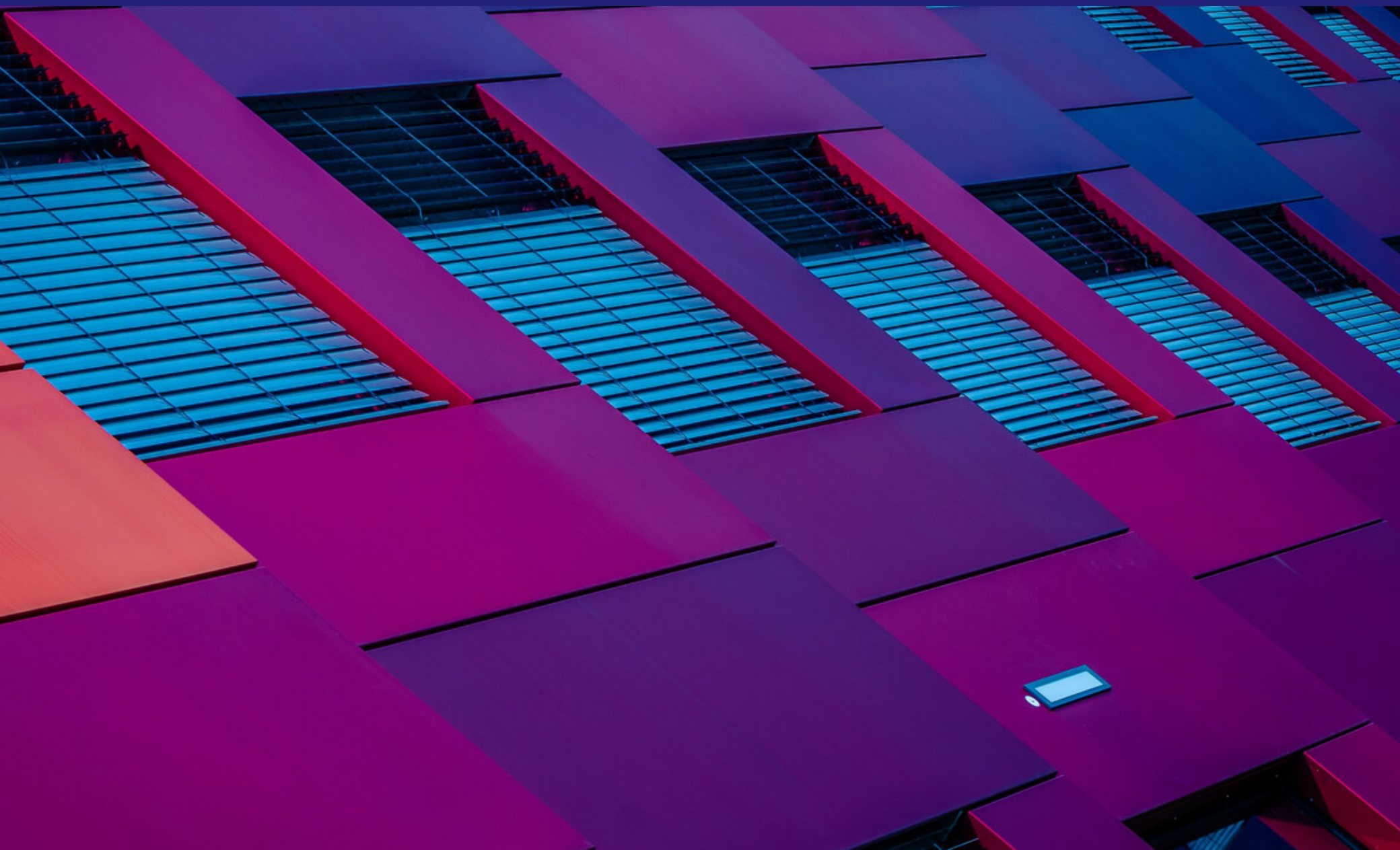


# Terraform

## 自动化管理云基础设施

基础设施即代码实战



徐小东

# Terraform： 自动化管理云基础设施

徐小东

献给

海燕和铭基

# 目录

<b>第一章 准备篇</b>	<b>1</b>
1.1 什么是基础设施即代码 . . . . .	1
1.1.1 基础设施即代码的价值 . . . . .	1
1.1.2 基础设施即代码的工具 . . . . .	3
1.2 Terraform 简介及工作原理 . . . . .	5
1.2.1 Terraform 简介 . . . . .	6
1.2.2 Terraform 的工作原理 . . . . .	6
1.3 Terraform 的使用场景 . . . . .	8
1.4 安装 Terraform . . . . .	9
1.4.1 在 Linux 上安装 Terraform . . . . .	9
1.4.2 在 Windows 上安装 Terraform . . . . .	10
1.4.3 在 macOS 上安装 Terraform . . . . .	11
1.4.4 利用 tfenv 管理多个 Terraform 版本 . . . . .	11
1.5 配置 Terraform 开发环境 . . . . .	15
1.5.1 Terraform 扩展简介 . . . . .	16
1.5.2 安装 Terraform 扩展 . . . . .	16
<b>第二章 入门篇</b>	<b>19</b>
2.1 创建云基础设施 . . . . .	19

2.1.1	准备 AWS 用户 . . . . .	19
2.1.2	配置 AWS 访问认证 . . . . .	20
2.1.3	Terraform 初始化 . . . . .	22
2.1.4	创建第一个云基础设施资源 . . . . .	25
2.1.5	HCL 配置语言基础 . . . . .	33
2.2	变更云基础设施 . . . . .	38
2.2.1	输入变量 . . . . .	39
2.2.2	使用数据源 . . . . .	46
2.2.3	输出变量 . . . . .	48
2.2.4	使用模板文件 . . . . .	50
2.2.5	处理依赖图 . . . . .	51
2.3	销毁云基础设施 . . . . .	52
<b>第三章 提高篇</b>		<b>55</b>
3.1	通过 Provisioner 配置服务器 . . . . .	55
3.1.1	Connection 块 . . . . .	59
3.1.2	remote-exec Provisioner . . . . .	60
3.1.3	file Provisioner . . . . .	62
3.2	利用 Terraform 模块重用资源 . . . . .	65
3.2.1	模块基础 . . . . .	66
3.2.2	模块输入 . . . . .	68
3.2.3	模块输出 . . . . .	70
3.2.4	调用模块 . . . . .	70
3.2.5	使用外部模块 . . . . .	76
3.3	与团队成员协同使用 Terraform . . . . .	78
3.3.1	Terraform 远端状态 . . . . .	78

3.3.2	共享状态文件 . . . . .	81
3.3.3	锁定状态 . . . . .	84
3.3.4	Terraform 工作区 . . . . .	85
3.4	测试云基础设施 . . . . .	87
3.4.1	准备测试环境 . . . . .	87
3.4.2	编写测试代码 . . . . .	90
3.4.3	执行测试 . . . . .	93
<b>第四章 技巧篇</b>		<b>95</b>
4.1	实现循环 . . . . .	95
4.1.1	利用 count 元参数实现循环 . . . . .	95
4.1.2	利用 for_each 元参数实现循环 . . . . .	98
4.2	条件选择 . . . . .	100
4.3	零宕机部署 . . . . .	102
4.4	格式化并验证 Terraform 代码 . . . . .	107
4.5	利用 terraform console 对表达式求值 . . . . .	109
4.6	Terraform 命令参考 . . . . .	110
4.7	Terraform 参考资源 . . . . .	111



# 表格

4.1	Terraform 命令参考 . . . . .	110
-----	--------------------------	-----





# 插图

1.1	Terraform 的工作原理 . . . . .	7
1.2	在 VS Code 中安装 Terraform 扩展 . . . . .	17
2.1	设置 IAM 用户名称及访问类型 . . . . .	20
2.2	设置 IAM 用户访问权限 . . . . .	21
2.3	IAM 用户的访问密钥 ID 和访问密钥 . . . . .	21
2.4	执行 Terraform 初始化 . . . . .	24
2.5	执行 Terraform 计划 . . . . .	27
2.6	执行 Terraform 计划 . . . . .	28
2.7	AWS EC2 Web 控制台 . . . . .	30
2.8	terraform.tfstate 文件的内容 . . . . .	32
2.9	添加的 SSH 公钥 . . . . .	37
2.10	创建的防火墙规则 . . . . .	37
2.11	替换 AWS EC2 实例 . . . . .	45
2.12	NGINX 默认服务页 . . . . .	46
2.13	资源依赖图 . . . . .	52
2.14	销毁云基础设施 . . . . .	53
3.1	初始化模块 . . . . .	75
3.2	搜索 Terraform 模块 . . . . .	77

3.3	在 Terraform Cloud 上新建组织 . . . . .	79
3.4	创建工作区时跳过连接 VCS . . . . .	79
3.5	设置工作区名称 . . . . .	80
3.6	配置执行模式 . . . . .	80
3.7	生成 Terraform Cloud 令牌 . . . . .	82
3.8	在 Terraform Cloud 上存储状态 . . . . .	83
3.9	状态锁 . . . . .	85
3.10	管理状态锁 . . . . .	85
3.11	执行 go test 的输出 . . . . .	94
4.1	利用 count 元参数循环创建基础设施 . . . . .	98

# 更新

你可以从 <https://selfhostedserver.com/terraform> 获取本书的更新版本，包括 PDF、ePub 和 Mobi 格式。

- 2019.09.27：初版



# 作者简介

徐小东, 网名 ~toy。GNU/Linux 爱好者, DevOps 践行者。喜技术, 好分享。通过 <https://linuxtoy.org> 网站数年间原创及翻译文章达 3000 余篇。另著有《像黑客一样使用命令行》<sup>1</sup>、《容器化工具三剑客: Podman、Buildah 和 Skopeo》<sup>2</sup>、《Terraform: 自动化管理云基础设施》<sup>3</sup>, 译有《笨办法学 Git》<sup>4</sup>、《Perl 程序员应该知道的事》等图书。Twitter: <https://twitter.com/linuxtoy>, Mail: [xuxiaodong@pm.me](mailto:xuxiaodong@pm.me)<sup>5</sup>。

---

<sup>1</sup><https://selfhostedserver.com/usingcli-book>

<sup>2</sup><https://selfhostedserver.com/nextcontainer>

<sup>3</sup><https://selfhostedserver.com/terraform>

<sup>4</sup><https://selfhostedserver.com/learngit>

<sup>5</sup><mailto:xuxiaodong@pm.me>



# 第一章 准备篇

随着云计算的逐渐流行，创建和管理诸如虚拟服务器、网络、存储、负载均衡等基础设施的方式已然发生转变。传统的手动或者脚本管理方式虽然能够应付数台服务器这种少量使用场景，但是对于多则成百上千的大规模服务器数量来说，显然不再适用和高效。为此，我们需要一种能够自动化的、可重复的以及可靠的创建和管理云基础设施的方法。来自 Google（谷歌）、Amazon（亚马逊）、Netflix（网飞）等互联网公司的做法表明，基础设施即代码正是用来解决此类问题的最佳实践。那么，究竟什么是基础设施即代码呢？

