Resource Formation Service

Template Reference

Issue	01
Date	2025-02-10

HUAWEI TECHNOLOGIES CO., LTD.

Huawei Technologies Co., Ltd.

Address:	Huawei Industrial Base Bantian, Longgang Shenzhen 518129 People's Republic of China
Website:	https://www.huawei.com
Email:	support@huawei.com

Security Declaration

Vulnerability

Huawei's regulations on product vulnerability management are subject to the *Vul. Response Process*. For details about this process, visit the following web page:
https://www.huawei.com/en/psirt/vul-response-process
For vulnerability information, enterprise customers can visit the following web page:
https://securitybulletin.huawei.com/enterprise/en/security-advisory

Contents

1 Templates......1

2 Syntax......2

2.1 Basic Syntax......2

2.2 Style Conventions......4

2.3 Expressions......5

2.4 Common Functions......7

3 Configuration Guide......13

3.1 Provider......13

3.2 Resources......13

3.3 Data Source......14

3.4 Variables......14

3.4.1 Input Variables......14

3.4.2 Output Variables......17

3.4.3 Local Variables......18

3.5 Metadata......18

3.5.1 Instruction......18

3.5.2 depends_on......19

3.5.3 count......19

3.5.4 for_each......20

3.5.5 provider......20

3.5.6 lifecycle......21

4 Template Constraints and Limitations......23

1 Templates

RFS consists of templates and stacks.

A stack is a collection of Huawei Cloud resources and created by users on RFS. A template is a script used to create and update stacks.

2 Syntax

2.1 Basic Syntax

The RFS configuration language is easy, highly readable, and compatible with the HCL and JSON syntax. This section describes the basic syntax and common functions of the HCL.

The RFS configuration language consists of arguments, blocks, expressions, and functions.

Arguments

Use an equal sign (=) to assign a value or expression to a particular name, which can contain letters, digits, underscores (_), and hyphens (-), but cannot start with a digit. For example:

```
image_id = "ad091b52-742f-469e-8f3c-fd81cadf0743"
```

Blocks

Aggregate multiple arguments and can contain another blocks. A block consists of type, label, and body. The format is as follows:

```
resource "myinstance" {
   name   = "myinstance"
    ......
    network {
        uuid = "55534eaa-533a-419d-9b40-ec427ea7195a"
    }
}
```

Before using a block, you must declare its type (**resource** and **network** in this example), where **resource** is the top-level block type and **network** is the nested block type. The top-level block type keywords supported by the HCL include **provider**, **resource**, **data**, **variable**, **output**, **module**, and **locals**.

Block labels are defined after the block type, and the number of block labels is determined by the block type. In the example, the **resource** block type expects : **myinstance**. The nested **network** type does not have block labels. The block body is defined at the last and delimited by the **{** and **}** characters. Other types can be nested in the block body to implement different layered structures.

Argument Types

The HCL supports the following argument types:

Basic types

- **string**: consists of one or more Unicode characters, for example, **hello**.
- **number**: can be an integer or a floating point number.
- **bool**: can only be **true** or **false**.

The HCL can automatically convert the **number** and **bool** types to the **string** type based on the argument type. If a string can be represented as a value of the **number** or **bool** type, it can also be converted to the other two types. Arguments of the three types can be directly assigned values. For example:

```
disk_type = "SSD"
disk_size = 40
enable    = true

# Strings can be of the number and bool types.
disk_size = "40"
enable    = "true"
```

Set types

- **map(...)**: a set of data elements combined using key-value pairs. The **key** is of the **string** type, while the **value** can be of the **string**, **number**, or **bool** type. The values of all elements must be of the same type.
- **list(...)**: a set of data elements of the same type. The elements can be of the basic type or block type. The list index starts from 0.
- **set(...)**: similar to the **list** type. Elements in a set are unique and do not have any auxiliary identifier or sequence.

