KLab TechBook

クラブテックブック

vol. 16

1 Python の抽象基底クラスについての紹介

2 SQLによるDBマイグレーションの再設計

3 TypeScript + GitHub Actions + Grafana Cloud で実現する現代的GAS開発環境

KLab Tech Book Vol. 16

KLab 技術書サークル　著

2025-11-15 版　　KLab 技術書サークル 発行

はじめに

このたびは本書をお手に取っていただきありがとうございます。本書は KLab 株式会社の有志にて作成された KLab Tech Book の第 16 弾です。

KLab では主にモバイルオンラインゲームを開発していますが、KLab Tech Book では所属するエンジニアが日々の業務や個人の探求の中で見つけた技術的な興味や試行錯誤について書き連ねました。表紙のデザインも社内のデザイナーの方にご協力いただき、KLab 感溢れる一冊に仕上がっています。

今回は 3 つの記事を収録しました。章ごとに内容が独立しているので、気になるものから順に読み進めていただいて問題ありません。触れられている技術領域は様々ですが、前提知識がなくても読めるよう心がけました。知らない分野であっても軽い気持ちで読んでいただければ、新たな発見や世界の広がりを感じていただけるかもしれません。

本書を通して、技術や知識に触れる楽しさを感じていただけたらさいわいです。

牧内　大輔

お問い合わせ先

本書に関するお問い合わせは tech-book@support.klab.com まで。

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

目次

第 1 章

Python の抽象基底クラスの紹介

Shunsuke Ito / @fgshun

1.1 はじめに

Python の抽象基底クラス（abstract base class; ABC）たちについて紹介します。抽象基底クラスとはなんなのか。そして公式に実装されているクラスたちにはどのようなものがあるのか、どのように役立てることができるのかを紹介します。

1.2 抽象基底クラスとは

あるクラスが特定の操作をサポートしているかを確認するにはどのような方法がありえるでしょうか？ たとえば継承。ある子クラスのインスタンスは、親クラスと同じ操作に対応しています。たとえばダックタイピング。ある名前のメソッドを持っているクラスが複数あれば、それらは同じ操作に対応しているのでしょう。

Python3 ではまた別の仕組みが用意されています。それが抽象基底クラスです。前述の方法たちのいいとこ取りです。継承して親子関係を構築する、 register メソッドで仮想の親子関係を登録する、特定のメソッドを持っている、いずれの場合でも `isinstance`, `issubclass` が True を返し、同一視できる仕組みとなっています。

抽象基底クラスは `abc.ABC` クラス[*1]を継承することで作成することができます。たとえば、クラス `Named` は リスト 1.1 のようにして作成します。そして、クラス `Spam`, `Ham`, `Eggs` はいずれも `Named` との `isinstance` チェックを通ります。`Spam` は `Named` を直接継承しているからです。`Ham` は `Named.register` により仮想の親子関係ができているからです。`Eggs` は `get_name` メソッドを持っているからです。

[*1] abc: `https://docs.python.org/ja/3.13/library/abc.html`

1.2 抽象基底クラスとは

リスト 1.1: 抽象基底クラスの例

```
from abc import ABC, abstractmethod

class Named(ABC):
    @classmethod
    def __subclasshook__(cls, sub):
        # get_name メソッドを持っているかをチェックする
        if cls is Named:
            if any('get_name' in B.__dict__ for B in sub.__mro__):
                return True
        return NotImplemented

    @abstractmethod
    def get_name(self): return 'Spam'

class Spam(Named):
    def get_name(self): return super().get_name() * 3

class Ham:
    pass
Named.register(Ham)  # 仮想の親子関係の登録

class Eggs:
    def get_name(self): return 'Eggs'
    # get_name メソッドがあるので Named.__subclasshook__ が True を返す

print(isinstance(Spam(), Named))  # True
print(isinstance(Ham(), Named))  # True
print(isinstance(Eggs(), Named))  # True
```

抽象基底クラスには抽象メソッドを持たせることができます。これを継承して作った具象クラスが実装すべきメソッドです。抽象メソッドは abc.abstractmethod デコレーターを用いて作成することができます。抽象メソッドをもったままであるクラスはインスタンス化することはできません。実行時エラーとなります。 このことにより抽象メソッドを実装し忘れていることを検知できます[*2]。

[*2] なお、このエラーは register による仮想の子クラスや __subclasshook__ が True を返す仮想の子クラスでは発生しません。あくまで直接継承して抽象メソッドを持っているクラスに限られます。

リスト 1.2: 抽象メソッドを持つクラスのインスタンス化はできない

```
# x = Named()  # TypeError をおこす。インスタンス化不可能。
a = Spam()  # こちらは get_name を実装済みなので可能。
```

抽象メソッドを持ったクラスはインスタンス化できないので、抽象メソッドを自クラスのインスタンスから呼ぶことはできません。それでも、抽象メソッドを子クラスから super 越しに呼ぶことは可能です。

リスト 1.3: 抽象メソッド Named.get_name を子クラス Spam から呼ぶ

```
#  print(Named().get_name())  # Named のインスタンスは得られていない
print(Spam().get_name())  # SpamSpamSpam
```

1.3 コンテナの抽象基底クラス

実用に耐える抽象基底クラス群を設計することは難しいものです。Python のこと、実装しようとしている対象のこと、双方の知識が問われます。ところで、Python 標準ライブラリには便利な抽象基底クラスが用意されています。これらがそのまま使えるのであれば、設計・実装という難題から解放されるわけです。そんな既存の抽象基底クラスを紹介していきます。

まずはコンテナ関連です。コンテナたちがどのような処理に対応していて、どこまでを同一視できるのかの判定のために collections.abc モジュール[*3]の抽象基底クラスが存在しています。

コンテナの基本的な機能として、いくつの要素を持っているか問う (len 関数に対応する __len__ メソッド)、要素をひとつずつ返すイテレータを得る (__iter__ メソッド)、ある要素を含んでいるか問う (in 演算子に対応する __contains__ メソッド)、といったものがあります。これらの機能を表現する抽象基底クラスとして Sized 、 Iterable 、 Container が用意されています。これらには対応するメソッドが抽象メソッドとして定義されています。

[*3] collections.abc: https://docs.python.org/ja/3.13/library/collections.abc.html

1.3 コンテナの抽象基底クラス

リスト 1.4: Sized の例

```
from collections.abc import Sized

class Zero(Sized):
    def __len__(self): return 0
class One:
    def __len__(self): return 1

assert issubclass(Zero, Sized)
assert issubclass(One, Sized)  # 継承していなくともメソッドを持っているだけでよい
```

そして、これら 3 つの抽象基底クラスを継承した、 3 つの機能を併せ持つ抽象基底クラス Collection があります。collections.abc ではこのようにシンプルなクラスを組み合わせで、さまざまなコンテナが表現されています。

リスト 1.5: Collection の例

```
from collections.abc import Collection

class Cons(Collection):
    def __init__(self, car, cdr):
        self.car = car
        self.cdr = cdr
    def __len__(self):
        return 2
    def __iter__(self):
        return iter((self.car, self.cdr))
    def __contains__(self, item):
        return self.car == item or self.cdr == item

ab = Cons('A', 'B')
assert isinstance(ab, Collection)
# Collections は Sized, Iterable, Container の機能を持つ
assert isinstance(ab, Sized)
assert isinstance(ab, Iterable)
assert isinstance(ab, Container)
# Collection にできることたち
assert len(ab) == 2  # Sized にできること
assert tuple(ab) == ('A', 'B')  # Iterable にできること
assert 'A' in ab  # Container にできること
```

Collection に要素の順序という概念を加えると Sequence になります。__getitem__ メソッドで N 番目の要素を返す、つまりインデックスアクセスができるようにする。そうすれば、 順序という概念を持つコンテナであるなら持つべき index などのメソッドを実装したコンテナクラスが得られます。

リスト 1.6: Sequence の例

```
from collections.abc import Sequence

class Spam(Sequence):
    def __init__(self):
        self.spam = 'spam'
        self.ham = 'ham'
        self.eggs = 'eggs'
    def __len__(self):
        return 3
    def __getitem__(self, index):
        if index < 0: index += len(self)
        if index == 0: return self.spam
        elif index == 1: return self.ham
        elif index == 2: return self.eggs
        else: raise IndexError

spam = Spam()
assert isinstance(spam, Sequence)
# Collection にできることたち
assert len(spam) == 3
assert tuple(spam) == ('spam', 'ham', 'eggs')
assert 'spam' in spam
# Sequence にならばできることたち
assert spam[0] == 'spam'
assert spam[-1] == 'eggs'
assert spam.index('spam') == 0
assert spam.count('spam') == 1
assert tuple(reversed(spam)) == ('eggs', 'ham', 'spam')
```

注目すべき点としては Collection 関連の抽象メソッドを Sequence の例では実装していないという点です。これは、 Sequence にはインデックスアクセスができることを用いて全探索を行うデフォルト実装が存在しているからです。もし、そのほかの前提、たとえば N 分探索木で作られている、などといった前提があったとすれば最高効率の実装ではないものではありますが、インデックスアクセスができるという最小限の前提にのみ依存したデフォルト実装です。これにより、とりあえず動かし始める、これで十分な性能が出て役目を果たせるのであれば追加の実装を行う必要はないという便利なものです。このように、 collections.abc の抽象基底クラスには適切な継承関係、適切な抽象メソッド、最小限の前提に則ったメソッド

