Puppet 基础知识介绍-《开源软件架构》翻译

介绍

Puppet 是一个采用 Ruby 编写的开源 IT 管理工具，在 Google，Twitter 和纽交所等很多公司中作为数据中心自动化和服务器管理工具使用。它主要由创建此项目的 Puppet Labs 维护。由一个或者数百系统管理员操作的 Puppet 能够管理小至 2 台大至 50000 台这样大规模的机器集群。

Puppet 是一个配置并维护你的计算机的工具；使用它简单的配置语言，你向 Puppet 解释你所希望的机器配置参数，然后它将根据需要更改配置来匹配你的要求。如果你的规格有所变化，比如包更新，添加新用户或者更新配置-Puppet 将自动更新你的机器来匹配。如果他们已经按照需要配置，那么 Puppet 就会什么也不做。

通常情况下，Puppet 将会充分利用已有的系统特性来完成它的工作；比如，在 Red Hat 上它将会用 yum 进行包管理而用 init.d 进行服务管理，但是在 OS X 上它会使用 dmg 进行包管理而用 launchd 进行服务管理。Puppet 的指导目标之一就是让你无论在查看 Puppet 代码或者是系统本身时都让它的工作有意义，因此符合系统标准至关重要。

Puppet 来源于其他多个传统工具。在开源世界里，它受到 CFEngine 以及 ISconf 影响最多。前者是第一个开源通用配置工具，后者使用 make 来做一切工作，而这鼓励人们关注整个系统中明确的依赖关系。在商业世界里，Puppet 是对 Bladelogic 和 Opsware（之后都被大公司收购）的回应，它们在 Puppet 诞生之时就已经非常成功，但是它们都专注于卖给大公司的管理层而不是直接为系统管理员提供伟大的工具。Puppet 是为了解决这些工具类似的问题，但是却专注于与前二者完全不同的用户。

下面简单的例子展示如何使用 Puppet，这是一段用于确保 SSH 被安装且被合理配置的代码片段：

```
1.class ssh {
2.  package { ssh: ensure => installed }
3.    file { "/etc/ssh/sshd_config":
4.      source => 'puppet:///modules/ssh/sshd_config',
5.      ensure => present,
6.     require => Package[ssh]
7.    }
8.    service { sshd:
9.      ensure  => running,
10.     require => [File["/etc/ssh/sshd_config"], Package[ssh]]
11.    }
12.  }
```

这确保了包已经安装，文件也在正确位置，而且服务正在运行。注意到我们已经明确了这些资源相互间的依赖关系，因此我们总能以正确的顺序执行这些工作。这个类可用于任意需要应用此配置的主机。注意到构建 Puppet 配置的块是结构化对象，在这个例子里 package，file 和 service。在 Puppet 中我们称这些对象为资源，每个 Puppet 配置中可以归结为这些资源和资源间的依赖关系。

一个普通的 Puppet 站点将会拥有成百上千这样的代码段，它们被称为类；将这些类以文件形式存储在硬盘上称之为清单，将相关的清单分组称之为模块。例如，你可能会有一个 SSH 的模块，它里面包含这些 SSH 类和任意其他相关类，当然也会有 mysql,apache 和 sudo 等的模块。

大多数 Puppet 交互是通过命令行或者长连接 HTTP 服务，但是对于有些情况比如报告处理也会有图形界面接口。Puppet Labs 也围绕 Puppet 推出商业产品，这些产品更倾向于基于 WEB 的图形化接口。

Puppet 的第一个原型诞生于 2004 年夏季，到 2005 年 2 月已然成为大家关注的焦点。它最初由 Luke Kanies 设计并编写，Luke Kanies 是一名在编写小工具方面有丰富经验，但是却从未编写过超过 10000 行代码的工具的系统管理员。其实，Luke 是在编写 Puppet 过程中学习成为一名程序员的，这都在 Puppet 架构中的优点和缺点中表现出来。