The **map** type is delimited in **{** and **}** and has flexible types. Key-value pairs can be connected using equal signs (=) or colons (:). If a key does not start with a digit, double quotation marks (") are not required. For multi-line mapping, key-value pairs can be separated by newline characters or commas (,). You are advised to use equal signs (=) to connect key-value pairs and separate them with newline characters. For example:

```
# Recommended format
tags = {
  foo = "bar"
  key = "value"
}

# Other formats
tags = {"foo" = "bar", "key" = "value"}
tags = {"foo" : "bar", "key" : "value"}
tags = {foo = "bar", key = "value"}
tags = {foo : "bar", key : "value"}
tags = {
  foo : "bar"
  key : "value"
}
```

The list type and set type are represented in the same way. The list/set whose elements are of the basic type is delimited using **[** and **]**, and the list/set whose elements are of the block type is represented in the form of repeated blocks. For example:

```
# List whose elements are of the basic type
security_groups = ["default", "internal"]

# List whose elements are of the block type
network {
  uuid = "55534eaa-533a-419d-9b40-ec427ea7195a"
}
network {
  uuid = "ad091b52-742f-469e-8f3c-fd81cadf0743"
}
```

Special types

- **null**: If a parameter is set to null, the parameter has no specified value. The HCL automatically ignores the parameter and uses the default value. Null is common in conditional expressions, for example, **var.test==""? null: var.test**, indicating that when the value of **var.test** is **""**, it is ignored.

Other Syntax

- A single-line comment starts with **#** or **//**.
- **/*** and ***/** are start and end delimiters for a comment that might span over multiple lines. Nested block comments are not supported.
- Terraform configuration files are UTF-8 encoded. Terraform accepts non-ASCII characters in identifiers, comments, and string values.
- A multi-line string starts with **<<EOF**, contains the string content in the middle, and ends with **EOF**. **EOF** can also be replaced with other characters. For example:

```
  ...
  website {
    ...
    routing_rules = <<EOF
[{
    "Condition": {
        "KeyPrefixEquals": "docs/"
    },
    "Redirect": {
        "ReplaceKeyPrefixWith": "documents/"
    }
}]
EOF
  }
}
```

2.2 Style Conventions

Style Conventions

The HCL has some idiomatic style conventions for consistency between files and modules written by different teams. The conventions are recommended for users to follow. They are as follows:

- 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 (=).

```
name            = "myinstance"
security_groups = ["default", "internal"]
```

- Use empty lines to separate logical groups of arguments within a block.
- 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.
- List meta-arguments at the top of the block body and separate them from other arguments with one blank line. Place meta-argument blocks at the end of the block body and separate them from other blocks with one blank line.

```
 count = 1

 bucket = "bucket_demo"
 acl    = "public-read"

 tags = {
   foo = "bar"
   env = "test"
 }

 lifecycle {
   create_before_destroy = true
 }
}
```

- Top-level blocks should always be separated from one another by one blank line.
- Nested blocks of the same type should be grouped together, while those of different types should be separated by blank lines.

Reference

https://www.terraform.io/docs/configuration/style.html

2.3 Expressions

Expressions refer to or compute values within a configuration. The simplest expressions are just literal values, like **hello world** or **5**. Terraform allows multiple expressions such as operators, conditional expressions, and built-in functions.

You can experience and test expressions and built-in functions using the Terraform expression console, by running the **terraform console** command.

Operators

Operators perform specific mathematical or logical operations. Terraform supports the following types of operators:

- Arithmetic operators: expect number values and produce number values as results, including +, - (subtraction), *, /, %, and - (multiplication by -1).
- Equality operators: both take two values of any type and produce bool values as results, including == and ! =.
- Comparison operators: expect number values and produce bool values as results, including >, >=, <, and <=.
- Logical operators: expect bool values and produce bool values as results, including ||, &&, and !.

When multiple operators are used together in an expression, they are evaluated in the following order of operations:

1. !, - (multiplication by -1)
2. *, /, %
3. +, - (subtraction)
4. >, >=, <, <=
5. ==, !=
6. &&
7. ||

Conditional Expressions

A conditional expression uses the value of a bool expression to select one of two values. The syntax is as follows:

```
condition ? true_value : false_value
```

This statement indicates that if **condition** is **true**, the result is **true_value**. Otherwise, the result is **false_value**. The result of a conditional expression can be of any type, but the types of **true_value** and **false_value** must be the same. A common use of conditional expressions is to define defaults to replace invalid values:

```
var.a != "" ? var.a : "default-a"
```

This statement indicates that if **var.a** is not empty, the actual value of **var.a** is returned. Otherwise, the result is **default-a**.

