

# CI/CDのエキスパートが解説: CircleCIで始めるCI/CD導入の基本のキ

## エンジニアのためのCI/CD再入門 第2回

金 洋国 (CircleCI Japan) [著]

第1回ではCI/CDに関する一般的な解説をしました。第2回と第3回でCI/CDの代表サービスの一つである CircleCI を使って実際にCI/CDを設定する感覚を学んでもらいたいと思います。今回はCircleCIの設定方法の基礎 から始めるので、記事の前半は比較的入門者向けの内容ですが、記事の後半では最新機能のVer.2.1も紹介します。これからCI/CDを導入する方にも、すでに活用されている方にも、CircleCIを通してCI/CDに 対する理解を深めるお手伝いができるれば幸いです。

- 第1回の記事:「[CI/CDのエキスパートが解説:CI/CDとは何か? なぜ今、必要とされるのか?](#)」
- 第3回の記事:「[モダンなCI/CDでは欠かせないワークフローを使った高度なビルド管理](#)」

### 対象読者

- CI/CDについて学びたい方
- CircleCIを使ってみたい方
- CircleCI Ver. 2.1について知りたい方

### 必要な環境／知識

- GitHubのアカウント
- ソフトウェアのテストについての一般的知識
- アジャイル開発についての一般知識

### 筆者について

元CircleCIの開発者で、現在はCircleCI初の海外支社である[CircleCI Japan](#)でさまざまな活動を行っています。

## CircleCIについて

CircleCIは数あるCI/CDサービスの中で代表的なサービスの一つです。CI/CDには主に自分たちで管理するホスティング型とサービス側が運用するクラウド型があるのですが、CircleCIはその両方を提供しています。今回はクラウド型を使って解説していきます。

### CircleCIの特徴とメリット

たくさんCI/CDサービスがある中でCircleCIを選ぶ理由は为什么呢？ 筆者の意見ではCircleCIには以下のようなメリットがあると思っています。

- モダンなCI/CDの機能の多くをサポートしている
- 今後日本語でのサポートや情報が期待できる

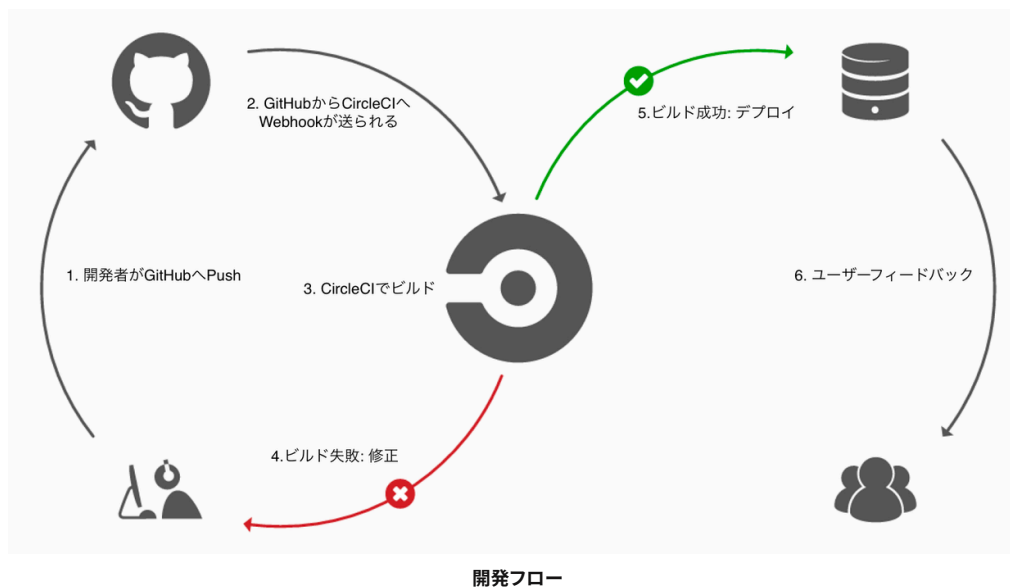
CircleCIは2018年9月を持って古い設定のサポートを終了しました。新しい設定はCircleCI 2.0 (以下、2.0) と呼ばれ、設定方法だけではなくアーキテクチャが一新されています。主に以下のような特徴があります。

