# Rego 入門

OPA のポリシー言語と Conftest

2026-11-27

# 目次

**はじめに** 9
この本の構成 . . . . . 9
環境の準備 . . . . . 9
インストールせずに試す . . . . . 10
サンプルコード . . . . . 10
ライセンス . . . . . 11

**第 I 部　概要** 13

**第 1 章　OPA・Rego・Conftest の関係** 15
1.1　ポリシーの判断を切り離す . . . . . 15
1.2　Rego の部分と OPA の部分 . . . . . 16
1.3　Conftest の位置づけ . . . . . 17

**第 II 部　Rego 言語** 19

**第 2 章　はじめての Rego** 21
2.1　ポリシーを書いて評価する . . . . . 21
2.2　入力を使う . . . . . 22
2.3　複数の条件はすべて成り立つ必要がある . . . . . 24
2.4　同じ名前のルールを複数書くと OR になる . . . . . 25
2.5　ルールの値を条件で決める . . . . . 26
2.6　まとめ . . . . . 27

**第 3 章　スカラー値と文字列** 29
3.1　スカラー値 . . . . . 29
3.1.1　値を比べる . . . . . 30

3.2 文字列 . . . 31
3.3 組み込み関数を呼び出す . . . 32
3.4 文字列補間 . . . 34
3.4.1 埋め込む値が undefined のとき . . . 35
3.5 まとめ . . . 37

**第 4 章 複合値** 39
4.1 配列 . . . 39
4.2 オブジェクト . . . 41
4.2.1 文字列以外のキー . . . 43
4.3 存在しない要素を参照したとき . . . 44
4.4 集合 . . . 45
4.5 複合値を比べる . . . 47
4.6 まとめ . . . 47

**第 5 章 変数と参照** 49
5.1 変数 . . . 49
5.2 参照の中の変数で要素を探す . . . 50
5.2.1 ルール本体で要素を探す . . . 53
5.3 `some ... in` で要素を 1 つずつ取り出す . . . 53
5.4 複数の反復を組み合わせる . . . 55
5.4.1 同じ配列どうしを組み合わせる . . . 57
5.5 集合に値が含まれるかを調べる . . . 57
5.6 まとめ . . . 59

**第 6 章 内包表記** 61
6.1 配列の内包表記 . . . 63
6.2 集合の内包表記 . . . 64
6.3 オブジェクトの内包表記 . . . 65
6.4 内包表記の変数 . . . 67
6.5 まとめ . . . 67

**第 7 章 ルール** 69
7.1 集合を作るルール . . . 71
7.2 オブジェクトを作るルール . . . 72
7.3 値が 1 つに決まるルールの衝突 . . . 74
7.4 ルール名に参照を書く . . . 75

7.5 関数 . . . 77
7.6 まとめ . . . 78

**第 8 章 否定** 79
8.1 not の基本 . . . 79
8.2 not と != の違い . . . 81
8.3 集合に含まれないものを探す . . . 82
8.4 not の中の変数 . . . 84
8.5 関数の呼び出しを否定するときの注意 . . . 84
8.6 まとめ . . . 86

**第 9 章 すべての要素に対する条件** 87
9.1 every . . . 87
9.2 否定と count で書く . . . 90
9.3 要素や値がないとき . . . 91
9.4 まとめ . . . 92

**第 10 章 モジュール** 93
10.1 コメント . . . 93
10.2 パッケージ . . . 93
10.3 インポート . . . 95
10.3.1 データを読み込む . . . 97
10.3.2 キーワードのインポート . . . 98
10.4 input と data . . . 98
10.4.1 data は 1 つの木 . . . 98
10.4.2 input は 1 つの値 . . . 101
10.5 まとめ . . . 102

**第 11 章 キーワードの整理** 103
11.1 in で値が含まれるかを調べる . . . 103
11.1.1 ポリシーの中で使う . . . 104
11.2 if の省略形 . . . 105
11.3 and と or . . . 106
11.4 キーワードの一覧 . . . 108
11.5 まとめ . . . 109

**第 12 章 with** 111

12.1 入力とデータを差し替える . . . 111
12.2 ルールの中で使う . . . 112
12.3 関数を差し替える . . . 114
12.4 まとめ . . . 115

**第 13 章 `default` と `else` 117**
13.1 `default` . . . 117
13.1.1 関数の既定値 . . . 119
13.1.2 既定値に使える値 . . . 120
13.2 `else` . . . 120
13.3 `default` と `else` の使い分け . . . 122
13.4 まとめ . . . 122

**第 14 章 演算子 123**
14.1 代入 `:=` . . . 123
14.2 比較 `==` . . . 125
14.3 単一化 `=` . . . 125
14.4 使い分け . . . 127
14.5 まとめ . . . 127

**第 15 章 組み込み関数 129**
15.1 名前空間 . . . 129
15.2 算術と集合の演算子 . . . 129
15.3 よく使う関数 . . . 130
15.4 `sprintf` と文字列補間 . . . 133
15.5 実行時のエラー . . . 134
15.6 まとめ . . . 135

**第 16 章 メタデータ 137**
16.1 アノテーションの書き方 . . . 137
16.2 スコープ . . . 139
16.3 アノテーションを読み出す . . . 139
16.4 まとめ . . . 142

**第 17 章 スキーマと strict モード 143**
17.1 スキーマで入力の形を教える . . . 143
17.2 アノテーションでスキーマを指定する . . . 145

17.3 strict モード . . . . . 147
17.4 まとめ . . . . . 147

第 III 部 OPA と Conftest 149

第 18 章 OPA と Conftest の概要 151
18.1 OPA . . . . . 151
18.2 Conftest . . . . . 152

第 19 章 OPA の CLI 153
19.1 `opa eval` . . . . . 153
19.1.1 結果の表示形式 . . . . . 153
19.1.2 ディレクトリからデータを読み込む . . . . . 155
19.2 `opa fmt` と `opa check` . . . . . 157
19.3 `opa test` . . . . . 158
19.4 `print` でデバッグする . . . . . 160
19.5 まとめ . . . . . 161

第 20 章 Conftest は何を評価しているか 163
20.1 全体の流れ . . . . . 163
20.2 Rego 側で評価しているもの . . . . . 165
20.3 Conftest が値のどこを見るか . . . . . 166
20.3.1 名前の規則 . . . . . 169
20.3.2 テスト数の数え方 . . . . . 169
20.4 `deny[msg] if` に注意 . . . . . 169
20.5 例外 . . . . . 170
20.6 名前空間 . . . . . 172
20.7 まとめ . . . . . 172

第 21 章 Conftest の使い方 175
21.1 入力のファイル . . . . . 175
21.1.1 複数のドキュメントを含む YAML . . . . . 175
21.1.2 ファイルをまとめて評価する . . . . . 177
21.1.3 パーサーを指定する . . . . . 179
21.2 データを読み込む . . . . . 179
21.3 結果の表示と終了コード . . . . . 181
21.4 ポリシーをテストする . . . . . 182

21.5 まとめ . . . . . 183

**第 22 章 Kubernetes マニフェスト 185**
22.1 ドメインの情報: マニフェストの構造 . . . . . 185
22.2 Rego の書き方 . . . . . 188
22.2.1 コンテナを取り出す補助のルール . . . . . 190
22.2.2 各ルール . . . . . 190
22.2.3 検査とテスト . . . . . 191
22.3 まとめ . . . . . 192

**第 23 章 Terraform 193**
23.1 ドメインの情報: HCL のソースと plan . . . . . 193
23.1.1 HCL のソースの構造 . . . . . 194
23.1.2 plan の JSON の構造 . . . . . 196
23.2 Rego の書き方 . . . . . 199
23.2.1 HCL のソースを検査する . . . . . 199
23.2.2 plan の JSON を検査する . . . . . 201
23.3 まとめ . . . . . 203

**第 24 章 Dockerfile 205**
24.1 ドメインの情報: Dockerfile の構造 . . . . . 205
24.2 Rego の書き方 . . . . . 208
24.2.1 最終ステージを取り出す . . . . . 211
24.2.2 最後の `USER` を調べる . . . . . 211
24.2.3 `ADD` と `ENV` . . . . . 211
24.2.4 検査とテスト . . . . . 212
24.3 まとめ . . . . . 214

# はじめに

この本は、ポリシー言語 Rego の入門書です。Rego を初めて書く人から、少し書いたことがある人までを対象にしています。

Rego は、Open Policy Agent（OPA）で使われる言語です。「この API を呼んでよいか」「この Kubernetes マニフェストは社内ルールに従っているか」のような判断を、プログラムの本体から切り離してコードとして書けます。

## この本の構成

この本は 3 部に分かれています。

第 I 部「概要」では、OPA、Rego、Conftest の関係を説明します。Rego を学ぶときにつまずきやすいのは、「言語としての Rego」の話と「OPA というツールの使い方」の話の区別です。最初にこの区別をつけておきます。

第 II 部「Rego 言語」では、Rego の文法と考え方を説明します。OPA 公式ドキュメントの Policy Language と同じ順に進むので、読み終えた後は公式ドキュメントを辞書のように引けるはずです。ただし、まだ説明していない機能は使いません。そのために説明の順番を入れ替えた箇所があります。

第 III 部「OPA と Conftest」では、OPA の CLI と Conftest を使って、実際のファイルを検査するポリシーを書きます。Kubernetes や Terraform などの題材ごとに、そのドメインの知識と、それに合わせた Rego の書き方を分けて説明します。

## 環境の準備

この本の例は、次のバージョンで動作を確かめています。

| ツール | バージョン |
|---|---|
| OPA | 1.23.0 |
| Conftest | 0.71.1 |

第 II 部では OPA だけを使います。Conftest は第 III 部で使います。

macOS では Homebrew でインストールできます。

```
brew install opa conftest
```

Homebrew で入るバージョンは最新版より遅れることがあります。同じバージョンをそろえたい場合や、ほかの OS を使う場合は、OPA と Conftest の GitHub Releases からバイナリを入手してください。

インストールできたら、バージョンを確認します。

```
$ opa version
$ conftest --version
```

### インストールせずに試す

OPA の公式サイトには、ブラウザで Rego を書いて評価できる Rego Playground があります。ポリシーと入力を貼り付けて「Evaluate」を押すと、結果が表示されます。手元に OPA を入れる前に雰囲気をつかみたいときは、第 II 部の例をここで試せます。

## サンプルコード

本文の例は、すべて GitHub のリポジトリ の `examples/` に置いてあります。本文に載せている実行結果は、本をビルドするときに実際にコマンドを実行して得たものです。

例えば次のように表示されているときは、例のディレクトリで `$` の後のコマンドを実行すると、その下の出力が得られます。

```
$ opa eval -d policy.rego --format pretty 'data.greeting.message'
"こんにちは"
```

# ライセンス

この本の文章とサンプルコードは CC0 1.0 で公開しています。

# 第 I 部

# 概要

# 第 1 章

# OPA・Rego・Conftest の関係

Rego について調べると、「OPA」「Rego」「Conftest」という名前がまとめて出てきます。この章では、3 つがそれぞれ何を担当しているかを整理します。ここで区別しておくと、後の章で「これは言語の話か、ツールの話か」を迷わずに読めます。

## 1.1 ポリシーの判断を切り離す

アプリケーションには、「管理者だけが削除できる」「本番環境のコンテナは特権モードで動かしてはいけない」のような決まりがあります。この本では、こうした決まりをポリシーと呼びます。

ポリシーをアプリケーションのコードに直接書くと、決まりを変えるたびにアプリケーションを修正することになります。複数のシステムに同じ決まりを適用したいときは、それぞれに同じ判断を実装しなければなりません。

OPA（Open Policy Agent）は、この判断をアプリケーションから切り離すためのツールです。アプリケーションは判断に必要な情報を OPA に渡し、OPA はポリシーに従って結果を返します。結果をどう使うか（リクエストを拒否する、エラーを表示するなど）はアプリケーションが決めます。

OPA が判断に使うものは 3 つあります。

| 名前 | 内容 | 例 |
|---|---|---|
| 入力(input) | 判断のたびに渡される情報 | リクエストしたユーザーとメソッド、検査したいマニフェスト |
| データ(data) | 判断に使う、あらかじめ用意した情報 | ユーザーと役割の対応表 |
| ポリシー | 入力とデータから結果を計算する規則 | 「役割が admin なら削除を許可する」 |

このポリシーを書くための言語が Rego です。

## 1.2　Rego の部分と OPA の部分

Rego と OPA は一緒に語られることが多いですが、担当する範囲は分かれています。

Rego は言語です。言語として決まっているのは、ポリシーの書き方と、それを評価したときにどんな値になるかです。Rego のポリシーは、入力とデータを受け取って値を計算するだけです。ファイルを読んだり、ネットワークに結果を送ったりはしません。[*1]

OPA は、Rego を評価するエンジンと、その周りの仕組みをまとめたものです。次のような使い方ができます。

- `opa eval` や `opa test` などの CLI で、手元でポリシーを評価する・テストする。
- `opa run --server` でサーバーとして起動し、REST API で判断を問い合わせる。
- Go のライブラリとしてアプリケーションに組み込む。

どこから入力を受け取り、どのルールの値を問い合わせ、返ってきた値をどう扱うかは、OPA の使い方の側で決まります。Rego の言語にとっては、`allow` という名前のルールも、ほかの名前のルールと変わりません。`allow` の値を「許可するかどうか」として扱うのは、それを問い合わせる側です。

この本の第 II 部では言語としての Rego を、第 III 部では OPA の CLI と Conftest という使い方の側を説明します。第 II 部でも例を動かすために `opa eval` を使いますが、使い方は必要な範囲に絞ります。

[*1] 外部の HTTP API を呼ぶ組み込み関数など、例外もいくつかあります。

## 1.3 Conftest の位置づけ

Conftest は、設定ファイルを Rego のポリシーで検査するためのツールです。Kubernetes のマニフェスト、Terraform のコード、Dockerfile などを対象にできます。

Conftest は OPA を Go のライブラリとして組み込んでいます。ポリシーは Rego で書くので、第 II 部で学ぶ言語の知識はそのまま使えます。Conftest が OPA に加えているのは、主に次の 2 つです。

- 設定ファイルを読み込んで、Rego の入力に変換する。YAML や JSON だけでなく、HCL や Dockerfile、INI などの形式にも対応しています。
- 決まった名前のルール(`deny`、`warn` など)を問い合わせて、その値を検査の成功・失敗として表示する。

2 つ目は、前の節で説明した「どのルールを問い合わせ、値をどう扱うか」を Conftest が決めているということです。Rego 側で何を評価し、Conftest がその結果のどこを見ているかは、第 III 部で詳しく説明します。

Conftest に組み込まれている OPA のバージョンは、Conftest のバージョンごとに決まっています。例えば Conftest 0.71.1 には OPA 1.21.1 が組み込まれています。`conftest --version` で確認できます。

```
$ conftest --version
Conftest: 0.71.1
OPA: 1.21.1
```

そのため、OPA の最新版で追加された機能は、Conftest ではまだ使えないことがあります。

# 第 II 部

# Rego 言語

# 第 2 章

# はじめての Rego

この章では、小さなポリシーを書いて `opa eval` で評価します。Rego のポリシーがルールの集まりであること、ルールの値が条件によって決まること、条件が成り立たないときにどうなるかを確かめます。

## 2.1 ポリシーを書いて評価する

最初のポリシーです。`policy.rego` という名前で保存します。

**コード 2.1** `policy.rego`

```
package greeting


message := "こんにちは"
```

1 行目の `package greeting` は、このファイルのルールが属するパッケージの名前です。パッケージはルールをまとめる名前空間で、Rego のファイルには、先頭に必ず `package` を書きます。パッケージについては「モジュール」の章で詳しく説明します。

3 行目の `message := "こんにちは"` がルールです。Rego のポリシーは、ルールを並べて書いたものです。このルールは、`message` という名前に `"こんにちは"` という値を定めています。文字列は JSON と同じように二重引用符で囲みます。

このポリシーを評価してみます。

```
$ opa eval -d policy.rego --format pretty 'data.greeting.message'
"こんにちは"
$ opa eval -d policy.rego --format pretty 'data.greeting'
{
  "message": "こんにちは"
}
```

`opa eval` は、ポリシーを読み込んで、指定した式（クエリ）を評価するコマンドです。

- `-d policy.rego` で、読み込むポリシーのファイルを指定します。
- `--format pretty` で、結果を読みやすい形で表示します。この指定がないと、結果は詳しい情報を含んだ JSON で表示されます。
- 最後の引数がクエリです。

読み込んだルールは、`data.` **パッケージ名**`.` **ルール名** という名前で参照できます。1 つ目のコマンドは `data.greeting.message` を評価して、ルールの値 `"`**こんにちは**`"` を得ています。2 つ目のように `data.greeting` を評価すると、パッケージのルールがすべて、ルール名をキーにしたオブジェクトとして返ります。

## 2.2　入力を使う

ポリシーの判断は、入力によって変わるのが普通です。入力は `input` という名前で参照します。

次のポリシーは、入力のユーザーが管理者かどうかを判断します。

**コード 2.2** `policy.rego`

```
package access

is_admin if {
    input.user.role == "admin"
}
```

入力として、次の 2 つのファイルを用意します。

**コード 2.3** admin.json

```
{
  "user": {
    "name": "alice",
    "role": "admin"
  }
}
```

**コード 2.4** guest.json

```
{
  "user": {
    "name": "bob",
    "role": "guest"
  }
}
```

`input.user.role` は、入力のオブジェクトから `user` を取り出し、さらにその `role` を取り出す、という意味です。JavaScript などと同じく、ドット(.)でオブジェクトのフィールドをたどります。このように値をたどる書き方を参照と呼びます。参照のほかの書き方は「変数と参照」の章で説明します。

`is_admin if { ... }` は、波かっこの中の条件が成り立つときに値が決まるルールです。`if` より前をルールヘッド、波かっこの部分をルール本体と呼びます。ルールヘッドに値を書いていないときは、条件が成り立つと値が `true` になります。

`==` は、両辺が等しいかを調べる演算子です。ここでは `input.user.role` が `"admin"` と等しいかを調べています。

`-i` で入力のファイルを指定して評価します。

```
$ opa eval -d policy.rego -i admin.json --format pretty 'data.access.is_admin'
true
$ opa eval -d policy.rego -i guest.json --format pretty 'data.access.is_admin'
undefined
$ opa eval -d policy.rego -i guest.json --format pretty 'data.access'
{}
```

1 つ目のコマンドでは、`role` が `"admin"` なので、`is_admin` は `true` になりました。

2 つ目のコマンドの結果は `undefined` です。Rego では、ルール本体の条件が成り立たないと、ルールの値は決まりません。この状態を undefined と呼びます。

3 つ目のコマンドで、パッケージ全体を評価すると `{}` になりました。値の決まらなかった `is_admin` は、結果のオブジェクトに含まれません。undefined は `false` という値ではなく、値がないことを表します。

条件が成り立たないときに `false` などの値を返したい場合は、既定値を指定する書き方があります。「`default` と `else`」の章で説明します。

## 2.3　複数の条件はすべて成り立つ必要がある

ルール本体には、条件を複数行に分けて書けます。次のルールは、ユーザーが管理者であり、メソッドが `DELETE` であり、多要素認証をしてから 10 分以内のときに `true` になります。入力の `mfa_age_minutes` は、多要素認証をしてからの経過時間(分)とします。

**コード 2.5** `policy.rego`

```
package access

can_delete if {
    input.user.role == "admin"
    input.method == "DELETE"
    input.user.mfa_age_minutes <= 10
}
```

`<=` は「左辺が右辺以下」を表す比較演算子です。Rego には、ほかの言語と同じ比較演算子があります。

| 演算子 | 意味 |
|---|---|
| `==` | 等しい |
| `!=` | 等しくない |
| `<` | より小さい |
| `<=` | 以下 |
| `>` | より大きい |

| 演算子 | 意味 |
|---|---|
| >= | 以上 |

ルール本体の条件は、すべて成り立つ必要があります。つまり、各行は AND でつながっています。`mfa_age_minutes` だけが違う 2 つの入力で評価してみます。

```
$ opa eval -d policy.rego -i recent.json --format pretty 'data.access.can_delete'
true
$ opa eval -d policy.rego -i stale.json --format pretty 'data.access.can_delete'
undefined
```

`stale.json` では `mfa_age_minutes` が 30 なので、3 つ目の条件が成り立たず、ルールの値は undefined になりました。ほかの 2 つの条件が成り立っていても、1 つでも成り立たなければ値は決まりません。

## 2.4　同じ名前のルールを複数書くと OR になる

条件のどれか 1 つが成り立てばよい場合は、同じ名前のルールを複数書きます。次のポリシーでは、メソッドが `GET` であるか、ユーザーが管理者であれば `allow` が `true` になります。

**コード 2.6** `policy.rego`

```
package access

allow if {
    input.method == "GET"
}

allow if {
    input.user.role == "admin"
}
```

```
$ opa eval -d policy.rego -i get.json --format pretty 'data.access.allow'
true
$ opa eval -d policy.rego -i post.json --format pretty 'data.access.allow'
```

```
undefined
```

get.json は管理者ではありませんが、メソッドが GET なので 1 つ目の定義の条件が成り立ち、allow は true になりました。post.json はどちらの条件も満たさないので undefined です。

1 つのルール本体の中の条件は AND、同じ名前の複数のルールは OR、と覚えておくと、Rego の条件の組み立て方が読めるようになります。

## 2.5　ルールの値を条件で決める

ルールヘッドに値を書くと、条件が成り立ったときにその値になります。また、ルール本体の中でほかのルールを参照することもできます。

次のポリシーは、ユーザーの契約プランから、作成できるプロジェクトの上限を決めます。

**コード 2.7** policy.rego

```
package quota

is_paid if {
    input.user.plan != "free"
}

max_projects := 100 if {
    is_paid
}

max_projects := 3 if {
    input.user.plan == "free"
}
```

max_projects := 100 if { ... } は、条件が成り立つと max_projects が 100 になるルールです。max_projects は 2 つ定義されていて、どちらか条件の成り立ったほうの値になります。

1 つ目の max_projects の本体では、is_paid を参照しています。同じパッケージのルールは、ルール名だけで参照できます。is_paid の値が true なら、この条件は成り立ちます。

is_paid が undefined のときは、条件は成り立ちません。

```
$ opa eval -d policy.rego -i pro.json --format pretty 'data.quota'
{
  "is_paid": true,
  "max_projects": 100
}
$ opa eval -d policy.rego -i free.json --format pretty 'data.quota'
{
  "max_projects": 3
}
```

free.json では、is_paid の条件が成り立たないので、is_paid は結果に含まれていません。それでも max_projects には、2 つ目の定義から 3 が決まりました。

同じ名前のルールの条件が同時に成り立ち、しかも値が異なる場合はエラーになります。この例ではプランが "free" かどうかで条件が分かれているので、そうはなりません。このエラーについては「ルール」の章で説明します。

## 2.6 まとめ

- Rego のポリシーは、package で始まり、ルールを並べたものです。ルールは data.**パッケージ名**.**ルール名** で参照します。
- ルールは、ルール本体の条件がすべて成り立ったときに値が決まります。成り立たないときの値は false ではなく undefined です。
- 同じ名前のルールを複数書くと、どれかの条件が成り立てば値が決まります。

