Tencent Infrastructure Automation for Terraform

User Guide

Product Documentation

Contents

User Guide

Configuration Guide

Variables

Last updated：2023-05-29 17:00:49

Input Variable

Input variables let you customize aspects of Terraform modules without altering the module's own source code. This allows you to share modules across different Terraform configurations, making your module composable and reusable.

Input variables can be dynamically passed in; for example, you can pass in variables when creating or modifying the infrastructure, replace hard-coded access keys with variables when defining a provider in the code, and let users (infrastructure creators) decide the desired server size.

You can understand a set of Terraform code as a function, so the input variables can be seen as function input parameters.

Defining input variable

Input variables are defined using a `variable` block. For example:

```
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"
    }
  ]
}
```

The label after the `variable` keyword is a name for the variable, which must be unique among all variables in the same module. You can reference the variable value in the code through `var.<NAME>` .

A `variable` block can be declared with the following optional arguments:

`default` : Specifies the default value of the input variable.

`type` : Specifies that the input variable can only be assigned with a specific value.

`description` : Specifies the description of the input variable.

`validation` : Specifies the validation rules for the input variable.

`sensitive` : Limits Terraform UI output when the variable is used in configuration.

`nullable` : Specifies whether the input variable can be `null` or not.

Type

A type is defined in an input variable block by `type` .

Basic types: `string` , `number` , `bool` .

Complex types: `list(<TYPE>)` , `set(<TYPE>)` , `map(<TYPE>)` , `object({<ATTR NAME> = <TYPE>, ... })` , `tuple([<TYPE>, ...])` .

Overview

You can briefly describe the purpose of each variable. For example:

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

Custom validation rules

Prior to Terraform 0.13.0, only type constraints could be used to ensure that input arguments were of the correct type. Terraform 0.13.0 introduced custom validation rules for input variables. For example:

```
}
```

```
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 \\"am
  }
}
```

The `condition` argument is a bool argument. You can use an expression to determine whether the input variable is valid. When the `condition` is `true` , the input variable is valid; otherwise, it is not. A `condition` expression can reference only the currently defined variable by `var.<Variable name>` and must not produce errors. If the failure of an expression is the basis of the validation decision, use the `can` function to detect such errors. For example:

```
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 \\"am
  }
}
```

In the above example, if the input `image_id` does not meet the requirements of the regex, the regex function call will throw an error, which will be captured by the `can` function to output `false`. If the condition expression outputs `false`, Terraform will return the error message defined in `error_message`, which should fully describe the reason for the failure of the input variable validation, along with the valid constraints on the input variable.

Using an input variable

You can access an input variable with `var.<VARIABLE NAME>` only within the module where the variable is declared. For example:

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

Assigning a value to an input variable

Command line argument

To specify individual variables on the command line, you need to use the `-var` option when running the `terraform plan` and `terraform apply` commands. For example:

```
terraform apply -var="image_id=ami-abc123"
terraform apply -var='image_id_list=["ami-abc123","ami-def456"]' -var="instance_typ
terraform apply -var='image_id_map={"us-east-1":"ami-abc123","us-east-2":"ami-def45
```

Argument files

When you set a large number of variables, we recommend you specify their values in the variable argument file (suffixed with `.tfvars` or `.tfvars.json`) and specify the file on the command line with `-var-file`. For example:

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

Argument definition files use the same basic syntax as Terraform language files but contain only variable name assignments. For example:

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

Terraform automatically loads a number of variable definition files:

Files named `terraform.tfvars` or `terraform.tfvars.json` .

Files suffixed with `.auto.tfvars` or `.auto.tfvars.json` .

`.json` files need to be defined with the JSON syntax. For example:

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

Environment variables

You can specify input variables by defining environment variables prefixed with `TF_VAR_` . For example:

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

Input variable priority

You may assign the same variable more than once when using multiple assignment methods at the same time. Terraform overwrites the old value with the new one and loads variable values in the following order:

Environment variable

`terraform.tfvars` files (if any)
`terraform.tfvars.json` files (if any)
All `.auto.tfvars` or `.auto.tfvars.json` files in alphabetical order
Input variables passed in through the `-var` or `-var-file` command line argument, in the order defined in such argument
If you have tried multiple assignment methods in vain, Terraform will try to use the default value. For variables without a default value defined, Terraform will ask you to input a value on an interactive UI. Some Terraform commands will report errors if they are executed with the `-input=false` argument that disables value passing on the interactive UI.

Output Variable

Output variables make information of the infrastructure available for the command line and other Terraform configurations. An output value is similar to a returned value in traditional programming languages.

Use cases

Child modules can use output variables to pass their resource attributes to modules.
Root modules can use output variables to print certain values in the CLI output after `terraform apply` is executed.
When the remote state is used, other configurations can access the root module output through the `terraform_remote_state` data source.

Defining an output variable

Output variables are declared using an `output` block. For example:

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

Optional argument

description

Specifies descriptive information of the output value. For example:

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

sensitive

Hides the value of the output variable on the CLI when the `terraform plan` and `terraform apply` commands are being executed.

depends_on

Terraform parses various data and resources defined by the code and their dependencies. For example, if the `image_id` argument used to create a virtual machine is queried by `data` , then the virtual machine instance depends on this image's `data` . Terraform creates `data` first and then the virtual machine `resource` after the query result is obtained. In general, the order for creating `data` and `resource` is automatically calculated by Terraform and does not need to be explicitly specified by the code writer. However, sometimes there are dependencies that cannot be derived through code analysis, in which case the dependencies can be explicitly declared in the code through `depends_on` . For example:

```
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,
    ]
  }
```

Local Variable

If you need to use a complex expression to calculate a value and use it repeatedly, you can assign the complex expression a local value and then reference it repeatedly. If you see the input variable as the function input and output value as the returned value of the function, the local value is equivalent to a local variable defined in the function.

Local variable definition

Local variables are declared using a `locals` block. For example:

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

Additionally, local variables include not only literal constants but also other variables of the module (variables, resource attributes, or other local values) in order to convert or combine them. For example:

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

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

Using local variable

Local variables are referenced through the `local.<NAME>` expression. For example:

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

  tags = local.common_tags
}
```

Resource

Last updated：2023-05-30 16:39:44

Resources are the most important element in the Terraform language. A resource is defined using a `resource` block, which describes one or more infrastructure objects, such as VPC and VM.

Resource syntax

A resource is defined using a `resource` block, which contains the `resource` keyword, resource type, resource name, and resource block body as shown below:

```
    }
}
```

```
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"
    }
}
```

Resource reference

A resource attribute is referenced in the syntax format of `<Resource type>.<Name>.<Attribute>` as shown below:

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

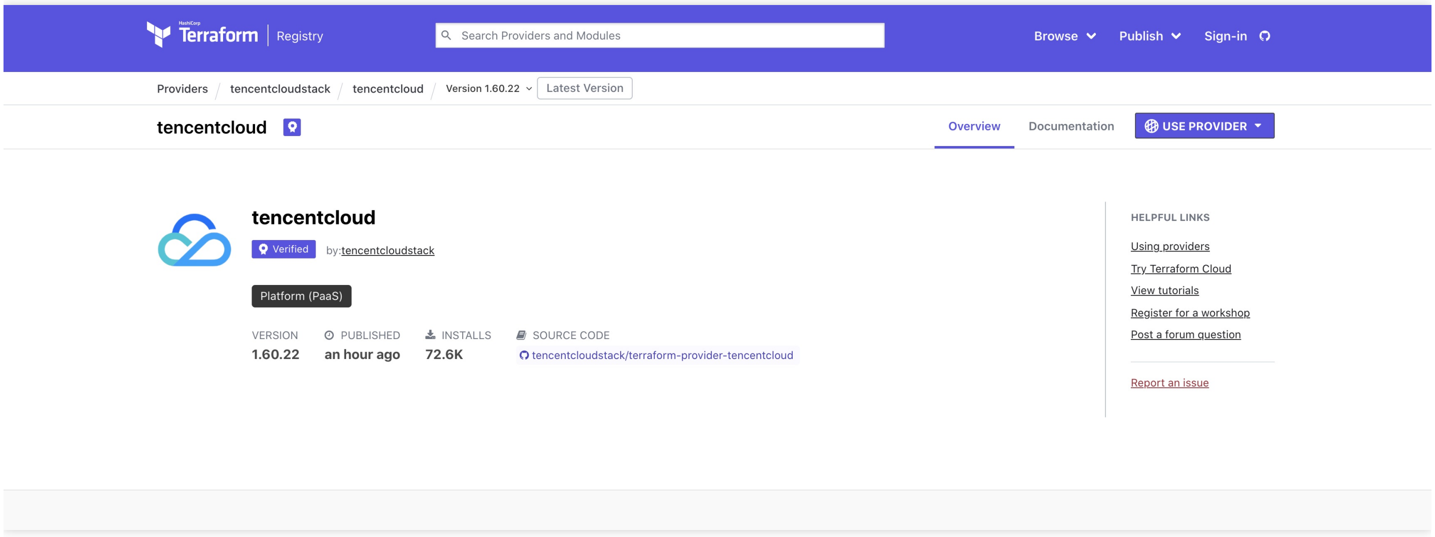
Provider

Last updated：2023-03-07 10:35:48

Terraform relies on provider plugins to interact with cloud providers, SaaS providers, and other APIs. Terraform configurations must declare which providers they require so that Terraform can install and use them. Additionally, some providers require configuration before they can be used. This document describes how to configure provider plugins.

Searching for a provider

Go to the Providers page, search, and enter the TencentCloud Provider page to view the user guide as shown below:



Downloading a provider

Run the following command to download the latest plugin version from Terraform's official repository by default.

```
terraform init
```

If you need to use a legacy version, you can specify the version information with the `version` argument as shown below:

```
terraform {
required_providers {
tencentcloud = {
```

```
source = "tencentcloudstack/tencentcloud"
# Specify the version by `version`
version = "1.60.18"
}
}
}
```

Provider declaration

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

Modules

Last updated：2023-03-07 10:35:48

A module is a folder containing a set of Terraform code, which is an abstraction and encapsulation of multiple resources.

Calling a Module

Modules are referenced in Terraform code using a `module` block, which contains the `module` keyword, `module` name, and `module` body (the part within the `{}`) as shown below:

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

servers = 5
}
```

A `module` can be called using the following arguments:

- `source` : Specifies the source of the referenced module.
- `version` : Specifies the version number of the referenced module.
- `meta-arguments` : It is a feature supported since Terraform 0.13. Similar to `resource` and `data` , it can be used to manipulate the behaviors of `module` . For more information, see MetaData.

Parameter Description

Source

The `source` argument tells Terraform where to find the source code for the desired child module. Terraform uses this during the module installation step of `terraform init` to download the source code to a directory on local disk so that it can be used by other Terraform commands.
The module can be installed from the following source types:

- Local paths

- Terraform Registry
- GitHub
- Bitbucket
- Generic Git, Mercurial repositories
- HTTP URLs
- S3 buckets
- GCS buckets
- Modules in Package Sub-directories

This document describes installation from local path, Terraform Registry, and GitHub.

Local path

Local paths can use child modules from the same project. Unlike other resources, local paths do not require the download of relevant code. For example:

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

Terraform Registry

Terraform Registry is currently the module repository solution recommended by Terraform. It uses Terraform's custom protocol and supports version management and module use. Terraform Registry hosts and indexes a large number of public modules, allowing quick search for a variety of official and community-supplied quality modules.

Modules in Terraform Registry can be referenced with source addresses in the format of `<namespace>/<name>/<provider>` . You can get the exact source address in the module description. For example:

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

For modules hosted in other repositories, you can add `<hostname>/` to the source address header to specify the server name of the private repository. For example:

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

GitHub

If Terraform reads a source argument value prefixed with `github.com` , it will automatically recognize it as a GitHub source. For example, you can clone a repository using the HTTPS protocol:

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

To use the SSH protocol, use the following address:

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

Note：

The GitHub source is handled in the same way as generic Git repositories, as both of them get Git credentials and reference specific versions through the `ref` argument in the same way. If you want to access private repositories, you need to configure additional Git credentials.

Version

When a registry is used as a module source, you can use the `version` meta-argument to constrain the version of the module used. For example:

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

servers = 3
}
```

The format of the `version` meta-argument is in line with the provider version constraint. In this case, Terraform will use the latest version of the module instance that has been installed. If no compliant version is currently installed, the latest compliant version will be downloaded.

The `version` meta-argument can only be used with the registry to support public or private module repositories. Other types of module sources, such as local path, do not necessarily support versioning.

MetaData

Last updated：2023-06-15 17:55:48

`Metadata` is a built-in meta-argument supported by Terraform and can be used in provider, resource, data, and module blocks. It mainly includes:

`depends_on` : Explicitly declares dependencies.

`count` : Creates multiple resource instances.

`for_each` : Iterates a collection to create a corresponding resource instance for each element in the collection.

`provider` : Specifies a non-default provider instance.

`lifecycle` : Customizes the lifecycle behavior of a resource.

`dynamic` : Builds repeatable nested blocks.

depends_on

`depends_on` explicitly declares implicit dependencies between resources that cannot be automatically deducted by Terraform. This is useful only if there is a dependency but no data reference between resources. For example:

```
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

The `count` argument can be any natural number. Terraform creates `count` resource instances, each corresponding to a separate infrastructure object that is created, updated, or terminated separately during Terraform code execution. For example:

```
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` : Represents the count subscript index (starting from 0) corresponding to the current object.
Access to object with multiple resource instances: `<TYPE>.<NAME>[<INDEX>] (For example:tencentcloud_instance.foo[0],tencentcloud_instance.foo[1])` .

for_each

`for_each` is a new feature introduced in Terraform 0.12.6. A `resource` block does not allow both `count` and `for_each` to be declared. The `for_each` argument can be a map or a set of strings. Terraform creates a separate infrastructure resource object for each element in the collection; just like with `count` , each infrastructure resource object is created, modified, or terminated separately during Terraform code execution. For example:
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

If multiple instances of the same type of provider are declared, you can specify a `provider` argument to select a provider instance to be used when creating a resource. If no `provider` argument is specified, Terraform will use the one corresponding to the first word in the resource type name by default. For example:

```
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

Each resource instance goes through creation, update, and termination, while the `lifecycle` block can specify a different behavior. Terraform supports the following types of `lifecycle` blocks:

create_before_destroy

By default, when Terraform needs to modify a resource that cannot be directly upgraded due to server-side API limitations, it will delete the existing resource object and replace it with one created using new configuration arguments. The `create_before_destroy` argument can modify this behavior so that Terraform creates a new object and terminates the old one only after it has been successfully replaced. For example:

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

Many infrastructure resources have a unique name or ID attribute, and this constraint applies to both the old and new objects when they coexist. Some resource types have special arguments that can add a random prefix to each object name to prevent conflicts, which is not adopted by Terraform by default. You need to understand the constraints for each resource type before using `create_before_destroy` .

prevent_destroy

The `prevent_destroy` argument is a safety measure. As long as it is set to `true` , Terraform will refuse to run any change plan that might terminate the infrastructure resource. It prevents accidental deletion of a critical resource,

such as erroneous execution of `terraform destroy` or accidental modification of an argument of a resource that makes Terraform decide to delete a resource instance and create a new one.
Declaring `prevent_destroy = true` inside a `resource` block will prevent `terraform destroy` from being executed. For example:

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

The `prevent_destroy` argument should be used with caution. Note that this measure does not prevent Terraform from deleting relevant resources after a `resource` block is deleted, as the corresponding `prevent_destroy = true` statement has also been deleted.

ignore_changes

By default, when Terraform detects any difference between the configuration described by the code and an actual infrastructure object, it will calculate a change plan to update the infrastructure object to match the state described by the code. In some very rare cases, the actual infrastructure object is modified by a process outside of Terraform, and Terraform will continually try to modify the object to bridge the discrepancy with the code. In such cases, you can instruct Terraform to ignore changes to certain attributes by setting `ignore_changes`. The value of `ignore_changes` defines a set of attribute names that need to be created according to the values defined by the code but do not need to be updated according to value changes. For example:

```
}
```

```
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

In a top-level block such as `resource` , you usually can only perform one-to-one assignments in a form like `name = expression` . This assignment form is generally available, except when some resource types contain repeatable nested blocks. For example:

```
resource "tencentcloud_tcr_instance" "foo" {
  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"
    }
  }
```

In this case, you can use the `dynamic` block to dynamically build repeatable nested blocks similar to `security_policy`. For example:

```
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` can be used in `resource` , `data` , `provider` , and `provisioner` blocks. Similar to a `for` expression, it produces nested blocks that iterate a complex type of data and generate a corresponding nested block for each element. In the above example:

The label of `dynamic` , i.e. `security_policy` , determines the type of nested blocks to be generated.

The `for_each` argument provides the complex type values to be iterated.

The `iterator` argument (optional) sets the name of a temporary variable that represents the current iteration element. If `iterator` is not set, the temporary variable name will default to the label of the `dynamic` block, i.e. `security_policy` .

The `labels` argument (optional) is an ordered list of block labels to generate a set of nested blocks in sequence. Temporary `iterator` variables can be used in expressions with the `labels` argument.

The nested `content` block defines the body of the nested block to be generated. Temporary `iterator` variables can be used inside the `content` block.

`for_each` argument:

As the `for_each` argument can be a collection or a structured type, you can use `for` or expanded expressions to convert the type of an existing collection.

The value of `for_each` must be a non-empty map or set. If you need to declare a collection of resource instances based on nested data structures or combinations of elements in multiple data structures, you can use Terraform expressions and functions to generate appropriate values.

The `iterator` variable (setting in the above example) has the following attributes:

`key` : If the iteration container is a map, then `key` is the key of the current element. If it is a list, then `key` is the subscript number of the current element in the list. In the case of a set produced by a `for_each` expression, `key` and `value` are equal, and `key` should not be used.

`value` : Value of the current element. A `dynamic` block can only generate nested block arguments within the current block definition. It is impossible to generate meta-arguments such as `lifecycle` and `provisioner` . Terraform must ensure the successful calculation of values for these meta-arguments.