For Expressions

A for expression creates a set type by traversing and transforming each element in another set type (map, list, or set). The type of brackets around the for expression decide what type of result it produces.

- Using **[** and **]** will generate a list.
- Using **{** and **}** will generate a map or object.

Assume that the value of **mylist** is **["AA", "BBB", "CCCC"]**. You can use the for expression to convert each string element in **mylist** to lowercase and output another list.

```
> [for str in var.mylist : lower(str)]
[
  "aa",
  "bbb",
  "cccc",
]
```

You can also output a map, which is determined by =>:

```
> {for str in var.mylist : str => lower(str)}
{
  "AA" = "aa"
  "BBB" = "bbb"
  "CCCC" = "cccc"
}
```

The for expression can also convert a map. Assume that the value of **mymap** is **{element1="aaa", element2="bbb", element3="ccc"}**. You can convert each value in the map to uppercase.

```
> {for key, value in var.mymap : key => upper(value)}
{
  "element1 = "AAA"
  "element2 = "BBB"
  "element3 = "CCC"
}
```

In addition, the for expression can use the if clause to filter elements:

```
> [for str in var.list : upper(str) if length(str) >= 3]
[
  "bbb",
  "cccc",
]
```

Reference

https://www.terraform.io/docs/configuration/expressions.html

2.4 Common Functions

The HCL supports various built-in functions you can call by function name for processing strings, calculating values, encrypting values, and converting types. The syntax is as follows:

```
<Function name>(<Argument 1>, <Argument 2>...)
```

This section summarizes common functions in HCL and uses examples to describe their usage. For details about the complete list of supported functions, see **Terraform Functions**.

String Functions

Table 2-1 String functions

Name	Description	Example Value	Output
format	Produces a string by formatting a number of other values according to a specification string.	format("Hello, %s!", "cloud")	Hello, cloud!
lower	Converts all letters in the given string to lowercase.	lower("HELLO")	hello
upper	Converts all letters in the given string to uppercase.	upper("hello")	HELLO

Name	Description	Example Value	Output
join	Produces a string by concatenating together all elements of a given list of strings with the given delimiter.	join(", ", ["One", "Two", "Three"])	One, Two, Three
split	Produces a list by dividing a given string at all occurrences of a given separator.	split(", ", "One, Two, Three")	["One", "Two", "Three"]
substr	Extracts a substring from a given string by offset and length.	substr("hello world!", 1, 4)	ello
replace	Searches a given string for another given substring, and replaces each occurrence with a given replacement string.	replace("hello, cloud!", "h", "H")	Hello, cloud!

Numeric Functions

Table 2-2 Numeric functions

Name	Description	Example Value	Output
abs	Returns the absolute value of the given number.	abs(-12.4)	12.4

max	Takes one or more numbers and returns the greatest number from the set.	max(12, 54, 6)
		max([12, 54, 6]...)
			54
			54

min	Takes one or more numbers and returns the smallest number from the set.	min(12, 54, 6)
		min([12, 54, 6]...)
			6
			6

log	Returns the logarithm of a given number in a given base.	log(16, 2)	4

Name	Description	Example Value	Output
power	Calculates an exponent, by raising its first argument to the power of the second argument.	power(3, 2)	9

Collection Functions

Table 2-3 Collection functions

Name	Description	Example Value	Output
element	Retrieves a single element from a list by an index.	element(["One", "Two", "Three"], 1)	Two
index	Finds the element index for a given value in a list. If the given value is not present in the list, an error is reported.	index(["a", "b", "c"], "b")	1

lookup	Retrieves the value of a single element from a map, given its key. If the given key does not exist, the given default value is returned instead.	lookup({IT="A", CT="B"}, "IT", "G")
		lookup({IT="A", CT="B"}, "IE", "G")
			A
			G

flatten	Replaces any elements that are lists with a flattened sequence of the list contents.	flatten([["a", "b"], [], ["c"]])	["a", "b", "c"]
keys	Returns a list containing the keys from a map.	keys({a=1, b=2, c=3})	["a", "b", "c"]

length	Determines the length of a given list, map, or string.	length(["One", "Two", "Three"])
		length({IT="A", CT="B"})
		length("Hello, cloud!")
			3
			2
			13