# 第 3 章

# スカラー値と文字列

前の章では、ルールの値として文字列、数値、真偽値を使いました。この章では、Rego で扱える値のうち、それ以上分解できない値（スカラー値）を説明します。後半では文字列を詳しく見ていきます。文字列を操作するために、組み込み関数の呼び出し方もここで説明します。

## 3.1 スカラー値

Rego の値は JSON の値と対応しています。スカラー値は、文字列、数値、真偽値、`null` の 4 種類です。

**コード 3.1** `policy.rego`

```
package server

name := "web"

port := 8080

cpu_ratio := 0.75

enabled := true

owner := null
```

```
$ opa eval -d policy.rego --format pretty 'data.server'
{
  "cpu_ratio": 0.75,
  "enabled": true,
  "name": "web",
  "owner": null,
  "port": 8080
}
```

それぞれのルールの値が、JSON の値としてそのまま返ってきました。

| 種類 | 例 | 説明 |
|---|---|---|
| 文字列 | `"web"` | 二重引用符で囲む |
| 数値 | `8080`、`0.75` | 整数と小数の区別はない |
| 真偽値 | `true`、`false` | |
| null | `null` | 値がないことを表す JSON の `null` |

`null` は、前の章で説明した undefined とは別のものです。`owner := null` の `owner` には `null` という値があり、結果のオブジェクトにも含まれています。undefined のルールは、結果に含まれませんでした。

### 3.1.1　値を比べる

`opa eval` は、`-d` でポリシーを指定しなくても使えます。クエリに式を直接書くと、その値を表示します。小さな動作を確かめたいときに便利です。

```
$ opa eval --format pretty '8080 == 8080.0'
true
$ opa eval --format pretty '8080 == "8080"'
false
$ opa eval --format pretty '"apple" < "banana"'
true
```

1 つ目のように、`8080` と `8080.0` は等しくなります。Rego の数値は、整数と小数を区別しないからです。

2 つ目の `8080` と `"8080"` は、数値と文字列なので等しくなりません。Rego は比較のとき

に型を変換しません。入力の JSON で数値が文字列として書かれていると、数値との比較は成り立たないので注意してください。

3 つ目のように、文字列どうしも < などで比べられます。文字列は先頭から 1 文字ずつ、文字のコード（Unicode のコードポイント）の順に比べられます。英小文字だけの文字列なら、辞書と同じ順になります。

## 3.2　文字列

文字列の書き方は 2 つあります。

1 つは、これまで使ってきた二重引用符の文字列です。JSON の文字列と同じく、バックスラッシュ（\）でエスケープを書きます。例えば \" は二重引用符、\\ はバックスラッシュ、\n は改行を表します。

もう 1 つは、バッククォート（`）で囲むローストリングです。ローストリングの中では、バックスラッシュはエスケープにならず、書いたとおりの文字になります。また、文字列の途中で改行できます。

**コード 3.2** policy.rego

```
package paths

home := "C:\\Users\\alice"

home_raw := `C:\Users\alice`

same if {
    home == home_raw
}

banner := `Welcome!
Have a nice day.`
```

```
$ opa eval -d policy.rego --format pretty 'data.paths'
{
  "banner": "Welcome!\nHave a nice day.",
  "home": "C:\\Users\\alice",
```

```
  "home_raw": "C:\\Users\\alice",
  "same": true
}
```

home と home_raw は、同じ文字列 C:\Users\alice を表しています。二重引用符の文字列では、バックスラッシュを \\ と 2 つ重ねて書く必要がありました。ローストリングでは、書いたとおりの文字になります。結果の JSON では、どちらも "C:\\Users\\alice" とエスケープして表示されています。

バックスラッシュを多く含む文字列を書くときは、ローストリングが読みやすくなります。正規表現が代表的な例です。正規表現を使う組み込み関数は「組み込み関数」の章で紹介します。

banner は、ローストリングの途中で改行した例です。値には改行(\n)が含まれています。

## 3.3　組み込み関数を呼び出す

Rego には、文字列の変換や数値の計算などを行う組み込み関数が多数用意されています。ここでは呼び出し方を説明します。どんな関数があるかは「組み込み関数」の章で紹介します。

組み込み関数は、**関数名(引数 1, 引数 2, ...)** の形で呼び出します。次の例では、文字列を大文字にする upper と、文字列が指定の文字列で始まるかを調べる startswith を使っています。

**コード 3.3** policy.rego

```
package naming

name := "web-server"

display_name := upper(name)

is_web if {
    startswith(name, "web-")
}

is_db if {
    startswith(name, "db-")
}
```

```
$ opa eval -d policy.rego --format pretty 'data.naming'
{
  "display_name": "WEB-SERVER",
  "is_web": true,
  "name": "web-server"
}
```

`upper(name)` は、`name` を大文字にした文字列を返します。`display_name` の値は、その戻り値の `"WEB-SERVER"` になりました。

`startswith(name, "web-")` は、`name` が `"web-"` で始まるので `true` を返します。ルール本体に書いた式の値が `true` なら、その条件は成り立ちます。

`is_db` が結果に含まれていないのは、`startswith(name, "db-")` が `false` を返したからです。ルール本体の式が `false` になると、`==` が成り立たなかったときと同じく、条件は成り立ちません。そのため `is_db` は undefined になります。

組み込み関数は、引数の型が決まっています。`upper` に数値を渡すと、評価の前にエラーになります。

**コード 3.4** policy.rego

```
package naming

port := 8080

display_port := upper(port)
```

```
$ opa eval -d policy.rego --format pretty 'data.naming'
1 error occurred: policy.rego:5: rego_type_error: upper: invalid argument(s)
        have: (number, ???)
        want: (x: string, y: string)
```

rego_type_error は、型が合わないことを表すエラーです。have は実際の型、want は関数が期待する型です。かっこの中の 2 つ目は戻り値の型を表していて、upper は文字列を受け取り、文字列を返す関数だと分かります。have の戻り値が ??? なのは、型が合わないため戻り値の型を決められなかったことを表しています。

文字列を操作する組み込み関数には、ほかに次のようなものがあります。

| 関数 | 説明 |
|---|---|
| lower(s) | 小文字にする |
| endswith(s, suffix) | s が suffix で終わるかを返す |
| replace(s, old, new) | s に含まれる old を new に置き換える |
| trim_space(s) | 前後の空白を取り除く |

## 3.4　文字列補間

文字列の中に値を埋め込みたいときは、文字列補間を使います。二重引用符の前に $ を付け、埋め込みたい式を波かっこ（{ }）で囲みます。

**コード 3.5** policy.rego

```
package greeting

name := "alice"

port := 8080

message := $"Hello, {upper(name)}!"

url := $"http://localhost:{port}/"

template := $"use \{name} to insert a value"
```

```
$ opa eval -d policy.rego --format pretty 'data.greeting'
{
  "message": "Hello, ALICE!",
  "name": "alice",
  "port": 8080,
  "template": "use {name} to insert a value",
  "url": "http://localhost:8080/"
}
```

message では、{upper(name)} の部分が upper(name) の値に置き換わりました。このように、波かっこの中には、ルールの名前だけでなく、関数の呼び出しなどの式も書けます。

url のように、数値などの文字列以外の値は、文字列に変換されて埋め込まれます。

波かっこそのものを文字列に含めたいときは、template のように \{ とエスケープします。

### 3.4.1　埋め込む値が undefined のとき

文字列補間で埋め込む値が undefined の場合、文字列全体は undefined になりません。その部分が <undefined> という文字列に置き換わります。

**コード 3.6** policy.rego

```
package greeting


message := $"role: {input.user.role}"
```

**コード 3.7** input.json

```
{
  "user": {
    "name": "alice"
  }
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.greeting.message'
"role: \u003cundefined\u003e"
$ opa eval -d policy.rego -i input.json --format raw 'data.greeting.message'
role: <undefined>
```

入力に role がないので、input.user.role は undefined です。1 つ目のコマンドの結果では、< と > が \u003c と \u003e というエスケープで表示されています。これは --format pretty が結果を JSON として表示するときのエスケープで、値は role: <undefined> です。

2 つ目のコマンドの --format raw は、結果の文字列を引用符やエスケープなしでそのまま表示する形式です。こちらでは <undefined> がそのまま表示されています。

値が欠けていてもメッセージを作れるので、エラーメッセージを組み立てるときに便利です。ただし、条件の判断に使う値が undefined になっていないかは、別に確かめる必要があります。

文字列補間は OPA 1.12.0 で追加された機能です。それより古い OPA や、古い OPA を組み込んだツールでは使えません。この本で使う Conftest 0.71.1 には OPA 1.21.1 が組み込まれているので、使えます。

## 3.5 まとめ

- スカラー値は、文字列、数値、真偽値、`null` の 4 種類で、JSON の値と対応しています。比較のときに型は変換されません。
- 文字列は二重引用符かバッククォートで囲みます。バッククォートのローストリングでは、バックスラッシュがエスケープになりません。
- 組み込み関数は **関数名**`(`**引数**`, ...)` で呼び出します。ルール本体の式が `false` になると、その条件は成り立ちません。
- `$"..."` の文字列補間で、文字列に式の値を埋め込めます。

# 第 4 章

# 複合値

前の章では、それ以上分解できないスカラー値を扱いました。この章では、複数の値をまとめた複合値を説明します。複合値には、配列、オブジェクト、集合の 3 種類があります。配列とオブジェクトは JSON と同じものです。集合は JSON にはない、Rego 独自の値です。

複合値の要素には、スカラー値だけでなく、別の複合値や、ほかのルールの値も入れられます。

## 4.1 配列

配列は、値を順番に並べたものです。JSON と同じく、角かっこ（[ ]）の中に要素をカンマで区切って書きます。

コード 4.1 policy.rego

```
package network

http_port := 80

https_port := 443

ports := [http_port, https_port]

first_port := ports[0]

listeners := [
    ["web", 80],
    ["api", 8080],
]

api_port := listeners[1][1]
```

```
$ opa eval -d policy.rego --format pretty 'data.network'
{
  "api_port": 8080,
  "first_port": 80,
  "http_port": 80,
  "https_port": 443,
  "listeners": [
    [
      "web",
      80
    ],
    [
      "api",
      8080
    ]
  ],
  "ports": [
```

```
    80,
    443
  ]
}
```

ports は、ルール http_port と https_port の値を要素にした配列です。結果では、それぞれのルールの値 80 と 443 に置き換わっています。

配列の要素は、ports[0] のように角かっこで番号（添字）を指定して取り出します。添字は 0 から始まるので、ports[0] は先頭の要素 80 です。

listeners は、配列を要素に持つ配列です。listeners[1][1] は、2 番目の要素 ["api", 8080] の、さらに 2 番目の要素 8080 を取り出しています。このように、添字を続けて書くと、入れ子になった値をたどれます。listeners のように、1 つの配列に文字列と数値など異なる型の値を混ぜることもできます。

## 4.2　オブジェクト

オブジェクトは、キーと値の組を集めたものです。JSON と同じく、波かっこ（{ }）の中に **キー: 値** の組をカンマで区切って書きます。

コード 4.2 policy.rego

```
package app

server := {
    "name": "web",
    "port": 8080,
    "labels": {
        "team": "platform",
        "app.kubernetes.io/name": "web-frontend",
    },
}

name := server.name

team := server.labels.team

app_name := server.labels["app.kubernetes.io/name"]
```

```
$ opa eval -d policy.rego --format pretty 'data.app'
{
  "app_name": "web-frontend",
  "name": "web",
  "server": {
    "labels": {
      "app.kubernetes.io/name": "web-frontend",
      "team": "platform"
    },
    "name": "web",
    "port": 8080
  },
  "team": "platform"
}
```

値は、server.name のようにドットの後にキーを書いて取り出します。前の章までに input.user.role のように使ってきた書き方です。server.labels.team のように、ドット

を続けて入れ子の値をたどれます。

キーが app.kubernetes.io/name のように . や / を含む場合、ドットの後には書けません。server.labels.app.kubernetes.io/name と書くと、. は値をたどる区切りとして、/ は割り算の記号として読まれてしまうからです。このようなときは、server.labels["app.kubernetes.io/name"] のように、角かっこの中にキーを文字列で書きます。

ドットの書き方と角かっこの書き方は同じ意味です。server.name は server["name"] とも書けます。ただし、opa fmt（Rego のコードを整形するコマンド）にかけると、ドットで書けるものはドットの書き方に直されます。

### 4.2.1 文字列以外のキー

JSON のオブジェクトのキーは文字列だけですが、Rego のオブジェクトでは数値などもキーにできます。

**コード 4.3** policy.rego

```
package network

services := {
    80: "http",
    443: "https",
}

https_service := services[443]
```

```
$ opa eval -d policy.rego --format pretty 'data.network'
{
  "https_service": "https",
  "services": {
    "443": "https",
    "80": "http"
  }
}
```

services[443] のように、数値のキーは数値で指定して取り出します。結果の JSON では

キーが `"443"` のように文字列で表示されていますが、これは JSON のキーが文字列でなければならないためです。Rego の中では、数値の `443` と文字列の `"443"` は別のキーです。

## 4.3　存在しない要素を参照したとき

配列の範囲外の添字や、オブジェクトにないキーを指定したときは、値は undefined になります。

**コード 4.4** `policy.rego`

```
package app

second_port := input.ports[1]

third_port := input.ports[2]

name := input.server.name

owner := input.server.owner
```

**コード 4.5** `input.json`

```
{
  "ports": [80, 443],
  "server": {
    "name": "web"
  }
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.app'
{
  "name": "web",
  "second_port": 443
}
```

`input.ports` には要素が 2 つしかないので、`input.ports[2]` は undefined です。`input.server` には `owner` というキーがないので、`input.server.owner` も undefined で

す。どちらのルールも、結果に含まれていません。

ただし、ポリシーの中で定数として書いた配列やオブジェクトに対して同じことをすると、評価の前にエラーになります。

**コード 4.6** policy.rego

```
package app

ports := [80, 443]

third_port := ports[2]
```

```
$ opa eval -d policy.rego --format pretty 'data.app'
1 error occurred: policy.rego:5: rego_type_error: undefined ref: data.app.ports[2]
        data.app.ports[2]
                       ^
                       index 2 out of bounds for array of length 2
```

OPA は、ポリシーを評価する前に、ルールの値の型を調べています（型検査）。前の章で `upper` に数値を渡してエラーになったのも、この型検査です。`ports` は定数なので、要素が 2 つしかないことが評価の前に分かります。そのため、`ports[2]` は必ず undefined になる誤りとして報告されます。

一方、入力の中身は評価するまで分かりません。そのため、`input.ports[2]` はエラーにならず、評価したときに undefined になります。

## 4.4 集合

集合は、重複のない値の集まりです。波かっこの中に、キーなしで値だけをカンマで区切って書きます。

**コード 4.7** policy.rego

```
package team

primary_role := "admin"

secondary_role := "admin"

roles := {"dev", primary_role, secondary_role}

role_count := count(roles)

no_roles := set()

empty_object := {}
```

```
$ opa eval -d policy.rego --format pretty 'data.team'
{
  "empty_object": {},
  "no_roles": [],
  "primary_role": "admin",
  "role_count": 2,
  "roles": [
    "admin",
    "dev"
  ],
  "secondary_role": "admin"
}
```

roles には primary_role と secondary_role を入れましたが、どちらも値は "admin" です。集合は同じ値を 1 つしか持たないので、roles の要素は "admin" と "dev" の 2 つになりました。

count は、要素の数を返す組み込み関数です。count(roles) は 2 になりました。count は、配列やオブジェクト（キーの数）、文字列（文字数）にも使えます。

集合には順番がありません。結果では配列の形で表示されていますが、これは JSON に集

合を表す書き方がないためです。表示される順番は、書いた順番とは限りません。

空の集合は `set()` と書きます。`{}` と書くと、空のオブジェクトになります。結果の `no_roles` は空の配列 `[]`、`empty_object` は空のオブジェクト `{}` として表示されています。

集合に特定の値が含まれるかを調べる方法は、「変数と参照」の章で説明します。

## 4.5 複合値を比べる

複合値も `==` で比べられます。順番の扱いは種類によって違います。

```
$ opa eval --format pretty '[1, 2] == [2, 1]'
false
$ opa eval --format pretty '{1, 2} == {2, 1}'
true
$ opa eval --format pretty '{"a": 1, "b": 2} == {"b": 2, "a": 1}'
true
$ opa eval --format pretty 'count([1, 2, 2])'
3
$ opa eval --format pretty 'count({1, 2, 2})'
2
```

配列は順番に意味があるので、同じ要素でも並び順が違えば等しくなりません。集合とオブジェクトには順番がないので、要素やキーと値の組が同じなら等しくなります。

最後の 2 つは、同じ要素を `count` で数えた結果です。配列は重複を残すので `3`、集合は重複をまとめるので `2` になりました。

## 4.6 まとめ

- 配列は値を順番に並べたもので、`ports[0]` のように 0 から始まる添字で要素を取り出します。
- オブジェクトはキーと値の組の集まりで、`server.name` または `server["name"]` で値を取り出します。キーには文字列以外の値も使えます。
- 集合は重複のない値の集まりで、空の集合は `set()` と書きます。
- 存在しない要素を参照すると undefined になります。定数に対する参照は、評価の前に型検査でエラーになります。

# 第 5 章

# 変数と参照

これまでの例では、`input.user.role` のように、決まった場所の値だけを取り出していました。実際のポリシーでは、「サーバーの一覧の中に本番環境のものがあるか」のように、配列やオブジェクトの要素を 1 つずつ調べる必要があります。

この章では、ルール本体で使う変数と、値をたどる書き方（参照）を説明します。参照の中に変数を書くと、要素を 1 つずつ調べる反復になります。

## 5.1 変数

ルール本体では、**変数名 := 値** で変数に値を代入できます。

**コード 5.1** `policy.rego`

```
package greeting


message := msg if {
    name := input.user.name
    role := input.user.role
    msg := $"{name} ({role})"
}
```

**コード 5.2** input.json

```
{
  "user": {
    "name": "alice",
    "role": "admin"
  }
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.greeting'
{
  "message": "alice (admin)"
}
```

name、role、msg はルール本体の中だけで使える変数です。msg := $"{name} ({role})" のように、代入した変数は後の式で使えます。ルールヘッドの message := msg のように、ルールの値にも使えます。

変数には一度しか代入できません。同じ変数に 2 回代入すると、エラーになります。

**コード 5.3** policy.rego

```
package greeting

message := msg if {
    msg := "hello"
    msg := "goodbye"
}
```

```
$ opa eval -d policy.rego --format pretty 'data.greeting'
1 error occurred: policy.rego:5: rego_compile_error: var msg assigned above
```

## 5.2　参照の中の変数で要素を探す

前の章までの参照では、ports[0] や server.name のように、取り出す場所を決めて書いていました。参照の添字やキーの位置に変数を書くと、OPA はその式が成り立つ値を変数に当てはめて探します。

ここからは、次の入力を使います。サーバーの一覧と、チームが担当するサーバーの一覧です。

**コード 5.4** input.json

```
{
  "servers": [
    {
      "name": "web-1",
      "env": "prod",
      "ports": [80, 443]
    },
    {
      "name": "web-2",
      "env": "staging",
      "ports": [80]
    },
    {
      "name": "db-1",
      "env": "prod",
      "ports": [5432]
    }
  ],
  "teams": [
    {
      "name": "frontend",
      "servers": ["web-1", "web-2"]
    },
    {
      "name": "data",
      "servers": ["db-1"]
    }
  ]
}
```

`opa eval` のクエリに、変数を含む参照を書いてみます。

```
$ opa eval -i input.json --format pretty 'input.servers[i].name'
┌──────────────────────────┬───┐
│ input.servers[i].name    │ i │
├──────────────────────────┼───┤
│ "web-1"                  │ 0 │
│ "web-2"                  │ 1 │
│ "db-1"                   │ 2 │
└──────────────────────────┴───┘
$ opa eval -i input.json --format pretty 'input.servers[i].env == "prod"'
┌───┐
│ i │
├───┤
│ 0 │
│ 2 │
└───┘
$ opa eval -i input.json --format pretty 'input.servers[_].name'
┌──────────────────────────┐
│ input.servers[_].name    │
├──────────────────────────┤
│ "web-1"                  │
│ "web-2"                  │
│ "db-1"                   │
└──────────────────────────┘
```

1 つ目のクエリ `input.servers[i].name` では、`i` に 0、1、2 が当てはめられました。結果の表は、`i` の値ごとに、参照の値を並べたものです。このように、変数を含む式は、成り立つ値の組み合わせが複数あると、そのすべてを結果として返します。

2 つ目のクエリでは、`input.servers[i].env == "prod"` が成り立つ `i` だけが返りました。`env` が `"staging"` の `i = 1` は、`==` が成り立たないので結果に含まれません。

3 つ目の `_` は、値を使わない変数です。添字に興味がなく、要素だけを調べたいときに使います。`_` を複数書いた場合、それぞれ別の変数として扱われます。

### 5.2.1 ルール本体で要素を探す

ルール本体に変数を含む式を書くと、「その式が成り立つ値が 1 つでもあれば条件が成り立つ」という意味になります。

**コード 5.5** policy.rego

```
package servers

has_staging if {
    some i
    input.servers[i].env == "staging"
}

has_dev if {
    input.servers[_].env == "dev"
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
{
  "has_staging": true
}
```

has_staging は、env が "staging" のサーバーが 1 つあるので true になりました。has_dev は、"dev" のサーバーがないので undefined です。

has_staging の some i は、i がこのルール本体で使う変数であることを宣言しています。some を書かなくても動きますが、同じパッケージに i という名前のルールがあると、i はそのルールの値を指してしまいます。参照の中で変数を使うときは、some で宣言しておくと安全です。_ は宣言しなくても、常に値を使わない変数として扱われます。

## 5.3 some ... in で要素を 1 つずつ取り出す

添字を使わずに、配列の要素を直接変数に取り出す書き方もあります。some **変数** in **配列** と書くと、配列の要素が 1 つずつ変数に当てはめられます。

コード 5.6 policy.rego

```
package servers

has_staging if {
    some server in input.servers
    server.env == "staging"
}

first_is_web_1 if {
    some i, server in input.servers
    i == 0
    server.name == "web-1"
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
{
  "first_is_web_1": true,
  "has_staging": true
}
$ opa eval --format pretty 'some x in ["a", "b"]'
┌─────┐
│  x  │
├─────┤
│ "a" │
│ "b" │
└─────┘
$ opa eval --format pretty 'some i, x in ["a", "b"]'
┌───┬─────┐
│ i │  x  │
├───┼─────┤
│ 0 │ "a" │
│ 1 │ "b" │
└───┴─────┘
$ opa eval --format pretty 'some k, v in {"x": 1, "y": 2}'
┌─────┬───┐
```

```
|  k  | v |
|-----|---|
| "x" | 1 |
| "y" | 2 |
+-----+---+

$ opa eval --format pretty 'some k, x in {"a", "b"}'

+-----+-----+
|  k  |  x  |
|-----|-----|
| "a" | "a" |
| "b" | "b" |
+-----+-----+
```

has_staging は、前の節と同じ判断を some ... in で書いたものです。some server in input.servers で各サーバーを server に取り出し、次の行でその env を調べています。添字の参照を何度も書かずに済むので、読みやすくなります。

some i, server in input.servers のように変数を 2 つ書くと、1 つ目に添字、2 つ目に要素が当てはめられます。

後半のクエリのように、some ... in は配列のほか、オブジェクトや集合にも使えます。集合には添字やキーがないので、変数を 2 つ書くと、どちらにも同じ要素が当てはめられます。

| 書き方 | 配列 | オブジェクト | 集合 |
|---|---|---|---|
| some x in **値** | x に要素 | x に値 | x に要素 |
| some k, x in **値** | k に添字、x に要素 | k にキー、x に値 | k と x に要素 |