Backend

Last updated：2023-03-07 10:35:48

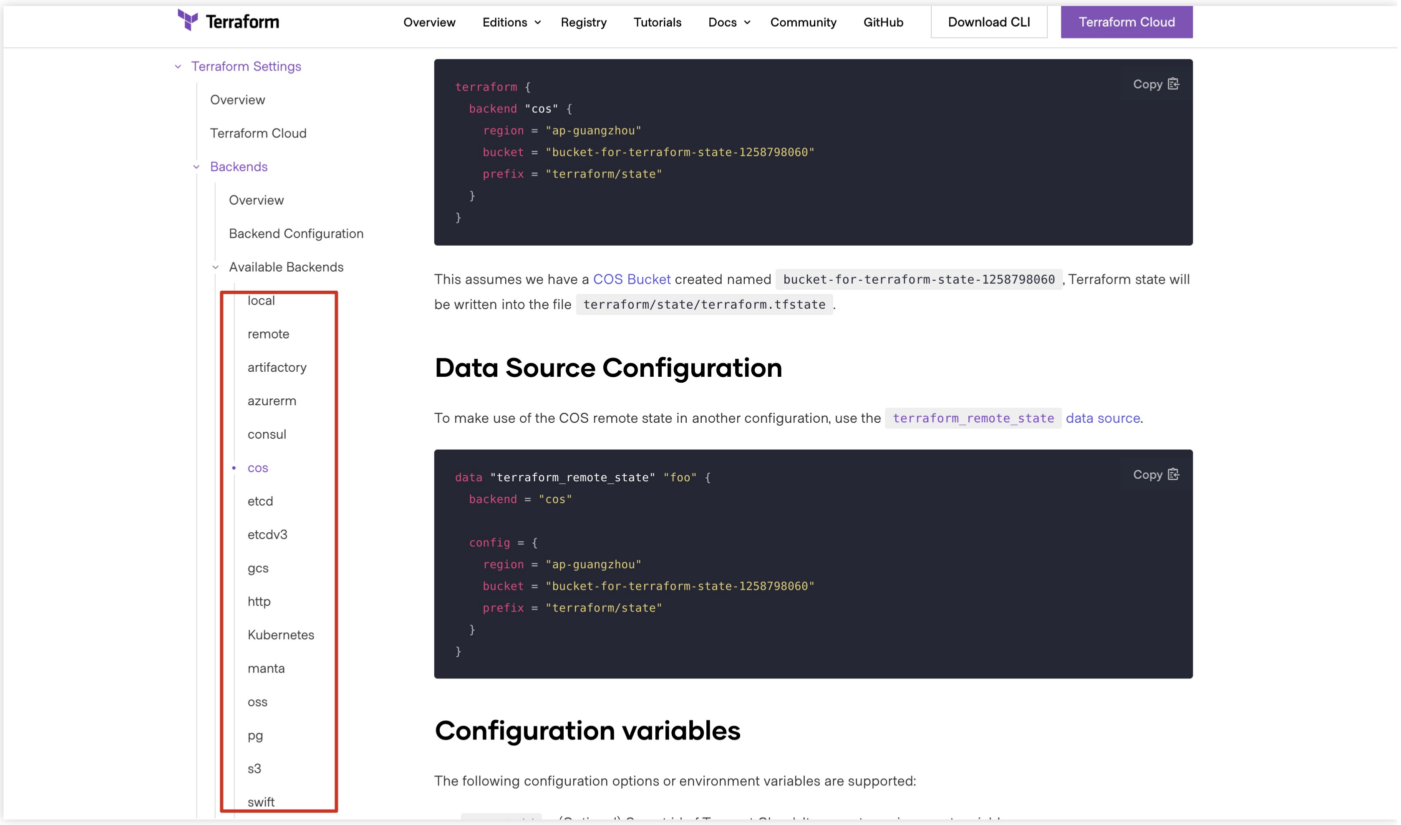
Remote state storage mechanism

Storing state files locally only may cause the following issues:

- A `tfstate` file is stored locally in the current working directory by default. If computer damage causes file loss, all the resources corresponding to the `tfstate` file will become unmanageable, leading to a resource leak.
- A `tfstate` file cannot be shared among team members.

To facilitate the storage and sharing of state files, Terraform introduces the remote state storage mechanism called "backend", an abstract remote storage API. Similar to a provider, backend supports a variety of remote storage services as described in Available Backends. A Terraform backend has two modes:

- **Standard**: Supports remote state storage and state locking.
- **Enhanced**: Supports remote operations (such as `plan` and `apply` on a remote server) in addition to the standard features.



```
terraform {
  backend "cos" {
    region = "ap-guangzhou"
    bucket = "bucket-for-terraform-state-1258798060"
    prefix = "terraform/state"
  }
}
```

This assumes we have a COS Bucket created named `bucket-for-terraform-state-1258798060`, Terraform state will be written into the file `terraform/state/terraform.tfstate`.

Data Source Configuration

To make use of the COS remote state in another configuration, use the `terraform_remote_state` data source.

```
data "terraform_remote_state" "foo" {
  backend = "cos"

  config = {
    region = "ap-guangzhou"
    bucket = "bucket-for-terraform-state-1258798060"
    prefix = "terraform/state"
  }
}
```

Configuration variables

The following configuration options or environment variables are supported:

Notes

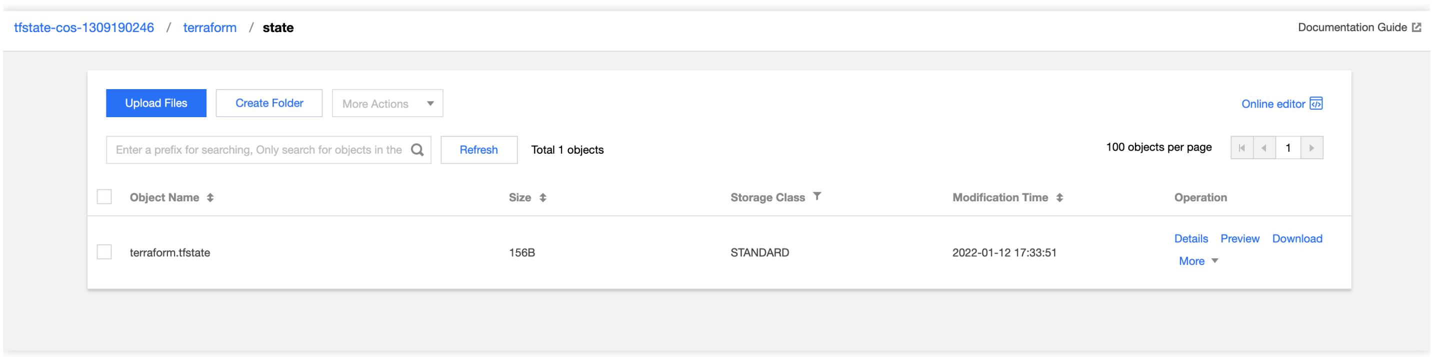
- After the backend configuration is updated, you need to run `terraform init` to verify and configure the backend.
- If no custom backend is configured, Terraform will use the local backend by default. For example, a `tfstate` file is stored in the local directory by default.
- Backend configuration is subject to the following restraints:
  - One configuration file provides only one backend block.
  - Backend blocks cannot reference named values (such as input variables, local variables, or data source attributes).

Using the backend

The definition of a `backend` block is nested in a top-level Terraform block. This document uses the Tencent Cloud Object Storage (COS) service as an example for configuration. For more information on other storage modes, see Available Backends.

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

If you have the `tfstate-cos-1308126961` bucket in COS, the Terraform state information will be written into the `terraform/state/terraform.tfstate` file as shown below:



Object Name	Size	Storage Class	Modification Time	Operation
terraform.tfstate	156B	STANDARD	2022-01-12 17:33:51	Details Preview Download More

