

HashiCorp Certified: Terraform Associate (004) 学習用テキスト

細部まで動かして覚える —— 読んで理解する1冊目

無料サンプル(第1章まで)

HashiCorp「Terraform Associate (004) exam content list」(8 分野 37 小項目・
Terraform 1.12) 準拠
2026年10月版

ご利用にあたって

- このテキストは「IT資格道場」(www.shikaku-doujou.com)の購入者限定テキストのうち、学習用テキストの第1章までを収録した**無料サンプル**です。どなたでもご覧いただけます。
- 無料サンプルも著作物です。転載・改変しての再配布・二次販売はできません。
- AI モデルの学習データとしてのご利用(学習目的の入力・データセット化を含む)はできません。
- 製品版(学習用テキスト・頻出度別ノート・暗記用ノートの3点)は、ご購入者を識別できる透かし入りのPDFとして提供されます。
- 本PDFはスマートフォン・タブレットでの閲覧に合わせたA5版面・文字サイズで組版しています。

目次

この教材群の使い方(3点セット)

この試験が問うもの

第1章 Infrastructure as Code (IaC) と Terraform [1.1/1.2/1.3]

IaC と宣言的・命令的の違い [1.1]

構成ドリフトと再現性 [1.1]

バージョン管理・レビュー・自動化の利点 [1.2]

幂等性と環境の一貫性、モジュールによる再利用 [1.2]

プロバイダによる複数基盤の管理 [1.3]

Terraform が管理できる対象と限界 [1.3]

サンプルはここまでです

この教材群の使い方(3点セット)

種類	役割
① 学習用テキスト	これだけで問題が解けるようになる本編
② 頻出度別ノート	テキストを読んだら次はこれ。頻出順に絞った問題と解説で「解ける」に橋渡し。試験直前の最終チェックにも
③ 暗記用ノート	覚えるべき要点だけを一問一答に凝縮。空き時間の反復と用語の再確認用

使い方: ① 学習用を読む → ② 頻出度別で解き方を掴む → ③ 問題演習で腕試し → ④ 頻出度別に戻って再確認(試験直前もここ)。暗記用は空き時間や用語の再確認に。

この試験が問うもの

HashiCorp Certified: Terraform Associate (004) は、Terraform を使ったインフラ管理の基本を問う認定試験です。出題範囲は公式が 8 つの分野と 37 の小項目で公開していて、このテキストの章立ても同じ順番です。

項目	内容
運営	HashiCorp
試験時間	1 時間
出題形式	選択式(2 つ選ぶ複数選択を含む)
受験方式	オンライン監督
試験の言語	英語
対象のバージョン	Terraform 1.12
配点	分野ごとの配点は公式に公表されていません

出題範囲は次の 8 分野です。

- 第1章 Infrastructure as Code (IaC) と Terraform
- 第2章 Terraform の基礎(プロバイダ・state)
- 第3章 Terraform の基本ワークフロー
- 第4章 Terraform の構成(HCL)
- 第5章 モジュール
- 第6章 state の管理
- 第7章 インフラの保守(import・state 操作・ログ)

- 第8章 HCP Terraform

コマンドの挙動、HCL の書き方と評価の結果、state と backend の扱い、モジュール、HCP Terraform の機能を問う試験です。例のコードは、手元の Terraform 1.12 で一度実行して、出力を確かめてください。HCP Terraform の設定は画面や API の話なので、断片のコードで示します。

SAMPLE

第1章 Infrastructure as Code (IaC) と Terraform

[1.1/1.2/1.3]

この章の内容: IaC の考え方、IaC を使う利点、Terraform がクラウドや SaaS をまたいで同じ手順で管理できる仕組み。[^1]

IaC と宣言的・命令的の違い [1.1]

IaC (Infrastructure as Code、コードとしてのインフラ) は、サーバーやネットワークや DNS の構成をファイルに書き、そのファイルから環境を作る考え方です。画面を手で操作して作った環境には、誰がいつ何を変えたかが残りません。同じものをもう一度作るのは簡単ではありません。コードにすれば、これらをソースコードと同じやり方で扱えます。