この本では、これ以降、要素を 1 つずつ調べるときは主に some ... in を使います。in には、値が含まれるかを調べる使い方もあります。そちらは「キーワードの整理」の章で説明します。

## 5.4　複数の反復を組み合わせる

1 つのルール本体に、反復を複数書けます。

**コード 5.7** policy.rego

```
package servers

prod_https if {
    some server in input.servers
    server.env == "prod"
    some port in server.ports
    port == 443
}

db_owner := team.name if {
    some team in input.teams
    some name in team.servers
    name == "db-1"
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
{
  "db_owner": "data",
  "prod_https": true
}
```

prod_https は、本番環境のサーバーの中に、443 番ポートを使うものがあるかを調べています。まず some server in input.servers で各サーバーを取り出し、env が "prod" のものに絞ります。次に、そのサーバーの ports から some port in server.ports でポートを取り出し、443 と比べています。すべての式が同時に成り立つ server と port の組み合わせが見つかれば、条件が成り立ちます。

db_owner は、db-1 を担当するチームの名前を求めています。チームを team に、そのチームが担当するサーバーの名前を name に取り出し、name が "db-1" になる組み合わせを探します。見つかった team の name が、ルールの値になります。

db_owner は値が 1 つに決まるルールなので、条件を満たすチームが複数あって名前が異なると、エラーになります。条件を満たす値をすべて集めるルールは、「ルール」の章で説明します。

### 5.4.1　同じ配列どうしを組み合わせる

同じ配列から要素を 2 回取り出して、要素どうしを比べることもできます。次のルールは、名前が重複しているサーバーがあるかを調べます。

**コード 5.8** policy.rego

```
package servers

duplicate_name if {
    some i, a in input.servers
    some j, b in input.servers
    i != j
    a.name == b.name
}
```

```
$ opa eval -d policy.rego -i ok.json --format pretty 'data.servers.duplicate_name'
undefined
$ opa eval -d policy.rego -i dup.json --format pretty 'data.servers.duplicate_name'
true
```

a と b には、それぞれサーバーが 1 つずつ当てはめられます。i != j で同じ要素どうしの組み合わせを除き、名前が等しい組み合わせを探しています。dup.json では 1 番目と 3 番目のサーバーの名前が同じなので、true になりました。

## 5.5　集合に値が含まれるかを調べる

前の章で予告した、集合に値が含まれるかを調べる方法です。集合に対して **集合 [値]** と参照すると、その値が集合に含まれていれば値そのものが返り、含まれていなければ undefined になります。

コード 5.9 policy.rego

```
package access

admins := {"alice", "bob"}

is_admin if {
    admins[input.user]
}

grants := {
    ["alice", "read"],
    ["bob", "write"],
}

allow if {
    grants[[input.user, input.action]]
}

has_any_grant if {
    grants[[input.user, _]]
}
```

コード 5.10 alice.json

```
{
  "user": "alice",
  "action": "write"
}
```

```
$ opa eval -d policy.rego -i alice.json --format pretty 'data.access'
{
  "admins": [
    "alice",
    "bob"
  ],
  "grants": [
```

```
    [
      "alice",
      "read"
    ],
    [
      "bob",
      "write"
    ]
  ],
  "has_any_grant": true,
  "is_admin": true
}
$ opa eval -d policy.rego --format pretty 'data.access.admins["alice"]'
"alice"
$ opa eval -d policy.rego --format pretty 'data.access.admins["carol"]'
undefined
```

後半の 2 つのクエリのように、admins["alice"] は "alice" を返し、admins["carol"] は undefined になります。

ルール本体の admins[input.user] は、input.user が "alice" なので "alice" を返します。ここで、ルール本体の式がいつ成り立つかを整理しておきます。式は、値が undefined でも false でもなければ成り立ちます。"alice" は false ではないので、is_admin の条件は成り立ちます。

grants は、ユーザー名と操作の組を配列にして、その配列を要素にした集合です。grants[[input.user, input.action]] は、["alice", "write"] という配列が grants に含まれるかを調べています。含まれていないので、allow は undefined になり、結果に含まれていません。

has_any_grant の grants[[input.user, _]] のように、配列の一部に _ を書くと、「1 つ目の要素が alice の組がどれか 1 つでもあるか」を調べられます。2 つ目の要素は何でもかまいません。

## 5.6 まとめ

- ルール本体では := で変数に値を代入できます。代入は 1 回だけです。

- 参照の中に変数を書くと、式が成り立つ値が探されます。ルール本体では、成り立つ値が 1 つでもあれば条件が成り立ちます。
- `some x in` **値** で、配列・オブジェクト・集合の要素を 1 つずつ取り出せます。反復を複数書くと、すべての式が同時に成り立つ組み合わせが探されます。
- ルール本体の式は、値が undefined でも `false` でもなければ成り立ちます。**集合**`[`**値**`]` は、値が含まれていれば成り立ちます。

# 第 6 章

# 内包表記

前の章では、反復を使って「条件を満たす要素が 1 つでもあるか」を調べました。この章では、条件を満たす要素をすべて集めて、配列・集合・オブジェクトを作る内包表記を説明します。

この章でも、前の章と同じサーバーとチームの入力を使います。

コード 6.1 input.json

```
{
  "servers": [
    {
      "name": "web-1",
      "env": "prod",
      "ports": [80, 443]
    },
    {
      "name": "web-2",
      "env": "staging",
      "ports": [80]
    },
    {
      "name": "db-1",
      "env": "prod",
      "ports": [5432]
    }
  ],
  "teams": [
    {
      "name": "frontend",
      "servers": ["web-1", "web-2"]
    },
    {
      "name": "data",
      "servers": ["db-1"]
    }
  ]
}
```

## 6.1　配列の内包表記

配列の内包表記は、**[値 | 本体]** の形で書きます。本体には、ルール本体と同じように条件を書きます。本体が成り立つ変数の値の組み合わせごとに、| の左の値を求め、それを並べた配列を作ります。

**コード 6.2** policy.rego

```
package servers

names := [server.name | some server in input.servers]

prod_names := [server.name |
    some server in input.servers
    server.env == "prod"
]

staging_names := [server.name | some server in input.servers; server.env == "staging"]

dev_names := [server.name | some server in input.servers; server.env == "dev"]

has_dev if {
    count(dev_names) > 0
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
{
  "dev_names": [],
  "names": [
    "web-1",
    "web-2",
    "db-1"
  ],
  "prod_names": [
    "web-1",
```

```
    "db-1"
  ],
  "staging_names": [
    "web-2"
  ]
}
```

names は、some server in input.servers で取り出した各サーバーの server.name を集めた配列です。

prod_names のように、本体には複数の式を書けます。ルール本体と同じく、すべての式が成り立つ組み合わせだけが集められます。本体の式は、prod_names のように改行で区切るか、staging_names のように ; で区切ります。

dev_names は、env が "dev" のサーバーがないので、空の配列 [] になりました。ここがルールとの大きな違いです。ルールは条件が成り立たないと undefined になりましたが、内包表記は、集める値が 1 つもなくても空の配列という値になります。

この性質を使うと、has_dev のように、集めた値の数で判断できます。count(dev_names) は 0 なので、count(dev_names) > 0 は成り立たず、has_dev は undefined になりました。

## 6.2　集合の内包表記

集合の内包表記は、{**値** | **本体**} の形で書きます。集め方は配列と同じですが、結果は集合なので、同じ値は 1 つにまとめられます。

**コード 6.3** policy.rego

```
package servers

envs := {server.env | some server in input.servers}

all_ports := {port |
    some server in input.servers
    some port in server.ports
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
{
  "all_ports": [
    80,
    443,
    5432
  ],
  "envs": [
    "prod",
    "staging"
  ]
}
```

envs は、各サーバーの env を集めた集合です。"prod" のサーバーは 2 台ありますが、集合なので "prod" は 1 つだけです。

all_ports では、本体に反復を 2 つ書いています。各サーバーの各ポートを集めているので、全サーバーで使われているポートの集合になります。80 は 2 台のサーバーで使われていますが、1 つにまとめられています。

## 6.3 オブジェクトの内包表記

オブジェクトの内包表記は、**{キー: 値 | 本体}** の形で書きます。本体が成り立つ組み合わせごとに、キーと値の組を作ります。

**コード 6.4** policy.rego

```
package servers

ports_by_name := {server.name: server.ports | some server in input.servers}

names_by_env := {env: names |
    some env in {server.env | some server in input.servers}
    names := [server.name | some server in input.servers; server.env == env]
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
{
  "names_by_env": {
    "prod": [
      "web-1",
      "db-1"
    ],
    "staging": [
      "web-2"
    ]
  },
  "ports_by_name": {
    "db-1": [
      5432
    ],
    "web-1": [
      80,
      443
    ],
    "web-2": [
      80
    ]
  }
}
```

ports_by_name は、サーバー名をキー、ポートの配列を値にしたオブジェクトです。

names_by_env は、env ごとにサーバー名を集めたオブジェクトです。内包表記の中に、内包表記を 2 つ書いています。

- 本体の 1 行目では、集合の内包表記で env の集合を作り、some env in でその要素を 1 つずつ取り出しています。
- 2 行目では、配列の内包表記で、env が同じサーバーの名前を集めて names に代入しています。内側の内包表記の中から、外側の変数 env を参照しています。

オブジェクトのキーは重複できません。同じキーに異なる値が集まると、エラーになります。

**コード 6.5** policy.rego

```
package servers


name_by_env := {server.env: server.name | some server in input.servers}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
1 error occurred: policy.rego:3: eval_conflict_error: object keys must be unique

Traceback:
  policy.rego:3: server.name
  policy.rego:3: {server.env: server.name | some server in input.servers}
  1:1: data.servers
```

name_by_env は、env をキー、サーバー名を値にしようとしています。"prod" のサーバーは web-1 と db-1 の 2 台あるので、キー "prod" に 2 つの異なる値が集まり、エラーになりました。

1 つのキーに複数の値をまとめたいときは、前の節の names_by_env のように、値の側を内包表記にして配列などにまとめます。

## 6.4 内包表記の変数

内包表記の本体で使った変数は、その内包表記の中だけで使えます。names_by_env では、2 つの内包表記の両方で server という変数を使っていますが、それぞれ別の変数として扱われます。

一方、names_by_env の内側の内包表記が env を参照していたように、内包表記の本体からは外側の変数を参照できます。

## 6.5 まとめ

- **[値 | 本体]** は配列、**{値 | 本体}** は集合、**{キー: 値 | 本体}** はオブジェクトを作る内包表記です。
- 本体はルール本体と同じように書き、本体が成り立つ組み合わせごとに値を集めます。

- 集める値がなくても、結果は undefined ではなく空の配列・集合・オブジェクトになります。
- オブジェクトの内包表記で同じキーに異なる値が集まると、エラーになります。

# 第 7 章

# ルール

これまでのルールは、`allow` や `max_projects` のように、値が 1 つに決まるものでした。この章では、条件を満たす値をすべて集めて集合やオブジェクトを作るルールと、引数を受け取る関数を説明します。値が 1 つに決まるルールについても、条件が同時に成り立ったときのエラーなど、後回しにしていた点を整理します。

この章でも、サーバーとチームの入力を使います。

コード 7.1 input.json

```
{
  "servers": [
    {
      "name": "web-1",
      "env": "prod",
      "ports": [80, 443]
    },
    {
      "name": "web-2",
      "env": "staging",
      "ports": [80]
    },
    {
      "name": "db-1",
      "env": "prod",
      "ports": [5432]
    }
  ],
  "teams": [
    {
      "name": "frontend",
      "servers": ["web-1", "web-2"]
    },
    {
      "name": "data",
      "servers": ["db-1"]
    }
  ]
}
```

## 7.1　集合を作るルール

**ルール名** `contains` **値** `if { 本体 }` と書くと、本体が成り立つ組み合わせごとに値を集めて、集合を作るルールになります。

**コード 7.2** `policy.rego`

```
package servers

prod_names contains server.name if {
    some server in input.servers
    server.env == "prod"
}

dev_names contains server.name if {
    some server in input.servers
    server.env == "dev"
}

open_ports contains port if {
    some server in input.servers
    some port in server.ports
}

open_ports contains 22
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
{
  "dev_names": [],
  "open_ports": [
    22,
    80,
    443,
    5432
  ],
```

```
  "prod_names": [
    "db-1",
    "web-1"
  ]
}
```

prod_names は、env が "prod" のサーバーの名前を集めた集合です。前の章の集合の内包表記 {server.name | some server in input.servers; server.env == "prod"} と同じ値になります。

dev_names は、条件を満たすサーバーがないので、空の集合（結果では []）になりました。contains のルールは、集める値が 1 つもなくても undefined にならず、空の集合になります。この点も内包表記と同じです。

open_ports は 2 回定義しています。1 つ目は全サーバーのポートを集め、2 つ目は本体のない open_ports contains 22 で、常に 22 を加えます。同じ名前の contains のルールを複数書くと、それぞれが集めた値をすべて合わせた集合になります。このように、1 つのルールを複数の定義に分けて書くことを増分定義と呼びます。

この本では、prod_names や open_ports のように複数の値を集めるルールを複数値ルール、max_projects のように値が 1 つに決まるルールを単一値ルールと呼びます。

## 7.2 オブジェクトを作るルール

**ルール名** [**キー**] := **値** if { **本体** } と書くと、本体が成り立つ組み合わせごとにキーと値の組を作り、オブジェクトを作るルールになります。

**コード 7.3** policy.rego

```
package servers

ports_by_name[server.name] := server.ports if {
    some server in input.servers
}

ports_by_name["bastion"] := [22]
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
{
  "ports_by_name": {
    "bastion": [
      22
    ],
    "db-1": [
      5432
    ],
    "web-1": [
      80,
      443
    ],
    "web-2": [
      80
    ]
  }
}
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers.ports_by_name["web-1"]'
[
  80,
  443
]
```

`ports_by_name` は、サーバー名をキー、ポートの配列を値にしたオブジェクトです。2 つ目の定義で、`"bastion"` というキーを追加しています。集合のルールと同じく、定義を複数書くと、それぞれが作った組を合わせたオブジェクトになります。

2 つ目のコマンドのように、`ports_by_name["web-1"]` と参照すると、キー `"web-1"` の値を取り出せます。

条件を満たす組がないときは、空のオブジェクトになります。また、オブジェクトの内包表記と同じく、同じキーに異なる値が作られるとエラーになります。

**コード 7.4** policy.rego

```
package servers

name_by_env[server.env] := server.name if {
    some server in input.servers
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
1 error occurred: policy.rego:3: eval_conflict_error: object keys must be unique

Traceback:
  policy.rego:3: server.name
  1:1: data.servers
```

## 7.3 値が 1 つに決まるルールの衝突

単一値ルールも、同じ名前で複数定義できます。第 2 章で見たように、どれかの定義の条件が成り立てば、その定義の値になります。

ただし、複数の定義の条件が同時に成り立ち、値が異なるとエラーになります。

**コード 7.5** policy.rego

```
package quota

max_projects := 100 if {
    input.user.plan == "pro"
}

max_projects := 3 if {
    input.user.trial
}
```

**コード 7.6** input.json

```
{
  "user": {
    "plan": "pro",
    "trial": true
  }
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.quota'
1 error occurred: policy.rego:7: eval_conflict_error: rule data.quota.max_projects produced confli
  100 at policy.rego:3
  3 at policy.rego:7

Traceback:
  policy.rego:8: input.user.trial
  1:1: data.quota
```

この入力では、plan が "pro" で、trial も true です。2 つの定義の条件が両方とも成り立ち、100 と 3 という異なる値になるので、エラーになりました。同時に成り立つ定義の値が同じであれば、エラーにはなりません。

単一値ルールを複数の定義に分けるときは、条件が同時に成り立たないようにするか、成り立つ場合に値がそろうようにします。条件に優先順位を付けたいときは、「default と else」の章で説明する else を使います。

## 7.4　ルール名に参照を書く

ルール名の位置には、limits.cpu のような参照も書けます。

コード 7.7 policy.rego

```
package servers

limits.cpu := 2

limits.memory := "4Gi"

names_by_team[team.name] contains name if {
    some team in input.teams
    some name in team.servers
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
{
  "limits": {
    "cpu": 2,
    "memory": "4Gi"
  },
  "names_by_team": {
    "data": [
      "db-1"
    ],
    "frontend": [
      "web-1",
      "web-2"
    ]
  }
}
```

`limits.cpu := 2` と `limits.memory := "4Gi"` は、`limits` というオブジェクトの `cpu` と `memory` を、それぞれ別のルールとして定義しています。結果の `limits` は、両方のキーを持つオブジェクトになりました。

`names_by_team[team.name] contains name` は、オブジェクトを作るルールと集合を作るルールを組み合わせたものです。チーム名をキーにして、そのチームが担当するサーバー名の集合を値にしています。前の章の `names_by_env` では、キーごとに値をまとめる処理を、

内包表記を入れ子にして書きました。ルール名に参照を書くと、同じような処理を入れ子なしで書けます。

参照を使ったルール名でも、同じ場所に異なる値が作られるとエラーになります。

## 7.5　関数

**関数名（引数）:= 値 if { 本体 }** と書くと、引数を受け取る関数を定義できます。呼び出し方は組み込み関数と同じです。

**コード 7.8** policy.rego

```
package servers

is_prod(name) if {
    some server in input.servers
    server.name == name
    server.env == "prod"
}

env_of(name) := server.env if {
    some server in input.servers
    server.name == name
}

first_and_second([a, b, _]) := $"{a} and {b}"

web_1_is_prod := is_prod("web-1")

web_2_is_prod := is_prod("web-2")

db_env := env_of("db-1")

cache_env := env_of("cache-1")

first_two := first_and_second(["x", "y", "z"])
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
{
  "db_env": "prod",
  "first_two": "x and y",
  "web_1_is_prod": true
}
```

`is_prod(name)` は、`name` という名前のサーバーが本番環境にあるかを返す関数です。ルールヘッドに値を書いていないので、本体が成り立つと `true` を返します。`is_prod("web-1")` は `true` になりました。

`is_prod("web-2")` は、`web-2` が `"staging"` なので本体が成り立たず、関数の結果は undefined になります。そのため、`web_2_is_prod` も undefined で、結果に含まれていません。ルールと同じく、関数も本体が成り立たなければ `false` ではなく undefined になります。

`env_of(name)` は、サーバーの `env` を返す関数です。`env_of("cache-1")` は、該当するサーバーがないので undefined になりました。

`first_and_second([a, b, _])` のように、引数の位置に配列などの形を書くこともできます。引数が要素 3 つの配列のとき、1 番目と 2 番目の要素がそれぞれ `a` と `b` に当てはめられます。

関数も、同じ名前で複数定義できます。単一値ルールと同じく、同じ引数に対して複数の定義が異なる値を返すとエラーになります。また、引数の数が異なる同じ名前の関数は定義できません。

## 7.6 まとめ

- **ルール名** `contains` **値** `if { ... }` は集合を、**ルール名** `[キー] :=` **値** `if { ... }` はオブジェクトを作る複数値ルールです。集める値がなければ、空の集合やオブジェクトになります。
- 同じ名前のルールを複数定義すると、集合やオブジェクトはそれぞれの結果を合わせたものになります。
- 単一値ルールの複数の定義が同時に成り立ち、値が異なるとエラーになります。オブジェクトの同じキーに異なる値が作られた場合もエラーです。
- **関数名**（**引数**）`:=` **値** `if { ... }` で関数を定義できます。本体が成り立たなければ、結果は undefined です。

# 第 8 章

# 否定

これまでの条件は、「〜である」ことを調べるものでした。この章では、「〜でない」ことを調べる `not` を説明します。Rego の `not` は undefined の扱いに特徴があり、使い方を誤ると意図しない結果になります。

## 8.1 `not` の基本

式の前に `not` を付けると、その式が成り立たないときに成り立つ条件になります。

**コード 8.1** `policy.rego`

```
package access

suspended if {
    input.user.suspended
}

allow if {
    input.user.role == "admin"
    not suspended
}
```

`allow` は、管理者で、かつ利用停止（`suspended`）されていないときに `true` になります。3 つの入力で評価します。

**コード 8.2** suspended.json

```
{
  "user": {
    "role": "admin",
    "suspended": true
  }
}
```

**コード 8.3** active.json

```
{
  "user": {
    "role": "admin",
    "suspended": false
  }
}
```

**コード 8.4** missing.json

```
{
  "user": {
    "role": "admin"
  }
}
```

```
$ opa eval -d policy.rego -i suspended.json --format pretty 'data.access.allow'
undefined
$ opa eval -d policy.rego -i active.json --format pretty 'data.access.allow'
true
$ opa eval -d policy.rego -i missing.json --format pretty 'data.access.allow'
true
```

suspended.json では、input.user.suspended が true なので suspended も true になり、not suspended は成り立ちません。そのため allow は undefined です。

active.json では、input.user.suspended が false です。ルール本体の式は false だと成り立たないので、suspended は undefined になります。not suspended は成り立ち、

allow は true になりました。

missing.json には suspended のキーがありません。input.user.suspended は undefined なので、suspended も undefined になり、やはり allow は true になりました。

このように、not 式 は、式の値が false のときと undefined のときに成り立ちます。

## 8.2 not と != の違い

「管理者でない」という条件は、not と != のどちらでも書けそうに見えます。しかし、値が undefined のときの結果が違います。

**コード 8.5** policy.rego

```
package access

not_admin if {
    not input.user.role == "admin"
}

role_is_not_admin if {
    input.user.role != "admin"
}
```

**コード 8.6** input.json

```
{
  "user": {
    "name": "alice"
  }
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.access'
{
  "not_admin": true
}
```

入力に role がないので、input.user.role は undefined です。

- `not_admin`: `input.user.role == "admin"` は undefined なので、`not` を付けた式は成り立ちます。
- `role_is_not_admin`: `input.user.role != "admin"` は、比べる値が undefined なので式全体も undefined になり、成り立ちません。

値が存在しないときに条件を成り立たせたいなら `not`、値が存在して、かつ特定の値と異なることを確かめたいなら `!=` を使います。入力に必ずあるとは限らない値を調べるときは、どちらの意味が必要かを考えて選んでください。

## 8.3　集合に含まれないものを探す

`not` は、集合に値が含まれないことを調べるのにも使えます。次のポリシーは、どのチームも担当していないサーバーを探します。

**コード 8.7** `policy.rego`

```
package servers

owned contains name if {
    some team in input.teams
    some name in team.servers
}

unowned contains server.name if {
    some server in input.servers
    not owned[server.name]
}
```

**コード 8.8** input.json

```
{
  "servers": [
    {"name": "web-1"},
    {"name": "db-1"},
    {"name": "cache-1"}
  ],
  "teams": [
    {
      "name": "frontend",
      "servers": ["web-1"]
    },
    {
      "name": "data",
      "servers": ["db-1"]
    }
  ]
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
{
  "owned": [
    "db-1",
    "web-1"
  ],
  "unowned": [
    "cache-1"
  ]
}
```

owned は、いずれかのチームが担当しているサーバー名の集合です。unowned では、各サーバーについて not owned[server.name] を調べています。「変数と参照」の章で説明したとおり、owned[server.name] は名前が集合に含まれないと undefined になるので、not を付けると成り立ちます。