CLI

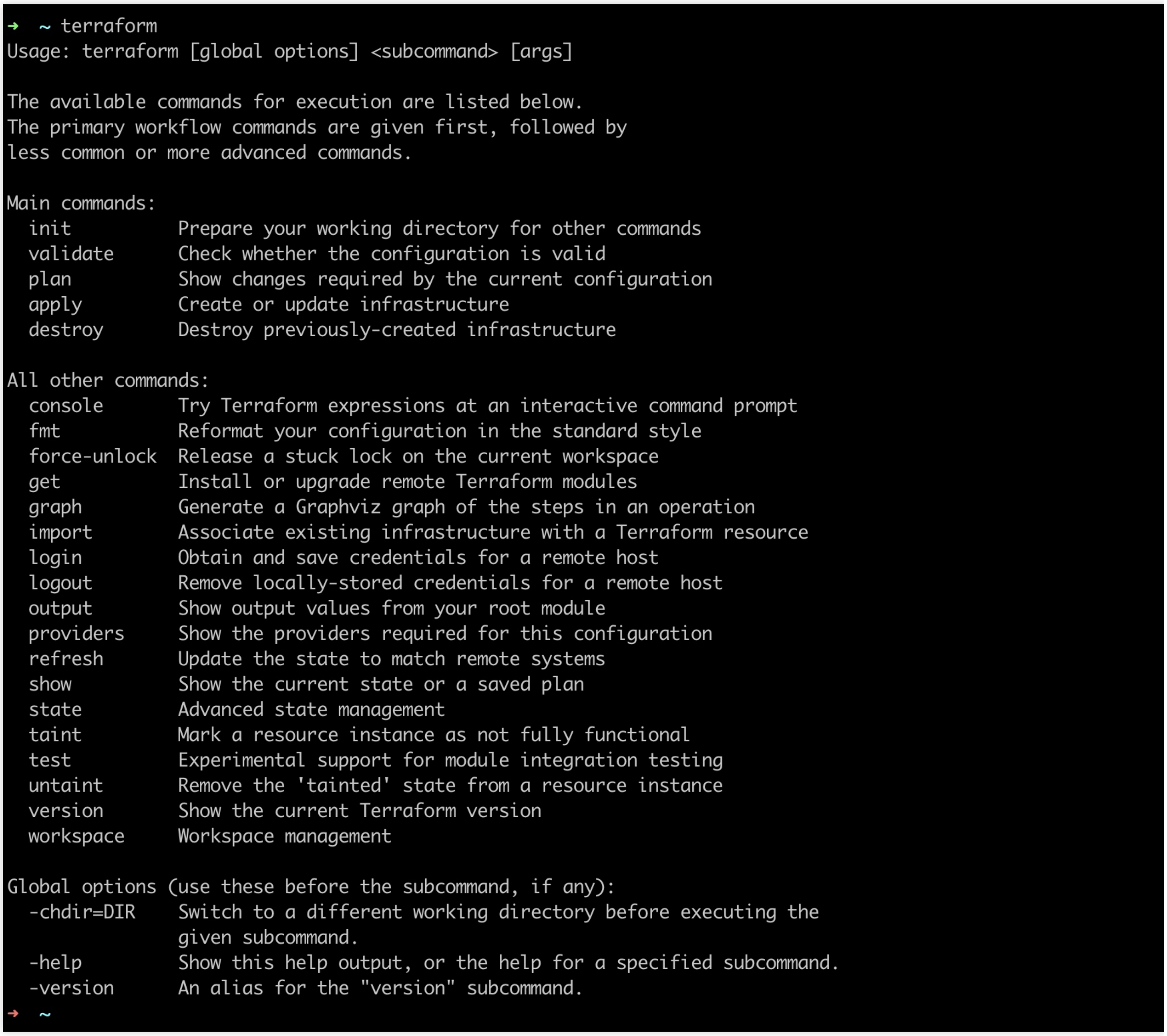
Last updated：2023-05-30 17:00:01

This document describes how to apply Terraform code and manage the infrastructure with the Terraform command-line interface (CLI).

Basic Features

Viewing command list

Terraform provides diversified command-line operations. Run `terraform` command line to view the complete list.



```
➜  ~ 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.
➜  ~
```

You can use `-help` to view the detailed help of specific subcommands. For example, you can run the `terraform validate -help` command to view the usage of the `validate` subcommand.

Switching working directory

The usual way to run Terraform is to first switch to the directory containing the .tf files for your root module (for example, with the `cd` command), so that Terraform will automatically find those code files and argument files to be executed.

Global option -chdir

In some cases, particularly when wrapping Terraform in automation scripts, you can easily run Terraform from a different directory than the root module directory. To allow that, Terraform supports a global option `-chdir=...` which you can include before the name of the subcommand you intend to run:

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

The `-chdir` option instructs Terraform to change its working directory to the given directory before running the given subcommand. This means that any files that Terraform would normally read or write in the current working directory will be read or written in the given directory instead. There are two exceptions where Terraform will use the original working directory even when you specify `-chdir` :

Settings in the CLI Configuration are not for a specific subcommand and Terraform processes them before acting on the `-chdir` option.

In case you need to use files from the original working directory as part of your configuration, a reference to `path.cwd` in the configuration will produce the original working directory instead of the overridden working directory. Use `path.root` to get the root module directory.

Auto-Completion

If `bash` or `zsh` is used, you can get the support for auto-completion with the following command.

```
terraform -install-autocomplete
```

You can run the following command to uninstall auto-completion.

```
terraform -uninstall-autocomplete
```

Basic Commands

terraform init

The `terraform init` command is used to initialize a working directory containing Terraform configuration files. You should run this command first after writing Terraform code or cloning a Terraform project.

Usage `terraform init [options]`

This command initializes the current directory in a series of steps. It is always safe to run multiple times and will not delete configuration files or state information even if an error is reported.

General options

`-input=true` : Whether to ask for input in case no input variable value is obtained.

`-lock=false` : Whether to lock a state file at runtime.

`-lock-timeout=\\` : Timeout period for trying to get a state file lock. The default value is 0, indicating that an error will be reported as soon as the lock is found to have been held by another process.

`-no-color` : Disables color codes in the command output.

`-upgrade` : Whether to upgrade module code and plugins.

Source module copying

By default, the `terraform init` command assumes that the working directory already contains a configuration and will attempt to initialize that configuration. You can also run `terraform init` in an empty working directory with the `-from-module=MODULE-SOURCE` option, in which case the specified module will be copied into the current directory before any other initialization steps are run. This special mode of operation supports two use cases:

Given a version control source, it can serve as a shorthand for checking out a configuration from version control and then initializing the working directory for it.

If the source refers to an example configuration, it can be copied into a local directory to be used as a basis for a new configuration.

For routine running operations, we recommend you check out the code from the version control system separately using the version control system's tool.

Backend initialization

During initialization, the root module code will be parsed to find the backend configuration, and the backend storage will be initialized with the given configuration settings.

Re-running `init` with an already-initialized backend will update the working directory to use the new backend settings. The `init` command may prompt you for confirmation of state migration based on the changes. You can use the following options as needed:

`-force-copy` : Skips the prompt to confirm the migration state directly.

`-reconfigure` : Makes `init` ignore any existing configuration to prevent any state migration.

`-backend=false` : Skips the backend configuration.

Note that some initialization steps require an already initialized backend, and we recommend you use this option only after the backend has been initialized.

`-backend-config` : Specifies the backend configuration dynamically .