Puppet 最初是为系统管理员设计的一个工具，以便使他们生活更轻松，工作更迅速有效率并且少犯错误。实现这一目标的第一个关键创新是上面提到的资源，它是 Puppet 的原语；它们既可以在大多数操作系统平台具有可移植性并且抽象具体实现细节，允许用户关注结果而不是怎样去实现。这些原语集合在 Puppet 的资源抽象层实现。

在给定主机上 Puppet 资源必须唯一。你能且仅能有一个叫'ssh'的包，一个叫'sshd'的服务，一个叫'/etc/ssh/sshd_config'的文件。这防止了不同部分的配置相互冲突，而且你可以在配置过程中很容易发现这些冲突。我们通过类型和名称来标识这些资源:比如 Package[ssh]和 Service[sshd]。你可以有拥有同样名称的包和服务因为它们是不同的类型，但是不能有两个包或者服务拥有相同的名称。

Puppet 的第二个关键创新点是能够明确资源间的相关 关系 。之前的工具关注单个任务的完成，而不管与其相关的大量工作怎样完成。Puppet 是第一个明确说依赖关系是配置的第一类而且必须以那种方式建模。它建立资源和资源间相互依赖关系的图作为核心数据类型，而且本质上 Puppet 中所有东西可以归结此图和它的顶点和边。

最后一个组成 Puppet 的主要组件是它的配置语言。这是一门解释性语言，而且它为了更多配置数据而不是完全的编程-它类似于 Nagios 的配置格式，但是确实受 CFEngine 和 Ruby 影响很大。

除了函数式组件，在 Puppet 整个开发过程中有两个指导性原则：它应该尽可能简单，甚至有时候会牺牲其性能来换取易用性；它首先应该被作为框架来构建其次才是应用程序，这样一来其他人就能够在 Puppet 内部原理之上建立自己想要的应用程序。众所周知，

Puppet 的框架需要一个杀手级应用程序来使其被广泛接受，但是框架始终是关注点而不是应用程序。大多数人认为 Puppet 是应用程序而不是隐藏在背后的框架。

当 Puppet 的原型第一次被构建出来时，Luke 本质上是一个拥有大量 shell 经验和一些 C 经验的资深 Perl 程序员，主要使用 CFEngine 工作。奇怪的是他有为一些简单语言构建解释器的经验，曾经编写过两个解释器，其中一个作为一个小工具的一部分而且也花费精力重写了 CFEngine 的解析器以使它更好维护（因为一些小的兼容性问题，这些代码从未被提交到此项目）。

选定一个动态语言作为 Puppet 的实现语言很容易，这基于更多高级工程师开发这的生产力和时间，但实践证明选择这门语言并不容易。最初的原型采用 Perl 无法继续，因此急需试验其他语言。Luke 尝试过 Python，但是发现这门语言从他如何思考这个世界的角度看时却有些怪异，基于从一个朋友那儿听到的关于效用的流言，Luke 尝试了 Ruby，只用四小时就构建了一个可用原型。当 Puppet 在 2005 年变成一个全程跟进的项目时，Ruby 完全是个未知数，因此坚持使用这种语言的决定具有很大风险，但是再一次程序员的生产力被认为是语言选择最重要的驱动力。Ruby 最重要的特点或者至少是不同于 Perl 的理由就是，Ruby 可以很容易就建立非分层次类之间的关系，而且同时能很好地映射 Luke 的大脑中，这竟然是至关重要的。

架构综述

本章主要是关于 Puppet 实现的架构（也就是说，那些我们用来让 Puppet 做它应该做的事情的代码），但是简单讨论它的应用程序架构是值得的（也就是说，这些部分是如何通信）。因此实现 使很多变得有意义。

Puppet 内部有两种模式：client/server 模式，一个中心服务器和多个运行在单独主机上的代理；serverless 模式，单个进程做所有工作。为了确保这些模式之间的一致性，Puppet 总是内部网络透明，因此两种模式无论是否经过网络都使用相同代码路径。每个可执行（进程）可以适当配置本地或者远程服务访问，但除此之外它们的行为是相同的。注意在 serverless 模式下你可以管理与 client/server 模式下同样规模的集群，通过将所有的配置文件推送到每个客户端本省并由它们直接解析。本部分将会关注 client/server 模式，因为它作为单独组建更容易被理解，但是记住这些对于 serverless 模式也同样正确。