## 8.4　not の中の変数

not を付けた式の中で使う変数は、not より前で値が決まっている必要があります。

**コード 8.9** policy.rego

```
package servers

owned := {"web-1", "db-1"}

unowned contains name if {
    not owned[name]
}
```

```
$ opa eval -d policy.rego --format pretty 'data.servers'
1 error occurred: policy.rego:6: rego_unsafe_var_error: var name is unsafe
```

not owned[name] の name は、ほかのどこでも値が決まっていません。「owned に含まれない name」は無数にあり、すべてを挙げることはできないので、このような変数は安全でない（unsafe）としてエラーになります。前の節の unowned では、some server in input.servers で先に server の値を決めていたので、エラーになりませんでした。

## 8.5　関数の呼び出しを否定するときの注意

not を付けた式に関数の呼び出しや参照が含まれていると、その途中の値が undefined になったときに、not が成り立たないことがあります。

次の 2 つのポリシーは、import future.keywords.not の行以外は同じです。ここでの import は、言語の新しい機能をそのポリシーで有効にする宣言で、package の後に書きます。import については「モジュール」の章で説明します。

**コード 8.10** without_import.rego

```
package without_import

not_web if {
    not startswith(input.server.name, "web-")
}
```

**コード 8.11** with_import.rego

```
package with_import

import future.keywords.not

not_web if {
    not startswith(input.server.name, "web-")
}
```

**コード 8.12** input.json

```
{
  "server": {}
}
```

```
$ opa eval -d without_import.rego -i input.json --format pretty 'data.without_import.not_web'
undefined
$ opa eval -d with_import.rego -i input.json --format pretty 'data.with_import.not_web'
true
```

入力の server には name がないので、startswith(input.server.name, "web-") の引数が undefined になります。「web- で始まらない」ので not_web は true になってほしいところですが、インポートしていない without_import では undefined になりました。

これは、OPA が not を評価する前に、関数の引数などの途中の値を先に求めているためです。途中の値が undefined になると、not を評価する前にルール本体が成り立たなくなります。

import future.keywords.not を書くと、途中の値も含めた式全体が成り立たないときに、not が成り立つようになります。OPA は、not を使うポリシーにはこのインポートを書

くことを推奨しています。この本でも、これ以降、`not` に関数の呼び出しなどを含めるポリシーには、このインポートを書きます。

`future.keywords.not` は OPA 1.17.0 で追加されました。Conftest 0.71.1（組み込まれている OPA は 1.21.1）でも使えます。なお、OPA 1.22.0 では、このインポートと `and`・`or` キーワード（「キーワードの整理」の章で説明します）をまとめて有効にする `import rego.v2` が追加されましたが、Conftest 0.71.1 では使えません。

## 8.6　まとめ

- `not` **式** は、式の値が `false` か undefined のときに成り立ちます。
- 値が undefined のとき、`not x == "a"` は成り立ちますが、`x != "a"` は成り立ちません。
- `not` の中の変数は、`not` より前で値が決まっている必要があります。
- `not` に関数の呼び出しなどを含めるときは、`import future.keywords.not` を書きます。

# 第 9 章

# すべての要素に対する条件

「変数と参照」の章で見たように、ルール本体の反復は「条件を満たす要素が 1 つでもあれば成り立つ」という意味でした。この章では、反対の「すべての要素が条件を満たす」ことを調べる方法を説明します。このような「すべての〜について」という条件を、論理学の用語で全称量化と呼びます。OPA のドキュメントでは、Universal Quantification の節で説明されています。

Rego では、この条件を `every` で書くほか、否定や `count` を使っても書けます。それぞれの書き方と、要素や値がないときの違いを見ていきます。

## 9.1 every

`every` **変数** `in` **値** `{` **本体** `}` と書くと、値のすべての要素について本体が成り立つときに成り立つ条件になります。

**コード 9.1** `policy.rego`

```
package servers

all_tls if {
    every server in input.servers {
        server.tls
    }
}
```

**コード 9.2** ng.json

```
{
  "servers": [
    {"name": "web-1", "tls": true},
    {"name": "db-1", "tls": false}
  ]
}
```

```
$ opa eval -d policy.rego -i ok.json --format pretty 'data.servers.all_tls'
true
$ opa eval -d policy.rego -i ng.json --format pretty 'data.servers.all_tls'
undefined
```

all_tls は、すべてのサーバーの tls が true のときに true になります。ok.json ではどちらのサーバーも tls が true なので成り立ちました。ng.json では db-1 の tls が false なので、本体が成り立たない要素があり、all_tls は undefined になりました。

some ... in と同じく、変数を 2 つ書くと、配列では添字と要素、オブジェクトではキーと値を受け取れます。本体には複数の式を書けて、要素ごとにすべての式が成り立つ必要があります。次の all_labels_filled は、オブジェクトのキーと値を受け取り、2 つの式で調べています。

**コード 9.3** policy.rego

```
package servers

all_short_names if {
    every server in input.servers {
        count(server.name) <= 8
    }
}

all_labels_filled if {
    every key, value in input.labels {
        key != ""
        value != ""
    }
}
```

**コード 9.4** input.json

```
{
  "servers": [
    {"name": "web-1"},
    {"name": "database-1"}
  ],
  "labels": {
    "team": "platform",
    "env": "prod"
  }
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
{
  "all_labels_filled": true
}
```

all_short_names は、database-1 の名前が 8 文字を超えるので成り立ちません。all_labels_filled は、すべてのラベルのキーと値が空文字列でないので成り立ちまし

た。

every の本体で使った変数は、本体の中だけで使えます。

## 9.2　否定と count で書く

every を使わなくても、これまでに説明した機能でこの条件を書けます。

**コード 9.5** policy.rego

```
package servers

has_plain if {
    some server in input.servers
    not server.tls
}

all_tls_by_not if {
    not has_plain
}

all_tls_by_count if {
    plain := [server | some server in input.servers; not server.tls]
    count(plain) == 0
}
```

```
$ opa eval -d policy.rego -i ok.json --format pretty 'data.servers'
{
  "all_tls_by_count": true,
  "all_tls_by_not": true
}
$ opa eval -d policy.rego -i ng.json --format pretty 'data.servers'
{
  "has_plain": true
}
```

all_tls_by_not は、「tls でないサーバーが 1 つもない」と言い換えて書いています。

has_plain で tls でないサーバーがあるかを調べ、not has_plain でそれを否定しています。

all_tls_by_count は、tls でないサーバーを内包表記で集め、その数が 0 であることを調べています。

どちらも ok.json では成り立ち、ng.json では成り立ちません。every と同じ結果です。

## 9.3　要素や値がないとき

3 つの書き方は、空の配列ではどれも成り立ちますが、値そのものがないときに結果が異なります。

**コード 9.6** policy.rego

```
package servers

all_tls_by_every if {
    every server in input.servers {
        server.tls
    }
}

has_plain if {
    some server in input.servers
    not server.tls
}

all_tls_by_not if {
    not has_plain
}

all_tls_by_count if {
    plain := [server | some server in input.servers; not server.tls]
    count(plain) == 0
}
```

```
$ opa eval -d policy.rego -i empty.json --format pretty 'data.servers'
{
  "all_tls_by_count": true,
  "all_tls_by_every": true,
  "all_tls_by_not": true
}
$ opa eval -d policy.rego -i missing.json --format pretty 'data.servers'
{
  "all_tls_by_count": true,
  "all_tls_by_not": true
}
```

`empty.json` のように `servers` が空の配列のときは、3 つとも成り立ちました。調べる要素が 1 つもないので、条件に反する要素もないからです。

`missing.json` のように `servers` そのものが入力にないときは、結果が分かれました。

- `all_tls_by_every`: `input.servers` が undefined なので、`every` は成り立ちません。
- `all_tls_by_not`: `has_plain` の本体が成り立たず undefined になるので、`not has_plain` は成り立ちます。
- `all_tls_by_count`: 内包表記は空の配列になるので、`count(plain) == 0` は成り立ちます。

入力に値がないことを「条件を満たさない」として扱いたいなら `every`、「違反がない」として扱いたいなら否定や `count` の書き方が合います。ただし、空の配列の場合はどの書き方でも成り立ちます。空の配列を認めたくない場合は、`count(input.servers) > 0` のような条件を別に加えます。

## 9.4　まとめ

- `every x in` **値** `{` **本体** `}` は、すべての要素について本体が成り立つときに成り立ちます。
- 「条件に反する要素がない」と言い換えて、否定や `count` でも書けます。
- 空の配列には、どの書き方でも成り立ちます。値そのものが undefined のときは、`every` だけが成り立ちません。

# 第 10 章

# モジュール

これまでの例では、ポリシーを 1 つのファイルに書いてきました。ポリシーが大きくなると、複数のファイルに分けたくなります。この章では、Rego のファイルの構成要素であるコメント・パッケージ・インポートと、複数のファイルを組み合わせる方法を説明します。最後に、ポリシーが参照する `input` と `data` を整理します。

Rego では、1 つのファイルをモジュールと呼びます。モジュールは、1 つの `package`、0 個以上の `import`、0 個以上のルールでできています。

## 10.1 コメント

`#` から行末まではコメントです。行の途中から書くこともできます。

## 10.2 パッケージ

第 2 章で説明したとおり、`package` はルールが属する名前空間を決めます。パッケージ名は、`app.common` のようにドットで区切って階層にできます。パッケージ `app.common` のルール `is_admin` は、`data.app.common.is_admin` で参照します。

次の 3 つのファイルを使います。

**コード 10.1** common.rego

```
# アプリケーション全体で使う判定
package app.common

# 入力のユーザーが管理者なら true
is_admin if {
    input.user.role == "admin"
}
```

**コード 10.2** authz.rego

```
package app.authz

allow if {
    data.app.common.is_admin
}
```

**コード 10.3** authz_message.rego

```
package app.authz

message := "admin only"
```

authz.rego の allow は、別のパッケージのルールを data.app.common.is_admin と参照しています。同じパッケージのルールはルール名だけで参照できましたが、別のパッケージのルールは data から始まる参照で指定します。

authz.rego と authz_message.rego は、同じ package app.authz です。同じパッケージを宣言したファイルのルールは、1 つのパッケージにまとめられます。

opa eval の -d を複数書くと、それぞれのファイルが読み込まれます。

**コード 10.4** input.json

```
{
  "user": {
    "name": "alice",
    "role": "admin"
  }
}
```

```
$ opa eval -d common.rego -d authz.rego -d authz_message.rego -i input.json --format pretty 'data.
{
  "authz": {
    "allow": true,
    "message": "admin only"
  },
  "common": {
    "is_admin": true
  }
}
```

data.app を評価すると、パッケージの階層どおりに入れ子になったオブジェクトが返りました。data.app.authz には、2 つのファイルで定義した allow と message がまとめられています。

## 10.3 インポート

別のパッケージのルールを何度も参照する場合、毎回 data.app.common.is_admin と書くのは手間がかかります。import を使うと、参照を短い名前で使えるようになります。

**コード 10.5** authz.rego

```
package app.authz

import data.app.common
import data.app.common as shared
import data.roles
import input.user

allow if {
    common.is_admin
}

allow_by_alias if {
    shared.is_admin
}

name := user.name

permissions := roles[user.role]
```

import は package の後、ルールより前に書きます。

| 書き方 | 意味 |
|---|---|
| `import data.app.common` | common で data.app.common を参照できる |
| `import data.app.common as shared` | shared で data.app.common を参照できる |
| `import data.roles` | roles で data.roles を参照できる |
| `import input.user` | user で input.user を参照できる |

import した名前は、参照の最後の部分（data.app.common なら common）になります。as を使うと、別の名前を付けられます。同じパッケージのルールと名前がぶつかるときや、短い名前にしたいときに使います。

### 10.3.1　データを読み込む

`import data.roles` の `data.roles` は、ルールではなく、JSON ファイルから読み込んだデータです。「OPA・Rego・Conftest の関係」の章で説明したとおり、OPA は判断に入力（`input`）とデータ（`data`）を使います。`opa eval` の `-d` に JSON ファイルを指定すると、その内容がデータとして読み込まれます。

**コード 10.6** `roles.json`

```
{
  "roles": {
    "admin": ["read", "write"],
    "developer": ["read"]
  }
}
```

```
$ opa eval -d roles.json --format pretty 'data.roles'
{
  "admin": [
    "read",
    "write"
  ],
  "developer": [
    "read"
  ]
}
$ opa eval -d common.rego -d authz.rego -d roles.json -i input.json --format pretty 'data.app.auth
{
  "allow": true,
  "allow_by_alias": true,
  "name": "alice",
  "permissions": [
    "read",
    "write"
  ]
}
```

1 つ目のコマンドのように、roles.json の内容は data の直下に読み込まれ、data.roles で参照できます。ポリシーのルールも、読み込んだデータも、同じ data の下に置かれます。

2 つ目のコマンドでは、roles[user.role] が data.roles["admin"] になり、permissions の値は ["read", "write"] になりました。

ユーザーと役割の対応のように、判断のたびには変わらない情報はデータとして用意しておき、ポリシーから参照します。

### 10.3.2　キーワードのインポート

「否定」の章で使った import future.keywords.not は、データではなく、言語の機能を有効にするインポートです。future.keywords で始まるインポートは、将来の標準になる予定のキーワードや挙動を、そのモジュールで先に使えるようにします。

## 10.4　input と data

ここまでの章で、ポリシーは input と data の 2 つを参照してきました。この 2 つの違いを整理します。

| | input | data |
|---|---|---|
| 中身 | 判断のたびに渡される 1 つの値 | 読み込んだデータと、ルールの値をまとめた木 |
| 例 | リクエストの内容、検査するファイルの内容 | ユーザーと役割の対応、許可するレジストリの一覧 |
| opa eval での指定 | -i でファイルを 1 つ指定する | -d でポリシーやデータのファイルを指定する |

### 10.4.1　data は 1 つの木

data には、読み込んだデータと、ルールの値の両方が入ります。前の節の roles.json と、次のポリシーを使って、data 全体を評価してみます。

**コード 10.7** policy.rego

```
package app.authz

import data.roles

permissions := roles[input.user.role]

is_known_role if {
    roles[input.user.role]
}
```

**コード 10.8** input.json

```
{
  "user": {
    "name": "alice",
    "role": "admin"
  }
}
```

```
$ opa eval -d policy.rego -d roles.json --format pretty 'data'
{
  "app": {
    "authz": {}
  },
  "roles": {
    "admin": [
      "read",
      "write"
    ],
    "developer": [
      "read"
    ]
  }
}
$ opa eval -d policy.rego -d roles.json -i input.json --format pretty 'data'
```

```
{
  "app": {
    "authz": {
      "is_known_role": true,
      "permissions": [
        "read",
        "write"
      ]
    }
  },
  "roles": {
    "admin": [
      "read",
      "write"
    ],
    "developer": [
      "read"
    ]
  }
}
```

1 つ目のコマンドは入力を指定していません。`data` には、`roles.json` から読み込んだ `roles` と、パッケージ `app.authz` のオブジェクトが並んでいます。`app.authz` が空なのは、どちらのルールも入力を参照していて、入力がないと値が決まらないからです。

2 つ目のコマンドは入力を指定しています。`app.authz` に、ルールの値 `permissions` と `is_known_role` が入りました。このように、`data` は、ファイルから読み込んだデータと、ルールが計算した値を、パッケージの階層に沿って 1 つの木にまとめたものです。OPA のドキュメントでは、外部から OPA に渡されるデータを base document、ルールが計算する値を virtual document と呼んでいます。判断のたびに渡される `input` も、base document の一種として扱われています。

同じ木なので、読み込んだデータとルールが同じ場所を指すことはできません。

**コード 10.9** conflict.json

```
{
  "app": {
    "authz": {
      "permissions": ["read"]
    }
  }
}
```

```
$ opa eval -d policy.rego -d roles.json -d conflict.json --format pretty 'data'
1 error occurred: policy.rego:5: rego_compile_error: conflicting rule for data path app/authz/perm
```

conflict.json は、data.app.authz.permissions にデータを置こうとしています。ここにはルール permissions があるので、エラーになりました。キーが重ならなければ、データとルールは 1 つのオブジェクトにまとめられます。

### 10.4.2 input は 1 つの値

input は、-i で指定したファイルの値そのものです。オブジェクトである必要はなく、配列や文字列などでもかまいません。指定しなければ、input は undefined になります。

**コード 10.10** servers.json

```
["web-1", "db-1"]
```

```
$ opa eval --format pretty 'input'
undefined
$ opa eval -i servers.json --format pretty 'input[0]'
"web-1"
```

input に依存するルールが、入力がないときに undefined になるのはこのためです。

data はポリシーと一緒に用意しておく情報、input は判断のたびに変わる情報、と使い分けます。

## 10.5　まとめ

- 1 つのファイルをモジュールと呼び、package、import、ルールでできています。
- パッケージ名はドットで階層にでき、ルールは data. **パッケージ名**. **ルール名** で参照します。同じパッケージを宣言したファイルのルールはまとめられます。
- import で、data や input の参照を短い名前で使えます。as で名前を変えられます。
- opa eval -d に JSON ファイルを指定すると、データとして data の下に読み込まれます。
- data は、読み込んだデータとルールの値を 1 つにまとめた木です。input は判断のたびに渡される 1 つの値で、指定しなければ undefined です。

# 第 11 章

# キーワードの整理

ここまでの章で、if、contains、some、in、every、not などのキーワードを、必要になった場面で少しずつ使ってきました。この章では、まだ説明していない使い方として、in による値が含まれるかの判定、if の省略形、and と or を説明します。最後に、これまでのキーワードを一覧にまとめます。

## 11.1 in で値が含まれるかを調べる

some x in **値** の in は反復に使いました。some を付けずに **値** in **コレクション** と書くと、値が含まれるかを調べる式になります。

```
$ opa eval --format pretty '"admin" in ["dev", "admin"]'
true
$ opa eval --format pretty '"admin" in {"dev", "admin"}'
true
$ opa eval --format pretty '"ops" in {"dev", "admin"}'
false
$ opa eval --format pretty '"admin" in {"role": "admin"}'
true
$ opa eval --format pretty '"role" in {"role": "admin"}'
false
$ opa eval --format pretty '"role", "admin" in {"role": "admin"}'
true
```

配列と集合では、要素に含まれるかを調べます。オブジェクトでは、キーではなく値に含ま

れるかを調べます。"admin" は値なので true、"role" はキーなので false になりました。

最後のように **キー，値 in オブジェクト** と書くと、そのキーと値の組が含まれるかを調べられます。配列では **添字，要素 in 配列** の形になります。

in の式は、含まれないときに undefined ではなく false になります。3 つ目の "ops" は含まれないので false です。

### 11.1.1　ポリシーの中で使う

**コード 11.1** policy.rego

```
package access

import future.keywords.not

read_methods := ["GET", "HEAD"]

is_read if {
    input.method in read_methods
}

is_unknown_role if {
    not input.user.role in {"admin", "developer"}
}

is_read_value := input.method in read_methods
```

**コード 11.2** input.json

```
{
  "method": "POST",
  "user": {
    "role": "guest"
  }
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.access'
{
  "is_read_value": false,
  "is_unknown_role": true,
  "read_methods": [
    "GET",
    "HEAD"
  ]
}
```

is_read は、メソッドが read_methods に含まれるかを調べています。"POST" は含まれないので、is_read は undefined です。

is_unknown_role のように not と組み合わせると、含まれないことを調べられます。in は関数の呼び出しと同じように扱われるので、「否定」の章で説明した import future.keywords.not を書いています。このインポートがないと、入力に role がないときに not が成り立ちません。

is_read_value は、in の式の値をそのままルールの値にしています。含まれないので false になりました。ルール本体の条件として使う場合と違い、false という値が得られます。

「変数と参照」の章では、集合に値が含まれるかを **集合 [値]** で調べました。in は、配列やオブジェクトにも使えること、結果が true か false になることが違います。

## 11.2 if の省略形

ルール本体の式が 1 つだけのときは、波かっこを省略して if の後に直接書けます。

**コード 11.3** policy.rego

```
package servers

is_admin if input.user.role == "admin"

names contains server.name if some server in input.servers
```

コード 11.4 input.json

```
{
  "user": {
    "role": "admin"
  },
  "servers": [
    {"name": "web-1"},
    {"name": "db-1"}
  ]
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.servers'
{
  "is_admin": true,
  "names": [
    "db-1",
    "web-1"
  ]
}
```

`is_admin if input.user.role == "admin"` は、`is_admin if { input.user.role == "admin" }` と同じ意味です。`names contains server.name if some server in input.servers` のように、`some ... in` の 1 行も書けます。

式が 2 つ以上あるときは、これまでどおり波かっこで囲みます。

## 11.3　and と or

ルール本体の式は、改行で区切ると AND になりました。OR を表すには、同じ名前のルールを複数定義していました。`and` と `or` を使うと、1 つの式の中で AND と OR を組み合わせられます。

**コード 11.5** policy.rego

```
package docs

import future.keywords.and
import future.keywords.or

can_edit if {
    input.user.admin or input.user.owner
}

can_publish if {
    input.user.admin or input.user.owner and input.doc.reviewed
}

can_publish_grouped if {
    (input.user.admin or input.user.owner) and input.doc.reviewed
}
```

and と or を使うには、import future.keywords.and と import future.keywords.or を書きます。

**コード 11.6** admin.json

```
{
  "user": {
    "admin": true,
    "owner": false
  },
  "doc": {
    "reviewed": false
  }
}
```

```
$ opa eval -d policy.rego -i admin.json --format pretty 'data.docs'
{
  "can_edit": true,
```

```
  "can_publish": true
}
$ opa eval -d policy.rego -i owner.json --format pretty 'data.docs'
{
  "can_edit": true
}
```

`can_edit` は、管理者か所有者であれば成り立ちます。どちらの入力でも成り立ちました。

`can_publish` と `can_publish_grouped` は、かっこの有無だけが違います。`and` は `or` より先に結び付くので、`can_publish` は「管理者である」または「所有者で、かつレビュー済みである」という意味になります。`admin.json` は管理者なので成り立ち、`owner.json` はレビュー済みでないので成り立ちません。

`can_publish_grouped` はかっこで `or` を先にまとめているので、「管理者か所有者で、かつレビュー済みである」という意味です。どちらの入力もレビュー済みでないので、成り立ちません。

`and` と `or` の式は、成り立つか成り立たないかだけを表し、値を持ちません。そのため、変数に代入したり、ルールの値にしたりはできません。値を返すルールで条件を分けたいときは、これまでどおり同じ名前のルールを複数定義します。

`and` と `or` は OPA 1.20.0 で追加されたキーワードです。Conftest 0.71.1（組み込まれている OPA は 1.21.1）でも使えます。