のデフォルト実装が含まれています。

Sequence に要素の変更・追加・削除処理を加えると MutableSequence となります。次の例では前に例示した Spam クラスに変更処理を加えた MutableSpam クラスを作る例となります。なお、 __delitem___ および insert はデフォルト実装を呼び出しています。MutableSequence のデフォルト実装は IndexError 例外を送出するというものになっています。つまり、一部の機能をサポート対象外としたままとする、などということが可能であるということです。MutableSequence としては風変わりな挙動ではあるので、ドキュメントやユニットテストでこのことを説明しておくのが無難ではあります。

リスト 1.7: MutableSequence の例

```
from collections.abc import MutableSequence

class MutableSpam(Spam, MutableSequence):
    def __init__(self):
        super().__init__()
    def __getitem__(self, index):
        return super().__getitem__(index)
    def __setitem__(self, index, value):
        if index < 0: index += len(self)
        if index == 0: self.spam = value
        elif index == 1: self.ham = value
        elif index == 2: self.eggs = value
        else: raise IndexError
    def __delitem__(self, index):
        # 削除はサポートしない
        super().__delitem__(index)
    def insert(self, index, value):
        # 挿入はサポートしない
        super().insert(index, value)

spam = MutableSpam()
assert spam[0] == 'spam'
spam[0] = 'foo'
assert spam[0] == 'foo'
assert tuple(spam) == ('foo', 'ham', 'eggs')
# 変更可能な順序付きコンテナであればインプレースで反転させることができる
spam.reverse()
assert tuple(spam) == ('eggs', 'ham', 'foo')
# pop を試みることはできるが、削除をサポートしていないので
# __delitem__ 由来の実行時エラーを起こす
spam.pop()  # IndexError
```

おわりに、コンテナの抽象クラスたちを列挙します。

Sequence

インデックスアクセスができ、要素が順番を持つコンテナ。 ex: list, tuple

MutableSequence

Sequence に要素の変更ができる機能を追加したもの。 ex: list

Set

集合を表現するコンテナ。 ex: set, frozenset

MutableSet

Set に要素の変更ができる機能を追加したもの。 ex: set

Mapping

キーと値の組を保持するコンテナ。 ex: dict, types.MappingProxyType

MutableMapping

Mapping に要素の変更ができる機能を追加したもの。 ex: dict

1.4 数の抽象基底クラス

続いて、数の抽象基底クラスを紹介していきます。これらは `numbers` モジュール[*4]に含まれています。コンテナ同様、性質が少ないものからより多いものへと発展させていくという設計となっています。

まずは、数であるということを表すだけの抽象基底クラス `Number` です。これにはなんの機能もありませんが、これから紹介する抽象基底クラスはこのクラスを継承して作られています。数の具象クラス `complex`、 `float`、 `int` らは `register` メソッドによる仮想の親子関係を構築されているため、 `issubclass` のチェックが可能です。Pure Python 製の `Fraction` は数の抽象基底クラスを直接継承しているため `issubclass` のチェックが可能です。

リスト 1.8: Number による型チェック

```
from numbers import Number
from fractions import Fraction

assert issubclass(complex, Number)
assert issubclass(float, Number)
assert issubclass(Fraction, Number)
assert issubclass(int, Number)
assert not issubclass(str, Number)  # str は数ではない
```

[*4] numbers: `https://docs.python.org/ja/3.13/library/numbers.html`

1.4 数の抽象基底クラス

これに複素数である、実数である、有理数である、整数である、と性質を追加していくことを表す抽象基底クラスたちが用意されています。Complex、 Real、 Rational、 Integral です。実数は虚部が 0i の複素数、有理数は整数の比で表すことができる実数、整数は分母が 1 の有理数と言えます。このことを表現するためにこれらのクラスは親子関係を持っています。

リスト 1.9: 数の抽象基底クラスたち

```
from numbers import Complex, Real, Rational, Integral
assert issubclass(Real, Complex)
assert issubclass(Rational, Real)
assert issubclass(Integral, Rational)
```

そして具象クラスたちにもこの親子関係が波及します。たとえば、complex は複素数ですが、実数ではありませんし、有理数ではありませんし、整数でもありません。たとえば、int は複素数であり、実数でもあり、有理数でもあり、そして整数です。

リスト 1.10: 数の具象クラスに対するチェック

```
from numbers import Complex, Real, Rational, Integral
from fractions import Fraction

assert issubclass(complex, Complex)
assert issubclass(float, Complex)
assert issubclass(float, Real)
assert issubclass(Fraction, Complex)
assert issubclass(Fraction, Real)
assert issubclass(Fraction, Rational)
assert issubclass(int, Complex)
assert issubclass(int, Real)
assert issubclass(int, Rational)
assert issubclass(int, Integral)
```

数の抽象クラスたちは、これら 5 つで完結しているものではありません。拡張していくことができます。例として、整数的な動きをする実数ではあるが、 Integral が可能な操作の一部、べき乗やビット演算などをサポートしない抽象基底クラス Spam を作成したとします。この自作クラスを既存の仕組みに後から挟み込むことが可能です。次のコードにより Rational、 Spam、 Integral という親子関係が形成されます。

リスト 1.11: 有理数と整数の中間の性質のクラスを挟み込む

```
from numbers import Rational, Integral

class Spam(Rational):
    @property
    def numerator(self): return +self
    @property
    def denominator(self): return 1
Spam.register(Integral)
```

古典的なクラスの継承では、親子関係の途中に挟み込むには既存のクラスに手を加えなければいけません。こういった操作ができるのは抽象基底クラスならではです。

■コラム: Python の数たちの実装の実例、 Fraction と int

紹介した組み込みの数のクラスのうち、 `Fraction` は Pure Python 製であると記しました。実際に、 `Fraction` における `Rational` 抽象基底クラスの継承は class 文で行われています。

リスト 1.12: Fraction と Rational - cpython/lib/fractions.py より

```
class Fraction(numbers.Rational):
    """This class implements rational numbers.
```

一方、 `int` クラスは C 言語で実装されています。C 製クラスが C 製クラスを継承しようとする場合、 `PyTypeObject` 構造体の `tp_base` スロットに型構造体を設定するのがひとつの方法です。しかし Pure Python 製クラス `numbers.Integral` には対応する型構造体が存在しないのでこの方法を用いることができません。動的な型オブジェクトの生成が必要となります。もしこれを行おうとすれば実装に変更を入れなければならなくなるでしょう。その変更が他の箇所に悪影響を及ぼさないことを保証することは困難です。

`int` の親に `Integral` を設定するために行われていることは、 `int` の実装コードに手を入れることなく `numbers` モジュールで `Integral.register` を呼び出すことです。

リスト 1.13: int と Integral - cpython/lib/numbers.py より

```
Integral.register(int)
```

抽象基底クラスには変更を加えるのは困難なクラスの実装に触れずにクラスの親子関係を構築することができるというメリットがあります。

1.5 io の抽象基底クラス

Python は様々な種類の I/O を扱い、そのためのストリームを表現するためのクラスを定義しています。そのクラスたちを分類するために、 io モジュール[*5]にて抽象基底クラスが 4 つ定義されています。これらを紹介していきます。

まずは IOBase です。ストリームであることを示します。Number とは異なり、いくつかのメソッド、抽象メソッドが含まれています。たとえば、 __enter__、 __exit__ らコンテキストマネージャとしてのメソッドがここに用意されています。そのため、 I/O クラスはすべて with 構文で用いることが可能です。with 文の突入時に I/O が閉じていないことを確認し自身を返す、 with 文の脱出時に close メソッドを呼ぶというものとなっています。

リスト 1.14: iobase.c より、コンテキストマネージャ関連メソッドの実装

```
/* Context manager */

static PyObject *
iobase_enter(PyObject *self, PyObject *args)
{
    if (iobase_check_closed(self))
        return NULL;

    Py_INCREF(self);
    return self;
}

static PyObject *
iobase_exit(PyObject *self, PyObject *args)
{
    return PyObject_CallMethodNoArgs(self, &_Py_ID(close));
}
```

[*5] io: https://docs.python.org/ja/3.13/library/io.html

なので、組み込みの I/O 、たとえばファイルオブジェクトは with 文で用いることができる、というわけなのです。

リスト 1.15: ファイルオブジェクトを with 文で用いる

```
from io import IOBase
with open('spam.txt') as fobj:
    ...
```

ふたつめは、 RawIOBase です。生のバイナリストリームを表現するために用意された抽象基底クラスです。IOBase を継承し、 NotImplementedError を起こす readinto や write デフォルト実装を持ちます。

3 つめは、 BufferedIOBase です。生のバイナリストリームをラップした、バッファリングされたバイナリストリームを表現するための抽象基底クラスです。IOBase を継承しています。注目すべきは RawIOBase を継承はしていない、というところです。BufferedIOBase は生のバイナリストーム RawIOBase そのものではなく、 RawIOBase に移譲するためのものだからです。

4 つめは、 TextIOBase です。テキストストリームを表現するための抽象基底クラスです。BufferedIOBase に移譲してバイナリデータを読み書きしつつ、その間であるエンコーディングに沿ってバイト列、文字列間の相互変換を行うことを想定して設計されています。IOBase を継承しています。テキストストリームもストリームには違いないからです。一方、まともに機能するテキストストリームにはバッファリングされたバイナリストリームがいるものですが、そのものではないがために BufferedIOBase を継承はしていません。

これらの抽象基底クラスを用いて、 open テキストモードで作ったファイルオブジェクトをチェックしてみます。ファイルオブジェクトは TextIOBase であることがわかります。そして TextIOBase の底には buffer 属性越しにアクセスできる、 BufferedIOBase 、バッファリングされたバイナリストリームがあることがわかります。さらに BufferedIOBase の底には raw 属性越しにアクセスできる、 RawIOBase 、生のバイナリストリームがあることがわかります。

