GitHub Actions 実践入門

miyajan /著

GitHub Actions 実践入門

宮田 淳平 (@miyajan)　著

2020-03-05 版　　発行

まえがき

本書、『GitHub Actions 実践入門』では、GitHub が提供する CI/CD サービスである **GitHub Actions** の入門から、実際に活用してみるところまでを扱います。この本は、主に次の三点を意識しながら書かれています。

- GitHub Actions について体系的に学べる
- 実際に手を動かしながら学べる
- 普段 GitHub Actions を利用する上でリファレンスとして使える

また、本書のサンプルは、GitHub 上で公開しています。

- `https://github.com/github-actions-up-and-running`

謝辞

まず、GitHub（と Microsoft）のみなさま、GitHub Actions という素晴らしい CI/CD サービスを提供していただき本当にありがとうございます。筆者は CI/CD 関係の勉強会に参加することが多いですが、GitHub Actions の登場によってまた一段と界隈が盛り上がったのを感じています。

そして、この本を手に取ってくださった読者のみなさまにも感謝します。この本を読むみなさまにとってなにかしら得るものがあると嬉しいです。

対象読者

この本は、GitHub Actions の入門者から中級者を対象としています。Git と GitHub については事前に知識があるという前提で書かれています。

GitHub Actions の設定ファイルは YAML[*1] で書かれます。YAML に馴染みがない人は、事前に入門者向けの記事[*2]を読んでおくことをおすすめします。

[*1] `https://yaml.org/`

[*2] `https://magazine.rubyist.net/articles/0009/0009-YAML.html` など

CI/CD については第 1 章で簡単に説明しますが、詳細については解説しません。CI/CD という技術的プラクティスについてより詳しく理解したい人は、『継続的インテグレーション入門』[*3]や『継続的デリバリー』[*4]という書籍をおすすめします。

また、Node.js[*5] や Docker[*6] を題材として扱うことがあります。これらの知識がない人でも理解できるように説明しますが、気になる点があればそれぞれの公式ドキュメントを参照してください。

動作環境

本書のハンズオンなどでローカル環境で動かすプログラムやコマンドは、次の環境で動作確認しています。

- OS: macOS Catalina 10.15.2
- Node.js: 10.14.1
- npm: 6.13.4

基本的には他の OS やバージョンでも動作するはずですが、もし動かないところがあれば適宜ご自身の環境に合わせて読み替えてください。

本書の構成

本書は、四つの章に分かれています。

第 1 章「GitHub Actions の基礎知識」では、GitHub Actions とはなにか、使うとなにが嬉しいのかといった基礎知識を学び、実際に動かしてみるところまでを体験します。

第 2 章「GitHub Actions の機能解説」では、GitHub Actions が持つ機能について、ワークフローの構文などを中心に説明します。

第 3 章「アクション」では、GitHub Actions のワークフローの中で実行するアクションについて、アクションの自作方法を中心に説明します。

第 4 章「サンプルレシピ」では、GitHub Actions を実際に活用するための具体的なサンプルをいくつか紹介します。

GitHub Actions についてなにも知らないという人は、最初から全体を通して読むことで体系的な知識を付けられます。すでに GitHub Actions について一通り知っているという人には、気になるところだけ拾い読みすることもできます。

[*3] https://shop.nikkeibp.co.jp/front/commodity/0000/P83950/
[*4] https://www.kadokawa.co.jp/product/301706000296/
[*5] https://nodejs.org
[*6] https://www.docker.com/

お問い合わせ

本書についてのご質問、誤りの指摘、ご感想などありましたら、次の URL から気軽にご報告お願いします。

- https://github.com/github-actions-up-and-running/contact/issues

twitter の @miyajan アカウントにメンションや DM でご連絡いただいても構いません。

この本は、GitHub 公式のものではありません。GitHub に本書の内容についてのお問い合わせはしないでください。GitHub Actions の不具合報告や機能リクエストについては、GitHub 公式のサポート[*7]にフィードバックしてください。

免責事項

この本の内容は、執筆した 2020 年 2 月時点のものです。最新の情報については公式ドキュメント[*8]を参照してください。

この本の内容を利用することで発生したトラブルや損失、損害に対して、筆者は一切責任を負いません。利用者の責任においてご利用ください。

[*7] https://support.github.com/contact/feedback?contact%5Bcategory%5D=actions

[*8] https://help.github.com/en/actions

目次

第 1 章

GitHub Actions の基礎知識

本章では、GitHub Actions の概要について説明します。

1.1 GitHub Actions とは

GitHub Actions は、一言で表すと GitHub 組み込みの CI/CD システムです[*1]。例えば、あなたが GitHub のリポジトリにコードの変更をプッシュしたタイミングで、ビルド、テスト、デプロイといった一連のフローを自動実行させるようなことができます。

しかし、GitHub Actions の用途は CI/CD だけではありません。後述するように、GitHub の様々なイベントにフックして、自動で処理を実行できます。これにより、issue が登録されたときに担当者を自動で割り当てたり、PR（プルリクエスト）が作成されたときに Slack に通知を飛ばしたりといったこともできます。GitHub Actions は、ソフトウェア開発におけるあらゆるワークフローの自動化に使えるのです。

1.1.1 歴史

2018 年の GitHub Universe（GitHub 最大のコミュニティイベント）で、GitHub Actions がベータとして限定公開されました[*2]。この時点では GitHub は Actions を CI/CD のためのものとは位置づけておらず、あくまでもワークフローの自動化のためのシステムとしていました。

しかし、2019 年 8 月の時点で GitHub Actions の大幅なリニューアルが発表さ

[*1] GitHub Enterprise は現時点で GitHub Actions に対応していません。しかし、将来的に対応が予定されています。

[*2] `https://github.blog/changelog/2018-10-16-github-actions-limited-beta/`

れました[*3]。コンセプト面では CI/CD を前面に押し出すようになり、Linux だけでなく Windows や macOS 環境も利用できるようになり、ワークフローの設定ファイルの記述方法も HCL[*4] から YAML に変わりました。この変更の背景として、GitHub 社が Microsoft に買収されたことが挙げられます。GitHub Actions のバックエンドとして Azure Pipelines に変更を加えた fork 版が使われるようになりました。このため、GitHub Actions について調べるときは、2019 年 8 月以前に書かれた情報については基本的にもう参考にならないと覚えておいてください。

そして、2019 年の GitHub Universe で GitHub Actions は GA (Generally Available) になり[*5]、現在に至ります。GA 後も継続的に機能改善がリリースされており、今後もより便利になっていくことが期待されます。

CI/CD とは

CI（継続的インテグレーション）とは、技術的プラクティスの一つです。チームのメンバーは一日に何回もバージョン管理システムのリポジトリに変更をマージし、そのたびにテストを含む自動ビルドが毎回実行されるようにします。

CI を行う目的は、問題を早期発見できるように高速なフィードバックをシステム化することです。また、変更が頻繁にマージされるようになれば、一回の変更ごとの差分が小さくなります。これによって問題が発生するリスクが減りますし、万が一なにか発生したとしても差分が小さいので調査が簡単になります。また、CI が成功しているか失敗しているかといった状態が可視化されることによって、開発が順調に進んでいるかチーム内のコミュニケーションが活発になります。開発者にとっては、変更ごとに CI で保証されることによる心理的安心感も重要です。

CD（継続的デリバリー）は、CI をさらに発展させた技術的プラクティスです。コードが変更されるたびに自動でテストなどが実行されるのは CI と同じです。CD では、さらにソフトウェアがいつでもリリースできる状態であるところまで毎回保証できるようにします。具体的になにを自動化するかはソフトウェアの性質によりますが、例えば E2E テスト、性能検証、脆弱性検証、アーカイブ作成といったことも含めます。そして、コードを変更してからリリースするまでのこれらの検証を複数のステージに分けてパイプラ

[*3] https://github.blog/2019-08-08-github-actions-now-supports-ci-cd/
[*4] HashiCorp 社製の設定言語。https://github.com/hashicorp/hcl
[*5] https://github.blog/changelog/2019-11-11-github-actions-is-generally-available/

インとして表現したものを、デプロイメントパイプラインと呼びます。

CD を行うことによってリリースのコストやリスクを大幅に抑えられます。いつでもリリースできる状態を保つことによって、リリース直前に行う作業がほとんどなくなるためです。理想的な CD を実践できれば、リリースはボタン一つでできるようになります。また、デプロイメントパイプラインによってコード変更からリリースまでのフローが可視化されるので、どの部分で問題が起きているか、ボトルネックになっているかといったことが可視化されます。これによって、リリースまでのフローの改善にもつながりやすいです。

世の中には CI/CD を実現するためのツールがいくつも存在します。代表的なのは Jenkins[*6] です。Jenkins は OSS で手軽に各自の環境に構築することが可能で、かなり長い期間に渡って CI/CD 界隈のデファクトツールとして存在し続けています。その後、Travis CI[*7] や CircleCI[*8] などの登場によって、CI/CD のクラウドサービス化が進みました。開発者はそれまでよりも素早くお手軽に CI/CD を導入できるようになったのです。そして、継続的デリバリーの普及に伴い、デプロイメントパイプラインのような複数のジョブによるフローがワークフローとして表現されるようになりました。さらに、Docker によりコンテナ技術が一般的になり、CI/CD ツールでもコンテナ技術が取り入れられました。

GitHub Actions は、こういった CI/CD ツールの長い歴史の流れをくむ、最新の CI/CD ツールなのです。

1.2 主な特徴

GitHub Actions は、既存の CI/CD ツールとどういった点が異なるのでしょうか？

まず、GitHub のファーストパーティ製のサービスであるということが挙げられます。なので、GitHub の権限を外に持ち出すことなく CI/CD を実現することができます。

また、GitHub のさまざまな Webhook[*9] のイベントに対応しているというのも

[*6] https://jenkins.io/

[*7] https://travis-ci.org/

[*8] https://circleci.com/

[*9] Web アプリのなにかしらのイベント発生時に、なにかしらの処理を実行する仕組み

これまでの CI/CD ツールとは大きく異なる点です。従来の CI/CD ツールは `push` イベントによる CI/CD がメインで、それ以外のイベントはほとんどサポートされていませんでした。なので、例えば issue が登録されたときになにかしらの処理を自動で実行したい場合、これまでは自前で AWS 上に API Gateway + Lambda を構築したりするなど、どうにかしてイベント発生時に処理を実行する仕組みを作るしかありませんでした。しかし、GitHub Actions では様々なイベント発生時にもワークフローを自動で実行できるので、イベントにフックするワークフローも GitHub 内で完結して実行できるようになりました。つまり、GitHub Actions は CI/CD 用途に限らず、開発における様々な場面におけるワークフローを自動化できるということです。GitHub Actions が対応しているイベントについては「2.7 イベントとアクティビティ」で解説します。

そして、パブリックリポジトリは完全無料というのも大きな特徴です。既存の CI/CD サービスでも OSS 向けに無料プランはこれまでもあったのですが、実行時間や並列実行数などの制限があり、ある程度の規模になってくると物足りない面がありました。しかし、GitHub Actions はパブリックリポジトリならどれだけの時間使っても完全に無料で、さらに最大 20 並列まで実行できるので、かなりの規模の OSS 開発でも実用的です。

さらに、GitHub Actions にはセルフホストランナーという、独自の環境でワークフローを実行するための仕組みがあります。GitHub Actions が管理する環境でも、Linux、Windows、macOS と多様な環境でのワークフロー実行が可能なのですが、より強いマシンリソースを利用したいとか、独自にカスタマイズした環境でビルドしたいとか、外部から接続できないネットワーク内にデプロイしたいといった場合に、用意された環境だけでは実現できないことがあります。セルフホストランナーの仕組みは、そういったユースケースを可能にしてくれます。セルフホストランナーの詳細については、「2.12 ランナーのセルフホスティング」で解説します。

もちろん、これまでの CI/CD ツールが持っていた機能についても、GitHub Actions は大部分をサポートしています。公開後もどんどん改善が進んでおり、GitHub の大きなコミュニティが GitHub Actions のエコシステムにも影響を与えることも予想され、今後の発展性も期待できます。

1.3 料金体系

前の節でも少し触れましたが、GitHub Actions の**料金体系**についてもう少し詳しく見ておきます。

1.3.1 パブリックリポジトリ

GitHub Actions は、パブリックリポジトリでの利用は完全に無料です。

1.3.2 プライベートリポジトリ

プライベートリポジトリで GitHub Actions を利用する場合は、GitHub のプランによって決まった量の無料利用できるビルド時間とストレージが提供されます。無料分を超えた分については料金がかかります。

無料で利用できるビルド時間とストレージ

▼表 1.1　無料で利用できるビルド時間とストレージ

プラン	ビルド時間	ストレージ
Free	2,000	500 MB
Pro	3,000	1 GB
Team	10,000	2 GB
Enterprise Cloud	50,000	50 GB

表 1.1 の通りです。ビルド時間は毎月リセットされますが、ストレージは月をまたいでもファイルが保存されたままである限りはそのまま使用中として計算されます。

ビルド時間

ビルド時間にかかる料金は、ビルドに使用する OS によって変わります。Linux で 1 分ビルドしたときはそのまま 1 分として計算されますが、Windows は 2 倍、macOS は 10 倍の実行時間として計算されます。無料分のビルド時間を超えると、ビルド 1 分につき $0.008 のレートで料金がかかります。ちなみに、セルフホストランナーを利用する場合は無料です。

▼表 1.2　ビルド時間でかかる料金

OS	料金
Linux	$0.008 / 分
Windows	$0.016 / 分
macOS	$0.08 / 分

表 1.2 が OS ごとのビルド時間ごとにかかる費用のまとめです。例えば、1,000 分

ビルドすると、Linux だと $8、Windows だと $16, macOS だと $80 になります。

ストレージ

ストレージ料金は、GitHub Actions のアーティファクト機能[*10]で利用している容量と、GitHub Packages[*11] で利用している容量の合計で計算されます。GitHub Packages 側の料金体系については本書では扱わないので、興味がある場合は GitHub Packages の公式ドキュメント[*12]を参照してください。

GitHub Actions では、無料分のストレージを超えた容量に対して、1 GB につき $0.25 が毎月かかります。月の途中で新しくストレージに追加された場合、この料金は時間単位で計算されます。例えば、合計日数が 30 日の月に、12 GB の容量を 10 日間保存したとすると、次の計算になります。

- (12 GB × 10 日 × 24 時間) / (30 日 × 24 時間) = 4 GB 月
- 4 GB 月 × $0.25 = $1

料金の上限

意図しない理由により料金がかかりすぎるのを防げるように、料金には上限が設定できるようになっています。この設定は、GitHub Actions と Packages を合計した月額料金の上限です。デフォルトでは、$0 が上限として設定されているので、無料分を超えて利用したい場合は料金の上限を変更する必要があります[*13]。

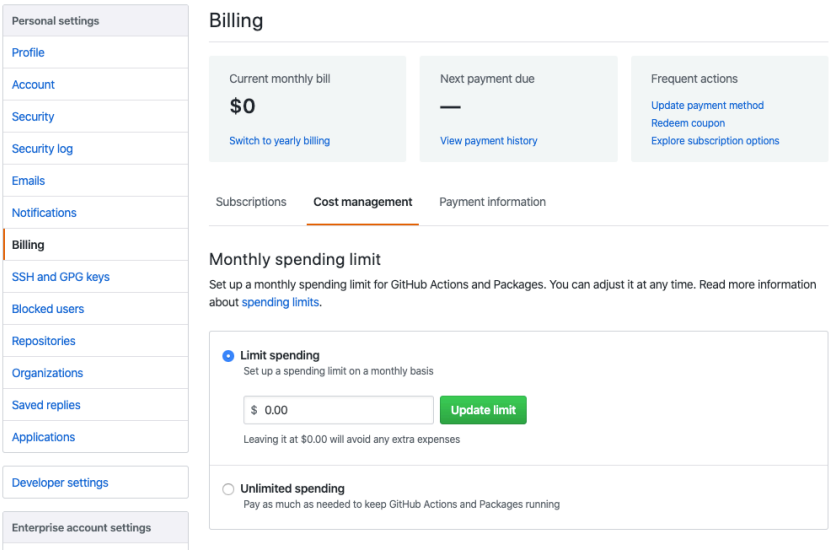
料金の上限の設定方法は、個人アカウントか organization か Enterprise アカウントかで変わってきます。参考までに個人アカウントでの設定方法を書きますと、GitHub をブラウザで開き、右上のヘッダーメニューから Settings → サイドバーの Billing → Cost management という順で遷移すると、料金の上限を設定できます。

[*10] 「2.6 アーティファクト」を参照してください

[*11] GitHub Packages はソフトウェアのパッケージをホスティングするためのサービスです。https://help.github.com/github/managing-packages-with-github-packages/about-github-packages

[*12] https://help.github.com/github/setting-up-and-managing-billing-and-payments-on-github/about-billing-for-github-packages

[*13] もちろん、事前にクレジットカードなどの支払い方法を設定する必要があります



▲図 1.1　料金の上限の設定変更画面

図 1.1 は筆者の個人アカウントの料金の上限の設定画面です。

制限事項

GitHub のレガシーな料金プラン（ユーザー数が無制限でプライベートリポジトリ数で月額料金が決まるプラン）では、GitHub Actions は利用できません。新しいプランにアップグレードする必要があります。

また、Enterprise アカウントで支払いを請求書払いにしている場合、料金の上限をブラウザ上から設定できません。GitHub に問い合わせて、事前に利用分を購入する必要があるようです。

1.4　Hello, World!

詳細な機能説明に入る前に、実際に GitHub Actions でワークフローを設定して動かす流れを体験してみましょう。まず、個人アカウント（もしくは organization）下に新しくリポジトリを作成し、次の内容で `.github/workflows/hello.yml` ファイルを作成します。

▼リスト 1.1　.github/workflows/hello.yml

```
name: Hello, World!
on: push

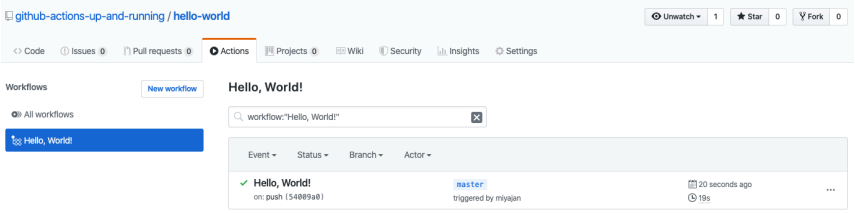
jobs:
  build:
    name: Greeting
    runs-on: ubuntu-latest
    steps:
      - run: echo "Hello, World!"
```

リスト 1.1 は、ワークフローを定義しています。

最初の name は、ワークフローの名前です。次の on:　push で、リポジトリへの push イベント時にこのワークフローが実行されるように定義しています。jobs ブロックでは、ワークフロー内で実行されるジョブを定義しています。build と書かれているところはジョブの ID で、jobs のキーとしてユニークであれば他の文字列でも大丈夫です。name でジョブ名、runs-on でこのジョブが実行される仮想環境を指定しています。steps でジョブ内で実行される処理を定義しています。今回のサンプルでは、echo が実行されるだけです。このあたりの設定の詳細については、第 2 章「GitHub Actions の機能解説」で解説します。

```
$ git add .github/workflows/hello.yml
$ git commit -m "first commit"
$ git push origin master
```

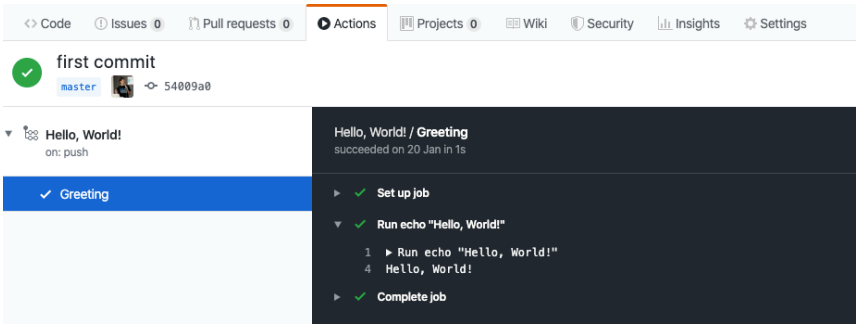
上記のように作成したファイルを push すると、GitHub Actions のワークフローが実行されます。GitHub のリポジトリをブラウザで開き、Actions タブをクリックすると図 1.2 のようにワークフローが実行されている様子を見られます。



▲図 1.2　Actions タブ

表示されている最新の行をクリックすると、ワークフローの実行ログが確認できる

ページに遷移します。各ステップが表示され、Run ステップをクリックすると、図 1.3 のように標準出力に Hello, World! と表示されていることが確認できます。



▲図 1.3　ログ

このサンプルは、次の GitHub リポジトリでご確認いただけます。

- https://github.com/github-actions-up-and-running/hello-world

以上で、リポジトリに push されるたびに Hello, World! と表示するだけの簡単なワークフローを GitHub Actions で作成できました。まだ本格的な CI/CD ではないですが、GitHub Actions でワークフローを作成する流れを体験していただけたと思います。

1.5　まとめ

本章では、GitHub Actions はどういったものか、どういった特徴があるのかといった話から、実際に簡単なワークフローを動かすところまで説明しました。

次章では、ワークフローの設定方法や GitHub Actions が持つ機能についてより詳しく見ていきます。

第 2 章

GitHub Actions の機能解説

本章では、ワークフローの設定方法を中心に、GitHub Actions が持つ様々な機能について説明します。GitHub Actions の入門書である本書において、コアとなる章です。

2.1 ワークフローとは

ここまでもワークフローという言葉を使ってきましたが、あらためて GitHub Actions におけるワークフローとその構成要素について説明します。

ワークフローは、リポジトリ内のソフトフェア開発におけるなにかしらのプロセスを自動化したものです。よくある例は、ソフトウェアのビルド、テスト、パッケージ作成、デプロイを自動化した CI/CD の役割としてのワークフローです。しかし、ワークフローは必ずしも CI/CD だけではなく、例えば issue やプルリクエストが新たに作成されたときに担当者を割り振る作業を自動化したものもワークフローの一つです。1 つのリポジトリに対して、複数のワークフローを作成できます。

ワークフローは、少なくとも一つ以上の複数のジョブから構成されます。それぞれのジョブは、ワークフロー内のなにかしらのタスクを実行します。ジョブ同士は並列に実行することもできますし、ジョブ A が成功したらジョブ B を実行するように依存関係を持たせることもできます。

ジョブは、複数のステップから構成されます。**ステップ**は、なにかしらのコマンドの実行か、なにかしらのアクションの呼び出しを行います。

アクションは、GitHub Actions が提供する、なにかしらの処理の固まりを表す単位です。公開したり共有したりすることが可能で、ステップの中で呼び出されます。アクションは自作することもできますし、コミュニティで共有されたアクションを利用することも可能です。

2.2 ワークフローファイルの保存場所

ワークフローの設定ファイルは、リポジトリのルートに `.github/workflows` という名前でディレクトリを作成し、その直下に保存する必要があります。

設定ファイルは、`.yml` か `.yaml` という拡張子で保存する必要があります。設定ファイルは YAML 形式で記述します。設定ファイルの構文の詳細については、「2.4 ワークフロー設定ファイルの構文」で解説します。「1.4 Hello, World!」では、設定ファイルの名前を `hello.yml` として保存していましたが、拡張子部分以外のファイル名は自由です。また、設定ファイルを複数作成することも可能です。なので、複数のワークフローファイルを作成する場合は、それぞれのファイルがどのような役割のワークフローを設定しているのかわかりやすい名前を付けることをおすすめします。

ワークフローの設定ファイルのディレクトリ構造は、リスト 2.1 のようになります。

▼リスト 2.1　ワークフローの設定ファイルのディレクトリ構造

```
(repository root)
|-- .github
|   `-- workflows
|       |-- continuous-delivery.yml
|       |-- issue-management.yml
|       ...
...
```

2.3 ワークフローが実行される仮想環境

GitHub Actions がワークフローを実行する環境について説明します。ワークフローの実行環境には、GitHub が提供する VM（仮想マシン）環境と、ユーザーが構築した独自の環境が使えます。ユーザー独自の環境については「2.12 ランナーのセルフホスティング」で解説するので、この節では GitHub が提供する環境について解説します。

GitHub が提供する VM 環境は、ジョブの実行ごとに毎回クリーンな環境が提供されます。一つのジョブ内のすべてのステップは同じ VM 内で実行されるため、ステップ間ではファイルシステム経由で情報のやりとりができます。しかし、異なるジョブ間では別の VM が使われるため、ジョブ間で情報をやりとりするためには「2.6 アーティファクト」で後述するアーティファクトなど、別の方法が必要になります。

2.3.1 OS とリソース

GitHub が提供する環境の OS には、Linux、Windows、macOS の三種類が存在します。環境のメンテナンスやアップグレードは GitHub 側でやってくれるため、ユーザー側はその環境を利用するだけです。

Linux と Windows の環境は、Microsoft Azure の Standard_DS2_v2[*1] サイズの VM です。次のスペックでワークフローが実行されます。

- vCPU: 2
- メモリ: 7 GiB
- SSD: 14 GiB

macOS の環境は、MacStadium[*2] という Mac の IaaS(infrastructure as a service) が使われています。

Linux と macOS 環境ではパスワードなしで `sudo` コマンドを実行できます。Windows 環境では UAC(User Account Control) を無効にした状態で管理者として実行されます。なので、VM に新しいパッケージのインストールをするなど、管理者権限が必要な操作もワークフロー内で実行可能です。

[*1] https://docs.microsoft.com/ja-jp/azure/virtual-machines/windows/sizes-general#dsv2-series

[*2] https://www.macstadium.com/

仮想環境の IP レンジ

Azure は IP レンジを JSON ファイルとして提供しており、次の URL からダウンロードできます。

- `https://www.microsoft.com/en-us/download/details.aspx?id=56519`

なので、Linux と Windows については、例えばプライベートネットワーク内の特定リソースへのアクセスを IP レンジで制限をかけて許可するということが可能とも考えられます。しかし、この IP レンジは GitHub Actions に限られたものではありません。それ以外の Azure 上のマシンからアクセスされる危険性を考えると、この方法は使うべきではないでしょう。

プライベートネットワーク内のリソースにアクセスさせたい場合は、「2.12 ランナーのセルフホスティング」で説明するセルフホストランナーの仕組みを使って、ネットワーク内に実行環境を構築するなどの手段を取るほうが安全です。

2.3.2 インストールされているソフトウェア

GitHub が提供する環境で、それぞれの OS の VM に最初からインストールされているソフトウェアの一覧は、公式ドキュメント上の次の URL で公開されています。インストールされているソフトウェアは常に固定ではなく、たびたび更新されています。

- https://help.github.com/en/actions/reference/software-installed-on-github-hosted-runners

開発に必要なコマンドラインツールや主要なプログラミング言語、AWS や Azure や GCP の CLI など、様々なソフトウェアが使えるようになっています。なので、ユースケースによっては新しいソフトウェアをインストールする必要がないと思います。しかし、インストールされていないソフトウェアを使いたい場合や、特定のバージョンに固定してソフトウェアを利用したい場合は、ワークフロー内でインストールするか、Docker を使うなどの方法をとる必要があります。

2.4 ワークフロー設定ファイルの構文

本節では、ワークフローを設定する YAML ファイルの構文を説明します。

2.4.1 サンプルリポジトリの作成

ワークフロー設定の前に、解説しやすくするために簡単な Node.js + npm のサンプルリポジトリを作成します。今後、このサンプルリポジトリに対してワークフローを設定しながら説明をしていきます。次の URL で、この章が終わった時点でのサンプルリポジトリの見本を公開しています。

- `https://github.com/github-actions-up-and-running/fizz-buzz`

サンプルリポジトリの中身を知らなくても説明は理解できるようになっているので、手元で動かさないという方は、「2.4.2 ワークフローとジョブの設定」まで読み飛ばしていただいても大丈夫です。ここでは、簡単な FizzBuzz プログラム[*3]を作成して、ユニットテストと lint による静的解析を実行できるようにします。

まず、個人アカウント（もしくは organization）下に新しくリポジトリを作成し、リスト 2.2 の内容で `package.json` ファイルを作成します。`package.json` は npm のメタデータを設定するファイルです。