Puppet 应用程序架构的决定性选择之一就是客户端不应该有对原生 Puppet 模块的访问权限；相反，它们获取编译后的配置。这有多个好处：第一，这保证了最小特权原则，让每个客户端仅仅知道它需要知道的（它应该如何被配置），但是它不知道任何其他的主机如何配置。第二，你可以完全将编译此配置（这可能包括对中心数据存储的访问权限）的权利和应用此配置的权力分开。第三，当它们重复应用一个配置时可以以断开连接模式来执行主机而不需要与中心服务器联系，这意味着即使服务器关闭或者客户端断开连接（就像在一个 mobile 安装环境中或者当你的客户端在 DMZ 环境）时你将保持工作。

鉴于这种选择，工作流程变得相对简单：

1.Puppet 代理进程收集关于运行此进程的主机信息，然后将这些信息传递给服务器

2.解析器使用这些系统信息和本地磁盘 Puppet 模块来编译针对特定主机的配置并将结果返回给这个代理

3.代理在本地应用此配置，因此影响本地主机状态和文件，并将结果报告给服务器

图 1 Puppet 数据流

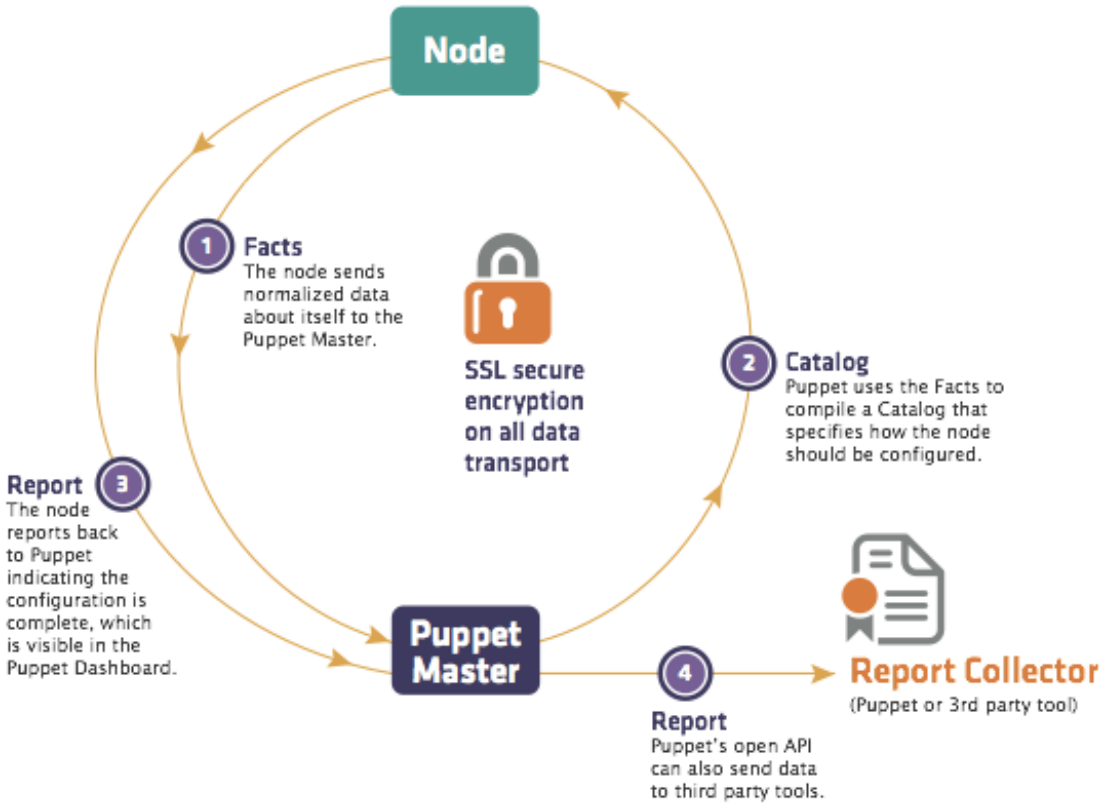
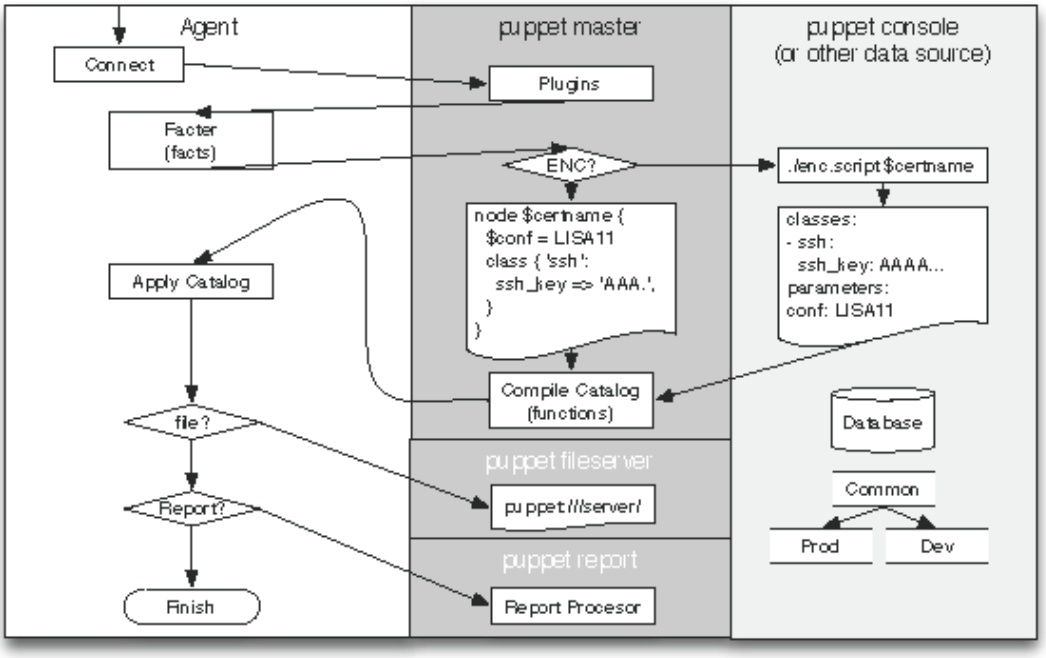