- **柔軟かつ明示的なビルドの設定:** 近年のソフトウェア開発ではさまざまなツールや開発手法が使われるようになり、CI/CDサービスにはそれらに柔軟に対応することが求められています。2.0はユーザーが設定を明示的に書くことで、CI/CDの設定を自由にカスタマイズできるようになりました。
- **Dockerのサポート:** Dockerは開発現場で急速に広まっていますが、CI/CDの世界も例外ではありません。2.0ではDockerのネイティブ対応で開発環境や本番環境で使っている同じDockerのイメージを使ってビルドすることができます。
- **ワークフローのサポート:** CI/CDでできること(後述するジョブ)はどんどん増えています。そうすると、ジョブを組み合わせたり制御したりしたくなってきますが、これを可能にするのが次回で紹介するワークフローという機能です。名前はサービスにより異なりますが、最新のCI/CDサービスには必ず求められる機能の一つです。

以上がモダンなCI/CDという視点から見たCircleCIを使うメリットです。そして、副次的なメリットとしてこれから日本語での情報がどんどん増えることが期待されます。2018年の時点では主要なCI/CDサービスはすべて英語圏のサービスなので、サポートや公式ドキュメントは英語で提供されます。対してCircleCIは、2018年6月に日本支社であるCircleCI Japanを設立し、日本のマーケットに対してコミットしていくことを明確にしました。これからは日本語でのサポートの開始や日本のユーザーによるコミュニティで知見などがどんどん共有されることが期待されます。

## CircleCIを使った開発フロー

具体的な設定の解説に入る前に、CircleCIがどのように開発フローの中で使われるか説明します。



ほとんどのCI/CDサービスやツールはGitHubのようなVCSと連携して動きます。開発者が新しい変更をVCSにPushするとWebhookによりCircleCIへ通知が送られ、CircleCIはその変更のリビジョンを使ってビルドを開始します。もしビルドが失敗した場合、開発者は成功するまで修正します。ビルドが成功すれば、開発者はmasterブランチにマージしてもう一度そこでビルドが実行されます。その後はチームや開発体制によって異なりますが、基本的には2通りあります。

1. masterブランチでビルドが成功したらCI/CDの1サイクルは終了
2. masterブランチでビルドが成功したら本番・ステージング環境などに自動デプロイ

[第1回](#)でも解説しましたが、自動デプロイまでできればさまざまなメリットがありますが、デプロイ環境をきちんと対応させる必要があるのでハードルは低くはありません。今回は基本的にmasterブランチでビルドが成功すれば1サイクル終了という方針で解説していきます。