▼リスト 2.2　package.json

```
{
  "name": "fizz-buzz",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "lint": "eslint .",
    "test": "mocha"
  },
  "license": "MIT"
}
```

次に、リスト 2.3 の内容で `index.js` ファイルを作成します。このファイルは FizzBuzz プログラムの本体です。

[*3] 引数を受け取り、その値が 3 の倍数だったら Fizz、5 の倍数だったら Buzz、3 と 5 の倍数だったら FizzBuzz、それ以外だったらその数字を返す

▼リスト 2.3　index.js

```
module.exports = function fizzbuzz(value) {
  if (value % 15 === 0) {
    return "FizzBuzz";
  }
  if (value % 3 === 0) {
    return "Fizz";
  }
  if (value % 5 === 0) {
    return "Buzz";
  }
  return String(value);
};
```

今回は、テストランナーとして Mocha[*4] というソフトウェアを利用するので、次のように `npm install` コマンドで依存パッケージをインストールします。

```
$ npm install --save-dev mocha
```

そして、リスト 2.4 の内容で `test/index.js` ファイルを作成します。このファイルは、FizzBuzz プログラムのユニットテストです。

▼リスト 2.4　test/index.js

```
const assert = require("assert");
const fizzbuzz = require("../index");

describe("fizzbuzz", () => {
  it("returns FizzBuzz when value is divisible by 15", () => {
    assert(fizzbuzz(30) === "FizzBuzz");
  });

  it("returns Fizz when value is divisible by 3", () => {
    assert(fizzbuzz(9) === "Fizz");
  });

  it("returns Buzz when value is divisible by 5", () => {
    assert(fizzbuzz(10) === "Buzz");
  });

  it("returns the value when value is not divisible by 3 or 5", () => {
    assert(fizzbuzz(7) === "7");
  });
});
```

[*4] https://mochajs.org/

この時点で npm test コマンドを実行すると、次のような結果になるはずです。

```
$ npm test

...


  fizzbuzz
    ✓ returns FizzBuzz when value is divisible by 15
    ✓ returns Fizz when value is divisible by 3
    ✓ returns Buzz when value is divisible by 5
    ✓ returns the value when value is not divisible by 3 or 5


  4 passing (5ms)
```

すべてのテストケースが通っていることが確認できます。

次に、ESLint[*5] で静的解析できるようにするために、次のように npm install コマンドで依存パッケージをインストールします。

```
$ npm install --save-dev eslint @cybozu/eslint-config prettier
```

ESLint の他にも 2 つのパッケージをインストールしています。@cybozu/eslint-config[*6] は、サイボウズ株式会社の開発で使われている ESLint のルールセットで、使い勝手がいいので入れています。Prettier は、JavaScript 界隈でよく使われているコードフォーマッターです。

そして、リスト 2.5 の内容で .eslintrc.js ファイルを作成します。

▼リスト 2.5　.eslintrc.js

```
1: module.exports = {
2:   extends: "@cybozu/eslint-config/presets/node-prettier",
3:   env: {
4:     mocha: true
5:   }
6: };
```

2 行目は、Node.js 向けのルールと Prettier のコードフォーマットを追加するルー

[*5] https://eslint.org/

[*6] https://github.com/cybozu/eslint-config

ルセットを適用しています。3 〜 5 行目は、Mocha で追加されるグローバル変数がエラーにならないように追加しています。

この時点で npm run lint コマンドを実行すると、次のような結果になるはずです。

```
$ npm run lint

> fizz-buzz@1.0.0 lint ...
> eslint .
```

メッセージがわかりにくいですが、エラーメッセージが出力されておらず、成功しています。

最後に、リスト 2.6 の内容で .gitignore ファイルを作成します。

▼リスト 2.6　.gitignore

```
node_modules
```

これは、npm install でダウンロードされる依存パッケージをリポジトリに含めないためです。

ここまでできたら、次のコマンドで作成したファイルをコミットしてリポジトリに push します。

```
$ git add .
$ git commit -m "first commit"
$ git push origin master
```

現時点で、リポジトリのディレクトリ構造はリスト 2.7 のようになるはずです。

▼リスト 2.7　サンプルリポジトリのディレクトリ構造

```
(repository root)
|-- README.md
|-- index.js
|-- package-lock.json
|-- package.json
`-- test
    `-- index.js
```

これでサンプルリポジトリが完成したので、ここから本題のワークフローの設定を見ていきましょう。

2.4.2 ワークフローとジョブの設定

サンプルリポジトリに対して、コードの変更ごとにユニットテストと lint を実行するワークフローを設定していきます。まず、リスト 2.8 の内容で `.github/workflows/ci.yml` ファイルを作成します。

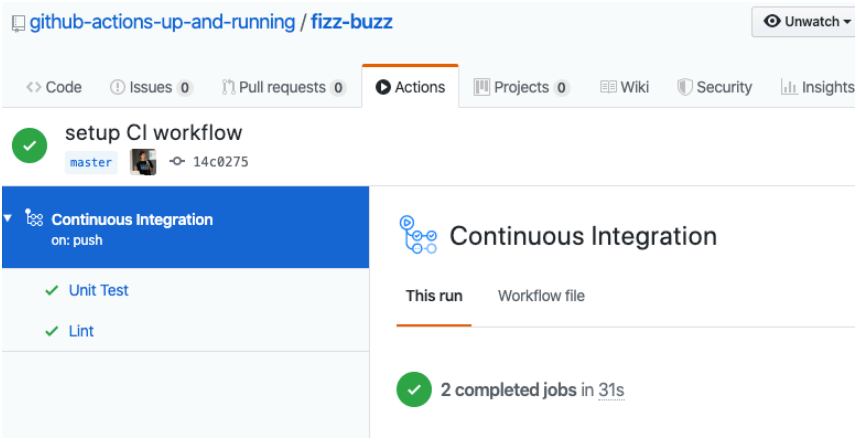
▼リスト 2.8　.github/workflows/ci.yml

```
 1: name: Continuous Integration
 2: on: push
 3:
 4: jobs:
 5:   unit-test:
 6:     name: Unit Test
 7:     runs-on: ubuntu-latest
 8:     steps:
 9:       - name: Checkout
10:         uses: actions/checkout@v2.0.0
11:       - name: Set Node.js 12.x
12:         uses: actions/setup-node@v1.3.0
13:         with:
14:           node-version: 12.x
15:       - name: Install dependencies
16:         run: npm ci
17:       - name: Test
18:         run: npm test
19:   lint:
20:     name: Lint
21:     runs-on: ubuntu-latest
22:     steps:
23:       - name: Checkout
24:         uses: actions/checkout@v2.0.0
25:       - name: Set Node.js 12.x
26:         uses: actions/setup-node@v1.3.0
27:         with:
28:           node-version: 12.x
29:       - name: Install dependencies
30:         run: npm ci
31:       - name: Lint
32:         run: npm run lint
```

そして、次のコマンドで作成したファイルをコミットしてリポジトリに `push` します。

```
$ git add .github/workflows/ci.yml
$ git commit -m "setup CI workflow"
$ git push origin master
```

リポジトリの Actions タブを開いて、図 2.1 のようにワークフローが問題なく動作していれば成功です。



▲図 2.1　リポジトリの Actions タブからワークフローを確認

ここからは、この設定ファイルで使われている構文を中心に、ワークフローの構文を解説していきます。

name

1 行目の `name` は、ワークフローの名前を表します。ブラウザ上でリポジトリの Actions タブを開いたときに表示されます。`name` を省略したときは、ワークフローファイルへの相対パス（今回だと `.github/workflows/ci.yml`）が代わりにワークフロー名となります。

on

2 行目の `on` は、ワークフローを実行するイベント名を表します。必須です。今回だと、`push` を指定しているので、リポジトリにコミットが `push` されるたびにワークフローが実行されます。指定できるイベントについては「2.7 イベントとアクティビティ」で解説します。文字列で一つのイベントのみ指定することもできますし、リストで複数のイベントを指定することもできます。例えば、リスト 2.9 のようにリストで指定すると、`push` と `pull_request` の両方のイベントでワークフローが実行されるようになります。

▼リスト 2.9　複数イベントの指定

```
on: [push, pull_request]
```

マップで記述するケースもあり、それについては「2.4.6 イベントのフィルタ」で解説します。

jobs

4 行目の jobs から、ジョブの定義が始まります。必須です。

jobs.<job_id>

5, 19 行目の jobs のマップのキー（<job_id> とします）で、ジョブの ID を定義します。ジョブの ID 同士は重複してはダメです。アルファベットか _ で始まる文字列で、使える文字列はアルファベット、数字、-、_ のみです。

jobs.<job_id>.name

6, 20 行目の name は、ジョブの名前を定義します。ブラウザ上で Actions タブを開いたときに表示されます。省略すると、ジョブの ID が表示されます。

jobs.<job_id>.runs-on

7, 21 行目の runs-on は、ジョブが実行されるマシンを定義します。必須です。

▼表 2.1　runs-on で指定できるマシン環境

runs-on	マシン環境
ubuntu-latest, ubuntu-18.04	Ubuntu 18.04
ubuntu-16.04	Ubuntu 16.04
windows-latest, windows-2019	Windows Server 2019
windows-2016	Windows Server 2016
macos-latest, macos-10.15	macOS Catalina 10.15
self-hosted	セルフホストランナー

表 2.1 は、runs-on で指定できるマシン環境の一覧です。今回のサンプルだと、ubuntu-latest を指定しているので、Ubuntu 18.04 環境でジョブが実行されます。セルフホストランナーについては、「2.12 ランナーのセルフホスティング」で解説します。

jobs.<job_id>.steps

8, 22 行目の `steps` から、ジョブが実行するステップの定義が始まります。必須です。それぞれのステップは、それぞれ異なるプロセスで実行されます。なので、ステップ内で環境変数を変更しても、後のステップには変更が反映されません。後のステップに環境変数の変更を反映する方法は、「2.8.1 環境変数の設定: `set-env`」を参照してください。

jobs.<job_id>.steps.name

9 行目を含む各ステップの `name` は、ステップの名前を定義します。ブラウザ上で表示されます。省略したときは、ステップで実行するコマンド（アクションの場合はアクション名）が表示されます。

jobs.<job_id>.steps.uses

10, 12, 24, 26 行目の `uses` は、アクションの実行を定義します。サンプルでは、パブリックリポジトリのアクションを実行しています。10, 24 行目では、次のリポジトリの v2.0.0 タグのアクションを実行します。

- https://github.com/actions/checkout

12, 26 行目では、次のリポジトリの v1.3.0 タグのアクションを実行します。

- https://github.com/actions/setup-node

`@` 以降のバージョン指定は必須です。タグだけでなく、リスト 2.10 のような形式でアクションのバージョンを指定できます。

▼リスト 2.10　アクションのバージョンの指定方法

```
# 特定のコミット SHA
- uses: actions/checkout@722adc63f1aa60a57ec37892e133b1d319cae598
# 特定のタグ
- uses: actions/checkout@v2.0.0
# 特定のブランチ
- uses: actions/checkout@master
```

アクションのバージョンをどの方法で指定するべきか

前述したように、アクションのバージョンの指定方法には、コミット SHA、タグ、ブランチの 3 種類の方法が存在します。では、どの方法で指定するのがいいのでしょうか。

まず、ブランチ指定ですが、これはデバッグ用途以外ではおすすめしません。例えば、master を指定することで常に最新バージョンのアクションが使えると考えるかもしれません。ですが、後方互換性のない変更が入った時点でワークフローが壊れてしまい、開発が止まってしまう危険があります。

次に、タグ指定です。こちらはアクション公開側がセマンティックバージョニング[*7]を意識していれば、後方互換性を崩さないようにバージョンを指定できます。例えば、`actions/checkout@v2` のようにメジャーバージョンのみ指定すれば、後方互換性を壊さずに機能追加や不具合回収の恩恵を受けることができます。しかし、この場合もアクションの開発者が意図しない不具合が入る可能性は残ります。また、タグを上書きできる以上は、正常だったアクションにあるとき急に悪意のあるコードが仕込まれる可能性もゼロとは言えないです。なので、GitHub 公式のような、よほど信頼できるアクション以外はタグを使った指定もあまりおすすめしません。

最後に、コミット SHA 指定です。コミット SHA で指定すればあとから上書きされる心配がないので、一度動いていたものが動かなくなることはないですし、セキュリティ面でも一番安全です。しかし、人の可読性の面で劣るので、コメントでわかりやすく補足してあげるとメンテナンスしやすいと思います。例として、リスト 2.11 では、`v2.0.0` タグのコミット SHA を指定していることがわかるようにしています。

▼リスト 2.11　コメントでコミット SHA を補足する

```
# v2.0.0
- uses: actions/checkout@722adc63f1aa60a57ec37892e133b1d319cae598
```

実行するアクションの指定方法は、サンプルで行っている方法以外にもあります。リポジトリ直下でなくサブディレクトリにアクションが配置されている場合、リスト 2.12 のようにサブディレクトリを指定することができます。

[*7] https://semver.org/

▼リスト 2.12　サブディレクトリのアクションを実行

```
- uses: GoogleCloudPlatform/github-actions/setup-gcloud@master
```

外部のリポジトリだけでなく、ワークフロー実行中のリポジトリ内にあるアクションを相対パスで指定することもできます。この場合はバージョン指定は必要ありません。

▼リスト 2.13　リポジトリ内のアクションを実行

```
- uses: ./.github/actions/your-action
```

Docker レジストリにあるイメージをアクションとして実行することもできます。この場合は、`@` ではなく `:` でイメージのタグを指定します。リスト 2.14 のように書くと、Docker Hub 上のイメージをアクションとして実行できます。

▼リスト 2.14　Docker Hub 上のイメージをアクションとして実行

```
- uses: docker://alpine:3.11
```

Docker Hub だけでなく、それ以外のパブリックな Docker レジストリのイメージもアクションとして実行できます。リスト 2.15 では、GCP の Cloud Build が提供している gcloud イメージをアクションとして実行しています。（`with` については後述します）

▼リスト 2.15　パブリックな Docker レジストリ上のイメージを指定

```
- uses: docker://gcr.io/cloud-builders/gcloud
  with:
    args: version
```

アクションの作り方については、第 3 章「アクション」で解説します。

jobs.<job_id>.steps.with

13, 27 行目の `with` はアクションのパラメータを定義します。マップのキーがパラメータ名、バリューが値です。サンプルだと、`actions/setup-node` アクションの `node-version` パラメータに `12.x` という値を渡しています。渡せるパラメータはアクション側のメタデータで定義されています。

パラメータは、パラメータ名を大文字にして先頭に INPUT_ を付けた環境変数としてアクションに渡されます。例えば、node-version だとしたら、INPUT_NODE-VERSION という環境変数に値が設定されます。

アクション側がパラメータをどのように定義するかについての詳細は、「3.3 アクションのメタデータ」で解説します。

jobs.<job_id>.steps.with.entrypoint

Docker コンテナアクションを実行する場合、entrypoint と args という二つのパラメータを渡せます。

entrypoint は、Dockerfile の ENTRYPOINT を上書きします。Dockerfile の ENTRYPOINT には実行するコマンドとその引数を記述できるのに対して、ワークフローの entrypoint では実行するコマンドしか渡せません。

jobs.<job_id>.steps.with.args

args は、コンテナアクションの実行時に ENTRYPOINT に渡される引数を文字列で定義します。

例えば、リスト 2.16 のようにすると、alpine イメージのコンテナで /bin/echo コマンドに Hello, World! を引数として渡して実行します。

▼リスト 2.16　Docker コンテナアクションのパラメータ

```
- uses: docker://alpine:3.11
  with:
    entrypoint: /bin/echo
    args: Hello, World!
```

jobs.<job_id>.steps.run

16, 18, 30, 32 行目の run は、コマンドラインの実行を定義します。16, 30 行目では npm ci コマンドで依存関係のインストール、18 行目では npm test でユニットテストの実行、30 行目では npm run lint で eslint の実行をそれぞれ行っています。

サンプルでは一つの run で一行のコマンドしか実行していませんが、リスト 2.17 のように書くことで複数行のコマンドを記述して実行することも可能です。

▼リスト 2.17　複数行のコマンド実行

```
- run: |
    npm ci
    npm test
```

jobs.<job_id>.steps.working-directory

run 実行時に working-directory を指定することでコマンド実行のワーキングディレクトリを変更できます。リスト 2.18 ではワーキングディレクトリを /tmp に変更してコマンドを実行しています。

▼リスト 2.18　ワーキングディレクトリを /tmp に変更してコマンド実行

```
- run: |
    pwd
    ls
  working-directory: /tmp
```

jobs.<job_id>.steps.shell

run 実行時に shell を指定することでコマンド実行時に使用されるシェルを変更できます。

▼表 2.2　コマンド実行時のシェルを変更する

shell	利用可能な OS	内部的に実行されるコマンド
bash	すべて	bash --noprofile --norc -eo pipefail {0}
pwsh	すべて	pwsh -command "& '{0}'"
python	すべて	python {0}
sh	Linux, macOS	sh -e {0}
cmd	Windows	%ComSpec% /D /E:ON /V:OFF /S /C "CALL "{0}""
powershell	Windows	powershell -command "& '{0}'"

GitHub が組み込みで用意しているシェルは 表 2.2 にまとめました。

bash は、Windows 以外でデフォルトのシェルで、Windows で指定すると Git for Windows のシェルが使われます。pwsh は、PowerShell Core[*8] です。python は、Python スクリプトを記述できます。sh は、Windows 以外のプラットフォーム

[*8] https://github.com/PowerShell/PowerShell

で bash がパスに存在しないときにデフォルトで使われます。cmd は、Windows のコマンドプロンプト（cmd.exe）です。powershell は、Windows でデフォルトのシェルです。{0} のところには、run にコマンドが書かれたファイルの名前が入ります。

例えば、Python スクリプトを実行するにはリスト 2.19 のように書きます。

▼リスト 2.19　Python スクリプトを実行

```
- run: |
    import sys
    print(sys.version)
  shell: python
```

また、表 2.2 に用意されたシェルだけでなく、shell に自前でコマンドを指定することもできます。リスト 2.20 では、bash -eux でコマンドを実行します。

▼リスト 2.20　独自のシェルコマンド

```
- run: echo "Hello, World!"
  shell: bash -eux {0}
```

shell で python を指定したときの Python のバージョン

shell で python を指定すると、run に直接 Python スクリプトを記述できます。しかし、Python のバージョンが OS によって異なるので注意が必要です。筆者が確認した時点では、デフォルトで次のバージョンの Python が使われるようになっていました。

- windows-latest, windows-2019: 3.7.6
- windows-2016: 3.7.6
- ubuntu-latest, ubuntu-18.04: 2.7.17
- ubuntu-16.04: 2.7.12
- macos-latest, macos-10.15: 2.7.17

事前に setup-python[*9] という特定のバージョンの Python を実行環境に設定するアクションを実行すると、そちらのバージョンが反映されます。

jobs.<job_id>.timeout-minutes

ジョブに timeout-minutes を指定することで、ジョブのタイムアウト時間を定義できます。指定しないときは、デフォルトで 360 分になります。リスト 2.21 のように書くと、unit-test ジョブのタイムアウト時間が 30 分に変更されます。

▼リスト 2.21　ジョブのタイムアウト時間を設定

```
jobs:
  unit-test:
    timeout-minutes: 30
    ...
```

jobs.<job_id>.steps.timeout-minutes

ステップに timeout-minutes を指定することで、ステップのタイムアウト時間を定義できます。デフォルトではステップのタイムアウト時間はなく、ジョブのタイムアウト時間が実質的なタイムアウト時間になります。リスト 2.22 のように書くと、ステップのタイムアウト時間が 1 分に変更されます。

▼リスト 2.22　ステップのタイムアウト時間を設定

```
steps:
  - run: sleep 120
    timeout-minutes: 1
```

jobs.<job_id>.continue-on-error

ジョブの continue-on-error に true を指定することで、そのジョブが失敗してもワークフローが失敗にならなくなります。リスト 2.23 では、exit 1 を実行してジョブは失敗となりますが、ワークフローは成功となります。

▼リスト 2.23　必ず成功するジョブ

```
jobs:
  always-success:
    runs-on: ubuntu-latest
    continue-on-error: true
    steps:
      - run: exit 1
```

[*9] https://github.com/actions/setup-python

jobs.<job_id>.steps.continue-on-error

ステップの continue-on-error に true を指定することで、ステップの終了コードに関わらずステップが成功として扱われます。リスト 2.24 では、exit 1 を実行していますが、ステップは成功となります。

▼リスト 2.24　必ず成功するステップ

```
steps:
  - run: exit 1
    continue-on-error: true
```

jobs.<job_id>.steps.id

ステップに id を設定することで、${{ steps.<step_id>.outputs.<output_name> }} のようにコンテキスト経由でステップのアウトプットにアクセスできます。コンテキストについては、「2.4.11 コンテキストと式を使った条件実行」で解説します。アウトプットの設定方法については、「2.8.2 アウトプットの設定: set-output」で解説します。

2.4.3　デフォルト設定

ワークフロー全体や、特定のジョブ全体にデフォルトで適用される設定について説明します。

defaults

ワークフローのトップレベルで defaults を指定すると、ワークフロー内のすべてのジョブにデフォルトで適用される設定を定義できます。

defaults.run

defaults で run を設定することで、ワークフロー内のすべての run ステップにデフォルトで適用される設定を定義できます。shell と working-directory が指定可能です。リスト 2.25 では、shell に python、working-directory に /tmp をデフォルトとして設定しています。

▼リスト 2.25　ワークフロー内のすべての run にデフォルトで適用される設定

```
defaults:
  run:
    shell: python
    working-directory: /tmp
```

jobs.<job_id>.defaults

ジョブに対して defaults を指定すると、特定のジョブだけにデフォルトで適用される設定を定義できます。もしワークフローとジョブで同じキーのデフォルト設定を指定した場合、ジョブに指定した設定が優先されます。

jobs.<job_id>.defaults.run

ジョブに対して defaults で run を設定することで、特定のジョブ内のすべての run ステップにデフォルトで適用される設定を定義できます。shell と working-directory が指定可能です。リスト 2.26 では、unit-test ジョブにデフォルトの shell と working-directory を設定しています。

▼リスト 2.26　ジョブ内のすべての run にデフォルトで適用される設定

```
jobs:
  unit-test:
    defaults:
      run:
        shell: python
        working-directory: /tmp
```

2.4.4　環境変数

ワークフロー全体や、特定のジョブ、ステップに環境変数を設定したいということはよくあります。この項では、環境変数の設定方法について説明します。

env

ワークフローのトップレベルで env を指定すると、ワークフロー内のすべてのジョブ、ステップに適用される環境変数を定義できます。マップで記述し、キーが環境変数名、バリューが値です。リスト 2.27 では、DEBUG という環境変数に true の値を設定しています。

▼リスト 2.27　ワークフロー全体に環境変数を設定

```
env:
  DEBUG: true
```

jobs.<job_id>.env

ジョブに対して env を指定すると、特定のジョブだけに適用される環境変数を定義できます。リスト 2.28 では、unit-test ジョブへ、DEBUG という環境変数に true の値を設定しています。

▼リスト 2.28　ジョブに環境変数を設定

```
jobs:
  unit-test:
    env:
      DEBUG: true
```

もしワークフローとジョブで同じ名前の環境変数を設定した場合、ジョブに設定した環境変数が優先されます。

jobs.<job_id>.steps.env

ステップに対して env を指定すると、特定のステップだけに適用される環境変数を定義できます。リスト 2.29 では、ステップへ DEBUG という環境変数に true の値を設定しています。

▼リスト 2.29　ステップに環境変数を設定

```
steps:
  - env:
      DEBUG: true
    run: echo ${DEBUG}
```

もしワークフローとジョブとステップで同じ名前の環境変数を設定した場合、ステップに設定した環境変数が優先されます。つまり、より限定的なスコープで設定した環境変数が常に優先されます。

環境変数名の命名規則

ワークフローの設定ファイルで環境変数を設定する場合、GITHUB_ で始まる環境変数名は GitHub が内部的に使用する可能性があるので使ってはいけません[*10]。

また、ディレクトリやファイルへのパスを環境変数として設定する場合は、環境変数名の最後を _PATH にすることが推奨されています。破ってもエラーになりませんが、わかりやすさのためにこの命名規則に従うことをおすすめします。

デフォルトの環境変数

ワークフロー実行時には、表 2.3 の環境変数がデフォルトで設定されます。

[*10] 公式ドキュメントでは GITHUB_ で始まる環境変数を指定するとエラーになると書いてあるのですが、この本の執筆時点ではエラーになりません

▼表 2.3　デフォルトの環境変数

環境変数名	説明
CI	常に true
HOME	ホームディレクトリへのパス
GITHUB_WORKFLOW	ワークフロー名
GITHUB_RUN_ID	ワークフローの実行ごとに設定されるリポジトリ全体でユニークな ID（ワークフローを再実行しても変更されない）
GITHUB_RUN_NUMBER	ワークフローの実行ごとに設定されるそれぞれのワークフローごとにユニークな 1 から始まる番号（ワークフローを再実行しても変更されない）
GITHUB_ACTION	現在実行中のアクションのユニークな ID（同じアクションでも複数回実行された場合は毎回異なる値になる、コマンドライン実行でも設定される）
GITHUB_ACTIONS	常に true
GITHUB_ACTOR	ワークフローを実行した人またはアプリの名前
GITHUB_REPOSITORY	リポジトリ名（<owner>/<repository> の形式）
GITHUB_EVENT_NAME	ワークフローを実行したイベント名
GITHUB_EVENT_PATH	ワークフローを実行したイベントのペイロードが保存されたファイルへのパス
GITHUB_WORKSPACE	ワークスペースディレクトリへのパス
GITHUB_SHA	ワークフローが実行されているコミット SHA
GITHUB_REF	ワークフローが実行されているブランチまたはタグへの参照（refs/heads/master のような形式）
GITHUB_HEAD_REF	ソースブランチ名（pull_request イベントのときのみ設定される）
GITHUB_BASE_REF	ベースブランチ名（pull_request イベントのときのみ設定される）

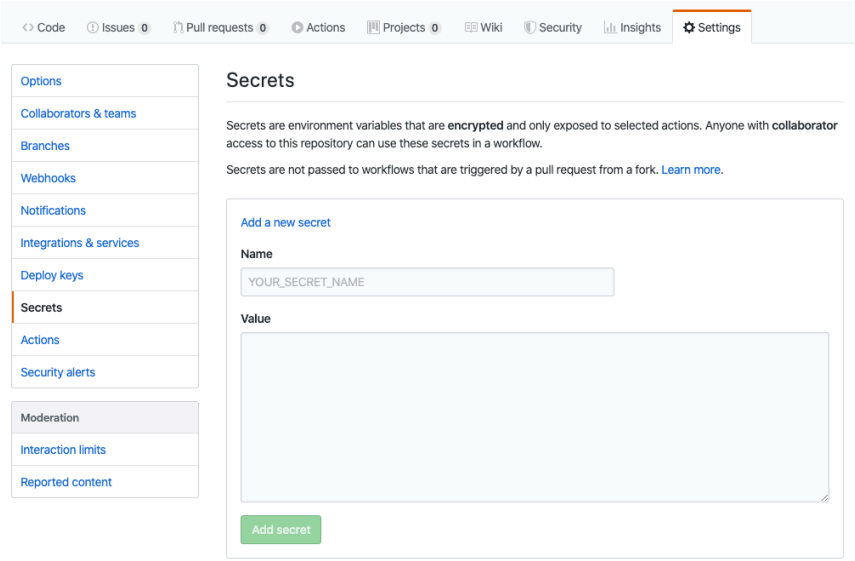
2.4.5　秘密情報

ワークフロー内で秘密情報を扱いたいケースはよくあります。例えば、外部サービスにアクセスする必要があるようなケースでは、そのサービスの認証情報が必要になります。この項では、GitHub Actions で秘密情報を扱う方法について説明します。

秘密情報の登録

GitHub Actions のワークフローで使用する秘密情報を、ユーザーはブラウザ上から登録できます。リポジトリを開き、"Settings" → "Secrets" の順で遷移すると、図