Type Conversion Functions

Table 2-4 Type conversion functions

Name	Description	Example Value	Output
toset	Converts a list value to a set value.	toset(["One", "Two", "One"])	["One", "Two"]
tolist	Converts a set value to a list value.	toset(["One", "Two", "Three"])	["One", "Two", "Three"]
tonumber	Converts a string value to a number value.	tonumber("33")	33
tostring	Converts a number value to a string value.	tostring(33)	"33"

Encoding Functions

Table 2-5 Encoding functions

Name	Description	Example Value	Output
base64encode	Encodes a UTF-8 string using Base64.	base64encode("Hello, cloud!")	SGVsbG8sIGNsb3VkIQ==
base64decode	Decodes a Base64-encoded string to its original UTF-8 string. (If the bytes after Base64 decoding are not valid UTF-8, an error is reported.)	base64decode("SGVsbG8sIGNsb3VkIQ==")	Hello, cloud!
base64gzip	Compresses a UTF-8 string with gzip and then encodes the result using Base64.	base64gzip("Hello, cloud!")	H4sIAAAAAAAA//JIzcnJ11FIzskvTVEEAAAA//8BAAD//wbrhYUNAAAA

Hash and Crypto Functions

Table 2-6 Hash and crypto functions

Name	Description	Example Value	Output
sha256	Computes the SHA256 hash (hexadecimal) of a given string.	sha256("Hello, cloud!")	0ad167d1e3ac8e9f4e4f7ba83e92d0e3838177e959858631c770caaed8cc5e3a
sha512	Computes the SHA512 hash (hexadecimal) of a given string.	sha512("Hello, cloud!")	6eb6ed9fc4edffaf90e742e7697f6cc7d8548e98aa4d5aa74982e5cdf78359e84a3ae9f226313b2dec765bf1ea4c83922dbfe4a61636d585da44ffbd7e900f56
base64sha256	Computes the SHA256 hash of a given string and encodes it using Base64.	base64sha256("Hello, cloud!")	CtFn0eOsjp9OT3uoPpLQ44OBd+lZhYYxx3DKrtjMXjo=
base64sha512	Computes the SHA512 hash of a given string and encodes it using Base64.	base64sha512("Hello, cloud!")	brbtn8Tt/6+Q50LnaX9sx9hUjpiqTVqnSYLlzfeDWehKOunyJjE7Lex2W/HqTIOSLb/kphY21YXaRP+9fpAPVg==
md5	Computes the MD5 hash of a given string.	md5("hello world")	5eb63bbbe01eeed093cb22bb8f5acdc3

NOTE

The output of **base64sha512("Hello, cloud!")** is not equal to that of **base64encode(sha512("Hello, cloud!"))**, because the hexadecimal output of **sha512** is Unicode-encoded in Terraform, not UTF-8.

Filesystem Functions

Table 2-7 Filesystem functions

Name	Description	Example Value	Output
abspath	Converts a string containing a filesystem path to an absolute path.	abspath("./hello.txt")	/home/demo/test/terraform/hello.txt

Name	Description	Example Value	Output
dirname	Removes the last portion from a string containing a filesystem path.	dirname("foo/bar/baz.txt")	foo/bar
basename	Removes all except the last portion from a string containing a filesystem path.	basename("foo/bar/baz.txt")	baz.txt
file	Reads the contents of a file at the given path and returns them as a string.	file("./hello.txt")	Hello, cloud!
filebase64	Reads the contents of a file at the given path and returns them as a Base64-encoded string.	filebase64("./hello.txt")	SGVsbG8sIGNsb3VkIQ==

3 Configuration Guide

3.1 Provider

Provider

The Terraform configuration file ends with .tf or .tf.json and consists of **providers**, **resources**, **data sources**, and **variables**.

Each provider represents a service provider. Terraform interacts with providers through plug-ins. A service provider is declared using the keyword **provider**. For details about the provider configuration parameters, see .

When you run the **terraform init** command, the plug-in required is downloaded. By default, the plug-in of the latest version is downloaded from the official Terraform registry. For Terraform of versions later than 0.13, you can use **required_providers** to specify the registry source and version of a provider.