## 1.1 什么是基础设施即代码

基础设施即代码，其对应的英文为 Infrastructure as code，意指通过编写并执行代码来定义、部署以及更新基础设施。采用基础设施即代码，需要我们转变过去的思维，将云主机、网络（包括路由、防火墙）、存储等所有基础设施都当作软件来对待，从而能够有效利用软件开发的良好实践。使用源代码来表示基础设施，不仅可对其进行编程处理，而且能版本化、共享、重用、调试，甚至在出现错误时还可能还原。对于在软件开发生命周期中进行的代码评审、自动化测试、CI（Continuous integration，持续集成）和 CD（Continuous delivery，持续交付）等活动，也可以同样应用到基础设施上。因此，我们就能创建更加容易追溯、监视和调试的服务运行环境。

### 1.1.1 基础设施即代码的价值

所有的云计算平台都无一例外的提供了 Web 操作界面。对于大部分人而言，通过 Web 操作界面只需要点点鼠标即可创建基础设施。显然，它看起来非常容



易使用。那么，我们为什么不推荐这种手动管理基础设施的方式呢？这是因为手动管理方法一般具有下列弊端：

- 十分容易出错。仔细回想一下，你上次手动创建基础设施犯错是什么时候？是不是才没过几天？
- 很难创建多个相同且一致的环境。每一个环境都很特殊。它们的配置各不相同。久而久之便成了雪花服务器。这些服务器脆弱不堪，一触即崩。“标准”二字在他们的字典里根本就没有存在过。
- 非常耗时。虽然这是手动操作所需付出的代价之一，但是我们实在没有必要为此浪费大把的时间，毕竟我们还有更好的选择。
- 想要保持环境最新真的是难。给老掉牙的系统打补丁？拜托别碰，你可能会陷入“依赖地狱”。
- 最重要的是你的基础设施无法像收银小票一样长期保留下来。它只散见于几篇过时的文档或某一时刻的大脑中。

而使用基础设施即代码这种实践将带来多方面的好处，我们认为它最重要的价值在于：

- **版本控制**

我们可以将用来定义基础设施的代码存储在版本控制系统（如 Git）中。一方面，从提交的版本记录我们可追溯基础设施从创建到更新的整个历史过程；另一方面，假如基础设施遇到故障，那么我们便能根据已有的历史记录进行排查。此外，利用版本控制系统，我们甚至还能回滚具有问题的基础设施，从而将其还原到正常状态的版本。

- **自动化**

执行定义的基础设施代码通常是自动化完成。显然，它比在 Web 界面中通过手动点击按钮来创建基础设施的过程要快。同时，它也更安全，手动操作一般容易导致人为错误；而自动化过程则更加一致且可重复。

- **重用**

通过把定义基础设施的代码打包成模块，我们从而能够将其变成可以重复使用的组件。对于开发、测试及生产环境中的同类型基础设施，我们无需从头开始构建，只需加以重复利用。因此，重用不仅可以节省我们创建基础设施的时间，而且经过测试的模块也将使基础设施更加可靠。

- **可执行的文档**

从传统上讲，创建基础设施的过程要么被记录到文档中，要么存在于某个运维专职人员的大脑里。使用文档的问题是，它总是容易过时，而且维护它需要耗费大量的时间。而将特定技术锁定在专职人员身上有时则让情况更加严重，假如该名人员不在工作岗位上，那么将没有别的人知道如何创建基础设施。如果我们将基础设施通过代码的形式定义，那么团队中的每一个人不但都可以查看或评审这些代码，而且还能直接予以执行。

周而复始的手动管理基础设施，时间长了难免不变得枯燥和乏味。如此工作既无创意，也无挑战。通过基础设施即代码的方法在改进我们工作的同时，让我们专注于从事更有价值的事情。

### 1.1.2 基础设施即代码的工具

我们已经看到了采用基础设施即代码具有巨大的优势。然而，我们如何才能实现它呢？根据 Yevgeniy Brikman（叶夫根尼 • 布里克曼）的说法<sup>1</sup>，如果你想要实现基础设施即代码，那么至少包括以下几类工具可以选择使用：

- **脚本**

通过编写脚本来自动化管理基础设施，这是最直接，同时也是历史最悠久的方式。在早期，用来编写脚本的语言通常以 Shell 为主（如 GNU Bash<sup>2</sup>），但后来更加通用的编程语言（如 Perl<sup>3</sup>、Python<sup>4</sup>、Ruby<sup>5</sup> 等）也加入了进来。使用脚本的好处是简单且上手相对更快，一般可以利用熟悉的编程语言来编写。然而，其缺点也非常明显，除了要编写大量的定制化代码之外，后期的维护更是颇成问题。

- **配置管理工具**

当下主流的配置管理工具包括 Ansible<sup>6</sup>、SaltStack<sup>7</sup>、Chef<sup>8</sup>、Puppet<sup>9</sup>、

---

<sup>1</sup><https://blog.gruntwork.io/a-comprehensive-guide-to-terraform-b3d32832baca>

<sup>2</sup><https://www.gnu.org/software/bash/>

<sup>3</sup><https://www.perl.org/>

<sup>4</sup><https://www.python.org/>

<sup>5</sup><https://www.ruby-lang.org/>

<sup>6</sup><https://www.ansible.com/>

<sup>7</sup><https://www.saltstack.com/>

<sup>8</sup><https://www.chef.io/>

<sup>9</sup><https://puppet.com/>

CFEngine<sup>10</sup> 等等，这些工具用来在现有的服务器上自动安装并配置软件。与脚本相比，一方面，使用这类工具通常采用描述或声明式的 DSL（特定领域语言）来编写。由于不再需要学习专门的编程语言，所以在使用难度上大为降低。

另一方面，使用脚本来实现幂等通常较为困难，而通过配置管理工具却能轻易实现。所谓幂等，就是说不管代码执行多少次，其最终行为都将是正确的。例如，假设我们想要为 `/etc/hosts` 追加一行条目，如果使用脚本，那么每次执行可能都将添加相同的内容，最终导致这些内容重复存在于文件中。经验和教训告诉我们，编写仅仅工作的脚本很容易，但要编写好好工作的脚本却十分困难。相反，使用配置管理工具则不会存在这样的问题。这是因为它们会根据描述或声明的状态进行判断，若该文件没有这行条目则添加，反之则什么也不做。

此外，脚本一般在单机上执行，而且临时脚本通常用完即弃。配置管理工具则往往针对分布式执行而设计，能够同时管理大量的远程服务器。

## • 服务器模板工具

代替在现有的服务器上安装和配置软件，我们也可以直接将预装好的环境打包成服务器模板。这种模式被称为不变基础设施模式，该模式现在已经得到了越来越多的采用。

一般用来制作服务器模板的技术包括容器化技术和虚拟化技术，而相应的工具则有 Docker<sup>11</sup>、Podman<sup>12</sup>、Packer<sup>13</sup>、Vagrant<sup>14</sup> 等等。通过这些工具打包好的服务器模板称为镜像，将镜像进行部署后最终成为运行的实例。

不变基础设施模式的惯用做法是，从来不对实例做任何变更。但是，当不得不进行变更时，则从服务器模板创建新的镜像。另外，如果运行的实例出现故障，那么也不用对它进行修复，而是将它抛弃，并根据镜像重新部署新的实例。

## • 服务器准备工具

无论是配置管理工具，还是服务器模板工具，它们定义的代码都针对现有的服务器运行。与它们不同的是，服务器准备工具则用来创建服务器本身。

---

<sup>10</sup><https://cfengine.com/>

<sup>11</sup><https://www.docker.com/>

<sup>12</sup><https://podman.io/>

<sup>13</sup><https://packer.io/>

<sup>14</sup><https://www.vagrantup.com/>

这类工具包括 Terraform<sup>15</sup>、AWS CloudFormation<sup>16</sup>、OpenStack Heat<sup>17</sup> 等等。

本书将专注于介绍 Terraform，根据我们长期观察，Terraform 是目前用来编写基础设施即代码的最佳开源工具之一。我们喜爱 Terraform，不仅因为它支持 Amazon Web Services (AWS)<sup>18</sup>、Google Cloud<sup>19</sup>、Microsoft Azure<sup>20</sup>、DigitalOcean<sup>21</sup>、Linode<sup>22</sup>、Vultr<sup>23</sup>、阿里云<sup>24</sup>、腾讯云<sup>25</sup>等市场上广泛的云供应商，而且也在于它的生态越来越成熟。

## 1.2 Terraform 简介及工作原理

与 Vagrant 和 Packer 一样，Terraform 也是 HashiCorp<sup>26</sup> 旗下的开源产品。Terraform 使用 Go<sup>27</sup> 编程语言开发而成，于 2014 年 7 月 28 日推出了第一个公开发布版本。从发布至今的 5 年间，Terraform 已经得到了包括 AWS、IBM Cloud<sup>28</sup>、Google Cloud、Microsoft Azure、Oracle Cloud<sup>29</sup>、阿里云、腾讯云、VMware vSphere<sup>30</sup>、OpenStack<sup>31</sup> 等多种公有云及私有云的支持。因为 Terraform 在目前是如此流行，掌握它的用法自然成为了一项必备的 IT (Information technology, 信息技术) 技能。

---

<sup>15</sup><https://www.terraform.io/>

<sup>16</sup><https://aws.amazon.com/cloudformation/>

<sup>17</sup><https://www.openstack.org/software/releases/stein/components/heat>

<sup>18</sup><https://aws.amazon.com/>

<sup>19</sup><https://cloud.google.com/>

<sup>20</sup><https://azure.microsoft.com/>

<sup>21</sup><https://www.digitalocean.com/>

<sup>22</sup><https://www.linode.com/>

<sup>23</sup><https://www.vultr.com/>

<sup>24</sup><https://www.alibabacloud.com/zh>

<sup>25</sup><https://cloud.tencent.com/>

<sup>26</sup><https://www.hashicorp.com/>

<sup>27</sup><https://golang.org/>

<sup>28</sup><https://www.ibm.com/cloud>

<sup>29</sup><https://www.oracle.com/cloud/>

<sup>30</sup><https://www.vmware.com/products/vsphere.html>

<sup>31</sup><https://www.openstack.org/>



### 1.2.1 Terraform 简介

Terraform 是用来创建和管理基础设施的得力而优雅的工具。它通过自创的 HCL<sup>32</sup> (HashiCorp Configuration Language, HashiCorp 配置语言) 配置语言来声明想要创建的基础设施的最终状态。例如, 假设我们打算在 AWS 上创建一台计算实例, 那么只需为其指定实例的类型 (由此决定可以使用多少 CPU、内存及网络带宽) 和操作系统的镜像 (如 Debian GNU/Linux) 即可, 而完全不必在意该计算实例是如何创建的过程。因为, Terraform 将为我们关心这一切。

