云资源自动化 for Terraform

工具指南

腾讯云

文档目录

工具指南
配置指南
Variables

最近更新时间：2023-09-21 20:57:31

输入变量

- 通过输入变量，可自定义 Terraform 模块，且无需修改模块本身的源代码。通过此特性，您可在不同的 Terraform 配置间共享模块，使模块可组合和可重用。
- 输入变量支持动态传入。例如，在创建或修改基础设施时传入值、在代码中定义 Provider 时用变量替代硬编码的访问密钥、由创建基础设施的用户决定创建所需尺寸的主机等。
- 您可将一组 Terraform 代码理解为一个函数，则输入变量即为函数的入参。

定义输入变量

输入变量通过 `variable` 块进行定义。例如：

```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

`variable` 关键字后为变量名。在一个 Terraform 模块（同一个文件夹中的所有 Terraform 代码文件，不包含子文件夹）中变量名必须唯一。在代码中可以通过 `var.<NAME>` 的方式引用变量的值。

`variable` 块可以使用下列的可选参数进行声明：

- `default`：指定输入变量默认值。
- `type`：指定输入变量只能被赋予特定的值。
- `description`： 指定输入变量的描述。
- `validation`：指定输入变量的验证规则。
- `sensitive`：在配置中使用变量时限制 Terraform UI 输出。
- `nullable`：指定输入变量是否可以为 `null` 。

类型

输入变量块中通过 type 定义类型：

- 基本类型： `string` 、 `number` 、 `bool`
- 复合类型： `list(<TYPE>)` 、 `set(<TYPE>)` 、 `map(<TYPE>)` 、 `object({<ATTR NAME> = <TYPE>, ... })` 、 `tuple([<TYPE>, ...])`

描述

简要描述每个变量的用途。例如：

```
variable "image_id" {
  type        = string
  description = "The id of the machine image (AMI) to use for the server."
}
```

自定义校验规则

Terraform 0.13.0 前，只能用类型约束确保输入参数的类型正确。Terraform 0.13.0 已引入输入变量自定义校验规则。例如：

```
variable "image_id" {
  type        = string
  description = "The id of the machine image (AMI) to use for the server."

  validation {
    condition     = length(var.image_id) > 4 && substr(var.image_id, 0, 4) == "ami-"
    error_message = "The image_id value must be a valid AMI id, starting with \"ami-
\"."
  }
}
```

其中，condition 参数为 bool 类型参数，可通过表达式来判断输入变量是否合法。当 contidion 为 true 时输入变量合法，否则不合法。condition 表达式中只能通过 `var.<变量名称>` 引用当前定义的变量，并且它的计算不能产生错误。若表达式的计算产生错误是输入变量验证的一种判定手段，那么可以使用 can 函数来判定表达式的执行是否抛错。例如：

```
variable "image_id" {
  type        = string
  description = "The id of the machine image (AMI) to use for the server."

  validation {
    # regex(...) fails if it cannot find a match
    condition     = can(regex("^ami-", var.image_id))
    error_message = "The image_id value must be a valid AMI id, starting with \"ami-
\"."
  }
}
```

上述示例中，若输入的 image_id 不符合正则表达式的要求，那么 regex 函数调用会抛出一个错误，并会被 can 函数捕获，输出 false。condition 表达式若为 false，Terraform 会返回 error_message 中定义的错误信息。error_message 应该完整描述输入变量校验失败的原因，以及输入变量的合法约束条件。

使用输入变量

输入变量可以通过 `var.<变量名称>` 的形式访问，只能在声明该变量的模块内访问。例如：

```
resource "tencentclould_instance" "example" {
  instance_type = "t2.micro"
  ami           = var.image_id
}
```

输入变量赋值

命令行参数

在命令行上指定单个变量，需要在执行 `terraform plan` 和 `terraform apply` 命令时使用 `-var` 选项。例如：

```
terraform apply -var="image_id=ami-abc123"
terraform apply -var='image_id_list=["ami-abc123","ami-def456"]' -
var="instance_type=t2.micro"
terraform apply -var='image_id_map={"us-east-1":"ami-abc123","us-east-2":"ami-
def456"}'
```

参数文件

设置大量变量时，推荐在变量参数文件中指定它们的值（文件名以 .tfvars 或 .tfvars.json 结尾），并在命令行上使用 -var-file 指定该文件。例如：

```
terraform apply -var-file="testing.tfvars"
```

定义参数文件使用与 Terraform 语言文件相同的基本语法，但仅包含变量名分配。例如：

```
image_id = "ami-abc123"
availability_zone_names = [
  "us-east-1a",
  "us-west-1c",
]
```

Terraform 会自动加载许多变量定义文件：

- 文件名为 `terraform.tfvars` 或 `terraform.tfvars.json` 的文件。
- 文件名称以 `.auto.tfvars` 或 `.auto.tfvars.json` 结尾的文件。
- 对于 `.json` 结尾的文件，需要使用 JSON 语法定义。例如：

```
{
  "image_id": "ami-abc123",
  "availability_zone_names": ["us-west-1a", "us-west-1c"]
}
```

环境变量

通过定义 `TF_VAR_` 为前缀的环境变量来指定输入变量。例如：

```
export TF_VAR_image_id=ami-abc123
export TF_VAR_availability_zone_names='["us-west-1b","us-west-1d"]'
```

输入变量优先级

- 若同时使用多种赋值方式时，同一个变量可能会被赋值多次。Terraform会使用新值覆盖旧值。Terraform 加载变量值的顺序是：
  - 环境变量。
  - `terraform.tfvars` 文件（若存在）。
  - `terraform.tfvars.json` 文件（若存在）。
  - 所有的 `.auto.tfvars` 或 `.auto.tfvars.json` 文件，以字母顺序排序处理。
  - 通过 -var 或 -var-file 命令行参数传递的输入变量，按照在命令行参数中定义的顺序加载。

- 若多种赋值方式均未能成功对变量赋值，那么 Terraform 会尝试使用默认值。对于没有定义默认值的变量，Terraform 会采用交互界面方式要求用户输入。对于某些 Terraform 命令，如果执行时带有 -input=false 参数禁用了交互界面传值方式，则会报错。

输出变量

输出变量使有关基础结构的信息在命令行上可用，并且可以供其他 Terraform 配置使用。输出值类似于传统编程语言中的返回值。

应用场景

- 子模块可以使用输出变量向父模块传递其资源属性。
- 根模块可以使用输出变量在运行 `terraform apply` 后在 CLI 输出中打印某些值。
- 使用远程状态时，其他配置可以通过terraform_remote_state数据源访问根模块输出。

定义输出变量

输出变量通过 `output` 块进行声明。例如：

```
output "instance_ip_addr" {
  value = tencentclould_instance.server.private_ip
}
```

可选参数

- **description**

  指定输出值的描述信息。例如：

  ```
  output "instance_ip_addr" {
    value       = tencentclould_instance.server.private_ip
    description = "The private IP address of the main server instance."
  }
  ```

- **sensitive**

  在执行 `terraform plan` 和 `terraform apply` 时，在CLI中隐藏输出变量的值。

- **depends_on**

  Terraform 会解析代码所定义的各种 data、resource 以及它们之间的依赖关系。例如，创建虚拟机所使用的 `image_id` 参数是通过 `data` 查询得到的，那么虚拟机实例就依赖于这个镜像的 `data` 。Terraform 会首先创建 `data` ，得到查询结果后，再创建虚拟机 `resource` 。一般来说， `data` 及 `resource` 之间的创建顺序是由 Terraform 自动计算的，不需要代码的编写者显式指定。但有时有些依赖关系无法通过分析代码得出，此时可以在代码中通过 `depends_on` 显式声明依赖关系。例如：