3.2 Resources

Resources are the most important element in the Terraform language and are declared using the keyword **resource**. Each cloud service supported by the provider corresponds to one or more resources. For example, indicates ECS, and indicates VPC.

Resource Reference

You can use an expression to reference a resource attribute in the format of **<*Resource type*>.<*Name*>.<*Attribute*>**. Assume that a **huaweicloud_compute_instance** resource named **myinstance** has been created. The following is an example:

```
# Instance ID
> huaweicloud_compute_instance.myinstance.id
55534eaa-533a-419d-9b40-ec427ea7195a

# Instance security group
> huaweicloud_compute_instance.myinstance.security_groups
["default", "internet"]
```

```
# IP address of the first NIC of the instance
> huaweicloud_compute_instance.myinstance.network[0].fixed_ip_v4
192.168.0.245

# IP addresses of all NICs of the instance
huaweicloud_compute_instance.myinstance.network[*].fixed_ip_v4
["192.168.0.24", "192.168.10.24"]

# Value of the tag key
> huaweicloud_compute_instance.myinstance.tags["key"]
value
```

3.3 Data Source

Data Source

A data source can be considered as a special resource and is declared using the keyword **data**. A data source is used to query the attributes and information of existing resources. For example, you can query the image ID and other attributes based on the image name of .

After an image is found, other resources can reference the attributes of the image to use it. The reference format is **data.<*Data type*>.<*Name*>.<*Attribute*>**.

3.4 Variables

3.4.1 Input Variables

Input variables are like arguments for a module. They are declared using the keyword **variable**. By defining input variables, you can flexibly modify the configuration without altering the source code of the module. You can use default values, CLI options, or environment variables to set the input variables' values.

Defining Input Variables

By convention, input variables are defined in a file named **variables.tf**. The input variable is declared using the keyword **variable**:

```
variable "iamge_id" {
  type        = string
  description = "image id of Ubuntu 1804"
}

variable "availability_zone_name" {
  type    = string
  default =
}
```

The label after the **variable** keyword is the name of the input variable, which must be unique among all variables in the same module. The name of a variable can be any valid identifier other than a reserved keyword. The reserved keywords include:

```
source    version    providers    count    for_each    lifecycle    depends_on    locals
```

A **variable** block contains the following arguments:

- **type**: specifies the type of a variable. The default value is **string**.
- **description**: describes the usage of a variable.
- **default**: specifies the default value of a variable. A variable with a default value can be regarded as an optional variable.
- **validation** block: specifies the customized validation rules of a variable.

If no variable type is specified, the default value **string** is used. You are advised to explicitly specify variable types; they can serve as helpful reminders for users of the module, and they allow Terraform to return a helpful error message if the wrong type is used. Terraform input variables support the following types:

- Basic types: string, number, and bool
- Compound types: list(<TYPE>), set(<TYPE>), map(<TYPE>)

The following example defines a variable of the compound type:

```
variable "availability_zone_names" {
  type    = list(string)
  default = []
}

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

Custom Validation Rules

You can use the **validation** nested block to specify custom validation rules for an input variable. This feature is supported in Terraform 0.13.0 and later versions. Example:

```
variable "iam_user_password" {
    type        = string
    description = "The password for iam user to log in."

    validation {
      condition     = length(var.iam_user_password)>=8
      error_message = "The password is too short."
    }
}
```

The **condition** argument is a Boolean expression. You can use a **can** function to check whether an error will be caused by the expression. Example:

```
variable "iam_user_name" {
    type        = string
    description = "This name is used for iam user to log in."

    validation {
      # regex(...) If the variable fails to match the following condition, an error is returned.
      condition     = can(regex("([a-zA-Z])", var.iam_user_name))
      error_message = "Incorrect user name. Please check whether it contains upper and lower case letters."
    }
}
```

If the result of **condition** is **false**, Terraform generates an error message that contains the character string defined by **error_message**. The value of **error_message** must include at least a complete sentence that starts with an uppercase letter and ends with a period (.) or question mark (?).

Referencing Input Variables

An input variable can be accessed as **var.<*Variable name*>** and only in the module that declares it.