## 11.4　キーワードの一覧

これまでに説明したキーワードをまとめます。

| キーワード | 使い方 | 説明した章 |
|---|---|---|
| `package` | `package app.authz` | はじめての Rego、モジュール |
| `import` | `import data.roles` | 否定、モジュール |
| `if` | **ルール** `if { 本体 }`、**ルール** `if` **式** | はじめての Rego、この章 |
| `contains` | **ルール** `contains` **値** `if { 本体 }` | ルール |

| キーワード | 使い方 | 説明した章 |
|---|---|---|
| some | some i、some x in **値** | 変数と参照 |
| in | some x in **値**、**値** in **コレクション** | 変数と参照、この章 |
| every | every x in **値** { **本体** } | すべての要素に対する条件 |
| not | not **式** | 否定 |
| and、or | **式** and **式**、**式** or **式** | この章 |

## 11.5 まとめ

- **値** in **コレクション** は、値が含まれるかを true か false で返します。オブジェクトでは値を調べ、**キー**, **値** in **オブジェクト** でキーと値の組を調べます。
- ルール本体の式が 1 つなら、**ルール** if **式** と波かっこを省略できます。
- and と or で、1 つの式の中で条件を組み合わせられます。and は or より先に結び付きます。

# 第 12 章

# with

ポリシーの結果は、入力とデータによって決まります。ポリシーを確かめるとき、入力のファイルを何通りも用意するのは手間がかかります。この章で説明する `with` を使うと、式を評価する間だけ、入力・データ・関数を別の値に差し替えられます。

`with` は、ポリシーのテスト（「OPA の CLI」の章で説明する `opa test`）でよく使います。

## 12.1 入力とデータを差し替える

**式 `with` 差し替える対象 `as` 値** と書くと、その式を評価する間だけ、対象を値に差し替えます。

**コード 12.1** `policy.rego`

```
package access

import data.roles

allow if {
    "write" in roles[input.user.role]
}
```

**コード 12.2** roles.json

```
{
  "roles": {
    "admin": ["read", "write"],
    "viewer": ["read"]
  }
}
```

```
$ opa eval -d policy.rego -d roles.json --format pretty 'data.access.allow with input as {
true
$ opa eval -d policy.rego -d roles.json --format pretty 'data.access.allow with input as {
undefined
$ opa eval -d policy.rego -d roles.json --format pretty 'data.access.allow with input as {
true
```

1 つ目と 2 つ目のコマンドでは、-i で入力のファイルを指定する代わりに、with input as {...} で入力を直接与えています。役割が "admin" なら allow は true、"viewer" なら undefined です。

3 つ目のコマンドのように、with は続けて複数書けます。with data.roles.viewer as [...] で、データの viewer の権限を差し替えたので、"viewer" でも allow が true になりました。

入力とデータでは、input 全体、input.user.role のような input の一部、data の下の値を差し替えられます。関数の差し替えは後の節で説明します。差し替えは with を付けた式の評価にだけ効き、ほかの式には影響しません。

## 12.2 ルールの中で使う

with は、ルール本体の式にも書けます。

**コード 12.3** policy.rego

```
package access

import data.roles

allow if {
    "write" in roles[input.user.role]
}

admin_can_write if {
    allow with input as {"user": {"role": "admin"}}
}

viewer_can_write if {
    allow with input.user.role as "viewer"
}
```

**コード 12.4** input.json

```
{
  "user": {
    "name": "bob",
    "role": "admin"
  }
}
```

```
$ opa eval -d policy.rego -d roles.json -i input.json --format pretty 'data.access'
{
  "admin_can_write": true,
  "allow": true
}
```

admin_can_write は、入力全体を差し替えて allow を評価しています。viewer_can_write は、input.user.role だけを "viewer" に差し替えています。入力の user は bob のままで、役割だけが変わります。viewer には write の権限がないので、viewer_can_write は undefined になりました。

allow そのものは、差し替えのない元の入力（役割は "admin"）で評価されて true です。with による差し替えが、その式の中だけに効いていることが分かります。

## 12.3 関数を差し替える

with では、組み込み関数やポリシーで定義した関数も差し替えられます。

**コード 12.5** policy.rego

```
package access

is_weekend if {
    time.weekday(time.now_ns()) in {"Saturday", "Sunday"}
}

display_name := upper(input.user.name)

bracket(s) := $"[{s}]"
```

```
$ opa eval -d policy.rego --format pretty 'data.access.is_weekend with time.weekday as "Su
true
$ opa eval -d policy.rego --format pretty 'data.access.is_weekend with time.weekday as "Mo
undefined
$ opa eval -d policy.rego --format pretty 'data.access.display_name with input.user.name a
"ALICE"
$ opa eval -d policy.rego --format pretty 'data.access.display_name with input.user.name a
"[alice]"
```

is_weekend は、現在の時刻（time.now_ns()）から曜日（time.weekday）を求めて、土曜か日曜かを調べるルールです。結果は評価した日によって変わるので、そのままでは確かめにくいルールです。

with time.weekday as "Sunday" と書くと、time.weekday を呼び出した結果が常に "Sunday" になります。このように、関数を値で差し替えると、関数はその値を返すようになります。

4 つ目のコマンドの with upper as data.access.bracket は、組み込み関数 upper を、ポリシーで定義した関数 bracket に差し替えています。display_name の

`upper(input.user.name)` は `bracket(input.user.name)` として評価され、`"[alice]"` になりました。関数を関数で差し替える場合は、引数の数がそろっている必要があります。

## 12.4 まとめ

- **式** `with` **対象** `as` **値** で、その式を評価する間だけ、入力・データ・関数を差し替えられます。
- `with` は続けて複数書けます。差し替えは `with` を付けた式の中だけに効きます。
- 関数を値で差し替えると、その値を返すようになります。関数で差し替えると、その関数が呼ばれます。

# 第 13 章

# default と else

Rego のルールは、条件が成り立たないと undefined になりました。判断の結果を受け取る側にとっては、allow が undefined よりも false のほうが扱いやすいことがあります。この章では、条件が成り立たないときの値を決める default と、条件を順番に試す else を説明します。

## 13.1 default

default **ルール名** := **値** と書くと、そのルールのどの定義の条件も成り立たないときの値（既定値）を決められます。

コード 13.1 policy.rego

```
package quota

default allow := false

allow if {
    input.user.role == "admin"
}

default max_projects := 3

max_projects := 100 if {
    input.user.plan == "pro"
}
```

コード 13.2 guest.json

```
{
  "user": {
    "role": "guest"
  }
}
```

```
$ opa eval -d policy.rego -i admin.json --format pretty 'data.quota'
{
  "allow": true,
  "max_projects": 100
}
$ opa eval -d policy.rego -i guest.json --format pretty 'data.quota'
{
  "allow": false,
  "max_projects": 3
}
```

admin.json では、allow と max_projects の条件が成り立つので、それぞれの定義の値 true と 100 になりました。

guest.json では、どちらの条件も成り立ちません。default がなければ 2 つのルールは undefined になり、結果に含まれませんでしたが、ここでは既定値の false と 3 になっています。

default は、値が 1 つに決まるルール(単一値ルール)に付けられます。contains で集合を作るルールには付けられません。集合を作るルールは、もともと条件を満たす値がなければ空の集合になるからです。

### 13.1.1 関数の既定値

関数にも既定値を付けられます。

**コード 13.3** policy.rego

```
package envs

default label(_) := "unknown"

label(env) := "production" if {
    env == "prod"
}

prod_label := label("prod")

dev_label := label("dev")
```

```
$ opa eval -d policy.rego --format pretty 'data.envs'
{
  "dev_label": "unknown",
  "prod_label": "production"
}
```

default label(_) := "unknown" は、どの定義の条件も成り立たないときに label が "unknown" を返すことを表します。引数の _ は、既定値ではどんな引数でも受け付けることを表しています。

### 13.1.2 既定値に使える値

既定値には、入力やほかのルールを参照する値は使えません。

**コード 13.4** policy.rego

```
package quota

default max_projects := input.default_max_projects
```

```
$ opa eval -d policy.rego --format pretty 'data.quota'
1 error occurred: policy.rego:3: rego_parse_error: illegal default rule (value cannot cont
        default max_projects := input.default_max_projects
        ^
```

既定値は、入力やデータによらず決まる定数でなければなりません。入力によって既定値を変えたい場合は、次の節の `else` を使うか、条件を分けたルールを書きます。

## 13.2 else

単一値ルールを複数定義したとき、条件が同時に成り立って値が異なるとエラーになりました。条件に優先順位を付けたいときは、`else` を使います。

**コード 13.5** policy.rego

```
package quota

plan := "pro" if {
    input.user.plan == "pro"
} else := "trial" if {
    input.user.trial
} else := "free"

is_pro if {
    input.user.plan == "pro"
} else := false
```

plan は、次の順に条件を試します。

1. input.user.plan が "pro" なら "pro"
2. そうでなく、input.user.trial が成り立つなら "trial"
3. どちらでもなければ "free"

**コード 13.6** pro.json

```
{
  "user": {
    "plan": "pro",
    "trial": true
  }
}
```

```
$ opa eval -d policy.rego -i pro.json --format pretty 'data.quota'
{
  "is_pro": true,
  "plan": "pro"
}
$ opa eval -d policy.rego -i trial.json --format pretty 'data.quota'
{
  "is_pro": false,
  "plan": "trial"
}
$ opa eval -d policy.rego -i none.json --format pretty 'data.quota'
{
  "is_pro": false,
  "plan": "free"
}
```

pro.json は、plan が "pro" で、trial も true です。2 つの条件が両方とも成り立ちますが、else は前の条件が成り立てば後ろを試さないので、エラーにならずに "pro" になりました。

trial.json は 1 つ目の条件が成り立たず、2 つ目が成り立つので "trial" です。none.json はどちらも成り立たないので、最後の else := "free" の値になりました。最後の else のように条件を書かなければ、それより前の条件がどれも成り立たないときに、必ずその値に

なります。

is_pro のように、真偽値のルールに else := false を付けると、条件が成り立たないときに false を返せます。default is_pro := false と書いた場合と同じ結果です。

else でつないだ定義は、全体で 1 つの定義として扱われます。同じ名前のルールを else の外で別に定義すると、これまでどおり、条件が同時に成り立って値が異なればエラーになります。

## 13.3 default と else の使い分け

| 書き方 | 向いている場面 |
|---|---|
| default | 条件の成り立たないときの値が定数で、条件どうしに優先順位がない |
| else | 条件に優先順位がある。上から順に試したい |

## 13.4 まとめ

- default ルール名 := 値 で、どの定義の条件も成り立たないときの値を決められます。既定値は定数でなければなりません。
- else でつなぐと、条件を上から順に試し、最初に成り立った条件の値になります。
- else を使うと、条件が同時に成り立っても値の衝突によるエラーになりません。

# 第 14 章

# 演算子

これまで、代入には :=、比較には == を使ってきました。Rego にはもう 1 つ、= という演算子があります。この章では、:=、==、= の違いと使い分けを説明します。in は「キーワードの整理」の章、比較演算子(< など)は「はじめての Rego」の章で説明しました。

## 14.1　代入 :=

:= は、左辺の変数に右辺の値を代入します。「変数と参照」の章で説明したとおり、同じ変数に 2 回代入するとエラーになります。

左辺に配列やオブジェクトの形を書くと、右辺の値を分解して、それぞれの変数に代入できます。これを分割代入と呼びます。

**コード 14.1** policy.rego

```
package profile

city := c if {
    [_, c, _] := split(input.address, ",")
}

name_and_role := [n, r] if {
    {"name": n, "role": r} := input.user
}

name_only := n if {
    {"name": n} := input.user
}
```

**コード 14.2** input.json

```
{
  "address": "1-2-3,Tokyo,Japan",
  "user": {
    "name": "alice",
    "role": "admin"
  }
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.profile'
{
  "city": "Tokyo",
  "name_and_role": [
    "alice",
    "admin"
  ]
}
```

split は、文字列を区切り文字で分けて配列にする組み込み関数です。split(input.address, ",") は ["1-2-3", "Tokyo", "Japan"] になり、[_, c, _] := ... で 2 番目の要素が c

に代入されました。配列の分割代入は、左辺と右辺の要素の数が同じときに成り立ちます。

オブジェクトの分割代入は、左辺と右辺のキーがすべて一致するときに成り立ちます。`name_and_role` は、`input.user` のキー `name` と `role` を両方書いているので成り立ちました。`name_only` は、`input.user` に `role` もあるのに左辺に `name` しか書いていないので成り立たず、結果に含まれていません。オブジェクトから一部の値を取り出すときは、分割代入ではなく `input.user.name` のような参照を使います。

## 14.2 比較 ==

== は、左辺と右辺の値が等しいかを調べます。== は値を比べるだけで、変数に値を代入しません。値の決まっていない変数を == に書くと、エラーになります。

**コード 14.3** `policy.rego`

```
package profile


is_tokyo if {
    city == "Tokyo"
}
```

```
$ opa eval -d policy.rego --format pretty 'data.profile'
1 error occurred: policy.rego:4: rego_unsafe_var_error: var city is unsafe
```

`city` はどこでも値が決まっていないので、「安全でない変数」としてエラーになりました。`:=` と間違えて == を書いたときなどに、このエラーで誤りに気づけます。

## 14.3 単一化 =

= は、左辺と右辺が等しくなるように、両辺の値の決まっていない変数に値を当てはめます。これを単一化と呼びます。変数の値がすでに決まっていれば比較として働き、決まっていなければ代入として働きます。

コード 14.4 policy.rego

```
package unify

pair := [x, y] if {
    [x, "world"] = ["hello", y]
}

ordered if {
    x > y
    y = 41
    x = 42
}

role := r if {
    input.user.role = r
}

is_admin if {
    input.user.role = "admin"
}
```

コード 14.5 input.json

```
{
  "user": {
    "role": "admin"
  }
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.unify'
{
  "is_admin": true,
  "ordered": true,
  "pair": [
    "hello",
    "world"
```

```
  ],
  "role": "admin"
}
```

pair の [x, "world"] = ["hello", y] では、両辺の配列が等しくなるように、左辺の x に "hello"、右辺の y に "world" が当てはめられました。:= は左辺の変数にしか代入できませんが、= は両辺の変数に値を当てはめられます。

ordered では、x > y を、x と y の値を決める = より前に書いています。= を使うと、OPA が式の順番を並べ替えて、値が決まってから x > y を評価します。

role の input.user.role = r は r に値を代入し、is_admin の input.user.role = "admin" は値を比較しています。同じ = が、場面によって代入にも比較にもなります。

## 14.4 使い分け

= は代入と比較のどちらにもなるので、読み手にはどちらの意味なのかが分かりにくくなります。また、変数の名前を書き間違えたとき、== ならエラーで気づけますが、= では新しい変数への代入として扱われ、気づけないことがあります。

| 演算子 | 使う場面 |
|---|---|
| := | 変数に値を代入する |
| == | 値を比較する |
| = | 両辺の変数に値を当てはめる必要があるときだけ |

この本でも、代入には :=、比較には == を使ってきました。= が必要になる場面は多くありません。

## 14.5 まとめ

- := は代入です。左辺に配列やオブジェクトの形を書くと分割代入になります。オブジェクトの分割代入はキーがすべて一致する必要があります。
- == は比較です。値の決まっていない変数を書くとエラーになります。
- = は単一化で、変数の値が決まっていれば比較、決まっていなければ代入として働きます。通常は := と == を使います。

# 第 15 章

# 組み込み関数

「スカラー値と文字列」の章で、組み込み関数の呼び出し方を説明しました。この章では、組み込み関数の名前空間、算術や集合の演算子、よく使う関数、実行時のエラーの扱いを説明します。

## 15.1　名前空間

組み込み関数には、`upper` や `count` のように名前だけのものと、`regex.match` や `time.now_ns` のようにドットで区切った名前のものがあります。ドットの前の部分は、関数の分類を表す名前空間です。例えば `regex` は正規表現、`time` は時刻、`units` は単位付きの値を扱う関数の名前空間です。

呼び出し方は、名前空間のない関数と同じです。

## 15.2　算術と集合の演算子

数値の計算と集合の演算には、演算子を使います。演算子も、内部では組み込み関数として扱われます。

```
$ opa eval --format pretty '7 / 2'
3.5
$ opa eval --format pretty '7 % 2'
1
$ opa eval --format pretty '{"a", "b"} | {"b", "c"}'
[
```

```
  "a",
  "b",
  "c"
]
$ opa eval --format pretty '{"a", "b"} & {"b", "c"}'
[
  "b"
]
$ opa eval --format pretty '{"a", "b"} - {"b", "c"}'
[
  "a"
]
```

数値の演算子は `+`（足し算）、`-`（引き算）、`*`（掛け算）、`/`（割り算）、`%`（割り算の余り）です。集合の演算子は次の 3 つです。

- **集合** `|` **集合**: 和集合（どちらかに含まれる要素）
- **集合** `&` **集合**: 積集合（両方に含まれる要素）
- **集合** `-` **集合**: 差集合（左にあって右にない要素）

`7 / 2` が `3.5` になるように、Rego の数値は整数と小数を区別しないので、割り算の結果は小数になることがあります。

## 15.3　よく使う関数

組み込み関数は約 200 個あります。ここでは、よく使うものを分類ごとに紹介します。

| 分類 | 関数 | 説明 |
|---|---|---|
| 集計 | `count(x)` | 要素の数 |
| | `sum(`**配列**`)`、`max(`**配列**`)`、`min(`**配列**`)` | 合計、最大値、最小値 |
| | `sort(`**配列**`)` | 並べ替えた配列 |
| 文字列 | `concat(`**区切り**`, `**配列**`)` | 配列の文字列を区切りでつなぐ |
| | `split(`**文字列**`, `**区切り**`)` | 文字列を区切りで分けた配列 |

| 分類 | 関数 | 説明 |
|---|---|---|
| | `sprintf(`**書式，配列**`)` | 書式に従って文字列を作る |
| | `startswith`、`endswith`、`indexof`、`replace`、`trim_space` | 文字列の判定・検索・置換・整形 |
| 正規表現 | `regex.match(`**パターン，文字列**`)` | 正規表現に一致するか |
| グロブ | `glob.match(`**パターン，区切り，文字列**`)` | * などのワイルドカードに一致するか |
| オブジェクト | `object.get(`**オブジェクト，キー，既定値**`)` | キーの値。キーがなければ既定値 |
| | `object.keys(`**オブジェクト**`)` | キーの集合 |
| 型 | `type_name(x)`、`is_string(x)` など | 値の型を調べる |
| 変換 | `to_number(x)` | 数値に変換する |
| | `json.unmarshal(`**文字列**`)`、`json.marshal(`**値**`)` | JSON の文字列と値を相互に変換する |
| 単位 | `units.parse_bytes(`**文字列**`)` | `"4Gi"` のような値をバイト数に変換する |
| 時刻 | `time.now_ns()`、`time.weekday(`**時刻**`)` | 現在時刻、曜日 |

いくつかを組み合わせた例です。

**コード 15.1** policy.rego

```
package containers

images_from_registry if {
    every c in input.containers {
        regex.match(`^registry\.example\.com/`, c.image)
    }
}

total_memory_bytes := sum([units.parse_bytes(c.memory) | some c in input.containers])

total_memory_gib := total_memory_bytes / units.parse_bytes("1Gi")

pull_policy(c) := object.get(c, "imagePullPolicy", "IfNotPresent")

pull_policies := {c.name: pull_policy(c) | some c in input.containers}

summary := sprintf("%d containers, %v GiB", [count(input.containers), total_memory_gib])
```

**コード 15.2** input.json

```
{
  "containers": [
    {
      "name": "app",
      "image": "registry.example.com/app:1.2.0",
      "memory": "512Mi",
      "imagePullPolicy": "Always"
    },
    {
      "name": "sidecar",
      "image": "registry.example.com/proxy:3.0.1",
      "memory": "1536Mi"
    }
  ]
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.containers'
{
  "images_from_registry": true,
  "pull_policies": {
    "app": "Always",
    "sidecar": "IfNotPresent"
  },
  "summary": "2 containers, 2 GiB",
  "total_memory_bytes": 2147483648,
  "total_memory_gib": 2
}
```

- `images_from_registry` は、`regex.match` で、すべてのイメージが `registry.example.com/` で始まるかを調べています。正規表現は、バックスラッシュをそのまま書けるロースト リングで書いています。
- `total_memory_bytes` は、`units.parse_bytes` で `"512Mi"` などをバイト数に変換し、`sum` で合計しています。`total_memory_gib` は、それを 1 GiB のバイト数で割っています。
- `pull_policy` は、`object.get` で `imagePullPolicy` を取り出し、キーがなければ `"IfNotPresent"` を返す関数です。`sidecar` にはキーがないので `"IfNotPresent"` になりました。
- `summary` は、`sprintf` で文字列を組み立てています。`%d` は整数、`%v` は任意の値、`%s` は文字列を埋め込む書式です。

すべての関数の一覧と詳しい説明は、OPA の公式ドキュメントの Policy Reference にあります。

## 15.4 sprintf と文字列補間

`sprintf` と文字列補間(`$"..."`)は、どちらも文字列に値を埋め込めます。違いは、埋め込む値が undefined のときです。

**コード 15.3** policy.rego

```
package messages

by_interpolation := $"role: {input.user.role}"

by_sprintf := sprintf("role: %s", [input.user.role])
```

**コード 15.4** input.json

```
{
  "user": {
    "name": "alice"
  }
}
```

```
$ opa eval -d policy.rego -i input.json --format raw 'data.messages.by_interpolation'
role: <undefined>
$ opa eval -d policy.rego -i input.json --format pretty 'data.messages.by_sprintf'
undefined
```

「スカラー値と文字列」の章で見たとおり、文字列補間は undefined の部分を <undefined> に置き換えて、文字列を作ります。一方、sprintf は引数の配列に undefined の値が含まれると、配列そのものが作れないので、結果全体が undefined になります。

値が欠けていてもメッセージを出したい場合は文字列補間、値がそろっているときだけメッセージを作りたい場合は sprintf が合います。

## 15.5　実行時のエラー

組み込み関数は、定数のように引数の型が評価の前に分かる場合、型が合わないと型検査でエラーになりました。一方、型は合っていても、値によっては評価の途中でエラーになることがあります。

**コード 15.5** policy.rego

```
package server

port := to_number(input.port)
```

**コード 15.6** input.json

```
{
  "port": "eighty"
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.server'
{}
$ opa eval -d policy.rego -i input.json --format pretty --strict-builtin-errors 'data.server'
1 error occurred: policy.rego:3: eval_builtin_error: to_number: strconv.ParseFloat: parsing "eight

Traceback:
  policy.rego:3: to_number("eighty")
  1:1: data.server
```

`to_number("eighty")` は、数値に変換できないのでエラーになります。しかし、1 つ目のコマンドではエラーは表示されず、`port` が undefined になっただけでした。OPA は既定では、組み込み関数の実行時のエラーを、その式が undefined になったものとして扱います。

2 つ目のコマンドのように `--strict-builtin-errors` を付けると、実行時のエラーで評価が止まり、エラーの内容が表示されます。ポリシーが期待どおりに動かないときは、このオプションを付けて評価してみると原因が分かることがあります。

## 15.6　まとめ

- 組み込み関数には、`regex.match` のように名前空間付きの名前のものがあります。
- 算術(`+ - * / %`)と集合(`|`、`&`、`-`)の演算子があります。
- `sprintf` は埋め込む値が undefined だと結果全体が undefined になり、文字列補間は `<undefined>` を埋め込みます。
- 組み込み関数の実行時のエラーは、既定では undefined として扱われます。`--strict-builtin-errors` でエラーとして扱えます。