图 2 puppet 进程和组件之间的数据流阵列



这样以来，代理能够访问它的系统信息，它的配置和它生成的每个报告。服务器有所有数据的副本，加上对所有 puppet 模块的访问权限和所有后台数据库和以及可能会需要用于编译此配置的服务。

除了这些我们将会在后面详细介绍的工作流程中的组件，Puppet 还有许多用于内部通信的数据类型。这些数据类型是至关重要的，因为它们是所有通信如何完成的而且是公开的类型,这样任何其他工具都可以生产和消费这些数据类型。

最重要的数据类型是:

1.Facts：在每个机器上收集的系统数据并用于编译配置
2.Manifest：包含 Puppet 代码的文件，通常以模块形式进行分组管理
3.Catalog：给定主机的需要管理的资源以及资源间依赖关系的图
4.Report：在一个给定 Catalog 应用程序中所有生成事件的集合

除了 facts,Manifests,catalogs 和 reports。puppet 支持针对文件，证书（用于认证）以及其他数据类型。

组件分析

代理

Puppet 运行第一个遇到的就是 agent 进程。过去这是一个单独的可执行命令称为 puppetd,但是在 2.6 版本我们将它减少到只有一个可执行命令因此现在使用命令 puppet

agent 来调用，这类似于 Git 是如何工作的。agent 进程自身功能有限；它主要是配置和实现上面描述的工作流客户端部分的代码。 ** facter**

agent 之后的一个组件是一个叫做 facter 的外部工具，这是一个简单的用于发现运行主机的信息的工具。这是些类似操作系统，IP 地址，和主机名的数据，但是 Facter 很容易扩展因此很多组织添加自己的插件来发现自定义数据。agent 向服务器发送 Facter 发现的数据，从这一意义上讲它控制着工作流程。