在应用声明配置之前, Terraform 做得非常好的一点是, 它允许我们先执行计划, 故而能够让我们清晰的看到即将实施的基础设施的具体变化。通过执行计划, 我们不仅可以详细了解创建、变更以及销毁的是哪些基础设施, 而且还能据此判断是否需要最终加以应用配置。

Terraform 还能与 Ansible、SaltStack、Chef、Puppet 等配置管理工具集成, 在创建基础设施的同时, 执行安装、配置软件及服务。利用 Terraform 这项功能, 我们可以重用现有的配置管理资源, 从而将以往的工作流和 Terraform 整合在一起。

目前, Terraform 主要用于管理云计算平台和 SaaS (Software as a Service, 软件即服务) 基础设施, 其范围包括虚拟服务器、数据存储、网络、DNS、CDN、数据库、监视服务等等。

### 1.2.2 Terraform 的工作原理

Terraform 的工作原理如图 1.1 所示<sup>33</sup>。以自顶向下的视角来看, 首先, 我们通过人类可读的 HCL 配置语言将基础设施表示为代码。在代码中, 我们声明想要创建的基础设施的最终状态。这一步, 我们既可以从 Terraform 公开的模块仓库及社区中重用基础设施资源, 也能够将 Terraform 状态文件存储到云中或从云中同步该文件以便与整个团队的所有成员协作。

接着, Terraform 负责解析我们在上一步编写的基础设施代码, 并生成相应的执行计划。这份执行计划向我们展示 Terraform 实际上在应用代码时所做的具体操作。

---

<sup>32</sup><https://github.com/hashicorp/hcl>

<sup>33</sup><https://www.hashicorp.com/products/terraform/>

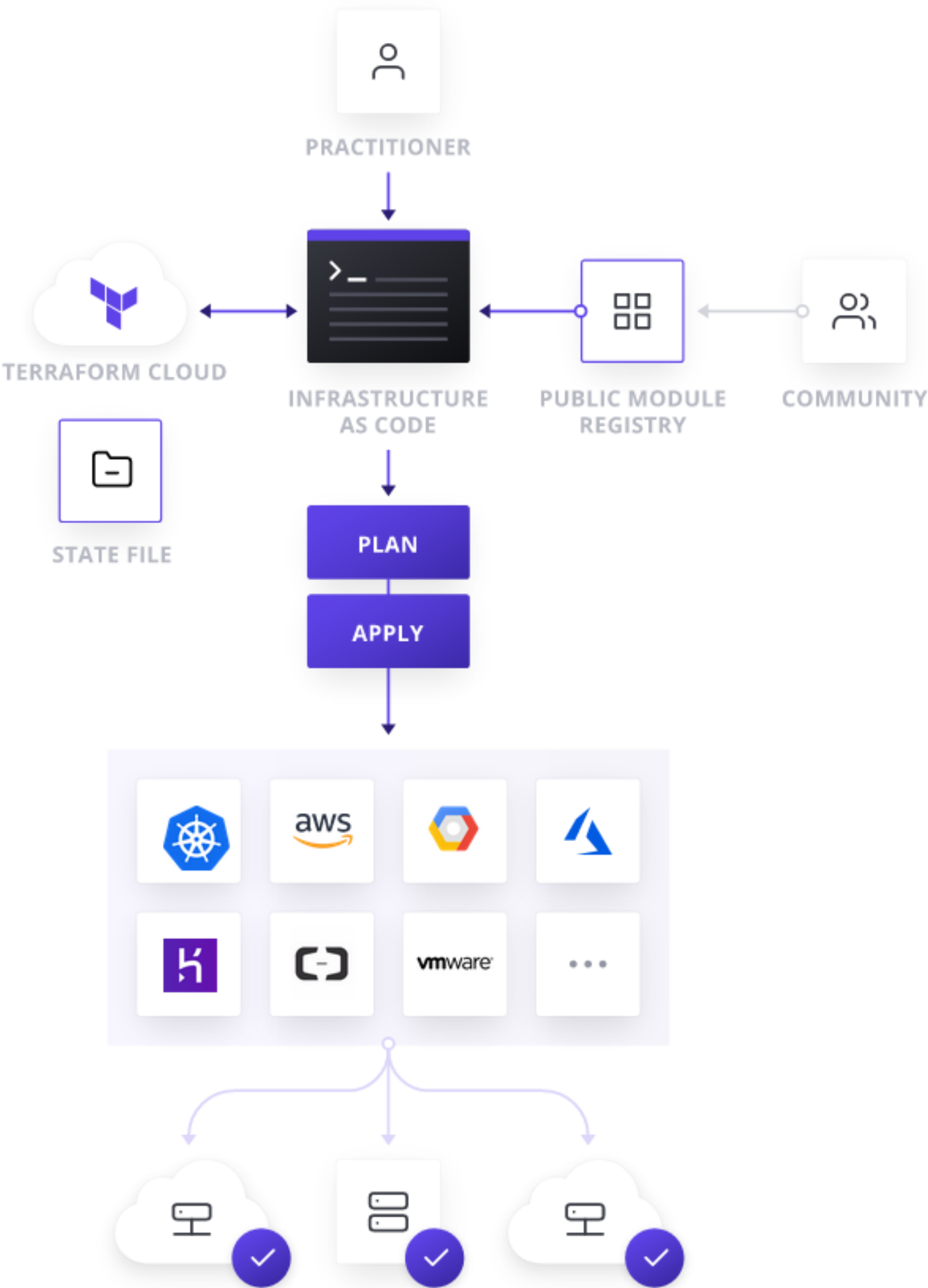


图 1.1: Terraform 的工作原理

最后，Terraform 应用我们定义的基础设施代码。在内部，Terraform 将 HCL 配置语言转换成向云供应商（如 AWS、Google Cloud、Microsoft Azure、阿里云等）或应用服务（如 Kubernetes、Heroku 等）进行调用的 API（Application programming interface，应用程序编程接口），从而最终实现创建和管理基础设施。

## 1.3 Terraform 的使用场景

我们选择 Terraform 的缘由之一在于它具有广泛地使用场景，这主要体现在下列方面：

- **自助式服务**

在一个团队中，通常由运维人员负责准备基础设施，并执行相应的管理过程。随着业务规模的增长，数量有限的运维人员很容易成为瓶颈。利用 Terraform，运维人员能够将基础设施定义成标准的资源和技术栈，从而以自助式的服务提供给团队中的其他人员消费。

- **多层应用程序**

Terraform 聚焦于基础设施的部署，其范围涵盖计算实例、存储及网络。这使它成为构建多层应用程序的理想工具。无论是具有数据库后端的经典 Web 应用，还是具有 Web、缓存、中间件、数据库的多层应用，每一层都可以通过 Terraform 定义成代码。代码一旦在手，它们将更容易管理和自动伸缩。

- **一致的环境**

在运维工作中一个常见的现实问题是，开发环境和测试环境总是很难与生产环境保持一致。结果即便应用程序通过了测试，然而在生产环境中仍有可能无法正常运行。通过 Terraform，我们把生产环境的基础设施代码化，然后将其与开发环境和测试环境共享，从而确保所有环境的一致性和兼容性。

- **CD（持续交付）**

如果你的应用程序已经通过 CI（持续集成）实现了自动化构建，那么不妨考虑更进一步，将 CI 中构建的制品进行自动化部署。使用 Terraform

自动准备各种环境，从而实现更有针对性的按需部署。同时，把它们串联在一起，形成真正的自动化部署流水线。

- 管理基础服务

除了管理各种云基础设施之外，Terraform 也能管理许多基础服务和工具。例如，GitHub<sup>34</sup> 组织和仓库、GitLab<sup>35</sup> 群组与项目、Grafana<sup>36</sup> 数据源及 Dashboard 等等。不管你信不信，有人甚至使用 Terraform 来订购匹萨<sup>37</sup>。Terraform 的用途之广由此可见一斑。

## 1.4 安装 Terraform

Terraform 支持 Linux、macOS、Windows、FreeBSD、OpenBSD、Solaris 等多种操作系统，因其使用单个二进制包进行分发，所以只需下载后将 Zip 文件解包，并移到操作系统的 **PATH** 所在目录中即可完成安装。

### 1.4.1 在 Linux 上安装 Terraform

如果你想要在 Linux 上安装 Terraform，那么在打开 Terraform 的官方下载页面<sup>38</sup>之后，找到适合 Linux 架构的版本（本书将使用 64 位架构，Terraform 也具有 32 位架构和 ARM 架构）。截至编写本书时为止，Terraform 的最新版本为 0.12.9。接着，按照下列步骤执行操作：

1. 使用 `wget`（或 `curl`）命令下载 Terraform 的 Zip 压缩包：

```
mkdir ~/terraform
cd ~/terraform
wget https://releases.hashicorp.com/terraform/0.12.9/
terraform_0.12.9_linux_amd64.zip
```

---

<sup>34</sup><https://github.com/>

<sup>35</sup><https://about.gitlab.com/>

<sup>36</sup><https://grafana.com/>

<sup>37</sup><https://github.com/ndmckinley/terraform-provider-dominos>

<sup>38</sup><https://www.terraform.io/downloads.html>



2. 执行 `unzip` 命令来解包：

```
unzip terraform_0.12.9_linux_amd64.zip
```

3. 将 `terraform` 二进制文件移到操作系统的 **PATH** 所在目录 `/usr/local/bin`：

```
sudo mv terraform /usr/local/bin
```

最后，我们通过执行 `terraform version` 命令来验证 Terraform 是否安装成功：

```
terraform version
Terraform v0.12.9
```

`terraform version` 命令的输出结果说明我们正在使用 Terraform 的 0.12.9 版本，同时这也意味着 Terraform 的安装一切正常。

值得一提的是，通常使用 Linux 发行版所带的包管理器（如 `apt`、`yum`、`pacman` 等等）也可以安装 Terraform。不过，极有可能不是最新的 Terraform 版本，各位读者朋友请在仔细斟酌后选择使用。

## 1.4.2 在 Windows 上安装 Terraform

除了与 Linux 类似的手动安装 Terraform 方式外，我们也可以通过 Chocolatey<sup>39</sup> 这个包管理器将 Terraform 安装到 Windows 上。如果你的 Windows 操作系统中还没有 Chocolatey 的话，那么不妨先按它的介绍文档<sup>40</sup>进行安装。Chocolatey 一旦安装完成之后，就能利用 `choco` 命令来安装 Terraform：

```
choco install terraform
```

---

<sup>39</sup><https://chocolatey.org/>

<sup>40</sup><https://chocolatey.org/docs/installation>

### 1.4.3 在 macOS 上安装 Terraform

在 macOS 上，我们同样可以按照 Linux 的方式来手动安装 Terraform。不过，比它更方便的安装方法是使用 Homebrew<sup>41</sup> 包管理器：

```
brew install terraform
```