```
output "instance_ip_addr" {
  value       = tencentclould_instance.server.private_ip
  description = "The private IP address of the main server instance."

  depends_on = [
    # Security group rule must be created before this IP address could
    # actually be used, otherwise the services will be unreachable.
    tencentclould_security_group_rule.local_access,
  ]
}
```

本地变量

若需使用较复杂的表达式计算某一个值，并且反复使用时，则可赋予复杂表达式一个局部值后，再反复引用该局部值。您可将输入变量理解为函数的入参，输出值为函数的返回值，则局部值相当于函数内定义的局部变量。

本地变量的定义

本地变量通过 `locals` 块进行声明。例如：

```
locals {
  service_name = "forum"
  owner        = "Community Team"
}
```

同时，本地变量不限于文字常量，也包括本模块的其他变量（变量、资源属性或其他局部值），以便转换或组合使用它们。例如：

```
locals {
  # Ids for multiple sets of EC2 instances, merged together
  instance_ids = concat(tencentclould_instance.blue.*.id,
tencentclould_instance.green.*.id)
}

locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Service = local.service_name
    Owner   = local.owner
  }
}
```

使用本地变量

通过 `local.<NAME>` 表达式引用本地变量。例如：

```
resource "tencentclould_instance" "example" {
  # ...

  tags = local.common_tags
}
```

Resource

最近更新时间：2023-09-22 10:03:51

资源是 Terraform 最重要的组成部分。资源通过 resource 块来定义，一个 resource 可以定义一个或多个基础设施资源对象。例如，私有网络 VPC、虚拟机等。

资源语法

资源通过 resource 块定义，一个 `resource` 块包含 `resource` 关键字、资源类型、资源名和资源块体三部分。如下所示：

```
resource "tencentcloud_vpc" "foo" {
  name        = "ci-temp-test-updated"
  cidr_block  = "10.0.0.0/16"
  dns_servers = ["119.29.29.29", "8.8.8.8"]
  is_multicast = false

  tags = {
    "test" = "test"
  }
}
```

资源引用

通过该语法格式 `<资源类型>.<名称>.<属性>` 引用资源属性。如下所示：

```
tencentcloud_vpc.foo.resource # ci-temp-test-updated
```

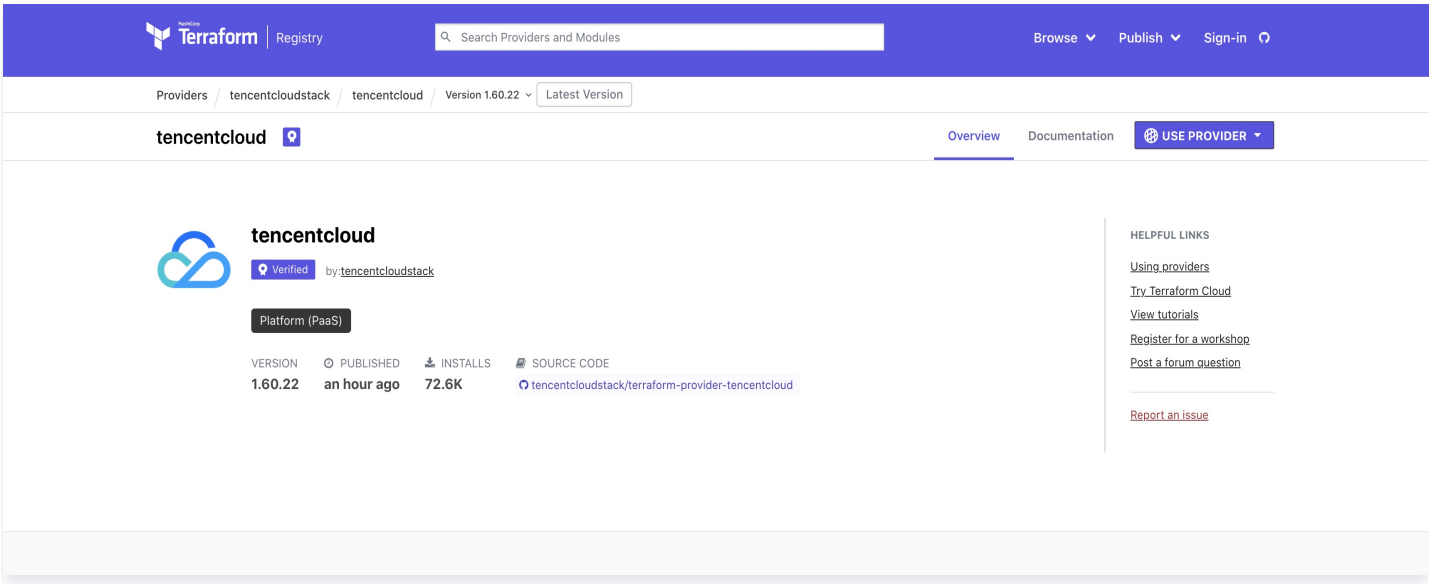
Provider

最近更新时间：2023-09-22 10:03:51

Terraform 依靠 Provider 插件与云提供商、SaaS 提供者和其他 API 进行交互。Terraform 配置必须声明插件需要哪些提供程序，以便 Terraform 可以安装和使用。此外，某些提供程序需要配置才能使用。本文介绍 Provider 插件配置方法。

搜索Provider

前往 Terraform 插件列表 页面，搜索并进入 TencentCloud Provider 页面查看使用指南。如下图所示：



下载 Provider

执行以下命令，默认从 Terraform 官方仓库下载最新版本的插件。

```
terraform init
```

若需使用历史版本，可通过 version 参数指定版本信息。代码示例如下：

```
terraform {
  required_providers {
    tencentcloud = {
      source = "tencentcloudstack/tencentcloud"
      # 通过version指定版本
      version = "1.60.18"
    }
  }
```

```
}
```

Provider声明

```
provider "tencentcloud" {
  region = "ap-guangzhou"
  secret_id = "my-secret-id"
  secret_key = "my-secret-key"
}
```

Modules

最近更新时间：2023-09-22 10:03:52

模块是包含一组 Terraform 代码的文件夹，是对多个资源的抽象和封装。

调用模块

在 Terraform 代码中使用 `module` 块引用模块。一个 `module` 块包含 `module` 关键字、 `module` 名称和 `module` 体（ `{}` 之间的部分）三部分。如下所示：

```
module "servers" {
  source = "./app-cluster"

  servers = 5
}
```

`module` 调用可以使用下列参数：

- `source` ：指定引用模块的源。
- `version` ：指定引用模块的版本号。
- `meta-arguments` ：Terraform 0.13开始支持的特性，类似 resource 与 data，可以用来操作 module 的行为，详情请参见 MetaData 。

参数说明

Source

`source` 参数用来指定模块源，指定 Terraform 获取子模块源代码的方式。Terraform 在 terraform init 的安装步骤中，使用它将源代码下载到本地磁盘上，以便其他 Terraform 命令可以使用。
模块安装程序支持从以下源类型进行安装：

- 本地路径
- TerraformRegistry
- GitHub
- Bitbucket
- Generic Git, Mercurial repositories
- HTTP URLs
- S3 buckets
- GCS buckets
- Modules in Package Sub-directories

本文介绍本地路径、TerraformRegistry 及 GitHub 的使用方法。

本地路径

本地路径可以使用同一项目中的子模块，与其他资源的区别是无需下载相关代码即可使用。例如：