IaC のツールは、書き方で二つに分かれます。命令的 (imperative) な方式は、「インスタンスを作る、タグを付ける、ロードバランサーに登録する」という手順を順番に書きます。クラウドの CLI を呼ぶシェルスクリプトが典型です。手順を並べるのは命令的な方式で、宣言的な方式ではありません。宣言的 (declarative) な方式は、できあがりの状態だけを書きます。そこへ至る手順はツールが決めます。Terraform は宣言的で、CloudFormation のテンプレートや Kubernetes のマニフェストも同じ側です。

命令的なスクリプトは、実行時点の現状を自分で調べなければなりません。「すでにあれば作らない」という分岐も自分で書きます。宣言的な方式では、現状と記述の差分を求める仕事をツールが引き受けます。Terraform の構成ファイルは HCL (HashiCorp Configuration Language) で書き、リソースごとに `resource` ブロックを置きます。

次の構成は、組み込みの `terraform_data` を 1 つ宣言しています。書いてあるのは「この名前のリソースが 1 つある」という最終形だけで、作成の手順はありません。

```
resource "terraform_data" "web" {  
  input = "web-server"  
}
```

```
# terraform_data.web will be created  
+ resource "terraform_data" "web" {  
Plan: 1 to add, 0 to change, 0 to destroy.
```

`terraform plan` は、記述と現状の差分から必要な操作を並べます。まだ何も作っていないので、ここでは作成 (+) が 1 件になります。`terraform apply` はその計画を実行します。差分のあるリソースだけが変更され、差分がなければ何も起きません。実行のたびに全リソースを消して作り直すことはありません。読み取り専用の道具でもなく、記述した状態へ向けて作成、変更、削除を行います。

構成ファイルの言語は、扱うサービスが AWS でも Azure でも Datadog でも HCL です。サービスごとに言語を書き分けて自動変換する仕組みはありません。

構成ドリフトと再現性 [1.1]

構成ドリフト (configuration drift) は、コードの記述と実機の状態がずれていくことです。Terraform で管理しているセキュリティグループに、運用担当者がコンソールから手作業でポートを 1 つ開けると、ドリフトが生じます。

コードを変えずに `terraform plan` を実行すると、Terraform は管理下のリソースの現在の状態を実機から読み取り、コードの記述と比べます。手作業で開けたポートは差分として検出されます。コードを正として扱うので、コードが自動で書き換わることはありません。計画には「開けたポートを閉じて記述どおりに戻す」変更が出ます。

管理下のリソースへの手作業は、Terraform の管理の外にはなりません。`plan` に表示されないこともありません。`state` ファイルが壊れたとみなされて、エラーで止まることもありません。`apply` まで進めると、手作業の変更はコードの記述で上書きされます。残したい変更は、先にコード側へ書き足してから `apply` します。

ドリフトを防ぐ取り組みと、早く見つける取り組みは、次の二つに絞られます。

- コンソールからの手作業を原則禁止し、変更をコードの修正とレビューに一本化する。
- スケジュール実行などで定期的に `terraform plan` を回し、差分を通知する。

`terraform validate` は構成の検査、`terraform fmt` は書式の整形です。どちらも実機とは比べません。`state` ファイルを手で書き換えて実機に合わせる方法は、ずれを隠すだけで、ドリフトの対策になりません。HCP Terraform にも、ドリフトを検出するヘルスアセスメント (health assessment) の機能があります。