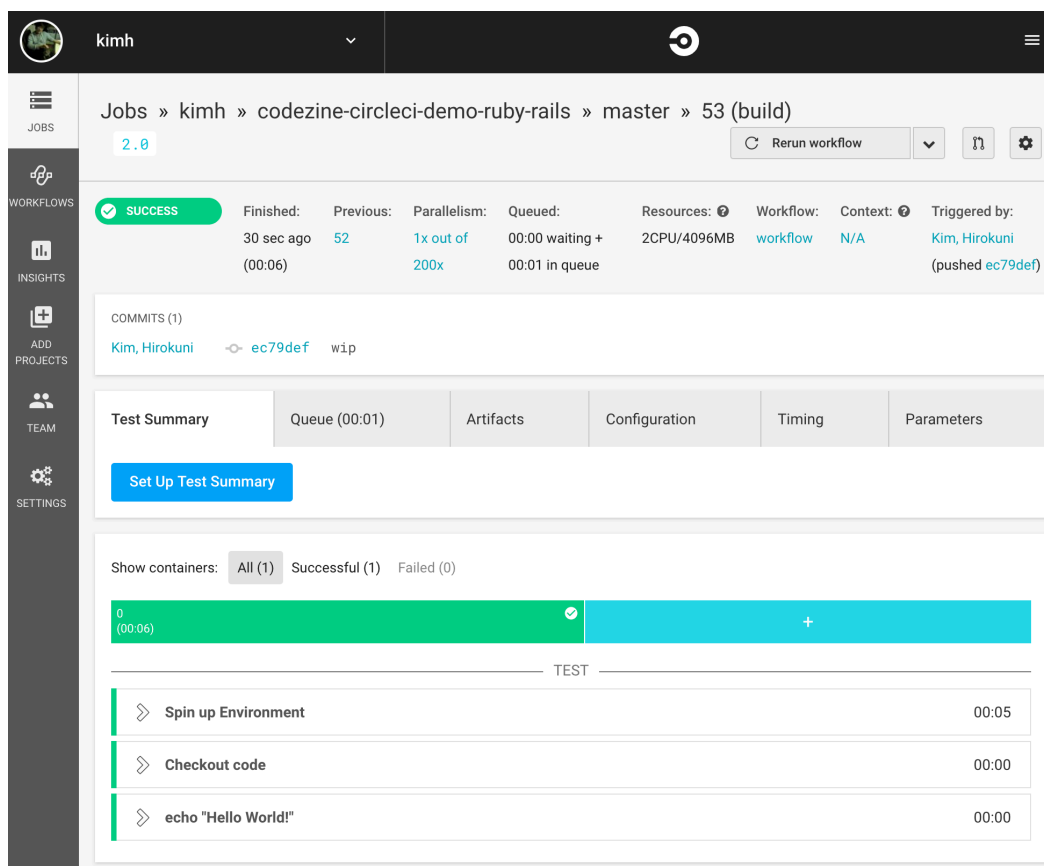
# Hello World

それではCircleCIを実際に使ってみましょう。入門としてHello Worldを表示させるところから始めることは、CI/CDでも例外ではありません。なお、本記事ではページ数の関係でCircleCIの[公式ドキュメント](#)の力を借りながら進めていきます。

CircleCIを使い始めるためには、まずはプロジェクトを追加する必要があります。詳しい説明は[公式ドキュメント](#)にゆずりますが、大まかに以下のような流れになります。

- [こちら](#)からサインアップしてGitHubかBitbucketを選択します。
- サインアップが完了するとビルドするプロジェクトを選びます。
- CircleCIが各言語にそっておすすめのビルドの設定を表示します。一から設定を書く手間が省けるのでCircleCIを初めて使う方は参考にしてください。
- “Start Building”というボタンを押せばビルドが開始します。

問題なくビルドが実行されれば以下の画像のように表示されます。



## 設定ファイル: .circleci/config.yml

次にCircleCIの設定方法を学んでいきましょう。次の設定はCircleCIで”Hello World!”と表示させるだけの設定です。

```
version: 2.0
jobs:
  build:
    docker:
      - image: circleci/node:4.8.2
    steps:
      - checkout
      - run: echo "Hello World!"
```

たったこれだけの設定ですが、アプリケーションのコンパイルやテストはなくてもCircleCIはちゃんとビルドとして処理してくれます。ユーザーが設定したようにビルドを実行する、2.0の柔軟さがこの設定でも垣間見えます。

### YAMLについて

YAMLは主に配列、ハッシュ、文字列や数字を表すスカラーの3つデータ構造を使って記述していきます。配列はハイフンと半角スペースを使います。

```
- foo
- bar
=> ["foo", "bar"]
```

ハッシュはコロンと半角スペースを使います。

```
- foo
- bar
=> {"foo", "bar"}
```

配列とハッシュをネストすることもできます。

```
jobs:
  build:
    docker:
      - image: circleci/node:4.8.2

=> {"jobs": {
  "build": {
    "docker": [{"image": "circleci/node:4.8.2"}]
  }
}}
```

YAMLは奥が深いデータ構造です。詳しく知りたい方は[この連載](#)を読むことをおすすめします。また、`.circleci/config.yml` がちゃんと書けているか確かめたい時はオンラインの[YAMLパーサー](#)などを使うとよいでしょう。

1つずつ説明していきます。まず、前提としてCircleCIの設定のほとんどは `.circleci/config.yml` というYAML形式のファイルに書きます。`.circleci/config.yml` をプロジェクト配下に置くと、CircleCIはここから設定を読み込んで、それに従ってビルドを実行していきます。

- `version: 2` : この設定ファイルがCircleCI 2.0対応ということを宣言しています。
- `jobs` : ジョブ(詳しくは後述)の設定をここ以降に書きます。
- `build` : `build` というジョブを設定していきます。
- `docker` : ビルドで使うDockerイメージを指定します。前述したようにCircleCIはDockerをネイティブサポートしているので、好きなDockerのイメージを使ってビルドの設定ができます。
- `steps` : ステップとは実際に実行されるコマンドやビルドのアクションです。それらを `steps` 配下を書いていくことでビルドの設定をしていきます。