外部节点分类器

在服务器端，要提到的第一个组件是外部节点分类器，或者称之为 ENC。ENC 接收主机名并返回一个包含该主机的高级配置的简单数据结构。ENC 通常是一个单独的服务或者应用；或者是另一个开源项目，例如 Puppet Dashboard 或者 Foreman；或者与已有数据存储集成，例如 LDAP。ENC 的作用是指定给定主机属于它的函数类和用于配置这些类所需的参数。例如，一个给定主机可能会在 debian 和 webserver 类中，并且有一个值为 atlanta 的参数 datacenter。

注意到直到 Puppet 2.7， ENC 都不是一个必须的组件；用户可以在 Puppet 代码中直接指定节点的配置。添加对 ENC 的支持是在 Puppet 发行 2 年后的时候，因为这时意识到对节点分类和配置是完全不同的，而且将这个问题用单独工具拆分比扩展语言来支持这两种功能更 有意义。ENC 总是被推荐的，而且在一定程度上很快将会成为一个必须的组件（在这一点上 Puppet 将会附带足够有用的要求而不会成为一个负担）。

一旦服务器接收到 ENC 中的分类信息和 Facter 的系统信息（通过 agent），它将所有信息封装成一个 Node 对象并将此对象传递给编译器。

编译器

正如之前所描述，Puppet 有一个用于描述特定系统配置的自定义语言。它的编译器实际上分三块：一个 Yacc 风格的解析器生成器和一个自定义的词法分析器；一组用于创建抽象语法树（AST）的类；以及用于处理这组类和作为部分系统 API 的函数之间的交互的编译器类。

关于此编译器最复杂的就是大多数 Puppet 配置在第一次引用时是推迟加载（用来同时减少加载和记录缺失但不需要依赖的不相关日志时间），这意味着并没有明确的调用来加载和解析代码。

Puppet 的解析器使用一个简单的采用开源工具 Racc 构建的 Yacc 风格的解析器生成器。不幸的是，Puppet 开始的时候还没有开源词法分析器，因此它使用了自定义的词法分析器。

因为在 Puppet 中使用抽象语法树，在 puppet 语法中每个语句赋值为 Puppet AST 类的一个实例（例如：Puppet::Parser::AST::Statement），而不是直接采取动作，而且这些 AST 实例被收集到一棵语法树。AST 在单个服务器编译来自多个不同节点配置时提供很大优势，因为这样能够一次解析但是多次编译。他也提供我们执行 AST 自省的机会。

Puppet 开始的时候只有非常少的 AST 例子可用，因此它经历了很多演变，现在已经到一个相对唯一且规范的阶段。我们创建很多小以它们名称为主键的 AST 而不是为整个配置创建单个 AST。例如，这段代码：

```
1.class ssh {
2.  package { ssh: ensure => present }
3.}
```

创建一个包含单个 Puppet::Parser::AST::Resource 实例的 AST，然后在将它按照名字"SSH"存在一个所有类的哈希的特殊环境下。（这里遗漏了其他类似类的设计，但是它们对于此短论没有必要）。

给定 AST 和节点对象（来自 ENC），编译器选择节点对象特定的类，查询并给它们评价。在整个评价过程中，编译器建立整个变量作用域树；每个类都有属于自己的附加在生成作用域的作用域。这意味着 Puppet 中是动态作用域；如果一个类包含另一个类，在包含类中可以直接查询被包含类的变量。这总是一个噩梦，我们正在消除这种能力的路上。

作用域树是临时的而且一旦编译完成就被丢弃，但是手工编译也逐步建立在整个编译过程中。我们称这个手工制品是 Catalog，但是它仅仅是资源及其关系图。没有任何变量，控制结构或者函数调用在 Catalog 中生存；它是纯数据，可以转换成 JSON，YAML 或者其他任何格式。