### 1.4.4 利用 tfenv 管理多个 Terraform 版本

tfenv<sup>42</sup> 是受 rbenv<sup>43</sup> 启发编写而成的实用工具，利用它我们可以非常方便的管理操作系统中的多个 Terraform 版本。对于应用多版本的 Terraform 场景而言，tfenv 显得尤其有用。例如，在生产环境中我们使用 Terraform 的稳定版本来创建和管理基础设施。同时，我们使用 Terraform 的测试版本在其它环境试验新增特性。

目前，tfenv 支持 Linux、macOS 以及 Windows 操作系统，包括 64 位和 ARM 架构。

#### 1.4.4.1 安装 tfenv

以 Linux 和 macOS 操作系统为例，在安装 tfenv 之前，确保系统中具有 Git 源代码管理工具。然后，执行以下命令将 tfenv 从 GitHub 上克隆到本机的 ~/.tfenv 目录，并对 tfenv 的可执行文件建立相应的符号链接：

```
git clone https://github.com/tfutils/tfenv.git ~/.tfenv
sudo ln -s ~/.tfenv/bin/* /usr/local/bin
```

当上述命令执行完成后，接着运行 tfenv 命令以便验证其安装确实可用：

---

<sup>41</sup><https://brew.sh/>

<sup>42</sup><https://github.com/tfutils/tfenv>

<sup>43</sup><https://github.com/rbenv/rbenv>

```
tfenv
tfenv 1.0.1-8-gc3f5d5e
Usage: tfenv <command> [<options>]

Commands:
  install      Install a specific version of Terraform
  use          Switch a version to use
  uninstall    Uninstall a specific version of Terraform
  list         List all installed versions
  list-remote  List all installable versions
```

从该命令的输出信息中，我们既可了解 tfenv 的基本用法（包括安装、列出、切换、卸载等操作），也能查看其版本号（1.0.1）。

#### 1.4.4.2 利用 tfenv 安装 Terraform

首先，我们来看看 tfenv 支持安装哪些 Terraform 版本（执行 `tfenv list-remote` 命令）：

```
tfenv list-remote
0.12.9
0.12.8
0.12.7
...
0.12.0
0.12.0-rc1
0.12.0-beta2
0.12.0-beta1
...
0.2.0
0.1.1
0.1.0
```

从最早的 0.1.0 版本到最新的 0.12.9 版本，从 Beta 测试版本到 RC 候选版本，tfenv 对 Terraform 各个版本的支持真可谓一应俱全。

接着，作为试验，我们安装 Terraform 0.11.14 版本。为此，执行以下命令：

```
tfenv install 0.11.14
[INFO] Installing Terraform v0.11.14
[INFO] Downloading release tarball from https://releases.hashicorp.com/terraform/0.11.14/terraform_0.11.14_linux_amd64.zip
#####100.0%
[INFO] Downloading SHA hash file from https://releases.hashicorp.com/terraform/0.11.14/terraform_0.11.14_SHA256SUMS
  INFO Starting keybase.service.
tfenv: tfenv-install: [WARN] Unable to verify OpenPGP signature
unless logged into keybase and following hashicorp
Archive:  tfenv_download.g3jsEy/terraform_0.11.14_linux_amd64.zip
  inflating: /home/xiaodong/.tfenv/versions/0.11.14/terraform
[INFO] Installation of terraform v0.11.14 successful
[INFO] Switching to v0.11.14
[INFO] Switching completed
```

同时，我们也安装一个 Terraform 的最新版本。代替在 `install` 命令后面指定具体的版本号，我们直接使用 `latest` 参数：

```
tfenv install latest
```

在安装完毕后，tfenv 将自动切换到安装的版本。执行 `tfenv list` 命令可以查看当前系统中已经安装的所有 Terraform 版本：

```
tfenv list
* 0.12.9 (set by /home/xiaodong/.tfenv/version)
  0.11.14
```

其中，列表条目前面的 \*（星号）表示当前正在使用的 Terraform 版本。本例中为 Terraform 0.12.9。

#### 1.4.4.3 切换 Terraform 版本

假如你想要切换系统中已经安装的 Terraform 版本,那么可以使用 `tfenv` 的 `use` 命令,例如:

```
tfenv use 0.11.14
[INFO] Switching to v0.11.14
[INFO] Switching completed
```

在此,我们将 Terraform 从最新版本 (0.12.9) 切换到 0.11.14 版本。

再譬如:

```
tfenv use latest
[INFO] Switching to v0.12.9
[INFO] Switching completed
```

这次又还原到最新的 Terraform 版本 0.12.9。

需要指出的是,利用 `tfenv use` 命令切换 Terraform 的版本,其影响范围是全局性的。另一种十分有用的固定 Terraform 版本的方法是使用 `.terraform-version` 文件。假如我们的 Terraform 项目针对的是 0.11.14 系列版本,那么只需在该项目的工作目录执行以下命令即可将 Terraform 的版本设置为 0.11.14。而一旦离开该项目的工作目录,Terraform 仍旧使用全局性的版本。

```
echo 0.11.14 > .terraform-version
terraform version
Terraform v0.11.14

cd ..
terraform version
Terraform v0.12.9
```

#### 1.4.4.4 卸载 Terraform 版本

如果确认先前安装的某个 Terraform 版本不再需要使用，那么可以将其卸载，以便腾出占用的磁盘空间。例如，下列命令将卸载 Terraform 0.11.0 版本：

```
tfenv uninstall 0.11.0
[INFO] Uninstall Terraform v0.11.0
[INFO] Terraform v0.11.0 is successfully uninstalled
```

## 1.5 配置 Terraform 开发环境

虽然你可以使用任意的文本编辑器来编写 Terraform 代码，但是对 Terraform 支持更佳的文本编辑器将让你用起来更加得心应手，而且生产效率也会更高。在令人眼花缭乱的代码编辑器中，我们确实更偏爱 Visual Studio Code<sup>44</sup>（简称 VS Code）一些。

VS Code 是由微软推出的一款代码编辑器，它不仅开源，而且跨平台，能够在包括 Linux、macOS、Windows 等在内的操作系统上运行。这意味着，当你转换开发系统平台时，原有的开发环境依然可以保持不变。

VS Code 具有名为 IntelliSense 的智能感知系统，除了支持常见的语法高亮之外，IntelliSense 还提供更加智能的自动补全特性，它能够基于变量类型、函数定义以及导入模块进行自动补全。利用智能感知系统的帮助，将极大的提高我们的代码输入效率。

在 VS Code 中所包含的另一项好特性是内建 Git 命令支持。Git 是目前相当流行的分布式版本控制系统，无需额外的 Git 工具，我们直接就可以在 VS Code 中执行诸如暂存文件、比较差异、创建提交等各种 Git 操作，使用非常方便。

此外，VS Code 还包括十分强大的扩展和定制化能力。通过扩展，我们能够根据自身的想法和需求，实现 VS Code 原本所不具有的功能。顺便提一句，对于我们接下来将要介绍的 Terraform 支持即是利用 VS Code 的扩展而实现。

---

<sup>44</sup><https://code.visualstudio.com/>

### 1.5.1 Terraform 扩展简介

VS Code 的 Terraform 扩展<sup>45</sup>由 Mikael Olenfalk 所开发，该扩展主要为 Terraform 代码带来了下列功能：

- 针对 `.tf`、`.tfvars` 及 `.hcl` 等文件格式进行语法高亮；
- 自动补全 Terraform 资源类型和属性；
- 利用 `terraform fmt` 命令自动格式化 Terraform 代码；
- 通过 TFLint<sup>46</sup> 来 Lint 代码；
- 验证 Terraform 项目；
- 浏览文档符号。

### 1.5.2 安装 Terraform 扩展

在打开 VS Code 后，按 **Ctrl + p** 组合键启动 **Quick Open**（快速打开）。

接着，将以下命令粘贴到文本框中并按**回车键**，如图 1.2 所示：

```
ext install mauve.terraform
```

一旦安装完成，我们就可以在 VS Code 中使用 Terraform 扩展了。

---

<sup>45</sup><https://marketplace.visualstudio.com/items?itemName=mauve.terraform>