### ステップについて

ステップには大きくわけて、ユーザーが任意のコマンドを実行できるステップと、CircleCIがあらかじめ用意しているビルトインステップがあります。前者はすべて `run:` ステップで書きます。

上記の例だと、シェルで `echo "Hello World!"` と実行することができます。詳しくは後述しますが、CircleCIの基本はこのrunステップを組み合わせてビルドの設定を作っていきます。

`checkout` はビルドインステップの一つです。GitHubやBitbucketなどのVCSからコードをダウンロードします。ビルドインステップはCircleCIがあらかじめ用意したCI/CDをする上で必須の処理をまとめたものです。すべてのステップは[ここ](#)から確認できます。よく使うステップについては本記事の中で紹介していくので、任意コマンドを実行するための `run` ステップとビルトインステップの2種類があることを覚えておいてください。

## 実際のアプリケーションの例

CircleCIの設定のイメージはつかんでもらえたでしょうか？ Hello Worldではもの足りないので、今度はもっと実践的な例で詳しく説明していきます。

コードは<https://github.com/kimh/codezine-circleci-demo-ruby-rails>にあります。これはRuby On Railsで書かれた簡単なログアプリです。このレポジトリを自分のGitHubアカウントにフォークして上記の手順に従いプロジェクトとして追加してください。

以下の設定はこのログアプリをCircleCI上でビルドするための最低限の設定です。すこし項目が多いですが順番に説明していきます。

```
version: 2
jobs:
  build:
    working_directory: ~/circleci-demo-ruby-rails
    docker:
      - image: circleci/ruby:2.4.1-node
        environment:
          RAILS_ENV: test
      - image: circleci/postgres:9.4.12-alpine
    steps:
      - checkout

      - restore_cache:
        keys:
          - v1-rails-demo-{{ checksum "Gemfile.lock" }}
          - v1-rails-demo

      - run: bundle install --path vendor/bundle

      - save_cache:
        key: v1-rails-demo-{{ checksum "Gemfile.lock" }}
        paths:
          - vendor/bundle

      - run:
        command: |
          bundle exec rake db:create
          bundle exec rake db:schema:load

      - run: bundle exec rspec
version: 2
jobs:
  build:
```

`build` というジョブの定義しています。CircleCIではビルドの設定をジョブという単位に分けて管理します。ジョブに分けるメリットを体験するには次回解説するワークフローを理解しないといけないので、ここでは `build` というジョブにビルドの設定を入れていくということだけ覚えておいてください。

```
working_directory: ~/circleci-demo-ruby-rails
```

`working_directory`: `steps` に書かれているコマンドをどのディレクトリで実行するかを指定します。指定しない場合は `~/project` が使われますが、明示的に `~/<リポジトリ名>` と指定するとよいでしょう。

```
docker:
  - image: circleci/ruby:2.4.1-node
    environment:
      RAILS_ENV: test
  - image: circleci/postgres:9.4.12-alpine
```

`docker`: このジョブがDocker Executorを使うということを定義しています。Executorとはビルドの実行環境のことです。

### その他の実行環境

CircleCIではDockerの他にiOSやmacOSをビルドするために使われる `macos` やVM環境でビルドするための `machine` を実行環境として指定できますが、有料プランのみの機能となるため今回は説明しません。詳しくは [公式ドキュメント](#) をご確認ください

`image`: 使用するDockerのイメージを指定します。ここで指定したイメージ上でコマンドが実行されます。CircleCIではDocker Hubのパブリック／プライベートだけではなく、その他のレジストリもサポートしています。例えば、DockerHubのプライベートなイメージを使う場合認証情報は `auth` を使って以下のように設定します。

```
docker:
  - image: kimh/my-private-image
    auth:
      username: mydockerhub-user
      password: $DOCKERHUB_PASSWORD
```

パスワードを直接書いてしまうのはまずいので、`$DOCKERHUB_PASSWORD` のように環境変数に設定しています。環境変数については後ほど `run` ステップのところで詳しく解説します。

```
environment:
  RAILS_ENV: test
```