整个编译过程中，我们创建包含关系；一个类"包含"包含这个类所有的资源（比如，上面的 ssh 类包含 ssh package）。一个类可能包含一个包含很多定义或者独立资源的定义。一个 Catalog 更像一个平面的，非连通图；许多类，每个不超过两层深度。

这个图比较尴尬的一点就是它同时包含"依赖"关系，例如一个服务依赖一个包（可能是因为这个包安装创建了这个服务），但是这些依赖关系确实作为资源上的参数值而不是作为图结构中的边。图类（因为历史原因称之为 SimpleGraph）并不支持在相同图中的包含和依赖边，因此我们必须在不用目的下将它们相互转化。

事务

一旦 catalog 被完整创建（假设这里没有错误）就会被传递给事务。在一个客户端和服务器单独的系统中，事务运行在客户端，它通过 HTTP（从服务器）拉取 Catalog。（图 18.2）

Puppet 的的事务类提供真正的影响系统的框架，而其他所有我们讨论的围绕对象的东西都只是构建和传递。事务不像在许多一般系统中一样，例如数据库，puppet 事务没有原子性行为。

事务执行一个相对直接的任务：沿着图以不同关系所指定的顺序执行，并确保每个资源都是同步的。正如上面所描述，它必须把图中的包含边（例如 Class[ssh]包含 Package[ssh]

和 Service[ssh]）转化为依赖边(例如 Service[ssh]依赖 Package[ssh])，然后它做一个标准拓扑排序图，按照顺序选择每个资源。

对于给定资源，执行一个简单的三步过程：检索当前资源的状态，将此状态与它想要的状态作对比，然后做任何改变来解决差异。例如，给定如下代码：

```
1.file { "/etc/motd":
2.  ensure  => file,
3.  content => "Welcome to the machine",
4.  mode    => 644
5.}
```

事务检查文件/etc/motd 的内容和模式，如果它们不匹配这些特定状态，它将会修复它们中一个或者两个全部。如果/etc/motd 是个目录，它会备份这个目录中所有文件，然后删除目录，让会用一个有着合适内容和模式的文件来替代。

这个发生改变的过程实际上是由一个简单的定义在事务和资源之间的接口 ResourceHarness 类所处理。它降低了类与类之间的连接数，而且它让任意单独的变化变得容易。

资源抽象层

事务类是 puppet 能够完成工作的核心，但是所有的工作实际上是由资源抽象层（RAL）完成，从体系结构上来讲这也是 puppet 中最有趣的组件。

RAL 是 puppet 中第一个被创建的组件，不同于所用的语言，它最清楚的定义了用户能做什么。RAL 的工作就是定义什么是一个资源和资源在系统上是如何完成工作的，而且 puppet 语言是建立在由 RAL 建模的资源的基础之上。因为这一点，它也是系统中最重要的而且最难改变的组件。在 RAL 中有许多我们想要解决的问题，而且在过去这些年我们已经做了很多关键性的改进（最重要的决定是添加 Providers），但是在很长时期里依然有大量关于 RAL 工作要做。

在编译器子系统，我们为资源和资源类型分别建模为不同的类（名称分别是 Puppet::Resource 和 Puppet::Resource::Type）。我们的目标是让这些类同时组成 RAL 的核心，但是现在这两种行为（资源和类型）在一个类中建模，即 Puppet::Type（这个类命名一般是因为它早于使用 Resource 这个术语，而且在那个时间当不同机器通信时直接序列化内存结构，因此修改类名是一件复杂的工作）。

当 puppet 第一次被创建时，它可能会很合理将资源和资源类型行为放在同一个类中；毕竟，资源是资源类型的实例。然而过了一段时间，资源和它的资源类型之间的关系变得清晰，在一个传统继承模型中已经不能很好建模。例如资源类型定义一个资源能有什么参数，但是不能确定是否接受此参数（它们都做）。因此，我们的基类 Puppet::Type 有类级别行为决定资源类型如何表现，以及实例级别行为决定资源实例如何表现。基类也有管理