2.2 の画面が表示され、秘密情報が登録できます。



▲図 2.2　秘密情報の登録

リポジトリ単位ではなく、organization 単位でも秘密情報を登録可能です。organization を開き、"Settings" → "Secrets" から秘密情報を登録可能です。organization の秘密情報へのアクセス権限は、次の 3 つから指定できます。

- すべてのリポジトリからアクセス可能
- private リポジトリからのみアクセス可能
- 選択したリポジトリからのみアクセス可能

登録した秘密情報は、GitHub Actions 以外では使われません。登録可能な秘密情報は、最大で 100 個までです。

秘密情報の名前には、アルファベットと _ しか使えません。

秘密情報の値は、ワークフロー実行中にログに出力されてしまったときは、生の文字列として表示されないように、自動でマスクされます。しかし、使う側はできるだけこの仕様はあてにせずに、秘密情報がログに出力されないように気をつけたほうがいいでしょう。また、JSON のような構造化された情報を秘密情報に登録すると、この自動でマスクする仕組みが正常に働かないおそれがあるため、できるだけ複雑な情

報は登録しないことをおすすめします。

これに加えて、秘密情報は最大で 64 KB までしか保存できません。なので、複雑な秘密情報を扱うときは、外部のサービスに保存するか、リポジトリ内に暗号化した情報として保存するなどして、それらを取得・復元するためのトークンやパスワードを秘密情報に登録するような工夫が必要です。

秘密情報をワークフロー内で使用する

秘密情報を使うときは、コンテキスト経由でアクセスします。コンテキストについては、「2.4.11 コンテキストと式を使った条件実行」を参照してください。

仮に、SECRET_TOKEN という名前で秘密情報が登録されているとします。このとき、リスト 2.30 のようにワークフローの設定ファイルに書くことで、秘密情報を参照できます。（ただし、前述したように echo した秘密情報はログ上ではマスクされます）

▼リスト 2.30　秘密情報をワークフロー内で使用

```
steps:
  - run: echo "${{ secrets.SECRET_TOKEN }}"
```

「2.4.2 ワークフローとジョブの設定」で説明した with や、「2.4.4 環境変数」 で説明した env などでもコンテキストにアクセスできるので、アクションにパラメータや環境変数経由で秘密情報を渡すこともできます。

ちなみに、fork されたリポジトリから実行されたワークフローからは秘密情報のコンテキストにアクセスすることはできません。（次に説明する GITHUB_TOKEN は例外です）

GITHUB_TOKEN

GitHub Actions は、ワークフロー実行時にデフォルトで GITHUB_TOKEN という秘密情報を設定します。通常の秘密情報と同様に、${{ secrets.GITHUB_TOKEN }} のようにコンテキスト経由でアクセスできます。

このトークンは、GitHub App のアクセストークンと同様でリポジトリの権限を与えられており、REST API の認証情報として使用することができます。リスト 2.31 では、GITHUB_TOKEN を使ってリポジトリの issue 一覧を取得する REST API を実行しています。

▼リスト 2.31　GITHUB_TOKEN を使って REST API を実行

```
steps:
  - run: |
      curl \
        --request GET \
        --url https://api.github.com/repos/${{ github.repository }}/issues \
        --header 'Authorization: Bearer ${{ secrets.GITHUB_TOKEN }}'
```

このトークンによってリポジトリに対して操作を行った場合、github-actions bot による操作として扱われます。このトークンは、fork されたリポジトリから実行されたワークフローでも使用することができますが、権限が制限されます。表 2.4 が、このトークンが持つ権限一覧です。

▼表 2.4　GITHUB_TOKEN の権限

権限名	権限	fork されたリポジトリから実行されたときの権限
actions	read/write	read
checks	read/write	read
contents	read/write	read
deployments	read/write	read
issues	read/write	read
metadata	read/write	read
packages	read/write	read
pull requests	read/write	read
repository projects	read/write	read
statuses	read/write	read

権限の詳細については、公式のドキュメントを参照してください。

- https://developer.github.com/v3/apps/permissions/

2.4.6　イベントのフィルタ

サンプルリポジトリでは、push イベントが発生したすべてのタイミングでワークフローが実行されるようになっています。しかし、イベント発生時でも特定の条件のときのみワークフローを実行してほしいケースがあります。そういった、イベントに対してワークフローの実行をフィルタする方法について説明します。

ブランチ・タグ

push イベントと pull_request イベントでは、特定のブランチまたはタグに対してイベントが発生したときのみワークフローを実行するようにフィルタできます。ちなみに、pull_request イベントでは、ベースとなるブランチではなく、ソースブランチ側に対してフィルタが適用されます。

例えば、master ブランチに push されたときのみワークフローを実行したい場合は、リスト 2.32 のように書きます。

▼リスト 2.32　master に push されたときのみワークフローを実行

```
on:
  push:
    branches:
      - master
```

ブランチとタグのフィルタに使えるキーワードには、表 2.5 のとおり 4 つあります。

▼表 2.5　ブランチとタグのフィルタ

キーワード	フィルタ
branches	指定したブランチのみ実行
branches-ignore	指定したブランチでは実行しない
tags	指定したタグのみ実行
tags-ignore	指定したタグでは実行しない

すべて指定しなかったときは、すべてのブランチとタグに対してイベントが発生したときワークフローが実行されます。

branches と tags のどちらかしか指定しなかったときは、指定しなかったほうのイベントはすべて除外されます。例えば、push イベントで branches のみ指定した場合は、タグに対しての push はワークフローが実行されなくなります。

branches と branches-ignore は、一つのイベントに対して同時に指定できません。tags と tags-ignore も同様です。なので、ホワイトリスト形式でフィルタしたいか、ブラックリスト形式でリストしたいかで使い分けることになります。

どの方法でフィルタするときでも、次の特殊文字が使えます。

- *: 0 文字以上の / を含まない文字列にマッチ
- **: 0 文字以上のすべての文字列にマッチ
- ?: 0 文字か 1 文字のすべての文字列にマッチ

- +: 1 文字以上のすべての文字列にマッチ
- []: 括弧内の 1 文字にマッチ
  - a-z, A-Z, 0-9 という書き方でのみ範囲指定が可能
  - 例えば、[0-9a-z] と書けば、すべての数字と小文字アルファベットにマッチ
- !: 一文字目に書くと、それ以降の文字列によるパターンを除外する

*, [, ! は YAML においても特殊文字なので、一文字目に使用するときは文字列をクオートする必要があります。

例えば、releases/ で始まるブランチのときだけワークフローを実行し、releases/hoge-testing のように -testing で終わる形式のブランチのときはワークフローを実行しないようにするには、リスト 2.33 のように書きます。

▼リスト 2.33　特殊文字を活用した例

```
on:
  push:
    branches:
      - releases/**
      - '!releases/**-testing'
```

ファイルパス

push イベントと pull_request イベントでは、変更されたファイルのパスによってワークフローの実行をフィルタできます。paths を指定すると、変更されたファイルのうち少なくとも一つのパスが指定されたパスにマッチするときのみワークフローが実行されます。paths-ignore を指定すると、変更されたファイルのうち少なくとも一つのパスが指定されたパスにマッチ**しない**ときのみワークフローが実行されます。使用できる特殊文字は、ブランチやタグのフィルタと一緒です。

例えば、リスト 2.34 のように書くと、任意の JavaScript ファイルが変更されたときにワークフローが実行されます。

▼リスト 2.34　任意の JavaScript ファイルが変更されたときのみ実行する

```
on:
  push:
    paths:
      - '**.js'
```

docs ディレクトリ下のファイルだけが変更されたときはワークフローを実行しないようにするには、リスト 2.35 のように書きます。

▼リスト 2.35　docs ディレクトリ下のファイルだけが変更されたときは実行しない

```
on:
  push:
    paths-ignore:
      - 'docs/**'
```

paths か paths-ignore が指定されていると、ファイルが一つも変更されなかったときはワークフローが実行されません。
変更されたファイル一覧は、次のように計算されます。

- pull_request: ベースとなるブランチとソースブランチが分岐するコミットから、ソースブランチの最新コミットまでの差分
- 既存ブランチへの push: push 前の HEAD からの差分
- 新規ブランチへの push: push されたコミットの祖先ですでに push されてた最新のコミットからの差分

差分に 1,000 コミット以上存在するときや、差分の計算に時間がかかりすぎるときは、指定した条件に関わらずワークフローが実行されてしまうらしいです。

特定の種類のアクティビティのみワークフローを実行

イベントによっては、複数の種類のアクティビティが存在する場合があります。例えば、issue_comment イベントであれば、created, edited, deleted という 3 種類のアクティビティが存在します。types を指定すると、特定のアクティビティが発生したときのみワークフローが実行されるようにフィルタできます。types は文字列かリストで指定できます。リスト 2.36 のように書くと、コメントが作成・編集されたときのみワークフローを実行します。

▼リスト 2.36　コメントが作成・編集されたときのみ実行

```
on:
  issue_comment:
    types: [created, edited]
```

イベントとアクティビティの組み合わせについては、「2.7 イベントとアクティビティ」 を参照してください。

2.4.7　ジョブ間の依存関係の制御

デフォルトでは、jobs で定義したジョブはすべて並列で実行されます。サンプルリポジトリでは、unit-test ジョブと lint ジョブは並列で実行されました。

しかし、ワークフローでは特定のジョブが成功した後に新たなジョブを実行したいという場合があります。例えば、テストや lint のジョブが成功したらデプロイしたい、という場合です。

jobs.<job_id>.needs

needs を指定すると、そのジョブの実行前に成功していないといけないジョブを定義できます。文字列またはリストで指定できます。

▼リスト 2.37　ジョブの依存関係の定義

```
jobs:
  unit-test:
    ...
  lint:
    ...
  deploy:
    needs: [unit-test, lint]
    ...
```

リスト 2.37 のように書くと、unit-test ジョブと lint ジョブが両方とも成功したら、deploy ジョブの実行が開始されます。unit-test か lint ジョブのどちらかが失敗した場合、deploy ジョブは実行されません。

2.4.8　ジョブ間のアウトプットの受け渡し

needs でジョブ間の依存関係を定義すると、あるジョブから後続のジョブになにかしらのアウトプットを引き継ぎたいという場合があります。例えば、あるジョブでアップロードしたアーカイブの URL を後続のジョブで参照したい、という場合です。

jobs.<job_id>.outputs

outputs を指定すると、そのジョブの後続のジョブに渡すアウトプットを定義できます。アウトプットは文字列で、「2.4.11 コンテキストと式を使った条件実行」で説明する式を記述することもできます。式は、ジョブの終了時に評価されます。

リスト 2.38 では、job1 のアウトプットを job2 で利用しています。

▼リスト 2.38　ジョブ間のアウトプットの受け渡し

```
jobs:
  job1:
    runs-on: ubuntu-latest
    outputs:
      random: ${{ steps.generate-random.outputs.random }}
    steps:
      - id: generate-random
        run: echo "::set-output name=random::$(openssl rand -base64 12)"
  job2:
    runs-on: ubuntu-latest
    needs: job1
    steps:
      - run: echo "${{ needs.job1.outputs.random }}"
```

outputs の random というキーに generate-random ステップのアウトプットを設定しています。式はジョブ終了時に評価されるため、ジョブ中のステップのアウトプットを参照する式を書けます。generate-random ステップでは、ステップのアウトプットにランダムな文字列を設定しています。echo でステップのアウトプットを設定する方法については、「2.8.2 アウトプットの設定: set-output」で詳しく説明します。

そして、job2 で job1 のアウトプットの値を needs というコンテキスト経由で利用しています。コンテキストについては、「2.4.11 コンテキストと式を使った条件実行」で解説します。

ジョブのアウトプットに「2.4.5 秘密情報」で説明した秘密情報が含まれる場合は、アウトプットの値がマスクされて空文字列となります。

2.4.9　マトリクスビルド

ワークフローを特定のパラメータの組み合わせで繰り返し実行したいという場合があります。例えば、ライブラリのテストを複数の OS やプログラミング言語のバージョンの組み合わせで行いたいというような場合です。GitHub Actions は、こういったマトリクスビルドを可能にする機能を提供しています。

試しに、サンプルリポジトリの ci.yml の unit-test ジョブを複数の OS と Node.js のバージョンで実行されるようにしてみます。リスト 2.39 のように unit-test ジョブを修正してください。

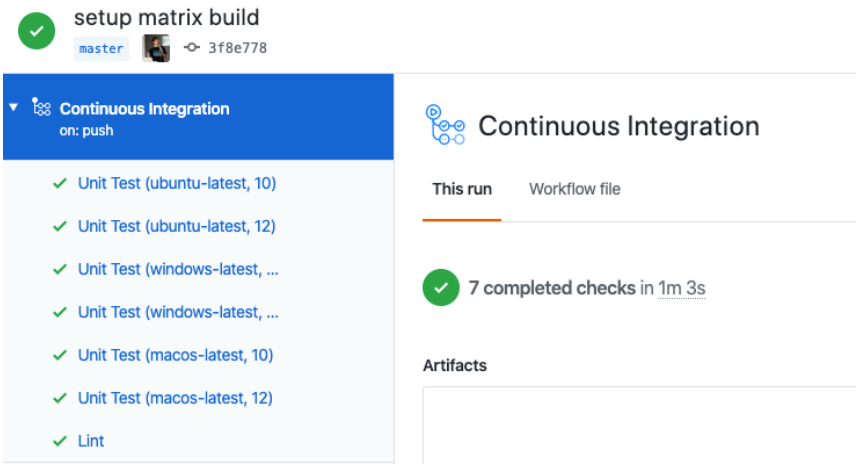
▼リスト 2.39　マトリクスビルド

```
 1: unit-test:
 2:   name: Unit Test
 3:   strategy:
 4:     matrix:
 5:       os: [ubuntu-latest, windows-latest, macos-latest]
 6:       node: [10, 12]
 7:   runs-on: ${{ matrix.os }}
 8:   steps:
 9:     - name: Checkout
10:       uses: actions/checkout@v2.0.0
11:     - name: Set Node.js ${{ matrix.node }}
12:       uses: actions/setup-node@v1.3.0
13:       with:
14:         node-version: ${{ matrix.node }}
15:     - name: Install dependencies
16:       run: npm ci
17:     - name: Test
18:       run: npm test
```

ファイルを修正したら、次のようにサンプルリポジトリに push します。

```
$ git add .github/workflows/ci.yml
$ git commit -m "setup matrix build"
$ git push origin master
```

実行されたワークフローをブラウザ上から確認します。図 2.3 のように、複数の OS と Node.js のバージョンの組み合わせで実行されているのがわかります。



▲図 2.3　マトリクスビルド

jobs.<job_id>.strategy

3 行目の `strategy` から、マトリクスビルドのための設定を定義しています。

jobs.<job_id>.strategy.matrix

4 行目の `matrix` で、マトリクスビルドの変数の組み合わせを定義しています。ここで定義した変数の組み合わせの数だけジョブのコピーが実行されます。例では、`os` を 3 種類、`node` を 2 種類定義しているので、合計 6 パターンの組み合わせでジョブが実行されます。仮に定義した変数をジョブ内で参照しなかったとしても、この組み合わせの数だけジョブは実行されます。（そんなことをする意味はまずないですが）

定義した変数は、7, 11, 14 行目のように `matrix` オブジェクト経由で参照できます。変数の参照については、「2.4.11 コンテキストと式を使った条件実行」でもう少し詳しく解説します。

jobs.<job_id>.strategy.matrix.include

`include` を指定すると、特定の変数の組み合わせのときだけ新しい変数を追加することができます。例えば、リスト 2.40 では `os` が `windows-latest` で `node` が `10` のときだけ、新しく `npm` という変数に `6` を設定しています。

▼リスト 2.40　特定の組み合わせのときに新しい変数を追加

```
strategy:
  matrix:
    os: [ubuntu-latest, windows-latest, macos-latest]
    node: [10, 12]
    include:
      - os: windows-latest
        node: 10
        npm: 6
```

また、include は、存在しない組み合わせを新たに追加することも可能です。例えば、リスト 2.41 では os が ubuntu-16.04、node が 8、npm が 6 という組み合わせを新たに追加しています。

▼リスト 2.41　新しい組み合わせを追加

```
strategy:
  matrix:
    os: [ubuntu-latest, windows-latest, macos-latest]
    node: [10, 12]
    include:
      - os: ubuntu-16.04
        node: 8
        npm: 6
```

jobs.<job_id>.strategy.matrix.exclude

exclude を指定すると、特定の変数の組み合わせをマトリクスビルドから除外できます。例えば、リスト 2.42 では os が macos-latest で node が 10 の組み合わせが除外され、5 パターンのジョブが実行されるようになります。

▼リスト 2.42　特定の組み合わせを除外

```
strategy:
  matrix:
    os: [ubuntu-latest, windows-latest, macos-latest]
    node: [10, 12]
    exclude:
      - os: macos-latest
        node: 10
```

exclude に指定する組み合わせはすべての変数を定義する必要はないです。例えば os だけ macos-latest を指定した場合は、すべての node 変数の組み合わせが除外されます。今回の例だと matrix の定義から削除すればいいですが、変数の数が

3 つ以上になってくると特定の 2 つの変数の組み合わせはすべて除外したいような場合に使い道が出てきます。

また、exclude の変数にリストで複数の値を渡すことはできません。具体的には、os: [windows-latest, macos-latest] のように書くとエラーになります。

jobs.<job_id>.strategy.fail-fast

fail-fast を指定すると、マトリクスビルドでいずれかのジョブが失敗したときにその他の実行中のジョブをキャンセルするかを定義できます。true だといずれかのジョブが失敗したときに実行中のジョブをすべてキャンセルし、false だとキャンセルせずにすべてのジョブを最後まで実行する挙動になります。指定しないときは、デフォルトで true になります。リスト 2.43 のように明示的に false を指定すると、いずれかの組み合わせが失敗しても実行中のジョブを途中でキャンセルしないようになります。

▼リスト 2.43　実行中のジョブを途中でキャンセルしない

```
strategy:
  matrix:
    os: [ubuntu-latest, windows-latest, macos-latest]
    node: [10, 12]
  fail-fast: false
```

jobs.<job_id>.strategy.max-parallel

max-parallel を指定すると、マトリクスビルドで実行するジョブの最大同時実行数を定義できます。指定しないときはデフォルトで可能な限り最大の数のジョブを同時実行します。リスト 2.44 のように書くと、マトリクスビルドで同時に実行するジョブの数を 2 つに制限できます。

▼リスト 2.44　最大同時実行数を制限

```
strategy:
  matrix:
    os: [ubuntu-latest, windows-latest, macos-latest]
    node: [10, 12]
  max-parallel: 2
```

2.4.10　スケジュールで定期実行

これまではイベント発生時のワークフローについて説明してきました。ですが、ワークフローを特定のスケジュールで定期実行したいケースもあります。例えば、依

存ライブラリの脆弱性チェックを日次で行いたいというような場合です。

ここでは、サンプルリポジトリに定期的に `npm audit`[*11] を実行して脆弱性情報をチェックするワークフローを新たに追加してみます。リスト 2.45 の内容で `.github/workflows/security.yml` を作成します。

▼リスト 2.45　.github/workflows/security.yml

```
 1: name: Security Audit
 2: on:
 3:   schedule:
 4:     - cron: '0 0 * * *'
 5:
 6: jobs:
 7:   audit:
 8:     name: Security Audit
 9:     runs-on: ubuntu-latest
10:     steps:
11:       - name: Checkout
12:         uses: actions/checkout@v2.0.0
13:       - name: Set Node.js 12.x
14:         uses: actions/setup-node@v1.3.0
15:         with:
16:           node-version: 12.x
17:       - name: Install dependencies
18:         run: npm ci
19:       - name: Audit
20:         run: npm audit
```

ファイルを作成したら、次のようにサンプルリポジトリに `push` します。

```
$ git add .github/workflows/security.yml
$ git commit -m "setup security audit workflow"
$ git push origin master
```

これで、毎日日本時間で午前 9 時になると脆弱性情報をチェックするワークフローが実行されます。（午前 9 時まで待てないという人は、この先の説明を参考に `cron` 部分を現在時刻の数分後に書き換えて動作確認してみてください）

on.schedule

`security.yml` の 3 行目で `schedule` を指定することで、ワークフローの定期実行を定義しています。`schedule` は、4 行目のようにリストで `cron` を持ち、POSIX

[*11] https://docs.npmjs.com/cli/audit

cron[*12] 形式で実行時刻を定義できます。ここでは、毎日 9 時ちょうどに実行されるように定義しています。

`cron` は空白で区切られた 5 つのフィールドを持つ文字列で記述でき、それぞれ次の単位を表します。

- 分（0 〜 59）
- 時（0 〜 23）
- 日（1 〜 31）
- 月（1 〜 12 または JAN 〜 DEC）
- 曜日（0 〜 6 または SUN 〜 SAT）

また、次の特殊文字が使えます。

- `*`: 任意の値
- `,`: AND
  - `0, 1,13 * * *` だったら、毎日 10 時と 22 時に実行される
- `-`: 範囲指定
  - `0 0-2 * * *` だったら、毎日 9 時、10 時、11 時に実行される
- `/`: 間隔指定
  - `0/15 * * * *` だったら、毎時 0 分、15 分、30 分、45 分に実行される

制限事項として、5 分間隔より短い間隔でワークフローを定期実行することはできません。なので、`* * * * *` のように毎分実行を指定してもワークフローは実行されず、最短間隔で実行できるのは `0/5 * * * *` となります。

また、`schedule` で定期実行されるワークフローは、デフォルトブランチの最新コミットで設定されているものだけとなります。デフォルトブランチ以外のブランチやタグでは定期実行できません。

2.4.11 コンテキストと式を使った条件実行

この項では、コンテキストと式を使って GitHub Actions が提供する情報にアクセスしたり、それらの情報を元にジョブやステップの実行を制御する方法について説明します。

[*12] https://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html#tag_20_25_07

式

式は、コンテキスト、リテラル、演算子、関数の組み合わせからなります。前に出てきたリスト 2.39 の `${{ matrix.os }}` は、`matrix` コンテキストへの参照を式として評価されるように書いたものです。`${{ <expression> }}` の書式で書くと、通常の文字列ではなく式として評価されます。このあと説明する `if` 条件式の中では、`${{}}` で囲わなくても式として評価されます。

コンテキスト

コンテキストは、ワークフローに与えられる実行時の情報です。表 2.6 は、提供されるコンテキスト情報の一覧です。

▼表 2.6　コンテキスト

コンテキスト名	説明
github	ワークフローを実行した GitHub イベントについての情報
env	環境変数
job	現在実行中のジョブの情報
steps	現在のジョブのステップ情報
runner	ランナーの情報
secrets	秘密情報（「2.4.5 秘密情報」）
strategy	現在のジョブの `strategy` 設定の情報（「2.4.9 マトリクスビルド」）
matrix	現在のジョブの `matrix` 設定の情報（「2.4.9 マトリクスビルド」）
needs	現在のジョブが依存するジョブの情報（「2.4.7 ジョブ間の依存関係の制御」）

コンテキストから取得できる情報の詳細は、公式ドキュメントを参照してください。

- https://help.github.com/en/actions/reference/context-and-expression-syntax-for-github-actions

それぞれのコンテキスト情報はオブジェクトです。例えば、`github` オブジェクトのコミット SHA 情報を参照するには次の 2 つの方法があります。

- `github['sha']`
- `github.sha`

jobs.<job_id>.if

ジョブに対して `if` を指定すると、条件式に一致するときだけジョブが実行されるようになります。リスト 2.46 のように書くと、`push` イベントが `master` ブランチに対して発生したときのみ `deploy` ジョブが実行されます。

▼リスト 2.46　ジョブの実行を条件式で制御

```
on: push
jobs:
  deploy:
    if: github.ref == 'refs/heads/master'
    steps:
      ...
```

条件式に一致しないときはジョブが実行されません。実行されなかったジョブを `needs` で指定していた場合、後続のジョブも実行されません。

jobs.<job_id>.steps.if

ステップに対して `if` を指定すると、条件式に一致するときだけステップが実行されるようになります。リスト 2.47 のように書くと、イベントが `pull_request` のときのみステップが実行されます。

▼リスト 2.47　ステップの実行を条件式で制御

```
steps:
  - if: github.event_name == 'pull_request'
    run: ...
```

条件式に一致しないときはステップが実行されませんが、ジョブはそこで終わらず後続のステップが実行されます。

リテラル

式の一部として使える**リテラル**には、表 2.7 の通り 4 種類あります。

▼表 2.7　リテラル

種類	説明
boolean	`true` か `false`
null	`null`
number	JSON の数値と同じフォーマット
string	シングルクォートで囲った文字列

string リテラル内でシングルクォートを使うときは、`${{ 'That''s right' }}` のようにシングルクォートを重ねてエスケープします。

演算子

表 2.8 は、式の一部として使える**演算子**の一覧です。

▼表 2.8　演算子

演算子	説明
()	関数呼び出し
[]	インデックス参照
.	プロパティ参照
!	not
<	より小さい
<=	以下
>	より大きい
>=	以上
==	等しい
!=	等しくない
&&	and
||	or

比較演算子で異なる型のデータを比較する場合、データは number 型に変換して比較されます。

▼表 2.9　number 型への変換

型	変換ルール
Null	0
Boolean	true のときは 1、false のときは 0
文字列	number としてパースできる場合はそのまま、空文字は 0、それ以外は NaN
配列	NaN
オブジェクト	NaN

NaN 同士の比較は false になります。文字列同士の比較は、大文字小文字を区別しません。オブジェクトと配列は同じインスタンスのときのみ等しいと判定されます。

関数

式の中で使える**関数**には、次のものがあります。

- contains(searchString, searchValue)
- startsWith(searchString, searchValue)
- endsWith(searchString, searchValue)
- format(string, replaceValue0, replaceValue1, ..., replaceValueN)
- join(element, optionalElem)
- toJson(value)
- fromJson(value)
- hashFiles(path)

contains(searchString, searchValue) は、searchString に searchValue が含まれていたら true を返します。

startsWith(searchString, searchValue) は、searchString が searchValue から始まっていたら true を返します。

endsWith(searchString, searchValue) は、searchString が searchValue で終わっていたら true を返します。

format(string, replaceValue0, replaceValue1, ..., replaceValueN) は、文字列に値を埋め込んで返します。例えば、format('Hello, {0} and {1}', 'Alice', 'Bob') のように書くと、"Hello, Alice and Bob" となります。

join(element, optionalElem) は、element は配列か文字列で、配列のときはすべての要素を結合した一つの文字列にして返します。optionalElem が指定されたときは、element に結合されます。join(['Hello', 'World']) のように書くと、"Hello World" となり、join('Hello', 'World') のように書いても、"Hello World" となります。

toJson(value) は、value をきれいにフォーマットした JSON 文字列にして返します。コンテキスト情報をデバッグするときによく使います。例えば、リスト 2.48 のように書くと github コンテキストの情報がログに出力されます。

▼リスト 2.48　コンテキスト情報を出力

```
steps:
  - run: echo "${{ toJson(github) }}"
```

fromJson(value) は、value を JSON 文字列として評価し、JSON オブジェクトに変換します。value に JSON として有効でない文字列を渡すと、エラーになります。例えば、リスト 2.49 のように書くと、{"greeting":"Hello, World!"} という文字列が JSON オブジェクトに変換され、greeting プロパティ経由で値にアクセスできるようになります。

▼リスト 2.49　文字列を JSON オブジェクトに変換

```
steps:
  - run: echo ${{ fromJson('{"greeting":"Hello, World!"}').greeting }}
```

hashFiles(path) は、path に一致するすべてのファイルのハッシュを計算します。path はワークスペースからの相対パスとして評価され、ワークスペース外のファイルは評価されません。「2.4.6 イベントのフィルタ」 と同様の特殊文字が使えます。例えば、hashFiles('**/package-lock.json') と書くと、リポジトリ内のすべての package-lock.json のハッシュが計算されます。ハッシュは、まず path に一致したそれぞれのファイルの SHA-256 ハッシュを計算し、最終的にすべてのハッシュから SHA-256 ハッシュを計算します。

