KLab Tech Book Vol. 17



1. Python の t-string の紹介
2. cgo の魔法 〜 ビルド編
3. 社内ゲームジャムに GitHub Issue だけで参加する
4. Copilot 主導で進める Unity ゲーム開発
5. Docker Compose + Ansible + Grafana Cloud による快適自宅サーバ管理
6. ルータ沼にはまってきた —— J4125 と OpenWrt で挑むトリプルスタック構築記

KLab Tech Book Vol. 17

KLab 技術書サークル　著

2026-04-11 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 17 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 6 つの記事を収録しました。プログラミング言語、AI コーディング、自宅インフラ環境と、幅広い技術領域の話題となっております。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。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

はじめに **2**

お問い合わせ先 2

免責事項 2

第 1 章 Python の t-string の紹介 **5**

1.1 はじめに 5

1.2 t-string とは 5

1.3 テンプレートオブジェクトの操作 7

1.4 使用例 10

1.5 おわりに 14

第 2 章 cgo の魔法 〜 ビルド編 **15**

2.1 cgo とは 15

2.2 cgo ビルドの全体像 16

2.3 cgo ツールが生成するコード 17

2.4 型情報の収集 20

2.5 まとめ 23

第 3 章 社内ゲームジャムに GitHub Issue だけで参加する **24**

3.1 はじめに 24

3.2 つくったもの 24

3.3 GitHub Copilot Coding Agent について 26

3.4 やったこと 27

3.5 エピソード 36

3.6 まとめ 38

第 4 章 Copilot 主導で進める Unity ゲーム開発 **39**

4.1 はじめに . . . 39
4.2 実装機能の全体像 . . . 40
4.3 タイトルとインゲームの 2 シーン構成 . . . 40
4.4 CPU 機能 . . . 41
4.5 オンラインマルチ機能 . . . 43
4.6 シーン・プレハブを生成するエディタ拡張 . . . 44
4.7 特定のゲーム状況を再現するセットアップツールの実装 . . . 45
4.8 完全イベント駆動のゲーム設計 . . . 46
4.9 複数のゲームモード構成 . . . 47
4.10 スクリプタブルオブジェクトを使ったゲームデータ設計 . . . 48
4.11 Unity のログを自動で読み取る仕組み . . . 49
4.12 Copilot 主導の開発で感じたこと . . . 50
4.13 おわりに . . . 51

第 5 章 Docker Compose + Ansible + Grafana Cloud による快適自宅サーバ管理 52
5.1 はじめに . . . 52
5.2 Docker Compose：設定とデータを一元化するコンテナ管理基盤 . . . 53
5.3 Ansible：大規模環境でなくても便利に使える IaC ツール . . . 55
5.4 Grafana Cloud：自宅サーバ・自作アプリでこそ活きる可視化基盤 . . . 60
5.5 おわりに . . . 63

第 6 章 ルータ沼にはまってきた —— J4125 と OpenWrt で挑むトリプルスタック構築記 64
6.1 発端：1000 円の検証機から始まった「ルータ沼」 . . . 64
6.2 敗北：Archer C6 が教えてくれた「CPU の壁」 . . . 65
6.3 転換：J4125 への移行と「OpenWrt ビルド」の洗礼 . . . 67
6.4 実装：DS-Lite × PPPoE × IPv6 リレーの「全部盛り」 . . . 71
6.5 凱旋：圧倒的な数値と、残された課題 . . . 78
6.6 結論：ルータの速度は「無線」ではなく「CPU」で決まる . . . 79
6.7 付記：フルビルド不要だったという話 . . . 80

付録 A 執筆者・スタッフコメント 81

第 1 章

Python の t-string の紹介

Shunsuke Ito / @fgshun

1.1 はじめに

Python 3.14 からテンプレート文字列リテラル (Template String Literals, 以下 t-string と略記)[*1] が追加されました。どのような機能なのか、そして似た既存機能である f-string にはできなかった「遅延評価」「文字列構築時に別の処理を割り込ませる」ことにより新たに可能になったこと、たとえば HTML テンプレートエンジンを f-string 風の文字列操作で実現しつつ XSS 対策を行う様子などを紹介します。