```
# variables.tf
variable "vpc_cidr" {
  type        = string
  description = "the CIDR of VPC"
}

# main.tf
```

Setting Variables

You can set input variables in either of the following ways:

- With the **-var** command line option.
- In variable definitions (**.tfvars**) files, either specified on the command line or automatically loaded.
- As environment variables.

Variable Definitions (.tfvars) Files

If many variables are used in the configuration, you are advised to set their values in a variable definitions file, and then use the **-var-file** option to specify that file.

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

A variable definitions (.tfvars) file uses the same basic syntax as the configuration files, but consists only of variable name assignments:

```
vpc_name = "my_vpc"
vpc_cidr = "192.168.0.0/16"
availability_zone_names = [
]
```

Terraform also automatically loads variable definitions files if they are present:

- Files named exactly terraform.tfvars or terraform.tfvars.json
- Any files with names ending in .auto.tfvars or .auto.tfvars.json

Files whose names end with .json are parsed instead as JSON objects.

```
{
   "vpc_name": "my_vpc"
}
```

Variable Definition Precedence

The above mechanisms for setting variables can be used together in any combination. For variables of the compound type, you are advised to use the variable definitions file to improve readability and avoid problems caused by escape. If you assign multiple values to the same variable, Terraform uses the last value it finds, overriding any previous values. Terraform loads variables in the following order, with later sources taking precedence over earlier ones:

1. Environment variables
2. terraform.tfvars or terraform.tfvars.json file
3. *.auto.tfvars or *.auto.tfvars.json file
4. **-var** and **-var-file** options in the command line

Note that the same variable cannot be assigned multiple values within a single source.

For more information about variables, see **Input Variables** in the Terraform documentation.

3.4.2 Output Variables

Output variables are like return values for a module. They are declared using the keyword **output**. Output variables can expose certain information. They can be used by a root module to output certain values after running the **terraform apply/output** command, or by a child module to expose a subset of its resource attributes to a parent module.

Declaring Output Variables

By convention, output variables are defined in a file named **variables.tf**. Output variables are declared using the keyword **output**.

```
output "ecs_address" {
  description = "The private IP address of my ECS"
}
```

The label immediately after the **output** keyword is the name, which must be a valid identifier. The **output** block contains the following arguments:

- **value** (mandatory): value of the output variable. Any valid expression is allowed as an output value.
- **description**: describes the usage of an output variable.
  ```
  output "vpc_id" {
    description = "Check out the VPC ID"
  }
  ```
- **sensitive**: marks output variables as sensitive and hides the output variable values on the CLI.
  ```
  output "vpc_id" {
    description = "Check out the VPC ID"
    sensitive   = true
  }

  $ terraform output
  vpc_id = <sensitive>
  ```
  Note: Output variables marked as sensitive are automatically hidden during output, but their output values can still display in the following ways:
  – The values of output variables are recorded in the **state** file and are visible to anyone who can access the file.

  - The sensitive output variable values in a child module can be invoked by its parent module and displayed on the CLI after being referenced by the related outputs and resources of the parent module.
- **depends_on**: specifies the dependency of an output variable. Since output variables are only a means of exporting data, you do not need to set the dependencies between output variables and other resources or data.

3.4.3 Local Variables

Local values are like temporary variables in a module. Their application scope is in the declared module. They are declared by the keyword **locals**. Local variables can be helpful to reduce code redundancy and make code easy to modify in scenarios where the same values or expressions are repeatedly defined in the configuration. However, if local variables are overused, the actual values are hidden, making code hard to read by future maintainers. Therefore, use local variables properly.

Declaring Local Variables

Local variables are declared using the keyword **locals**.

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

Expressions of local variables are not limited to character and numeric constants. They can also use references and expression results of input variables, resource attributes, or other local values.

```
locals {
}

locals {
  common_tags = {
    Service = local.service_name
    Owner   = local.owner
  }
}
```

Referencing Local Variables

After declaring a local variable, you can use **local.<*Variable name*>** to reference it.