```
module "consul" {
  source = "./consul"
}
```

Terraform Registry

Registry 目前是 Terraform 官方推荐的模块仓库方案，采用了 Terraform 定制的协议，支持版本化管理和使用模块。官方提供的 公共仓库 中保存和索引了大量公共模块，可以快速搜索到各类官方和社区提供的高质量模块。公共仓库的模块可以用 `<NAMESPACE>/<NAME>/<PROVIDER>` 形式的源地址来引用，您可在模块介绍中获取确切的源地址。例如：

```
module "consul" {
  source = "hashicorp/consul/xxx"
  version = "0.1.0"
}
```

托管在其他仓库的模块，则需在源地址头部添加 <HOSTNAME>/ 部分，指定私有仓库的主机名。例如：

```
module "consul" {
  source = "app.terraform.io/example-corp/k8s-cluster/azurerm"
  version = "1.1.0"
}
```

GitHub

若 Terraform 读取 source 参数值，是以 `github.com` 为前缀时，会将其自动识别为一个 GitHub 源。例如，使用 HTTPS 协议克隆仓库：

```
module "consul" {
  source = "github.com/hashicorp/example"
}
```

如需使用 SSH 协议，则请使用如下地址：

```
module "consul" {
  source = "git@github.com:hashicorp/example.git"
}
```

> **说明：**
> GitHub 源的处理与通用 Git 仓库一致，它们获取 git 凭证和通过 ref 参数引用特定版本的方式均一致。如需访问私有仓库，则需额外配置 git 凭证。

Version

使用 `registry` 作为模块源时，可以使用 version 元参数约束使用的模块版本。例如：

```
module "consul" {
  source  = "hashicorp/consul/xxx"
  version = "0.0.5"

  servers = 3
}
```

`version` 元参数的格式与 Provider 版本约束的格式一致。在满足版本约束的前提下，Terraform 会使用当前已安装的最新版本的模块实例。若当前没有满足约束的版本被安装过，则会下载符合约束的最新的版本。

`version` 元参数只能配合 registry 使用，支持公共的或私有的模块仓库。其他类型的模块源不一定支持版本化，本地路径模块不支持版本化。

MetaData

最近更新时间：2023-09-22 10:03:52

Metadata 是 Terraform 支持的内置元参数，可以在 provider、resource、data、module 块中使用。主要包括：

- depends_on：显式声明依赖关系。
- count：创建多个资源实例。
- for_each：迭代集合，为集合中每一个元素创建一个对应的资源实例。
- provider：指定非默认 Provider 实例。
- lifecycle：自定义资源的生命周期行为。
- dynamic：构建重复内嵌块。

depends_on

使用 `depends_on` 可以显式声明资源间由 Terraform 无法自动推导出的隐含依赖关系。适用于仅当资源间确实存在依赖关系，但彼此间没有数据引用的场景。例如：

```
variable "availability_zone" {
  default = "ap-guangzhou-6"
}

resource "tencentcloud_vpc" "vpc" {
  name       = "guagua_vpc_instance_test"
  cidr_block = "10.0.0.0/16"
}

resource "tencentcloud_subnet" "subnet" {
  depends_on = [tencentcloud_vpc.vpc]
  availability_zone = var.availability_zone
  name              = "guagua_vpc_subnet_test"
  vpc_id            = tencentcloud_vpc.vpc.id
  cidr_block        = "10.0.20.0/28"
  is_multicast      = false
}
```

count

`count` 参数可以是任意自然数，Terraform 会创建 `count` 个资源实例，每一个实例都对应一个独立的基础设施对象，并且在执行 Terraform 代码时，这些对象是被分别创建、更新或者销毁的。例如：

```
resource "tencentcloud_instance" "foo" {
  availability_zone = var.availability_zone
  instance_name     = "terraform-testing"
  image_id          = "img-ix05e4px"
  ...
  count             = 3
  tags = {
    Name = "Server ${count.index}"
  }
  ...
```

- count.index：代表当前对象对应的 count 下标索引（从0开始）
- 访问多资源实例对象：
  `<TYPE>.<NAME>[<INDEX>]` (例如: tencentcloud_instance.foo[0], tencentcloud_instance.foo[1])

for_each

`for_each` 是 Terraform 0.12.6 引入的新特性。一个 resource 块不允许同时声明 `count` 与 `for_each` 。for_each 参数可以是一个 map 或是一个 set(string)，Terraform 会为集合中每一个元素都创建一个独立的基础设施资源对象，并且和 count 一样，每一个基础设施资源对象在执行 Terraform 代码时都是独立创建、修改、销毁的。例如：

- map

```
resource "tencentcloud_cfs_access_group" "foo" {
for_each = {
   test1_access_group = "test1"
   test2_access_group = "test2"
}
name        = each.key
description = each.value
}
```

- set(string)

```
resource "tencentcloud_eip" "foo" {
  for_each = toset(["awesome_gateway_ip1", "awesome_gateway_ip2"])
  name = "awesome_gateway_ip"
}
```

provider

若声明了同一类型 Provider 的多个实例，则在创建资源时可以通过指定 provider 参数选择要使用的 Provider 实例。若未指定 provider 参数，那么 Terraform 默认使用资源类型名中第一个单词所对应的 Provider 实例。例如：

```
provider "tencentcloud" {
  region = "ap-guangzhou"
  # secret_id = "my-secret-id"
  # secret_key = "my-secret-key"
}

provider "tencentcloud" {
  alias  = "tencentcloud-beijing"
  region = "ap-beijing"
  # secret_id = "my-secret-id"
  # secret_key = "my-secret-key"
}

resource "tencentcloud_vpc" "foo" {
    name        = "ci-temp-test-updated"
    cidr_block   = "10.0.0.0/16"
    dns_servers  = ["119.29.29.29", "8.8.8.8"]
    is_multicast = false

    tags = {
        "test" = "test"
    }
    provider     = tencentcloud.tencentcloud-beijing
}
```

lifecycle

每个资源实例都具有创建 、更新和销毁三个阶段，而 `lifecycle` 块可指定一个不同的行为方式。Terraform 支持如下几种 `lifecycle` ：

- **create_before_destroy**

  默认情况下，当 Terraform 需要修改一个由于服务端 API 限制导致无法直接升级的资源时，Terraform 会删除现有资源对象，再用新的配置参数创建一个新的资源对象取代。 `create_before_destroy` 参数可以修改这个行为，使 Terraform 首先创建新对象，只有在新对象成功创建并取代老对象后再销毁老对象。例如：

  ```
  lifecycle {  create_before_destroy = true}
  ```

  许多基础设施资源需具备唯一的名字或标识属性，而新老对象并存时也需符合该约束。有些资源类型具备特别的参数，可为每个对象名称添加一个随机前缀以防止冲突，而 Terraform 不能默认采用这种行为，您需要在使用

`create_before_destroy` 前了解每一种资源类型在这方面的约束。

- **prevent_destroy**

  `prevent_destroy` 参数是一个保险措施，只要它被设置为 true 时，Terraform 会拒绝执行任何可能会销毁该基础设施资源的变更计划。`prevent_destroy` 参数可以预防意外删除关键资源，例如错误地执行了 terraform destroy，或者是意外修改了资源的某个参数，导致 Terraform 决定删除并重建新的资源实例。在 resource 块内声明了 `prevent_destroy = true` 会导致无法执行 terraform destroy。例如：