1.2 t-string とは

t-string とはテンプレートオブジェクト[*2]を作るためのリテラル表記です。f-string に似たような見た目をしていますが、できあがるものが文字列ではないという違いがあります。

[*1] https://docs.python.org/ja/3.14/library/stdtypes.html#template-string-literals-t-strings

[*2] https://docs.python.org/ja/3.14/library/string.templatelib.html#string.templatelib.Template

1.2 t-string とは

リスト 1.1: f-string と t-string の例

```
book = "KLab Tech Book Vol. 17"
year = 2026
f = f'{book} 発行 {year} 年'  # 'KLab Tech Book Vol. 17 発行 2026 年'
t = t'{book} 発行 {year} 年'
# Template(
#  strings=('', ' 発行 ', ' 年'),
#  interpolations=(
#   Interpolation('KLab Tech Book Vol. 17', 'book', None, ''),
#   Interpolation(2026, 'year', None, '')))
```

t-string は f-string 同様の中括弧に囲まれた「置換フィールド」を持ちます。テンプレートオブジェクトの `strings` 属性には文字列のタプルが、 `interpolations` 属性には置換フィールド情報 `Interpolation` オブジェクトのタプルが収まっています。`strings` と `interpolations` の内容を交互に連結すると t-string に現れた順となるようになっています。

`Interpolation` オブジェクトは次の属性を持ちます。置換フィールドに対応した値 `value` 、その元となった式 `expression` 、型変換の指定 `conversion` 、そして書式指定 `format_spec` です。

リスト 1.2: Interpolation オブジェクト

```
i = t.interpolations[0]
i.value        # 'KLab Tech Book Vol. 17'
i.expression   # 'book'
i.conversion   # None
i.format_spec  # ''

f'{1+2!r:>5s}'.interpolations[0]  # '    3'
j = t'{1+2!r:>5s}'.interpolations[0]
j.value        # 3
j.expression   # '1+2'
j.conversion   # 'r'
j.format_spec  # '>5s'
```

つまり、できあがったテンプレートオブジェクトには t-string リテラルのテキスト情報や置換フィールドに対応するオブジェクトの情報が余すことなく保持されているのです。t-string がどのようにテンプレートオブジェクトを作成するのか、そこに含まれる置換フィールド情報 `Interpolation` とはどのようなものか、については以上となります。

1.3 テンプレートオブジェクトの操作

テンプレートオブジェクトを作っただけでは何かを成したわけではありません。ここからはテンプレートオブジェクトをどのように使うのかについて説明します。

情報の取り出し方

テンプレートオブジェクトから含まれる情報を取り出す標準的な方法は、イテレートすることです。`strings` 属性の内容と `interpolations` 属性の内容が現れた順に取り出されます[*3]。

リスト 1.3: t-string を for 文で処理する

```
for i in t:
  print(repr(i))
# Interpolation('KLab Tech Book Vol. 17', 'book', None, '')
# ' 発行 '
# Interpolation(2026, 'year', None, '')
# ' 年'
```

文字列と `Interpolation` オブジェクトをとり分けるには match 文を用いると便利です。クラス `Interpolation` には match 文用設定である `__match_args__` クラス変数が `('value', 'expression', 'conversion', 'format_spec')` と設定されているので match 文の位置引数パターンを次のように使用することができます。

リスト 1.4: str と Interpolation を match 文で分ける

```
for i in t:
  match i:
    case str() as s:
      ...
    case Interpolation(value, _, _, format_spec):
      # value と format_spec に当該属性を代入済みの状態にて処理を記述できる
      ...
```