また、if 条件式の中だけで使える**ジョブのステータスをチェックする関数**として、次のものがあります。

- success()
- always()
- cancelled()
- failure()

success() は、それより以前のステップがすべて成功している場合に true を返します。if 条件式で他の 3 つの関数が呼び出されなかったときは、デフォルトでこの success() が評価されます。

always() は、ジョブのステータスに関わらず常に true を返します。リソース削除のような、ジョブの成功・失敗に関わらず実行されてほしいステップに対して使います。

cancelled() は、それより以前のステップ実行中にキャンセルされた場合に true を返します。キャンセルされたときのみ特別な処理を行いたいときに使います。

failure() は、それより以前のステップのいずれかが失敗している場合に true を返します。失敗したときに通知を飛ばしたいような場合に使います。

リスト 2.50 は、ジョブが失敗したときのみ実行されるステップの例です。

▼リスト 2.50　以前のステップが失敗しているときのみ実行

```
steps:
  ...
  - if: failure()
    run: ...
```

オブジェクトフィルタ

オブジェクトの配列から、特定のプロパティのみ抜き出した配列を取得する**オブジェクトフィルタ**記法があります。例えば、リスト 2.51 のようなオブジェクトの配列が `commits` という名前の変数で存在したとします。

▼リスト 2.51　commits

```
[
  { "id": "abc", ... }
  { "id": "def", ... }
  { "id": "ghi", ... }
]
```

このとき、`commits.*.id` と書くと、`[ "abc", "def", "ghi" ]` という配列を取得できます。

2.4.12　ジョブのコンテナ内実行

ここまでワークフロー内のすべてのジョブを VM 上で直接実行してきましたが、GitHub Actions ではジョブをコンテナ内で実行できます。リスト 2.52 のように書くと、`steps` 内の処理がコンテナ内で実行されるようになります。

▼リスト 2.52　ジョブのコンテナ内実行

```
jobs:
  unit-test:
    runs-on: ubuntu-latest
    container:
      image: node:12.14.1-stretch
      env:
        NODE_ENV: development
      ports:
        - 8080
      options: --cpus 1
    steps:
      ...
```

jobs.<job_id>.container

`container` を指定すると、そのジョブのステップがすべてコンテナ内で実行されるようになります。ただし、`runs-on` で指定する動作環境の OS が Linux である必要があります。

`container` を指定した上でコンテナアクションを実行した場合は、同じネット

ワークを共有した別コンテナとして実行されます。また、ホーム（$HOME）とワークスペース（$GITHUB_WORKSPACE）、そして webhook イベントペイロードファイル（$GITHUB_EVENT_PATH）が存在するディレクトリは、ホストマシンの同じディレクトリがそれぞれのコンテナにマウントされて共有されます。

container を指定しなかった場合は、VM 上ですべてのステップ（コンテナアクションを除く）が実行されます。

jobs.<job_id>.container.image

image では、コンテナのイメージ名を定義します。リスト 2.52 では、node:12.14.1-stretch イメージを使うように設定しています。「jobs.<job_id>.steps.uses」のコンテナアクションと同様に、Docker Hub 上のイメージか、パブリックな Docker レジストリ上のイメージを指定できます。

container に image 以外のキーを指定しない場合は、リスト 2.53 のように省略して書けます。

▼リスト 2.53　image のみ指定する場合は省略可能

```
container: node:12.14.1-stretch
```

jobs.<job_id>.container.env

env は、コンテナ内の環境変数を定義します。リスト 2.52 では、NODE_ENV という環境変数に development という値を設定しています。

「2.4.4 環境変数」で解説している方法でワークフロー、ジョブ、ステップのいずれかで同じ名前の環境変数を設定した場合、そちらの設定で上書きされます。また、デフォルトの環境変数は、コンテナ内の環境に合わせて新たに設定されているので、問題なく使えます。

jobs.<job_id>.container.ports

ports は、ホストマシンに公開するポート番号を定義します。リスト 2.52 では、8080 番ポートをホストに公開しています。docker create コマンドの -p,--publish オプションと同様の効果です。

jobs.<job_id>.container.volumes

公式のドキュメントによると、volumes を指定するとホストマシン上のディレクトリをコンテナ内にマウントできると書いてあります。しかし、この本の執筆時点では、volumes を指定してもマウントしないようです。GitHub Community Forum

でも報告されています。

- https://github.community/t5/GitHub-Actions/Container-volumes-key-not-mounting-volume/m-p/34798

代わりに、次に紹介する `options` で `-v` オプションを指定すればマウントできます。

jobs.<job_id>.container.options

`options` は、コンテナ作成時に `docker create` コマンドに渡るオプションを定義します。リスト 2.52 では、`--cpus 1` を指定して、コンテナが使用する CPU リソースの上限を 1 コアに制限しています。どのようなオプションが指定できるかは `docker create` の公式ドキュメント[*13]を参照してください。

2.4.13 サービスコンテナ

ワークフローの実行中に、なにかしらのサービスをコンテナで立ち上げたいという場合があります。例えば、テストを実行するためにデータベースが必要なときなどです。リスト 2.54 では、PostgreSQL をサービスコンテナとして起動しています。

▼リスト 2.54　サービスコンテナ

```
 1: jobs:
 2:   unit-test:
 3:     runs-on: ubuntu-latest
 4:     services:
 5:       postgres:
 6:         image: postgres:12.1
 7:         env:
 8:           POSTGRES_USER: pguser
 9:           POSTGRES_PASSWORD: password
10:         ports:
11:           - 5432:5432
12:         options: --health-cmd pg_isready --health-interval
    10s --health-timeout 5s --health-retries 5
```

jobs.<job_id>.services

`services` で、サービスコンテナを定義します。`services` は、マップで指定でき、複数のサービスコンテナを立ち上げ可能です。

ジョブをコンテナ内で実行したり、コンテナアクションを実行する場合は、それぞ

[*13] https://docs.docker.com/engine/reference/commandline/create/#options

れ同じネットワークを共有した別コンテナとして実行されます。また、services のキー名（リスト 2.54 だと postgres）をホスト名として参照できます。

container とは異なり、こちらはホストマシンのホームやワークスペースはデフォルトでは共有されないようです。

jobs.<job_id>.services.<service_name>

services のマップのキー（<service_name> とします）で、サービスコンテナの名前を定義します。サービスの名前同士は重複してはダメです。アルファベットか _ で始まる文字列で、使える文字列はアルファベット、数字、-、_ のみです。

前述したように、この名前はホスト名として他コンテナから参照できます。

jobs.<job_id>.services.<service_name>.image

image では、サービスコンテナのイメージ名を定義します。リスト 2.54 では、postgres:12.1 イメージを使うように設定しています。「jobs.<job_id>.steps.uses」のコンテナアクションと同様に、Docker Hub 上のイメージか、パブリックな Docker レジストリ上のイメージを指定できます。

services.<service_name> に image 以外のキーを指定しない場合は、リスト 2.55 のように省略して書けます。

▼リスト 2.55　image のみ指定する場合は省略可能

```
services:
  postgres: postgres:12.1
```

jobs.<job_id>.services.<service_name>.env

env は、コンテナ内の環境変数を定義します。リスト 2.54 では、POSTGRES_USER という環境変数に pguser、POSTGRES_PASSWORD という環境変数に password という値を設定しています。

container とは異なり、「2.4.4 環境変数」で解説しているデフォルトの環境変数はサービスコンテナには設定されないようです。代わりに、GITHUB_ACTIONS という環境変数に true が設定されます。

jobs.<job_id>.services.<service_name>.ports

ports は、サービスコンテナからホストマシンに公開するポート番号を定義します。リスト 2.54 では、5432 番ポートをホストに公開しています。docker create コマンドの -p,--publish オプションと同様の効果です。VM 上から直接コマンドやアクションが実行される場合は、ホストマシンからサービスコンテナのポートにア

クセスできるように公開する必要があります。

ちなみに、サービスコンテナが公開しているポート番号がホストマシン上でどのポート番号に割り当てられてるかはコンテキストの `${{ job.services.<service_name>.ports }}` から参照できます。コンテキストについては、「2.4.11 コンテキストと式を使った条件実行」で解説しています。

例えば、リスト 2.56 のように書くと、`nginx` コンテナのポート 8080 がホストマシン上でランダムなポート番号に割り当てられます。

▼リスト 2.56　ランダムな番号で公開されたポート

```
services:
  nginx:
    image: nginx:1.17.8
    ports:
      - 8080
```

このとき、`${{ job.services.nginx.ports['8080'] }}` でホストマシン上に割り当てられたポート番号を参照できます。

jobs.<job_id>.services.<service_name>.volumes

公式のドキュメントによると、`volumes` を指定するとホストマシン上のディレクトリをコンテナ内にマウントできると書いてあります。しかし、`container` と同様に、この本の執筆時点では、`volumes` を指定してもマウントしないようです。

代わりに、次に紹介する `options` で `-v` オプションを指定すればマウントできます。

jobs.<job_id>.services.<service_name>.options

`options` は、サービスコンテナ作成時に `docker create` コマンドに渡るオプションを定義します。リスト 2.54 では、`--health-cmd pg_isready --health-interval 10s --health-timeout 5s --health-retries 5` を指定して、サービスコンテナのヘルスチェックを設定しています。どのようなオプションが指定できるかは `docker create` の公式ドキュメント[*14]を参照してください。

[*14] https://docs.docker.com/engine/reference/commandline/create/#options

YAML のアンカーとエイリアス

ここまで紹介したように、GitHub Actions のワークフローの設定ファイルは YAML で記述します。しかし、YAML のアンカーとエイリアス機能はサポートしていません。

YAML のアンカーとエイリアスは、同じ記述を繰り返し書かなくてすむように共通化する上で便利な機能です。例えば、リスト 2.57 のような YAML があったとします。

▼リスト 2.57　アンカーとエイリアスを使用しない YAML

```
- people:
    name: Alice
    age: 20
- people:
    name: Alice
    age: 20
```

これをアンカーとエイリアスを使うと、リスト 2.58 のように書くことができます。

▼リスト 2.58　アンカーとエイリアスを使用した YAML

```
- people: &alice
    name: Alice
    age: 20
- people: *alice
```

`&alice` で、後で参照することのできるアンカーを作成しています。このアンカーを、`*alice` というエイリアスで参照できます。

ワークフローの設定ファイルでもこの機能が使えれば大規模なワークフローを作成するときに設定が共通化できるのでとても便利になりそうなのですが、残念ながら現時点では対応されていません。使用するとエラーになってしまいます。

コミュニティフォーラムでも要望は上がっており[*15]、GitHub 側も認識しているようなので、今後の対応を期待しましょう。

[*15] https://github.community/t5/GitHub-Actions/Support-for-YAML-anchors/td-p/

2.5 キャッシュ

CI/CD では、リポジトリが依存するパッケージのダウンロードが原因でビルド時間が長くなってしまうことがあります。近年の CI/CD サービスでは、ビルドごとに完全にクリーンな実行環境が用意され、前回のビルドでダウンロードしたファイルが持ち越されないからです。

この問題を解決するためには、CI/CD が提供するキャッシュ機能を用いて、異なるビルド間でダウンロードしたパッケージを使い回して高速化することが一般的です。GitHub Actions でも `actions/cache` を使うことでキャッシュ機能が利用可能です。この節では、GitHub Actions のキャッシュ機能について解説します。

2.5.1 サンプルリポジトリにキャッシュを導入

まず、実際にサンプルリポジトリでキャッシュを導入してみます。リスト 2.59 のように、`unit-test` ジョブの `npm ci` コマンドを実行するステップの前に 2 つのステップを追加する修正を入れます。また、`lint` ジョブも同様の修正をしてください。

▼リスト 2.59　キャッシュ

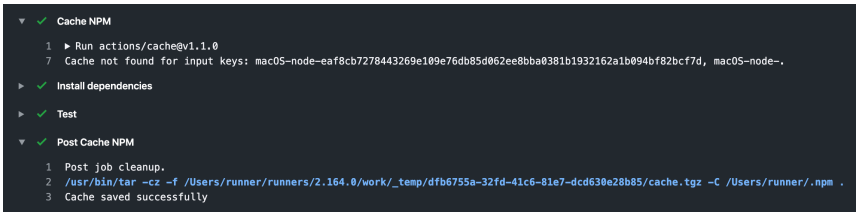
```
 1: name: Continuous Integration
 2: on: push
 3:
 4: jobs:
 5:   unit-test:
 6:     ...
 7:     steps:
 8:       ...
 9:       - name: Get NPM cache directory
10:         id: npm-cache
11:         run: |
12:           echo "::set-output name=dir::$(npm config get cache)"
13:       - name: Cache NPM
14:         uses: actions/cache@v2.0.0
15:         with:
16:           path: ${{ steps.npm-cache.outputs.dir }}
17:           key: ${{ runner.os }}-node-
    ${{ hashFiles('**/package-lock.json') }}
18:           restore-keys: |
19:             ${{ runner.os }}-node-
20:       - name: Install dependencies
21:         run: npm ci
22:       ...
```

30336

変更できたら、次のコマンドで変更をリポジトリに push します。

```
$ git add .github/workflows/ci.yml
$ git commit -m "cache NPM dependencies"
$ git push origin master
```

すると、ワークフローが実行され、初回はキャッシュが存在しないので図 2.4 のようにキャッシュが復元されず、ジョブの最後にキャッシュが保存されるようになります。



▲図 2.4　キャッシュが存在しないとき

もう一度ワークフローを実行するために、次のコマンドで空の変更をリポジトリに push します。

```
$ git commit --allow-empty -m "trigger workflow"
$ git push origin master
```

すると、今度は先程保存されたキャッシュが存在するので、図 2.5 のようにキャッシュが復元されるようになります。また、すでにキャッシュが存在するときはジョブの最後で上書きされずに終了します。

▲図 2.5　キャッシュが存在するとき

リスト 2.59 の 9 〜 12 行目のステップでは、NPM のキャッシュディレクトリを取得してアウトプットに格納しています。アウトプットの設定方法については、「2.8 ログによるコマンド実行」で詳細に解説します。このステップを実行するのは、NPM のキャッシュディレクトリが OS によって異なるためです。Linux しかビルドしないということであれば Linux のパス（~/.npm）に決め打ちしてしまっても問題ないと思いますが、互換性を保つためにコマンド経由で取得するほうが安全です。

13 〜 19 行目のステップでは、キャッシュを設定しています。

`actions/cache` アクションを実行すると、その時点で `key` に指定したキーに完全一致するキャッシュが存在するときは、`path` で指定したファイルパスに復元されます。

`key` で指定したキーでキャッシュが存在しない場合は、`restore-keys` で指定したキーに前方一致するキーを持つキャッシュが復元されます。キーとキャッシュのマッチングについての詳細は、「2.5.3 キーのマッチング順序」で説明します。どちらのキーでもキャッシュが存在しない場合は、キャッシュの復元は行われません。

また、`actions/cache` アクションを実行すると、ジョブの最後に指定したキーで指定したファイルパスの内容がキャッシュとして保存されるようになります。すでに同じキーでキャッシュが存在するときは保存されず、キャッシュの上書きは行われません。

リスト 2.59 の `key` では、コンテキストから複数の値を取得してキーを組み立てています。`${{ runner.os }}` でジョブ実行環境の OS を取得して、キーに含めています。このジョブでは複数の OS でマトリクスビルドを行っており、NPM のキャッシュする内容は OS によって異なってくる可能性があるためです。

また、`${{ hashFiles('**/package-lock.json') }}` でリポジトリ内の依存関係を定義するすべてのロックファイルのハッシュをキーに含めています。これにより、NPM の依存関係に変更があったときは新たにキャッシュが作成され直します。このリポジトリにはルートディレクトリにしか `package-lock.json` が存在しないので `**/` は必要ないですが、リポジトリ内で複数のモジュールを管理するケースを

考えて全ディレクトリの package-lock.json を指定しています。

2.5.2 actions/cache

actions/cache は GitHub が公式で提供している、ワークフロー実行中にキャッシュを扱うためのアクションです。次のリポジトリで公開されています。

- https://github.com/actions/cache

@actions/cache[*16] という NPM パッケージを利用することで、Node.js からも同様のキャッシュ操作が可能です。
ここからは、actions/cache アクションについて説明します。

パラメータ

actions/cache アクションは、次のパラメータを受け取ります。

- path: キャッシュとして保存・復元するファイルパス
  - 必須
  - ファイル、ディレクトリ、ワイルドカードによるパターン[*17]を複数指定可能
- key: キャッシュを保存・復元するためのキー
  - 必須
- restore-keys: キャッシュ復元時に key に完全に一致するキャッシュが存在しなかったときに使われるキーの一覧
  - YAML のリストではなく、改行区切りの文字列で指定する

アウトプット

actions/cache アクションは、次のアウトプットを設定します。

- cache-hit: key に完全一致するキャッシュが存在したときは true、存在しなかったときは false
  - restore-keys でマッチングしても false

[*16] https://www.npmjs.com/package/@actions/cache
[*17] https://github.com/actions/toolkit/tree/master/packages/glob

プログラミング言語ごとの例

公式で主要なプログラミング言語とパッケージマネージャーごとの例が用意されています。どのディレクトリをキャッシュすればいいか、どのファイルのハッシュをキーに含めればいいか、といったことがわからないときに参考になります。

- `https://github.com/actions/cache/blob/master/examples.md`

2.5.3 キーのマッチング順序

ここでは、`key` と `restore-keys` によるキャッシュとキーのマッチング順序について説明します。

まず、検索されるブランチの順序についてですが、現在ワークフローを実行しているブランチ→親ブランチの順序で、各ブランチで作成されたキャッシュを `key` と `restore-keys` を使って検索します。例えば、`push` イベントや `pull_request` イベントであれば次のような順序でキーが検索されます。

- `push`: 現在のブランチ → デフォルトブランチ
- `pull_request`: 現在のブランチ → ベースブランチ → デフォルトブランチ

次に、キーによるキャッシュの検索ですが、まず `key` に完全一致するキャッシュが検索されます。存在しなかった場合、`restore-keys` に書かれたキーを上から順に前方一致で検索します。なので、`restore-keys` に複数のキーを指定するときは、より長い順に書いていくのがよさそうです。restore-keys の特定のキーに前方一致するキャッシュが複数あるときは、より最近に作成されたキャッシュが使われます。

2.5.4 ジョブ失敗時

キャッシュが保存されるのはジョブ成功時のみで、ジョブが失敗・キャンセルされた場合はキャッシュが保存されません。

2.5.5 キャッシュの無効化

一度作成したキャッシュの上書きはできません。なので、キャッシュを無効化したいときは、新しいキーでキャッシュを作成し直す必要があります。そのためには、リスト 2.60 のように、環境変数でキーの先頭にバージョン名を入れておくと、その数字を増やすだけでキャッシュをまとめて無効化できるので便利です。

▼リスト 2.60　キーの先頭にバージョンを含める

```
...
env:
  cache-version: v1

jobs:
  unit-test:
    ...
    steps:
      ...
      - uses: actions/cache@v2.0.0
        with:
          path: ${{ steps.npm-cache.outputs.dir }}
          key:  ${{ env.cache-version }}-${{ runner.os }}-node-
    ${{ hashFiles('**/package-lock.json') }}
          restore-keys: |
            ${{ env.cache-version }}-${{ runner.os \}\}-node-
```

2.5.6　権限

リポジトリに対してプルリクエストを作成できる権限があれば、誰でもキャッシュにアクセス可能です。これは、fork してプルリクエストを作成した場合でもキャッシュにアクセス可能です。

つまり、リポジトリの read 権限があるユーザーなら誰でもキャッシュの中身を閲覧することができてしまうということです。キャッシュには秘密情報を含めないように注意しましょう。

2.5.7　制限事項

キャッシュが利用できるイベントタイプ

GitHub Actions のキャッシュは、‘GITHUB_REF‘ が存在するすべてのイベントで利用可能です。イベントの種類については、「2.7 イベントとアクティビティ」を参照してください。

ブランチ

ワークフローからアクセスできるキャッシュは、現在のブランチ、プルリクエストのベースブランチ、デフォルトブランチで作成されたキャッシュのみです。それ以外のブランチで作成されたキャッシュにはアクセスできません。

キャッシュの保存期間

7 日間アクセスされなかったキャッシュは、削除されます。

また、キャッシュの数には制限はありませんが、キャッシュのサイズには次のような上限があります。

- リポジトリ全体で最大 5GB のキャッシュサイズ上限がある
  - 圧縮した後のファイルサイズで判定される様子
  - 5GB を超えた場合、キャッシュのアクセス時間が古い順に追い出される

サイズ上限がある代わりに、現時点では GitHub Actions のキャッシュ機能は完全に無料で利用できます。

2.6 アーティファクト

CI/CD では、ビルド中に生成したファイルをビルド後に利用できるように保存したいことがあります。例えば、バイナリやアーカイブなどの成果物や、デバッグ用のログやテスト結果、カバレッジなどの情報を保存したいということがよくあります。

GitHub Actions では、アーティファクトという機能を用いてワークフロー実行時の成果物の保存が実現可能です。

2.6.1 サンプルリポジトリにアーティファクトを導入

実際に、サンプルリポジトリでアーティファクトを利用してみます。ここでは、`unit-test` ジョブでテストカバレッジを HTML ファイルに生成し、アーティファクトに保存してみます。

まず、次のコマンドでテストカバレッジを取得するための依存モジュールを追加します。

```
$ npm install --save-dev nyc
```

`nyc`[*18] は、Istanbul[*19] という Node.js のテストカバレッジを取得するツールの CLI です。

次に、`package.json` の `scripts` 部分をリスト 2.61 のように修正します。

[*18] https://www.npmjs.com/package/nyc
[*19] https://istanbul.js.org/

▼リスト 2.61　package.json

```
"scripts": {
  "lint": "eslint .",
  "test": "nyc --reporter=html --reporter=text mocha"
},
```

この変更によって、npm test 実行時に標準出力と HTML ファイルにテストカバレッジが出力されるようになります。

生成されたテストカバレッジファイルや一時ファイルを間違ってリポジトリに追加しないように、.gitignore にリスト 2.62 の 2 行を追加します。

▼リスト 2.62　.gitignore

```
.nyc_output
coverage
```

最後に、.github/workflows/ci.yml をリスト 2.63 のように修正して、unit-test ジョブの最後で actions/upload-artifact アクションを実行します。

▼リスト 2.63　.github/workflows/ci.yml

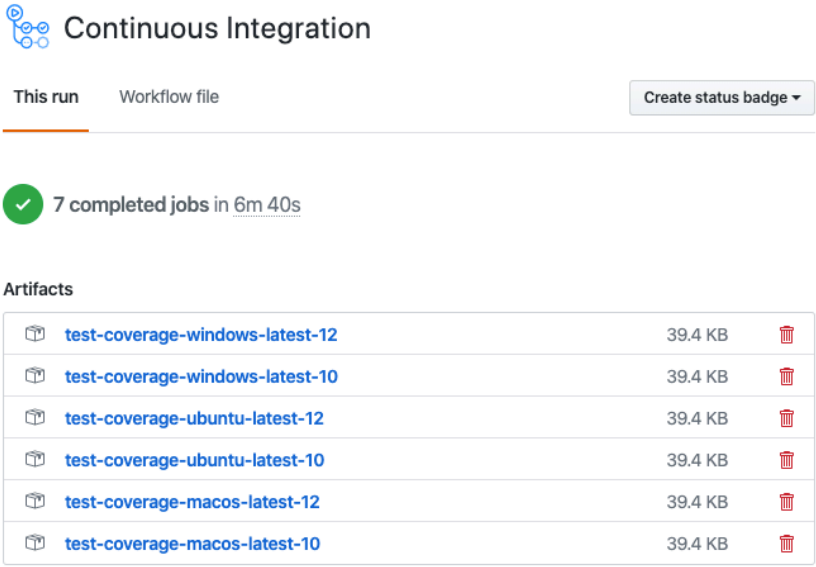
```
 1: jobs:
 2:   unit-test:
 3:     ...
 4:     steps:
 5:       ...
 6:       - name: Test
 7:         run: npm test
 8:       - name: Upload test coverage
 9:         uses: actions/upload-artifact@v2
10:         with:
11:           name: test-coverage-${{ matrix.os }}-${{ matrix.node }}
12:           path: coverage
```

修正できたら、次のコマンドで変更をリポジトリに push します。

```
$ git add .github/workflows/ci.yml .gitignore package-lock.json package.json
$ git commit -m "upload test coverage"
$ git push origin master
```

すると、ワークフローが実行され、すべてのジョブ完了後にブラウザからワークフローを開くと、図 2.6 のようにアーティファクトの一覧が表示され、ダウンロードで

きます。ファイルは zip 形式でダウンロードされ、展開するとテストカバレッジの HTML ファイルが展開されます。



▲図 2.6　アーティファクト一覧

ちなみに、右のゴミ箱アイコンをクリックすることでアーティファクトを削除することも可能です。

リスト 2.63 の 8 〜 12 行目でアーティファクトのアップロードを定義しています。`actions/upload-artifact` は、`path` パラメータで指定したディレクトリ下のファイルを、`name` パラメータで指定したアーティファクト名でアーティファクトに保存します。

`name` には、`${{ matrix.os }}` と `${{ matrix.node }}` を含めています。これは、同じ名前でアーティファクトを保存すると上書きしてしまうので、マトリクスビルドのそれぞれのジョブで異なるファイルとして保存されるようにするためです。[*20]

`path` には、`coverage` ディレクトリを指定しています。これは、`nyc` がデフォルトで `coverage` ディレクトリ下にテストカバレッジを作成するからです。

[*20] 実際には、すべての組み合わせでテストカバレッジを取得する必要は通常ないです

2.6.2 actions/upload-artifact

actions/upload-artifact は、GitHub が公式で提供している、ワークフロー実行中にアーティファクトをアップロードするためのアクションです。次のリポジトリで公開されています。

- https://github.com/actions/upload-artifact

パラメータ

actions/upload-artifact アクションは、次のパラメータを受け取ります。

- name: アーティファクトの名前
  - 省略するとデフォルトで artifact という名前になる
- path: アーティファクトとしてアップロードするファイルもしくはディレクトリ
  - 必須
  - 相対パスを指定したときはワークスペースからの相対パスと判定される
  - ワイルドカードを利用可能[*21]

ブラウザ上からアーティファクトをダウンロードすると、<name>.zip という名前でダウンロードされます。この zip を開くと、path で指定したファイル（ディレクトリを指定したときはディレクトリ下のファイル）がそのまま展開されます。

同じ name で複数回 upload-artifact アクションを実行すると、path で指定したファイルが存在すれば上書き、存在しなければ追加されます。

このアクションには、アウトプットは存在しません。

2.6.3 actions/download-artifact

actions/download-artifact は、GitHub が公式で提供している、ワークフロー実行中にアーティファクトをダウンロードするためのアクションです。次のリポジトリで公開されています。

- https://github.com/actions/download-artifact

このアクションを使うことで、ワークフロー実行中に複数のジョブ間でアーティファクト経由でファイルの受け渡しが可能です。例えば、リスト 2.64 のように書く

[*21] https://github.com/actions/toolkit/tree/master/packages/glob

と、upload ジョブでアップロードしたアーティファクトを download ジョブ中でダウンロードして使用できます。

▼リスト 2.64　actions/download-artifact

```
jobs:
  upload:
    runs-on: ubuntu-latest
    steps:
      - run: |
          mkdir data
          echo "Hello" > data/hello.txt
      - uses: actions/upload-artifact@v2
        with:
          name: data
          path: data
  download:
    runs-on: ubuntu-latest
    needs: upload
    steps:
      - uses: actions/download-artifact@v2
        with:
          name: data
      - run: cat hello.txt
```

ただし、このアクションでは複数のワークフロー実行をまたいだアーティファクトの受け渡しはできません。別のワークフロー実行で保存されたアーティファクトをダウンロードするには、REST API を使うか、Amazon S3 などの外部サービスを使う必要があります。REST API については、「2.14 REST API」で解説します。

パラメータ

actions/download-artifact アクションは、次のパラメータを受け取ります。

- name: アーティファクトの名前
  - 省略するとすべてのアーカイブがダウンロードされる
- path: アーティファクトをダウンロードして展開する先のファイルパス
  - 省略するとワークスペース下に name と同じ名前のディレクトリが作成され、その下に展開される
  - 相対パスを指定したときはワークスペースからの相対パスと判定される