<sup>46</sup><https://github.com/wata727/tflint>



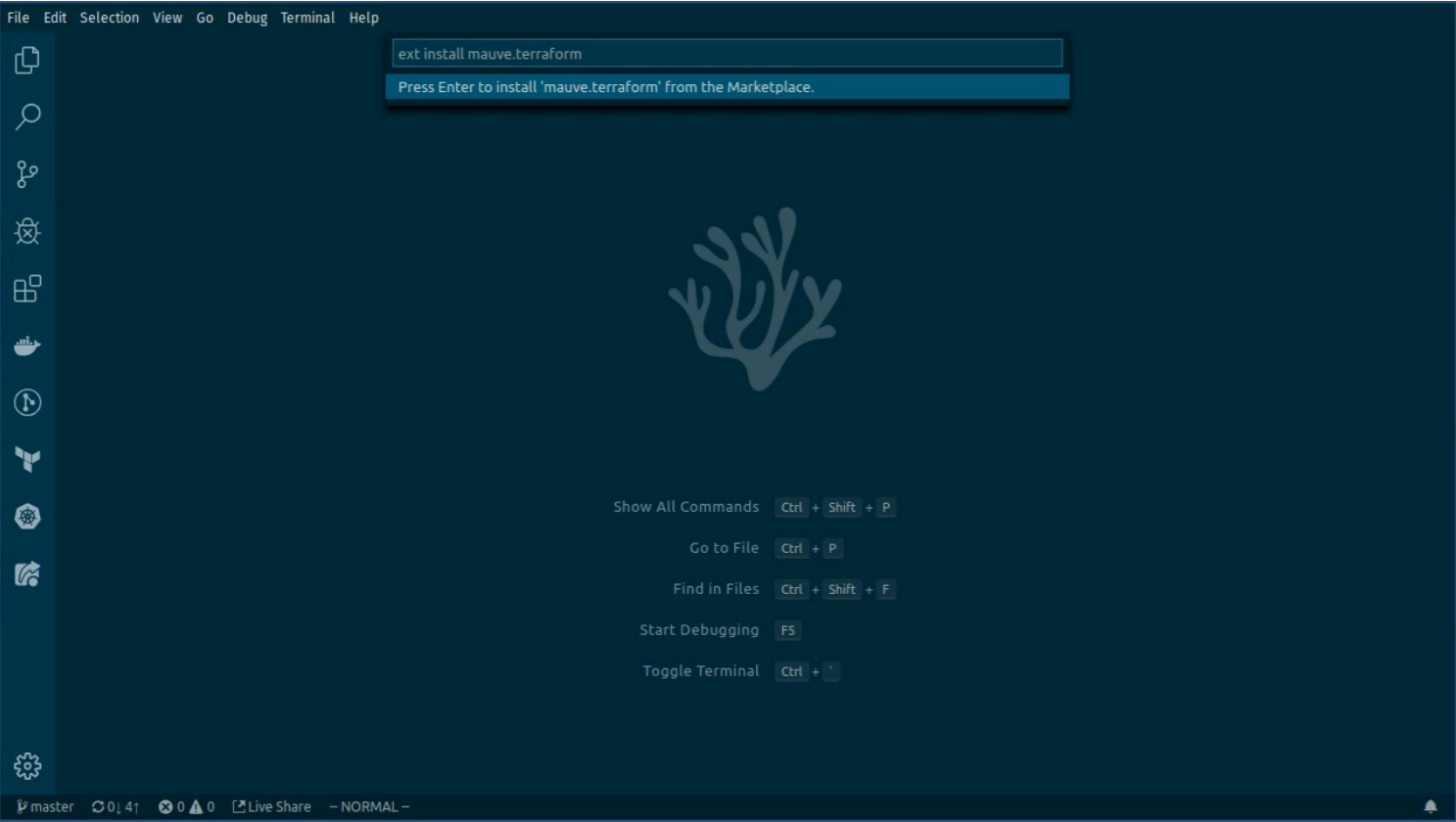


图 1.2: 在 VS Code 中安装 Terraform 扩展





## 第二章 入门篇

在上一章，我们既让 Terraform 在操作系统中安了家，同时也配置了 VS Code 开发环境。现在，是时候开始编写真正的 Terraform 代码来创建基础设施了。我们将先介绍 Terraform 配置语言的基本用法，然后再过渡到变量、函数以及模板等更加高级的用法。到本章结束时，我们相信，你应当能够利用 Terraform 舒服的创建、管理以及销毁常见的云基础设施。

### 2.1 创建云基础设施

我们将以在 AWS 云上创建基础设施为例进行说明。为此，我们需要准备一个 IAM (Identity and Access Management, 身份及访问管理) 用户，以便 Terraform 能够正常访问 AWS。

#### 2.1.1 准备 AWS 用户

如果你打算新建 AWS IAM 用户，那么只需按下列步骤操作即可：

1. 登录 AWS 控制台<sup>1</sup>。
2. 点击 **IAM** → **Users (用户)** → **Add user (添加用户)** 来创建一个新用户。
3. 按如图 2.1 所示设置用户名称及访问类型，在 **User name (用户名称)** 后输入你的 IAM 用户名（如 selfhosted），同时勾选 **Access type (访问类型)** 后面的 **Programmatic access (编程访问)**。

---

<sup>1</sup><https://console.aws.amazon.com/>

4. 选择 **Attach existing policies directly**（直接附加现有的策略），并按如图 2.2 所示设置权限，勾选 **AdministratorAccess**（管理员访问）。
5. 跳过 **Add tags**（添加标签）。
6. 确认无误后点击 **Create user**（创建用户）。

## Set user details

You can add multiple users at once with the same access type and permissions. [Learn more](#)

User name\*

selfhosted

[+ Add another user](#)

## Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in

Access type\*



**Programmatic access**

Enables an **access key ID** and **secret access key** tools.



**AWS Management Console access**

Enables a **password** that allows users to sign-in to the console.

图 2.1: 设置 IAM 用户名称及访问类型

最后，记录用户的 Access key ID（访问密钥 ID）和 Secret access key（访问密钥）备用，如图 2.3 所示。

### 2.1.2 配置 AWS 访问认证

在我们开始编写第一行 Terraform 代码之前，还有一件事情需要完成，那就是配置 AWS 用户的访问认证。还记得上一小节我们记录的访问密钥 ID 和访问密钥吗？怎么才能让 Terraform 使用它们呢？

因为 Terraform 在执行时，能够通过环境变量获取相应的信息。所以，我们只要在 Shell 中导出 `AWS_ACCESS_KEY_ID` 和 `AWS_SECRET_ACCESS_KEY` 这两个环境变量，Terraform 就能找到 AWS 用户的认证信息，从而实现对 AWS 的正常访问。

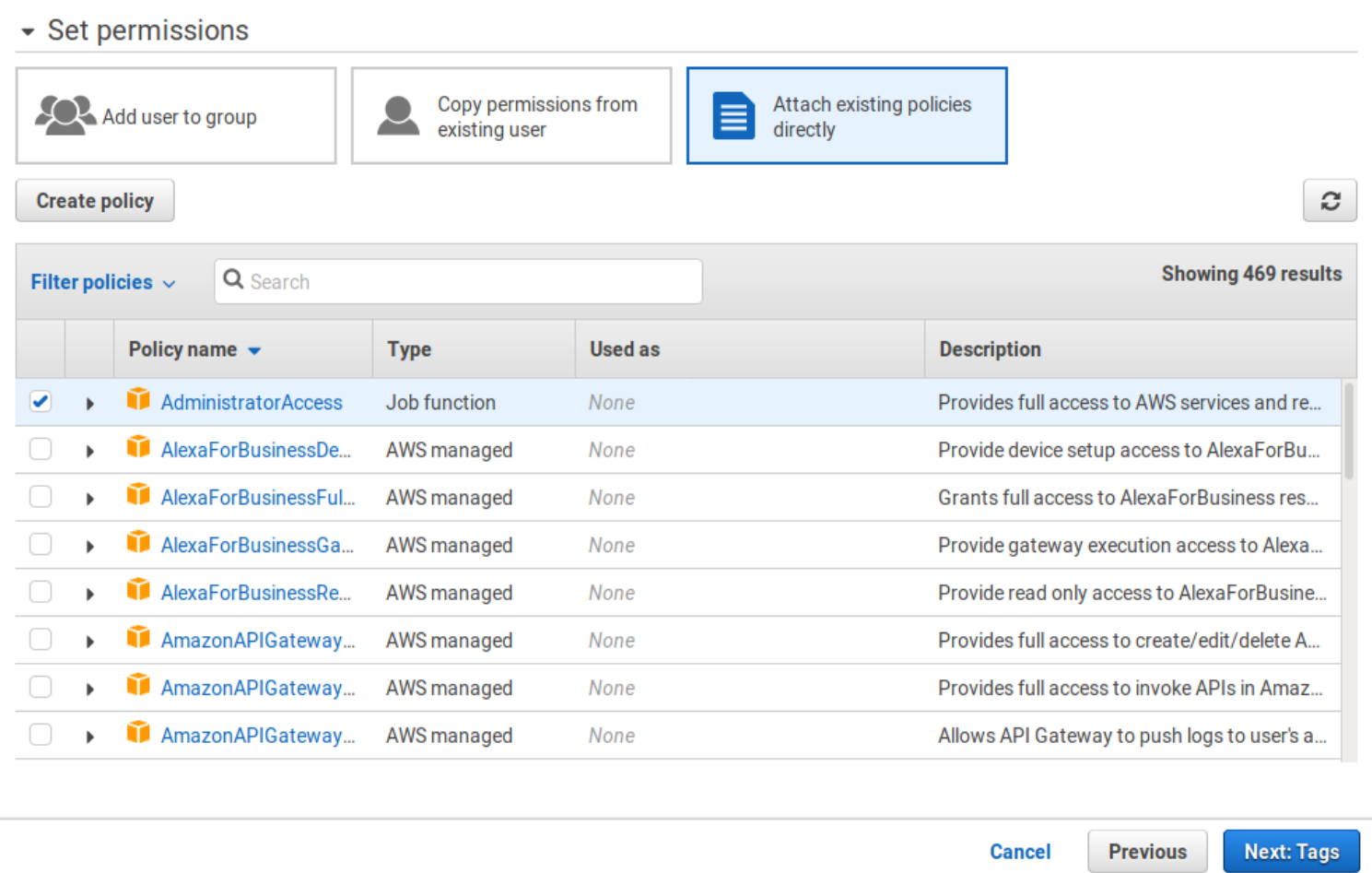


图 2.2: 设置 IAM 用户访问权限

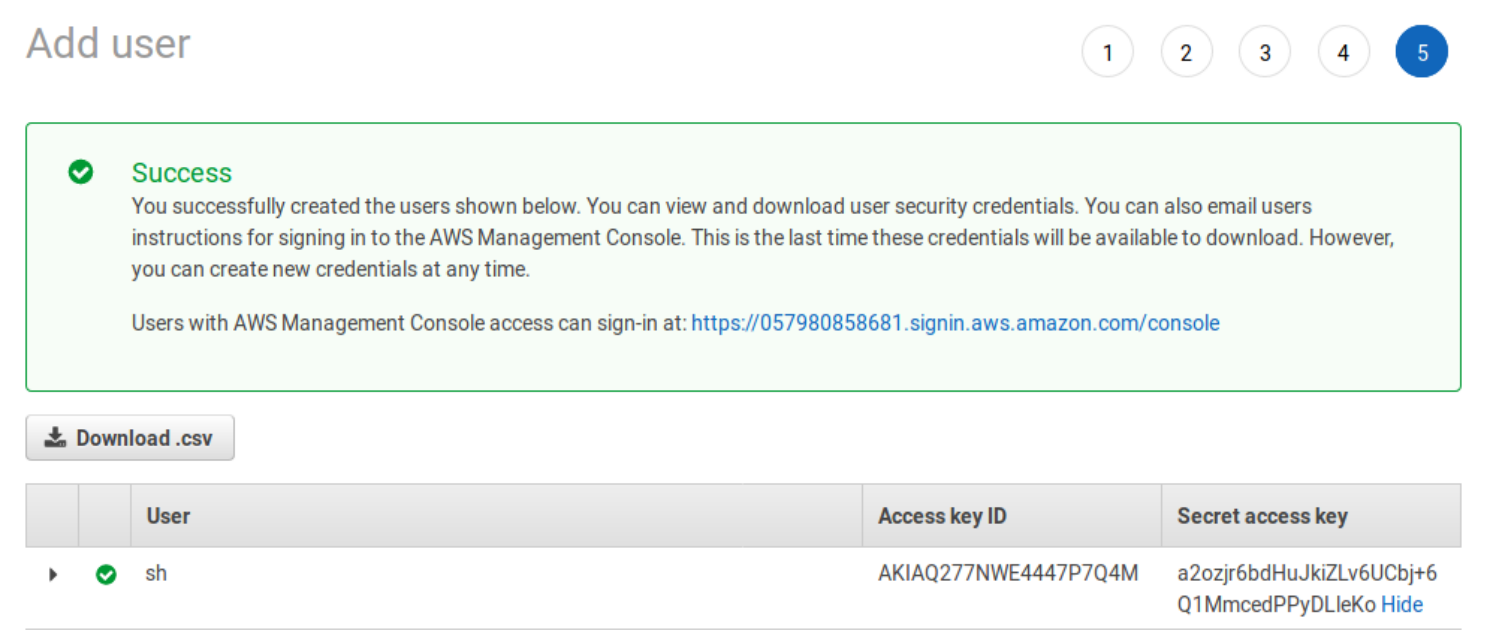


图 2.3: IAM 用户的访问密钥 ID 和访问密钥

```
export AWS_ACCESS_KEY_ID="AKIAQ277NWE4447P7Q4M"  
export AWS_SECRET_ACCESS_KEY="a2ozjr6bdHuJkiZLv6UCbj+6Q1MmcedPPy  
DLleKo"
```