[*3] strings 属性には先頭・末尾・置換フィールドが隙間なく置かれていた場合に空文字列が含まれることがありますが、イテレートした時の出力からは省略されます。この例では先頭の空文字列が省略されています。

f-string の再実装

取り出した情報の使い方の一例を f-string の再実装と機能追加を通じて説明します。t-string は f-string と同じ前処理を行っています。なので、その後の文字列化処理まで同じにすれば f-string を再実装することが可能です。型変換の指定（`s`, `r`, `a`）の処理は `string.templatelib.convert` として、書式指定の処理は組み込み関数 `format` として用意されています。これを順に適用し、文字列として連結します。

リスト 1.5: f-string の再実装

```
from string.templatelib import Interpolation, Template, convert
from typing import Iterator

def _fstr(t: Template) -> Iterator[str]:
  for i in t:
    match i:
      case str():
        yield i
      case Interpolation(value, _, conversion, format_spec):
        value = convert(value, conversion)
        value = format(value, format_spec)
        yield value

def fstr(t: Template) -> str:
  return "".join(_fstr(t))
```

遅延評価

f-string はその場で文字列への変換処理が走るのに対して、 t-string はオブジェクトを保持するだけです。なので、必要になるまで前述の fstr 関数のような変換処理関数にかけずにテンプレートオブジェクトのまま引き回すだけで、処理を遅らせることができます。

また、このやり方は外から変換処理を注入するのにも向きます。置換対象となるオブジェクトをその変換処理関数の部分関数でラップしておきます。その呼び出しをテンプレート処理関数内で行うようにすればよいのです。

リスト 1.6: 置換対象として実行可能オブジェクトを渡す

```
import time
from functools import partial
from string.templatelib import Interpolation, Template

def slow_str(o) -> str:
  # なんらかの重たい変換処理の代わり
  time.sleep(1)
  return str(o)

spam = 'spam'
ham = 'ham'
f = f'{slow_str(spam)} {slow_str(ham)}'  # これだと変換が完了するまで進めない
t = t'{partial(slow_str, spam)} {partial(slow_str, ham)}'  # 止まらずに進める
# 必要になってから slow_str による変換を始める
def my_conv(t: Template):
  temp = []
  for i in t:
    match i:
      case Interpolation(value):
        temp.append(value())  # ここで slow_str が走る
  return ''.join(temp)
ret = my_conv(t)
print(ret)
```

ここから少し手を入れることで asyncio による非同期化も可能です。前述の my_conv における slow_str 関数の非同期版を用意すれば、I/O バウンドの他の処理と並行して変換処理を走らせることができます。

リスト 1.7: asyncio 対応する

```
import asyncio
from string.templatelib import Interpolation, Template

async def aslow_str(o) -> str:
  # なんらかの重たい変換処理の代わり
  await asyncio.sleep(1)
  return str(o)

async def amy_conv(t: Template):
  temp = []
  for i in t:
    match i:
      case Interpolation(value):
```

```
        temp.append(await value)  # ここで aslow_str が走る
  return ''.join(temp)

async def main():
  spam = 'spam'
  ham = 'ham'
  t = t'{aslow_str(spam)} {aslow_str(ham)}'
  ret = await amy_conv(t)
  print(ret)

asyncio.run(main())
```

1.4 使用例

ここからは t-string の使用事例を紹介します。

logging のメッセージに str.format 形式の書式を用いる

Python には `logging` ライブラリがありますが、その出力文字列をフォーマットする方法として対応されたものは `%` を使ったものだけでした。`logging` ライブラリが作られた時期が `str.format` や f-string のない頃であったので当然ではあります。だからといって `str.format` 形式の書式が使いたいからとメッセージに f-string を用いてしまうと、 `logging` ライブラリが持つ「必要とされるまで文字列を構築しない」という機能の恩恵にあずかることができません。

Logging クックブック[*4]にはこの問題を回避しつつ `str.format` の記法を使うための手法が紹介されています。`str.format` を用いてメッセージをフォーマットするクラス `BraceMessage` でラップする、そして `BraceMessage` クラスは短く書けるようにアンダースコアなどの別名をつけておくというものです。

[*4] https://docs.python.org/ja/3/howto/logging-cookbook.html#use-of-alternative-formatting-styles