リスト 1.16: ファイルオブジェクトを isinstance でチェック

```
from io import *
with open('spam.txt') as fobj:
    print(isinstance(fobj, IOBase))  # True
    print(isinstance(fobj, RawIOBase))   # False
    print(isinstance(fobj, BufferedIOBase))   # False
    print(isinstance(fobj, TextIOBase))  # True
    print(isinstance(fobj.buffer, IOBase))  # True
    print(isinstance(fobj.buffer, RawIOBase))   # False
    print(isinstance(fobj.buffer, BufferedIOBase))   # True
    print(isinstance(fobj.buffer, TextIOBase))  # False
    print(isinstance(fobj.buffer.raw, IOBase))  # True
    print(isinstance(fobj.buffer.raw, RawIOBase))   # True
    print(isinstance(fobj.buffer.raw, BufferedIOBase))   # False
    print(isinstance(fobj.buffer.raw, TextIOBase))  # False
```

1.6 おわりに

Python の抽象基底クラスたちについて紹介しました。継承とダックタイピングのいいとこ取りをし、メソッドのデフォルト実装を提供したり、仮想の親子関係を後付けしたりできる面白い機構です。そして抽象メソッドにより、子がどのようなメソッドを持つべきか、つまりどのような機能・責務を持つべきかを示すことができます。これにより、設計をコードの形で残すことができます。ぜひ、抽象基底クラスを活用してみてください。

第 2 章

SQL による DB マイグレーションの再設計

Daisuke Makiuchi / @makki_d

2.1　スキーマ管理とマイグレーションツール

スキーマの管理、どうしてますか？ アプリケーションの開発が進むと同時に、DB のスキーマも変化していきます。開発が進むたび、あるいは Git のブランチを切り替えるたびに、どのような変更を DB に適用すべきかを手作業で追うのは簡単ではありません。単純な適用漏れで DB とコードの整合性が崩れ、アプリが正しく動かないなんてこともあります。本番環境でそんなことが起これば即障害です。

この問題に対処してくれるのが「マイグレーションツール」です。これは、スキーマの定義や変更内容をアプリと同じ言語や独自の DSL でコードとして記述し、それを DB へ適用することでアプリの状態と DB の状態を揃えてくれるツールです。これによってスキーマをアプリのコードと一緒にバージョン管理することもできます。

マイグレーションツールはフレームワークや ORM に組み込まれていることもありますし、独立したツールも存在します。また、ORM や DSL でスキーマを記述することで、安全性を高めたり複数の RDB に対応しやすいというメリットもあります。

2.2　ORM/DSL の限界と SQL が必要になる場面

ORM や DSL は一見便利に見えるかもしれませんが、実際の運用ではそれだけでは対応しきれない場合があります。いくつか例を見てみましょう。

データコピーを伴うテーブルの変更

たとえば、ユーザーごとに 1 つのメールアドレスを登録できるサービスがあるとします。あるとき、「ユーザーが複数のメールアドレスを登録できるようにしたい」という仕様変更が入りました。

正攻法として正規化を行い、新しく emails テーブルを追加してユーザーとメールアドレスを 1:N で管理します。これを ER 図で表すと図 2.1 のようになります。

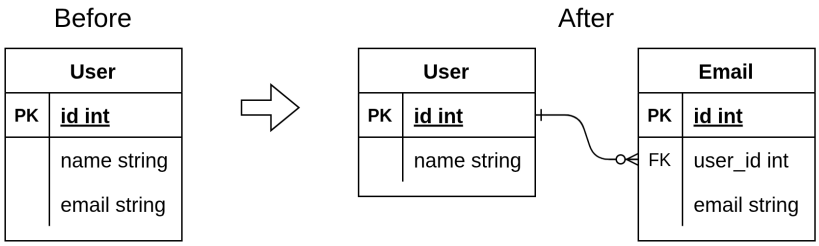

図 2.1: スキーマの変更

テーブルを追加したあと、users テーブルに登録済みのメールアドレスを新しい emails テーブルにコピーし、そのあとで古いカラムを削除します。これを PHP の Laravel[*1]でマイグレーションする場合、リスト 2.1 のようになるでしょう。

リスト 2.1: Laravel でのマイグレーション

```
return new class extends Migration
{
    public function up(): void
    {
        // emailsテーブルを作成
        Schema::create('emails', function (Blueprint $table) {
            $table->id();
            $table->foreignId('user_id')->constrained('users');
            $table->string('email');
        });

        // 既存のemailデータをemailsテーブルにコピー
        DB::table('users')
            ->whereNotNull('email')
            ->chunk(100, function ($users) {
```

[*1] https://laravel.com/

```
                foreach ($users as $user) {
                    DB::table('emails')->insert([
                        'user_id' => $user->id,
                        'email' => $user->email,
                    ]);
                }
            });

        // usersテーブルからemailカラムを削除
        Schema::table('users', function (Blueprint $table) {
            $table->dropColumn('email');
        });
    }

    public function down(): void
    {
        // 省略
    }
};
```

このマイグレーションスクリプトには、パフォーマンス上の問題があります。既存のメールアドレスを新テーブルにコピーするために、アプリケーション側でデータを取得して 1 件ずつ挿入しています。レコード数が多い場合、この処理は数分〜数十分かかることもあります。

SQL には INSERT INTO ... SELECT 構文があり、DB 内部で直接データをコピーすることができます。これを使った SQL によるマイグレーションはリスト 2.2 のように書けます。

リスト 2.2: SQL によるマイグレーション

```
-- emailsテーブルを作成
CREATE TABLE emails (
    id      int NOT NULL AUTO_INCREMENT,
    user_id int NOT NULL,
    email   varchar(255) NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY fk_user (user_id) REFERENCES users(id)
);

-- 既存のemailデータをemailsテーブルにコピー
INSERT INTO emails (user_id, email)
  SELECT id, email FROM users WHERE email IS NOT NULL;

-- usersテーブルからemailカラムを削除
ALTER TABLE users DROP COLUMN email;
```

この方法ならデータをアプリケーション側に転送する必要がなく、さらに挿入クエリを多数発行する必要もありません。また、DB 内部でコピー処理が完結していて実行も最適化されるため、件数が多くてもせいぜい数秒〜数十秒で完了します。

Laravel に限らず ORM や DSL の多くは、このような SQL 特有の構文をサポートしておらず、必要に応じて生の SQL を記述する方法に頼ることになります。結局のところ、性能要件を満たすためには SQL を直接書くことが求められます。

ダウンタイムなしのオンライン変更

ある機能をリリースした直後に、想定以上に DB 負荷が高まっていることがわかったとします。調べてみると、特定のテーブルにインデックスを付け忘れていたことが原因でした。単純にインデックスを追加すれば解消しますが、できればメンテナンスに入れることなく、まだレコード数が少ないうちにオンラインで追加してしまいたいと考えました。

さて、リスト 2.3 のようなマイグレーションは、果たしてオンラインでも安全に実行できるでしょうか。

リスト 2.3: Laravel でインデックスを追加するマイグレーション

```
public function up(): void
{
    Schema::table('highscore', function (Blueprint $table) {
        $table->index('user_id');
    });
}

public function down(): void
{
    // 省略
}
```

おそらく内部では単純な `ALTER TABLE` 文が発行されると思われます。しかし、それは本当にオンラインでの実行に耐えうるでしょうか。安全のために広い範囲のロックが掛けられたりしないでしょうか。このような懸念を払拭するには、最終的にどんな SQL が実行されるかを確認する必要があります。

さらに言えば、MySQL や PostgreSQL にはリスト 2.4 のようなテーブルへの読み書きをブロックせずにインデックスを追加できる構文が用意されています。これを使えばオンラインでも問題なく安全にインデックスを追加できるでしょう。

しかし、このような構文を ORM や DSL がサポートしていることはほとんどなく、SQL を記述することになります。

リスト 2.4: ブロックしないインデックスの追加

```
-- MySQLの場合。PostgreSQLでは"CREATE INDEX CONCURRENTLY"
CREATE INDEX idx_user ON highscore (user_id) ALGORITHM=INPLACE, LOCK=NONE;
```

ここまで 2 つの例を見てきましたが、性能や可用性を意識したマイグレーションを行うには、SQL を直接書く必要が出てきます。また、安全な DB 操作を実現するには、ORM や DSL で書かれたコードが最終的にどのような SQL として実行されるかを常に意識しなければなりません。

であるならば、最初から SQL ですべて記述したほうが合理的ではないでしょうか。そうすれば DB のすべての機能を利用できるうえ、どのような SQL が実行されるかも一目瞭然となり、挙動を完全に把握できます。

2.3 SQL によるマイグレーションの課題

これまで SQL でのマイグレーションを行いたくなる理由を述べてきました。しかし、世の中にあまり普及していないのはなぜでしょうか。Flyway[*2]や Goose[*3]のように SQL で記述するマイグレーションツールもありますが、あまり一般的ではありません。SQL によるマイグレーションには多くの課題があるからです。

ここではそれらの課題を紹介し、どのように対処したら SQL のみでマイグレーションが実現できるかを考えていきます。

アプリケーションコードとの連携

ORM やフレームワークのマイグレーション機能は、アプリケーションコードと密接に統合されています。テーブルの定義がそのままモデルクラスと直接対応していることも多く、コードの変更とスキーマの変更を同時に管理できることが高い利便性となっています。