# 第 16 章

# メタデータ

ポリシーが増えてくると、それぞれのルールが何を検査しているのか、誰が管理しているのかを記録しておきたくなります。Rego では、パッケージやルールに、決まった形式のコメントでメタデータ（アノテーション）を付けられます。この章では、アノテーションの書き方と、ポリシーやコマンドから読み出す方法を説明します。

## 16.1　アノテーションの書き方

アノテーションは、`# METADATA` で始まるコメントのブロックです。2 行目以降に YAML でアノテーションを書き、パッケージやルールの直前に置きます。

**コード 16.1** policy.rego

```
# METADATA
# title: コンテナイメージのポリシー
# description: デプロイするコンテナのイメージを検査する。
# authors:
# - Platform Team <platform@example.com>
package containers

import future.keywords.not

# METADATA
# title: イメージは社内のレジストリから取得する
# description: 公開レジストリのイメージは使わない。
# custom:
#   severity: high
#   ticket: SEC-123
untrusted_images contains c.image if {
    some c in input.containers
    not startswith(c.image, "registry.example.com/")
}

# METADATA
# title: レポート
report := {
    "title": rego.metadata.rule().title,
    "package_title": rego.metadata.chain()[1].annotations.title,
}

# METADATA
# custom:
#   severity: high
severity := rego.metadata.rule().custom.severity
```

このファイルでは、パッケージと 3 つのルールにアノテーションを付けています。主なアノテーションは次のとおりです。

| アノテーション | 内容 |
|---|---|
| `title` | 名前 |
| `description` | 説明 |
| `authors` | 作成者 |
| `related_resources` | 関連する資料の URL など |
| `custom` | 任意のキーと値(重要度やチケット番号など) |
| `scope` | アノテーションを適用する範囲 |
| `schemas` | 入力やデータのスキーマ(次の章で説明します) |

`custom` には、決まった形式のない値を自由に書けます。この例では、重要度(`severity`)とチケット番号(`ticket`)を書いています。

## 16.2　スコープ

アノテーションがどこに適用されるかを、スコープと呼びます。`scope` を書かなければ、置いた場所によって決まります。

| スコープ | 適用される範囲 | 書かなかったときにこのスコープになる場所 |
|---|---|---|
| `package` | そのパッケージ | `package` の直前 |
| `rule` | 直後のルールの 1 つの定義 | ルールの直前 |
| `document` | 同じ名前のルールのすべての定義 | (なし) |
| `subpackages` | そのパッケージと、その下の階層のパッケージ | (なし) |

同じ名前のルールを複数の定義に分けて書いている場合、ルールの直前のアノテーションはその 1 つの定義にだけ適用されます。すべての定義に適用したいときは、どれか 1 つの定義の前に `scope: document` を書きます。

## 16.3　アノテーションを読み出す

アノテーションは、ポリシーの中から組み込み関数で読み出せます。

**コード 16.2** input.json

```
{
  "containers": [
    {"image": "registry.example.com/app:1.2.0"},
    {"image": "docker.io/library/nginx:latest"}
  ]
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.containers'
{
  "report": {
    "package_title": "コンテナイメージのポリシー",
    "title": "レポート"
  },
  "severity": "high",
  "untrusted_images": [
    "docker.io/library/nginx:latest"
  ]
}
$ opa inspect -a policy.rego
NAMESPACES:
┌──────────────────┬──────────────┐
│    NAMESPACE     │     FILE     │
├──────────────────┼──────────────┤
│ data.containers  │ policy.rego  │
└──────────────────┴──────────────┘
ANNOTATIONS:
コンテナイメージのポリシー
=====================================

デプロイするコンテナのイメージを検査する。


Package:  containers
Location: policy.rego:6
Scope: package
```

```
Authors:
 Platform Team <platform@example.com>

レポート
============

Package:  containers
Rule:     report
Location: policy.rego:23
Scope: rule

containers.severity
===================

Package:  containers
Rule:     severity
Location: policy.rego:31
Scope: rule

Custom:
 severity: "high"

イメージは社内のレジストリから取得する
============================================================

公開レジストリのイメージは使わない。

Package:  containers
Rule:     untrusted_images
Location: policy.rego:16
Scope: rule

Custom:
 severity: "high"
 ticket:   "SEC-123"
```

1 つ目のコマンドの結果を見ると、次のことが分かります。

- `rego.metadata.rule()` は、そのルール自身に付けたアノテーションを返します。`report` の `title` は `"レポート"`、`severity` は `custom` に書いた `"high"` になりました。
- `rego.metadata.chain()` は、そのルールから外側に向かって、アノテーションを順に並べた配列を返します。`report` では、0 番目がルール自身、1 番目がパッケージのアノテーションです。`package_title` には、パッケージの `title` が入りました。

`rego.metadata.rule()` で読めるのは、そのルールに付けたアノテーションだけです。アノテーションを付けていないルールで `rego.metadata.rule().title` のように書くと、型検査でエラーになります。

2 つ目のコマンドの `opa inspect -a` は、ポリシーのファイルに書かれたアノテーションを一覧にして表示します。パッケージやルールごとに、タイトル、説明、場所、スコープ、`custom` の内容が表示されています。ポリシーの一覧表を作るときなどに使えます。

## 16.4　まとめ

- `# METADATA` で始まるコメントのブロックに YAML でアノテーションを書き、パッケージやルールの直前に置きます。
- アノテーションの適用範囲（スコープ）は、置いた場所か `scope` で決まります。
- `rego.metadata.rule()` と `rego.metadata.chain()` でポリシーの中から読み出せます。`opa inspect -a` で一覧を表示できます。

# 第 17 章

# スキーマと strict モード

Rego では、存在しない値を参照しても、エラーにならずに undefined になるだけです。そのため、入力のフィールド名を打ち間違えても、ルールが黙って成り立たなくなるだけで、誤りに気づきにくいことがあります。この章では、入力の形を JSON Schema で OPA に教えて誤りを検出する方法と、使われていないコードを指摘する strict モードを説明します。

## 17.1　スキーマで入力の形を教える

次のポリシーは、`role` を `rol` と打ち間違えています。

**コード 17.1** `policy.rego`

```
package access


allow if {
    input.user.rol == "admin"
}
```

**コード 17.2** input.json

```
{
  "user": {
    "name": "alice",
    "role": "admin"
  }
}
```

入力の形を表す JSON Schema を用意します。JSON Schema は、JSON の値の形(オブジェクトのキーや値の型)を JSON で記述する形式です。

**コード 17.3** input.schema.json

```
{
  "type": "object",
  "properties": {
    "user": {
      "type": "object",
      "properties": {
        "name": {"type": "string"},
        "role": {"type": "string"}
      },
      "required": ["name", "role"]
    }
  }
}
```

このスキーマは、入力が user というオブジェクトを持ち、その中に文字列の name と role があることを表しています。

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.access'
{}
$ opa eval -d policy.rego -i input.json -s input.schema.json --format pretty 'data.access'
1 error occurred: policy.rego:4: rego_type_error: undefined ref: input.user.rol
        input.user.rol
                  ^
                  have: "rol"
```

```
                    want (one of): ["name" "role"]
```

1 つ目のコマンドでは、input.user.rol が undefined になり、allow は結果に含まれません。エラーは何も表示されないので、打ち間違いに気づけません。

2 つ目のコマンドでは、-s でスキーマのファイルを指定しています。OPA は、スキーマから input.user のキーが name と role だと分かるので、rol を参照している誤りを評価の前に型エラーとして報告しました。

スキーマは、値の型の誤りも検出します。

**コード 17.4** policy.rego

```
package access

allow if {
    input.user.role == 1
}
```

```
$ opa eval -d policy.rego -i input.json -s input.schema.json --format pretty 'data.access'
1 error occurred: policy.rego:4: rego_type_error: match error
        left  : string
        right : number
```

スキーマでは role は文字列なので、数値の 1 と比べても成り立つことはありません。このような比較も、型エラー（match error）として報告されます。

## 17.2　アノテーションでスキーマを指定する

前の章で紹介したアノテーションの schemas を使うと、ルールごとに使うスキーマを指定できます。

**コード 17.5** policy.rego

```
package access


# METADATA
# schemas:
# - input: schema.user_request
allow if {
    input.user.rol == "admin"
}
```

-s にディレクトリを指定すると、その中の JSON ファイルが schema.**ファイル名** という名前で参照できるようになります。ここでは schemas/user_request.json を schema.user_request として、allow の入力のスキーマに指定しています。

**コード 17.6** user_request.json

```
{
  "type": "object",
  "properties": {
    "user": {
      "type": "object",
      "properties": {
        "name": {"type": "string"},
        "role": {"type": "string"}
      },
      "required": ["name", "role"]
    }
  }
}
```

```
$ opa eval -d policy.rego -i input.json -s schemas --format pretty 'data.access'
1 error occurred: policy.rego:7: rego_type_error: undefined ref: input.user.rol
        input.user.rol
                  ^
                  have: "rol"
                  want (one of): ["name" "role"]
```

入力の形がルールによって違う場合は、このようにアノテーションでスキーマを指定します。

## 17.3 strict モード

`opa check` は、ポリシーの構文や型の誤りを調べるコマンドです。`--strict` を付けると、誤りではないものの、不要なコードも指摘します。

**コード 17.7** `policy.rego`

```
package access

import data.roles

allow if {
    name := input.user.name
    input.user.role == "admin"
}
```

```
$ opa check policy.rego
$ opa check --strict policy.rego
2 errors occurred:
policy.rego:3: rego_compile_error: import data.roles unused
policy.rego:6: rego_compile_error: assigned var name unused
```

`--strict` なしの `opa check` は何も報告しませんでした。`--strict` を付けると、使われていない `import data.roles` と、代入したのに使われていない変数 `name` が報告されました。

使われていない変数は、書き忘れや打ち間違いのしるしであることがあります。例えば、`name` を使った条件を書くつもりで忘れていたのかもしれません。strict モードで、このような見落としに気づけます。

## 17.4 まとめ

- 存在しない値の参照は undefined になるだけなので、打ち間違いに気づきにくいことがあります。
- `-s` で JSON Schema を指定すると、入力の参照や型の誤りを評価の前に検出でき

ます。アノテーションの `schemas` で、ルールごとにスキーマを指定することもできます。

- `opa check --strict` は、使われていない `import` や変数などを指摘します。

# 第 III 部

# OPA と Conftest

# 第 18 章

# OPA と Conftest の概要

第 III 部では、Rego を評価するツールである OPA と Conftest を使って、実際のファイルを検査します。この章では、2 つのツールの概要を説明します。

## 18.1 OPA

OPA（Open Policy Agent）は、Rego で書いたポリシーを評価するエンジンです。ポリシーの判断をアプリケーションから切り離すために作られていて、使い方に合わせて次のような形で動かせます。

| 使い方 | 内容 |
|---|---|
| CLI | `opa eval` で評価し、`opa test` でテストし、`opa fmt` で整形する |
| サーバー | `opa run --server` で起動し、アプリケーションが REST API で判断を問い合わせる |
| ライブラリ | Go のアプリケーションに組み込んで評価する |

どの使い方でも、Rego の言語は同じです。違うのは、入力をどこから受け取り、どのルールの値を問い合わせ、結果をどう使うかです。

この本では、ポリシーを書くときに使う CLI を扱います。第 II 部の例を動かすのに使ってきた `opa eval` に加えて、テストや整形のコマンドを「OPA の CLI」の章で説明します。

## 18.2　Conftest

Conftest は、設定ファイルを Rego のポリシーで検査するためのコマンドラインツールです。OPA を Go のライブラリとして組み込んでいて、次のことを担当します。

- YAML、JSON、HCL、Dockerfile など、さまざまな形式の設定ファイルを読み込んで、Rego の入力に変換する
- `deny` や `warn` という名前のルールを評価して、検査の失敗や警告として表示する
- 失敗があれば 0 以外の終了コードで終わる

最後の点により、CI に組み込むと、決まりを守っていない設定の変更を、マージする前に見つけられます。

Conftest のポリシーも Rego で書くので、第 II 部で説明した言語の知識はそのまま使えます。Conftest に固有なのは、どのルールを評価し、その値をどう扱うかという部分です。

# 第 19 章

# OPA の CLI

第 II 部では、Rego の例を動かすために `opa eval` を使ってきました。この章では、OPA の CLI のうち、ポリシーを書くときによく使う `opa eval`、`opa fmt`、`opa check`、`opa test` をまとめます。

## 19.1 opa eval

`opa eval` は、ポリシーとデータを読み込んで、クエリを評価するコマンドです。これまでに使ったオプションをまとめます。

| オプション | 説明 |
|---|---|
| `-d` **ファイル** | ポリシー(`.rego`)やデータ(`.json`、`.yaml`)を読み込む。ディレクトリも指定できる |
| `-i` **ファイル** | 入力を読み込む |
| `-s` **ファイル** | 入力のスキーマを読み込む |
| `--format` **形式** | 結果の表示形式 |
| `--strict-builtin-errors` | 組み込み関数の実行時のエラーで評価を止める |

### 19.1.1 結果の表示形式

この本では、主に `--format pretty` を使ってきました。ほかの形式も使えます。

```
$ opa eval --format json 'some x in ["a", "b"]'
{
  "result": [
    {
      "expressions": [
        {
          "value": true,
          "text": "some x in [\"a\", \"b\"]",
          "location": {
            "row": 1,
            "col": 1
          }
        }
      ],
      "bindings": {
        "x": "a"
      }
    },
    {
      "expressions": [
        {
          "value": true,
          "text": "some x in [\"a\", \"b\"]",
          "location": {
            "row": 1,
            "col": 1
          }
        }
      ],
      "bindings": {
        "x": "b"
      }
    }
  ]
}
```

```
$ opa eval --format bindings 'some x in ["a", "b"]'
{
  "x": "a"
}
{
  "x": "b"
}
$ opa eval --format raw '["a", "b"][_]'
a
b
```

| 形式 | 内容 |
|---|---|
| json（指定しないときの形式） | 結果をすべて含む JSON。式の値（value）と、変数の値（bindings）が、結果ごとに含まれる |
| pretty | 人が読みやすい形。結果が 1 つで変数がなければ値の JSON、それ以外は表 |
| bindings | 変数の値だけを、結果ごとに JSON で表示する |
| values | 式の値だけを、結果ごとに JSON の配列で表示する |
| raw | 式の値だけを、結果ごとに 1 行で表示する。文字列は引用符なしで表示する |

raw は、シェルスクリプトで結果を使うときに便利です。

### 19.1.2 ディレクトリからデータを読み込む

-d にディレクトリを指定すると、その中のポリシーとデータのファイルがすべて読み込まれます。

**コード 19.1** data.json

```
{
  "admin": ["read", "write"],
  "viewer": ["read"]
}
```

**コード 19.2** data.yaml

```
team: platform
```

**コード 19.3** policy.rego

```
package access

import data.roles

can_write(role) if {
    "write" in roles[role]
}
```

```
$ opa eval -d bundle --format pretty 'data.roles'
{
  "admin": [
    "read",
    "write"
  ],
  "viewer": [
    "read"
  ]
}
$ opa eval -d bundle --format pretty 'data.team'
"platform"
$ opa eval -d bundle --format pretty 'data.access.can_write("admin")'
true
```

データのファイルは、置かれたディレクトリの名前が `data` の下のパスになります。`bundle/roles/data.json` の内容は `data.roles` に、`bundle` の直下にある `data.yaml` の内

容は data の直下に読み込まれました。YAML のデータも JSON と同じように読み込まれます。

## 19.2 opa fmt と opa check

opa fmt はポリシーを決まった形に整形し、opa check は構文や型の誤りを調べます。

**コード 19.4** messy.rego

```
package messy
allow if {input.a==1
  input.b == 2}
```

**コード 19.5** broken.rego

```
package broken

allow if {
    input.a == 1
```

```
$ opa fmt --list messy.rego
messy.rego
$ opa fmt messy.rego
package messy

allow if {
        input.a == 1
        input.b == 2
}
$ opa check broken.rego
1 error occurred during loading: broken.rego:5: rego_parse_error: unexpected eof token
        input.a == 1
                   ^
```

- opa fmt --list は、整形が必要なファイルの名前を表示します。
- opa fmt はファイル名だけを指定すると、整形した結果を表示します。-w を付けると、ファイルを書き換えます。

- `opa check` は、`broken.rego` の閉じかっこがないことを構文エラーとして報告しました。「スキーマと strict モード」の章で説明した `--strict` を付けると、不要なコードも指摘します。

この本のサンプルも、すべて `opa fmt` で整形しています。

## 19.3　opa test

`opa test` は、ポリシーのテストを実行するコマンドです。テストも Rego で書きます。

**コード 19.6** `policy.rego`

```
package access

import data.roles

default allow := false

allow if {
    "write" in roles[input.user.role]
}
```

**コード 19.7** `policy_test.rego`

```
package access_test

import data.access

test_admin_can_write if {
    access.allow with input as {"user": {"role": "admin"}}
        with data.roles as {"admin": ["read", "write"]}
}

test_viewer_cannot_write if {
    not access.allow with input as {"user": {"role": "viewer"}}
        with data.roles as {"viewer": ["read"]}
}
```

名前が test_ で始まるルールがテストです。テストのルールが成り立てば成功、成り立たなければ失敗です。テストは、テスト対象とは別のパッケージに書き、import でテスト対象を参照するのが一般的です。ファイル名は _test.rego で終わる名前にする慣習があります。

テストでは、「with」の章で説明した with で入力とデータを差し替えます。入力ごとにファイルを用意しなくても、さまざまな場合を確かめられます。with は長くなるので、2 行に分けて書いています。

```
$ opa test policy.rego policy_test.rego
PASS: 2/2
$ opa test -v policy.rego policy_test.rego
policy_test.rego:10:
data.access_test.test_viewer_cannot_write: PASS (227.417µs)
policy_test.rego:5:
data.access_test.test_admin_can_write: PASS (225.845µs)
--------------------------------------------------------------------------------
PASS: 2/2
```

-v を付けると、テストごとの結果が表示されます。

テストが失敗すると、-v では失敗の理由も表示されます。次のテストは、viewer が書き込めると誤って期待しています。

**コード 19.8** policy_test.rego

```
package access_test

import data.access

test_viewer_can_write if {
    access.allow with input as {"user": {"role": "viewer"}}
        with data.roles as {"viewer": ["read"]}
}
```

```
$ opa test -v policy.rego policy_test.rego
policy_test.rego:5:
data.access_test.test_viewer_can_write: FAIL (227.587µs)
--------------------------------------------------------------------------------
```

```
FAILURES
--------------------------------------------------------------------------------
data.access_test.test_viewer_can_write: FAIL (227.587µs)

  query:1                  Enter data.access_test.test_viewer_can_write = _
  policy_test.rego:5       | Enter data.access_test.test_viewer_can_write
  policy.rego:7            | | Enter data.access.allow
  policy.rego:8            | | | Fail internal.member_2("write", __local1__)
  policy_test.rego:6       | | Fail data.access.allow with input as {"user": {"role": "vie
  query:1                  | Fail data.access_test.test_viewer_can_write = _

--------------------------------------------------------------------------------
FAIL: 1/1
```

FAILURES の下に、評価の経過が表示されています。policy.rego:8 の internal.member_2("write", ...) で失敗しています。internal.member_2 は、"write" in roles[...] の in を表す内部の関数で、viewer の権限に "write" が含まれないために成り立たなかったことが分かります。

opa test は、失敗したテストがあると、0 以外の終了コードで終わります。CI でテストを実行すれば、ポリシーの変更で既存の判断が変わったことに気づけます。

## 19.4　print でデバッグする

ポリシーが期待どおりに動かないときは、ルール本体に print を書くと、評価の途中の値を表示できます。

**コード 19.9** policy.rego

```
package access

allow if {
    print("role:", input.user.role)
    input.user.role == "admin"
}
```

**コード 19.10** input.json

```
{
  "user": {
    "role": "viewer"
  }
}
```

```
$ opa eval -d policy.rego -i input.json --format pretty 'data.access.allow'
role: viewer
undefined
```

print の出力は、評価の結果とは別に表示されます。print はルール本体の条件としては常に成り立つので、ほかの式の結果には影響しません。opa test では、失敗したテストの print の出力が表示されます。-v を付けると、成功したテストの出力も表示されます。

## 19.5　まとめ

- opa eval は --format で結果の表示形式を選べます。-d にディレクトリを指定すると、ディレクトリの名前がデータのパスになります。
- opa fmt で整形し、opa check で構文や型の誤りを調べます。
- opa test は test_ で始まるルールをテストとして実行します。with で入力とデータを差し替えて書きます。
- print で評価の途中の値を表示できます。

# 第 20 章

# Conftest は何を評価しているか

Conftest のポリシーは、`deny contains msg if { ... }` のように書くのが定番です。しかし、`deny` という名前は Rego の言語にとって特別なものではありません。どのルールを評価し、その値をどう「失敗」や「警告」に変えるかは、Conftest が決めています。

この章では、Rego 側で何が評価されているかと、Conftest がその結果のどこを見ているかを分けて説明します。ここで説明する Conftest の動作は、Conftest 0.71.1 のソースコードと、実際の実行結果で確かめたものです。

「OPA と Conftest の概要」の章で説明したとおり、Conftest は OPA を Go のライブラリとして組み込み、ファイルを入力に変換すること、決まった名前のルールを評価して成否を決めること、結果を表示して終了コードを返すことを担当しています。この章では、2 つ目のルールの評価と成否の決め方を詳しく見ます。

## 20.1 全体の流れ

`conftest test` **ファイル** を実行すると、次の順に処理されます。

1. ファイルを、形式（YAML、JSON、HCL、Dockerfile など）に合ったパーサーで JSON の値に変換し、`input` にします。
2. ポリシーのディレクトリ（指定しなければ `policy`）にある `.rego` ファイルを読み込みます。
3. 対象の名前空間（指定しなければ `main`）のパッケージから、名前が `deny`・`violation`・`warn`、またはそれに `_`**名前** を続けたルールを探します。
4. 見つけたルールごとに、`data.` **名前空間** `.` **ルール名** を評価します。

5. 評価した値から、失敗・警告・成功を決めて表示します。

例で確かめます。

**コード 20.1** deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 1
```

**コード 20.2** main.rego

```
package main

import future.keywords.not

deny contains msg if {
    input.kind == "Deployment"
    input.spec.replicas < 2
    msg := $"{input.metadata.name}: replicas must be at least 2"
}

warn contains "metadata.labels is not set" if {
    not input.metadata.labels
}
```

```
$ conftest parse deployment.yaml
{
  "apiVersion": "apps/v1",
  "kind": "Deployment",
  "metadata": {
    "name": "web"
  },
  "spec": {
    "replicas": 1
```

```
  }
}
$ opa eval -d policy -i <(conftest parse deployment.yaml) --format pretty 'data.main'
{
  "deny": [
    "web: replicas must be at least 2"
  ],
  "warn": [
    "metadata.labels is not set"
  ]
}
$ conftest test deployment.yaml
WARN - deployment.yaml - main - metadata.labels is not set
FAIL - deployment.yaml - main - web: replicas must be at least 2

2 tests, 0 passed, 1 warning, 1 failure, 0 exceptions
```

1 つ目の `conftest parse` は、ファイルを `input` に変換した結果を表示するコマンドです。YAML が、同じ構造の JSON の値になっています。