リスト 1.8: str.format 、中括弧を用いるログメッセージ

```
class BraceMessage:
  def __init__(self, fmt, /, *args, **kwargs):
    self.fmt = fmt
    self.args = args
    self.kwargs = kwargs

  def __str__(self):
    return self.fmt.format(*self.args, **self.kwargs)
_ = BraceMessage

num = 2
name = "placeholders"
print(_('Message with {num} {name}', num=num, name=name))
# もしくは locals 辞書を用いて
print(_('Message with {num} {name}', **locals()))
#  Message with 2 placeholders
```

この手法は t-string とリスト 1.5 の `fstr` 関数を用いることでより良くすることができます。ログに埋め込む変数を引数として渡す処理が省かれることでシンプルな記述ができるようになります。

リスト 1.9: t-string を用いたログメッセージ

```
from string.templatelib import Template

class BraceTMessage:
  def __init__(self, template: Template):
    self.template = template
  def __str__(self):
    return fstr(self.template)
_ = BraceTMessage

num = 2
name = "placeholders"
print(_(t"Message with {num} {name}"))
#  Message with 2 placeholders
```

サードパーティライブラリ tdom

tdom[*5] は Dave Peck 氏[*6] が作成した HTML テンプレートエンジンです。t-string の「自身はデータを蓄えることに専念」「処理は受け取る側で定めることができる」「遅延評価される」という性質を存分に活用して作られています。これにより、「テンプレートのパースと構築」「セキュリティ」「再利用性」を実現しています。

tdom.html 関数に t-string を渡すことで HTML 要素オブジェクトが作成されます。

リスト 1.10: tdom で HTML 要素を表現する

```
from tdom import html
greeting = html(t"<h1>Hello, World!</h1>")
print(type(greeting))  # <class 'tdom.nodes.Element'>
print(greeting)  # <h1>Hello, World!</h1>
```

テンプレートエンジンを名乗るには値の埋め込み機能は当然欲しいところです。もちろん、可能となっています。

リスト 1.11: 値の埋め込み

```
name = "Alice"
age = 30
user_info = html(t"<p>Hello, {name}! You are {age} years old.</p>")
print(user_info)  # <p>Hello, Alice! You are 30 years old.</p>
```

値の埋め込みを行う際に XSS 攻撃は対策しなければなりません。このためのエスケープ機能を持ち合わせています。

リスト 1.12: エスケープ処理に対応していることを確認

```
user_name = "<script>alert('owned')</script>"
safe_output = html(t"<p>Hello, {user_name}!</p>")
print(safe_output)  # <p>Hello, &lt;script&gt;alert('owned')&lt;/script&gt;!</p>
```

[*5] https://pypi.org/project/tdom/

[*6] t-string の提案がなされた PEP 750 の Author の一人です

埋め込むことができるのは文字列だけに限りません。HTML 要素に HTML 要素を埋め込むことも可能となっています。

リスト 1.13: HTML 要素の埋め込み

```
content = t"<h1>My Site</h1>"
element = html(content)
page = html(t"<div>{element}</div>")
# <div><h1>My Site</h1></div>

# t-string を埋め込むこともできる
page = html(t"<div>{content}</div>")
```

置換フィールドの位置によって、挙動が変わります。tdom.html 関数は HTML を解釈し、置換フィールドがどこにあるかによって処理を使い分けます。たとえば リスト 1.14 ではダブルクォートが必要なところかどうかを判断できていることがわかります。

リスト 1.14: 置換フィールドの位置による挙動の変化

```
button_text = "Click me"
element_id = "my-button"
classes = ["btn", "btn-primary", "active"]
button = html(t'<button id={element_id} class={classes}>{button_text}</button>')
# id, class はクォートされ、テキストClick me はクォートされていない
# <button id="my-button" class="btn btn-primary active">Click me</button>

# なお、 HTML 属性値の置換フィールドにダブルクォートがあってもよい。
# 置換フィールドがすでにダブルクォートの中にあることを tdom は認識する。
button = html(t'<button id="{element_id}" class="{classes}">Click me</button>')
```