Child module initialization

The `init` command will search for `module` blocks and get the module code with the `source` argument. You

can use the following options as needed:
`-upgrade` : upgrades all modules to the latest code version. By default, re-running the `init` command after module installation will continue to install the modules added after the last `init` execution, but will not modify the already installed modules.
`-get=false` : skips child module installation steps.
Note that other initialization steps require a complete module tree, and we recommend you use this option only after the module has been successfully installed.
Plugin installation
The parameters are described as follows:
`-upgrade` : Upgrades all previously installed plugins to the latest version in line with the version constraint. This option is invalid for manually installed plugins.
`-get-plugins=false` : Skips plugin installation. Terraform will use plugins already installed in the current working directory or the plugin cache path. If these plugins are not sufficient to meet the requirements, `init` will fail.
`-plugin-dir=PATH` : Skips plugin installation and loads plugins only from a specified directory. This option will skip plugins in the user plugin directory and all the current working directories. To restore the default behavior after this option is used, re-run `init` with the `-plugin-dir=""` options.
`-verify-plugins=false` : (not recommended) skips signature verification after plugin downloading (Terraform does not verify signatures of manually installed plugins). Official plugins are signed by HashiCorp and verified by Terraform.

terraform plan

The `terraform plan` command is used to create a change plan. Terraform will first run a refresh (this behavior can also be disabled explicitly):
If changes are detected, you can decide which change to be executed in order to migrate the existing state to the desired state as described by the code. You can also use the optional `-out` option to save the change plan in a file for execution later using the `terraform apply` command.
If no change is detected, you will be prompted that there is no change to be executed.
This command makes it easy to review all the details of a state migration without actually changing the existing resources and state files. For example, you can run `terraform plan` before committing the code to the version control system to confirm that the changes behave as expected.
Usage `terraform plan [options]`
By default, the plan command does not require options. It runs a refresh using the code and state files in the current working directory.
General options:
`-compact-warnings` : Displays alarms only with a message summary if Terraform generates only alarm information but no error information.

`-destroy` : Generates a plan to terminate all resources.
`-detailed-exitcode` : Returns a detailed exit code when the command exits:
0 = Succeeded with empty diff (no change)
1 = Error
2 = Succeeded with non-empty diff (with changes)
`-input=true` : Whether to ask you to specify the input variable value in case no value is obtained.
`-lock=true` : Similar to that of the `apply` command.
`-lock-timeout=0s` : Similar to that of the `apply` command.
`-no-color` : Disables color output.
`-out=path` : Saves the change plan to a file in the specified path for execution using `terraform apply` .
`-parallelism-n` : Limits the number of concurrent operations as the Terraform traverse graph, with a default value of 10.
`-refresh=true` : Executes a refresh before calculating changes.
`-state=path` : Location of a state file, with a default value of `"terraform.tfstate"` . This option is invalid if remote backend is enabled.
`-target=resource` : Address of the target resource. This option can be declared repeatedly to perform partial updates to the infrastructure.
`-var 'foo=bar'` : Similar to that of the `apply` command.
`-var-file=foo` : Similar to that of the `apply` command.

Partial update

Using the `-target` option allows Terraform to focus on a subset of resources, which can be marked with a resource address as described below:
If a resource can be located at a given address, only the resource will be marked. If the resource uses the `count` argument without a given access subscript, all instances of the resource will be marked.
If a module is located at a given address, all resources in the module and its embedded module resources will be marked.

Note：

Exceptional scenarios, such as recovery from a previous error or bypassing certain Terraform limitations, require the partial resource targeting and plan update calculating capability. The `-target` option is not recommended for routine operations, as it can cause undetectable configuration drift and make it impossible to deduce the current actual state from the code.

Security alarm

Change plan files saved (using the `-out` option) may contain sensitive information. We strongly recommend you encrypt the files by yourself for movement or save as Terraform does not encrypt them. Terraform is expected to launch new features to enhance the security of plan files.

terraform apply

As the most important command in Terraform, `terraform apply` is used to generate and (optionally) run an execution plan so that the infrastructure resource state matches the code description.

Usage `terraform apply [options] [dir-or-plan]`

By default, the `apply` command scans the current directory for code files and performs changes accordingly. Other code file directories can also be specified through options.

When designing an automation pipeline, you can also explicitly divide the process into two steps: creating an execution plan and running the plan with the `apply` command. If no change plan file is explicitly specified, `terraform apply` will automatically create a change plan and ask you whether to approve the execution. If the created plan does not contain any changes, `terraform apply` will exit immediately without prompting you for input.

General options

`-backup-path` : Path to save a backup file, with a default value of the `-state-out` option plus the `".backup"` suffix. You can set it to "-" to disable backup (not recommended).

`-compact-warnings` : Displays alarms only with a message summary if Terraform generates only alarm information but no error information.

`-lock=true` : Whether to lock a state file first before execution.

`-lock-timeout=0s` : Interval to attempt to get a state lock again.

`-input=true` : Whether to prompt you for input in case no input variable value is obtained.

`-auto-approve` : Skips interactive confirmation and runs changes directly.

`-no-color` : Disables color output.

`-parallelism-n` : Limits the number of concurrent operations as the Terraform traverse graph, with a default value of 10.

`-refresh=true` : Whether to query the current state of the recorded infrastructure object to refresh the state file first before specifying and running a change plan. This option is invalid if the command line specifies the change plan file to be executed.

`-state=path` : Path to save a state file, with a default value of `"terraform.tfstate"` . This option is invalid if a remote backend is used. It does not affect other commands; for example, the state file it sets will not be found when `init` is executed. If you want all commands to use the same location-specific state file, use a local backend.

`-state-out=path` : Path to write an updated state file, with a default value of `-state` . This option is invalid if a remote backend is used.

`-target=resource` : Specifies to update target resources by specifying resource addresses.

`-var 'foo=bar'` : sets the value of a set of input variables. This option can be set repeatedly to pass in multiple input variable values.

`-var-file=foo` : specifies an input variable file.

terraform destroy

The `terraform destroy` command can be used to terminate and repossess all the Terraform-managed infrastructure resources.

Usage

`terraform destroy [options]`

Before Terraform-managed resources are terminated, you will be asked for confirmation on an interactive UI. This command can accept all options of the `apply` command, but cannot specify a plan file.

If the `-auto-approve` option is `true` , resources will be terminated without your confirmation.

If a resource is specified with the `-target` option, the resource will be terminated along with all the resources that depend on it.

Note：

All the operations to be performed by `terraform destroy` can be previewed at any time by running the `terraform plan -destroy` command.

terraform graph

The `terraform graph` command is used to generate a visual graph of the infrastructure of the code description or the execution plan. Its output is in DOT format and can be converted to an image using GraphViz.

Usage `terraform graph [options] [DIR]`

This command generates a visual dependency graph of Terraform resources as described by the code lock in the DIR path (the current working directory is used if the default DIR option is used).

General options

`-type` : specifies the type of diagram to be output. Terraform creates different graphs for different operations. The default type of the code file is `"plan"` , and that of the change plan file is `"apply"` .

`-draw-cycles` : highlights the rings in the graph with colored edges to help analyze ring errors in the code (Terraform prohibits ring dependencies).

`-type=plan` : generates different types of diagrams, including `plan` , `plan-destroy` , `apply` , `validate` , `input` , and `refresh` .