2 つ目では、その `input` を使って、同じポリシーを `opa eval` で評価しています(`<(...)` は、コマンドの出力をファイルのように渡すシェルの書き方です)。`data.main` の `deny` と `warn` は、どちらも文字列の集合になりました。

3 つ目の `conftest test` は、`deny` の要素を `FAIL`、`warn` の要素を `WARN` として表示しています。2 つ目の結果と見比べると、Conftest が表示しているのは、ルールの値(集合)の要素そのものだと分かります。

## 20.2 Rego 側で評価しているもの

Rego 側から見ると、`deny contains msg if { ... }` は、「ルール」の章で説明した集合を作るルール(複数値ルール)です。条件を満たす `msg` を集めた集合が、`deny` の値になります。条件を満たすものがなければ、`deny` は空の集合です。

`deny` や `warn` という名前に、Rego の言語としての特別な意味はありません。上の例のように、`opa eval` で評価すれば、ただの集合として値が返ります。Rego 側でしていることは、これまでの章と同じく、「入力から値を計算する」ことだけです。

## 20.3　Conftest が値のどこを見るか

Conftest は、評価したルールの値の種類によって扱いを変えます。いろいろな値を返すルールで確かめます。

**コード 20.3** config.yaml

```yaml
name: web
```

**コード 20.4** main.rego

```
package main

deny_bool if {
    input.name == "web"
}

deny_string := "name is web" if {
    input.name == "web"
}

deny_object contains {"msg": "name is web", "severity": "high"} if {
    input.name == "web"
}

deny_empty contains "" if {
    input.name == "web"
}

deny_number contains 42 if {
    input.name == "web"
}

deny_nothing contains "name is db" if {
    input.name == "db"
}
```

```
$ opa eval -d policy -i <(conftest parse config.yaml) --format pretty 'data.main'
{
  "deny_bool": true,
  "deny_empty": [
    ""
  ],
  "deny_nothing": [],
  "deny_number": [
    42
  ],
  "deny_object": [
    {
      "msg": "name is web",
      "severity": "high"
    }
  ],
  "deny_string": "name is web"
}
$ conftest test config.yaml
FAIL - config.yaml - main - name is web

6 tests, 5 passed, 0 warnings, 1 failure, 0 exceptions
$ conftest test -o json config.yaml
[
        {
                "filename": "config.yaml",
                "namespace": "main",
                "successes": 5,
                "failures": [
                        {
                                "msg": "name is web",
                                "metadata": {
                                        "query": "data.main.deny_object",
                                        "severity": "high"
                                }
```

```
                        }
                ]
        }
]
```

6 つのルールはどれも名前が deny_ で始まり、deny_nothing 以外は条件が成り立っています。しかし、失敗になったのは deny_object だけでした。

| ルール | Rego で評価した値 | Conftest の扱い |
|---|---|---|
| deny_bool | true | 成功 |
| deny_string | "name is web" | 成功 |
| deny_object | msg を持つオブジェクトの集合 | 失敗（msg がメッセージ） |
| deny_empty | 空文字列の集合 | 成功 |
| deny_number | 数値の集合 | 成功 |
| deny_nothing | 空の集合 | 成功 |

Conftest の扱いは、次の規則で決まっています。

- **値が配列でなければ、何も見ない。**Rego の集合は JSON では配列になるので、contains のルールは配列として扱われます。deny_bool の true や、deny_string の文字列は配列ではないので無視され、成功として数えられます。
- **配列の要素が文字列なら、それがメッセージになる。**ただし空文字列は、メッセージがないものとして成功に数えられます。
- **要素がオブジェクトなら、msg のキーの文字列がメッセージになる。**msg 以外のキーは、3 つ目のコマンドの -o json の出力のように、メタデータとして結果に含まれます。msg がないオブジェクトは、エラーになります。
- **それ以外の要素（数値など）は無視される。**

特に注意したいのは deny_bool です。deny if { ... } と書くと、条件が成り立っても Conftest は失敗として扱いません。エラーにもならないので、検査が効いていないことに気づきにくい書き方です。deny は必ず contains を使った集合のルールとして書きます。

### 20.3.1 名前の規則

Conftest が評価するのは、次の名前のルールです。**名前** は英数字で、deny_no_root のように **_名前** を複数続けることもできます。denyfoo のように _ なしで続けた名前は対象になりません。

| 名前 | 扱い |
|---|---|
| deny、deny_**名前**、violation、violation_**名前** | 失敗 |
| warn、warn_**名前** | 警告 |

これ以外の名前のルールは、ほかのルールから参照される補助のルールとして評価されるだけで、結果には表示されません。

### 20.3.2 テスト数の数え方

conftest test の最後の行の数は、次のように数えています。

- 失敗・警告は、メッセージ 1 つごとに 1 つと数えます。1 つのルールが 3 つのメッセージを出せば、3 つです。
- 例外(後で説明します)は、例外が当てはまったルールごとに 1 つと数えます。
- 成功は、おおむね、メッセージを出さなかったルールの定義の数です。

上の例では、6 つのルールの定義のうち、メッセージを出したのは deny_object の 1 つだけなので、「6 tests, 5 passed, 1 failure」になりました。

## 20.4 deny[msg] if に注意

古い Rego（v0)では、集合を作るルールを deny[msg] { ... } と書いていました。Rego v1 で同じように deny[msg] if { ... } と書くと、意味が変わります。

**コード 20.5** main.rego

```
package main

deny[msg] if {
    input.name == "web"
    msg := "name is web"
}
```

```
$ opa eval -d policy -i <(conftest parse config.yaml) --format pretty 'data.main'
{
  "deny": {
    "name is web": true
  }
}
$ conftest test config.yaml
Error: running test: load: loading policies: rule is using 'if' keyword without 'contains'
```

opa eval の結果を見ると、deny は集合ではなく、{"name is web": true} というオブジェクトになっています。Rego v1 では、deny[msg] if は「ルール」の章で説明したオブジェクトを作るルール(値を書かなければ true)になるからです。

このまま評価すると、Conftest は値が配列でないので何も見ず、検査が効かなくなってしまいます。そのため Conftest は、ポリシーを読み込む時点でこの書き方を見つけて、エラーにしています。インターネット上の古い例を参考にするときは、deny contains msg if に書き直してください。

## 20.5　例外

特定のファイルだけ、あるルールの検査を外したいことがあります。exception という名前のルールで、検査を外すルールを指定できます。

**コード 20.6** main.rego

```
package main

deny_latest contains "do not use the latest tag" if {
    endswith(input.image, ":latest")
}

deny_root contains "do not run as root" if {
    input.user == "root"
}

exception contains rules if {
    data.conftest.file.name == "legacy.yaml"
    rules := ["latest"]
}
```

```
$ conftest test app.yaml legacy.yaml
FAIL - app.yaml - main - do not use the latest tag
FAIL - app.yaml - main - do not run as root
FAIL - legacy.yaml - main - do not run as root
EXCP - legacy.yaml - main - data.main.exception[_][_] == "latest"

4 tests, 0 passed, 0 warnings, 3 failures, 1 exception
```

exception は、検査を外すルール名の配列を要素に持つ集合です。ルール名は、先頭の deny_ などを除いた部分で書きます。["latest"] は、deny_latest を指しています。

Conftest は、deny_latest を評価する前に、data.main.exception[_][_] == "latest" を評価しています。これが成り立てば、deny_latest の結果はすべて例外(EXCP)として扱われ、失敗に数えられません。

exception の本体では、data.conftest.file.name を参照しています。Conftest は、検査しているファイルの名前を data.conftest.file.name に、ディレクトリを data.conftest.file.dir に入れてから評価します。そのため、ポリシーからファイル名を使った条件を書けます。この例では、legacy.yaml のときだけ latest の検査を外しています。deny_root は外していないので、legacy.yaml でも失敗しています。

## 20.6　名前空間

Conftest は、指定しなければ `main` パッケージのルールだけを評価します。

**コード 20.7** `kubernetes.rego`

```
package kubernetes

warn contains "checked by kubernetes" if {
    input.name == "web"
}
```

```
$ conftest test config.yaml
WARN - config.yaml - main - checked by main

1 test, 0 passed, 1 warning, 0 failures, 0 exceptions
$ conftest test --namespace kubernetes config.yaml
WARN - config.yaml - kubernetes - checked by kubernetes

1 test, 0 passed, 1 warning, 0 failures, 0 exceptions
$ conftest test --all-namespaces config.yaml
WARN - config.yaml - kubernetes - checked by kubernetes
WARN - config.yaml - main - checked by main

2 tests, 0 passed, 2 warnings, 0 failures, 0 exceptions
```

`--namespace` でパッケージを指定でき、`--all-namespaces` ですべてのパッケージを評価できます。結果の 3 番目の列(`main`、`kubernetes`)が、ルールのパッケージを表しています。

## 20.7　まとめ

| | Rego 側 | Conftest 側 |
|---|---|---|
| 入力 | `input` を参照する | ファイルを変換して `input` を作る |

| | Rego 側 | Conftest 側 |
|---|---|---|
| `deny` などのルール | 名前に特別な意味はない。集合などの値を計算する | 名前で選んで `data.`**名前空間**`.`**ルール名** を評価する |
| 値 | 条件を満たせば値が決まる | 配列の要素の文字列(またはオブジェクトの `msg`)だけをメッセージとして見る |
| 例外 | `exception` の値を計算する | `exception[_][_] == "`**ルール名**`"` を評価して、結果を例外にする |

- Conftest は、名前が `deny`・`violation`・`warn`（とそれに **_名前** を続けたもの)のルールの値を見て、成否を決めます。
- 値が配列でなければ無視されます。`deny if` や `deny[msg] if` ではなく、`deny contains msg if` と書きます。
- 配列の要素は、文字列か、`msg` を持つオブジェクトにします。

# 第 21 章

# Conftest の使い方

前の章では、Conftest がルールの値をどう扱うかを説明しました。この章では、`conftest test` の主なオプションと、ポリシーのテストを実行する `conftest verify` を説明します。

## 21.1 入力のファイル

`conftest test` には、検査するファイルを複数指定できます。ディレクトリを指定すると、その中のファイルがまとめて検査されます。ポリシーは、指定しなければ `policy` ディレクトリから読み込まれ、`-p` で別のディレクトリを指定できます。

### 21.1.1 複数のドキュメントを含む YAML

Kubernetes のマニフェストのように、1 つの YAML ファイルに `---` で区切った複数のドキュメントを書くことがあります。

**コード 21.1** `manifests.yaml`

```yaml
kind: Service
metadata:
  name: web
---
kind: Deployment
metadata:
  name: web
```

コード 21.2 main.rego

```
package main

deny contains msg if {
    input.kind == "Deployment"
    not input.metadata.labels
    msg := $"{input.metadata.name}: Deployment must have labels"
}
```

```
$ conftest parse manifests.yaml
[
  {
    "kind": "Service",
    "metadata": {
      "name": "web"
    }
  },
  {
    "kind": "Deployment",
    "metadata": {
      "name": "web"
    }
  }
]
$ conftest test manifests.yaml
FAIL - manifests.yaml - main - web: Deployment must have labels

2 tests, 1 passed, 0 warnings, 1 failure, 0 exceptions
```

conftest parse では、2 つのドキュメントが配列として表示されます。しかし、conftest test は、ドキュメントを 1 つずつ別の input として評価します。そのため、ポリシーでは input.kind のように、1 つのドキュメントを対象に書けます。結果の「2 tests」は、2 つのドキュメントをそれぞれ評価した数です。

### 21.1.2 ファイルをまとめて評価する

ファイルどうしの関係を調べたいときは、`--combine` を使います。

**コード 21.3** `deployment.yaml`

```yaml
kind: Deployment
metadata:
  name: web
```

**コード 21.4** `service.yaml`

```yaml
kind: Service
metadata:
  name: api
```

**コード 21.5** `main.rego`

```
package main

import future.keywords.not

service_names contains file.contents.metadata.name if {
    some file in input
    file.contents.kind == "Service"
}

deny contains msg if {
    some file in input
    file.contents.kind == "Deployment"
    name := file.contents.metadata.name
    not name in service_names
    msg := $"{file.path}: no Service named {name}"
}
```

```
$ conftest parse --combine deployment.yaml service.yaml
[
        {
                "path": "deployment.yaml",
                "contents": {
                        "kind": "Deployment",
                        "metadata": {
                                "name": "web"
                        }
                }
        },
        {
                "path": "service.yaml",
                "contents": {
                        "kind": "Service",
                        "metadata": {
                                "name": "api"
                        }
                }
        }
]

$ conftest test --combine deployment.yaml service.yaml
FAIL - Combined - main - deployment.yaml: no Service named web

1 test, 0 passed, 0 warnings, 1 failure, 0 exceptions
```

`--combine` を付けると、すべてのファイルが 1 つの `input` にまとめられます。`input` は配列で、要素はファイルごとの `path`（ファイル名）と `contents`（内容）を持つオブジェクトです。

このポリシーは、Deployment と同じ名前の Service があるかを調べています。`service_names` で Service の名前を集め、Deployment の名前が含まれていなければ失敗にしています。結果の 2 番目の列は、ファイル名ではなく `Combined` になります。

### 21.1.3　パーサーを指定する

Conftest は、ファイルの拡張子からパーサー（ファイルを読み込んで input に変換する部分）を選びます。拡張子から形式が分からないときは、--parser で指定します。

**コード 21.6** app.conf

```
[server]
port = 8080
```

```
$ conftest parse app.conf
Error: parse configurations: new parser: new: unknown parser: conf, path: app.conf
$ conftest parse --parser toml app.conf
{
  "server": {
    "port": 8080
  }
}
```

.conf という拡張子からはパーサーが決まらないので、エラーになりました。--parser toml を付けると、TOML として読み込まれます。指定できるパーサーは conftest test --help で確認できます。

## 21.2　データを読み込む

-d でディレクトリを指定すると、その中の JSON や YAML のファイルがデータとして読み込まれます。データのパスは、opa eval -d と同じく、ディレクトリの名前で決まります。

**コード 21.7** data.json

```
{
  "allowed": ["registry.example.com", "ghcr.io"]
}
```

コード 21.8 pod.yaml

```yaml
kind: Pod
spec:
  containers:
    - name: app
      image: ghcr.io/example/app:1.4.2
    - name: proxy
      image: docker.io/library/nginx:1.27
```

コード 21.9 main.rego

```
package main

import future.keywords.not

registry(image) := split(image, "/")[0]

deny contains msg if {
    some c in input.spec.containers
    not registry(c.image) in data.registries.allowed
    msg := $"{c.name}: registry {registry(c.image)} is not allowed"
}
```

```
$ opa eval -d data --format pretty 'data'
{
  "registries": {
    "allowed": [
      "registry.example.com",
      "ghcr.io"
    ]
  }
}
$ conftest test -d data pod.yaml
FAIL - pod.yaml - main - proxy: registry docker.io is not allowed

1 test, 0 passed, 0 warnings, 1 failure, 0 exceptions
```

1 つ目のコマンドのように、data/registries/data.json の内容は、ディレクトリの名前から data.registries に読み込まれます。-d に指定したディレクトリ（data）の名前はパスに含まれません。また、ファイルを data/registries.json のように置くと、ファイル名はパスにならず、内容は data の直下に読み込まれます。

2 つ目のコマンドでは、許可した ghcr.io のイメージを使う app は通り、docker.io のイメージを使う proxy だけが失敗しました。許可するレジストリの一覧をデータに分けておくと、ポリシーを変えずに一覧だけを更新できます。

## 21.3　結果の表示と終了コード

-o で結果の表示形式を選べます。--fail-on-warn を付けると、警告でも 0 以外の終了コードで終わります。

**コード 21.10** main.rego

```
package main

warn contains "replicas should be at least 2" if {
    input.replicas < 2
}
```

```
$ conftest test -o table config.yaml
┌─────────┬─────────────┬───────────┬──────────────────────────────────┐
│ RESULT  │    FILE     │ NAMESPACE │            MESSAGE               │
├─────────┼─────────────┼───────────┼──────────────────────────────────┤
│ warning │ config.yaml │ main      │ replicas should be at least 2 │
└─────────┴─────────────┴───────────┴──────────────────────────────────┘
$ conftest test config.yaml; echo "exit code: $?"
WARN - config.yaml - main - replicas should be at least 2

1 test, 0 passed, 1 warning, 0 failures, 0 exceptions
exit code: 0
$ conftest test --fail-on-warn config.yaml; echo "exit code: $?"
WARN - config.yaml - main - replicas should be at least 2

1 test, 0 passed, 1 warning, 0 failures, 0 exceptions
```

```
exit code: 1
```

`-o table` は表の形で表示します。ほかに `json`、`junit`（CI のテスト結果の形式）、`github`（GitHub Actions の注釈の形式）、`sarif` などを選べます。

`conftest test` は、失敗があると 1 で終わり、警告だけなら 0 で終わります。`--fail-on-warn` を付けると、警告だけでも 1 で終わります。逆に `--no-fail` を付けると、失敗があっても 0 で終わります。

## 21.4　ポリシーをテストする

`conftest verify` は、ポリシーのディレクトリにある `test_` で始まるルールを実行します。「OPA の CLI」の章の `opa test` と同じように、テストも Rego で書きます。

**コード 21.11** `main_test.rego`

```
package main

test_deployment_without_labels if {
    cfg := parse_config("yaml", `
kind: Deployment
metadata:
  name: web
`)
    deny["web: Deployment must have labels"] with input as cfg
}

test_deployment_with_labels if {
    cfg := parse_config("yaml", `
kind: Deployment
metadata:
  name: web
  labels:
    app: web
`)
    count(deny) == 0 with input as cfg
}
```

parse_config は、Conftest が追加している組み込み関数です。1 つ目の引数のパーサーで、2 つ目の引数の文字列を値に変換します。テストの中に YAML をそのまま書いて、conftest test で読み込んだときと同じ input を作れます。YAML はインデントが意味を持つので、改行をそのまま書けるローストリングを使っています。

deny["web: Deployment must have labels"] は、「変数と参照」の章で説明した集合の参照です。deny にそのメッセージが含まれていれば成り立ちます。count(deny) == 0 は、deny が空であることを調べています。

```
$ conftest verify

2 tests, 2 passed, 0 warnings, 0 failures, 0 exceptions, 0 skipped
$ conftest verify --report full
policy/main_test.rego:12:
data.main.test_deployment_with_labels: PASS (344.281µs)
policy/main_test.rego:3:
data.main.test_deployment_without_labels: PASS (401.345µs)
--------------------------------------------------------------------------------
PASS: 2/2
```

--report full を付けると、テストごとの結果が表示されます。

parse_config を使うテストは Conftest の組み込み関数に依存するので、opa test では実行できません。Conftest のポリシーのテストは conftest verify で実行します。

## 21.5 まとめ

- conftest test は、複数ドキュメントの YAML をドキュメントごとに評価します。--combine で全ファイルを 1 つの input にまとめられます。
- パーサーは拡張子で決まり、--parser で指定できます。-d でデータを読み込めます。
- -o で表示形式を、--fail-on-warn や --no-fail で終了コードを変えられます。
- conftest verify でポリシーのテストを実行します。parse_config で、テストの中で設定ファイルの文字列を input に変換できます。

# 第 22 章

# Kubernetes マニフェスト

この章では、Conftest で Kubernetes のマニフェストを検査します。前半の「ドメインの情報」で検査する対象の構造を説明し、後半の「Rego の書き方」でポリシーを説明します。前半は Rego と関係なく、Kubernetes を知っている人なら読み飛ばせる内容です。

## 22.1 ドメインの情報: マニフェストの構造

Kubernetes では、アプリケーションの構成をマニフェストという YAML のファイルに書きます。マニフェストの各ドキュメントは、1 つのリソース(Kubernetes が管理するもの)を表します。どのリソースにも、次の項目があります。

| 項目 | 内容 |
|---|---|
| `apiVersion` | リソースの API のバージョン(`v1`、`apps/v1` など) |
| `kind` | リソースの種類(`Pod`、`Deployment`、`Service` など) |
| `metadata` | 名前(`name`)やラベル(`labels`)など |
| `spec` | リソースの内容。種類ごとに構造が違う |

コンテナを動かすリソースのうち、Pod はコンテナを直接定義します。Deployment などは、Pod の雛形(テンプレート)を持ち、そこから Pod を作ります。そのため、コンテナの一覧が置かれる場所が違います。

| 種類 | コンテナの一覧の場所 |
|---|---|
| Pod | `spec.containers` |

| 種類 | コンテナの一覧の場所 |
|---|---|
| Deployment、StatefulSet、DaemonSet、ReplicaSet、Job | `spec.template.spec.containers` |

コンテナの主な項目は次のとおりです。

| 項目 | 内容 |
|---|---|
| `name` | コンテナの名前 |
| `image` | コンテナイメージ。**レジストリ/名前：タグ**の形で、タグを省略すると `latest` になる |
| `resources.limits` | CPU やメモリの使用量の上限 |
| `securityContext.privileged` | `true` にすると、コンテナがホストのほぼすべての権限を持つ |

検査するマニフェストです。4 つのドキュメントを `---` で区切っています。

**コード 22.1** manifests.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  labels:
    app: web
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: registry.example.com/web:1.4.2
          resources:
            limits:
              cpu: 500m
              memory: 256Mi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: batch
spec:
  replicas: 1
  selector:
    matchLabels:
      app: batch
  template:
    metadata:
      labels:
        app: batch
    spec:
      containers:
```

- `web` の Deployment は、イメージのタグを固定し、メモリの上限を設定しています。
- `batch` の Deployment は、タグが `latest` で、特権コンテナ(`privileged: true`)を使い、メモリの上限がありません。`metadata.labels` もありません。
- `debug` の Pod は、イメージ `busybox` にタグがなく、メモリの上限もありません。
- `web` の Service は、コンテナを持たないリソースです。

## 22.2 Rego の書き方

次の 4 つの決まりを検査します。

- 特権コンテナを使わない(失敗)
- コンテナにメモリの上限を設定する(失敗)
- イメージのタグを固定し、`latest` を使わない(失敗)
- Deployment などに `app` ラベルを付ける(警告)

コード 22.2 main.rego

```
package main

import future.keywords.not

workload_kinds := {"Deployment", "StatefulSet", "DaemonSet", "ReplicaSet", "Job"}

# Pod と、Pod を作るワークロードから、コンテナを取り出す
containers contains c if {
    input.kind == "Pod"
    some c in input.spec.containers
}

containers contains c if {
    input.kind in workload_kinds
    some c in input.spec.template.spec.containers
}

name := input.metadata.name

deny contains msg if {
    some c in containers
    c.securityContext.privileged == true
    msg := $"{input.kind}/{name}: container {c.name} must not be privileged"
}

deny contains msg if {
    some c in containers
    not c.resources.limits.memory
    msg := $"{input.kind}/{name}: container {c.name} must set resources.limits.memory"
}

deny contains msg if {
    some c in containers
    not has_fixed_tag(c.image)
    msg := $"{input.kind}/{name}: container {c.name} must use a fixed image tag"
}