一方で、SQL によるマイグレーションでは SQL がアプリケーションコードから独立するため、モデルとテーブル構造を自動的に同期させることができません。

しかし、よく考えてみてください。アプリケーションにとって必要なモデルは「クエリ結果

[*2] https://github.com/flyway/flyway
[*3] https://github.com/pressly/goose

を受け取るデータ構造」であって、必ずしもテーブル構造と一致している必要はないのです。リポジトリ層の内側がどのようなテーブル構造になっているかは、本来アプリケーションから隠蔽されるべきです。すなわち、このような連携の不足は欠点ではなく、むしろ「責務の分離」であり、あるべき姿ともいえます。

アプリケーションからスキーマの定義を切り離し、モデルクラスの責務を明確に分けることが健全な開発につながります。モデルの生成や型安全なアクセスが必要な場合、sqlc[*4]のようなクエリからコードを生成する ORM が良い解決策となるでしょう。

適用順序と履歴の管理

SQL でマイグレーションを記述する場合、スキーマに対する変更を積み重ねていくことになります。ここで問題になるのは、各変更をどの順序で適用するか、そしてどこまで適用したかをどのように管理するかという点です。管理を曖昧にしたままマイグレーションを実行すると、同じ変更を重ねて適用してしまったり、適用順序を誤って整合性が崩れてしまうおそれがあります。

既存の SQL 記述型のマイグレーションツールが解決しているのはこの問題です。これらのツールでよく用いられているのは、ファイル名にバージョン番号を含めることで順序を表し、DB に適用履歴テーブルを作って管理する方法です。SQL だけでマイグレーションを行う場合も、この考え方をそのまま取り入れることができます。ここでは MySQL 向けの具体的な手順を示します。

まず、適用履歴テーブルをリスト 2.5 のように定義します。

リスト 2.5: マイグレーション適用履歴テーブル

```
CREATE TABLE _migrations (
   id      INTEGER NOT NULL,
   applied DATETIME,
   title   VARCHAR(255),
   PRIMARY KEY (id)
);
```

`id` は適用順序を表す番号とします。各マイグレーションは必ず up と down のペアとし、`up.sql` にスキーマを更新する SQL を、`down.sql` にはそれを元に戻す SQL を記述します。

たとえば `id = 1` で、`create_tables` というタイトルのマイグレーションのファイルは次のようになります。

[*4] `https://github.com/sqlc-dev/sqlc`

- 000001_create_tables.up.sql
- 000001_create_tables.down.sql

2 つのファイルに分けているのは、SQL を適用するときにファイル単位で DB に適用できるようにするためです。ここまでは既存のマイグレーションツールと同様ですが、履歴テーブルへの挿入や多重適用の回避もツールに頼らず SQL で行うようにするために、ファイル先頭にいくつか SQL を記述します。

リスト 2.6: 000001_create_tables.up.sql

```
INSERT INTO _migrations (id, title, applied) VALUES (1, 'create_tables', NOW());
-- ここから下に適用したいSQLを記述
CREATE TABLE ... 省略
```

この SQL ファイルを DB に適用すると、最初に適用履歴テーブルへの挿入が行われます。このとき、すでに適用済みで同じ id のレコードが存在している場合、PRIMARY KEY 制約によってこの INSERT 文は失敗します。この時点で処理が停止するため、続く CREATE 文は多重に実行されることはありません。

一方で、down.sql では適用済みでレコードが存在するときのみ実行したいものです。しかし単純に SELECT するだけでは、結果が 0 件でもエラーとはならずに、そのまま次の SQL が実行されてしまいます。このような場合に処理を停止させるには、MySQL ではリスト 2.7 のようなストアドプロシージャを使用します。引数として id を受け取り、同一 id のレコードが存在しなかった場合に SIGNAL でエラーを発生させるものです。

リスト 2.7: マイグレーションが適用済みか確認するストアドプロシージャ

```
CREATE PROCEDURE _migration_exists(IN input_id INTEGER)
BEGIN
  IF NOT EXISTS (SELECT 1 FROM _migrations WHERE id = input_id) THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'migration not found';
  END IF;
END
```

down.sql の先頭でこのストアドプロシージャを使って適用済みかどうか判定し、適用済みだった場合にはレコードを削除してから目的の SQL を実行します。

リスト 2.8: 000001_create_tables.down.sql

```
CALL _migration_exists(1);
DELETE FROM _migrations WHERE id = 1;
-- ここから下に元に戻すSQLを記述
DROP TABLE ... 省略
```

このようにすることで、どのマイグレーションが適用されているかは適用履歴テーブルを見ればわかり、多重適用しようとしても適用前にエラーで停止するようになります。これらの SQL は定型文なので、`id` と `title` からファイルを生成できるようにすると便利です。

この仕組みにより、一般的なマイグレーションツールと同等の履歴管理を、外部ツールに依存せずに SQL だけで実現できます。

スキーマ全体像の把握

ORM や DSL でマイグレーションを記述する場合、モデルクラスやスキーマ定義ファイルを見れば最新状態の構造をひと目で把握できます。一方で SQL では、`ALTER TABLE` 文のような変更を順に積み重ねていく形式になります。この状態では、最終的にどのようなテーブル構造になっているか把握しづらく、開発やレビューの負担が増してしまいます。

マイグレーション SQL をすべて順に適用した結果のダンプがあればよいのですが、これを都度作るのは手間がかかりるので自動化したいところです。

変更の可逆性の担保

マイグレーションにおいて、変更の可逆性は重要です。本番環境の更新作業の切り戻しの際や、開発中 Git のブランチを切り替えたときに DB を適切な状態にするためには、適用した変更を元に戻せる必要があります。

ORM や DSL を元に SQL を生成するようなマイグレーションツールでは、可逆性もある程度自動的に担保できるでしょう。しかし、SQL を直接書くような場合は、書く人間が責任を持って可逆性を維持しなければなりません。先に述べたように、ORM や DSL によるマイグレーションツールを利用していても、直接 SQL を書く必要性が出てきます。つまり、ほとんどすべてのマイグレーションツールが、この可逆性を完全には担保できていないのです。

もちろん SQL だけで SQL 自身の可逆性を確認することはできません。もし SQL の正当性・可逆性を確認することができるツールがあれば、SQL でマイグレーションを記述することはアドバンテージにさえなりえます。

このためのツールとして「**migy**」を開発しました。migy は自由な SQL で書かれた up/down の SQL ファイルの正当性と可逆性をチェックするサポートツールです。また、マイグレーション適用後の状態を SQL としてダンプする機能も備えています。これにより、可逆性を確認するとともにスキーマの最終状態を明確に把握することが可能になります。

それでは、migy について詳しく紹介したいと思います。

2.4 SQL マイグレーション支援ツール「migy」

- GitHub リポジトリ: `https://github.com/makiuchi-d/migy`

migy は、SQL によるマイグレーションを支援するスタンドアロンツールです。これまでに挙げた課題を解決するために開発されました。migy は内部に MySQL 互換のインメモリ DB エンジンを持っているため、外部 DB を用意しなくても SQL を実行することができます。

ここでは、基本的な使い方をなぞりながら、migy が SQL によるマイグレーションの課題をどのように解決するのかを説明します。

1. init: セットアップ SQL を生成

```
$ migy init
writing: 000000_init.all.sql
```

SQL によるマイグレーションの管理には、リスト 2.5 で紹介した履歴管理テーブルとリスト 2.7 の適用済みかどうかを確認するストアドプロシージャを用います。`init` サブコマンドは、これらを定義する SQL ファイル「`000000_init.all.sql`」を生成します。このファイルと同じディレクトリに、続く更新 SQL（up.sql と down.sql のペア）を配置していきます。

2. create: マイグレーション SQL の雛形を作成

```
$ migy create add_users_table
writing: 000010_add_users_table.up.sql
writing: 000010_add_users_table.down.sql
```

create サブコマンドは up/down のファイルペアを生成します。番号は自動的に採番され、ファイル先頭にはリスト 2.6 とリスト 2.8 で示したような、管理テーブルへの挿入や削除を行う SQL が書き込まれています。そのため、開発者はスキーマを更新する SQL の記述に集中できます。

それではリスト 2.9 のように更新 SQL として CREATE TABLE 文を記述してみましょう。

リスト 2.9: 000010_add_users_table.up.sql

```
-- Generated by migy (https://github.com/makiuchi-d/migy)
INSERT INTO _migrations (id, title, applied)
 VALUES (10, 'add_users_table', now());
-- Write your forward migration SQL statements below.
CREATE TABLE users (
  id    int NOT NULL PRIMARY KEY AUTO_INCREMENT,
  name  varchar(255) NOT NULL
);
```

down.sql を書く前に、このマイグレーションの可逆性を確認してみましょう。

3. check: 可逆性のチェック

```
$ migy check
applying: 000000_init.all.sql
---- up/down
applying: 000010_add_users_table.up.sql
applying: 000010_add_users_table.down.sql
checking...
unexpected "users" table found
check failed
```

check サブコマンドは、up/down を適用する前後で DB の状態を比較し、一致していない箇所を報告します。実際に SQL を実行するので、文法ミスなどもチェックできます。この例ではまだ down.sql を書いていないので、予期しない"users"テーブルが存在することが検出されました。

それではリスト 2.10 のように down.sql に DROP TABLE 文を書いて、もう一度 check してみます。

リスト 2.10: 000010_add_users_table.down.sql

```
-- Generated by migy (https://github.com/makiuchi-d/migy)
CALL _migration_exists(10);
DELETE FROM _migrations WHERE id = 10;
-- Write your rollback SQL statements below.
DROP TABLE users;
```

```
$ migy check
applying: 000000_init.all.sql
---- up/down
applying: 000010_add_users_table.up.sql
applying: 000010_add_users_table.down.sql
checking...
ok
```