このイメージに環境変数を設定しています。ここで指定した環境変数はこのイメージで実行されるコマンドのすべてで共有されます。今回の場合だと `RAILS_ENV: test` と指定しているのでRailsのテストを実行する際に `RAILS_ENV=test <test-command>` のように毎回環境変数を指定する必要がありません。

```
- image: circleci/postgres:9.4.12-alpine
```

2つ目のDockerイメージを指定しています。CircleCIではジョブで使うイメージを複数指定することができます。1つ目に指定したイメージ(ここでは `circleci/ruby:2.4.1-node`) はプライマリイメージと呼ばれ、この上で実際のテストやビルドのコマンドが実行されます。それ以降はサービスイメージと呼ばれ、アプリケーションが必要なデータベースなどのサービスを提供するために使われます。今回の例だと、ログアプリを動かすためにPostgreSQLが必要なのでサービスイメージで指定しています。なお、サービスイメージは複数指定できるので、例えばビルドの中で複数のデータベースを使ったりもできます。

```
steps:
```

`steps`:ビルドの中で実行される各処理をステップと呼びます。これ以降の階層にステップを定義していきます。

```
- checkout
```

`checkout`:ソースコードをVCS (GitHubやBitbucket) からダウンロードします。通常コードがないとビルドを始められないので、最初を書いておくといでしょう

## 依存関係のキャッシュについて

以下の設定では依存関係をインストールしています。毎回ジョブを実行するたびに依存関係を最初からインストールするのは無駄なので、CircleCIでは依存関係のキャッシュをサポートしています。

```
- restore_cache:
  keys:
    - v1-rails-demo-{{ checksum "Gemfile.lock" }}
    - v1-rails-demo

- run: bundle install --path vendor/bundle

- save_cache:
  key: v1-rails-demo-{{ checksum "Gemfile.lock" }}
  paths:
    - vendor/bundle
```

`save_cache` でキャッシュをアップロードして `restore_cache` でダウンロードしてします。`restore_cache` の直後に `bundle install` で依存関係をインストールしていることに注目してください。Ruby On Railの標準パッケージマネージャーであるBundlerはすでにダウンロードされている依存関係はスキップして、新しい依存関係だけをダウンロードしてくれます。つまり、`restore_cache` でダウンロードしたキャッシュに含まれなかった依存関係だけをインストールします。

最後に `save_cache` でインストールした依存関係をアップロードします。こうすることで最新の依存関係をキャッシュできます。Bundlerに限らず最近のパッケージマネージャーであれば同じような挙動をするので、`restore_cache` ⇒ 依存関係のインストールコマンド⇒ `save_cache` はCircleCIでもっともよく使われるパターンの一つです。

`save_cache` と `restore_cache` について詳しく見ていきましょう。どちらのステップにも `v1-rails-demo-{{ checksum "Gemfile.lock" }}` と書かれています。これはキャッシュする依存関係に対して一意なキーを設定しています。もう少し噛み砕いて言うと、CircleCIではキャッシュは単に依存関係をtarでまとめた1つのファイルで、`restore_cache` でどのファイルをダウンロードするか指定するために、一意なキーで名前をつけています

`v1-rails-demo-` は文字列リテラルです。`{{ checksum "Gemfile.lock" }}` はCircleCIが特別に用意してあるテンプレートで、指定したファイルのチェックサムに展開されます。具体的にはファイルのSHA256ハッシュを取って、それをBase64でエンコードした値になります

以下は、`v1-rails-demo-{{ checksum "Gemfile.lock" }}` で指定したキャッシュが実際に使われているところです。`bundle install` で `Using...` となっていることに注目してください。依存関係を新たにダウンロードせずにキャッシュからインストールしています。

#### Restoring Cache

00:01

```
Found a cache from build 7 at v1-rails-demo-pXnA+Fb8SnE0G+Id0
9aec+4B1DSQtQgtxx_FawTtYrU=
Size: 42 MB
Cached paths:
  * /home/circleci/circleci-demo-ruby-rails/vendor/bundle

Downloading cache archive...
Validating cache...

Unarchiving cache...
```