```
lifecycle {
  prevent_destroy = true
}
```

  需谨慎使用 `prevent_destroy` 参数，需要注意的是，该措施无法防止在删除 resource 块后 Terraform 删除相关资源，因对应的 `prevent_destroy = true` 声明也被一并删除了。

- **ignore_changes**

  默认情况下，Terraform 检测到代码描述的配置与真实基础设施对象之间有任何差异时，都会计算一个变更计划来更新基础设施对象，使之符合代码描述的状态。在一些非常罕见的场景下，实际的基础设施对象会被 Terraform 之外的流程所修改，这就会使 Terraform 不停地尝试修改基础设施对象以弥合和代码之间的差异。此时，可以通过设定 `ignore_changes` 来指示 Terraform 忽略某些属性的变更。`ignore_changes` 的值定义了一组在创建时需要按照代码定义的值来创建，但在更新时不需要考虑值的变化的属性名。例如：

```
resource "tencentcloud_instance" "foo" {
...
lifecycle {
  ignore_changes = [
    # Ignore changes to tags, e.g. because a management agent
    # updates these based on some ruleset managed elsewhere.
    tags,
  ]
}
```

dynamic

在例如 resource 的顶级块中，通常只能以类似 `name = expression` 的形式进行一对一的赋值。一般情况下均可使用该赋值形式，但当某些资源类型包含了可重复的内嵌块时，无法使用表达式循环赋值。例如：

```
name               = "example"
instance_type          = "basic"
open_public_operation = true
security_policy {
  cidr_block = "10.0.0.1/24"
}
```

```
  security_policy {
    cidr_block = "192.168.1.1/24"
  }
}
```

此时，可使用 dynamic 块来动态构建重复且类似 security_policy 的内嵌块。例如：

```
resource "tencentcloud_tcr_instance" "foo" {
  name                = "example"
  instance_type       = "basic"
  open_public_operation = true
  dynamic "security_policy" {
    for_each = toset(["10.0.0.1/24", "192.168.1.1/24"])
    content {
      cidr_block = security_policy.value
    }
  }
}
```

`dynamic` 可以在 resource、data、provider 和 provisioner 块内使用。dynamic 块类似于 for 表达式，它产生的是内嵌块，可以迭代一个复杂类型数据并为每一个元素生成相应的内嵌块。在上述示例中：

- **dynamic** 的标签（也就是 "security_policy"）确定了要生成的内嵌块种类。
- **for_each** 参数提供了需要迭代的复杂类型值。
- **iterator** 参数（可选）设置了表示当前迭代元素的临时变量名。若未设置 iterator，则临时变量名默认为 dynamic 块的标签（即 security_policy）。
- **labels** 参数（可选）是一个表示块标签的有序列表，用来按次序生成一组内嵌块。有 labels 参数的表达式中可使用临时的 iterator 变量。
- 内嵌的 **content** 块定义了要生成的内嵌块的块体。可以在 content 块内部使用临时的 iterator 变量。

`for_each` 参数：

- 由于 **for_each** 参数可以是集合或者结构化类型，可使用 for 表达式或展开表达式来转换一个现有集合的类型。
- **for_each** 的值必须是不为空的 map 或者 set。如需根据内嵌数据结构或者多个数据结构的元素组合来声明资源实例集合，可使用 Terraform 表达式和函数来生成合适的值。

`iterator` 变量（上述示例中的 setting）具备以下属性：

- **key**：若迭代容器是 map，则 key 为当前元素的键。若迭代容器是 list，则 key 为当前元素在 list 中的下标序号。若是由 for_each 表达式产出的 set，则 key 和 value 相等，此时不应使用 key。
- **value**：当前元素的值。一个 dynamic 块只能生成属于当前块定义过的内嵌块参数。无法生成例如 lifecycle、provisioner 这样的元参数，Terraform 必须在确保对这些元参数求值的计算是成功的。

Backend

最近更新时间：2023-09-22 10:03:52

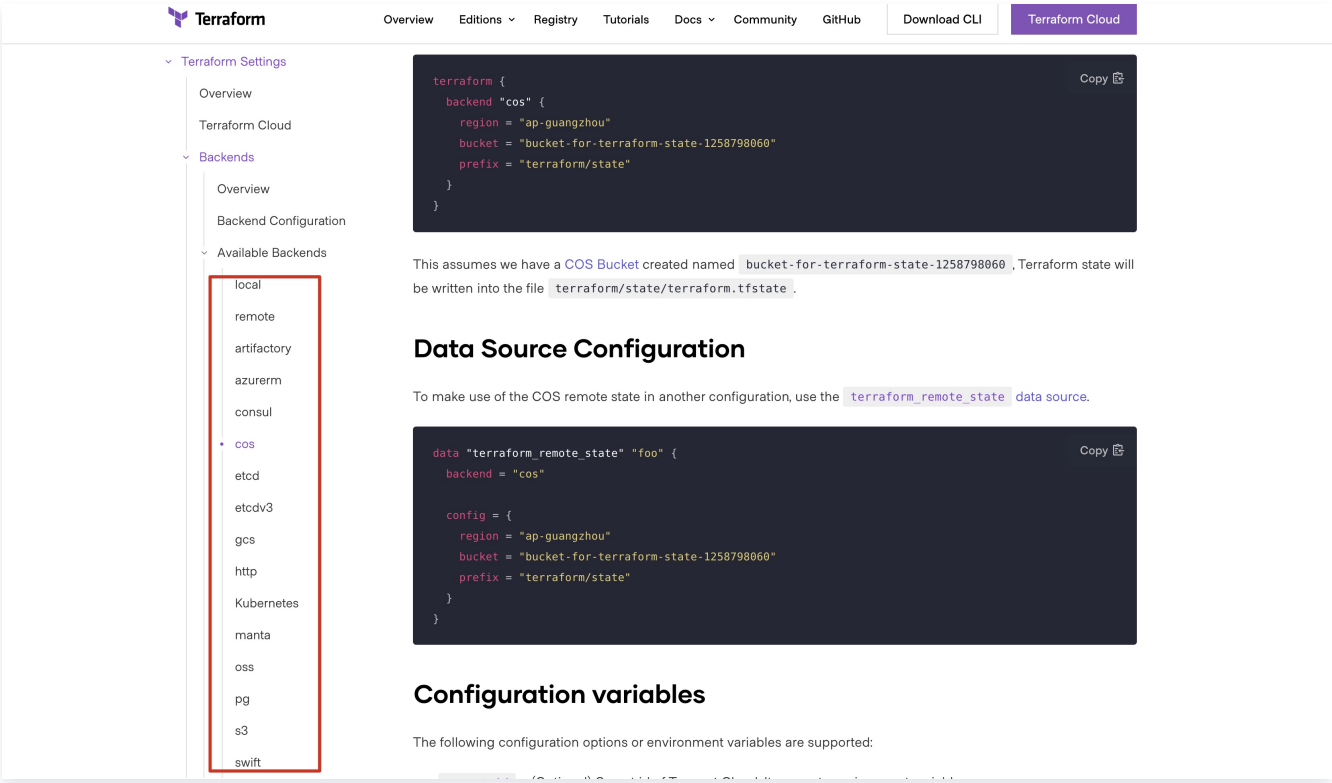
远程状态存储机制

状态文件若仅存储在本地，将可能存在以下问题：

- `tfstate` 文件默认保存在当前工作目录下的本地文件，若计算机损坏导致文件丢失，tfstate 文件所对应的资源都将无法管理，产生资源泄漏。
- 团队成员间无法共享 `tfstate` 文件。

为了解决状态文件的存储和共享问题，Terraform 引入了远程状态存储机制 Backend。Backend 是一种抽象的远程存储接口，类似 Provider、Backend 也支持多种不同的远程存储服务，详情请参见 Available Backends 。Terraform Backend 分为两种：

- **标准**：支持远程状态存储与状态锁。
- **增强**：在标准的基础上支持远程操作（在远程服务器上执行 plan、apply 等操作）。



说明事项

- backend 配置更新后需运行 `terraform init` 来验证和配置 backend。
- 未配置 backend 时，Terraform 默认使用本地 backend。例如，tfstate 文件默认是存储在本地目录下的。
- backend 配置存在以下重要限制：

- 一个配置只能提供一个后端块。
- 后端块不能引用命名值（如输入变量、局部变量或数据源属性）。

使用 Backend

`backend` 块嵌套定义在顶级 terraform 块中，本文以使用腾讯云对象存储 COS 服务进行配置。示例如下，如需使用其他存储模式，可前往 Available Backends 了解更多信息。