今度はチェックを通過しました。

migy のチェックは、テーブルの有無やスキーマ構造だけでなく、テーブルに含まれるレコードの差分も検出します。ただし、単純にレコードを比較してしまうと、`up.sql` で削除したカラムを `down.sql` で復元した場合に元のデータが消えてしまっているため、すべてのレコードが差分ありとして検出されてしまいます。このような場合に備え、`down.sql` に `migy: ignore table.column` のようなコメントを書くことで、特定のカラムを無視して差分検出する機能もあります。詳しくは migy の `README.md` をご覧ください。

4. snapshot: スキーマ全体をダンプ

```
$ migy snapshot
applying: 000000_init.all.sql
applying: 000010_add_users_table.up.sql
========
writing: 000010_add_users_table.all.sql
```

`snapshot` サブコマンドは、マイグレーションを順に適用し、最終的な DB の状態を `all.sql` として出力します。今回の状態では、出力されたファイルはリスト 2.11 のようになってい

ます。_migrations テーブルに履歴レコードが挿入され、users テーブルが作られていることがわかります。

リスト 2.11: 000010_add_users_table.all.sql

```
-- Generated by migy (https://github.com/makiuchi-d/migy)

CREATE TABLE `_migrations` (
  `id` int NOT NULL,
  `applied` datetime,
  `title` varchar(255),
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_bin;

INSERT INTO `_migrations` (`id`,`applied`,`title`) VALUES
  (0, '2025-11-04 14:35:19', 'init'),
  (10, '2025-11-04 14:35:19', 'add_users_table');

CREATE TABLE `users` (
  `id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_bin;

DELIMITER //

CREATE PROCEDURE _migration_exists(IN input_id INTEGER)
BEGIN
  IF NOT EXISTS (SELECT 1 FROM _migrations WHERE id = input_id) THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'migration not found';
  END IF;
END//

DELIMITER ;
```

このようにマイグレーション適用済みの状態でのスナップショットができるので、ALTER TABLE 文を積み重ねていても最終的なテーブルの構造がこのファイルを見ればわかります。加えて、コンテナなどで開発環境を頻繁に新しく立ち上げる場合でも、最新のスナップショットからマイグレーションすることで素早く DB を準備できます。

5. apply/list: マイグレーションの適用とリストアップ

apply サブコマンドは、引数で指定された DB に migy が直接接続し、必要なマイグレー

ションを適用します。この方法は他の一般的なマイグレーションツールと同様です。

しかし、migy の主目的は「SQL によるマイグレーション」のサポートです。サーバー上に migy コマンドがなくても、SQL ファイルだけでマイグレーションできるのが強みです。ここで役に立つのが `list` サブコマンドです。

まずサーバー上で `mysqldump` コマンドを使って `_migrations` テーブルをダンプし、その結果を `dump.sql` とします。ここでは例として、`000000_init.all.sql` のみ適用した状態の `dump.sql` を用意しました[*5]。次のように `migy list` を実行すると、マイグレーションに必要な SQL ファイルが適用すべき順序で列挙されます。

```
$ migy list dump.sql
000010_add_users_table.up.sql
```

この SQL ファイルをサーバーにコピーしてリストの順に `mysql` コマンドで適用していけば、サーバー上に `migy` コマンドをインストールしなくても、マイグレーションを完遂できます。

ここで、`list` の実践的な使い方をひとつ紹介します。MySQL には、`.mylogin.cnf` にログイン情報を暗号化して保存することで、パスワードを明示せずに `mysql` コマンドで DB に接続できるようにする機能があります。筆者も実際に、本番 DB のパスワードを知らされていない状態[*6]で運用オペレーションを行っています。パスワードがわからないので、MySQL 標準コマンド以外のツールは DB に接続できず利用が難しいですが、migy なら `list` で必要な SQL ファイルを列挙し、それを順に mysql コマンドに流し込むだけでマイグレーションを実行できます。

例として、筆者が本番環境で行っているマイグレーションコマンド[*7]を次に示します。サーバー上に `migy` コマンドは用意されているので、パイプで `mysql` コマンドに SQL を流し込んでいます。`mysqldump` と `mysql` コマンドは`.mylogin.cnf` を読み取って DB に接続するので、パスワードを知る必要はありません。

```
$ mysqldump _migrations > dump.sql
$ migy list dump.sql | xargs cat | mysql
```

[*5] お手元で試したい場合、`000000_init.all.sql` をコピーして代用できます
[*6] 聞けば教えてもらえるはずですが、今のところ不要なので聞いていません
[*7] 省略しましたが、適用前にはリストを表示して内容を確認しています

2.5 技術の寿命から考える SQL マイグレーションの合理性

技術の寿命に目を向けると、SQL でマイグレーションを行う合理性が見えてきます。

サービスが終了したあともデータは活用され続けることからわかるとおり、データの寿命がとても長いことはよく知られています。では、スキーマの寿命はどうでしょうか。

まずは DB について考えてみましょう。これまでもオンプレからクラウド、最近では分散 DB への移行というムーブメントがみられます。このように、データの寿命よりは短いかもしれませんが、DB も数年から十数年単位という長いライフサイクルをもっていると言えます。

DB を移行する場合、主キーの選び方や制約の調整など、スキーマも適した構造に作り直してこそ、DB 本来の性能を発揮できるようになります。つまり、スキーマは DB と同じ寿命をもっているものなのです。

ではフレームワークやアプリケーションコードの寿命はどうでしょうか。セキュリティやパフォーマンス、スケーラビリティなどの要件の変化にあわせて、フレームワーク、あるいは実装言語そのものを入れ替えるという話は、技術系カンファレンスでよく語られるテーマです。この場合、多くのプロジェクトで既存の DB を維持したまま、アプリケーションを置き換えるという戦略がよく取られます。このことからも、DB よりもフレームワークやアプリコードのほうが寿命が短いとわかります。アプリが利用するライブラリやツールがさらに短命であることは言うまでもありません。

ここからわかるように、スキーマは DB とともに生きるものであり、アプリコードやフレームワークより寿命がずっと長いものなのです。そんなスキーマを管理するなら、寿命の短い技術に依存するのではなく、DB と同じく長く生きる形式、すなわち SQL を使うのが合理的です。SQL によるマイグレーションならば、アプリコードが寿命を迎えても、DB が生きている限り同じようにスキーマを管理し続けられるでしょう。

歴戦のエンジニアなら一度は、「この SQL を ORM でどう書けばいいのか」と悩んだり、「この条件なら SQL で書いたほうが早い」と感じたことがあるでしょう。複数のプロジェクトを渡り歩いてさまざまな言語やフレームワークを使っても、根底にはいつも同じように SQL がいます。SQL は時代が変わっても陳腐化しにくい技術なのです。

どんな言語やフレームワークを使っていても SQL を理解していれば、データ構造を読み解き、性能を診断し、正しいスキーマを設計できます。マイグレーションを SQL で書くということは、単に DB を操作するというだけでなく、時代を超えて生き続ける知識と道具を手に入れることでもあります。

2.6 おわりに

マイグレーションは、単なるスキーマ変更の履歴ではなく、アプリケーションと DB の関係の歴史そのものです。だからこそ、変化の早いアプリケーション側の技術よりも、DB と同等に長く安定して使える SQL という共通言語で記述する意義があります。

SQL によるマイグレーションは、一見すると原始的で手間のかかる方法に見えるかもしれません。しかし、スキーマの寿命が DB のそれと同じであることを考えれば、その管理に DB と密接に関連する言語である SQL を使うのは合理的な選択です。スキーマの変更履歴を、アプリケーションのコードやフレームワークに縛られず、純粋に DB の視点で残せるという点でも価値があります。

SQL でマイグレーションを書くことは、変化の制御をアプリケーションから切り離し、データの永続性に責任を持つという選択です。そのための一助として、SQL の力を素直に活かせる道具を手に取ってもらえれば幸いです。

第 3 章

TypeScript + GitHub Actions + Grafana Cloud で実現する現代的 GAS 開発環境

Yoshio HANAWA / @hnw

3.1 はじめに

Google Apps Script (GAS) は、Google アカウントさえあればブラウザ上で即座にコーディングを開始でき、`SpreadsheetApp` や `GmailApp` といった API を通じて Google サービスを自在に操れる、手軽で強力な実行環境です。

一方で、標準的な GAS の開発環境は、前時代的とも言える課題を抱えています。

- **Web エディタが非効率:** Web エディタは手軽ですが、ローカル開発環境では当たり前に使えるコード補完やバージョン管理の恩恵を受けられないうえ、import/export によるモジュール管理もできません。
- **品質担保の仕組みがない:** 単体テストや型安全性の保証など、現代的な品質担保の仕組みが標準で備わっていません。
- **本番環境しかない:** Web エディタで「保存」ボタンを押した瞬間、本番環境にコードが反映されます。ステージング環境での事前確認ができないのは事故と隣り合わせです。
- **オブザーバビリティの不足:** デバッグ手段は貧弱で、`Logger.log()` をコードに埋め込むやり方がほぼ唯一の選択肢です。実行状況の可視化や、エラー発生時のアラートなどの仕組みも不十分です。

これらの課題は、スクリプトが小規模なうちは問題になりません。しかし、スクリプトが複雑化し、ビジネス上重要な役割を担うようになると、その開発体験の悪さが保守性の低さに直結します。

本稿では、GAS の開発環境を TypeScript、webpack、Jest、GitHub Actions、Grafana Loki といった現代的な技術スタックで再構築し、堅牢で保守性の高い環境に変えるための具体的なステップを紹介します。