Creating an image file

The `terraform graph` command outputs data in DOT format, which can be converted to an image file with the following command on GraphViz:

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

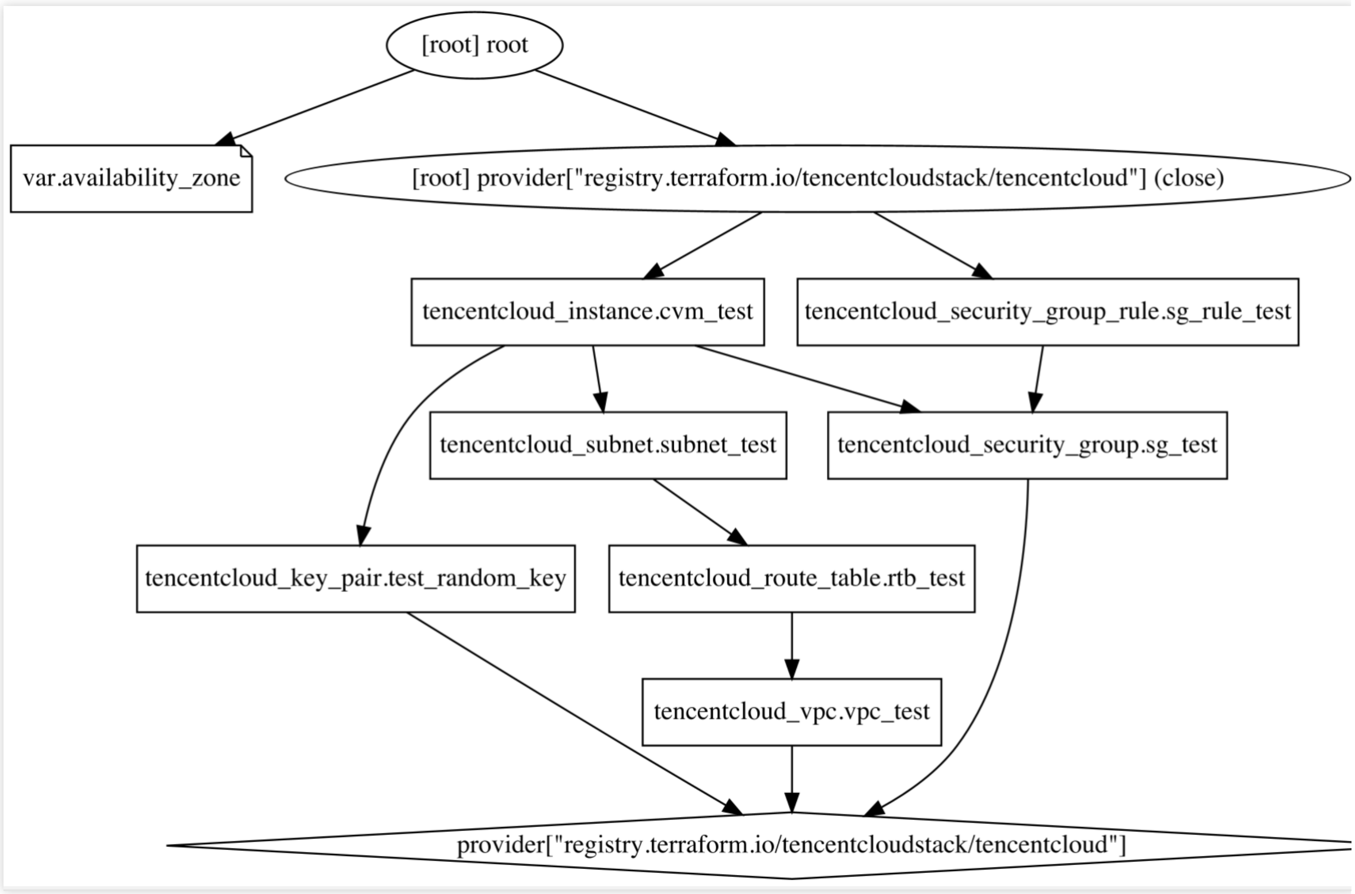
If Graphviz is not installed, you can install it with the following command:

CentOS: `yum install graphviz`

Windows: `choco install graphviz`

macOS: `brew install graphviz`

The output image is similar to the one as shown below:


[root] root
var.availability_zone
[root] provider["registry.terraform.io/tencentcloudstack/tencentcloud"] (close)
tencentcloud_instance.cvm_test
tencentcloud_security_group_rule.sg_rule_test
tencentcloud_subnet.subnet_test
tencentcloud_security_group.sg_test
tencentcloud_key_pair.test_random_key
tencentcloud_route_table.rtb_test
tencentcloud_vpc.vpc_test
provider["registry.terraform.io/tencentcloudstack/tencentcloud"]

terraform show

The `terraform show` command prints readable output from a state file or change plan file, which can be used to check the change plan to ensure that all operations are as expected, or to review the current state file.

Note：

Machine-Readable JSON data can be output by adding the `-json` option, but all the data marked as `sensitive` will be output in plaintext.

Usage

`terraform show [options] [path]`

General options

`path` : Specifies a state file or change plan file. If no path is given, the state file corresponding to the current working directory will be used.

`-no-color` : Similar to that of the `apply` command.

`-json` : Outputs in JSON format.

Output in JSON Format

The state information can be printed in JSON format using the `terraform show -json` command. If a change plan file is specified, `terraform show -json` will record the change plan, configuration, and current state in JSON format.

terraform import

The `terraform import` command is used to import an existing resource object into Terraform.
If there exists a set of running infrastructure resources that are not built or managed using Terraform and corresponding Terraform code has been written for them, you can use `terraform import` to "import" the resource objects into the Terraform state file.

Usage

`terraform import [options] ADDRESS ID`

`terraform import` finds the corresponding resource based on its resource ID (subject to the type of the imported resource object) and imports its information to the resource corresponding to `ADDRESS` in the state file. `ADDRESS` must be in the valid resource address format as described in the resource address. `terraform import` can import resources into not only root modules but also child modules.

Note：

Each resource object in Terraform corresponds to only one actual infrastructure object. You need to avoid importing the same object to two and more addresses, which can lead to unpredictable behaviors in Terraform.

General options

`-backup=path` : address of the generated state backup file, with a default value of the `-state-out` path plus the `".backup"` suffix. You can set it to "-" to disable backup (not recommended).
`-config=path` : path to the folder containing the Terraform code with the import target. The default path is the current working directory.
`-input=true` : whether to allow prompting for the input of provider configuration information.
`-lock=true` : whether to lock a state file with backend support.
`-lock-timeout=0s` : interval to attempt to get a state lock again.
`-no-color` : disables color output.
`-parallelism=n` : limits the maximum parallelism of the Terraform traversing graph, with a default value of 10.
`-state=path` : address of the state file to be read. The default address is the configured backend storage address or the `"terraform.tfstate"` file.
`-state-out=path` : specifies the path to save a modified state file. By default, the source state file is overwritten. This option can be used to create a state file while avoiding corrupting the existing one.
`-var 'foo=bar'` : sets the input variable value on the command line.
`-var-file=foo` : similar to that of the `apply` command.