```
terraform {
  backend "cos" {
    region = "ap-nanjing"
    bucket = "tfstate-cos-1308126961"
    prefix = "terraform/state"
  }
}
```

若您具备 COS 的 `tfstate-cos-1308126961` 桶，则 Terraform 状态信息将会写进文件 `terraform/state/terraform.tfstate` 中。如下图所示：

CLI

最近更新时间：2023-09-22 10:03:52

本文介绍如何使用 Terraform 命令行工具应用 Terraform 代码和管理基础设施。

基本功能

查看命令列表

Terraform 提供了丰富的命令行操作，可以在命令行输入 `terraform` 查看完整命令列表。

```
➜  ~ terraform
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.

Main commands:
  init          Prepare your working directory for other commands
  validate      Check whether the configuration is valid
  plan          Show changes required by the current configuration
  apply         Create or update infrastructure
  destroy       Destroy previously-created infrastructure

All other commands:
  console       Try Terraform expressions at an interactive command prompt
  fmt           Reformat your configuration in the standard style
  force-unlock  Release a stuck lock on the current workspace
  get           Install or upgrade remote Terraform modules
  graph         Generate a Graphviz graph of the steps in an operation
  import        Associate existing infrastructure with a Terraform resource
  login         Obtain and save credentials for a remote host
  logout        Remove locally-stored credentials for a remote host
  output        Show output values from your root module
  providers     Show the providers required for this configuration
  refresh       Update the state to match remote systems
  show          Show the current state or a saved plan
  state         Advanced state management
  taint         Mark a resource instance as not fully functional
  test          Experimental support for module integration testing
  untaint       Remove the 'tainted' state from a resource instance
  version       Show the current Terraform version
  workspace     Workspace management

Global options (use these before the subcommand, if any):
  -chdir=DIR    Switch to a different working directory before executing the
                given subcommand.
  -help         Show this help output, or the help for a specified subcommand.
  -version      An alias for the "version" subcommand.
➜  ~
```

对于特定的子命令可以通过 `-help` 参看详细用法，例如需要查看 `validate` 子命令的完整用法，可以使用命令 `terraform validate -help` 。

切换工作目录

运行 Terraform 时一般要首先切换当前工作目录到包含有想要执行的根模块 .tf 代码文件的目录下（如使用 cd 命令），这样 Terraform 才能够自动发现要执行的代码文件以及参数文件。

全局参数 -chdir

在某些场景下，例如将 Terraform 封装进某些自动化脚本时，从其他路径直接执行特定路径下的根模块代码将十分便捷。可使用全局参数 `-chdir=...` 实现这一目的，您可在任意子命令的参数中使用该参数指定要执行的代码路径。例如：

```
terraform -chdir=environments/production apply
```

`-chdir` 参数指引 Terraform 在执行具体子命令之前切换工作目录，使用该参数后 Terraform 将会在指定路径下读写文件，而非当前工作目录下的文件。在以下两种场景时，指定 `-chdir` 参数将无效，Terraform 仍会使用当前的工作目录：

- Terraform 处理命令行配置文件中的设置而非执行某个具体的子命令时，该阶段发生在解析 `-chdir` 参数之前。
- 若需使用当前工作目录下的文件作为配置的一部分时，可通过代码中的 `path.cwd` 变量获得对当前工作路径的引用。此时不使用 -chdir 参数指定路径， 可通过 `path.root` 来获取代表根模块所在的路径。

自动补全

如果使用的是 `bash` 或 `zsh` ，则可通过以下命令获取自动补全的支持。