另外，Terraform 也可以与 AWS CLI<sup>2</sup>（AWS 命令行客户端）共享用户的认证信息。使用 AWS CLI 要求 Linux 和 macOS 操作系统具有 Python 2.6.5 及以上版本，其安装可以通过 `pip` 命令完成。Windows 操作系统能够下载并执行 MSI 文件进行安装。

Linux 或 macOS 操作系统可执行以下命令安装 AWS CLI：

```
pip install awscli
```

在安装 AWS CLI 后，通过 `aws` 命令来配置用户的访问密钥。执行 `aws configure` 命令，按提示依次输入访问密钥 ID 和访问密钥，以及默认的区域（region）名称（如 `ap-northeast-1`，亚太）。

```
aws configure  
AWS Access Key ID [*****KVJJ]: AKIAQ277NWE4447P7Q4M  
AWS Secret Access Key [*****vcZ7]: a2ozjr6bdHuJkiZLv6U  
Cbj6Q1MmcedPPyDLleKo  
Default region name [ap-northeast-1]:  
Default output format [None]:
```

`aws` 命令最终把认证信息保存到 `~/.aws/credentials` 文件，其它配置则保存到 `~/.aws/config` 文件。

### 2.1.3 Terraform 初始化

我们创建 `learning-terraform` 目录来保存即将编写的 Terraform 代码，同时使用 Git 将该目录初始化为源代码仓库，以便进行版本控制：

---

<sup>2</sup><https://aws.amazon.com/cli/>

```
mkdir ~/learning-terraform
cd !$
git init
Initialized empty Git repository in /home/xiaodong/
learning-terraform/.git/
```

现在，启动 VS Code 代码编辑器，点击 **File (文件) → Open Folder... (打开文件夹)** 菜单条目，然后在打开的对话框中导航到 `learning-terraform` 目录，并单击 **OK** 按钮确定。

接着，在 VS Code 窗口左边的 **Explorer (资源管理器)** 中，点击 **New File (新建文件)** 按钮，同时在文本框中输入 `main.tf` 文件名。完成后，按回车键确认。

然后，在代码编辑器中输入下列内容：

```
provider "aws" {
    region = "ap-northeast-1"
}
```

这段 Terraform 代码到底是什么意思呢？第一行开头的关键字 **provider** 表明该代码块为 `provider`（供应商，如：云供应商、服务供应商等）块。由于每一个云供应商和服务供应商所提供的 API 都不相同，所以 Terraform 为了支持它们，采用基于插件化的架构，并通过各自独立的 **provider** 实现。

我们想要创建和管理的云基础设施资源类型最终确定了我们选择哪一种供应商。例如，在本例中，我们打算创建 AWS 云基础设施，所以就选用 AWS `provider`。Terraform 目前具有数百个 `provider`，有一些是官方提供维护支持<sup>3</sup>，而另一些则由社区会员贡献<sup>4</sup>。

正如你所见，紧接着 **provider** 关键字右边双引号中的内容为该供应商的名称。由于我们使用的云供应商是 AWS，故而此处以 `aws` 命名。假如你要使用阿里云供应商，那么可以将 `aws` 替换成 `alicloud`。

在第一行末尾的左花括号表示 **provider** 块的开始，而最后一行的右花括号则表示块的结束。

---

<sup>3</sup><https://www.terraform.io/docs/providers/index.html>

<sup>4</sup><https://www.terraform.io/docs/providers/type/community-index.html>



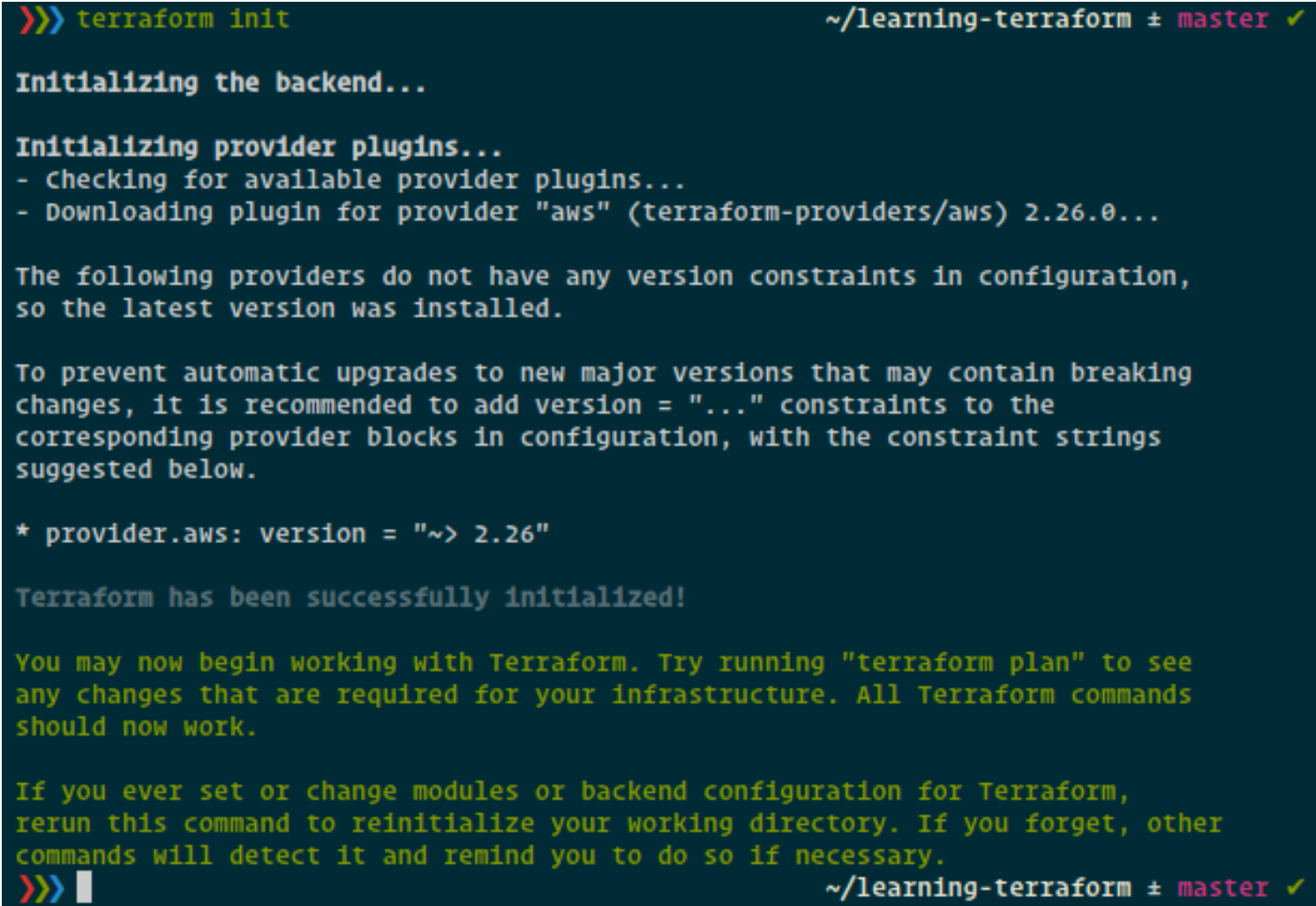
花括号中间的内容是 **provider** 块的主体，其格式为键值对 (key => value) 的参数声明。本例中的 **region** 为 AWS 云数据中心所在的地理区域。我们将其设置为 **ap-northeast-1**，也就是东京。

一旦理解了 **provider** 的用途，就不难明白每当我们使用新的供应商时执行 Terraform 初始化的必要性了。

导航到 `~/learning-terraform` 目录，并执行 `terraform init` 命令来初始化 Terraform：

```
cd ~/learning-terraform
terraform init
```

Terraform 在执行初始化时，将下载 **provider** 插件 (terraform-providers/aws)，并提供一些有关限制 **provider** 版本以及接下来可运行的 Terraform 子命令的建议。该命令输出中的消息 “Terraform has been successfully initialized!” 说明 Terraform 已经成功初始化，如图 2.4 所示。



```
>>> terraform init                                     ~/learning-terraform ± master ✓

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (terraform-providers/aws) 2.26.0...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 2.26"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
>>> |                                                    ~/learning-terraform ± master ✓
```

图 2.4: 执行 Terraform 初始化

初学 Terraform 时，你也许会感到迷惑。究竟在哪些情况下需要我们执行 Terraform 初始化呢？除了供应商外，如果配置了 Terraform 状态文件的存储后端，或者使用了 Terraform 模块，那么你也必须重新执行 Terraform 的初始化过程。

最后，我们把 `.terraform` 目录添加到 `.gitignore` 文件。该目录用来保存 Terraform 插件和模块信息，并不需要进行版本控制。现在，完成我们的第一个 Git 提交：

```
echo .terraform > .gitignore
git add .
git commit -m "first commit"
[master (root-commit) 69c4478] first commit
 2 files changed, 4 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 main.tf
```

#### 2.1.4 创建第一个云基础设施资源

接下来，我们继续往 `main.tf` 文件添加 Terraform 代码，以便创建我们的第一个 AWS 云基础设施资源——EC2（Elastic Compute Cloud）实例。

```
provider "aws" {
    region = "ap-northeast-1"
}

resource "aws_instance" "web" {
    ami           = "ami-a3ed72c5"
    instance_type = "t2.micro"
}
```

除开 `provider` 块，这次我们着重考察 `resource`（资源）块。`resource` 块用来声明我们想要创建或更改的基础设施资源对象，比如计算实例、对象存储、虚拟网络、数据库实例等等。

在 `resource` 关键字右边双引号中的内容是该资源的类型，本例中为 `aws_instance`，即 AWS EC2 实例。资源类型通常将 `provider` 的名称



(**aws**) 作为前缀，并通过 `_` (下划线) 拼接在一起。跟着资源类型的是该资源的名称 (**web**)。虽然资源名称可以任意指定，但是我们建议使用具有描述性的文本命名。当别人阅读我们的 Terraform 代码时，就会更容易明白我们的意图。另外，资源名称在同一个模块中应当是唯一的，Terraform 不允许有相同的资源名称存在。

Terraform 把资源类型和资源名称连起来作为该资源的唯一标识，以便在需要时对该资源加以引用。例如，本例中的资源标识为 **aws\_instance.web**，资源类型与资源名称之间使用 `.` (英文句点) 拼接。