このアクションには、アウトプットは存在しません。

2.6.4　キャッシュとアーティファクトの違い

キャッシュとアーティファクトは、ファイルを保存するための仕組みという点ではよく似ています。しかし、役割はそれぞれ異なります。

キャッシュは、ジョブやワークフローの複数の実行の間でファイルを再利用することが目的です。基本的には、依存パッケージのような、たまにしか変更されないようなファイルに対して使用します。

一方で、アーティファクトは、ワークフローの一回の実行内でファイルを複数のジョブ間で受け渡したり、ワークフロー完了後に利用するファイルを保存するために使います。バイナリやアーカイブなどの成果物や、デバッグ用のログやテスト結果、カバレッジなどの情報を保存するために使うことが多いです。

2.6.5 権限

アーティファクトのダウンロードはリポジトリの read 権限があるユーザーなら誰でもできます。アーティファクトに秘密情報を含めないように注意しましょう。

また、ユーザーがアーティファクトを削除するには、リポジトリの write 権限が必要です。

2.6.6 制限事項

アーティファクトの保存期間

保存したアーティファクトは、90 日で削除されます。ただし、プルリクエストのアーティファクトについては、新しいコミットがプルリクエストに push されるたびに保存期間が 90 日にリセットされます。

料金

プライベートリポジトリではアーティファクトによるストレージ使用量によって料金がかかります。詳細については、「1.3 料金体系」を参照してください。

2.7 イベントとアクティビティ

ここまでにも見てきたように、GitHub Actions のワークフローは、**イベント**によって実行が開始されます。さらに、「2.4.6 イベントのフィルタ」でも解説したとおり、イベントによっては複数の**アクティビティ**が存在し、特定の種類のアクティビティのときのみワークフローを実行するように設定できます。

また、イベントは**ペイロード**という情報を持っており、`github.event` コンテキスト経由でワークフロー内でアクセスできます。コンテキストについては、「2.4.11 コンテキストと式を使った条件実行」を参照してください。

イベントとアクティビティの一覧については、数が多いため本書では解説しません。公式ドキュメントを参照してください。

- https://help.github.com/en/actions/reference/events-that-trigger-workflows

ここでは、GitHub Actions でイベントを扱う上で気をつけるべき点について説明します。

pull_request のデフォルトアクティビティ

`pull_request` イベントは、デフォルトでは次のアクティビティのみワークフローを実行します。

- `opened`: プルリクエストが作成されたとき
- `synchronize`: プルリクエストが新しい push で更新されたとき
- `reopened`: プルリクエストがリオープンされたとき

それ以外のアクティビティでワークフローが実行されるようにするには「2.4.6 イベントのフィルタ」で説明した `types` を明示的に書く必要があります。例えば、リスト 2.65 のように書くとプルリクエストのラベルに変更があったときにワークフローを実行します。

▼リスト 2.65　pull_request のアクティビティを指定

```
on:
  pull_request:
    types: [labeled, unlabeled]
```

それ以外のイベントではデフォルトですべてのアクティビティでワークフローが実行されるので、注意してください。

2.7.1　ワークフロー内からのイベント実行

「2.4.5 秘密情報」で解説した `GITHUB_TOKEN` を使ってワークフロー内でリポジトリを操作してなにかしらのイベントを発生させても、新たなワークフローを実行することはありません。例えば、push イベント時に実行されるワークフロー内のステップで `GITHUB_TOKEN` を使ってリポジトリに push しても、新しいワークフローは実行されません。もしワークフロー実行中に別のワークフローを実行したい場合は、`GITHUB_TOKEN` 以外のパーソナルアクセストークンなどを使ってイベントを発生させる

必要があります。

2.7.2 GITHUB_SHA と GITHUB_REF

「2.4.4 環境変数」でも説明したとおり、GitHub Actions はイベント発生時に GITHUB_SHA と GITHUB_REF という環境変数を設定します。このときの GITHUB_SHA で実行されるワークフローが決まります。

例えば、push イベントであれば、GITHUB_SHA は push されたコミットの SHA が設定されます。なので、push した時点でのそのブランチ上のワークフロー設定ファイルを元に実行されるワークフローが決定されます。

今度は、issue_comment イベントの場合を考えてみます。プルリクエストにコメントが追加されたとき、そのプルリクエストのソースブランチ上のワークフロー設定ファイルで issue_comment に設定されているワークフローが実行されることを期待するかもしれません。しかし、issue_comment イベントで GITHUB_SHA に設定されるのはデフォルトブランチの最新コミットです。なので、実際にはデフォルトブランチ上のワークフロー設定ファイルで issue_comment に設定されているワークフローが実行されます。actions/checkout アクションについても同様で、GITHUB_SHA を元にチェックアウトしてくるコミットが決まります。

イベントごとに GITHUB_SHA に設定される値は異なるので、どのコミットが設定されるかは前述したイベント一覧の公式ドキュメントを参照してください。issue_comment だけでなく、多くのイベントがデフォルトブランチの最新コミット上のワークフロー設定を使います。「2.4.10 スケジュールで定期実行」で説明した schedule イベントや、次に説明する repository_dispatch イベントも同様なので、注意が必要です。

2.7.3 repository_dispatch で外部からワークフロー実行

GitHub Actions は、リポジトリを直接操作しなくても、外部からワークフローを実行するためのエンドポイントを用意しています。そのエンドポイントにリクエストを投げると、repository_dispatch というイベントが発生します。

例として、ワークフロー側はリスト 2.66 のように記述します。

▼リスト 2.66　repository_dispatch

```
on: repository_dispatch

jobs:
  dispatch:
    runs-on: ubuntu-latest
    steps:
      run: echo "${{ toJson(github.event) }}"
```

そして、curl で次のようにリクエストを投げます。

```
$ curl \
    -X POST \
    --header "Authorization: token <PERSONAL_ACCESS_TOKEN>" \
    --data '{"event_type": "hello"}' \
    https://api.github.com/repos/<OWNER>/<REPO>/dispatches
```

PERSONAL_ACCESS_TOKEN には自身のパーソナルアクセストークン[*22]を設定してください。スコープには、repo（パブリックリポジトリのときは public_repo）が必要です。<OWNER> と <REPO> には、それぞれワークフローを設定しているリポジトリの所有者名とリポジトリ名を入れてください。

リクエストを送ると、ブラウザからワークフローが実行されていることを確認できます。

API パラメータ

```
POST https://api.github.com/repos/<OWNER>/<REPO>/dispatches
```

API は、次のパラメータを受け取ります。

- event_type: イベント名の文字列
  - 必須
- client_payload: イベントのペイロードに含まれるオブジェクト

event_type と client_payload を使うことで、リクエストを投げる側からワークフローに情報を渡すことができます。ワークフロー側は、github.event コンテキ

[*22] https://help.github.com/en/github/authenticating-to-github/creating-a-personal-access-token-for-the-command-line

ストから情報にアクセスできます。コンテキストについては、「2.4.11 コンテキストと式を使った条件実行」を参照してください。

2.8 ログによるコマンド実行

ワークフロー実行中に、echo などでログに特定のフォーマットの文字列を出力することで、次のようなことができます。

- 環境変数の設定
- アウトプットの設定
- PATH の追加
- debug メッセージの出力
- warning メッセージの出力
- error メッセージの出力
- ログのマスク
- コマンド実行の停止・再開

コマンドの実行は、次のような形式で文字列を出力する必要があります。標準出力でも標準エラー出力でもコマンドは実行されます。

```
$ echo "::<COMMAND> <PARAM1>=<VALUE1>,<PARAM2>=<VALUE2>::<COMMAND VALUE>"
```

例えば、環境変数の設定を行う set-env コマンドであれば次のように実行します。

```
$ echo "::set-env name=TZ::Asia/Tokyo"
```

この例では、set-env コマンドを実行し、name パラメータに TZ を、コマンドに Asia/Tokyo という値を渡しています。これによって、TZ という環境変数に Asia/Tokyo という値が設定されます。

ここからは、それぞれのコマンドについて解説していきます。

2.8.1 環境変数の設定: set-env

set-env コマンドは、環境変数を設定します。

```
$ echo "::set-env name=TZ::Asia/Tokyo"
```

name パラメータで環境変数の名前、コマンドの値で環境変数の値を設定します。設定した環境変数には、ジョブ内の次のステップからアクセスできるようになります。set-env コマンドを実行したステップではアクセスできないので注意が必要です。

2.8.2 アウトプットの設定: set-output

set-output コマンドは、ステップにアウトプットを設定します。

```
$ echo "::set-output name=result::true"
```

name パラメータでアウトプットの名前、コマンドの値でアウトプットの値を設定します。設定したアウトプットには、steps.<step_id>.outputs.<output_name> コンテキストからアクセスできます。コンテキストについては、「2.4.11 コンテキストと式を使った条件実行」を参照してください。

set-output コマンドは、アクションがアウトプットを設定するために使われることが多いです。アクションのアウトプットの定義については、「3.3 アクションのメタデータ」を参照してください。

2.8.3 PATH の追加: add-path

add-path コマンドは、PATH に新しいディレクトリを追加します。

```
$ echo "::add-path::${{ github.workspace }}"
```

コマンドの値が、PATH の先頭に追加されます。指定したディレクトリは、次のステップから PATH に反映されます。add-path コマンドを実行したステップでは PATH に反映されないので注意が必要です。

2.8.4 デバッグメッセージの出力: debug

debug コマンドは、デバッグメッセージをログに出力します。

```
$ echo "::debug::Debug Message"
```

コマンドの値が、デバッグメッセージとして出力されます。デバッグメッセージは、秘密情報で ACTIONS_STEP_DEBUG に true を設定しないと表示されません。ACTIONS_STEP_DEBUG に true を設定すると、GitHub Actions 組み込みのデバッグメッセージも出力されるようになります。秘密情報については、「2.4.5 秘密情報」を参照してください。

2.8.5 警告メッセージの出力: warning

warning コマンドは、警告メッセージをログに出力します。

```
$ echo "::warning::Warning Message"
```

コマンドの値が、警告メッセージとして出力されます。次のパラメータを指定することで、ブラウザ上でプルリクエストのファイル差分上にも警告メッセージを表示できます。

- file: ファイル名
- line: 指定したファイルの行数
- col: 指定した行の文字数

```
$ echo "::warning file=README.md,line=1,col=1::Warning Message"
```

静的解析ツールなどで、ファイルに対しての警告メッセージをプルリクエストのファイル差分上で見られるようにしたいという場面で使えます。

2.8.6 エラーメッセージの出力: error

error コマンドは、エラーメッセージをログに出力します。

```
$ echo "::error::Error Message"
```

コマンドの値が、エラーメッセージとして出力されます。次のパラメータを指定することで、ブラウザ上でプルリクエストのファイル差分上にもエラーメッセージを表

示できます。

- file: ファイル名
- line: 指定したファイルの行数
- col: 指定した行の文字数

```
$ echo "::error file=README.md,line=1,col=1::Error Message"
```

静的解析ツールなどで、ファイルに対してのエラーメッセージをプルリクエストのファイル差分上で見られるようにしたいという場面で使えます。

2.8.7 ログのマスク: add-mask

add-mask コマンドは、ログ上で特定の文字列をマスクするように設定します。

```
$ echo "::add-mask::password"
```

コマンドの値が、ログ上からマスクされるようになります。マスクされるのはコマンドを実行したタイミングからになり、コマンド実行自体のログもマスクされます。

コマンド実行前のログはマスクされません。また、ステップ名もマスクされないようなので、注意が必要です。

基本的には、秘密情報はログに出力されないように気をつけるべきで、このコマンドはあくまで保険として使うことをおすすめします。

2.8.8 コマンド実行の停止・再開: stop-commands

stop-commands コマンドは、実行するとここまで紹介したコマンドをコマンドとして処理しなくなり、単なる文字列として出力されるようになります。

```
$ echo "::stop-commands::stop"
```

stop-commands コマンドが実行された直後のログから、コマンドが処理されなくなります。

コマンドの値として指定した文字列を次のように使うと、コマンド実行の処理が再開されます。

```
$ echo "::stop::"
```

文字列のエスケープ

コマンド実行時に、文字列が次のような文字を含む可能性がある場合は、エスケープが必要です。

- % → %25
- \n → %0A
- \r → %0D

例えば、ENV_VAR という環境変数にエスケープを行う場合は、次のようになります。

```
$ ENV_VAR="${ENV_VAR//'%'/'%25'}"
$ ENV_VAR="${ENV_VAR//$'\n'/'%0A'}"
$ ENV_VAR="${ENV_VAR//$'\r'/'%0D'}"
```

set-output コマンドでアウトプットを設定する値が改行を含むようなときに問題になることがあるので、知っていると役に立ちます。

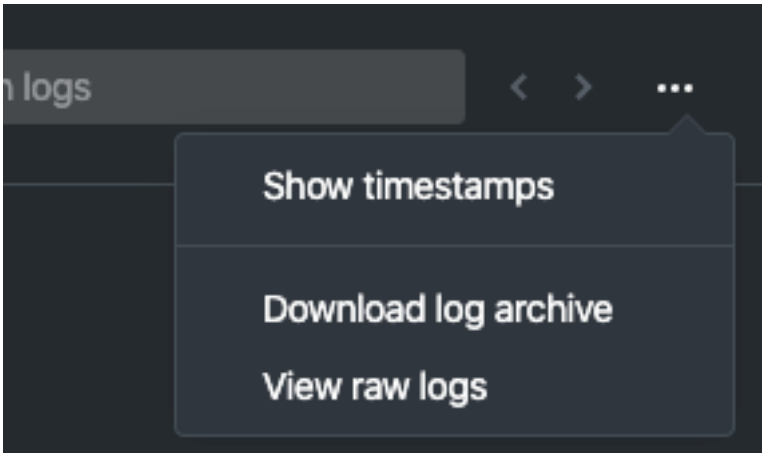
2.9 ワークフローのデバッグ

2.9.1 ステップのデバッグログ

「2.8.4 デバッグメッセージの出力: debug」で書いたように、秘密情報で ACTIONS_STEP_DEBUG に true を設定すると、ログにステップのデバッグ情報が出力されるようになります。

2.9.2 オプションメニュー

ジョブの右上のオプションメニューから、図 2.7 のようにワークフローの実行情報を取得することができます。



▲図 2.7　オプションメニュー

タイムスタンプの表示

"Show timestamps" をクリックすると、ブラウザ上のログにタイムスタンプが表示されるようになります。

ログのダウンロード

"Download log archive" をクリックすると、ワークフローのログが入った zip ファイルをダウンロードできます。

生のログの表示

"View raw logs" をクリックすると、現在選択しているジョブの生のログをブラウザ上で閲覧できます。

2.9.3　ランナーのデバッグログ

秘密情報で `ACTIONS_RUNNER_DEBUG` に `true` を設定すると、ランナーのデバッグ情報がログに出力されるようになります。ただし、このデバッグログはブラウザ上では表示されず、「2.9.2 オプションメニュー」でダウンロードできるログのアーカイブの中の "runner-diagnostic-logs" というディレクトリに含まれます。

2.10 権限

リポジトリの Actions タブの閲覧は、リポジトリの read 権限があれば誰でも可能です。read 権限があれば、アーティファクトを自由にダウンロードすることもできます。

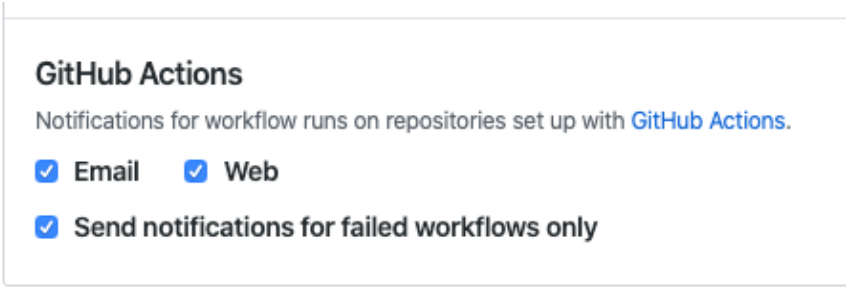
また、read 権限があるユーザーであれば fork してプルリクエストを作成することで、`pull_request` イベントによりリポジトリ上で任意のワークフローを実行させることが可能です。これによって、キャッシュ情報にも自由にアクセス可能です。しかし、「2.4.5 秘密情報」でも書いたとおり、fork したリポジトリから実行されたワークフローからは秘密情報にアクセスできず、`GITHUB_TOKEN` の write 権限もなくなります。

なので、基本的に認証情報などの見られてはいけない情報は秘密情報や外部サービスに置いておき、リポジトリ内やアーティファクト、キャッシュといったところには置かないようにしましょう。

ワークフローのキャンセルや再実行は、リポジトリの write 権限が必要です。

2.11 通知

ブラウザで GitHub の Settings → Notifications を開くと、図 2.8 のような設定があり、GitHub Actions の通知について設定できます。



▲図 2.8　GitHub Actions の通知設定

"Email" でメール通知、"Web" で Web Notifications の通知の ON/OFF を設定できます。"Send notifications for failed workflows only" を有効にするとワークフロー失敗時のみ通知が飛ぶようになります。

この設定画面にない、Slack に通知を飛ばしたいといった独自の通知を行いたい場合は、自前でワークフローの設定側に通知する処理を入れる必要があります。

2.12 ランナーのセルフホスティング

2.12.1 セルフホストランナーとは

GitHub Actions のワークフローの実行は、デフォルトでは GitHub が管理する環境上で実行されます。しかし、セルフホストランナーを使うことで、あなたは自身が管理するマシン、VM、コンテナ上などでワークフローを実行することができます。

セルフホストランナーを使うことでより柔軟な環境でワークフローのジョブを実行できます。例えば、次のようなケースで役に立ちます。

- CPU やメモリを激しく消費するジョブを実行したい
- プライベートネットワーク内でジョブを実行したい
- GitHub が提供していない OS 上でジョブを実行したい

セルフホストランナーは、リポジトリ単位、もしくは organization 単位で追加できます。リポジトリにセルフホストランナーをを追加すると、そのリポジトリからのみ追加されたセルフホストランナーを利用できます。organization にセルフホストランナーを追加すると、その organization 内の複数のリポジトリから追加されたセルフホストランナーを利用できます。

セルフホストランナーは OSS で、次のリポジトリ上で公開されています。

- https://github.com/actions/runner

GitHub が管理する環境との違い

GitHub が管理するランナーには、次のような特徴があります。

- 環境が自動で更新される
- GitHub が管理するのでユーザーは運用コストがかからない
- ジョブ実行ごとにクリーンな VM を使用する
- プライベートリポジトリだと料金がかかる

一方で、セルフホストランナーには、次のような特徴があります。

- 既存のローカルマシンやクラウドサービス上のリソースを利用できる
- 独自の要件（ハードウェア、OS、ソフトウェア、セキュリティ）に合わせて環境をカスタマイズできる

- 複数のジョブ実行間で環境を持ち越すことができる
- プライベートリポジトリでも完全無料
- 運用コストがかかる

基本的には、GitHub が管理する環境を使うほうがユーザーはお手軽かつ簡単にワークフローを実行できるでしょう。しかし、独自の環境でワークフローを動かさないといけない場合には、セルフホストランナーを利用することで、ユーザーはこれまで実現できなかったワークフローを実行できるようになります。例えば、大規模なリソースを使ったビルドを行ったり、プライベートネットワーク内へのデプロイといったことが可能になります。

動作環境

セルフホストランナーがサポートする OS は公式ドキュメントを参照してください。

- https://help.github.com/en/actions/hosting-your-own-runners/about-self-hosted-runners#supported-operating-systems-for-self-hosted-runners

Linux, macOS, Windows の主要な OS をサポートしています。

セキュリティ

セルフホストランナーは、GitHub と HTTPS で通信します。ランナーは、Long Polling という技術を使っており、GitHub 側からランナー方向へのリクエストは発生せず、ランナーから GitHub 方向へのリクエストのみ発生します。なので、ランナーを動かすネットワークに外部からアクセスできるように穴をあける必要はありません。また、ランナーと GitHub Actions 間のメッセージのやりとりは非対称暗号で暗号化されています。

しかし、パブリックリポジトリ上では、セルフホストランナーを使用することは推奨されていません。これは、プルリクエストを作成することによって、セルフホストランナーを動かしてるマシンで危険なコードを実行される恐れがあるからです。

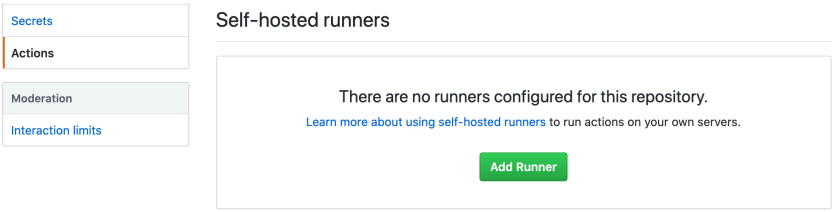
GitHub が管理する環境は毎回クリーンで隔離された VM 上で実行されて削除されるので、問題になりません。しかし、セルフホストランナーを動かす環境では、特にジョブ実行間でマシン環境が使い回される場合に、リスクが高まります。具体的には、次のようなリスクがあります。

- 悪意のあるコードを実行する
- ランナーが動作しているネットワークに外部からのアクセスを許す
- 危険なデータがマシン上に残り続ける

しっかりと隔離された環境を構築できる場合以外では、セルフホストランナーは利用者が限定されたプライベートリポジトリのみで使うことをおすすめします。

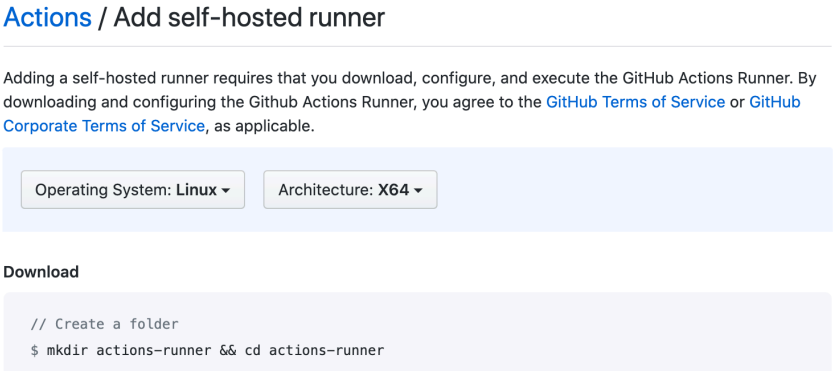
2.12.2 セルフホストランナーの起動

セルフホストランナーの起動方法は、ブラウザ上から確認できます。リポジトリにランナーを追加する場合、リポジトリの Settings タブを開き、サイドメニューの Actions を開くと図 2.9 のような画面が表示されます。



▲図 2.9　Settings → Actions

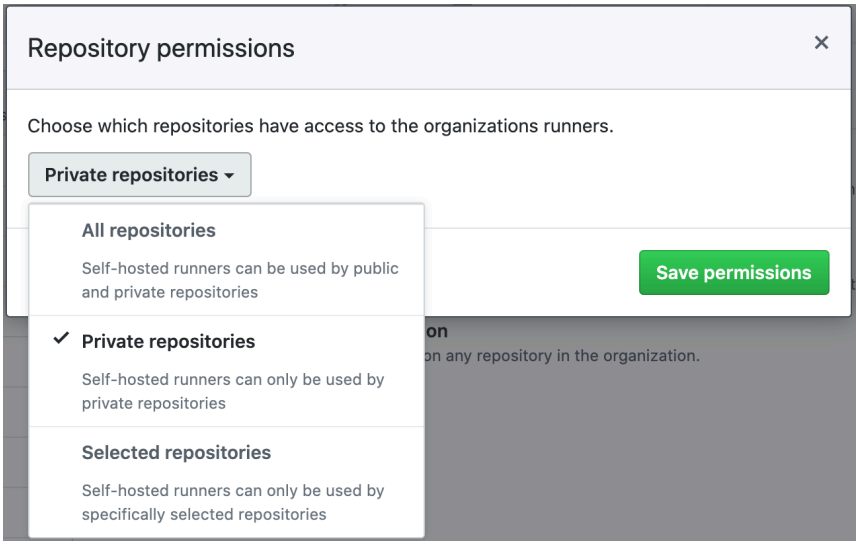
ここで、"Add Runner" ボタンをクリックすると、図 2.10 のように、各 OS やアーキテクチャでランナーを起動するための手順が表示されます。



▲図 2.10　セルフホストランナーの起動手順

organization にランナーを追加する場合、organization の Settings タブを開き、サイドメニューの Actions を開くと、同様の設定が可能です。organization は、図

2.11 のように、どのリポジトリがセルフホストランナーを利用できるか設定できます。



▲図 2.11　どのリポジトリがセルフホストランナーを利用できるか設定できる

次の 3 つの設定があります。

- All repositories: organization 内のすべてのリポジトリから利用できる
- Private repositories: organization 内のプライベートリポジトリからのみ利用できる
- Selected repositories: organization 内の選択したリポジトリからのみ利用できる

セルフホストランナーの起動に必要なトークン情報を取得するためには、それぞれ次の権限が必要です。

- リポジトリ: リポジトリの admin 権限、もしくは organization の owner 権限
- organization: organization の owner 権限

ここからは、リポジトリに macOS でランナーを追加してみます。

ダウンロード

まずは、最新バージョンのランナーをダウンロードします。ランナーを動かしたいディレクトリ上で次のコマンドを実行して、アーカイブのダウンロードと展開を行います。

```
$ mkdir actions-runner && cd actions-runner
$ curl -O -L https://github.com/actions/runner/releases/download/v2.169.1/
    actions-runner-osx-x64-2.169.1.tar.gz
$ tar xzf ./actions-runner-osx-x64-2.169.1.tar.gz
```

起動

次のコマンドでランナーを起動します。

```
$ ./config.sh --url https://github.com/<OWNER>/<REPO> --token <TOKEN>
$ ./run.sh
```

自身のリポジトリに対しての URL やトークンは、ブラウザ上で閲覧できる手順から確認してください。

`config.sh` を実行すると、図 2.12 のような画面が表示され、ランナー名やワークフォルダを設定できます。

▲図 2.12　config.sh 実行

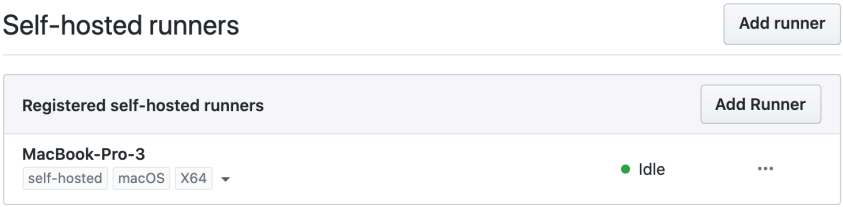
run.sh を実行して、次のように表示されれば成功です。

```
√ Connected to GitHub

2020-04-29 19:47:55Z: Listening for Jobs
```

ステータス確認

先ほどのコマンドラインの出力からもランナーが動いていることはわかりますが、ブラウザ上からもランナーのステータスを確認できます。Settings タブを開き、サイドメニューの Actions を開くと、図 2.13 のように追加されたランナーのステータスが表示されます。



▲図 2.13　ランナーのステータス

ランナーのステータスには、次の 3 つがあります。

- Idle: ランナーは GitHub に接続されていて、ジョブを実行する準備ができている
- Active: ランナーはジョブを実行中
- Offline: ランナーが GitHub に接続されていない
  - マシンが落ちている、ランナーが動いていない、ランナーが GitHub と通信できない、など

2.12.3　ワークフローでセルフホストランナーを使う

セルフホストランナーには、次のラベルが自動で追加されます。

- `self-hosted`: すべてのセルフホストランナーにデフォルトで適用される
- `linux`, `windows`, `macos`: OS によって適用される
- `x86`, `x64`, `ARM`, `ARM64`: CPU アーキテクチャの種類によって適用される

セルフホストランナー上でワークフローを実行するには、設定ファイルの `runs-on` でラベルを指定します。リスト 2.67 のように `self-hosted` ラベルを指定すると、リポジトリに登録されている任意のセルフホストランナー上でジョブが実行されます。