```
terraform -install-autocomplete
```

如需卸载自动补全，可执行以下命令。

```
terraform -uninstall-autocomplete
```

基本命令

terraform init

`terraform init` 命令用于初始化一个包含 Terraform 配置文件的工作目录。在编写 Terraform 代码或是克隆了 Terraform 项目后应首先执行该命令。

- **用法**
  `terraform init [options]`
  该命令执行一系列不同的初始化步骤来初始化当前目录。即使多次运行也是安全的，若报错也不会删除配置文件或者状态信息。
- **常用参数**
  - `-input=true`：是否在取不到输入变量值时提示用户输入。
  - `-lock=false`：是否在运行时锁定状态文件。
  - `-lock-timeout=\`：尝试获取状态文件锁时的超时时间，默认为0，意为一旦发现锁已被其他进程获取立即报错。
  - `-no-color`：禁止输出中包含颜色。
  - `-upgrade`：是否升级模块代码以及插件。
- **拷贝源模块**
  默认情况下，`terraform init` 命令认为工作目录存在配置并尝试初始化。您也可以通过 `-from-module=MODULE-SOURCE` 选项在空白工作目录下运行 `terraform init`，则会先将指定模块拷贝到当前目录再执行初始化操作，这种特殊的使用方式适用于以下两种场景：
  - 对于 source 对应的版本控制系统，可使用该方式签出指定版本代码并为它初始化工作目录。
  - 如果模块源指向的是一个样例项目，该方式可以把样例代码拷贝到本地目录以便后续基于样例编写新的代码。
    如果是常规运行操作建议用独立的步骤从版本控制系统中签出代码，使用版本控制系统所属的工具。
- **Backend 初始化**
  在执行 init 时，会分析根模块代码以寻找 Backend 配置，并使用给定的配置设定初始化 Backend 存储。若在已经初始化 Backend 后重复执行 init 命令，会更新工作目录以使用新的 Backend 设置。init 可能会根据改变的内容提示用户是否确认进行状态迁移。您可按需使用以下参数：
  - `-force-copy`：可跳过提示直接确认迁移状态。
  - `-reconfigure`：使 init 忽略任何现有配置，防止任何状态迁移。
  - `-backend=false`：可跳过 Backend 配置。
    注意某些 init 步骤需要已经被初始化的 Backend，推荐只在已经初始化过 Backend 后使用该参数。
  - `-backend-config`：可以用来动态指定 Backend 配置。
- **初始化子模块**
  init 会搜索 module 块，并通过 source 参数取回模块代码。您可按需使用以下参数：
  - `-upgrade`：将所有模块升级到最新版本的代码。默认情况下，模块安装之后重新运行 init 命令会继续安装上次执行 init 后新增的模块，但不会修改已被安装的模块。
  - `-get=false`：可跳过子模块安装步骤。
    需注意其他 init 步骤需要模块树完整，建议只在成功安装过模块以后使用该参数。
- **插件安装**
  参数说明如下：

- `-upgrade`：将之前所有已安装的插件升级到符合 version 约束的最新版本，此参数对手动安装的插件无效。
- `-get-plugins=false`：跳过插件安装。Terraform 会使用已安装在当前工作目录下或是插件缓存路径中的插件。如果这些插件不足以覆盖需求，那么 init 会失败。
- `-plugin-dir=PATH`：跳过插件安装，只从指定目录加载插件。该参数会跳过用户插件目录以及所有当前工作目录下的插件。要在使用过该参数后恢复默认行为，请使用 `-plugin-dir=""` 参数重新执行 init。
- `-verify-plugins=false`：在下载插件后跳过验证签名（不推荐）。官方插件都会经 HashiCorp 签名，Terraform 会验证这些签名。可以使用该参数跳过签名验证（Terraform 不会验证手动安装的插件的签名）。

terraform plan

`terraform plan` 命令用来创建变更计划。Terraform 会先运行一次 refresh（该行为也可以被显式关闭）：

- 若检测到变更后，可决定要执行的变更使现有状态迁移到代码描述的期待状态。您还可使用可选参数 `-out`，将变更计划保存在一个文件中，以便日后使用 `terraform apply` 命令来执行该计划。
- 若未检测到任何变更，则会提示无任何需执行的变更。

该命令可以方便地审查状态迁移的所有细节，而不会实际更改现有资源以及状态文件。例如，在将代码提交到版本控制系统前可以先执行 `terraform plan`，确认变更行为符合预期。

- **用法**

  `terraform plan [options]`

  默认情况下，plan 命令不需要参数，使用当前工作目录下的代码和状态文件执行 refresh。
- **常用参数**：
  - `-compact-warnings`：如果 Terraform 仅生成了告警信息而无错误信息，则以只显示消息总结的精简形式展示告警。
  - `-destroy`：生成销毁所有资源的计划。
  - `-detailed-exitcode`：当命令退出时返回一个详细的返回码。如果有该参数，那么返回码将会包含更详细的含义：
  - 0 = 成功的空计划（没有变更）
  - 1 = 错误
  - 2 = 成功的非空计划（有变更）
  - `-input=true`：在取不到值的情况下是否提示用户给定输入变量值。
  - `-lock=true`：与 apply 类似。
  - `-lock-timeout=0s`：与 apply 类似。
  - `-no-color`：关闭彩色输出。
  - `-out=path`：将变更计划保存到指定路径下的文件中，之后可使用 `terraform apply` 执行该计划。
  - `-parallelism-n`：限制 Terraform 遍历图的最大并行度，默认值为10。

- `-refresh=true`：计算变更前先执行 refresh。
- `-state=path`：状态文件的位置，默认为 `"terraform.tfstate"`。如果启用了远程 Backend 则该参数设置无效。
- `-target=resource`：目标资源的地址，该参数可反复声明，用以对基础设施进行部分更新。
- `-var 'foo=bar'`：与 apply 类似。
- `-var-file=foo`：与 apply 类似。

- **部分更新**
  使用 `-target` 参数可以使 Terraform 专注于一部分的资源。可以使用资源地址来标记这个集合，资源地址说明如下：
  - 若给定地址能够定位到一个资源，那么只该资源会被标记。如果该资源使用了 `count` 参数而未给定具体访问下标，则该资源所有实例都会被标记。
  - 如果给定地址定位到了模块，那么该模块内所有资源及其内嵌模块资源都会被标记。

> **说明：**
> 标记部分资源并计算更新计划的能力针对较罕见场景，例如，从之前的错误中恢复或绕过某些 Terraform 的设计限制。对于常规操作不推荐使用 `-target` 参数，因为它会造成无法检测的配置漂移以及使人无法从代码推导出当前真实的状态。

- **安全警告**
  被保存的变更计划文件（使用 `-out` 参数）内部可能含有敏感信息，Terraform 本身并不会加密计划文件。如需移动或保存该文件，强烈建议您自行加密。Terraform 预期将增强计划文件的安全性，您可持续关注。

terraform apply

`terraform apply` 为 Terraform 中最重要的命令。apply 命令用来生成执行计划（可选）并执行，使得基础设施资源状态符合代码的描述。

- **用法**
  `terraform apply [options] [dir-or-plan]`
  默认情况下，apply 会扫描当前目录下的代码文件，并执行相应的变更。也可以通过参数指定其他代码文件目录。
  在设计自动化流水线时也可以显式分为创建执行计划、使用 apply 命令执行该执行计划两个独立步骤。如果没有显式指定变更计划文件，terraform apply 会自动创建一个新的变更计划，并提示用户是否批准执行。如果生成的计划不包含任何变更，terraform apply 会立即退出，不会提示用户输入。
- **常用参数**
  - `-backup-path`：保存备份文件的路径。默认等于 `-state-out` 参数后加上 `".backup"` 后缀。设置为 "–" 可关闭备份（不推荐）。
  - `-compact-warnings`：如果 Terraform 仅生成了告警信息而无错误信息，则显示消息仅以总结的精简形式展示告警。

- `-lock=true`：执行时是否先锁定状态文件。
- `-lock-timeout=0s`：尝试重新获取状态锁的间隔。
- `-input=true`：在无法获取输入变量的值时是否提示用户输入。
- `-auto-approve`：跳过交互确认步骤，直接执行变更。
- `-no-color`：禁用输出中的颜色。
- `-parallelism=n`：限制 Terraform 遍历图时的最大并行度，默认值为10。
- `-refresh=true`：指定变更计划及执行变更前是否先查询记录的基础设施对象现在的状态以刷新状态文件。如果命令行指定了要执行的变更计划文件，该参数设置无效。
- `-state=path`：保存状态文件的路径，默认值 `"terraform.tfstate"` 。如果使用了远程 Backend 则该参数设置无效。该参数不影响其他命令，例如执行 init 时会找不到它设置的状态文件。如果要使所有命令都可以使用同一个特定位置的状态文件，请使用 Local Backend。
- `-state-out=path`：写入更新的状态文件的路径，默认情况使用 `-state` 的值。该参数在使用远程 Backend 时设置无效。
- `-target=resource`：通过指定资源地址指定更新目标资源。
- `-var 'foo=bar'`：设置一组输入变量的值。该参数可以反复设置以传入多个输入变量值。
- `-var-file=foo`：指定一个输入变量文件。

terraform destroy

`terraform destroy` 命令可以用来销毁并回收所有 Terraform 管理的基础设施资源。

用法

`terraform destroy [options]`

Terraform 管理的资源会被销毁，在执行销毁动作前会通过交互式界面征求用户的确认。该命令可以接收所有 apply 命令的参数，但不可以指定 plan 文件。

- 若 `-auto-approve` 参数为 true，则不会征求用户确认直接销毁。
- 若使用 `-target` 参数指定了某项资源，那么不但会销毁该资源，同时也会销毁一切依赖于该资源的资源。

> **说明：**
> `terraform destroy` 将执行的所有操作都可以随时通过执行 `terraform plan -destroy` 命令来预览。

terraform graph

`terraform graph` 命令用于生成代码描述的基础设施或是执行计划的可视化图形。它的输出是 DOT 格式，可以使用 GraphViz 来生成图片。

- **用法**
  `terraform graph [options] [DIR]`

该命令生成 DIR 路径下的代码锁描述的 Terraform 资源的可视化依赖图（如果 DIR 参数缺省则使用当前工作目录）。

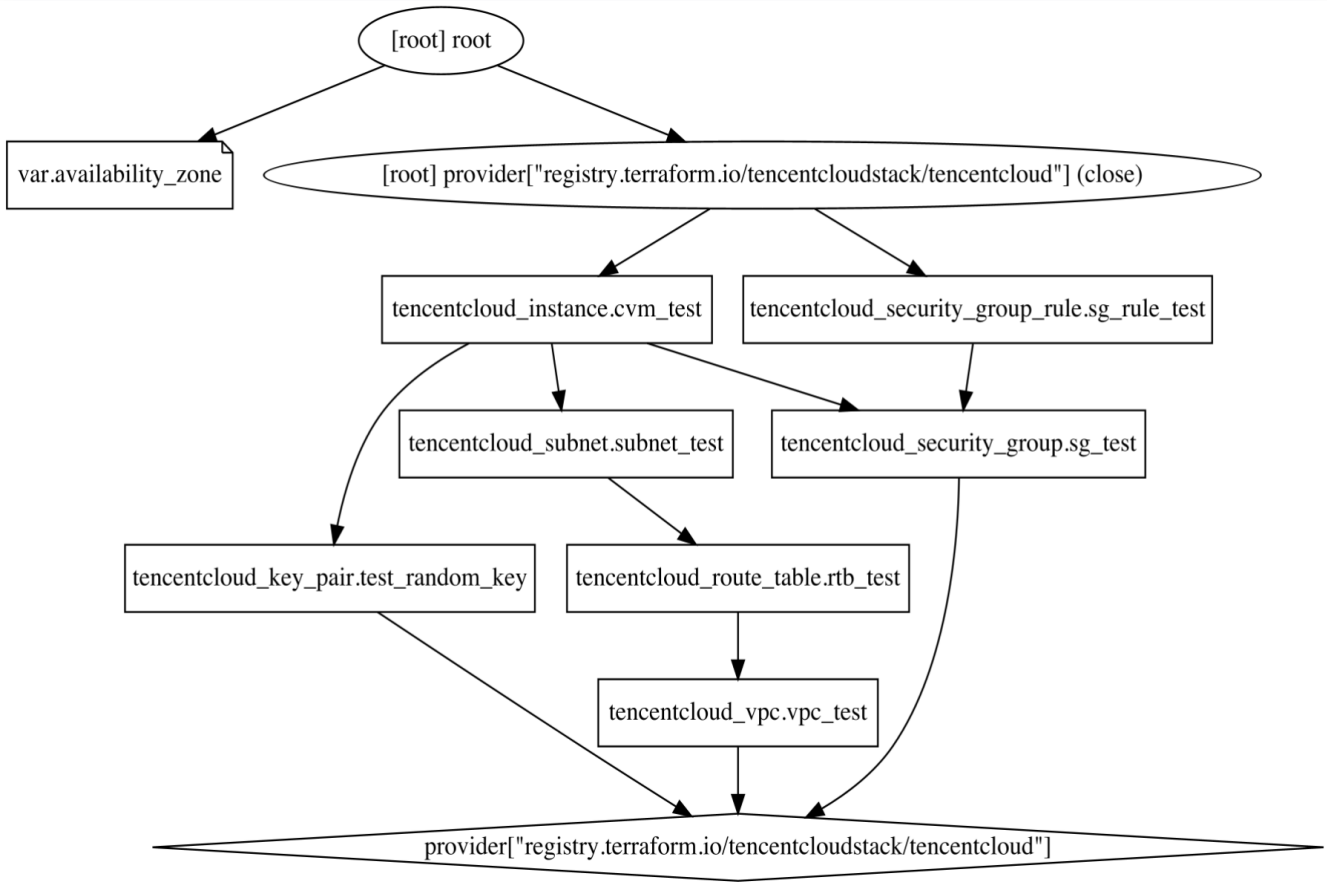
- **常用参数**
  - `-type`：用来指定输出的图表的类型。Terraform 为不同的操作创建不同的图。代码文件默认类型为 `"plan"`，变更计划文件默认类型为 `"apply"`。
  - `-draw-cycles`：用彩色的边高亮图中的环，可以帮助分析代码中的环错误（Terraform 禁止环状依赖）。
  - `-type=plan`：生成图表的类型。包含 plan、plan-destroy、apply、validate、input、refresh。
- **创建图片文件**

  `terraform graph` 命令输出的是 DOT 格式的数据，可通过以下命令 GraphViz 转换为图形文件：