3.2 開発環境のローカル化

GAS 環境改善の最初のステップは、Web エディタから脱却し、ローカルのエディタ（VS Code など）による開発環境を整えることです。 これを実現するのが、Node.js を基盤とする現代的な Web 開発のツール群です。

GAS の実行環境は、Node.js とは大きく異なる、制限の強い JavaScript 環境です。しかし、npm によるライブラリ管理や、webpack によるコードのバンドル（分割管理）といった、Node.js が Web 開発で培ってきたノウハウは GAS でも活用できます。

npm (Node Package Manager)

npm は Node.js のライブラリ（パッケージ）を管理するツールです。GAS は標準機能だけでも強力ですが、npm を使えば実績のあるライブラリを GAS でも使えるようになります。

また、npm には開発プロセスを支援する機能もあります。 `npm run deploy` のような定型コマンドを登録することで、ビルドやデプロイといった作業を自動化・簡略化することができます。

注意点ですが、全ての Node.js ライブラリが GAS 環境で利用できるわけではありません。トラブルの事例をコラム「Node.js 向けライブラリを GAS 環境で使うメリットとデメリット」で紹介しています。

webpack

GAS は、Node.js のような import 構文を使ったファイルの分割管理に対応していません。そこで「モジュールバンドラ」の webpack を使います。

webpack は、`src/` ディレクトリ内の複数の TypeScript/JavaScript ソースコードを、GAS が実行可能な単一の JavaScript ファイル（`dist/Code.js`）に束ねる役割を果たします。

webpack を導入することで、GAS 環境でも Node.js のモジュールを import したり、ES モジュールの形でファイル分割したりすることができます。

clasp

clasp は、Google が提供する GAS 管理用のコマンドラインツールです。webpack が生成した `dist/Code.js` と設定ファイル `appsscript.json` を、実際の GAS プロジェクトにデプロイする役割を担います。

設定例

これらのツールを利用する設定例をリスト 3.1、リスト 3.2、リスト 3.3 に示します。

リスト 3.1: package.json

```
{
  "name": "example-modern-gas-project",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "build": "webpack",
    "deploy": "npm run build && clasp push"
  },
  "devDependencies": {
    // （開発用パッケージの一覧）
  }
}
```

リスト 3.2: webpack.config.js

```
const path = require("path");
const GasPlugin = require("gas-webpack-plugin");
const CopyPlugin = require("copy-webpack-plugin");

module.exports = {
  mode: "development",
  entry: "./src/main.js", // エントリーポイント（TypeScript対応後は要修正）
  resolve: {
    extensions: [".ts", ".js"],
  },
  output: {
    filename: "Code.js",
    path: path.resolve(__dirname, "dist"),
    clean: true,
  },
```

```
  plugins: [
    new GasPlugin(),
    new CopyPlugin({
      patterns: [
        { from: "src/appsscript.json", to: "." },
      ],
    }),
  ],
};
```

リスト 3.3: .clasp.json

```
{
  "scriptId": "<スプレッドシートのスクリプトID>",
  "rootDir": "./dist"
}
```

この設定により、`npm run deploy` コマンドで `src/` 以下の TypeScript コードを `dist/Code.js` にバンドルし、clasp で GAS 環境にデプロイできるようになります。

■コラム: google/aside のススメ

本稿で紹介したローカル開発環境は、Google が公式に提供する `google/aside`[*1] というボイラープレートと、その目的や技術スタックが非常に似ています。

実を言うと、筆者はこの記事で紹介した環境の多くを構築し終えるまで、`google/aside` の存在を知りませんでした。筆者は本稿で紹介した環境を一から構築しましたが、読者の皆さんは `google/aside` をベースに拡張することをお勧めします。

`google/aside` はローカル開発環境や TypeScript、Jest の基盤を提供してくれます。本稿の以降の章では、`google/aside` に含まれていない、以下の内容についても詳しく解説します。

- Zod, Husky の利用
- GitHub Actions を使った CI/CD
- Grafana Cloud を使ったオブザーバビリティ

3.3 コード品質の担保

GAS にローカル開発環境を導入するだけでも大きなメリットがありますが、さらに品質を高めるための必須ツールを紹介します。

TypeScript

GAS 環境では JavaScript しか動作しませんが、clasp と webpack を導入したことで、TypeScript での開発が可能になります。静的な型チェックは、GAS のグローバル API（例: `SpreadsheetApp`）の型補完を効かせたり、開発段階で多くの型関連のバグを防いだりする上で不可欠です。

Zod: 外部から取得する値の型定義と型バリデーション

TypeScript の静的型チェックはコンパイル時には強力ですが、実行時に外部から受け取るデータの型まで保証するものではありません。例えば、GAS のスクリプトプロパティは、すべての値が `string` 型（または `null`）として返されます。

これをプログラムで安全に扱うためには、`string` を `number` や `boolean` に変換したり、JSON 文字列をパース（`JSON.parse`）したりする処理や、その結果が期待通りであるかを検証する処理が必要になります。こうした文字列からの型変換とバリデーションの処理は冗長で保守性の低いコードになりがちです。

この問題を解決するのが、TypeScript で広く使われている Zod パッケージです。Zod はオブジェクトをスキーマ定義として宣言的に記述することで、TypeScript の型推論と実行時検証の両方を実現します。

リスト 3.4 は GAS のスクリプトプロパティから `AppConfig` オブジェクトへ型変換する例です。

リスト 3.4: Zod によるスクリプトプロパティの型チェック

```
import { z } from "zod";

// スキーマ定義
const AppConfigSchema = z.object({
  SPREADSHEET_ID: z.string().min(1, "SPREADSHEET_ID は必須です"),
```

[*1] https://github.com/google/aside

```
  // 正の整数に変換する。デフォルトは1000。
  QUERY_LIMIT: z.coerce.number().int().positive().default(1000),
  // 規定のログレベルのいずれかが指定されていればそれを採用する。デフォルトはINFO。
  LOG_LEVEL: z.enum(["DEBUG", "INFO", "WARN", "ERROR"]).default("INFO"),
});

// スキーマから TypeScript の型を自動生成
export type AppConfig = z.infer<typeof AppConfigSchema>;

// GASのスクリプトプロパティを検証し、成功したらAppConfig型として返す
export function getConfig(): AppConfig | null {
  const rawConfig: Record<string, string> =
    PropertiesService.getScriptProperties().getProperties();

  const result = AppConfigSchema.safeParse(rawConfig);

  if (!result.success) {
    // 検証失敗（SPREADSHEET_ID が無い、など）
    Logger.log("スクリプトプロパティの読み込みに失敗しました。");
    Logger.log(result.error.message);
    return null;
  }
  return result.data;
}
```

このように、Zod を使うことで宣言的にバリデーションルールを定義し、シンプルかつ安全にデフォルト値や型変換を適用できます。

■コラム: Node.js 向けライブラリを GAS 環境で使うメリットとデメリット

webpack を採用することで GAS でも Node.js ライブラリを import できると紹介しましたが、注意点もあります。GAS 環境はかなり特殊な環境なので、期待通り動かないことも多いのです。

例えば、今回紹介した Zod で言うと、文字列が正しい URL かを検証する `z.string().url()` が期待通りに動きません。内部的に利用している `new URL()` が GAS の V8 エンジン上では提供されないためです。代わりに `z.string().regex(/^https:\/\/.+/)` などと記述する必要があります。

こうした環境依存の問題はトラブルの元ですが、それでも実績のある高機能なライブラリを利用できるメリットは計り知れません。GAS 開発でもできるだけ巨人の肩に乗っていきましょう。

Jest: ローカルでの高速な単体テスト

標準的な GAS の開発環境で単体テストを動かすのは非現実的ですが、ローカル環境であれば Jest で簡単に実施できます。Jest は Node.js で一般的に使われているテストフレームワークです。

GAS のコードは `SpreadsheetApp` などのグローバル API に依存しているため、そのままではローカルでテストできません。しかし、Jest の機能を使ってこれらのグローバル API をモックに差し替えることで、ローカルでの単体テストが可能になります。

リスト 3.5 に GAS API のうち `PropertiesService` のモックを定義するコード例を示します。

リスト 3.5: tests/setup.ts

```
import Properties = GoogleAppsScript.Properties.Properties;
import PropertiesService = GoogleAppsScript.Properties.PropertiesService;

// PropertiesServiceのモック
let props: Record<string, string> = {};
const mockProperties: Partial<Properties> = {
  getProperty: (key: string) => props[key],
  getProperties: () => ({ ...props }),
  setProperty: (key: string, value: string) => {
    props[key] = value.toString();
    return mockProperties as Properties;
  },
  deleteAllProperties: () => {
    props = {};
    return mockProperties as Properties;
  },
};

globalThis.PropertiesService = {
  getScriptProperties: jest.fn(() => mockProperties as Properties),
} as unknown as jest.Mocked<PropertiesService>;

export {};
```

Jest の設定をリスト 3.6 のようにすることで、GAS API のモックを全てのテストの前に読み込むことができます。

リスト 3.6: jest.config.cjs

```
/** @type {import('jest').Config} */
const config = {
  // TypeScriptをJestで実行するための設定 (ts-jest)
  transform: {
    // .ts ファイルを ts-jest でトランスフォーム
    "^.+\\.ts$": "ts-jest",
  },

  // GAS APIのグローバルモックを読み込む
  setupFilesAfterEnv: ["<rootDir>/tests/setup.ts"],

  // テストの独立性を担保
  clearMocks: true,
};

module.exports = config;
```

このモックを前提としたテストの例がリスト 3.7 です。利用する全ての API のモックを整備することで、ローカルで単体テストが動かせるようになり、デプロイ前にコードの品質を担保できます。