注册和检索资源类型的职责；如果你需要"user"类型，你可以调用 Puppet::Type.type(:user)。

混合的行为使得 Puppet::Type 很难去维护。整个类小于 2000 行代码，但是工作层次有三层-资源，资源类型，资源类型管理器，这让它变得复杂。这很明显是重构的重要目标，但是它比面向用户更直接，所以这里很难证明努力还不如直接在功能上扩展。

除了 Puppet::Type，RAL 中有两个重要类型的类，最有趣的是我们称为 Providers 的类。当 RAL 刚被开发出来，每个资源类型和那些附带能够告诉我们如果管理的代码的参数混合定义。例如，我们可能会定义一个”content”参数，然后提供一个能读取文件内容和另一个能够修改文件内容的方法。

```
1.Puppet::Type.newtype(:file) do
2.    ...
3.    newproperty(:content) do
4.        def retrieve
5.            File.read(@resource[:name])
6.        end
7.        def sync
8.            File.open(@resource[:name], "w") { |f| f.print @resource[:content] }
9.        end
10.    end
11.end
```

这是个相当简化的例子。（比如我们使用内部提供的检查校验和，而不是整个文件字符串等），但是你明白了这个思想。

当我们支持多种不同给定资源类型时管理变得几乎不可能。puppet 现在支持超过 30 中类型包管理，并且使用一个包管理资源类型变得不可能。取而代之的是，我们提供一个资源类型定义（本质上讲，资源类型名称）和它支持的属性（来自怎样管理这种类型资源）之间的干净接口。Providers 以一种很明显的命名方式为所有资源类型的属性定义 getter 和 setter 方法。例如，这是一个上面特性的 provider 可能的样子：

```
1.Puppet::Type.newtype(:file) do
2.    newproperty(:content)
3.end
4.Puppet::Type.type(:file).provide(:posix) do
5.    def content
6.        File.read(@resource[:name])
7.    end
8.    def content=(str)
9.        File.open(@resource[:name], "w") { |f| f.print(str) }
10.    end
11.end
```

这是一个写了更多代码的最简单的例子，但是却更容易理解和维护，特别是当任意属性或者 providers 数目增加时。

在这一阶段的开始我说过事务并不能真正的直接影响系统，反过来它会依赖 RAL 去做这些。现在很清楚这其实都是 providers 在实际做的工作。事实上，通常 providers 是 puppet 唯一的接触机器的部分。事务请求文件的内容，然后 provider 收集文件内容；事务指定文件内容应该变化，然后 provide 来改变文件内容。注意，虽然这样，但是 provider 从来不决定去影响系统--事务拥有决定权，然后 provider 来完成工作。这给了事务完全的控制权而不需要去了解文件，用户或者包，这个分离也使得我们能够在保证系统不被影响的情况下 puppet 拥有完全仿真模式。

在 RAL 中第二个重要的类类型是参数对自己的负责。我们真正支持三种类型的参数：metaparameters,它影响所有资源类型（比如，是否应该运行在仿真模式下）；parameters,它是没有反应在磁盘上的值（例如，是否应该追踪文件中的链接）；以及 properties，它为能够在磁盘上改变的资源建模（比如，文件的内容，服务是否在运行）。properties 和 parameters 之间的区别令很多人很疑惑，但是如果你将 properties 看成是 providers 里有 getter 和 setter 方法，它就变得相当直接。

报告

随着事务沿图执行和使用 RAL 来改变系统配置，它逐渐建立一个报告。这个报告主要由发生在系统上的变化的事件生成，这些事件反过来，能够反映已经完成的工作；它们保存一个时间戳和资源改变，之前的值，改变的新值，任何生成的消息以及改变是成功还是失败（或者是在仿真模式下）。

事件被包括在映射到每个资源的 ResourceStatus 对象中。因此，对于一个给定事务，你知道所有运行的资源，以及发生的任何变化，同时所有你可能需要知道的关于这些变化的元数据。