埋め込むオブジェクトによっても挙動が変わります。文字列、 t-string の出力であるテンプレートオブジェクト、 tdom.html の出力である Element オブジェクトだけでなく、 tdom.html の出力である Element オブジェクトを返す関数を HTML タグ名であるかのように埋め込むなんてこともできます。これにより、再利用可能な HTML コンポーネントの表現が可能です。

リスト 1.15: 置換フィールドの位置による挙動の変化

```
from typing import Any, Iterable
from tdom import Node, html

def MyComponent(children: Iterable[Node], **attrs: Any) -> Node:
  return html(t"<div {attrs}>Cool: {children}</div>")
result = html(t"<{MyComponent} id='comp1'>Hello, Component!</{MyComponent}>")
# <div id="comp1">Cool: Hello, Component!</div>
```

tdom は t-string の能力を存分に活かしたライブラリとなっています。

1.5 おわりに

t-string を用いると f-string を超えた柔軟な文字列構築が可能となります。それだけでなく、新たなテンプレート言語や DSL (Domain Specific Language) を実装できる力をも秘めています。ぜひ、使って、楽しんでみてください。

第 2 章

cgo の魔法 ～ ビルド編

Daisuke Makiuchi / @makki_d

2.1 cgo とは

Go 言語には、コード中に C 言語のコードを埋め込む **cgo** という機能があります。これにより、Go 言語から C や C++ のライブラリを利用するなどの相互連携が可能となります。

リスト 2.1 は cgo の最小のサンプルです。Go のソースのコメント中に C 言語で add 関数を定義し、Go の main 関数で C の add 関数を呼び出しています。

リスト 2.1: cgo の最小のサンプル（main.go）

```
package main

/*
int add(int a, int b)
{
    return a + b;
}
*/
import "C"

func main() {
    s := C.add(1, 2)
    println("add(1,2) =>", s)
}
```

このように、あたかも"C"というパッケージがあるかのように import し、直前のコメント中の C 言語で定義されている関数や変数、マクロなどへのアクセスは普通のパッケージと同

様に . を使って行います。

ここでいくつか疑問が浮かびます。"C"というパッケージは実際には存在しないのに、なぜ import できるのでしょうか。またパッケージ内の識別子へのアクセスは大文字で始まる公開されたものしか許されないはずですが、add は小文字で始まっています。

そもそも、C 言語で定義された識別子や型をどうやって認識しているのでしょうか。#include ディレクティブでの外部ヘッダ読み込みや、マクロの使用など自由な書き方ができます。それにもかかわらず、Go から呼び出すときには関数の引数の数や型の不一致などが検出されます。とても強力な機能ですが、どうやって実現しているのでしょうか。

この章では、cgo を利用したビルドでは何が起こっているのか解説し、これらの疑問を解消します。

2.2 cgo ビルドの全体像

まず最初に、ビルドの流れを俯瞰してみます。cgo を利用したビルドは次のような段階を経ます。

1. Go コードの解析（import "C"の検出）
2. cgo ツールによるコード生成
3. 生成された C コードのコンパイル
4. 生成された Go コードのコンパイル
5. リンク

通常のビルドでは Go のコードがそのままコンパイルされますが、import "C"が含まれる場合、コンパイル前に **cgo** というツールによるコード生成処理が挿入されます。

cgo ツールの役割は、Go コード内のコメントとして書かれている C コードを切り出し C 言語としてコンパイル可能な形にするとともに、Go から利用できるように型やラッパー関数の定義を生成します。

生成された C のコードは gcc や clang などの C コンパイラによってコンパイルされ、.o ファイルが生成されます。生成された Go のコードも Go コンパイラによってコンパイルされ、最終的にすべてのオブジェクトが C リンカによってリンクされて 1 つの実行バイナリになります。

つまり、Go コンパイラが C 言語を直接解釈しているわけではなく、cgo ツールが C と Go を分離し、それぞれのコンパイラで処理できるようにコードを生成していることです。このとき、それぞれの言語向けに適切な型情報が付与されることで、コンパイル時の型チェックもできるようになっています。