```
terraform graph | dot -Tsvg > graph.svg
```

  若未安装 GraphViz，您可通过以下命令进行安装：
  - CentOS：`yum install graphviz`
  - Windows：`choco install graphviz`
  - Mac：`brew install graphviz`

    得到输出的图片类似下图所示：



terraform show

`terraform show` 命令从状态文件或是变更计划文件中打印可读的输出信息。这可以用来检查变更计划以确定所有操作都是预期的，或是审查当前的状态文件。

> **注意：**
> 可通过添加 `-json` 参数输出机器可读的 JSON 格式输出，但输出时所有标记为 sensitive 的敏感数据都会以明文形式被输出。

- **用法**
  `terraform show [options] [path]`
- **常用参数**
  - `path`：指定一个状态文件或是变更计划文件。如果没有给定 path，那么会使用当前工作目录对应的状态文件。
  - `-no-color`：与 apply 类似。
  - `-json`：以 JSON 格式输出。
- **JSON 格式输出**
  可以使用 `terraform show -json` 命令打印 JSON 格式的状态信息。如果指定了一个变更计划文件，

`terraform show -json` 会以 JSON 格式记录变更计划、配置以及当前状态。

terraform import

`terraform import` 命令用来将已经存在的资源对象导入 Terraform。
若存在已有一组运行着的基础设施资源，但未使用 Terraform 来构建和管理。此时已为其编写了对应的 Terraform 代码，则可使用 `terraform import` 将资源对象“导入”到 Terraform 状态文件中去。

- **用法**
  `terraform import [options] ADDRESS ID`
  `terraform import` 会根据资源 ID（ID 取决于被导入的资源对象的类型）找到相应资源，并将其信息导入到状态文件中 ADDRESS 对应的资源上。ADDRESS 必须符合在资源地址中描述的合法资源地址格式，`terraform import` 不但可以把资源导入到根模块中，也可以导入到子模块中。

  > **注意：**
  > Terraform 中每个资源对象都仅对应唯一一个实际的基础设施对象，需避免将同一个对象导入到两个以及更多不同的地址上，这会导致 Terraform 产生不可预测的行为。

- **常用参数**
  - `-backup=path`：生成状态备份文件的地址，默认情况下是 `-state-out` 路径加 `".backup"` 后缀名。设置为"-"可以关闭备份（不推荐）。
  - `-config=path`：包含含有导入目标的 Terraform 代码的文件夹路径。默认为当前工作目录。
  - `-input=true`：是否允许提示输入 Provider 配置信息。
  - `-lock=true`：如果 Backend 支持，是否锁定状态文件。
  - `-lock-timeout=0s`：重试获取状态锁的间隔。
  - `-no-color`：如果指定，则不会输出彩色信息。
  - `-parallelism=n`：限制 Terraform 遍历图的最大并行度，默认值为10。
  - `-state=path`：要读取的状态文件的地址。默认为配置的 Backend 存储地址，或是 `"terraform.tfstate"` 文件。
  - `-state-out=path`：指定修改后的状态文件的保存路径，默认情况下覆盖源状态文件。使用该参数可以生成一个新的状态文件，避免破坏现有状态文件。
  - `-var 'foo=bar'`：通过命令行设置输入变量值。
  - `-var-file=foo`：类似 apply 命令。

- **Provider 配置**
  Terraform 会尝试读取要导入的资源对应的 Provider 的配置信息。如果找不到相关 Provider 的配置，Terraform 会提示输入相关的访问凭据。您可输入凭据，也可通过环境变量来配置访问凭据。
  Terraform 在读取 Provider 配置时唯一的限制是不能依赖于“非输入变量”的输入。例如，Provider 配置不能依赖于数据源的输出。

若您需导入腾讯云资源，Terraform 会使用 `secret_id` 及 `secret_key` 输入变量来配置 tencentcloudProvier。配置文件如下：