# イメージの最後の "/" より後ろに ":タグ" があり、タグが latest でない
has_fixed_tag(image) if {
```

### 22.2.1 コンテナを取り出す補助のルール

コンテナの一覧の場所は種類によって違うので、containers という補助のルールで違いを吸収しています。containers は、「ルール」の章で説明した集合を作るルールで、2 つの定義に分けて書いています。

- 1 つ目の定義は、kind が Pod のとき、spec.containers からコンテナを集めます。
- 2 つ目の定義は、kind が workload_kinds に含まれるとき、spec.template.spec.containers から集めます。

2 つの定義の結果は合わせられるので、containers には、どの種類でもそのリソースのコンテナが集まります。Service のようにコンテナを持たないリソースでは、containers は空の集合になり、コンテナについての deny はどれも成り立ちません。

deny の各ルールは、some c in containers でコンテナを 1 つずつ取り出して調べています。種類による違いを気にせずに書けます。

containers と name は、名前が deny や warn で始まらないので、前の章までに説明したとおり、Conftest の結果には表示されません。

### 22.2.2 各ルール

- 特権コンテナの検査は、c.securityContext.privileged == true で書いています。securityContext がないコンテナでは参照が undefined になり、条件は成り立ちません。
- メモリの上限の検査は、not c.resources.limits.memory で、値がないことを調べています。「否定」の章で説明したとおり、途中の resources や limits がない場合も、not は成り立ちます。
- タグの検査は、関数 has_fixed_tag で書いています。regex.match で、イメージの最後の / より後ろに : **タグ** があるかを調べ、さらに :latest で終わらないことを調べています。registry.example.com:5000/web のように、レジストリにポート番号が付いていても誤判定しません。
- ラベルの検査は warn にしています。守ってほしいが、守らなくても動作に問題はない決まりは、警告にしておくと段階的に導入しやすくなります。

メッセージには、{input.kind}/{name} とコンテナ名を入れています。1 つのファイルに複数のリソースがあっても、どのリソースのどのコンテナの問題かが分かります。

### 22.2.3 検査とテスト

**コード 22.3** main_test.rego

```
package main

test_privileged_container_is_denied if {
    cfg := parse_config("yaml", `
kind: Pod
metadata:
  name: p
spec:
  containers:
    - name: c
      image: registry.example.com/app:1.0.0
      resources:
        limits:
          memory: 64Mi
      securityContext:
        privileged: true
`)
    deny["Pod/p: container c must not be privileged"] with input as cfg
}

test_compliant_deployment_passes if {
    cfg := parse_config("yaml", `
kind: Deployment
metadata:
  name: d
  labels:
    app: d
spec:
  template:
    spec:
      containers:
        - name: c
          image: registry.example.com/app:1.0.0
          resources:
            limits:
              memory: 64Mi
```

```
$ conftest test manifests.yaml
WARN - manifests.yaml - main - Deployment/batch: metadata.labels.app is not set
FAIL - manifests.yaml - main - Deployment/batch: container worker must not be privileged
FAIL - manifests.yaml - main - Deployment/batch: container worker must set resources.limit
FAIL - manifests.yaml - main - Deployment/batch: container worker must use a fixed image t
FAIL - manifests.yaml - main - Pod/debug: container shell must set resources.limits.memory
FAIL - manifests.yaml - main - Pod/debug: container shell must use a fixed image tag

16 tests, 10 passed, 1 warning, 5 failures, 0 exceptions
$ conftest verify

2 tests, 2 passed, 0 warnings, 0 failures, 0 exceptions, 0 skipped
```

`conftest test` は、`batch` の Deployment と `debug` の Pod の問題を報告し、`web` の Deployment と Service については何も報告していません。

テストでは、特権コンテナの Pod が `deny` されることと、決まりを守った Deployment で `deny` と `warn` が空になることを確かめています。ポリシーを変えたときに、守っている設定まで失敗にしてしまう誤りに気づけます。

## 22.3 まとめ

- Kubernetes のマニフェストは、`apiVersion`、`kind`、`metadata`、`spec` を持ちます。コンテナの一覧の場所は、Pod と Deployment などで違います。
- 種類による違いは、集合を作る補助のルールで吸収すると、検査のルールを種類によらず書けます。
- メッセージにリソースの種類と名前を入れると、どこの問題かが分かります。

# 第 23 章

# Terraform

この章では、Terraform の設定を検査します。Terraform は、Conftest で検査できる対象が 2 つあります。HCL で書いたソース（`.tf` ファイル）と、変更の計画（plan）を JSON にしたものです。前半でそれぞれの構造を説明し、後半で同じ決まりを両方に対して書いて比べます。

## 23.1 ドメインの情報: HCL のソースと plan

Terraform では、作りたいクラウドのリソースを HCL という言語で `.tf` ファイルに書きます。`terraform plan` を実行すると、Terraform は現在の状態と比べて、どのリソースをどう変更するかの計画を作ります。この計画を `terraform show -json` で JSON に変換したものを、ここでは plan の JSON と呼びます。

```
$ terraform plan -out tfplan
$ terraform show -json tfplan > plan.json
```

この章では、次の 3 つのリソースを例にします。

コード 23.1 main.tf

```
variable "tags" {
  type = map(string)
  default = {
    Owner = "platform"
  }
}

resource "aws_s3_bucket" "logs" {
  bucket = "example-logs"
  tags   = var.tags
}

resource "aws_s3_bucket" "tmp" {
  bucket = "example-tmp"
}

resource "aws_security_group_rule" "ssh" {
  type              = "ingress"
  from_port         = 22
  to_port           = 22
  protocol          = "tcp"
  cidr_blocks       = ["0.0.0.0/0"]
  security_group_id = "sg-12345678"
}
```

- aws_s3_bucket.logs は、S3 のバケットです。タグを変数 var.tags で指定していて、その既定値は {Owner = "platform"} です。
- aws_s3_bucket.tmp は、タグのないバケットです。
- aws_security_group_rule.ssh は、SSH (22 番ポート)への接続を、すべての IP アドレス(0.0.0.0/0)から許可する設定です。

### 23.1.1 HCL のソースの構造

Conftest は、.tf ファイルを次のような input に変換します。

```
$ conftest parse main.tf
{
  "resource": {
    "aws_s3_bucket": {
      "logs": [
        {
          "bucket": "example-logs",
          "tags": "${var.tags}"
        }
      ],
      "tmp": [
        {
          "bucket": "example-tmp"
        }
      ]
    },
    "aws_security_group_rule": {
      "ssh": [
        {
          "cidr_blocks": [
            "0.0.0.0/0"
          ],
          "from_port": 22,
          "protocol": "tcp",
          "security_group_id": "sg-12345678",
          "to_port": 22,
          "type": "ingress"
        }
      ]
    }
  },
  "variable": {
    "tags": [
      {
        "default": {
```

```
          "Owner": "platform"
        },
        "type": "${map(string)}"
      }
    ]
  }
}
```

resource の下に、リソースの種類(aws_s3_bucket)、名前(logs)の順に入れ子になっています。ブロックの内容は配列の要素として入っています。

注目したいのは、logs の tags が "${var.tags}" という文字列になっていることです。HCL のソースには、変数を参照するという書き方しか書かれていません。変数の値を展開するのは Terraform なので、ソースを読むだけでは実際のタグは分かりません。

### 23.1.2 plan の JSON の構造

plan の JSON は、Terraform が変数やモジュールを展開した後の値を含みます。plan の JSON を作るために使った .tf ファイルには、AWS に接続せずに plan を作るためのプロバイダの設定を加えています。

**コード 23.2** `main.tf`

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 6.0"
    }
  }
}

provider "aws" {
  region                      = "ap-northeast-1"
  access_key                  = "dummy"
  secret_key                  = "dummy"
  skip_credentials_validation = true
  skip_requesting_account_id  = true
  skip_metadata_api_check     = true
}

variable "tags" {
  type = map(string)
  default = {
    Owner = "platform"
  }
}

resource "aws_s3_bucket" "logs" {
  bucket = "example-logs"
  tags   = var.tags
}

resource "aws_s3_bucket" "tmp" {
  bucket = "example-tmp"
}

resource "aws_security_group_rule" "ssh" {
  type              = "ingress"
  from_port         = 22
  to_port           = 22
  protocol          = "tcp"
```

変更の内容は resource_changes にあります。1 件を、json.filter（オブジェクトから指定したパスだけを残す組み込み関数）で主な項目に絞って表示します。

```
$ opa eval -i plan.json --format pretty 'json.filter(input.resource_changes[0], ["address"
{
  "address": "aws_s3_bucket.logs",
  "change": {
    "actions": [
      "create"
    ],
    "after": {
      "bucket": "example-logs",
      "force_destroy": false,
      "region": "ap-northeast-1",
      "tags": {
        "Owner": "platform"
      },
      "tags_all": {
        "Owner": "platform"
      },
      "timeouts": null
    }
  },
  "mode": "managed",
  "name": "logs",
  "type": "aws_s3_bucket"
}
```

| 項目 | 内容 |
|---|---|
| address | リソースの住所（**種類**. **名前**） |
| mode | 作るリソースは managed、読み込むだけのデータソースは data |
| type、name | リソースの種類と名前 |
| change.actions | 変更の種類。create、update、delete など。作り直す場合は ["delete", "create"] |
| change.after | 変更後の値。削除する場合は null |

change.after.tags には、変数を展開した {"Owner": "platform"} が入っています。

## 23.2 Rego の書き方

次の決まりを検査します。

- S3 のバケットには Owner のタグを付ける
- SSH をすべての IP アドレスに開放しない
- （plan だけ）リソースを削除しない

### 23.2.1 HCL のソースを検査する

**コード 23.3** main.rego

```
package main

import future.keywords.not

deny contains msg if {
    some name, blocks in input.resource.aws_s3_bucket
    some bucket in blocks
    not bucket.tags.Owner
    msg := $"aws_s3_bucket.{name}: tags.Owner is required"
}

deny contains msg if {
    some name, blocks in input.resource.aws_security_group_rule
    some rule in blocks
    rule.type == "ingress"
    rule.from_port <= 22
    rule.to_port >= 22
    "0.0.0.0/0" in rule.cidr_blocks
    msg := $"aws_security_group_rule.{name}: SSH must not be open to the internet"
}
```

```
$ conftest test main.tf
FAIL - main.tf - main - aws_s3_bucket.logs: tags.Owner is required
FAIL - main.tf - main - aws_s3_bucket.tmp: tags.Owner is required
FAIL - main.tf - main - aws_security_group_rule.ssh: SSH must not be open to the internet

3 tests, 0 passed, 0 warnings, 3 failures, 0 exceptions
```

`input.resource.aws_s3_bucket` はリソース名をキーにしたオブジェクトなので、`some name, blocks in` で名前とブロックの配列を取り出し、さらに `some bucket in blocks` でブロックを取り出しています。

結果を見ると、`logs` も `tags.Owner is required` で失敗しています。`logs` の `tags` は `"${var.tags}"` という文字列なので、`bucket.tags.Owner` は undefined になるからです。実際には `Owner` のタグが付くのに、ソースからは分からないので、誤って失敗にしてしまいました。

HCL のソースを検査する方法は、`terraform plan` を実行しなくてよいので手軽です。ただし、変数・モジュール・関数で決まる値は検査できません。

### 23.2.2 plan の JSON を検査する

コード 23.4 main.rego

```
package main

import future.keywords.not

# 作成・更新後の値があるリソースの変更
changes contains rc if {
    some rc in input.resource_changes
    rc.mode == "managed"
    is_object(rc.change.after)
}

deny contains msg if {
    some rc in changes
    rc.type == "aws_s3_bucket"
    not rc.change.after.tags.Owner
    msg := $"{rc.address}: tags.Owner is required"
}

deny contains msg if {
    some rc in changes
    rc.type == "aws_security_group_rule"
    after := rc.change.after
    after.type == "ingress"
    after.from_port <= 22
    after.to_port >= 22
    "0.0.0.0/0" in after.cidr_blocks
    msg := $"{rc.address}: SSH must not be open to the internet"
}

deny contains msg if {
    some rc in input.resource_changes
    "delete" in rc.change.actions
    msg := $"{rc.address}: deleting resources is not allowed"
}
```

changes は、作成・更新後の値(change.after)があるリソースの変更を集める補助のルールです。削除の場合は change.after が null なので、is_object で除いています。データソース(mode が data)も除いています。

削除を禁止するルールは、change.actions に "delete" が含まれるかを調べています。作り直し(["delete", "create"])も含まれます。この例の plan には削除がないので、テストで確かめます。

**コード 23.5** main_test.rego

```
package main

test_delete_is_denied if {
    deny["aws_s3_bucket.old: deleting resources is not allowed"] with input as {"resource_
        "address": "aws_s3_bucket.old",
        "mode": "managed",
        "type": "aws_s3_bucket",
        "change": {"actions": ["delete"], "after": null},
    }]}
}
```

```
$ conftest test plan.json
FAIL - plan.json - main - aws_s3_bucket.tmp: tags.Owner is required
FAIL - plan.json - main - aws_security_group_rule.ssh: SSH must not be open to the interne

3 tests, 1 passed, 0 warnings, 2 failures, 0 exceptions
$ conftest verify

1 test, 1 passed, 0 warnings, 0 failures, 0 exceptions, 0 skipped
```

conftest test plan.json で失敗したのは、tmp のバケットと SSH の設定だけです。plan では logs の実際のタグが見えるので、誤って失敗にすることがなくなりました。conftest verify では、削除のテストが成功しています。

plan の JSON を検査する方法は、Terraform が計算した実際の値を検査できます。ただし、terraform plan を実行する必要があり、そのためにはプロバイダの認証情報などが必要になることがあります。また、作成するまで決まらない値(ID など)は、plan では値がなく after_unknown に記録されます。

## 23.3 まとめ

| | HCL のソース | plan の JSON |
|---|---|---|
| 準備 | 不要 | `terraform plan` と `terraform show -json` |
| `input` の構造 | `resource.` **種類**`.` **名前** の下にブロックの配列 | `resource_changes` の配列。値は `change.after` |
| 変数などの値 | 展開されない | 展開される |
| 変更の種類 | 分からない | `change.actions` で分かる |

- HCL のソースは手軽に検査できますが、変数などで決まる値は検査できません。
- plan の JSON は、変数を展開した実際の値と、変更の種類を検査できます。

# 第 24 章

# Dockerfile

この章では、コンテナイメージを作るための Dockerfile を検査します。前の 2 つの章と同じく、前半で Dockerfile の構造を、後半でポリシーを説明します。

## 24.1 ドメインの情報: Dockerfile の構造

Dockerfile は、コンテナイメージを作る手順を、1 行に 1 つの命令で書いたファイルです。主な命令は次のとおりです。

| 命令 | 内容 |
|---|---|
| `FROM` | 元にするイメージ(ベースイメージ)を指定する。`AS` **名前** でステージに名前を付けられる |
| `RUN` | コマンドを実行する |
| `COPY`、`ADD` | ファイルをイメージにコピーする。`ADD` は URL からのダウンロードやアーカイブの展開もする |
| `ENV` | 環境変数を設定する |
| `USER` | 以降の命令とコンテナの実行に使うユーザーを指定する。ベースイメージでも指定されていなければ root になる |
| `CMD` | コンテナを起動したときに実行するコマンドを指定する |

FROM を複数書くと、マルチステージビルドになります。FROM ごとに新しいステージが始まり、最終的なイメージになるのは最後のステージです。前のステージは、ビルドに必要なツールを使ってファイルを作るために使い、COPY --from=**ステージ名** で必要なファイルだけを最後のステージにコピーします。

Conftest は、Dockerfile を次のような input に変換します。

**コード 24.1** Dockerfile

```
FROM golang:1.27 AS build
RUN go build -o /app .
FROM debian:13.2-slim
ENV LOG_LEVEL=info TZ=UTC
```

```
$ conftest parse Dockerfile
[
  [
    {
      "Cmd": "from",
      "Flags": [],
      "JSON": false,
      "Stage": 0,
      "SubCmd": "",
      "Value": [
        "golang:1.27",
        "AS",
        "build"
      ]
    },
    {
      "Cmd": "run",
      "Flags": [],
      "JSON": false,
      "Stage": 0,
      "SubCmd": "",
      "Value": [
        "go build -o /app ."
```

```
      ]
    },
    {
      "Cmd": "from",
      "Flags": [],
      "JSON": false,
      "Stage": 1,
      "SubCmd": "",
      "Value": [
        "debian:13.2-slim"
      ]
    },
    {
      "Cmd": "env",
      "Flags": [],
      "JSON": false,
      "Stage": 1,
      "SubCmd": "",
      "Value": [
        "LOG_LEVEL",
        "info",
        "=",
        "TZ",
        "UTC",
        "="
      ]
    }
  ]
]
```

命令ごとに 1 つのオブジェクトになり、それらが書かれた順に配列に並んでいます。

| 項目 | 内容 |
|---|---|
| `Cmd` | 命令の名前(小文字) |

| 項目 | 内容 |
|---|---|
| `Value` | 命令の引数の配列。`FROM` などは空白で区切られるが、シェル形式の `RUN` はコマンド全体が 1 つの文字列になる |
| `Flags` | `--from=build` のようなオプション |
| `Stage` | 何番目のステージの命令か(0 から始まる) |

`ENV` の `Value` は、**キー**、**値**、区切りの 3 つずつが並んだ配列です。`LOG_LEVEL=info TZ=UTC` は、`["LOG_LEVEL", "info", "=", "TZ", "UTC", "="]` になっています。

`conftest parse` の結果は、命令の配列をさらに配列で囲んだ形になっています。「Conftest の使い方」の章の複数ドキュメントの YAML と同じく、`conftest test` では内側の配列(命令の配列)が `input` になります。

## 24.2 Rego の書き方

次の決まりを検査します。

- 最終ステージのベースイメージは、タグを固定する(`latest` やタグなしにしない)
- 最終ステージは、root 以外のユーザーで実行する
- `ADD` で URL からダウンロードしない
- `ENV` に秘密情報(パスワードやトークン)を入れない

検査する Dockerfile です。

コード 24.2 Dockerfile

```
FROM golang:1.27 AS build
WORKDIR /src
COPY . .
RUN go build -o /app .

FROM debian:latest
ENV LOG_LEVEL=info API_PASSWORD=secret
ADD https://example.com/config.tar.gz /etc/app/
COPY --from=build /app /usr/local/bin/app
USER root
CMD ["app"]
```

コード 24.3 main.rego

```
package main

import future.keywords.not

# 最後のステージ(最終的なイメージを作るステージ)の番号
final_stage := max({instr.Stage | some instr in input})

final_instructions contains instr if {
    some instr in input
    instr.Stage == final_stage
}

# 最終ステージのベースイメージのタグを固定する
deny contains msg if {
    some instr in final_instructions
    instr.Cmd == "from"
    image := instr.Value[0]
    image != "scratch"
    not has_fixed_tag(image)
    msg := $"FROM {image}: use a fixed tag for the base image"
}

has_fixed_tag(image) if {
    regex.match(`:[^/:]+$`, image)
    not endswith(image, ":latest")
}

has_fixed_tag(image) if {
    regex.match(`@sha256:[0-9a-f]{64}$`, image)
}

# 最終ステージを root 以外のユーザーで実行する
users := [instr.Value[0] |
    some instr in input
    instr.Stage == final_stage
    instr.Cmd == "user"
]

deny contains "the final stage must set USER" if count(users) == 0
```

### 24.2.1 最終ステージを取り出す

final_stage は、すべての命令の Stage の最大値で、最後のステージの番号です。final_instructions は、最後のステージの命令を集めた集合です。ベースイメージの検査では、final_instructions から FROM を探しています。ビルド用のステージ（golang:1.27）は、最終的なイメージに含まれないので検査しません。

has_fixed_tag は、「Kubernetes マニフェスト」の章と同じ考え方の関数です。2 つ目の定義で、@sha256:... のようにダイジェストで指定したイメージも、固定されているものとして扱っています。

### 24.2.2 最後の USER を調べる

USER は複数書くことができ、最後に書いたものがコンテナの実行時のユーザーになります。そのため、最終ステージの USER の値を、書かれた順に並べる必要があります。

users は、配列の内包表記で作っています。集合だと順番が失われますが、配列の内包表記は input の順番を保つので、最後の要素 users[count(users) - 1] が最後の USER になります。

- USER がなければ失敗にしています。ベースイメージがユーザーを設定していなければ root で実行されますが、それは Dockerfile だけでは分からないので、最終ステージで明示することを求めています。
- 最後の USER が root か 0（root のユーザー ID）なら失敗にしています。USER 10001:10001 のようにグループを付けて書けるので、split で : の前だけを取り出して比べています。

### 24.2.3 ADD と ENV

ADD の検査では、最初の引数が http:// か https:// で始まるかを正規表現で調べています。

ENV の検査では、some i, key in instr.Value で添字と値を取り出し、i % 3 == 0 で 3 つずつの並びの先頭（キー）だけを調べています。正規表現の (?i) は、大文字と小文字を区別しない指定です。

### 24.2.4　検査とテスト

コード 24.4 Dockerfile

```
FROM golang:1.27 AS build
WORKDIR /src
COPY . .
RUN go build -o /app .

FROM debian:13.2-slim
ENV LOG_LEVEL=info
COPY --from=build /app /usr/local/bin/app
USER 10001:10001
CMD ["app"]
```

コード 24.5 main_test.rego

```
package main

test_missing_user_is_denied if {
    some doc in parse_config("dockerfile", `
FROM alpine:3.20
CMD ["sh"]
`)
    deny["the final stage must set USER"] with input as doc
}

test_user_in_build_stage_does_not_count if {
    some doc in parse_config("dockerfile", `
FROM alpine:3.20 AS build
USER builder
FROM alpine:3.20
USER root
`)
    deny["USER root: the final stage must not run as root"] with input as doc
}

test_last_user_wins if {
    some doc in parse_config("dockerfile", `
FROM alpine:3.20
USER root
RUN apk add --no-cache curl
USER app
`)
    count(deny) == 0 with input as doc
}
```

```
$ conftest test Dockerfile
FAIL - Dockerfile - main - ADD https://example.com/config.tar.gz: download files with RUN instead
FAIL - Dockerfile - main - ENV API_PASSWORD: do not put secrets in environment variables
FAIL - Dockerfile - main - FROM debian:latest: use a fixed tag for the base image
```

```
FAIL - Dockerfile - main - USER root: the final stage must not run as root

5 tests, 1 passed, 0 warnings, 4 failures, 0 exceptions
$ conftest test good/Dockerfile

5 tests, 5 passed, 0 warnings, 0 failures, 0 exceptions
$ conftest verify

3 tests, 3 passed, 0 warnings, 0 failures, 0 exceptions, 0 skipped
```

`Dockerfile` では 4 つの決まりすべてに違反し、`good/Dockerfile` ではすべてを守っています。

テストでは、`parse_config("dockerfile", ...)` で Dockerfile の文字列を変換しています。`parse_config` は、`conftest parse` と同じく、命令の配列をさらに配列で囲んだ値を返します。そのため、`some doc in parse_config(...)` で内側の配列を取り出して、`input` に使っています。2 つ目のテストでは、ビルド用のステージの `USER` が、最終ステージの検査に影響しないことを確かめています。3 つ目のテストでは、同じステージに `USER` が 2 つあるとき、最後のものが使われることを確かめています。

## 24.3 まとめ

- Conftest は、Dockerfile を命令ごとのオブジェクトの配列に変換します。`Stage` でステージを区別できます。
- 最終的なイメージを検査するときは、最後のステージの命令に絞ります。
- 命令の順番が意味を持つ検査では、集合ではなく配列の内包表記で、順番を保ったまま集めます。