一旦事务结束，一些技术的度量指标被计算然后存储在报告中，接着这些报告会被发送给服务器（如果配置发送报告）。随着报告的发送，配置阶段就结束了，agent 进入休眠或者直接结束进程。

基础设施

现在我们地 puppet 做什么和怎么做有了个完整的了解，需要花费点时间来关注与能力无关却对完成工作依然很重要的部分。

插件

puppet 伟大之处之一就是它的可扩展性。在 puppet 中至少有 12 中不同类型的可扩展性，而且大多数这些扩展都能被任何人使用。例如，你可以为这些地方创建自定义插件：

```
1.资源类型和自定义 providers
2.报告处理器，例如在自定义数据库存储报告
```

```
3.现有数据存储交互的 Indirector 插件
4.发现机器额外信息的 facts
```

然而，Puppet 的分布式本质意味着 agents 需要一种方式来检索和加载新的插件。因此，在每次 puppet 运行的开始，我们需要做的第一件事就是下载所有的服务器可用的插件。这些可能包括新的资源类型或者 providers，新的 facts，或者甚至是新的报告处理器。

这使得大尺度更新 puppet agent 而不改变核心 puppet 包变得可能。这对于高度自定义安装 puppet 非常有用。

Indirector

到现在你已经知道在 puppet 中我们有个传统就是类的命名不好，根据大多数人，这点不重要 。Indirector 是一个相对标准的具有很好扩展性的反转控制框架。反转控制系统允许你将功能开发和怎样控制使用哪个功能分开。在 puppet 的例子中，这允许我们使用许多插件提供不同的功能，例如通过 HTTP 获取编译器或者在进程内加载，并且在一个小的配置总切换它们而不是改变代码。换句话说，puppet 的 indirector 是一个服务定位器的基本实现，正如维基百科中对“inversion of control”的描述一样。从一个类到另一个类的所有流程都要由类似标准 RESTful 接口经过 indirector, (例如，我们支持查找，保存和销毁作为方法)，当配置 agent 使用 HTTP 端点而不是使用编译器端点来检索 catalog 时 puppet 从 serverless 模式切换到 client/server 模式是个很大的问题。

因为它是一个发转控制框架而配置严格不同于代码路径，这个类很难去理解，尤其是当你调试为什么要使用一个给定的路径。

网络

puppet 的原型编写于 2004 年夏季，当时最大的问题是不知该使用 XMLRPC 还是 SOAP。我们选择了 XMLRPC，而且它们工作良好，但是有也就少不了其他人所有人都有会遇到的很多问题：它不能很好支持组件间标准的接口，而且结果很容易趋于过度复杂。我们也有严重的内存问题，因为 XMLRPC 编码需要导致每个对象在内存中至少出现两次，这对于大文件严重负载。

对于我们的 0.25 版本（开始于 2008），开始进行将所有网络向类似 restful 模型切换，但是我们选择了一个比仅仅改变网络更复杂的路径。我们开发了 indirector 作为标准内部组件通信框架，并且将构建 REST 端点作为一个可选方案。两个版本之后开始全面支持 REST，而且我们还没有完全转换到使用 JSON(而非 YAML）作序列化。两个重要原因使我们着手切换到使用 JSON：第一，YAML 处理 ruby 异常慢，而纯 ruby 处理 JSON 非常快；第二，大多数 YAML 不支持跨语言移植，而且在不用 puppet 版本之间通常也不能移植，因为本质上它们序列化内部 ruby 对象。

我们的下一个 puppet 重大版本将会完全移除 XMLRPC 支持。

为了能够和大家更好的交流和学习 Puppet，本人 2014 年又新开辟了微信公众号进行交流学习，目前已经有 300 多人同时收听，喜欢 Puppet 的大神们可自行加入哦。

如果你有好的有关 Puppet 的咨询也可以给我投稿，投稿地址：admin@kisspuppet.com

微信公众号：“puppet2014”，可搜索加入，也可以扫描以下二维码