▼リスト 2.67　セルフホストランナー上でワークフロー実行

```
jobs:
  unit-test:
    ...
    runs-on: self-hosted
    ...
```

複数のラベルを組み合わせて指定することもできます。リスト 2.68 のように指定

すると、リポジトリに登録されている任意の macOS のセルフホストランナー上でジョブが実行されます。

▼リスト 2.68　macOS のセルフホストランナー上でワークフロー実行

```
jobs:
  unit-test:
    ...
    runs-on: [self-hosted, macos]
    ...
```

リスト 2.69 のように指定すると、リポジトリに登録されている任意の ARM64 の Linux のセルフホストランナー上でジョブが実行されます。

▼リスト 2.69　ARM64 の Linux のセルフホストランナー上でワークフロー実行

```
jobs:
  unit-test:
    ...
    runs-on: [self-hosted, ARM64, linux]
    ...
```

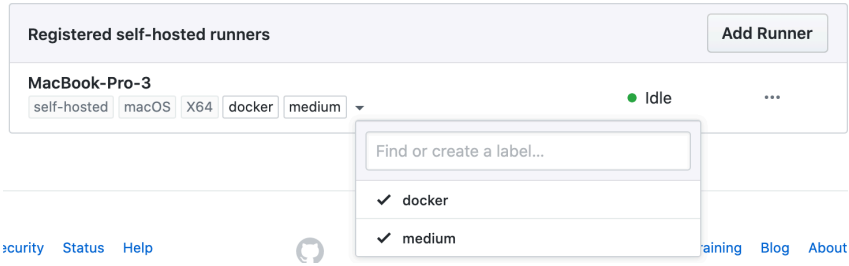
2.12.4　カスタムラベル

セルフホストランナーには、組み込みの OS と CPU アーキテクチャのラベル以外にも、ユーザーが任意のカスタムラベルを設定することができます。これにより、用途に応じてセルフホストランナーを柔軟に使い分けることができるようになります。

「起動」で実行した `config.sh` で対話的に入力するか、`--labels` オプションで指定することによって、カスタムラベルを設定できます。`--labels` オプションはカンマ区切りで複数のラベルを指定でき、次の例では `docker` ラベルと `medium` ラベルを設定しています。

```
$ ./config.sh --labels docker,medium --url ... --token ...
```

また、「ステータス確認」の画面から、図 2.14 のようにカスタムラベルを追加・削除することも可能です。



▲図 2.14　ブラウザ上からカスタムラベルを追加・削除

カスタムラベルも組み込みのラベルと同様に、設定ファイルの `runs-on` で指定できます。

2.12.5　セルフホストランナーの削除

セルフホストランナーをリポジトリまたは organization から削除する場合、ブラウザ上とコマンドライン上から削除できます。ブラウザ上からは、ステータス確認の図 2.13 と同じところに "Remove" ボタンが存在します。

コマンドラインからは、次のコマンドのように `config.sh remove` に起動時と同じトークンを渡すことで削除可能です。（ちなみに、このトークンは一定時間で期限切れとなるので、エラーとなった場合はまたブラウザ上からトークンを取得し直してください）

```
$ ./config.sh remove --token <TOKEN>
```

コマンドラインを実行すると、次の処理が行われます。

- リポジトリまたは organization からランナーを削除
- ランナーの設定ファイルをマシン上から削除
- サービスがインストールされている場合は削除

なお、2020/05/20 から、30 日以上 GitHub Actions への接続がないセルフホストランナーは、自動で GitHub から削除されます。[*23]

2.12.6 proxy

ランナーに proxy 経由で通信を行ってほしい場合、次の環境変数を設定します。

- https_proxy: HTTPS 通信を経由する proxy の URL を指定する
- http_proxy: HTTP 通信を経由する proxy の URL を指定する
- no_proxy: proxy を経由しないホストをカンマ区切りで指定する

ファイル経由でも proxy を指定できます。ランナーのアーカイブを展開したディレクトリに `.env` ファイルを置き、リスト 2.70 のように記述します。

▼リスト 2.70　.env

```
https_proxy=http://proxy.local:8080
http_proxy=http://proxy.local:8080
no_proxy=example.com,myserver.local:443
```

Docker コンテナ内でランナーを動かす場合、コンテナの proxy 設定は Docker の公式ドキュメントを参照してください。

- `https://docs.docker.com/network/proxy/`

2.12.7 ランナーのサービス化

ランナーのアーカイブを展開したディレクトリに存在する `svc.sh` を使うことで、ランナーをサービスとして動かせます。

まず、次のコマンドでサービスをインストールします。（`run.sh` でランナーを起動中だったら事前に停止してください）

```
$ ./svc.sh install
```

[*23] https://github.blog/changelog/2020-04-20-github-actions-cleanup-dormant-self-hosted-runners/

次のコマンドでサービスを起動します。

```
$ ./svc.sh start
```

次のコマンドでサービスのステータスを表示します。

```
$ ./svc.sh status
```

次のコマンドでサービスを停止します。

```
$ ./svc.sh stop
```

次のコマンドでサービスをアンインストールします。

```
$ ./svc.sh uninstall
```

Enterprise Cloud で許可 IP アドレスを設定した場合

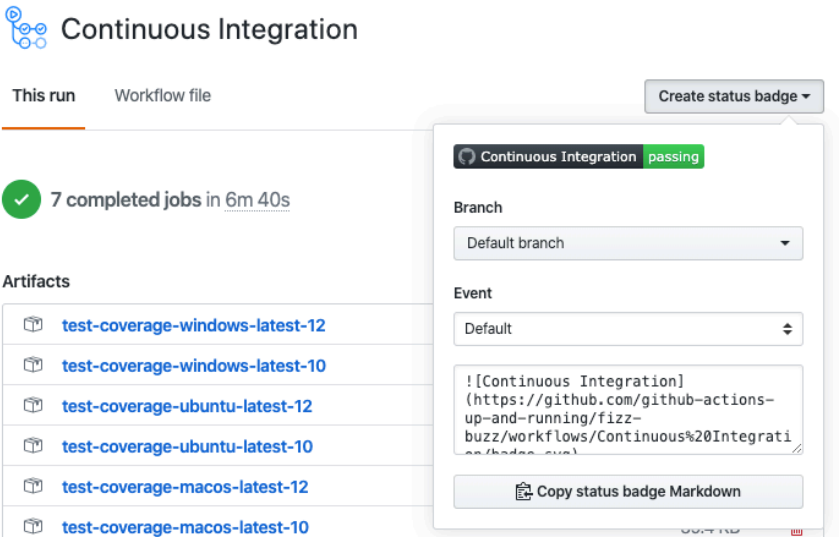
Enterprise Cloud プランでは、許可 IP アドレスを設定する機能があります[*24]。この機能により Enterprise アカウント下のリポジトリにアクセスできる IP アドレスを制限できます。例えば、会社のネットワーク内からのみアクセスを許可するようなことが可能です。

この許可 IP アドレス機能を使用する場合、GitHub Actions は GitHub が管理するランナーでは実行できなくなるようです。許可した IP レンジ内の IP を付与した環境上に、セルフホストランナーを構築して使わなければなりません。

[*24] https://help.github.com/en/github/setting-up-and-managing-your-enterprise-account/enforcing-security-settings-in-your-enterprise-account#managing-allowed-ip-addresses-for-organizations-in-your-enterprise-account

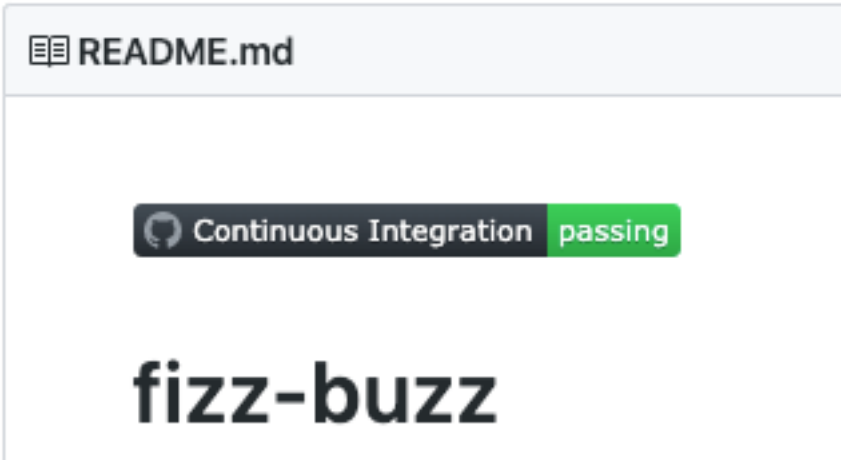
2.13　バッジ

GitHub Actions には、ワークフローの最新のステータスを表示するためのバッジが用意されています。ブラウザ上で Actions タブからワークフローを開くと、図 2.15 の "Create status badge" のところから、バッジを表示するための markdown がコピーできます。



▲図 2.15　ブラウザ上からバッジの markdown 取得

この markdown をリポジトリトップの README.md に埋め込むと、図 2.16 のようにバッジが表示されます。



▲図 2.16　ワークフローのステータスバッジ

バッジの URL は、次のようになります。

```
https://github.com/<OWNER>/<REPOSITORY>/workflows/<WORKFLOW_NAME>/badge.svg
```

`WORKFLOW_NAME` はワークフローの `name` を URL エンコードした文字列になります。`name` を省略している場合は、代わりにワークフロー設定ファイルのパス（`.github/workflows/ci.yml` など）が入ります。

また、バッジの URL はパラメータとして次の値を指定することができます。

- branch: 指定したブランチ名の最新のステータスを表示
- event: 指定したイベントに絞って最新のステータスを表示

2.14　REST API

GitHub Actions には REST API が用意されています。詳細については公式ドキュメントを参照してください。

- `https://developer.github.com/v3/actions/`

REST API で次のようなことが可能です。

- アーティファクトの情報取得、ダウンロード、削除
- 秘密情報の設定、削除
- セルフホストランナーのステータスの取得
- ワークフローの実行情報の取得、再実行、キャンセル

2.15 制限事項

GitHub Actions の利用については、次の制限があります。

- リポジトリごとに同時実行できるワークフローの数は最大で 20 まで
- ワークフロー内の各ジョブの実行時間は最大で 6 時間まで
- アカウントごとに全リポジトリで同時実行できるジョブの数はプランによって異なる
  - 無料プラン: 20（macOS ジョブは 5）
  - Pro: 40（macOS ジョブは 5）
  - Team: 60（macOS ジョブは 5）
  - Enterprise: 180（macOS ジョブは 15）

この制限が適用されるのは GitHub が提供するランナーのみで、セルフホストランナーの場合は制限はありません。

さらに、当たり前ですが暗号通貨採掘のような本来使われるべきでない目的での利用も明確に禁止されています。

2.16 まとめ

本章では、ワークフローの設定方法や、GitHub Actions が持つ機能について解説しました。

GitHub Actions が持つ機能は多岐にわたるので、最初からすべての機能を完全に理解する必要はありません。あらためて必要になったタイミングで本書を読み直してもらえれば幸いです。

次章では、アクションの詳細や作り方について詳しく解説します。

第 3 章

アクション

本章では、アクションの作り方を中心に、アクションについて解説します。

3.1 アクションとは

GitHub Actions では、**アクション**という単位で実行可能なタスクを作成できます。前章までは既存の公開されているアクションを利用するだけでしたが、GitHub Actions ではユーザーが自由にアクションを作って公開することが可能です。

3.1.1 アクションの種類

アクションには 2 種類存在し、表 3.1 にある通りそれぞれ利用可能な OS が異なります。

▼表 3.1 アクションの種類

種類	利用可能な OS
JavaScript	Linux, macOS, Windows
Docker コンテナ	Linux

JavaScript アクションは、GitHub が提供するすべての VM 環境から直接実行できます。Docker コンテナアクションと比べて、イメージのダウンロードやビルド、コンテナの起動コストが発生しないので、実行が高速です。

Docker コンテナアクションは、GitHub が提供する環境では、OS が Linux の場合のみ実行できます。他のプロセスとは隔離された環境を作れるというコンテナの性質上、いつも同じ環境でアクションを実行したいという場合には、Docker コンテナアクションのほうがおすすめです。しかし、イメージのダウンロードやビルド、コンテナの起動コストがあるので、JavaScript アクションよりは遅くなりがちです。

3.1.2 アクションの保存場所

公開するアクションの場合、他のソフトウェアとは別にアクションのためのリポジトリを作成して、ソースコードを管理することが推奨されています。これは、他のソフトウェアとは別にアクションのリリースサイクルを管理しやすくなるからです。また、利用者から見てもわかりやすいです。

非公開なアクションの場合は、そのアクションを利用するリポジトリ内にソースコードを保存します。リポジトリ内のどこのディレクトリに保存してもいいですが、`.github/actions` ディレクトリ下にアクションごとにディレクトリを作成して保存することが推奨されています。例えば、`.github/actions/action-a`、`.github/actions/action-b` のようになります。

3.1.3 アクションのバージョン指定

「2.4.2 ワークフローとジョブの設定」の `uses` でも説明しましたが、公開されているアクションを利用するときは、次の 3 つの方法でアクションのバージョンを指定できます。

▼表 3.2　アクションのバージョン指定

参照方法	例
commit SHA	actions/setup-node@1c5c137
タグ	actions/setup-node@v1.4.0
ブランチ	actions/setup-node@master

アクション作成者は、タグ作成時にセマンティックバージョニング[*1]に従って作成することが推奨されています。

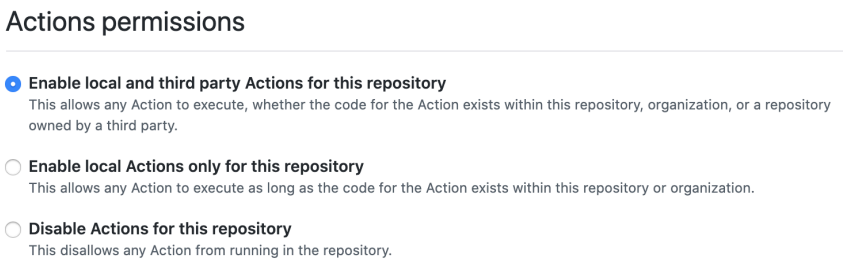
「2.4.2 ワークフローとジョブの設定」のコラム「アクションのバージョンをどの方法で指定するべきか」でも書きましたが、アクションを利用する側は、commit SHA 方式で参照するのが一番安全です。というのも、タグやブランチは上書き可能だからです。特に、信頼性の低いサードパーティーのアクションを利用するときは、安全性を確認した時点での commit SHA を指定して利用するのが無難です。公式や自分が作った信頼性の高いアクションはタグ指定、それ以外は commit SHA 指定ぐらいがほどよい運用かと思います。

[*1] `https://semver.org/`

3.1.4 利用できるアクションの制御

アクションはとても便利ですが、セキュリティやコンプライアンスの関係などで利用を制限したいケースがあります。こういった場合、リポジトリまたは organization 単位で、利用できるアクションの制御を行うことが可能です。

リポジトリまたは organization の Settings タブを開き、サイドバーの Actions をクリックすると、図 3.1 のような設定があります。



▲図 3.1　利用できるアクションの制御

次のような制御が可能です。

- Enable local and third party Actions for this repository (organization)
  - すべてのアクションを利用可能（デフォルト）
- Enable local Actions only for this repository (organization)
  - リポジトリが存在する organization 内のアクションのみ利用可能
- Disable Actions for this repository (organization)
  - すべてのワークフローが実行されなくなる

"Enable local Actions only ..." を設定すると、`actions/checkout` のような公式アクションも使えなくなります。なので、チェックアウトするためには `actions/checkout` をリポジトリが存在する organization 下に fork するような工夫が必要になります。

"Disable Actions ..." を設定すると、アクションというよりワークフロー自体が実行されなくなります。GitHub Actions を完全に使わせたくないような場合に設定するものと思われます。

3.2 JavaScript アクション

3.2.1 JavaScript アクションを作成

まずは、JavaScript アクションの作り方を理解するために、簡単なアクションを作成してみます。「3.7 公式テンプレートリポジトリ」で説明する公式テンプレートを使うと楽ですが、ここではアクションの作り方を学ぶために一から作成してみます。

個人アカウント（もしくは organization）下に新しくリポジトリを作成し、リスト 3.1 のような `action.yml` ファイルを作成します。

▼リスト 3.1　action.yml

```
name: "Hello World"
runs:
  using: "node12"
  main: "index.js"
```

詳細は「3.3 アクションのメタデータ」で解説しますが、この `action.yml` はアクションのメタデータを定義しています。`name` でアクションの名前を定義しています。`runs` の `using` で JavaScript アクションであることを、`main` でアクション実行時に読み込まれる JavaScript ファイルを定義しています。

そして、リスト 3.2 のような、"Hello, World!" を出力するだけの `index.js` ファイルを作成します。

▼リスト 3.2　index.js

```
console.log("Hello, World!");
```

これは、`action.yml` の `main` で指定した JavaScript ファイルです。ワークフロー内でこのアクションが実行されたときに、この `index.js` がワークフロー実行環境の VM 上で実行されます。

動作確認のために、リスト 3.3 の内容で `.github/workflows/test.yml` を作成します。

▼リスト 3.3　.github/workflows/test.yml

```
name: Test
on: push

jobs:
  test:
    runs-on: ubuntu-latest
    name: Test
    steps:
      - name: Checkout
        uses: actions/checkout@v2.0.0
      - name: Verify hello action works
        uses: ./
```

uses は、action.yml が存在するディレクトリへの相対パスを指定してアクションを実行することが可能です。なので、actions/checkout を実行した後に、uses : ./ を指定すれば作成したアクションを実行して動作確認することが可能です。

この時点で、リポジトリのディレクトリ構造はリスト 3.4 のようになるはずです。

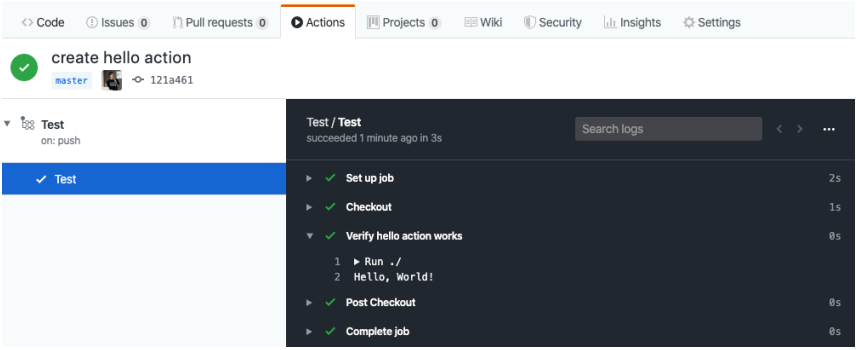
▼リスト 3.4　JavaScript アクションのディレクトリ構造

```
(repository root)
|-- .github
|   `-- workflows
|       `-- test.yml
|-- README.md
|-- action.yml
`-- index.js
```

ファイルを作成できたら、次のコマンドですべてのファイルをリポジトリに push します。

```
$ git add .
$ git commit -m "create hello action"
$ git push origin master
```

ブラウザから Actions タブを開くと、図 3.2 のように、作成したアクションがワークフロー内で実行されていることを確認できます。



▲図 3.2　JavaScript アクションの動作確認

ここまでで作成したアクションのサンプルは、次のリポジトリで公開しています。

- `https://github.com/github-actions-up-and-running/hello-action`

3.2.2　JavaScript アクションを使う

今度は、作成したアクションを別リポジトリから実行できることを確認します。そのために、まずはアクションを作成したリポジトリで v1.0.0 タグを作成します。

```
$ git tag v1.0.0
$ git push origin v1.0.0
```

次に、JavaScript アクションを使う側として、新しいリポジトリを作成します。そして、リスト 3.5 の内容で `.github/workflows/hello.yml` を作成します。

▼リスト 3.5　.github/workflows/hello.yml

```
name: Run Hello Action
on: push

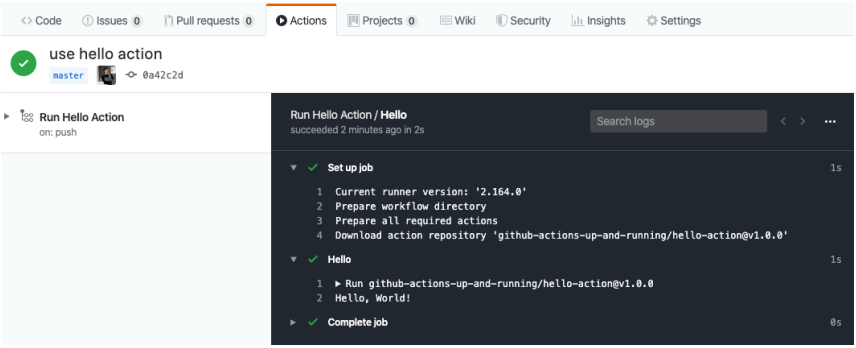
jobs:
  hello:
    runs-on: ubuntu-latest
    name: Hello
    steps:
      - name: Hello
        uses: github-actions-up-and-running/hello-action@v1.0.0
```

今度は、uses でリポジトリとタグを指定しています。このとき、アクション実行のためにチェックアウトは必要なくなり、ワークフロー実行時に対象のリポジトリの中身が自動でダウンロードされます。

ファイルを作成したら、次のコマンドでリポジトリに push します。

```
$ git add .github/workflows/hello.yml
$ git commit -m "use hello action"
$ git push origin master
```

ブラウザから Actions タブを開くと、図 3.3 のように、ワークフロー内でアクションが実行されることを確認できます。



▲図 3.3　JavaScript アクションの利用

ログを見ると、"Set up job" の段階で、作成したアクションがリポジトリからダウンロードされていることがわかります。

ここまでで、"Hello, World!" と表示するだけの簡単なアクションが作成＆公開でき、他のリポジトリからも利用可能になったことを確認できました。作成したアクションは、必要最低限の簡単なものです。実際にはこれに加えて、アクションにパラメータを渡したり、次以降のステップのために値を出力したり、依存ライブラリを扱ったりといったことが必要になってきます。そのあたりの詳細について、ここから先で説明していきます。

3.3 アクションのメタデータ

アクションには、YAML 形式で書かれたメタデータファイルが必要です。ファイル名は、`action.yml` で作成します[*2]。ここでは、メタデータで定義できる情報について説明します。

name

アクション名を文字列で定義します。GitHub Marketplace に公開する場合は必須で、既存のアクション、ユーザー、organization と名前が重複してはいけません。

author

アクションの作成者名を文字列で定義します。

description

アクションの説明文を文字列で定義します。GitHub Marketplace に公開する場合は必須で、125 文字以内で書く必要があります。

inputs

アクション実行時に、アクションに渡すパラメータの情報をマップで定義します。アクションの利用者は、ワークフロー定義ファイル内で `with` でパラメータを指定します。ワークフローの設定ファイル側についての詳細は、「2.4.2 ワークフローとジョブの設定」を参照してください。

`actions/setup-node` だと、リスト 3.6 のように `node-version` パラメータの情報を定義しています。

▼リスト 3.6　inputs の例

```
inputs:
  node-version:
    description: 'Version Spec of the version to use. Examples: 10.x,
      10.15.1, >=10.15.0'
    required: false
    default: '10.x'
```

[*2] 公式ドキュメントでは `action.yaml` でもいいと書かれていますが、動きませんでした

inputs.<input_id>

パラメータの ID を文字列で定義します。パラメータの ID 同士は重複してはいけません。アルファベットか _ で始まる文字列で、使える文字列はアルファベット、数字、-、_ のみです。

パラメータは、パラメータ ID を大文字にして、先頭に INPUT_ を付けた環境変数としてアクションに渡されます。例えば、node-version だとしたら、INPUT_NODE-VERSION という環境変数に値が設定されます。

パラメータ ID は、小文字が推奨されています。

inputs.<input_id>.description

パラメータの説明文を文字列で定義します。

inputs.<input_id>.required

パラメータが必須かどうかを boolean で定義します。省略した場合は、デフォルトで false になります。

inputs.<input_id>.default

パラメータのデフォルト値を文字列で定義します。アクションの利用者側でパラメータを指定しなかったときにこの値が使われます。

outputs

このアクション以降のステップで利用するためのアウトプットの情報をマップで定義します。

設定したアウトプットには、steps.<step_id>.outputs.<output_name> コンテキストからアクセスできます。コンテキストについては、「2.4.11 コンテキストと式を使った条件実行」を参照してください。

outputs で定義しなかった ID でアウトプットを設定しても、次以降のステップで利用することが可能です。アウトプットの設定方法については、「2.8.2 アウトプットの設定: set-output」を参照してください。

actions/cache だと、リスト 3.7 のように cache-hit パラメータの情報を定義しています。

▼リスト 3.7　outputs の例

```
outputs:
  cache-hit:
    description: 'A boolean value to indicate an exact match was found
      for the primary key'
```

outputs.<output_id>

アウトプットの ID を文字列で定義します。アウトプットの ID 同士は重複してはいけません。アルファベットか _ で始まる文字列で、使える文字列はアルファベット、数字、-、_ のみです。

outputs.<output_id>.description

アウトプットの説明文を文字列で定義します。

runs

アクションが実行する処理についての情報をメタデータで定義します。必須です。

runs.using

アクションの種類を文字列で定義します。現時点では、次の 2 種類を指定できます。

- `node12`: JavaScript アクション
- `docker`: Docker コンテナアクション

JavaScript アクションが使用する Node.js のバージョン

現時点だと、`action.yml` の `runs` の `using` に指定できるのは、JavaScript アクションでは `node12` だけです。これを指定すると、Node.js のバージョンは 12 系が使用されます。

仮に、アクションを呼び出す前に `actions/setup-node` でワークフロー実行環境の Node.js のバージョンを変更していても、アクションの実行時には Node.js の 12 系が使用されます。アクションの実行に使用される Node.js は、ある程度は環境と切り離されてると考えていいようです。

なので、JavaScript アクション開発時は、Node.js 12 系をローカルの開発環境やテスト環境に用意しましょう。

runs.main

実行する JavaScript ファイルへのパスを定義します。JavaScript アクションでのみ必須です。

runs.image

Docker コンテナアクションで使用する Docker イメージを文字列で定義します。Docker コンテナアクションでのみ必須です。

指定方法として、リスト 3.8 のようにリポジトリ内の Dockerfile へのパスか、リスト 3.9 やリスト 3.10 のようにパブリックな Docker Hub や Docker レジストリのイメージへの URL を指定できます。リポジトリ内の Dockerfile へのパスを指定するときは、メタデータからの相対パスを指定します。

▼リスト 3.8　リポジトリ内の Dockerfile を指定

```
runs:
  using: docker
  image: Dockerfile
```

▼リスト 3.9　Docker Hub 上のイメージを指定

```
runs:
  using: docker
  image: docker://alpine:3.11
```

▼リスト 3.10　パブリックな Docker レジストリ上のイメージを指定

```
runs:
  using: docker
  image: docker://gcr.io/cloud-builders/gcloud
  args:
    - version
```

runs.env

Docker コンテナに渡す環境変数をマップで定義します。キーが環境変数名、値が環境変数の値になります。Docker コンテナアクションでのみ指定できます。

runs.entrypoint

Dockerfile の `ENTRYPOINT` を文字列で定義します。Docker コンテナアクションでのみ指定できます。

runs.args

Dockerfile の引数を文字列のリストで定義します。Docker コンテナアクションでのみ指定できます。リスト 3.11 のように書けます。

▼リスト 3.11　args

```
runs:
  using: docker
  image: Dockerfile
  args: ['Hello', 'World']
```

branding

アクションを Marketplace に公開したときに、Marketplace 上で表示されるアイコンの情報を定義します。

branding.icon

アイコンの名前を文字列で定義します。アイコン名は、Feather[*3] のアイコン名を指定します。

指定できるアイコン名は、公式ドキュメントを参照してください。

[*3] `https://feathericons.com/`

- https://help.github.com/en/actions/building-actions/metadata-syntax-for-github-actions#icon

branding.color

アイコンの背景色を文字列で定義します。次の色を指定できます。

- `white`
- `yellow`
- `blue`
- `green`
- `orange`
- `red`
- `purple`
- `gray-dark`