跟 **provider** 块一样，花括号表示 **resource** 块的主体。其中同样也是用于声明键值对的参数。在本例中，我们主要使用了以下两个配置参数：

1. **ami**: AMI 即 Amazon Machine Image，它是 AWS 用来启动 EC2 实例的操作系统镜像。在此，我们指定 AMI 的 ID，也就是 Debian GNU/Linux 系统。值得注意的是，AMI 跟 AWS 数据中心的地理区域相关，换句话说，不同的区域具有不同的 AMI ID。这里的 AMI ID (**ami-a3ed72c5** 特定存在于 **ap-northeast-1** 区域中)。
2. **instance\_type**: 该参数表示将要启动的 EC2 实例的类型。实例类型表明我们能够使用哪种配置的服务器，如 CPU 有多少、内存有多大、网络带宽占好多等等。本例使用 **t2.micro** 实例类型，该类型属于免费层级。如果你打算使用其它实例类型，那么需要时刻关注 AWS 的账单，以免产生高额费用。

**aws\_instance** 资源还包括许多其它的配置参数<sup>5</sup>，只有这两个配置参数必须提供。在后续章节中，我们将继续介绍该资源的其它配置参数。

#### 2.1.4.1 执行 Terraform 计划

现在，让我们来看看 Terraform 创建基础设施的执行计划。回到命令行，运行以下命令：

```
terraform plan
```

执行 **terraform plan** 命令的输出结果如图 2.5 和 2.6 所示。

```

>>> terraform plan                                     ~/learning-terraform ± master ✓
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

-----

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

# aws_instance.web will be created
+ resource "aws_instance" "web" {
  + ami                  = "ami-a3ed72c5"
  + arn                  = (known after apply)
  + associate_public_ip_address = (known after apply)
  + availability_zone     = (known after apply)
  + cpu_core_count        = (known after apply)
  + cpu_threads_per_core   = (known after apply)
  + get_password_data      = false
  + host_id               = (known after apply)
  + id                    = (known after apply)
  + instance_state         = (known after apply)
  + instance_type          = "t2.micro"
  + ipv6_address_count     = (known after apply)
  + ipv6_addresses        = (known after apply)
  + key_name               = (known after apply)
  + network_interface_id   = (known after apply)
  + password_data          = (known after apply)
  + placement_group        = (known after apply)
  + primary_network_interface_id = (known after apply)
  + private_dns            = (known after apply)
  + private_ip             = (known after apply)
  + public_dns             = (known after apply)
  + public_ip              = (known after apply)
  + security_groups        = (known after apply)
  + source_dest_check       = true
  + subnet_id              = (known after apply)
  + tenancy                = (known after apply)
  + volume_tags            = (known after apply)
  + vpc_security_group_ids = (known after apply)

```

图 2.5: 执行 Terraform 计划

从该计划中，我们可以看到 Terraform 将要执行的操作，也就是创建 **aws\_instance.web** 资源。该条目前面的 +（加号）说明，本次计划为全新创建资源。除了 + 之外，Terraform 也使用其它符号进行说明：

- -（减号）：销毁资源
- -/+：先销毁资源，然后再创建资源
- ~（波浪号）：变更资源

<sup>5</sup><https://www.terraform.io/docs/providers/aws/r/instance.html>

```

+ ebs_block_device {
+   delete_on_termination = (known after apply)
+   device_name           = (known after apply)
+   encrypted             = (known after apply)
+   iops                   = (known after apply)
+   kms_key_id            = (known after apply)
+   snapshot_id           = (known after apply)
+   volume_id             = (known after apply)
+   volume_size           = (known after apply)
+   volume_type           = (known after apply)
+ }

+ ephemeral_block_device {
+   device_name = (known after apply)
+   no_device   = (known after apply)
+   virtual_name = (known after apply)
+ }

+ network_interface {
+   delete_on_termination = (known after apply)
+   device_index           = (known after apply)
+   network_interface_id   = (known after apply)
+ }

+ root_block_device {
+   delete_on_termination = (known after apply)
+   encrypted             = (known after apply)
+   iops                   = (known after apply)
+   kms_key_id            = (known after apply)
+   volume_id             = (known after apply)
+   volume_size           = (known after apply)
+   volume_type           = (known after apply)
+ }
}

Plan: 1 to add, 0 to change, 0 to destroy.

-----

Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.

>>> ~/learning-terraform ± master ✓

```

图 2.6: 执行 Terraform 计划

这些符号跟 Git 版本控制系统所用的相似。

同时，Terraform 还列出 `aws_instance.web` 资源的所有配置参数。其中，有些为我们先前指定的内容（如 `ami` 和 `instance_type`），有些使用其默认值（如 `get_password_data`、`source_dest_check` 等），而更多的参数值为 `known after apply`，这表示目前 Terraform 尚不能了解该参数的值，需要在创建该资源后才能明朗化。

另外，Terraform 在执行计划的最后对要创建、变更及销毁的资源数量进行统计。此刻，只有一个资源待创建（1 to add）。

### 2.1.4.2 应用 Terraform 代码

从 Terraform 的执行计划看起来一切尚好，下面我们就来应用 Terraform 代码，让 Terraform 自动为我们创建 AWS EC2 实例。为此，我们执行 `terraform apply` 命令：

```
terraform apply
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

# aws_instance.web will be created
+ resource "aws_instance" "web" {
    + ami                = "ami-a3ed72c5"
    + arn                = (known after apply)
    + associate_public_ip_address = (known after apply)
    + availability_zone   = (known after apply)
    ...
}
```

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.

Enter a value:

Terraform 在生成执行计划后，同时提醒我们是否需要执行这些操作，输入 `yes` 并按回车键即可。如果你想让 Terraform 跳过交互环节直接予以执行，那么可以在上面的命令后指定 `-auto-approve` 选项。Terraform 立即开始为我们创建基础设施，其输出结果如下：



```
aws_instance.web: Creating...
aws_instance.web: Still creating... [10s elapsed]
aws_instance.web: Still creating... [20s elapsed]
aws_instance.web: Still creating... [30s elapsed]
aws_instance.web: Still creating... [40s elapsed]
aws_instance.web: Creation complete after 40s
    [id=i-0ba4c374050b0eeeb]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Terraform 告诉我们，在 40 秒（噢，真快！）后，AWS EC2 实例创建完毕，且该实例的 ID 为 i-0ba4c374050b0eeeb。最后一行的统计结果也表明，本次 Terraform 总共为我们创建了一个资源。

在网络浏览器中打开 AWS 的 Web 控制台，我们立刻可以看到，Terraform 创建的 EC2 实例已经处于运行状态，如图 2.7 所示。

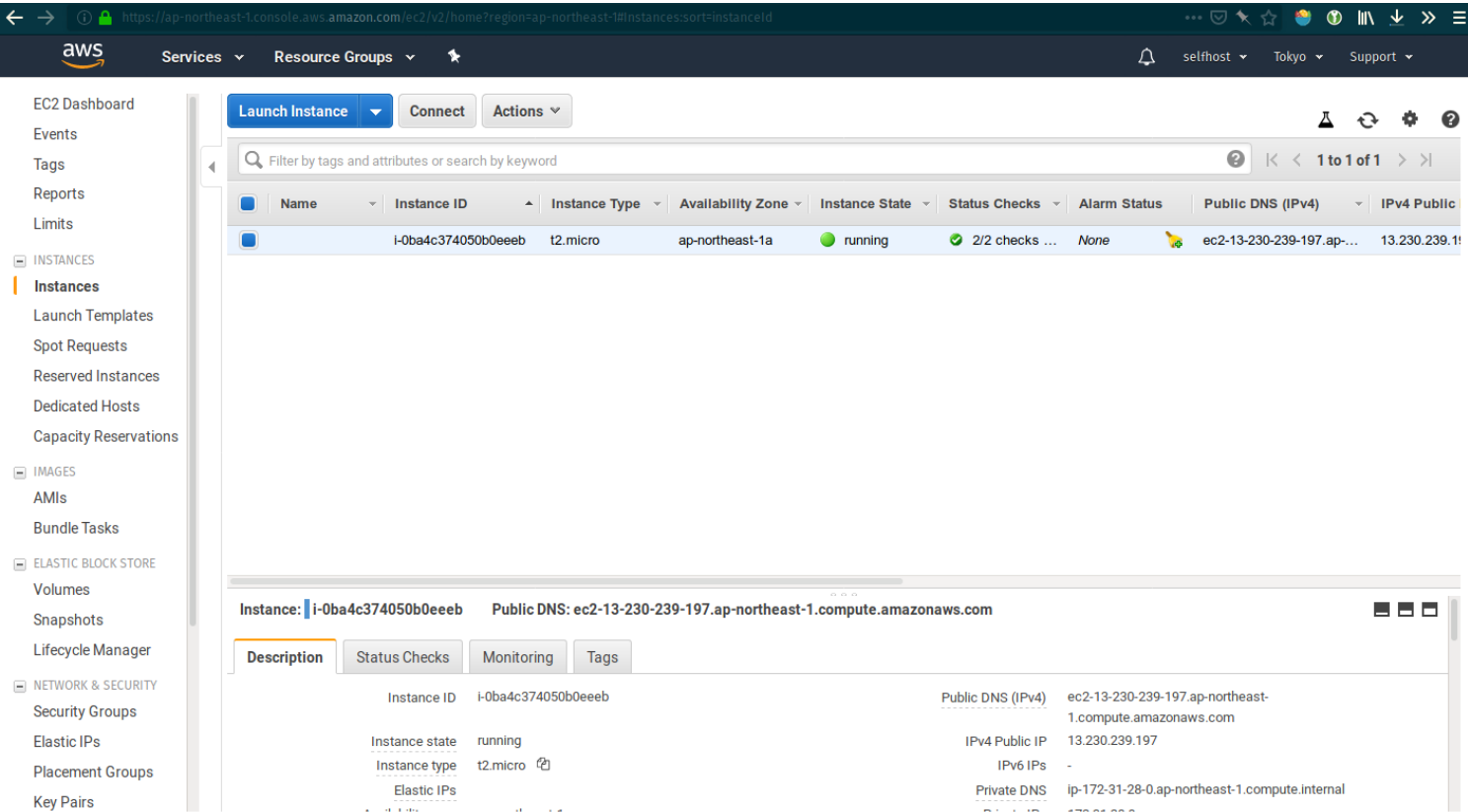


图 2.7: AWS EC2 Web 控制台

既然第一个基础设施资源 AWS EC2 实例已经创建成功，那么我们把新增的 Terraform 代码提交到 Git 仓库。

