware of what they’re meant to do, thus making a generic [top.sls](http://docs.saltstack.com/ref/states/#the-top-file) easy to write.

Thanks to [Jinja2](http://jinja.pocoo.org/), a *sls* file can be made more dynamic, and we can write a very complex scenario within a couple of lines:

$ cat top.sls

base:

{% set states = salt['cp.list\_states'](env) %}

'\*':

- common

{% if 'roles' in grains %}

{% for role in grains['roles'] %}

{% if role in states %}

- {{ role }}

{% endif %} # state exists

{% endfor %} # for roles

{% endif %} # role exists

'virtual\_subtype:Xen PV DomU':

- match: grain

- domus

{% if grains['host'] in states %}

{{ grains['host'] }}:

- {{ grains['host'] }}

{% endif %}

There we go, for every *role* a *minion* has, we can add a *sls* to match that *role* globally, as we can also add a very specific, host-related *sls*. Beautiful and handy, that’s what*Salt* is.

# 自动化测试

## 协议分析

## Spice

## CELT codec

is a compression algorithm for audio

FTP 协议和 HTTP 协议的12点比较（文件上传/下载）

iFCP

以太网光纤通道（Fibre Channel over Ethernet，FCoE）

 Internet SCSI (iSCSI)

远程直接内存访问 (RDMA-iSER)

SCSI RDMA 协议（SCSI RDMA Protocol，SRP）

# SOA架构与实施

## SOA基本概念与术语

服务架构的类型如下：

* 服务架构（Service Architectures）
* 服务组合架构（Service Composition Architecture）
* 服务库存架构（Service Inventory Architecture）
* 面向服务的企业架构（Service-Oriented Enterprise Architecture）

在Web服务合约内部，服务能力叫做服务操作

服务模型：

任务服务

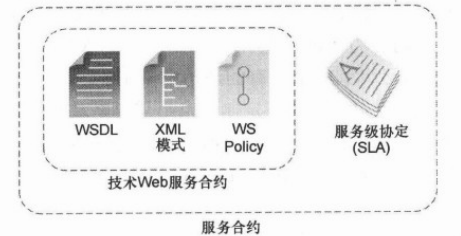
实体服务

工具服务

服务组合

服务库存

服务候选



组成Web服务常见文档

## JAX-RS规范

JAX-RS: Java API for RESTful Web Services是一个Java编程语言的应用程序接口,支持按照 表象化状态转变 (REST)架构风格创建Web服务Web服务. JAX-RS使用了Java SE 5引入的Java 标注来简化Web服务客户端和服务端的开发和部署。

JAX-RS提供了一些标注将一个资源类，一个POJOJava类，封装为Web资源。标注包括：

（1）、@Path，标注资源类或方法的相对路径

（2）、@GET，@PUT，@POST，@DELETE，标注方法是用的HTTP请求的类型

（3）、@Produces，标注返回的MIME媒体类型，（注解标注，这个注解可以包含一组字符串,默认值是\*/\*，它指定REST 服务的响应结果的MIME 类型，例如：application/xml、application/json、image/jpeg 等），你也可以同时返回多种类型，但具体生成结果时使用哪种格式取决于ContentType。CXF 默认返回的是JSON 字符串。

（4）、@PathParam，@QueryParam，@HeaderParam，@CookieParam，@MatrixParam，@FormParam,分别标注方法的参数来自于HTTP请求的不同位置，例如@PathParam来自于URL的路径，@QueryParam来自于URL的查询参数，@HeaderParam来自于HTTP请求的头信息，@CookieParam来自于HTTP请求的Cookie。

# 行业应用

【互联网广告技术目录】

1、广告投放技术

2、广告效果监测

3、数据处理及动态优化

4、网页监测及网络访问行为分析

5、目标受众精细分类

6、智能匹配广告

# 解决方案