3.3.1 公式ドキュメントに載ってないメタデータ

ここまでは、公式ドキュメントに書かれてる設定のみを紹介してきました。しかし、公式のアクションのメタデータファイルを見てると、公式ドキュメントに書かれていない設定がいくつかあります。今後変更になる可能性もあるので、使う場合は注意して使ってください。

- `inputs.<input_id>.deprecationMessage`
  - deprecated なパラメータを使ったときに警告を表示する
- `runs.post`
  - ジョブ完了後に実行される後処理用の JavaScript ファイル
- `runs.post-if`
  - `runs.post` を実行する条件文（例: `success()`）

3.4 actions/toolkit

JavaScript アクションを作りやすくするために、`actions/toolkit` という公式リポジトリで、複数の Node.js パッケージが公開されています。

- `https://github.com/actions/toolkit`

詳細については各パッケージの `README.md` を見てもらうのが一番いいですが、簡

単に説明すると次のような機能を持つパッケージが存在します。

- `@actions/artifact`: アーティファクトのアップロード、ダウンロード
- `@actions/core`: パラメータの読み込み、アウトプットの設定、環境変数の出力、シークレットの設定、PATH の追加、終了ステータスの変更、ログ、アクションのステート管理
- `@actions/exec`: コマンドライン実行
- `@actions/github`: `github` コンテキストへのアクセス、proxy 設定を読み込む Octokit[*4] クライアント
- `@actions/glob`: glob パターンにマッチするファイル一覧を検索
- `@actions/io`: ファイルシステム操作
- `@actions/tool-cache`: ダウンロード、展開、キャッシュ（ジョブ間のキャッシュではなく、そのジョブ内のみで使えるキャッシュ）
  - 名前がややこしいけど、「2.5 キャッシュ」で出てくるキャッシュとは完全に別物です

3.5 JavaScript アクションの依存関係の管理

JavaScript アクションは、実行時に指定したリポジトリのソースコードをまるごとダウンロードして利用します。このとき、`npm install` などのコマンドは実行されません。なので、Node.js の依存ライブラリがダウンロードされる `node_modules` ディレクトリも、JavaScript アクションのリポジトリに含める必要があります。

しかし、`node_modules` ディレクトリをリポジトリに含めてしまうと、コミットの差分が増えて、開発がしづらくなってしまいます。そこで、`@zeit/ncc` という Node.js パッケージを使うと、すべての依存関係を 1 つの JS ファイルにまとめることができるので、`node_modules` ディレクトリをリポジトリに含めなくてすみます。

3.5.1 @zeit/ncc

`@zeit/ncc` は、Node.js プロジェクトを一つの .js ファイルにコンパイルするパッケージです。作成してる Zeit 社は、Zeit Now という PaaS を開発しており、React フレームワークの Next.js を開発してる会社でもあります。

次のようにコマンド実行すると、`index.js` の依存関係をすべて含んだ一つの `dist/index.js` というファイルにコンパイルしてくれます。

[*4] `https://octokit.github.io/rest.js/`

```
$ npm install --global @zeit/ncc
$ ncc build index.js
```

node_modules の代わりに dist/index.js をリポジトリに含めれば、余分なファイルは一つにまとまり、ノイズがだいぶ減ります。

@zeit/ncc は、TypeScript もサポートしています。

「3.7 公式テンプレートリポジトリ」の JavaScript と TypeScript の公式テンプレートでも @zeit/ncc は使われています。なので、JavaScript アクションを作る場合は、@zeit/ncc を使うのが公式でも推奨と考えて問題ないと思います。

3.6 README.md

パブリックなアクションを保存するリポジトリの README.md には、次の情報を含めることが推奨されています。

- アクションがなにをするかについての詳細な説明
- inputs と outputs についての説明
- アクションが使用する秘密情報についての説明
- アクションが使用する環境変数についての説明
- アクションのワークフロー内での使用例

公式が提供しているアクションのサンプルリポジトリを見るとわかりやすいので、おすすめです。

3.7 公式テンプレートリポジトリ

GitHub には、リポジトリをテンプレート化して、同様の構成のリポジトリを作成しやすくするための機能があります[*5]。

GitHub Actions にも、アクション作成用のテンプレートリポジトリがいくつか用意されています。ここでは、次の 3 種類のテンプレートリポジトリを紹介します。

- actions/javascript-action
- actions/typescript-action
- actions/container-action

[*5] https://help.github.com/en/github/creating-cloning-and-archiving-repositories/creating-a-template-repository

3.7.1 actions/javascript-action

- `https://github.com/actions/javascript-action`

JavaScript アクションを作成するためのテンプレートリポジトリです。このテンプレートリポジトリからリポジトリを作成することで、アクションのメタデータ、テストや Lint、動作確認のワークフローといったものが設定された状態から始めることができます。

3.7.2 actions/typescript-action

- `https://github.com/actions/typescript-action`

TypeScript[*6] で JavaScript アクションを作成するためのテンプレートリポジトリです。`actions/javascript-action` に加えて、TypeScript のコンパイルが設定された状態から始めることができます。

3.7.3 actions/container-action

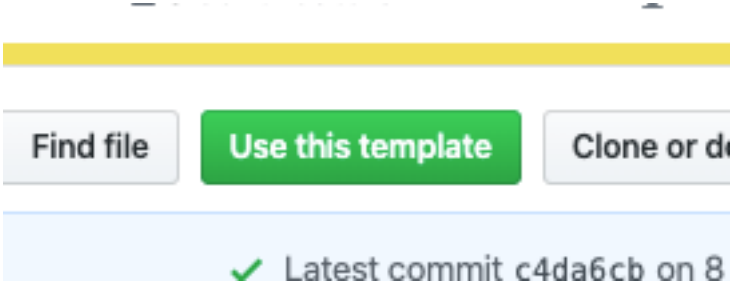
- `https://github.com/actions/container-action`

Docker コンテナアクションを作成するためのテンプレートリポジトリです。本書執筆時点では、`Dockerfile` やメタデータなどの最低限のファイルしか置かれておらず、少しメンテナンスされてない感じがします。

3.7.4 テンプレートリポジトリの使い方

ブラウザで各テンプレートリポジトリを開くと、図 3.4 のように "Use this template" ボタンがあります。

[*6] JavaScript に静的型付けとクラスベースオブジェクト指向を追加した言語



▲図 3.4　テンプレートリポジトリからリポジトリ作成

これをクリックすると、テンプレートリポジトリからリポジトリを作成できます。あとは、自分の作りたいアクションに合わせてリポジトリの中身を修正していきましょう。

慣れてきたら、自分用のテンプレートリポジトリを作成しておくのもいいと思います。

3.8　TypeScript アクション

ここまでの情報を元に、「3.2 JavaScript アクション」よりもう少し実践的なアクションを TypeScript で作成してみます。TypeScript でコードを書きますが、最終的にはコンパイルされて JavaScript アクションとなります。

ここでは、`pull_request` イベント時に、プルリクエストにコメントを書き込むためのアクションを作成します。

3.8.1　TypeScript アクションを作成

まず、「3.7.2 actions/typescript-action」で説明した公式テンプレートリポジトリから新たにリポジトリを作成します。

そして、`action.yml` をリスト 3.12 のように修正します。

▼リスト 3.12　action.yml

```
name: 'PR Comment'
description: 'Post a comment on PR'
author: 'miyajan'
inputs:
  repo-token:
    description: 'GITHUB_TOKEN secret'
    required: true
  message:
    description: 'Message to comment'
    required: true
outputs:
  comment-url:
    description: 'The PR comment URL'
runs:
  using: 'node12'
  main: 'dist/index.js'
branding:
  icon: 'message-square'
  color: 'green'
```

inputs の repo-token で GitHub の認証情報トークンを、message でプルリクエストに書き込むコメントの内容を指定します。書き込んだコメントへの URL が outputs の comment-url に保存され、次以降のステップで使えるようにする想定です。author は各自に合わせて書き換えてください。

次に、テンプレートリポジトリに含まれてない、アクションに必要なパッケージを次のように npm install します。

```
$ npm install --save @actions/github @octokit/webhooks
$ npm install --save-dev nock
```

@actions/github は、GitHub の REST API を使ってプルリクエストにコメントを投稿するために使います。@octokit/webhooks は、イベントのペイロードに型を付けるために使います。nock は、ユニットテストで REST API をモックするために使います。

package.json がテンプレートリポジトリのままだと実態に合わないので、リスト 3.13 のようにアクションに合わせて修正します。

▼リスト 3.13　package.json

```
{
  "name": "pr-comment",
  "version": "1.0.0",
  "private": true,
  "description": "GitHub Action to comment on PR",
  "main": "lib/main.js",
  "scripts": {
    ...
    "all": "npm ci && npm run build && npm run format && npm run lint &&
            npm run pack && npm test"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com//github-actions-up-and-running/
            pr-comment.git"
  },
  "keywords": [
    "actions",
    "node"
  ],
  "author": "miyajan",
  "license": "MIT",
  "dependencies": {
    ...
  },
  "devDependencies": {
    ...
  }
}
```

scripts の all の最初に、npm ci の実行を追加しています。これは、ncc で生成されるファイルに差分が発生してしまうことを防ぐためです。npm run all 実行時に、ncc で dist/index.js として 1 つのファイルにコンパイルされた JavaScript ファイルが生成されます。しかし、npm install と npm ci のどちらで依存関係をインストールしたかによって、依存関係は変わっていないのに ncc で生成されるファイルに差分が発生することがあるのです。(npm の内部的な話になってしまうのですが、手元で確認したところ、npm install と npm ci では、node_modules 下の package.json に追加されるメタデータに違いが発生することがあることがわかりました。npm ci は、node_modules が存在するときは一回削除した上で新しく依存関係をダウンロードし直します。なので、npm run all 実行時に最初に npm ci を実行して、依存関係が変わらなければ node_modules 下が不変になるようにしています。)

repository や author は、各自に合わせて修正してください。

次に、src/main.ts をリスト 3.14 のように修正します。

▼リスト 3.14　src/main.ts

```
import {run} from './run'

run()
```

テンプレートリポジトリだと src/main.ts 内に run 関数が書かれていますが、ここでは src/run.ts から読み込んで実行するようにしました。これは、あとで run 関数のテストコードを書くためです。

src/run.ts は新たに作成し、リスト 3.15 のように書きます。

▼リスト 3.15　src/run.ts

```
import * as core from '@actions/core'
import * as github from '@actions/github'
import * as Webhooks from '@octokit/webhooks'

export async function run(): Promise<void> {
  try {
    // The pull_request exists on payload when a pull_request event is
      triggered.
    // Sets action status to failed when pull_request does not exist on
      payload.
    const pr = github.context.payload
      .pull_request as Webhooks.WebhookPayloadPullRequest
    if (!pr) {
      core.setFailed('github.context.payload.pull_request not exist')
      return
    }

    // Get input parameters.
    const token = core.getInput('repo-token')
    const message = core.getInput('message')
    core.debug(`message: ${message}`)

    // Create a GitHub client.
    const client = new github.GitHub(token)

    // Get owner and repo from context
    const owner = github.context.repo.owner
    const repo = github.context.repo.repo

    // Create a comment on PR
    // https://octokit.github.io/rest.js/#octokit-routes-issues-create-comment
    const response = await client.issues.createComment({
      owner,
      repo,
      // eslint-disable-next-line @typescript-eslint/camelcase
      issue_number: pr.number,
      body: message
    })
    core.debug(`created comment URL: ${response.data.html_url}`)

    core.setOutput('comment-url', response.data.html_url)
  } catch (error) {
    core.setFailed(error.message)
  }
}
```

そして、テンプレートリポジトリに存在した `__tests__/main.test.ts` を `__tests__/run.test.ts` に移して、リスト 3.16 のように修正しました。

▼リスト 3.16　__tests__/run.test.ts

```
import {run} from '../src/run'
import * as core from '@actions/core'
import * as github from '@actions/github'
import nock from 'nock'
import * as process from 'process'

beforeEach(() => {
  jest.resetModules()

  github.context.payload = {
    action: 'opened',
    pull_request: {
      number: 1
    }
  }
})

test('comments on PR', async () => {
  process.env['INPUT_REPO-TOKEN'] = 'test-github-token'
  process.env['INPUT_MESSAGE'] = 'Test Comment'
  process.env['GITHUB_REPOSITORY'] = 'testowner/testrepo'

  nock('https://api.github.com')
    .post(
      '/repos/testowner/testrepo/issues/1/comments',
      body => body.body === 'Test Comment'
    )
    .reply(200, {
      html_url: 'https://github.com/testowner/testrepo/issues
                    /1#issuecomment-1'
    })
  const setOutputMock = jest.spyOn(core, 'setOutput')

  await run()

  expect(setOutputMock).toHaveBeenCalledWith(
    'comment-url',
    'https://github.com/testowner/testrepo/issues/1#issuecomment-1'
  )
})
```

inputs のパラメータは、環境変数の INPUT_<パラメータ名の大文字> として渡されるので、process.env に値を入れてモックしています。コンテキストからリポジトリの所有者とリポジトリ名を取得するところも実態は環境変数なので、process.env でモック可能です。さらに nock を使えば、GitHub API とのやりとりも含めて、GitHub Actions の一連の流れをテストすることが可能です。

実際にこのテストを書くかと言われると微妙なところですが、あくまでモックしてアクションのテストを書くサンプルとしてお読みください。

続いて、.github/workflows/test.yml もリスト 3.17 のように修正します。

▼リスト 3.17　.github/workflows/test.yml

```
name: "build-test"
on: # rebuild any PRs and main branch changes
  pull_request:
  push:
    branches:
      - master
      - 'releases/*'

jobs:
  build: # make sure build/ci work properly
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2.0.0
      - uses: actions/setup-node@v1.4.0
        with:
          node-version: 12.x
      - run: |
          npm ci
          npm run all
      - name: Verify no unstaged changes
        run: |
          if [[ "$(git status --porcelain)" != "" ]]; then
            git status
            echo "::error::Unstaged changes detected.
                  Run 'npm run all' before commit"
            exit 1
          fi
  test: # make sure the action works on a clean machine without building
    if: github.event_name == 'pull_request'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2.0.0
      - id: prComment
        uses: ./
        with:
          repo-token: ${{ secrets.GITHUB_TOKEN }}
          message: PR comment test
      - name: Print comment URL
        run: echo "Comment URL - ${{ steps.prComment.outputs.comment-url }}"
```

build ジョブで actions/setup-node を呼び出して Node.js 環境を 12 系に固定しています。また、CI 用途では npm install より npm ci がいいので、修正しています。さらに、steps の最後に、git の差分が存在しないことを確認する処理を追加しています。これは、コミット時に npm run all を忘れると、ncc によるコンパイルが行われず、dist/index.js が更新されない問題を防ぐためです。

test ジョブは、動作確認をできるのは pull_request イベント時のみなので、if を使って pull_request イベント時だけジョブが実行するように修正しています。また、パラメータに repo-token と message を渡し、アウトプットの comment-url をログに出力するようにしています。GITHUB_TOKEN については、「2.4.5 秘密情

報」を参照してください。

README.md もアクションに合わせてリスト 3.18 のように修正します。

▼リスト 3.18　README.md

````
<p align="center">
  ...
</p>

# PR Comment Action

This action comments a message on PR.

## Inputs

### `repo-token`

**Required** The GitHub Token for comment on PR.

### `message`

**Required** The message to comment on PR.

## Outputs

### `commentUrl`

The PR comment URL.

## Example Usage

```yaml
uses: github-actions-up-and-running/pr-comment@v1.0.0
with:
  repo-token: ${{ secrets.GITHUB_TOKEN }}
  message: Nice PR!
```
````

「3.6 README.md」の内容を含めています。バッジの URL や Example Usage の uses は、各自のリポジトリに合わせて修正してください。

LICENSE ファイルも Copyright の部分を修正する必要がありますが、ここでは省略します。

テンプレートに含まれてる src/wait.ts は、いらなくなるので消します。

```
$ git rm src/wait.ts
```

最終的に、リポジトリのディレクトリ構造はリスト 3.19 のようになるはずです。

▼リスト 3.19　TypeScript アクションのディレクトリ構造

```
(repository root)
|-- .eslintignore
|-- .eslintrc.json
|-- .github
|   `-- workflows
|       `-- test.yml
|-- .gitignore
|-- .prettierignore
|-- .prettierrc.json
|-- LICENSE
|-- README.md
|-- __tests__
|   `-- run.test.ts
|-- action.yml
|-- dist
|   `-- index.js
|-- jest.config.js
|-- package-lock.json
|-- package.json
|-- src
|   |-- main.ts
|   `-- run.ts
`-- tsconfig.json
```

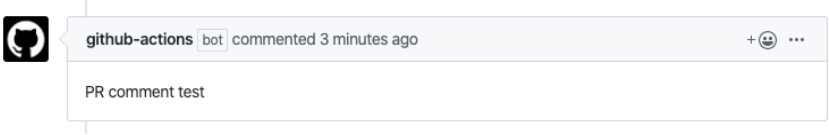
すべての修正が終わったら、次のように `npm run all` を実行して `dist/index.js` を再コンパイルしてから、ブランチを作成して push します。

```
$ npm run all
$ git checkout -b pr-comment
$ git add .
$ git commit -m "create PR comment action"
$ git push origin pr-comment
```

プルリクエストを作成すると、`pull_request` イベントでワークフローが実行されます。`test` ジョブで作成したアクションによってプルリクエストにコメントが投稿され、図 3.5 のように最後にコメントへの URL が出力されます。

▲図 3.5　作成したアクションの動作確認

そして、図 3.6 のようにプルリクエスト上でコメントを確認できます。



▲図 3.6　プルリクエストコメント

`repo-token` に `GITHUB_TOKEN` を渡しているため github-actions bot によってコメントが作成されます。自分がコメントしてるように表示させたいのであれば、自分のパーソナルアクセストークンを `repo-token` パラメータに渡す必要があります。

最後に、プルリクエストを master にマージし、公開するためにタグを作成します。

```
$ git checkout master
$ git tag v1.0.0
$ git push origin v1.0.0
```

ここまでで作成したアクションのサンプルは、次のリポジトリで公開しています。

- `https://github.com/github-actions-up-and-running/pr-comment`

タグができたら、「3.2.2 JavaScript アクションを使う」を参考に、別リポジトリからアクションを実行できることを確認してみてください。

3.9 Docker コンテナアクション

今度は、Docker コンテナアクションを作成してみます。

ここでは、パラメータで受け取った文字列を大文字にして返すアクションを作成します。

3.9.1 Docker コンテナアクションを作成

まず、「3.7.3 actions/container-action」で説明した公式テンプレートリポジトリから新たにリポジトリを作成します。

そして、`action.yml` をリスト 3.20 のように修正します。

▼リスト 3.20　action.yml

```
name: 'Uppercase'
description: 'Convert text to uppercase'
author: 'miyajan'
inputs:
  text:
    description: 'Text to be converted to uppercase'
    required: true
outputs:
  uppercase-text:
    description: 'Text converted to uppercase'
runs:
  using: 'docker'
  image: 'Dockerfile'
  args:
    - ${{ inputs.text }}
branding:
  icon: 'chevrons-up'
  color: 'green'
```

次に、`Dockerfile` をリスト 3.21 のように修正します。

▼リスト 3.21　Dockerfile

```
FROM debian:10.2

COPY LICENSE README.md /

COPY entrypoint.sh /entrypoint.sh

ENTRYPOINT ["/entrypoint.sh"]
```

`FROM` の部分を修正してベースイメージを変更しただけです。

続いて、entrypoint.sh をリスト 3.22 のように修正します。

▼リスト 3.22　entrypoint.sh

```
#!/bin/bash -l

echo "::set-output name=uppercase-text::${1^^}"
```

受け取った文字列の大文字化を bash で行い、アウトプットに設定しています。
README.md もアクションに合わせてリスト 3.23 のように修正します。

▼リスト 3.23　README.md

```
<p align="center">
  ...
</p>

# Uppercase Action

This action converts a text to uppercase.

## Inputs

### `text`

**Required** The text to be converted to uppercase.

## Outputs

### `uppercase-text`

The text converted to uppercase.

## Example Usage

```yaml
uses: github-actions-up-and-running/uppercase@v1.0.0
with:
  text: Hello, World!
```
```

「3.6 README.md」の内容を含めています。バッジの URL は、各自のリポジトリに合わせて修正してください。

そして、Docker コンテナアクションにはワークフローが設定されていないので、動作確認のために .github/workflows/test.yml をリスト 3.24 のように作成します。

▼リスト 3.24　.github/workflows/test.yml

```
name: "test"
on: # rebuild any PRs and main branch changes
  pull_request:
  push:
    branches:
      - master
      - 'releases/*'

jobs:
  test: # make sure the action works on a clean machine without building
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2.0.0
      - id: uppercase
        uses: ./
        with:
          text: Hello, World!
      - name: Print converted text
        run: echo "${{ steps.uppercase.outputs.uppercase-text }}"
```

test ジョブで作成したアクションの動作確認をしています。パラメータに text を渡し、アウトプットの uppercase-text をログに出力するようにしています。

LICENSE ファイルも Copyright の部分を修正する必要がありますが、ここでは省略します。

最終的に、リポジトリのディレクトリ構造はリスト 3.25 のようになるはずです。

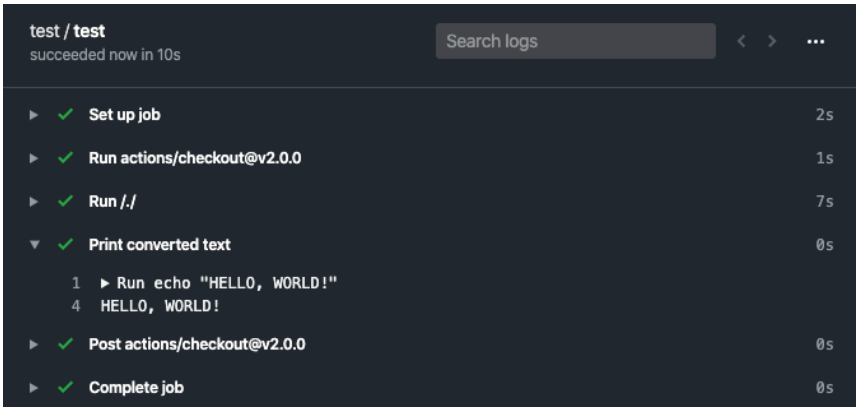
▼リスト 3.25　Docker コンテナアクションのディレクトリ構造

```
(repository root)
|-- .github
|   `-- workflows
|       `-- test.yml
|-- Dockerfile
|-- LICENSE
|-- README.md
|-- action.yml
`-- entrypoint.sh
```

すべての修正が終わったら、次のように push します。

```
$ git add .
$ git commit -m "create uppercase action"
$ git push origin master
```

すると、図 3.7 のように、Docker コンテナアクションが動作して、パラメータ `text` で渡した文字列が大文字になっていることを確認できます。



▲図 3.7　Docker コンテナアクションの動作確認

最後に、公開用のタグを作成します。

```
$ git tag v1.0.0
$ git push origin v1.0.0
```

ここまでで作成したアクションのサンプルは、次のリポジトリで公開しています。

- `https://github.com/github-actions-up-and-running/uppercase`

タグができたら、「3.2.2 JavaScript アクションを使う」を参考に、別リポジトリからアクションを実行できることを確認してみてください。

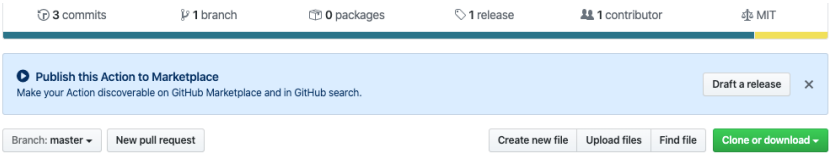
3.10　アクションを GitHub Marketplace へ公開

GitHub には Apps やアクションを公開するための、GitHub Marketplace が存在します。本書執筆時点では、アクションについては課金要素はありません。Marketplace に公開するメリットは、現時点では Marketplace の検索に引っかかるようになることだけだと思います。

次の条件を満たすとき、アクションを Marketplace に公開できます。

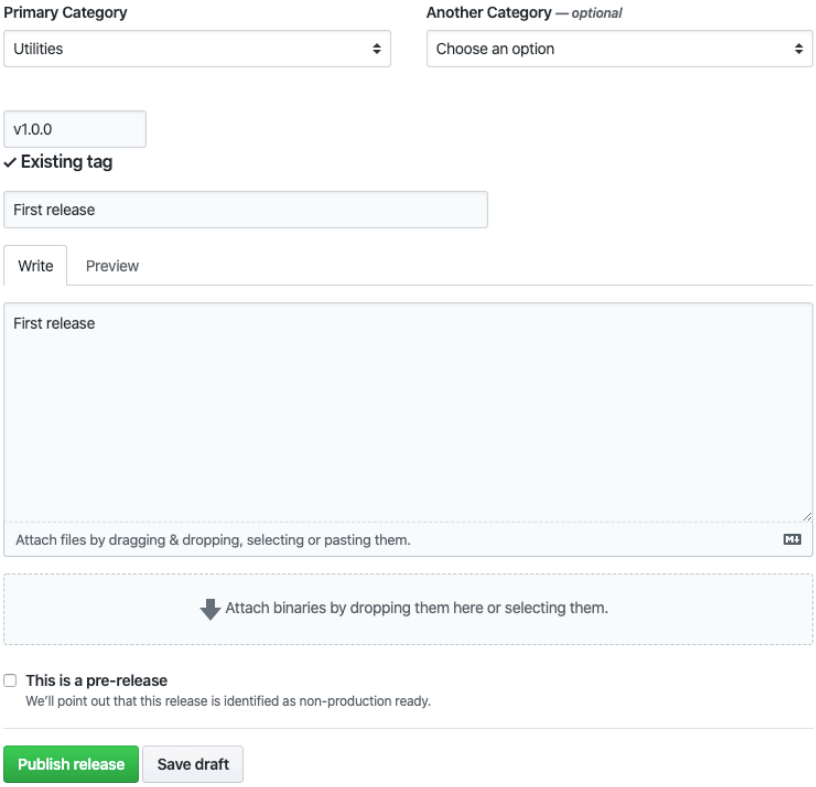
- アクションがパブリックリポジトリに存在する
- 1 リポジトリに 1 アクションだけ存在する
- メタデータファイル（`action.yml`）がリポジトリのルートディレクトリに存在する
- アクションの `name` が次の条件を満たす
  - Marketplace 内の他のアクション名と重複しない
  - 自分が所有者でないユーザー名、organization 名と重複しない
  - GitHub の予約語と重複しない

ためしに、「3.8 TypeScript アクション」で作成したアクションを Marketplace に公開してみます。仕組みとしては、これまでの GitHub のリリース機能に Marketplace 公開フラグが付いたような感じです。まず、ブラウザでアクションを作成したリポジトリを開くと、図 3.8 のように "Draft a release" ボタンが表示されるので、そこからアクションを公開できます。



▲図 3.8　アクションを Marketplace に公開する

リリースの編集画面が表示されたら、"Primary Category" に "Utilities"、"Tag version" に "v1.0.0" を入力します。"Release title" と "Describe this release" には本来はそのリリースによる主要な差分を書くところですが、今回は初回なので "First release" としておきます。図 3.9 のようになったら、"Publish release" ボタンを押します。

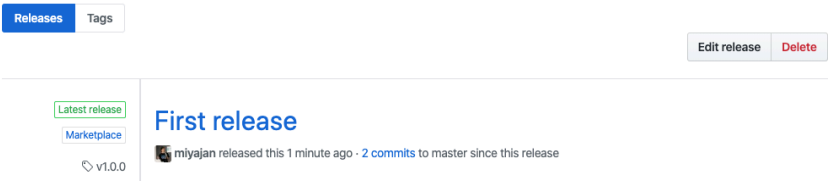


▲図 3.9　アクションをリリースする

すると、アクションが公開されます。今回公開したサンプルリポジトリのアクションは、次の URL で見られます。

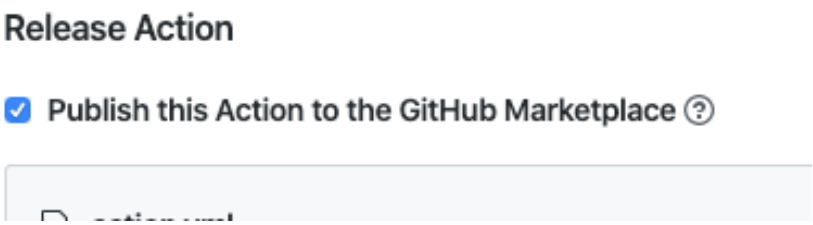
- `https://github.com/marketplace/actions/pr-comment`