リスト 3.7: tests/config.test.ts

```
import { getConfig } from "../src/config";

describe("getConfig", () => {
  beforeEach(() => {
    PropertiesService.getScriptProperties().deleteAllProperties();
  });

  test("should return a valid config object when all properties are set", () => →
{
    const scriptProperties = PropertiesService.getScriptProperties();
    scriptProperties.setProperty("SPREADSHEET_ID", "test-sheet-id");

    const config = getConfig();
    expect(config).not.toBeNull();
    if (config !== null) {
      expect(config.SPREADSHEET_ID).toBe("test-sheet-id");
      expect(config.QUERY_LIMIT).toBe(1000);
      expect(config.LOG_LEVEL).toBe("INFO");
    }
  });
});
```

Husky: Git フックによる規約の強制

Husky は、git のコミットやプッシュといった特定のアクションの直前に、あらかじめ定義したスクリプト（フック）を自動実行するツールです。

これにより、品質の低いコードやフォーマットが崩れたコードがリポジトリにコミットされるのを防ぎます。筆者は pre-commit フックを利用し、lint-staged を連携させました。

```
$ echo "npx lint-staged" > .husky/pre-commit && chmod +x .husky/pre-commit
```

lint-staged は、ステージングされたファイル（`git add` されたファイル）に対してのみ、ツールを実行する仕組みです。リスト 3.8 の設定により、ESLint（構文チェックと自動修正）と Prettier（コードフォーマッター）を自動実行しています。

リスト 3.8: package.json の lint-staged の設定

```
{
   （省略）
  "lint-staged": {
    "*.{js,ts}": [
      "eslint --fix",
      "prettier --write"
    ]
  }
}
```

3.4　環境分離とデプロイ自動化

ローカルでの開発・テスト環境は整いましたが、まだ「本番環境しかない」という GAS 最大のリスクが残っています。

この問題を解決するため、GAS プロジェクト自体を「ステージング環境用」と「本番環境用」の 2 つに分離し、GitHub Actions による CI/CD パイプラインを構築します。

- `develop` ブランチへの Push → ステージング環境へ自動デプロイ
- `main` ブランチへの Push → 本番環境へ自動デプロイ

clasp の認証情報と、各環境の scriptId は、リポジトリの GitHub Secrets に登録しておきます。

- CLASPRC_JSON: ~/.clasprc.json ファイルの中身全体
- STAGING_SCRIPT_ID: ステージング用 GAS プロジェクトの ID
- PRODUCTION_SCRIPT_ID: 本番用 GAS プロジェクトの ID

リスト 3.9 に、これらの Secrets を使って認証情報とデプロイ先を切り替える GitHub Actions の YAML を示します。

リスト 3.9: .github/workflows/ci-cd.yml

```
name: CI/CD Pipeline
on:
  push:
    branches: [main, develop] # main または develop ブランチへの push で発火

jobs:
  deploy-staging:
    name: Deploy to Staging
    runs-on: ubuntu-latest
    # develop ブランチへの push 時にのみ実行
    if: github.ref == 'refs/heads/develop' && github.event_name == 'push'
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: "20"
          cache: "npm"
      # 1. Secretから認証ファイル（~/.clasprc.json）を復元
      - name: Setup clasp authentication
        run: echo "$CLASPRC_JSON" > ~/.clasprc.json
        env:
          CLASPRC_JSON: ${{ secrets.CLASPRC_JSON }}
      - name: Build
        run: npm run build
      # 2. ステージング用の .clasp.json を動的に生成
      - name: Generate .clasp.json (Staging)
        run: |
          echo '{"scriptId":"${{ secrets.STAGING_SCRIPT_ID }}", "rootDir": "./di→
st"}' > .clasp.json
      # 3. clasp でデプロイ実行
      - name: Deploy to Staging
        run: npx clasp push

  # (deploy-production: deploy-stagingと同様)
```

GAS はローカルで全機能をテストできないため、デプロイ後の初回実行が E2E テストの側面を持ちます。本番環境の利用者に迷惑をかけないためにも、複数環境を構築する意義は非常に大きいと言えるでしょう。

3.5 オブザーバビリティの改善

最後の課題は「オブザーバビリティ（可観測性）」です。GAS 標準の `Logger.log` は実行ログの確認や検索が難しく、エラー発生時のアラートも貧弱です。

そこで、筆者のプロジェクトでは、スクリプト自身の実行ログ（処理開始、成功、エラー、実行時間など）を、オブザーバビリティ基盤サービスの Grafana Cloud に送信する「自己監視」の仕組みを導入しました。

Grafana Cloud は無料プランで月 50GB のログを管理できるなど、無料枠が大きいのが特徴です。

実行ログを Loki に一括送信する

Loki にログを送信する際、`UrlFetchApp.fetch()` で Loki の Push API を呼び出す必要があります。しかし、`UrlFetchApp` の呼び出しには実行回数上限があるため、`logger.info()` のたびに API を呼び出すと、すぐに上限に達してしまいます。

この問題を解決するため、筆者のプロジェクトではログを内部バッファに一時的に蓄積し、スクリプトの実行終了時に `flushLogs()` 関数で一括送信するロガーを実装しました。

リスト 3.10: logger 実装（概要）

```
let logBuffer: LogBufferEntry[] = []; // ログバッファ

export const logger = {
  info: (messageObject: Record<string, unknown>) => {
    // ログを logfmt 形式の文字列に変換
    const logfmtLine = toLogfmt({ level: "info", ...messageObject });
    logBuffer.push({ timestamp: Date.now(), logfmt: logfmtLine });
  }
  // (debug, warn, error も同様...)
};

export function flushLogs() {
  const logsToSend = [...logBuffer];
  logBuffer = [];
```

```
  const lokiPayload = {
    streams: [
      {
        stream: { job: "gas", env: "test" }, // ログのラベル
        values: logsToSend.map(entry => [
          String(entry.timestamp * 1_000_000),
          entry.logfmt,
        ]),
      },
    ],
  };

  UrlFetchApp.fetch("https://loki.example.com/loki/api/v1/push", {
    method: "post",
    contentType: "application/json",
    payload: JSON.stringify(lokiPayload),
    muteHttpExceptions: true,
  });
}
```

このロガーの利用例をリスト 3.11 に示します。メインの処理を `try...finally` で囲み、処理が成功しても失敗しても、最後に必ず `flushLogs()` が呼ばれるようにしました。

リスト 3.11: main.ts（logger の flush 処理の抜粋）

```
import { logger, flushLogs } from "./logger";

export function main() {
  try {
    logger.info({ event: "process_start" });
    // メイン処理を実行...
    logger.info({ event: "data_received", value: 123 });
  } finally {
    flushLogs();
  }
}
```

これにより、`UrlFetchApp` の呼び出し回数を 1 実行あたり 1 回に抑えつつ、詳細な実行ログを Loki に集約できます。

measure 関数の導入と Grafana での可視化

ログが Loki に集約されると、Grafana で強力な可視化が可能になります。これを活用するため、筆者のプロジェクトでは measure という汎用的なパフォーマンス計測関数を導入しました。これは、関数ごとに累計の実行時間と呼び出し回数を集計し、Loki に送信するものです。実装の概要をリスト 3.12 に示します。

リスト 3.12: measure.ts（抜粋）

```
export interface PerformanceMetricsEntry {
  total_duration_ms: number;
  call_count: number;
}

export type PerformanceMetricsStore = Record<string, PerformanceMetricsEntry>;

let __performanceMetrics: PerformanceMetricsStore = Object.create(null);

export function measure<T>(functionName: string, fn: () => T): T {
  const startTime = Date.now();
  try {
    return fn();
  } catch (err) {
    throw err;
  } finally {
    const duration = Date.now() - startTime;
    const entry = __performanceMetrics[functionName] ??= {
      total_duration_ms: 0,
      call_count: 0,
    };
    // 関数ごとに実行時間と呼び出し回数を加算
    entry.total_duration_ms += duration;
    entry.call_count += 1;
  }
}
```

この measure 関数は、例えばリスト 3.13 のように使うことができます。

リスト 3.13: `measure` の呼び出し例

```
const ss = measure("SpreadsheetApp.openById", () =>
  SpreadsheetApp.openById(config.SPREADSHEET_ID);
)
```

`measure` で測定した実行時間を可視化したものが、図 3.1 の上側のグラフです[*2]。1 回の GAS 実行ごとに集計されており、関数ごとの積み上げグラフになっています。いつも遅い関数や、普段は速いのに時々遅くなる関数などが一目でわかるので、性能改善に役立ちました。