#### bundle install --path vendor/bundle

00:00

```
$ #!/bin/bash -eo pipefail
bundle install --path vendor/bundle

Using rake 12.0.0
Using concurrent-ruby 1.0.5
Using i18n 0.8.1

Exit code: 0
```

もう一つ重要なポイントはCircleCIは依存関係をダウンロードする時に最もマッチするキーを優先的に使おうとします。分かりやすいように具体例で説明します。

1. Gemfile.lockのチェックサムが`abc123`だったとします(実際はBase64なのでもっと長いです)。
2. CircleCIは`v1-rails-demo-abc123`というキーのキャッシュがすでに保存されているかチェックします。
3. もしあればその依存関係をビルドにダウンロードします。
4. なければ次のキーである`v1-rails-demo`に前方マッチするキャッシュがあればダウンロードします。

ここまで理解できれば、`save_cache`の動きは簡単に理解できるのではないのでしょうか? `key`でアップロードするキャッシュに対してキーを決めます。`paths`はキャッシュする依存関係があるディレクトリを指定しています。

以上がキャッシュに関する基本的な説明です。実はCircleCIには`checksum`以外にも使えるテンプレートが用意されています。紙面の関係ですべてを紹介することはできませんが、コミットのSHAに展開される`{{ .Revision }}`や任意の環境変数に展開される`{{ .Environment.variableName }}`などもあります。詳しくは[公式ドキュメント](#)をご覧ください。

## Runステップでコマンドを実行する

テストの実行などのコマンドは`run`ステップで実行します。

```
- run:
  command: |
    bundle exec rake db:create
    bundle exec rake db:schema:load

- run: bundle exec rspec
```

### YAMLでの改行

`command:`の直後の`|`は複数行を書く時に便利です。こうすると、改行されていても、1つの文字列として扱われます。YAMLの標準機能の一つです。



上の例ではテストデータベースをセットアップするコマンドとテストコマンドを `run` で実行しています。`run` ステップで指定したコマンドは新しいシェルを介して実行されます。使うイメージにインストールされていれば、`/bin/bash` が、なければ `/bin/sh` が使われます。

ここで注意したい点は、各 `run` ステップは新しいシェル上で実行されるということです。つまり、前とその後の `run` ステップでは環境が引き継がれません。例えば以下のようにしてもうまく動きません。

```
- run: export Foo=foo
- run: echo $FOO # $FOOには何も入っていない
```

このようなことを実現するためにCircleCIでは環境変数がサポートされています。使い方は2通りあって、`.circleci/config.yml` に書く方法とプロジェクトの設定画面を使う方法があります。

`.circleci/config.yml` で設定する場合は `environment` を使います。3つの設定方法があります。

## 1: ジョブのトップレベルキーとして指定する

```
version: 2
jobs:
  build:
    environment:
      FOO: bar
```

この方法だと `build` ジョブの中のすべてのステップからFOOが参照できるようになります。

## 2: イメージの中で指定する

```
- image: postgres:9.4.1
  environment:
    POSTGRES_USER: root
```

ある特定のイメージだけに環境変数を指定したい場合には `image` で指定することもできます。

## 3: runで指定する

```
- run:
  environment:
    RAILS_ENV: "test"
  command: |
    bundle exec rspec # RAILS_ENV=test bundle exec rspecと同じ
```

`run` の中だけで有効な環境変数も設定できます。

### 環境変数の展開は行えない

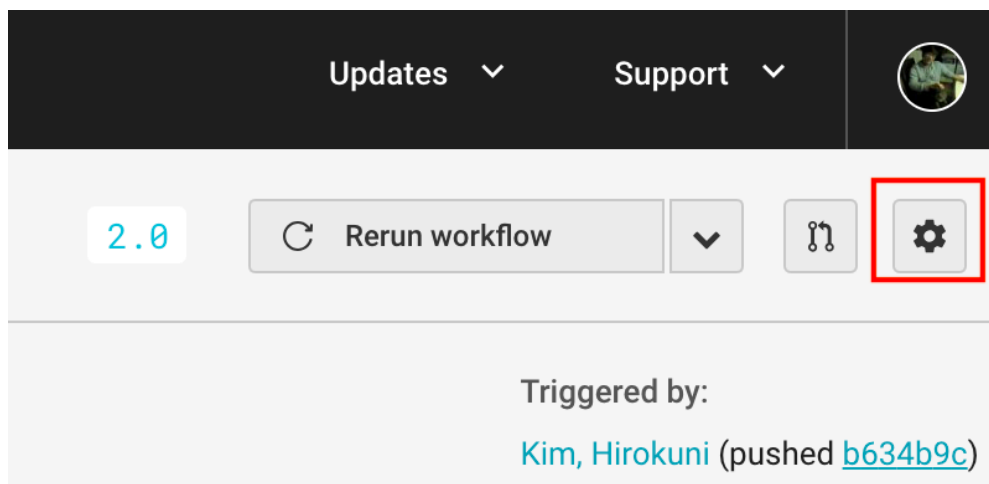
CircleCIでは `environment` 内での変数展開はサポートされていません。

```
environment:
  PATH: $PATH:/sbin
```