```
  ...
  tags = local.common_tags
}
```

3.5 Metadata

3.5.1 Instruction

Metadata refers to built-in meta-arguments supported by Terraform and can be used in the **provider**, **resource**, and **data** blocks. This section describes the meta-arguments supported by the **resource** block, including:

- **depends_on**: specifies the dependencies of a resource.
- **count**: creates multiple resources with the same configuration.
- **for_each**: creates multiple resources based on mappings and string sets.
- **lifecycle**: customizes the lifecycle of a resource.

3.5.2 depends_on

A Terraform configuration file can contain multiple resources.

By referencing the attribute values of other resources in a resource, Terraform can automatically infer the dependencies of the resource. However, the dependencies of some resources are invisible to Terraform. Therefore, **depends_on** is needed to create explicit dependencies. You can use **depends_on** to change the creation or execution sequence of resources so that the resources are processed after the dependent resources.

The expression of **depends_on** is the address list of dependent resources.

3.5.3 count

By default, only one resource is configured for a **resource** block of Terraform. When multiple same resources need to be created, configuring multiple independent **resource** blocks is redundant and difficult to maintain. You can use the **count** or **for_each** arguments to manage multiple identical resources in the same **resource** block. A given resource block cannot use both **count** and **for_each**. Example:

Three identical EVS disks are created based on the preceding configurations. In many cases, the provider requires that some arguments for creating resources be unique. You can use the **count.index** attribute (an index value starting from **0**) to distinguish the arguments.

```
}
```

Two VPCs (**myvpc_0** and **myvpc_1**) with the same CIDR value are created based on the preceding configuration. To modify the CIDR value, you can declare a string list to store the CIDR values of different VPCs, and then use **count.index** to access the list elements.

```
variable "name_list" {
  type    = list(string)
  default = ["vpc_demo1", "vpc_demo2"]
}
variable "cidr_list" {
  type    = list(string)
  default = ["192.168.0.0/16", "172.16.0.0/16"]
}

  count = 2
  name  = var.name_list[count.index]
  cidr  = var.cidr_list[count.index]
}
```

An index is required to access a resource created using **count**. The format is <*Resource type*>.<*Name*>[*Index*].

```
> huaweicloud_vpc.vpcs[0]

# ID for accessing the first VPC
> huaweicloud_vpc.vpcs[0].id
```

```
# ID for accessing all VPCs
> huaweicloud_vpc.vpcs[*].id
```

3.5.4 for_each

The function of **for_each** is similar to that of **count**. **for_each** uses key-value pairs or string sets to quickly fill values in corresponding attributes. This optimizes the script structure and helps understand the relationship between multiple instances.

When using the mapping type expression, you can use **each.key** and **each.value** to access the key and value of the mapping. For example, to create a VPC, you can use the key-value pair in **for_each** to flexibly configure the VPC name and CIDR.

```
  for_each = {
    vpc_demo1 = "192.168.0.0/16"
    vpc_demo2 = "172.16.0.0/16"
}

  name = each.key
  cidr = each.value
}
```

When a string set is used, **each.key** is equivalent to **each.value** and generally **each.key** is used. In addition, you can use the **toset()** function to convert the defined **list** type.

```
  for_each = toset(["secgroup_demo1", "secgroup_demo2"])
  name     = each.key
}

# Use variables to indicate for _each.
variable "secgroup_name" {
  type = set(string)
}
  for_each = var.secgroup_name
  name     = each.key
}
```

A key is required to access a resource created using **for_each**. The format is <*Resource type*>.<*Name*>[*Key*].

```
# Access vpc_demo1.
# ID for accessing vpc_demo1
```

Both **count** and **for_each** can be used to create multiple resources. You are advised to select either of them based on the following rules:

1. If the arguments of a resource instance are completely or mostly the same, you are advised to use count.

2. If some arguments of a resource need to use distinct values that cannot be directly derived from an integer, **for_each** is recommended.

3.5.5 provider

In Terraform, you can use **provider** blocks to create multiple configurations, among which, one block is the default configuration, and other blocks are labeled as non-default configurations using **alias**. You can use the meta-argument **provider** in a resource to select a non-default **provider** block. For example, to manage resources in different regions, you need to declare multiple **provider** blocks.