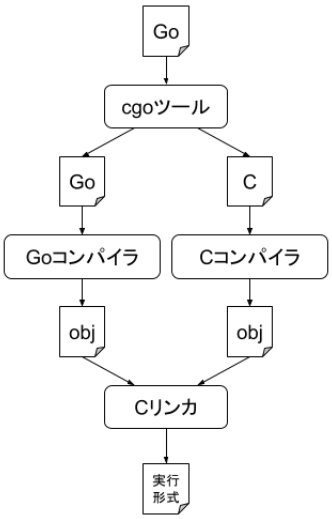


図 2.1: ビルドの流れ

ちなみに、Go のみであれば Go 独自のリンカでリンクされますが、cgo を利用するときは C リンカが使われます。これはシステムライブラリや OS といったプラットフォーム依存のリンク処理の差異を、それぞれの環境ごとの C リンカに任せるようにしているからです。このため、もしクロスコンパイルをしたい場合はそれに対応した C のコンパイラ・リンカが必要になります。

2.3 cgo ツールが生成するコード

cgo ツールはビルド時に暗黙的に呼び出されますが、手動で実行することもできます。リスト 2.1 の `main.go` を処理するには、次のコマンドを実行します。

```
go tool cgo main.go
```

このコマンドを実行すると、筆者の環境[*1]では次のファイルが生成されます。これらのファイルは GitHub にも公開しているので適宜参照してください[*2]。

[*1] Kubuntu 22.04, x86_64, Go 1.26.1, gcc 11.4.0
[*2] `https://github.com/makiuchi-d/cgo-tool-sample`

- _obj/
  - _cgo_export.c
  - _cgo_export.h
  - _cgo_main.c
  - main.cgo1.go
  - main.cgo2.c
- _cgo_2.o

これらのファイルのなかで注目すべきは、main.go から直接変換されている main.cgo1.go と main.cgo2.c、そして C の型や関数定義を Go から参照できるようにしている_cgo_gotypes.go です。

それでは具体的に、元の main.go の add 関数の定義や呼び出しがどのように書き換えられているか見てみましょう。

C コードの切り出し

main.go でコメントとして記述されていた C のコードは、main.cgo2.c の中に他のコードとともにそのまま埋め込まれます。リスト 2.2 の「#line 3 ...」と「#line 1 ...」の間が、元のコメントそのままになっていることがわかります。このファイルが C コンパイラで処理されることになるため、Go のコメント中では C 言語として正しければどんな複雑な書き方も許容されます。

リスト 2.2: main.cgo2.c（抜粋）

```
...(略)...
#line 3 "/home/makki/Projects/test/cgo/main.go"

int add(int a, int b)
{
    return a + b;
}

#line 1 "cgo-generated-wrapper"
...(略)...
```

また、このファイルの末尾にはリスト 2.3 のようなラッパー関数_cgo_eb03ac014299_Cfunc_add が定義されています。これは_cgo_a 構造体を利用して、引数や戻り値を適切に型変換して add 関数を呼び出す、Go の世界との橋渡しをする関数です。

リスト 2.3: main.cgo2.c（抜粋）

```
...(略)...
CGO_NO_SANITIZE_THREAD
void
_cgo_eb03ac014299_Cfunc_add(void *v)
{
    struct {
        int p0;
        int p1;
        int r;
        char __pad12[4];
    } __attribute__((__packed__, __gcc_struct__)) *_cgo_a = v;
    char *_cgo_stktop = _cgo_topofstack();
    __typeof__(_cgo_a->r) _cgo_r;
    _cgo_tsan_acquire();
    _cgo_r = add(_cgo_a->p0, _cgo_a->p1);
    _cgo_tsan_release();
    _cgo_a = (void*)((char*)_cgo_a + (_cgo_topofstack() - _cgo_stktop));
    _cgo_a->r = _cgo_r;
    _cgo_msan_write(&_cgo_a->r, sizeof(_cgo_a->r));
}
```