```
variable "secret_id" {}
variable "secret_key" {}

provider "tencentcloud" {
    secret_id = var.secret_id
    secret_key = var.secret_key
  }
```

配置完成后，您可执行类似如下命令，导入资源：

```
terraform import tencentcloud_instance.foo ins-2s6ewubw
```

DataSource

最近更新时间：2023-09-22 10:03:52

数据源允许在 Terraform 外部定义，由另一个单独的 Terraform 配置定义或由函数修改的信息。

使用数据源

数据源通过一种特殊类型的资源进行访问，该资源使用 `data` 块声明。如下所示：

```
data "tencentcloud_availability_zones" "my_favourite_zone" {
  name = "ap-guangzhou-3"
}
```

引用数据源

引用数据源数据的语法是 `data.<TYPE>.<NAME>.<ATTRIBUTE>` 。如下所示：

```
resource "tencentcloud_subnet" "app" {
  ...
  availability_zone = data.tencentcloud_availability_zones.default.zones.0.name
  ...
}
```

语法指南
基本语法

最近更新时间：2023-09-22 10:03:52

基本类型

基元类型是一种简单类型，它不是由任何其他类型构成的。Terraform 中的所有基元类型都由 type 关键字表示。可用的基元类型包括：

- `string`：表示某些文本（如 "hello"）的 Unicode 字符序列。
- `number`：代表数字，可以为整数或小数。
- `bool`：代表布尔值，为 true 或 false。

示例如下：

```
id = 123
vpc_id = "123"
status = true
```

复合类型

复合类型是由一组值组合的复合类型。

集合类型

一个集合包含了一组同一类型的值。包括：

- `list(...)`：由从零开始的连续整数标识的值序列。
- `map(...)`：每个值都由字符串标签标识的一组值。
- `set(...)`：一组唯一值的集合。

结构类型

- `object(...)`：自定义类型，包含自己的命名属性。
- `tuple(...)`：由从零开始的连续整数标识的元素序列，其中每个元素都有自己的类型。

特殊类型

- `null`：如果将一个参数设置为 null，表示这个参数未填写，Terraform 会自动忽略该参数，并使用默认值。
- `any`：any 是 Terraform 中非常特殊的一种类型约束，它本身并非一个类型，而只是一个占位符。每当一个值被赋予一个由 any 约束的复杂类型时，Terraform 会尝试计算出一个最精确的类型来取代 any。

参数

参数赋值即将一个值赋给一个特定的名称，参数名称可以使用字母、数字、下划线(_)和连接符(-)表示，且首字母不能是数字。例如：

```
id = "123"
```

块

一个块是包含一组参数的容器，例如：

```
resource "tencentcloud_instance" "foo" {
  tags                          = {}
  vpc_id                         = "vpc-5bt2ix8p"
}
```

注释

Terraform 支持以下三种注释：

- `#` ：单行注释，其后的内容为注释。
- `//` ：单行注释，其后的内容为注释。
- `/*` 和 `*/` ：多行注释，应以注释多行。

表达式

最近更新时间：2023-09-22 10:03:52

运算符

运算符是进行算术或逻辑运算的操作。

算数运算符

运算符	说明
a + b	返回 a 与 b 的和。
a - b	返回 a 与 b 的差。
a * b	返回 a 与 b 的积。
a / b	返回 a 与 b 的商。
a % b	返回 a 与 b 的模。该操作符一般仅在 a 与 b 是整数时有效。
-a	返回 a 的相反数。

比较运算符

运算符	说明
a == b	如果 a 与 b 类型与值都相等则返回 true，否则返回 false。
a != b	与 == 相反。
a < b	如果 a 比 b 小则为 true，否则为 false。
a > b	如果 a 比 b 大则为 true，否则为 false。
a <= b	如果 a 比 b 小或者相等则为 true，否则为 false。
a >= b	如果 a 比 b 大或者相等则为 true，否则为 false。

逻辑运算符

运算符	说明
a || b	a 或 b 中有至少一个为 true 则为 true，否则为 false。
a && b	a 与 b 都为 true 则为 true，否则为 false。

!a	如果 a 为 true 则为 false，如果 a 为 false 则为 true。

条件表达式

条件表达式是判断一个布尔表达式的结果，以便于在后续两个值当中选择一个。例如：

```
condition ? one_value : two_value
```

FOR 表达式

FOR 表达式可用来遍历一组集合，并将一种集合类型映射为另一种类型。例如：

```
[for item in items : upper(item)]
```

展开表达式

展开表达式是一种类似 for 表达式的简洁表达方式。例如：

```
[for o in var.list : o.id]
等价于
var.list[*].id
```

函数表达式

Terraform 支持在计算表达式时使用一些内建函数，函数调用表达式类似操作符。例如：

```
upper("123")
```

函数

最近更新时间：2023-11-28 15:09:22

数值函数

函数名称	功能	示例	结果
abs	取绝对值	abs(-1024)	1024
ceil	向上取整	ceil(5.1)	6
floor	向下取整	floor(4.9)	4
log	计算对数	log(16, 2)	4
pow	计算指数幂	pow(3,2)	9
max	取最大值	max(12,54,3)	54
min	取最小值	min(12, 54, 3)	3

字符串函数

函数名称	功能	示例	结果
chomp	删除字符串末尾换行符	chomp("hello\n")	"hello"
format	格式化字符串	format("Hello, %s!", "Ander")	"Hello, Ander!"
lower	字符串转小写	lower("HELLO")	"hello"
upper	字符串转大写	upper("hello")	"HELLO"
join	将字符串列表使用指定分隔符拼接	join(", ", ["foo", "bar", "baz"])	"foo, bar, baz"
replace	替换字符串中的指定字符	replace("1 + 2 + 3", "+", "-")	"1 - 2 - 3"

更多函数信息请参见 Terraform 官网 。

Terraform 样式

最近更新时间：2023-09-22 10:03:52

Terraform 语言具备惯用的样式约定，建议用户始终遵循约定，以确保不同团队编写的文件和模块之间的一致性。同时自动格式化工具也需应用约定，可以使用 `terraform fmt` 进行格式化。

代码约束

- 每个嵌套级别缩进两个空格。
- 当具有单行值的多个参数出现在同一嵌套级别的连续行上时，需对齐等号。例如：

```
ami         = "abc123"
instance_type = "t2.micro"
```

- 当参数和块同时出现在块体中时，需将所有参数一起放在顶部，使用一个空行将其与块分开，在空行下面放置嵌套块。
- 参数与内嵌块之间需空一行分隔。
- 对于同时包含参数和 "meta-arguments"（由 Terraform 语言语义定义）的块，需列出所有元参数，并放置在其他参数与块之间，上下分别使用空行分开。例如：

```
resource "tencentclould_instance" "example" {
  count = 2 # meta-argument first

  ami         = "abc123"
  instance_type = "t2.micro"

  network_interface {
    # ...
  }

  lifecycle {
    # meta-argument block last
    create_before_destroy = true
  }
}
```

- 顶级块应始终由一个空行彼此分隔。嵌套块也需使用空行分隔，除非将相同类型的相关块组合在一起（如资源中的多个供应器块）。
- 避免将多个相同类型的块与其他不同类型的块分开，除非这些块类型是由语义定义的以形成一个族。