再現性は、同じコードから同じ構造の環境を何度でも作れる性質です。環境ごとの違い(台数やサイズなど)は、コードを複製せず、変数に入れて切り替えます。

```
# 実行: plan -var env=test

variable "env" {
  type = string
}

resource "terraform_data" "app" {
  input = "app-${var.env}"
}
```

```
+ input = "app-test"

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

`-var env=prod` に替えれば、同じコードのまま本番用の値で計画が出ます。環境ごとにコードをコピーして別々に保守すると、時間とともに内容がずれ、構造の一貫性は自然には保たれません。

バージョン管理・レビュー・自動化の利点 [1.2]

構成をコードにして Git などで管理すると、変更の履歴が残ります。誰がいつ何を変えたかが記録され、プルリクエストでレビューしてから反映できます。問題があれば、以前の版へ戻す判断もしやすくなります。ブランチやタグといった既存の開発の流れが、そのまま使えます。

コミットしただけでは、リソースは作られません。反映には `terraform apply` が必要です。ブランチを分けるだけでクラウドの利用料金が下がることもありません。`state` ファイルは機密値を含み得るので、コードと同じリポジトリには置きません。共有できる保存先（リモートの `backend`）に置きます。

自動化の代表は、プルリクエストのたびに CI で `terraform plan` を実行し、結果をレビューに添える運用です。何が作られ、変わり、消えるのかを、マージする前に関係者全員が確認できます。マージ後に `apply` を自動で実行すれば、本番の状態を、コードの履歴と一致させ続けられます。

`plan` の結果は、レビューの代わりにはなりません。構文チェックだけで本番へ反映する運用でもありません。`plan` はプロバイダの API の仕様変更の影響を受けます。共有の `state` の保存先を用意しないと、複数人で同じインフラを管理できません。

冪等性と環境の一貫性、モジュールによる再利用 [1.2]

冪等性 (idempotency) は、同じ構成を何度適用しても、すでに望ましい状態にあるものには変更を加えず、結果が同じになる性質です。同じ構成を `apply` してから、コードも実機も変えずにもう一度 `apply` した場合を考えます。差分がないので何も変わらず、`No changes.` と表示されて終わります。リソースが作り直されることはありません。同じものが重複して作られることもありません。リソースの ID が振り直されず、`state` のロックが残ってエラーになることもありません。

この性質があるので、パイプラインで `apply` を繰り返しても状態は変わらず、再実行が安全です。手作業で変更されたリソースは、冪等性によって `state` から除外されるわけではありません。次の `plan` で差分として見つかります。

同じ構成を複数のプロジェクトで使い回す単位は、モジュール (module) です。関連するリソースをまとめ、入力変数と出力値で受け渡します。呼び出す側は、値を変えて何度でも使えます。プロバイダやワークスペースやバックエンドは、構成をまとめて再利用する単位ではありません。似た名前の機能との違いは、次のとおりです。

機能	役割
モジュール (module)	関連するリソースをまとめて再利用する単位
プロバイダ (provider)	サービスの API を扱うプラグイン
ワークスペース (workspace)	同じ構成で <code>state</code> を切り替える仕組み
バックエンド (backend)	<code>state</code> の保存先と操作の方法を決める設定

モジュールは、`module` ブロックの `source` で場所を指定して呼び出します。次の例は、同じ子モジュールを環境名だけ変えて 2 回使っています。

```
# ファイル: web/main.tf
variable "env" {
  type = string
}

resource "terraform_data" "server" {
  input = "server-${var.env}"
}
```

```
module "dev" {  
  source = "../web"  
  env    = "dev"  
}
```

```
module "prod" {  
  source = "../web"  
  env    = "prod"  
}
```

```
# module.dev.terraform_data.server will be created  
# module.prod.terraform_data.server will be created  
Plan: 2 to add, 0 to change, 0 to destroy.
```

リソースのアドレスに `module.dev.` のような接頭辞が付き、同じ子モジュールから作った 2 組が区別されます。モジュールは Terraform Registry や Git リポジトリでも共有できますが、`state` までは共有されません。

プロバイダによる複数基盤の管理 [1.3]

マルチクラウド (multi-cloud) は複数のクラウドを使う形態で、ハイブリッドクラウド (hybrid cloud) はオンプレミスとクラウドを組み合わせる形態です。Terraform は、どちらも同じワークフローで扱えます。対象はクラウドだけではありません。Datadog や Cloudflare のような SaaS も、プロバイダ経由で管理できます。

Terraform 本体 (Terraform Core) が受け持つのは、構成の読み込み、依存関係の解決、計画と適用という共通の流れです。AWS、Azure、Datadog、

Cloudflare といった各サービスの API とのやり取りは、プロバイダ (provider) というプラグインが担当します。プロバイダは、各サービスが公開している通常の API を呼びます。サービスの側に、Terraform 専用の管理窓口は要りません。本体に、あらゆるサービスの API 呼び出しが組み込まれているわけでもありません。新しいサービスに対応するときは、本体を更新せずにプロバイダを足します。

このため、性質の違うサービスが一つの構成に混在していても、書き方と操作は同じです。 `required_providers` に必要なプロバイダを並べ、 `init`、`plan`、`apply` の同じ流れで管理します。基盤ごとに別の製品や別のコマンド体系を使う必要はありません。次の例は、 `random` と `local` という二つのプロバイダのリソースを、一つの構成で参照し合っています。