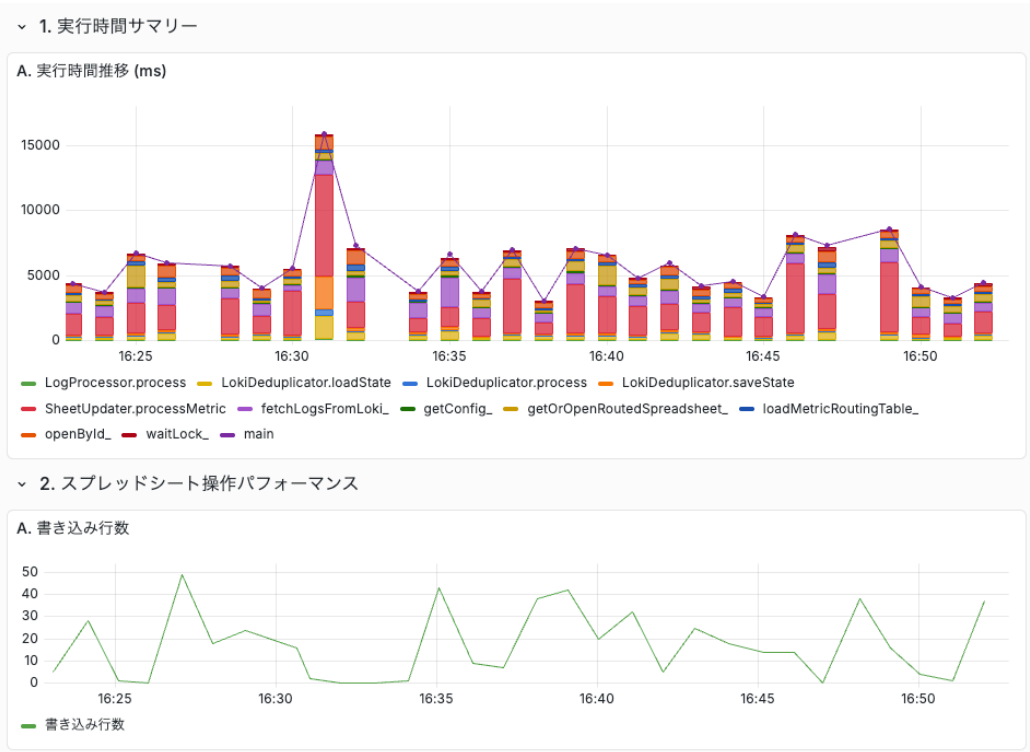

図 3.1: Grafana ダッシュボードの例

[*2] 可視化の対象は `measure` で測定した値だけではありません。図 3.1 の下側のグラフのように、任意のメトリクスを GAS から Loki に送って可視化することができます。

3.6　現代的な環境だからこそ実現できた改善の具体例

この開発環境を整えたことで、従来の GAS 開発では困難だった、より高度な改善も可能になりました。筆者が実際に業務で取り組んだ内容を簡単に紹介します。

実行時エラーに素早く対応

GAS では、様々な理由で実行時エラーが発生します。一時的なエラーであれば無視してもよいのですが、問題の原因を取り除かないと解決できないエラーの場合は素早い検知と対応が求められます。しかし、GAS の標準のエラー通知はメールのみであり、見逃してしまうのが悩みでした。

今回、Grafana の設定で `ERROR` レベルのエラーを Slack に通知するようにしました。図 3.2 は実際に発生したエラーが Slack に転送されている様子です。

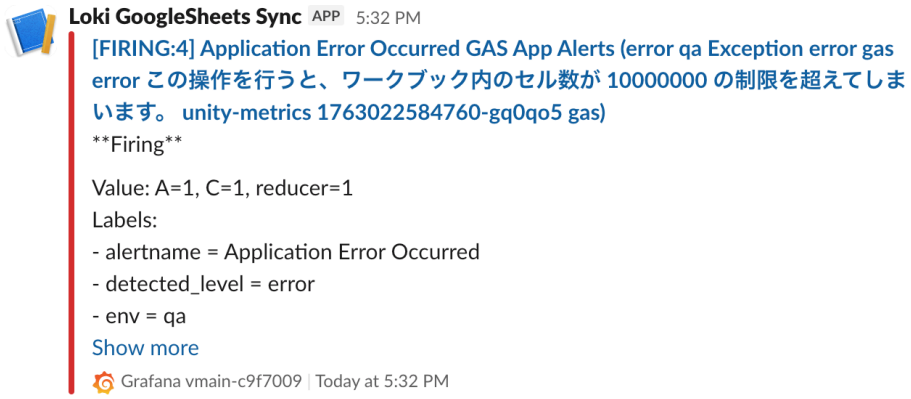

図 3.2: GAS のエラーが Slack に通知される様子

筆者が実装していた GAS はスプレッドシートに書き込むものだったのですが、スプレッドシートのセル数の上限に到達してエラーが出ていました。「上限ってあるんだ…」というのが筆者の正直な感想でした。[*3]

[*3] 実は事前に調べて上限値があることは知っていたのですが、性能劣化して動作保証ができなくなる目安だと想像しており、エラーで停止するのは予想外でした

ハッシュ計算の高速化

筆者が実装していた GAS では、ハッシュ値を使った重複検出のロジックがあり、ハッシュ関数として GAS の `Utilities.computeDigest` を使用していました。GAS 公式の API ですから、これが遅いと疑う人は少ないでしょう。

筆者もこの関数が遅いとは考えていませんでしたが、 `measure` 関数で測定したところ、まれに極端に遅くなる（数十 ms）ことがわかりました。推測ですが、この関数は内部的にネットワーク通信を行っており、リトライなどの理由で遅いことがあるのでしょう。

そこで、このハッシュ計算をピュア JavaScript ライブラリである crypto-js に置き換えました。ハッシュ計算や暗号化のような処理はピュア JavaScript に向かないため、通常であれば最適化された API の方が速いはずです。しかし、今回 `measure` 関数で集計したところ、crypto-js の方が GAS の API より速いことがわかりました。

npm と webpack の採用により、既存ライブラリを簡単に試せる環境になったことが活きた例と言えるでしょう。

`Proxy` によるスプレッドシート API の透過的キャッシュ

筆者が実装していた GAS ではスプレッドシートの読み書きが大量に発生するため、全体の実行時間の遅さに悩まされていました。そこですべてのスプレッドシート API 呼び出しを `measure` 関数で計測してみました。

実行時間を可視化してみると、シンプルに思える API でも最悪時は非常に遅いことがわかりました。たとえば、シートの行数を取得する関数 `sheet.getLastRow()` は軽量だろうと想像して繰り返し呼び出していたのですが、最悪時は数秒かかっていました。

そこで、ES2015 (ES6) で導入された `Proxy` を使って、GAS の `Spreadsheet` オブジェクトや `Sheet` オブジェクトを透過的にラップするキャッシュ層を実装しました。これを使うと、たとえば `sheet.getLastRow()` を呼び出すと 2 回目以降はキャッシュした結果を返すようになります。

リスト 3.14 に実装の抜粋を示します。

リスト 3.14: Proxy による透過的メモ化

```
export type Memoized<T> = T & { readonly raw: T };

export function createMemoizedSpreadsheet<T extends Spreadsheet>(
  spreadsheet: T,
): Memoized<T> {
  const sheetCache = new Map<string, Memoized<Sheet>>();
  const handler: ProxyHandler<Spreadsheet> = {
    get(target, prop, receiver) {
      const original = Reflect.get(target, prop, receiver);
      if (prop === "getSheetByName") {
        // getSheetByNameをキャッシュ化する
        return function memoizedGetSheetByName(this: unknown, name: string) {
          // 同じ引数で過去に呼び出されていたらキャッシュを返し、本来の実装を呼び出さない
          if (sheetCache.has(name)) return sheetCache.get(name)!;
          // 本来の実装を呼び出して得られたシートをキャッシュ層でラップして返す
          const sheet = original.apply(target, [name]);
          const wrapped = createMemoizedSheet(sheet);
          sheetCache.set(name, wrapped);
          return wrapped
        }
      } else {
        // 他の関数もラップしてキャッシュ実装に差し替える
      }
    }
  }
  const proxy = new Proxy(spreadsheet, handler) as Memoized<T>;
  return proxy;
}
```

Proxy は、元になるオブジェクトの関数やプロパティを上書きするような仕組みで、概念としては Python のデコレーターに近いものです。これを使って、Spreadsheet、Sheet、Range など関連するオブジェクトを全てラップすることでキャッシュ機構が実現できます。

この使い方ですが、リスト 3.15 のように Spreadsheet オブジェクトを createMemoizedSpreadsheet でラップするだけで、あとはオリジナルと同じインターフェースでメソッドを呼び出せます。キャッシュなしの前提で書いたコードのまま実行性能を改善できるのが利点です。

リスト 3.15: createMemoizedSpreadsheet 利用コード

```
    const rawSs = SpreadsheetApp.openById(config.SPREADSHEET_ID);
    const spreadsheet = createMemoizedSpreadsheet(rawSs);
```

リスト 3.14 では読み取り系 API の結果をキャッシュするところだけ紹介しましたが、実際は書き込み系 API の呼び出し時にキャッシュを無効化しています。

3.7 おわりに

本稿では、Google Apps Script の開発環境をモダン化するノウハウを紹介しました。GAS で大規模開発をしたり長期メンテナンスしたりする際に活用してみてください。

特に、「ステージング環境を用意する」「オブザーバビリティを改善する」の 2 点は他であまり紹介されていない内容だと思いますが、どちらも非常に強力であり、多くの人に体験してもらいたい手法です。

本稿の内容が GAS 開発で苦労している方の助けになれば幸いです。

付録 A

執筆者・スタッフコメント

第 1 章 Shunsuke Ito / @fgshun

コーヒー淹れて。のんびり、まったりと。

第 2 章 Daisuke Makiuchi / @makki_d

眼鏡っ娘が好きです

第 3 章 Yoshio HANAWA / @hnw

老眼が進んできて、原稿を A5 で印刷すると読めないことに気づきました。本誌は B5 なので安心です！

企画進行・イラスト・デザイン

umezawa-to

企画進行を担当しました。社内のみなさんの協力により発行し続けられており、感謝。

okazaki-e

表紙担当しました。好き放題色々こねくりまわして楽しく制作しました！

既刊・電子版ダウンロード

https://www.klab.com/jp/blog/tech/2025/tbf19.html

KLab Tech Book Vol. 16

2025 年 11 月 15 日　技術書典 19 版（1.0）
著　者　KLab 技術書サークル
編　集　牧内 大輔、梅澤 寿史
発行所　KLab 技術書サークル
印刷所　日光企画

KLab Tech Book
vol.16