Go コードの変換

main.cgo1.go は、元の main.go の"C"パッケージ参照部分が置換されたものです。

リスト 2.4: main.cgo1.go

```
// Code generated by cmd/cgo; DO NOT EDIT.

//line /home/makki/Projects/test/cgo/main.go:1:1
package main

/*
int add(int a, int b)
{
    return a + b;
}
*/
import _ "unsafe"

func main() {
    s := ( /*line :12:7*/_Cfunc_add /*line :12:11*/)(1, 2)
    println("add(1,2) =>", s)
}
```

「import "C"」は削除され、C.add 関数の呼び出しは_Cfunc_add という関数に書き換えられています。ここからわかるように、"C"はパッケージではなく cgo を利用するためのマーカーであり、外部パッケージではないためにパブリックでない小文字の識別子でも問題なかったわけです。

_Cfunc_add の定義は、_cgo_gotypes.go に埋め込まれています（リスト 2.5）。

リスト 2.5: _cgo_gotypes.go（抜粋）

```
...(略)...
//go:cgo_unsafe_args
func _Cfunc_add(p0 _Ctype_int, p1 _Ctype_int) (r1 _Ctype_int) {
    _cgo_runtime_cgocall(_cgo_eb03ac014299_Cfunc_add, uintptr(unsafe.Pointer(&p0→
)))
    if _Cgo_always_false {
        _Cgo_use(p0)
        _Cgo_use(p1)
    }
    return
}
```

_Cfunc_add 関数では、_cgo_runtime_cgocall を経由して、C で定義されたラッパー関数_cgo_eb03ac014299_Cfunc_add を呼び出しています。引数や戻り値は C 言語の int を表す_Ctype_int 型になっていて、コンパイル時に型チェックできるようになっています。

また、_cgo_runtime_cgocall の実体は runtime.cgocall 関数で、引数や戻り値の制御はもちろんのこと、C 言語の関数が正しく動くようにさまざまな処理を行っています。C 言語と Go 言語では、スタックやスレッドの扱いといった ABI（Application Binary Interface）の違いがあるため、直接関数ポインタにジャンプするだけでは正しく実行できません。この違いを吸収するために、runtime.cgocall 関数が必要になります。このランタイム時の挙動もまた非常に面白いので、次の機会に「ランタイム編」として紹介したいと思います。

このように、コメントとして記述された C のコードがコンパイル可能な形のコードへと変換され、また Go のコードからの呼び出し部分も変換されて、生成コードを介して C 言語の世界へアクセスできるようになります。

2.4 型情報の収集

ここまでで、cgo ツールがコードを生成していることがわかりました。Go の_Cfunc_add 関数や C の_cgo_a 構造体では、引数や戻り値の型が正しく型付けされています。では、この

型情報はどこから来たのでしょうか？ C 言語では#include での外部ヘッダ読み込みやマクロ展開もあるため、文字列解析だけでは正確な型情報を得られません。

そこで cgo ツールは、埋め込まれた C のコードを一度コンパイルして、.o ファイルからデバッグ情報（DWARF 情報[*3]）を読み取るという方法を取っています。つまり、型情報を取得するためのコンパイルと、型情報付きのコードを埋め込んだあとの実行可能にするためのコンパイルの、合計 2 回のコンパイルを実行することになります。一見すると無駄に見えますが、こうすることで自由な形での C の記述を許しつつ、型情報などを確実に抽出、生成できるようにしているのです。

DWARF 情報を見る

それではここで、DWARF 情報から add 関数の型情報を抽出してみましょう。DWARF 情報には関数名、引数、型の情報がすべて含まれているので、ひとつひとつ追いかけていきます。詳しいフォーマットの説明は省いて、必要な部分だけを見ていきます。

Linux でコンパイルした.o ファイルは elf 形式なので、readelf コマンドで DWARF 情報を閲覧できます。

```
readelf -wi _cgo_2.o
```