```
terraform {  
  required_providers {  
    random = {  
      source = "hashicorp/random"  
    }  
    local = {  
      source = "hashicorp/local"  
    }  
  }  
}  
  
resource "random_pet" "name" {}  
  
resource "local_file" "memo" {  
  content = "host-${random_pet.name.id}"  
  filename = "${path.module}/memo.txt"  
}
```

```
# local_file.memo will be created  
# random_pet.name will be created  
Plan: 2 to add, 0 to change, 0 to destroy.
```

`random_pet.name.id` のように、別のプロバイダのリソースの属性を直接参照できます。出力値を手作業で転記する必要はありません。参照から依存関係が決まるので、作成の順番も Terraform が解決します。AWS のロードバランサ

一の DNS 名を Cloudflare の DNS レコードの値に渡す構成も、同じ書き方です。

複数のプロバイダを使っても、`state` が基盤ごとに強制的に分かれることはありません。1 つの構成には、既定では 1 つの `state` が対応します。

基盤を乗り換えるときは、書き直しが要ります。リソース型と引数はプロバイダごとに定義されるからです。`aws_instance` で書いた構成のプロバイダを `azurerm` に差し替えても、Azure の仮想マシンにはなりません。Azure 用のリソース型 (`azurerm_linux_virtual_machine` など) で書き直します。変わらないのは `init`、`plan`、`apply` という操作の流れです。Azure 専用のデプロイコマンドはありません。リソース型の名前が共通になることもありません。Terraform は基盤の違いを隠す抽象化ではなく、共通のワークフローを提供する道具です。

Terraform が管理できる対象と限界 [1.3]

Terraform が管理するのは、API で作成、変更、削除できるインフラやサービスの構成です。クラウドの仮想マシンやネットワークはもちろん、DNS のゾーンとレコード、Kubernetes のクラスター、GitHub のリポジトリとチームの権限、監視サービスの設定も対象です。「あるべき構成を作り、維持する」ことが得意な道具です。

稼働中のアプリケーションのログをリアルタイムに解析し、障害を検知するような実行時の処理は、本来の用途ではありません。サーバー内部の OS 設定やアプリの配備は、構成管理ツールの得意な領域です。Terraform だけで完全に置き換えられるわけではありません。

限界は次の三つです。

- プロバイダが対応していないサービスや機能は、Terraform の構成から管理できません。
- Terraform の外で作られたリソースは、`import` しない限り `state` に含まれず、管理対象になりません。
- 構成に書かれていない既存のリソースを、Terraform が見つけて自動で取り込むことはありません。

リソース型は、クラウド側が新機能を出しても自動では増えません。使うには、プロバイダが対応した版へ更新します。管理外のリソースを取り込むには、`import` ブロックか `terraform import` コマンドを使います。取り込んだあとは、コード側にも対応する記述を書き、差分がない状態にします。

サンプルはここまでです

続き(第2章～巻末)は、HashiCorp Certified: Terraform Associate (004) 問題集のご購入で、頻出度別ノート・暗記用ノートと合わせて3点すべてダウンロードできます。

SAMPLE

HashiCorp Certified: Terraform Associate (004) 学習用テキスト (無料サンプル)

版

2026年10月版(HashiCorp「Terraform Associate (004) exam content list」(8分野 37小項目・Terraform 1.12) 準拠)

発行

IT資格道場(<https://www.shikaku-doujou.com>)

お問い合わせ

info@shikaku-doujou.com

Terraform・HCP Terraform・Vault は HashiCorp, Inc. (IBM グループ) の商標または登録商標です。記載されているその他の会社名・製品名・サービス名は、各社の商標または登録商標です。

本教材は公式の出題範囲 (exam content list) と Terraform 1.12 の公式文書の内容を基に作成したもので、HashiCorp との提携・承認関係はありません。

本文・図表・設問はすべてオリジナルの著作物です。無断での転載・複製・再配布・二次販売はできません。お気づきの点は info@shikaku-doujou.com までご連絡ください。