つまり、上記のようにすると値は文字列として解釈され、結果、PATHに `$PATH:/sbin` という文字列がセットされてしまいます。CircleCIで最もよく陥る間違いの一つなので気をつけてください。

`.circleci/config.yml` はソースコードと一緒にレポジトリで管理されるので、パスワードのような内容を隠したい環境変数は書くことができません。そのような場合はプロジェクト設定画面から追加しましょう。

右上にあるギアのアイコンをクリックするとプロジェクトの設定ページへ行ることができます。そこから、`Environment Variables` をクリックすると、ここで追加した環境変数を使うことができます。



例えばパスワードで認証が必要なサイトをcurlで取得していた場合、`$$SITE_PASSWORD` をプロジェクト設定から追加すれば以下のようにすることができます。

```
- run: curl -u my-user:$$SITE_PASSWORD http://hoge.com
```

本格的にCI/CDの設定をする場合、環境変数の利用は避けられないのでぜひマスターしてください。もっと詳しく知りたい場合は[公式ドキュメント](#)読むことをおすすめします。

## Ver. 2.1

さて、ここまでCircleCIの設定方法の基本について若干駆け足で紹介しました。

実は、CircleCIの設定はバージョンを指定することができ、ここまで紹介した設定方法はすべて2.0です(`.circleci/config.yml` の先頭に `version: 2.0` と書いたことを思い出してください)。

ここからはまだ公開されたばかりの2.1を解説していきたいと思います。(2018/10月執筆時)

### Ver. 2.0の問題点

Ver. 2.0で設定の柔軟性が格段に向上しましたが、問題点もありました。2.0ではユーザーがCI/CDのほぼすべての設定を書くので、`.circleci/config.yml` が肥大化しがちです。筆者は2000行(!)を超える `.circleci/config.yml` を見たこともあります。また、CircleCIとYAML両方の制限により設定の再利用性があまりありません。

Ver. 2.1には2.0をベースとし、これらの問題を解決するためのいくつかの構文が追加されました。

### 2.1の仕組み: Build Processing

2.1ではBuild Processingという仕組みが導入されました。これは、内部的なアーキテクチャのアップデートで、これにより `.circleci/config.yml` でシンタックスシュガーが使えるようになりました。2.1で導入された新しい構文は基本シンタックスシュガーなので、ビルドの実行時にすべて2.0の設定に展開されます。2.1のビルドのページのConfigurationというタブを見るとそれが分かります。

```

22     - run:
23       command: |
24         bundle exec rake db:create
25         bundle exec rake db:schema:load
26     - run:
27       command: bundle exec rspec
28 workflows:
29   version: 2
30   workflow:
31     jobs:
32     - build
33
34   # Original config.yml file:
35   # version: 2.1
36   # commands:
37   #   run-rails-test:
38   #     description: \"Rails\u306E\u30C6\u30B9\u30C8\u3092\u5B9F\u884C\"
39   #     steps:
40   #       - run:
41   #         command: |
42   #           bundle exec rake db:create
43   #           bundle exec rake db:schema:load
44   #       - run: bundle exec rspec

```

それでは、2.1で新たに追加された設定を見ていきましょう。

## Commands

`commands` は複数のステップを1つにまとめて再利用する方法です。

```

version: 2.1
commands:
  run-rails-test:
    description: “Railsのテストを実行”
    steps:
      - run:
        command: |
          bundle exec rake db:create
          bundle exec rake db:schema:load
      - run: bundle exec rspec