ちなみに、アクションの公開を取り消す場合は、まず公開したリリースの編集画面を開きます。図 3.10 のリリースの詳細画面から、"Edit release" ボタンを押します。



▲図 3.10　リリースを編集

そして、図 3.11 のチェックボックスを外して、"Update release" を押します。



▲図 3.11　アクションの公開を取り消す

すると、アクションの公開が取り消されます。

3.11　アクションのデバッグ

アクションのデバッグ方法も、基本的には「2.9 ワークフローのデバッグ」で紹介した方法と同じなので、そちらを参照してください。

3.12 アクションの探し方

3.12.1 GitHub 公式のアクション

- `https://github.com/actions`

actions organization 下に、GitHub 公式のアクションのリポジトリが複数存在します。（アクション以外の toolkit とかもありますが）

利用する上でも便利なアクションが多いですし、アクションを作成するときの参考にもなります。

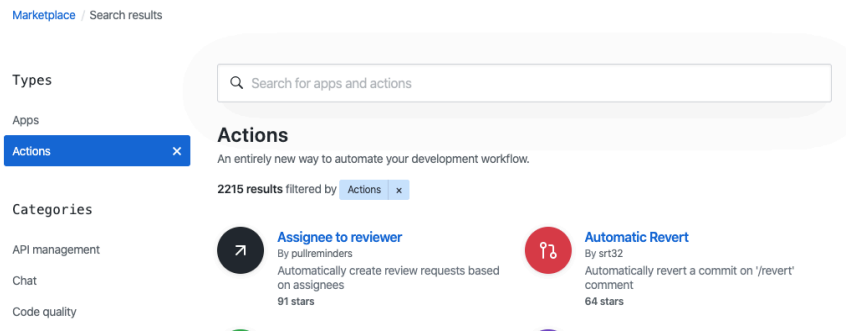
3.12.2 GitHub Marketplace

「3.10 アクションを GitHub Marketplace へ公開」でも書いたとおり、GitHub には Apps やアクションを公開するための、GitHub Marketplace が存在します。

- `https://github.com/marketplace`

GitHub Marketplace で "Types" を "Actions" で絞り込むと、アクション一覧を表示できます。

図 3.12 のように、カテゴリで絞り込んだり、キーワードで検索したりして、アクションを探すことができます。



▲図 3.12　GitHub Marketplace のアクション一覧

Marketplace 上では、GitHub 公式のアクションとサードパーティーが作成したアクションのどちらも公開されています。

Marketplace というと有料かと思われるかもしれませんが、本書執筆時点では公開されているアクションはすべて無料で利用できます。

3.13 まとめ

GitHub Actions のアクションの作り方を、サンプルを示しつつ解説しました。

基本的には、公式のテンプレートリポジトリから作成し、公式のアクションを参考にしつつ、toolkit などの便利ツールを活用して作っていくのがいいと思います。

第 4 章

サンプルレシピ

本章では、より実践的な GitHub Actions のユースケースのサンプルをいくつか紹介します。

4.1 ジョブ失敗時に Slack に通知

「2.11 通知」で説明したように、GitHub Actions の通知は、公式ではメール通知と Web Notifications のみ存在します。Slack に通知を飛ばしたいといった場合には、ワークフローの設定側で通知処理を入れる必要があります。

今回は、`rtCamp/action-slack-notify`[*1] というサードパーティーのアクションを使用して、ジョブ失敗時の Slack 通知を実現します。

まず、通知を飛ばす Slack チャンネルに Incoming Webhooks[*2] アプリを追加します。Incoming Webooks アプリは、外部から Slack チャンネルに投稿をするための URL を生成するアプリです。

チャンネルにアプリを追加したら、"Webhook URL" が表示されるので、この URL をコピーします。適当なリポジトリを作成し、Webhook URL をリポジトリの秘密情報に `SLACK_WEBHOOK` という名前で保存します。秘密情報の設定方法については、「2.4.5 秘密情報」を参考にしてください。

そして、リポジトリにリスト 4.1 のようなワークフローを設定して実行してみてください。

[*1] `https://github.com/marketplace/actions/slack-notify`

[*2] `https://slack.com/apps/A0F7XDUAZ-incoming-webhooks`

▼リスト 4.1　ジョブ失敗時に Slack に通知

```
name: Slack notification on job failure
on: push
jobs:
  slack-notify:
    runs-on: ubuntu-latest
    steps:
      - name: Fail job
        run: exit 1
      - name: Slack notification
        if: failure()
        # v2.0.1
        uses: rtCamp/action-slack-notify
                @8a36de024523717693a9a7b80c6ead640d8873ce
        env:
          SLACK_COLOR: danger
          SLACK_WEBHOOK: ${{ secrets.SLACK_WEBHOOK }}
```

ワークフローが実行されると、Slack のチャンネルに図 4.1 のようなメッセージが投稿されます。

▲図 4.1　ジョブ失敗時に Slack に通知

`rtCamp/action-slack-notify` を実行する前のステップでは、`exit 1` を実行して強制的にジョブを失敗させています。実際のジョブでは、この部分が自動テストなどの本来の処理に置き換わります。

`if: failure()` を使うことで、ジョブが失敗したときのみ Slack への通知が実行されるようにしています。成功・失敗に関わらず常に通知を飛ばしたいということであれば、`if: always()` を指定することになります。

環境変数では、`SLACK_COLOR` で Slack メッセージの色を設定し、`SLACK_WEBHOO`

K で秘密情報に登録した Webhook URL を設定しています。このあたりの環境変数で指定できる設定の詳細については、アクションの説明を参照してください。

今回実験したサンプルリポジトリは、次の URL で見られます。

- `https://github.com/github-actions-up-and-running/slack-notify`

この方法の難点は、ジョブ単位でこのアクションを使用するように設定しないといけないということです。大規模なワークフローになってくるとジョブの数も多くなってくるので、一つ一つのジョブにこの設定を追加するのは少々面倒に感じるかもしれません。GitHub Actions にワークフロー全体の後処理みたいな機能が追加されると嬉しいのですが、今後に期待です。

4.2 ワークフロー実行環境に SSH してデバッグ

CI/CD を本格的に活用していると、手元では動くのに CI/CD 環境では動かないということがたまにあります。大抵の場合、これは環境の差異が原因です。オンプレに構築される CI/CD ツールの場合は手軽に環境にログインできるかもしれませんが、クラウド上の CI/CD サービスの場合、ビルド環境にアクセスするために一工夫必要になります。例えば、CircleCI という CI/CD サービスでは、SSH してデバッグするための機能を用意しています。

GitHub Actions では、公式でワークフロー実行環境に SSH する機能を提供していません。代わりに、`mxschmitt/action-tmate`[*3] というサードパーティーのアクションを実行することで、ワークフロー実行環境で tmate が起動し、環境に SSH できるようになります。

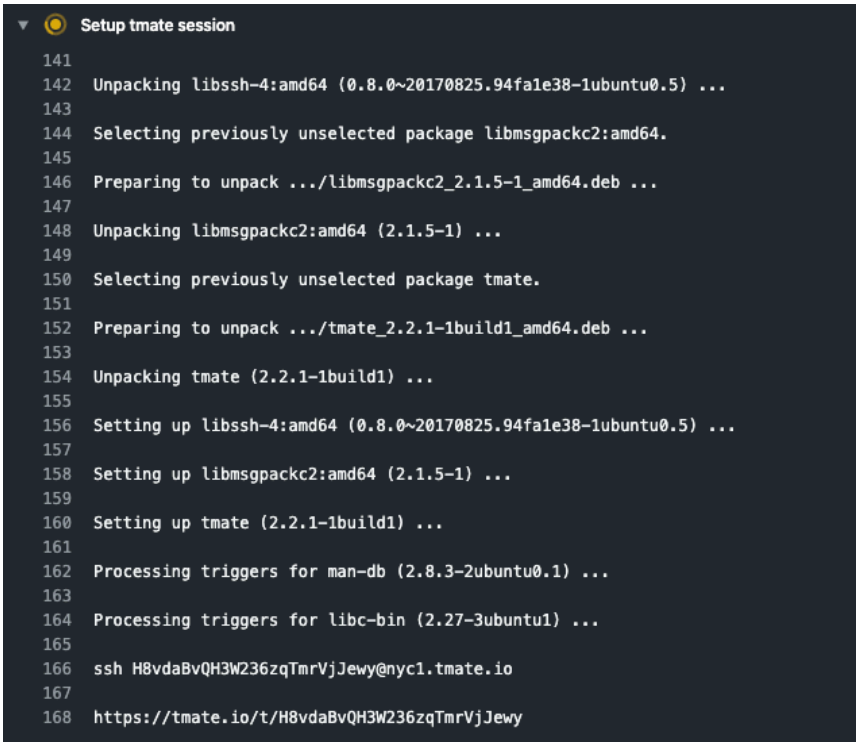
試しに、適当なリポジトリでリスト 4.2 のようなワークフローを実行してみてください。

▼リスト 4.2　tmate で SSH デバッグ

```
name: SSH Debug
on: push
jobs:
  ssh-debug:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2.0.0
      - name: Setup tmate session
        # v1.0.2
        uses: mxschmitt/action-tmate@b85ae7b45d889eec8b1772f81370970c15b78092
```

[*3] `https://github.com/marketplace/actions/debugging-with-tmate`

ワークフローが実行されると、mxschmitt/action-tmate アクションを実行するステップでワークフローが止まります。そして、図 4.2 のように、ワークフローのログに SSH コマンドと URL が表示されます。



▲図 4.2　ワークフローのログに SSH コマンドが表示される

この SSH コマンドを実行するか、URL を開くかすると、ワークフロー実行環境に SSH したターミナルが表示されます。これにより、ワークフロー実行環境内で自由にデバッグできるようになります。

このアクションは SSH セッションを終了する方法を提供していないので、ワークフローを終了させるにはワークフローをキャンセルする必要があります。

今回実験したサンプルリポジトリは、次の URL で見られます。

- https://github.com/github-actions-up-and-running/ssh-debug

注意点としては、ログが見られる人なら誰でも環境に SSH できるようになってしまうので、環境に渡された秘密情報が漏洩する恐れがあることです。プライベートリポジトリか、秘密情報が渡らないジョブに対してのみ実行するようにしましょう。

使い勝手面では、いちいちワークフロー設定ファイルを修正しないと SSH できるようにならないので、正直今ひとつなところがあります。CircleCI のように、公式で SSH デバッグ機能がサポートされることを期待しています。

tmate とは

tmate[*4] は、tmux[*5] の fork で、手軽に SSH のセッションをユーザー間で共有することができるツールです。

tmate は起動したマシンから `tmate.io` にトンネルを開けます。`mxschmitt/action-tmate` は、これを利用して外部から GitHub Actions のワークフロー実行環境に SSH できるようにしています。

4.3 Docker イメージを GitHub Packages で公開

GitHub には、GitHub Packages というパッケージをホスティングするためのサービスが存在します。詳細については、GitHub Packages の公式ドキュメント[*6]を参照してください。

GitHub Packages は、Docker レジストリにも対応しているので、GitHub Actions から Docker イメージをビルドして公開するためのワークフローを作成してみます。`docker/build-push-action`[*7] という Docker 公式のアクションを利用することで、簡単に実現できます。

まず、適当なリポジトリを作成して、リポジトリのルートディレクトリに `Dockerfile` を作成します。そして、リスト 4.3 のようなワークフローを作成します。

[*5] `https://tmate.io/`
[*6] `https://github.com/tmux/tmux`
[*6] `https://help.github.com/packages`
[*7] `https://github.com/docker/build-push-action`

▼リスト 4.3　Docker イメージを GitHub Packages で公開

```
name: Publish Docker Image
on:
  push:
    tags:
      - "[0-9]+.[0-9]+.[0-9]+"
jobs:
  publish:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2.0.0
      - name: Publish to GitHub Packages
        uses: docker/build-push-action@v1.1.0
        with:
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}
          registry: docker.pkg.github.com
          repository: ${{ github.repository }}/hello
          tag_with_ref: true
```

このワークフローは、セマンティックバージョン（例: 1.0.0）のタグが push されたときだけ実行されます。`docker/build-push-action` には、次のパラメータを渡しています。

- `username`: Docker レジストリにログインするために使うユーザー名
- `password`: Docker レジストリにログインするために使う認証情報
- `registry`: Docker レジストリのホスト名
- `repository`: イメージ名
- `tag_with_ref`: `true` を渡すと、git の ref から自動的にタグを生成する

「2.4.5 秘密情報」で書いたとおり、`GITHUB_TOKEN` は GitHub Packages の write 権限もあるので、今回のようなユースケースでも使えます。

`tag_with_ref: true` を設定すると、git のタグ名が "1.0.0" のとき、git の ref の `refs/tags/1.0.0` から、自動的に `1.0.0` を Docker イメージのタグ名として設定してくれます。

"1.0.0" のようなタグを作成して push するとワークフローが実行され、イメージが GitHub Packages で公開されます。ブラウザでリポジトリを開き、package を開くと、公開された Docker イメージが図 4.3 のように確認できます。

▲図 4.3　GitHub Packages で公開された Docker イメージ

公開されたイメージを使うときは、次のコマンドのように先に `docker login` コマンドが必要です。

```
$ docker login -u <USERNAME> -p <TOKEN> docker.pkg.github.com
$ docker pull docker.pkg.github.com/github-actions-up-and-running/
    publish-docker-image/hello:1.0.0
```

`TOKEN` には `read:packages` 権限のあるパーソナルアクセストークンを使います。今回実験したサンプルリポジトリは、次の URL で見られます。

- https://github.com/github-actions-up-and-running/publish-docker-image

4.4　reviewdog で Lint の結果をプルリクエストに表示

reviewdog[*8] というツールを使うと、静的解析の結果をプルリクエストの差分上にコメントとして表示することができます。今回は、ESLint で問題があったときに、この reviewdog でプルリクエストにコメントするように GitHub Actions を設定してみます。

reviewdog は、公式で各種 Lint のアクションを提供しています。今回は、`reviewdog/action-eslint`[*9] アクションを利用しますが、他にも様々な Lint のアクショ

[*8] https://github.com/reviewdog/reviewdog
[*9] https://github.com/marketplace/actions/run-eslint-with-reviewdog

ンが用意されているので、興味がある人は調べてみてください。

新しくリポジトリを作成し、ESLint に引っかかる適当な JavaScript ファイルと package.json を作成してください。reviewdog/action-eslint アクションは、eslint パッケージが必要なのでインストールしておきます。

```
$ npm install --save-dev eslint
```

そしたら、リスト 4.4 のようなワークフローを作成します。

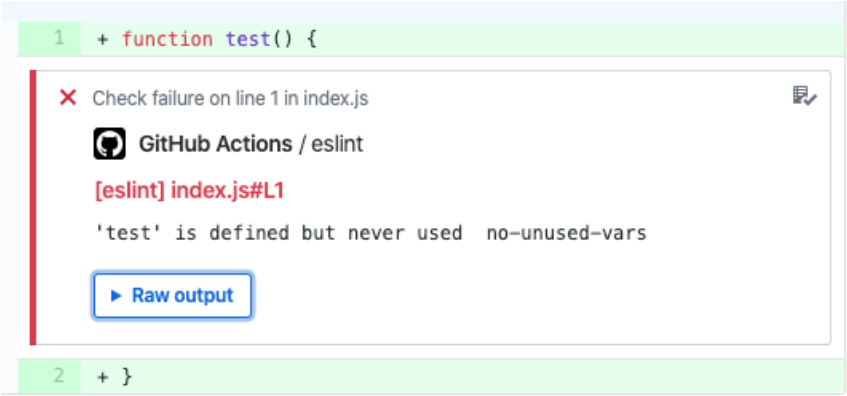
▼リスト 4.4　reviewdog + ESLint

```
name: reviewdog ESLint
on: pull_request
jobs:
  eslint:
    name: reviewdog ESLint
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2.0.0
      - name: ESLint
        uses: reviewdog/action-eslint@v1.2.0
        with:
          github_token: ${{ secrets.GITHUB_TOKEN }}
```

reviewdog/action-eslint アクションは、GitHub の API を利用するので、github_token パラメータで認証情報を渡します。今回はこのパラメータだけしか指定していませんが、他にもパラメータはあるので、気になる人は reviewdog/action-eslint アクションの説明[*10]を見てみてください。

そして、プルリクエストを作成すると、ワークフローが実行されます。ワークフロー自体は ESLint に引っかかっても成功しますが、図 4.4 のようにプルリクエストの差分上に ESLint が発見した問題が表示されます。

[*10] https://github.com/marketplace/actions/run-eslint-with-reviewdog



▲図 4.4　reviewdog + ESLint

今回実験したサンプルリポジトリは、次の URL で見られます。

- https://github.com/github-actions-up-and-running/reviewdog-eslint

4.5　Terraform GitHub Actions で AWS にデプロイ

Terraform[*11] は、HashiCorp 社が公開している、AWS などの IaaS のインフラ構築をコードで定義できるようにするツールです。HashiCorp 社は、公式で GitHub Actions 向けに Terraform GitHub Actions[*12] というアクションを公開しているので、これを使って AWS にデプロイするワークフローを作成してみます。

今回作成するワークフローは 2 つあります。一つは、プルリクエスト作成時に、Terraform ファイルの問題を検知する `terraform fmt` や `terraform validate`、リソースの差分を表示する `terraform plan` を実行するワークフローを用意します。もう一つは、master に push されたときに実行される、Terraform のデプロイを行う `terraform apply` を実行するワークフローです。

まず、新たにリポジトリを作成し、適当に AWS のリソースを定義する Terraform のファイルを作成します。そして、秘密情報に AWS の認証情報として、`AWS_ACCESS_KEY_ID` と `AWS_SECRET_ACCESS_KEY` を登録します。その上で、プルリクエストイベント時のワークフローをリスト 4.5 のように作成します。

[*11] https://www.terraform.io/
[*12] https://github.com/hashicorp/terraform-github-actions

▼リスト 4.5　プルリクエストのワークフロー

```
name: 'terraform fmt, init, validate and plan on PR'
on: pull_request
jobs:
  terraform:
    name: 'Terraform'
    runs-on: ubuntu-latest
    env:
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      TERRAFORM_VERSION: 0.12.20
    steps:
      - name: 'Checkout'
        uses: actions/checkout@v2.0.0
      - name: 'Terraform Format'
        uses: hashicorp/terraform-github-actions@v0.7.1
        with:
          tf_actions_version: ${{ env.TERRAFORM_VERSION }}
          tf_actions_subcommand: 'fmt'
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
      - name: 'Terraform Init'
        uses: hashicorp/terraform-github-actions@v0.7.1
        with:
          tf_actions_version: ${{ env.TERRAFORM_VERSION }}
          tf_actions_subcommand: 'init'
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
      - name: 'Terraform Validate'
        uses: hashicorp/terraform-github-actions@v0.7.1
        with:
          tf_actions_version: ${{ env.TERRAFORM_VERSION }}
          tf_actions_subcommand: 'validate'
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
      - name: 'Terraform Plan'
        uses: hashicorp/terraform-github-actions@v0.7.1
        with:
          tf_actions_version: ${{ env.TERRAFORM_VERSION }}
          tf_actions_subcommand: 'plan'
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

環境変数が terraform コマンドにそのまま渡されるので、AWS_ACCESS_KEY_ID と AWS_SECRET_ACCESS_KEY を秘密情報から取得して渡しています。

Terraform GitHub Actions は、tf_actions_subcommand パラメータで実行する terraform のサブコマンドを定義できます。ここでは、terraform fmt、terraform init、terraform validate、terraform plan コマンドが実行されます。

そして、tf_actions_version パラメータで terraform コマンドのバージョンを定義します。環境変数で GITHUB_TOKEN を渡すのは、Terraform GitHub Actions がプルリクエストにコメントを付けられるようにするためです。

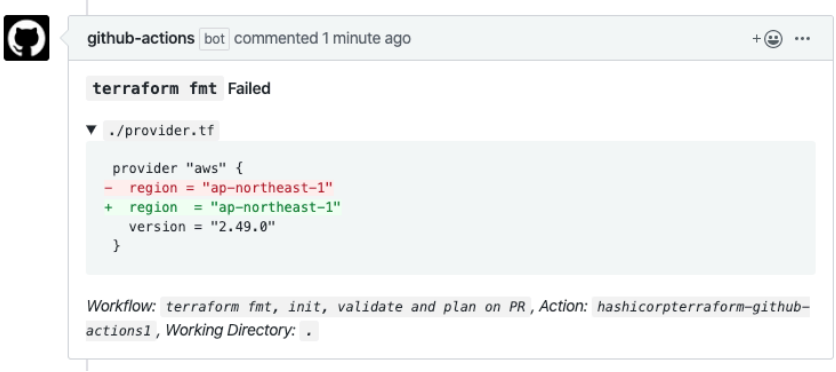
次に、master に push したときのワークフローをリスト 4.6 のように作成します。

▼リスト 4.6　master に push したときのワークフロー

```
name: 'terraform apply on master push'
on:
  push:
    branches:
      - master
jobs:
  terraform:
    name: 'Terraform'
    runs-on: ubuntu-latest
    env:
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      TERRAFORM_VERSION: 0.12.20
    steps:
      - name: 'Checkout'
        uses: actions/checkout@master
      - name: 'Terraform Init'
        uses: hashicorp/terraform-github-actions@v0.7.1
        with:
          tf_actions_version: ${{ env.TERRAFORM_VERSION }}
          tf_actions_subcommand: 'init'
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
      - name: 'Terraform Apply'
        uses: hashicorp/terraform-github-actions@v0.7.1
        with:
          tf_actions_version: ${{ env.TERRAFORM_VERSION }}
          tf_actions_subcommand: 'apply'
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

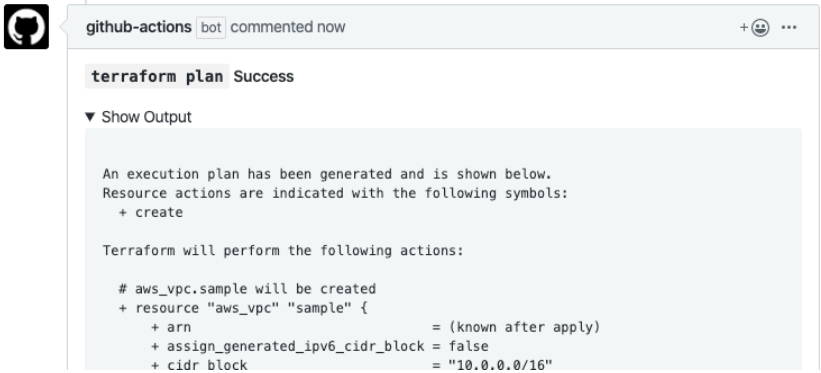
ここでは、`terraform init`、`terraform apply` が実行されます。

この状態でプルリクエストを作成すると、例えば `terraform fmt` で差分が発生した場合はワークフローが失敗し、図 4.5 のようにプルリクエストに問題箇所がコメントされます。



▲図 4.5　terraform fmt

問題がなければ、図 4.6 のように `terraform plan` によるリソースの差分がコメントされます。



▲図 4.6　terraform plan

これによって、プルリクエスト上で `terraform` による変更内容のレビューが行えるようになります。

差分を見て問題なければ、master にマージすることによって今度はデプロイのワークフローが実行されます。

今回実験したサンプルリポジトリは、次の URL で見られます。

- https://github.com/github-actions-up-and-running/terraform-github-actions

このワークフローの注意点としては、プルリクエストで差分を確認してから master にマージするまでの間に他の変更が適用されてしまうと、想定外の結果になってしまう可能性があるということです。この問題を回避するためには、`terraform plan` の結果をアーティファクトなどに保存しておいて、master にマージしたタイミングでそれを使うようにするような修正が必要になります。興味がある人は探求してみてください。

4.6 まとめ

公開されているアクションを活用した、実践的なワークフローのサンプルをいくつか紹介しました。

公式・非公式を合わせると、現時点でも 2,000 以上の数のアクションが Marketplace に登録されています。この章で紹介したアクションを参考にすれば、自分で新たなアクションを作成するきっかけにもなりやすいと思います。

ここで紹介したサンプルが、読者の方々が GitHub Actions を活用するなにかしらのきっかけになれば幸いです。

付録 A

コミュニティ

GitHub Actions のコミュニティについてご紹介します。

A.1 GitHub Community Forum

GitHub 公式のフォーラムで、GitHub Actions のカテゴリがあります。

- `https://github.community/t5/GitHub-Actions/bd-p/actions`

すべて英語ですが、質問すれば GitHub の中の人から回答をもらえる可能性もあります。GitHub Actions で詰まったときはここで検索すると回答が見つかることも多いです。

A.2 connpass

GitHub 非公認ですが、connpass 上に、次の 2 つの日本国内コミュニティが存在し、不定期でミートアップを開催しています。

- GitHub Actions User Group（大阪）
  - `https://gaug.connpass.com/`
- GitHub Actions User Group Tokyo
  - `https://gaugt.connpass.com/`

GitHub Actions User Group Tokyo は、筆者が運営に関わっています。GitHub Actions についてワイワイ盛り上がりたい人はぜひご参加ください。

A.3 Slack

GitHub 非公認ですが、日本語で GitHub Actions について気軽にワイワイ話せるように、Slack を用意しました。次の URL に参加リンクがあります。

- `https://github-actions-jp.github.io/`

雑談や質問をするためのチャンネルもあるので、気軽に書き込んでください。
こちらも筆者が運営に関わっています。

あとがき

執筆当初、本書はだいたい 100 ページぐらいになるだろうと見積もっていました。最終的にどうなったかは、現在のページ数をご覧いただければわかるかと思います。開発の見積もりは当てにならないということは知っていましたが、これは同人誌の執筆にも当てはまるということがよくわかりました。

本書は、GitHub Actions について、基本的な知識から少し実践的なところまで扱いました。自分は、GitHub Actions の次のようなところにこれまでにはない可能性を感じています。

- GitHub 内で完結してワークフローを組める
- CI/CD 用途だけでなく Webhook 用途もカバーできる
- 全部入り VM とコンテナのハイブリッドな構成
- パブリックリポジトリでの利用は完全無料
- セルフホストランナーで独自の環境を持ち込める仕組み
- アクションというエコシステム

本書によって、そういった GitHub Actions のおもしろさが少しでも伝われば嬉しいです。

読者のみなさまは、本書によってなにかしら得るものがあったでしょうか。GitHub Actions を触ってみるきっかけになった、これまで知らなかった機能を知れた、あとで「これ『GitHub Actions 実践入門』で読んだやつだ！」となった、ということがありましたら幸いです。もし気が向いたら、ご感想をまえがきの問い合わせ先にでも登録してもらえれば筆者が喜びます。

最後になりますが、本書をお手にとっていただき誠にありがとうございました。まだ書き足りないトピックもあるので、また改訂版などを出すかもしれません。他に書いてみたいテーマもあるので、そのときは twitter ででも告知いたします。

それでは、素晴らしい GitHub Actions ライフをお過ごしください！

著者紹介

宮田 淳平 / @miyajan

2009 年にサイボウズ株式会社へ入社。大規模向けグループウェア『ガルーン』やビジネスアプリ作成プラットフォーム『kintone』の開発を経験した後、生産性向上チームを立ち上げ、組織横断で開発基盤の整備と自動化の推進を行っています。

普段は、自動テスト、CI/CD 系のコミュニティに顔を出すことが多いです。GitHub Actions Meetup Tokyo コミュニティを運営しています。

索引

GitHub Actions 実践入門

2020 年 2 月 27 日　v1.0.0
2020 年 6 月 15 日　v1.2.0

著　者　宮田 淳平 (@miyajan)