Provider configuration

Terraform will try to read the configuration information of the provider corresponding to the resource to be imported. If no relevant provider configuration is found, Terraform will ask you to input relevant access credentials. You can either input credentials or configure them through environment variables.
The only restriction to read provider configuration in Terraform is that the configuration cannot rely on the input of "non-input variables", such as data sources.

If you need to import Tencent Cloud resources, Terraform will use the `secret_id` and `secret_key` input variables to configure TencentCloud Provider. The configuration files are as follows:

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

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

Upon the completion of configuration, you can import resources by running a command similar to the following:

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

DataSource

Last updated：2023-03-07 10:35:48

Data sources allow Terraform to use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.

Using a data source

A data source is accessed via a special kind of resource known as a data resource, declared using a `data` block as shown below:

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

Referencing a data source

The syntax for referencing data from a data source is `data.<type>.<name>.<attribute>` as shown below:

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

Syntax Guide

Expression

Last updated：2023-03-07 10:35:48

Operators

Operators are used for arithmetic or logical operations.

Arithmetic operators

Operator	Description
a + b	Returns the result of adding \`a\` and \`b\` together.
a - b	Returns the result of subtracting \`b\` from \`a\`.
a * b	Returns the result of multiplying \`a\` and \`b\`.
a / b	Returns the result of dividing \`a\` by \`b\`.
a % b	Returns the remainder of dividing \`a\` by \`b\`. This operator is generally useful only when used with whole numbers.
-a	Returns the result of multiplying \`a\` by -1.

Comparison operators

Operator	Description
a == b	Returns \`true\` if \`a\` and \`b\` both have the same type and the same value, or \`false\` otherwise.
a != b	Contrary to \`==\`.
a < b	Returns \`true\` if \`a\` is less than \`b\`, or \`false\` otherwise.
a > b	Returns \`true\` if \`a\` is greater than \`b\`, or \`false\` otherwise.
a <= b	Returns \`true\` if \`a\` is less than or equal to \`b\`, or \`false\` otherwise.
a >= b	Returns \`true\` if \`a\` is greater than or equal to \`b\`, or \`false\` otherwise.

Logical operators

Operator	Description
a || b	Returns `true` if either `a` or `b` is `true`, or `false` if both are `false`.
a && b	Returns `true` if both `a` and `b` are `true`, or `false` if either one is `false`.
!a	Returns `true` if `a` is `false`, or `false` if `a` is `true`.

Conditional Expression

A conditional expression uses the value of a Boolean expression to select one of two values. For example:

```
condition ? one_value : two_value
```

for Expression

A `for` expression can be used to traverse a set of collections and map one collection type to another. For example:

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

Splat Expression

A splat expression provides a more concise way to express a common operation that could otherwise be performed with a for expression. For example:

```
[for o in var.list : o.id]
is equivalent to
var.list[*].id
```

Function Expression

Terraform supports the use of some built-in functions when expressions are calculated. An expression to call a function is similar to an operator. For example:

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

Function

Last updated：2023-03-07 10:35:48

Numeric functions

Function	Feature	Sample	Result
abs	Returns an absolute value	abs(-1024)	1024
ceil	Rounds up	ceil(5.1)	6
floor	Rounds down	floor(4.9)	4
log	Calculates a logarithm	log(16, 2)	4
pow	Calculates an exponent	pow(3,2)	9
max	Returns the maximum value	max(12,54,3)	54
min	Returns the minimum value	min(12, 54, 3)	3

String functions

Function	Feature	Sample	Result
chomp	Removes newline characters at the end of a string	chomp("hello\n")	"hello"
format	Formats a string	format("Hello, %s!", "Ander")	"Hello, Ander!"
lower	Converts all cased letters in a string to lowercase	lower("HELLO")	"hello"
upper	Converts all cased letters in a string to uppercase	upper("hello")	"HELLO"
join	Concatenates elements of a string list with the given delimiter	join(", ", ["foo", "bar", "baz"])	"foo, bar, baz"
replace	Replaces specified characters in a string	replace("1 + 2 + 3", "+", "-")	"1 - 2 - 3"

For more information on functions, see Built-in Functions.

Basic Syntax

Last updated：2023-05-29 17:32:17

Basic Type

A primitive type is simple type that is not made from any other types. All primitive types in Terraform are represented by a type keyword. Available primitive types include:

- `string` : A sequence of Unicode characters representing some text (such as "hello").
- `number` : A numeric value, which can be an integer or a decimal.
- `bool` : A Boolean value, which can be `true` or `false` .

Sample:

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

Complex Type

A complex type is a type that groups multiple values into a single value.

Collection type

A collection contains a set of values of the same type:

- `list(...)` : A sequence of values identified by consecutive integers starting with 0.
- `map(...)` : A set of values, each of which is identified by a string label.
- `set(...)` : A set of unique values.

Structural type

- `object(...)` : A custom type that contains its own named attributes.
- `tuple(...)` : A sequence of elements identified by consecutive integers starting with 0, where each element has its own type.

Special type

- `null` : An argument set to null is considered omitted. Terraform will automatically ignore the argument and use the default value.

- `any` : It is a very special type constraint in Terraform. It is not a type but simply a placeholder. Whenever a value is given a complex type constrained by `any` , Terraform will try to calculate the most accurate type to replace `any` .

Argument

Argument assignment means assigning a value to a specific argument name. The name can contain letters, numbers, underscores, and hyphens and cannot begin with a number, such as:

```
id = "123"
```

Block

A block is a container for a set of arguments, such as:

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

Comment

Terraform supports the following three types of comments:

- `#` : A single-line comment followed by the comment content.

- `//` : A single-line comment followed by the comment content.

- `/*` and `*/` : Multi-line comments that might be span over multiple lines.

Terraform Style

Last updated：2023-05-29 17:37:47

The Terraform language has some idiomatic style conventions, and we recommend you always follow them to ensure consistency between files and modules written by different teams. In addition, automatic formatting tools also need to conform to such conventions, and you can use `terraform fmt` for formatting.

Code constraint

- Indent two spaces for each nesting level.
- When multiple arguments with single-line values appear on consecutive lines at the same nesting level, align their equals signs:

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

- When both arguments and blocks appear together inside a block body, place all of the arguments together at the top and then place nested blocks below them. Use one blank line to separate the arguments from the blocks.
- Use blank lines to separate logical groups of arguments within a block.
- For blocks that contain both arguments and "meta-arguments" (as defined by the Terraform language semantics), list meta-arguments first and separate them from other arguments with one blank line. Place meta-argument blocks last and separate them from other blocks with one blank line. For example:

```
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
}
}
```

- Top-Level blocks should always be separated from one another by one blank line. Nested blocks should also be separated by blank lines, except when grouping together related blocks of the same type (like multiple `provisioner` blocks in a resource).
- Avoid separating multiple blocks of the same type with other blocks of a different type, unless the block types are defined by semantics to form a family.