```

上記の例ではテストDBの作成とテスト実行を1つにまとめた `run-rails-test` というCommandsを定義しています。定義したコマンドは通常のステップのように使うことができます。

```

steps:
  - run-rails-test

```

今回の設定ではCommandsを使うメリットは分かりづらいかもしれませんが、次回に説明するワークフローを使い始めると、異なるジョブでなんども同じコマンドを実行する必要がある場合Commandsが役に立ちます。

## Parameters

Commandsにはパラメータを渡すこともできます。 `parameters` を使います。

```

commands:
  say-greeting:
    description: “パラメータの例”
    parameters:
      greeting:

```

```
    type: string
    default: "Hi!"
  steps:
    - run: echo << parameters.greeting >>
```

これも実践的な例ではないのですが、挨拶をするだけのCommandsを定義して、挨拶の言葉をパラメータで渡しています。`parameters` キー配下に実際のパラメータを定義します。ここでは `greeting` というパラメータを定義しています。`type` は渡すパラメータの型で、おなじみの文字列型 (string) と真偽型 (boolean) や別のステップを渡すためのステップ型 (step)、また選択肢を指定できる列挙型 (enum) が現在サポートされています。`default` は何も渡さなかった時のデフォルトです。

```
steps:
  - say-greeting:
    greeting: "Good morning!"
```

## When: ステップの条件実行

`when` はステップを条件付きで実行したい時に使います。以下はもっとも簡単な例です。

```
steps:
  - when:
    condition: true
    steps:
      - run: echo "hello!"
```

条件分岐の判定を直接書いても意味がないので今度は `parameters` を使って分岐するようにします。

```
myjob:
  ...
  parameters:
    preinstall-foo:
      type: boolean
      default: false
  steps:
    - when:
      condition: << parameters.preinstall-foo >>
      steps:
        - run: echo "hello!"
  ...

workflows:
  workflow:
    jobs:
      - myjob:
        preinstall-foo: false
      - myjob:
        preinstall-foo: true
```

`myjob` を定義してワークフローで同じジョブを2度実行しています。1回目は `preinstall-foo` がfalseなので `echo "hello!"` は実行されませんが、2回目はtrueなので実行されます。

なお、見慣れない `workflows` という設定がありますが、ジョブやワークフローについては次回で詳しく説明します。

## Executors

`executors` はジョブを実行する環境を簡潔に書くことができます。

```
executors:
  my-executor:
    docker:
      - image: circleci/ruby:2.5.1-node-browsers

jobs:
  job-one:
    executor: my-executor
    steps:
      ...

  job-two:
    executor: my-executor
    steps:
      ...
```

2.1以前では上記のように `job-one` と `job-two` を定義する場合、`docker` の定義をそれぞれのジョブに書かないといけませんでしたが、2.1では `my-executor` でまとめることができます。

## Pre/Postステップ

似たようなジョブを複数定義する場合、ほとんど同じだけど少しだけ違う挙動をさせたい場合があります。このような場合、pre/postステップで簡単に微調整できます。

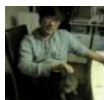
```
workflows:
  build:
    jobs:
      - job-one:
          pre-steps:
            - run:
                command: echo "this is pre!"
      - job-two:
          post-steps:
            - run:
                command: echo "this is post!"
```

## まとめ

CircleCIを設定していくイメージを掴んでもらえたでしょうか？ [第1回](#)の記事でも書きましたが、CI/CDはプロジェクトの初期段階で導入するのがおすすめです。今回紹介した機能を使えばCircleCIを使って基本的なCI/CDを始めることができるので、この記事を参考にして導入してみてもいいでしょうか？

[第3回](#)はもっと高度なCI/CDを可能にするワークフローを解説するのでご期待ください。

### 著者プロフィール



#### 金 洋国 (CircleCI Japan) (キムヒロクニ)

CircleCIで2.0などのプロダクト開発に携わった後、CircleCI Japanを立ち上げてからはTech Leadとして技術全般を担当。趣味は電動キックボードで日本で普及するように様々な活動をしています。

※プロフィールは、執筆時点、または直近の記事の寄稿時点での内容です  
Article copyright © 2018 Hirokuni Kim, Shoeisha Co., Ltd.



無料で始める circleci.jp