```
provider "huaweicloud" {
  region = "cn-north-1"
  ...
}

provider "huaweicloud" {
  alias  = "guangzhou"
  region = "cn-south-1"
  ...
}
```

In the example, Huawei Cloud providers in Beijing and Guangzhou are declared, and the provider in Guangzhou is labeled with an alias. You can use the meta-argument **provider** in a resource to select a non-default **provider** block in the format of ***<Provider name>.<Alias>***.

```
resource "huaweicloud_networking_secgroup" "mysecgroup" {
# Use the name and alias of the non-default provider block.
  provider = huaweicloud.guangzhou
  ...
}
```

Huawei Cloud providers allow you to specify the **region** argument in a resource to create resources in different regions. Compared with labeling providers with aliases, this mode is more flexible and simple.

```
provider "huaweicloud" {
  region = "cn-north-1"
  ...
}

resource "huaweicloud_vpc" "example" {
  region = "cn-south-1"
  name   = "terraform_vpc"
  cidr   = "192.168.0.0/16"
}
```

3.5.6 lifecycle

Resource instances have three phases: creation, update, and destruction. The lifecycle of a resource instance involves two or three of the phases. The meta-argument **lifecycle** can be used to modify the lifecycle of a resource instance and includes the following arguments:

- **create_before_destroy**

  By default, when you need to change a resource argument that cannot be updated, Terraform destroys the existing instance and then uses the newly configured arguments to create a new object for replacement. When you set the **create_before_destroy** argument to **true**, Terraform creates a new instance before destroying the existing instance. This argument applies to scenarios where service continuity needs to be ensured. Ensure in advance that each resource instance has a unique name and other constraints are met so that old and new instances can co-exist.

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

- **prevent_destroy**

  When **prevent_destroy** is set to **true**, Terraform blocks the deletion of the resource and returns an error. This meta-argument can be used as a security measure to prevent high-cost instances, such as database instances, from being recreated due to unexpected operations. To delete the resource, you need to delete the configuration and then perform the destroy operation.

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

- **ignore_changes**

  By default, the **Terraform plan/apply** operation detects the differences between the cloud resource attributes and the local resource blocks. If they are inconsistent, the update or rebuild operation is invoked to match the configuration. You can use **ignore_changes** specify arguments that Terraform should ignore when planning updates or rebuilds. The value of **ignore_changes** can be the relative address list of the attributes. The Map and List elements can be referenced using index notation, such as **tags["Name"]** and **list[0]**.

  ```
   ...
   lifecycle {
     ignore_changes = [
       name,
     ]
   }
  }
  ```

  In this case, Terraform ignores the modification of the **name** argument. In addition to the list, you can also use the keyword **all** to ignore the updates of all attributes.

  ```
   ...
   lifecycle {
     ignore_changes = all
   }
  }
  ```

4 Template Constraints and Limitations

Using RFS to deploy templates, there are several constraints listed as follows:

- The number of modules is limited to 25 and the module nesting depth is limited to 3.
- The **Provisioners**, **Backend Configuration** and **Cloud** features are not supported.
- The **Module Sources** feature is supported. However, you can only use Local Modules.
- The **HuaweiCloud Provider** is supported with certain prohibited resources which are listed as follows:
  - huaweicloud_vod_watermark_template
  - huaweicloud_compute_keypair
  - huaweicloud_identity_access_key
  - huaweicloud_images_image_v2
  - huaweicloud_kps_keypair
  - huaweicloud_obs_bucket_object
  - huaweicloud_iotda_batchtask
  - huaweicloud_cce_chart
  - huaweicloud_iotda_batchtask_file
  - huaweicloud_cse_microservice
- The following built-in functions are prohibited:
  - **abspath**
  - **basename**
  - **dirname**
  - **file**
  - **filebase64**
  - **filebase64sha256**
  - **filebase64sha512**
  - **fileexists**
  - **fileset**

- **filemd5**
- **filesha1**
- **filesha256**
- **filesha512**
- **pathexpand**
- **templatefile**

WARNING

The use of the nonsensitive method to output sensitive information is not recommended. Random use of this method may result in sensitive information being printed out in plaintext by the service when it should have been hidden, leading to leakage of sensitive information.If output is necessary, it is recommended to prioritize encoding before outputting (e.g., nonsensitive(sha256(var.sensitive_value)))