```
git add main.tf
git commit -m "add ec2 instance"
[master 559bd88] add ec2 instance
1 files changed, 89 insertions(+), 1 deletion(-)
```

### 2.1.4.3 Terraform 状态

在上一小节，我们提交 Terraform 代码时，你应当已经注意到我们只添加了 `main.tf` 文件。如果你足够仔细，那么可能会发现在当前目录中还新增了一个名为 `terraform.tfstate` 的文件。该文件在我们执行 `terraform apply` 命令后产生，它用来保存 Terraform 所管理的基础设施的状态。

`terraform.tfstate` 文件为 JSON 格式，如图 2.8 所示。这个文件开头的部分为 Terraform 状态的元数据，如状态文件版本（4）、所用的 Terraform 版本（0.12.7）、序号（1）等等。其重头部分主要是我们所创建的 AWS EC2 实例资源，包括模式（managed，托管）、类型（aws\_instance）、名称（web）、属性（如 AMI ID）等。

因为 JSON 格式对人类并不友好，其可读性较差。代替直接查看 Terraform 状态文件，我们也可以通过 `terraform show` 命令来检查基础设施的状态。`terraform show` 命令以人类更可读的输出格式显示 Terraform 状态。

```
terraform show
# aws_instance.web:
resource "aws_instance" "web" {
  ami                    = "ami-a3ed72c5"
  arn                    = "arn:aws:ec2:ap-northeast-1:
057980858681:instance/i-04ef5392a47ba9065"
  associate_public_ip_address = true
  availability_zone        = "ap-northeast-1a"
  cpu_core_count           = 1
  cpu_threads_per_core     = 1
  disable_api_termination  = false
  ebs_optimized            = false
```

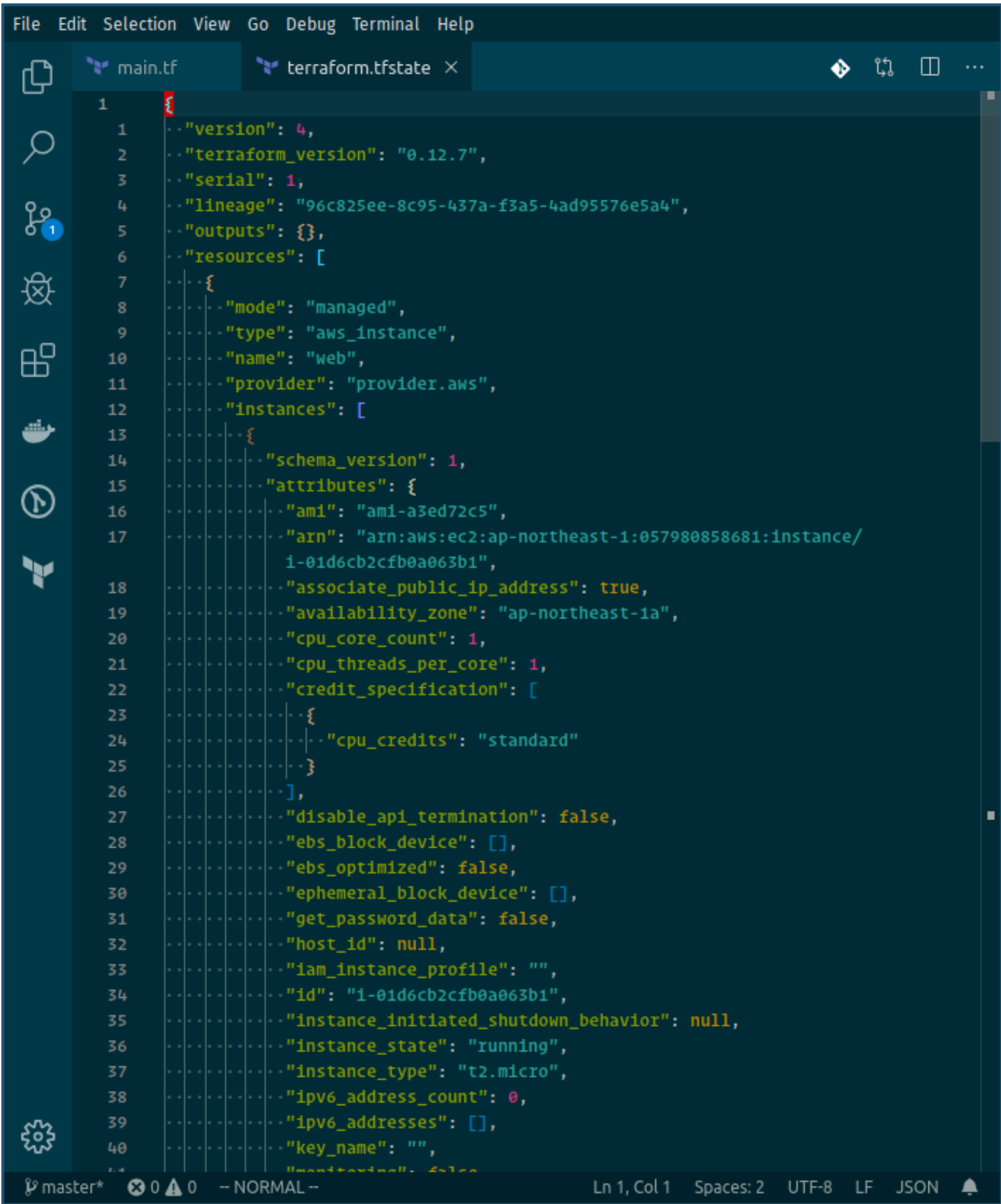


图 2.8: terraform.tfstate 文件的内容

```
get_password_data      = false
id                     = "i-04ef5392a47ba9065"
instance_state         = "running"
instance_type          = "t2.micro"
ipv6_address_count     = 0
ipv6_addresses         = []
monitoring             = false
primary_network_interface_id = "eni-02f89359b79c6ec9a"
...
```

对 Terraform 而言, `terraform.tfstate` 文件非常重要。只有通过该文件, Terraform 才能知晓哪些基础设施是由它自身所管理。有鉴于此, 我们不应该手动编辑此文件, 否则可能会导致意想不到的结果。如果你真的想要修改 Terraform 的状态, 那么我们建议使用 `terraform state` 命令<sup>6</sup>。

因为 `terraform.tfstate` 使用明文存储内容, 其中可能包含敏感信息, 所以我们不将其加入 Git 仓库:

```
echo 'terraform.tfstate*' >> .gitignore
git commit -am "update .gitignore"
```

### 2.1.5 HCL 配置语言基础

当 Terraform 在当前目录中运行时, 它将载入该目录中所有的配置文件。除配置文件之外的其它文件, Terraform 则予以忽略。另外, Terraform 不会递归加载子目录中的任何配置文件。Terraform 按字母顺序载入目录中的每一个配置文件, 并最终将它们的内容合并为统一的配置。

一般而言, Terraform 的配置文件只是常规的文本文件, 其后缀要么为 `.tf`, 要么是 `.tf.json`。其中, 带 `.tf` 后缀的文件是 Terraform 的原生配置文件格式, 而 `.tf.json` 文件为 JSON 配置文件格式。

或许你会对此产生疑惑, Terraform 为什么要支持两种不同类型的配置文件格

---

<sup>6</sup><https://www.terraform.io/docs/commands/state/index.html>



式呢？简而言之，它们适合不同的目标受众，前者对人类编写和阅读比较友好，而后者则主要用于机器读写。

后缀为 `.tf` 的文件格式也称作 HCL (HashiCorp Configuration Language, HashiCorp 配置语言) 配置语言。它是一种人类更可读的配置语言，我们推荐各位读者优先使用它。在本书中的所有 Terraform 代码示例都以 HCL 配置语言呈现。

现在，让我们来扩展一点先前的例子，以便能够通过 SSH 登录 AWS EC2 实例。

```
/*
 * 创建 AWS EC2 实例
 * 允许通过 SSH 登录
 */
provider "aws" {
    region = "ap-northeast-1"
}

# 创建 AWS EC2 实例资源
resource "aws_instance" "web" {
    ami          = "ami-a3ed72c5"
    instance_type = "t2.micro"
    key_name      = aws_key_pair.ssh.key_name
}

# 添加 SSH 登录密钥
resource "aws_key_pair" "ssh" {
    key_name      = "admin"
    public_key    = "ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA1u/s1R3zGuK1
p1xoVwVqd0IjYTYLTIITCL5a/b2tfPunwUKNlKqq9wzdPhHm1Ger6yRQvavPTn
noTQZruAB/a4ZHqi2uH9GJRk7IkPHPaKFnNZz9F2mUiG06kec0AqwiZs2+LRNR
9qRBIC7BK3K7YU1J3trVlcpwZGxKTfMc0dGGvFrR90BSLklihTMsUWJmkfNjYa
r4jW0LBm4Ng9dxm3VS8GZHJtF7TFJmLFhF5gjUdn9gszvsDevwy7B5LGE+1DXd
M10VRQ8/KM6jD4FxtvL/qwaMNH/8LaVjdtV+Hb7qqpIIrrF9f6oXvNjVlm1X8B
2GE7uPPfi4jYGv8G2R5Q== for mt host"
```

```

}

// 开放 22 端口，允许 SSH 登录
resource "aws_security_group_rule" "ssh" {
  type           = "ingress"
  from_port      = 22
  to_port        = 22
  protocol       = "tcp"
  cidr_blocks    = ["0.0.0.0/0"]
  security_group_id = "sg-a361e5da"
}

```

首先，在这段 Terraform 代码中，我们在开头添加了注释，从而说明它的意图。该注释由 `/*`（开始）和 `*/`（结束）表示，而且支持多行。另外，行首的 `#` 和 `//` 也表示引起注释，不过它们只支持单行。

其次，我们添加了两个新的 `resource`（AWS 资源）块。在每一个块中都包含了若干参数，参数由参数名称和参数值组成。参数值可以是任意表达式，它既包含字符串（如 “admin”）、数字（如 22）、布尔（`true` 或 `false`）、列表（如 `["0.0.0.0/0"]`，由 `[]` 包围）、映射（如 `{ap-northeast-1 = "ami-a3ed72c5"}`，由 `{}` 包围）等类型的简单表达式，也包括算术运算、条件求值、函数等复杂的表达式。块与参数是 HCL 配置语言的基本构件。

1. **aws\_key\_pair**：该资源用于添加 SSH 公钥。由此，我们就能通过 SSH 免密码登录 AWS EC2 实例。其中，**key\_name** 参数将密钥名称设置为 `admin`，而 **public\_key** 参数指定 SSH 公钥的内容。如果你还没有自己的 SSH 公钥，那么不妨参考 GitHub 上面的文章<sup>7</sup>生成一个。
2. **aws\_security\_group\_rule**：我们通过这个资源设置一条防火墙规则，以便开放 22 端口，从而允许从 SSH 登录 AWS EC2 实例。

**type** 参数指定防火墙规则的类型，`ingress` 表示入站。如果是出站，则将其设置为 `egress`。

**from\_port** 参数和 **to\_port** 参数表示要开放的端口的起始和结束范围。在本例中，由于我们想要进行 SSH 登录，所以只需将首尾端口设为 22 即可。

---

<sup>7</sup><https://help.github.com/en/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>