このコマンドを実行するとリスト 2.6 の出力が得られます。

リスト 2.6: _cgo_2.o の DWARF 情報（抜粋）

```
...(略)...
 <1><53>: Abbrev Number: 4 (DW_TAG_base_type)
    <54>   DW_AT_byte_size   : 4
    <55>   DW_AT_encoding    : 5 (signed)
    <56>   DW_AT_name        : int
...(略)...
 <1><13d>: Abbrev Number: 14 (DW_TAG_subprogram)
    <13e>   DW_AT_external    : 1
    <13f>   DW_AT_name        : add
    <143>   DW_AT_decl_file   : 2
    <144>   DW_AT_decl_line   : 4
    <145>   DW_AT_decl_column : 5
    <146>   DW_AT_prototyped  : 1
```

[*3] https://dwarfstd.org/

```
    <147>   DW_AT_type        : <0x53>
    <14b>   DW_AT_low_pc      : 0x3e
    <153>   DW_AT_high_pc     : 0x56
    <15b>   DW_AT_frame_base  : 0x0 (location list)
    <15f>   DW_AT_GNU_all_call_sites: 1
    <160>   DW_AT_sibling     : <0x17f>
 <2><164>: Abbrev Number: 15 (DW_TAG_formal_parameter)
    <165>   DW_AT_name        : a
    <167>   DW_AT_decl_file   : 2
    <168>   DW_AT_decl_line   : 4
    <169>   DW_AT_decl_column : 13
    <16a>   DW_AT_type        : <0x53>
    <16e>   DW_AT_location    : 2 byte block: 91 6c      (DW_OP_fbreg: -20)
 <2><171>: Abbrev Number: 15 (DW_TAG_formal_parameter)
    <172>   DW_AT_name        : b
    <174>   DW_AT_decl_file   : 2
    <175>   DW_AT_decl_line   : 4
    <176>   DW_AT_decl_column : 20
    <177>   DW_AT_type        : <0x53>
    <17b>   DW_AT_location    : 2 byte block: 91 68      (DW_OP_fbreg: -24)
 <2><17e>: Abbrev Number: 0
...(略)...
```

まず DW_AT_name が add となっている DW_TAG_subprogram を探すと、オフセット<13d>に見つかります。関数の戻り値を表す DW_AT_type は<0x53>となっているので、オフセット<53>の DW_TAG_base_type の DW_AT_name をみると、この型は int とわかります。

DW_TAG_subprogram に続く、先頭が子要素を表す<2>になっている DW_TAG_formal_parameter がこの関数の引数で、DW_AT_name が a と b の 2 つあり、DW_AT_type はどちらも<0x53>、つまり int です。

これらから、add 関数のプロトタイプは次の形であることがわかりました。

- `int add(int a, int b)`

このようにして、cgo ツールは C.add のように C の識別子にアクセスしているものすべての型を調べます。その型情報を元に、_cgo_gotypes.go の型定義や関数定義、*.cgo2.c のラッパー関数を生成していきます。

cgo ツールの解析戦略

Go 言語には Go のコードをパースする標準パッケージ go/parser があるため、Go のコードを解析してコード生成することは容易です。cgo ツールも標準パッケージを利用して Go の

コードをパースしてコード生成に利用しています。本章では触れませんでしたが、Go 側で export した関数から C のコードを生成して、C から Go の関数を呼び出せるようにする機能もあります。

一方で、C のコードについては自前でパースすることなく、C コンパイラに任せて DWARF 情報を利用するという戦略を取っています。外部の C コンパイラに依存してしまうためクロスコンパイルが難しくなる問題はあるものの、複雑な C コードであっても cgo で利用できるようになっています。

このように、cgo ツールはそれぞれの言語に対する信頼できる仕組みを利用し、堅実で現実的な実装をしているといえるでしょう。

2.5 まとめ

この章では、Go 言語の cgo のビルド時の挙動、特に cgo ツールによるコード生成について紹介しました。このコード生成によって、C で定義した関数や定数などが Go としても型付けされた形で利用できます。cgo は単純に C を呼び出す仕組みを提供しているのではなく、型安全に利用するための機構まで含めている高度な機能です。Go に埋め込んだ C のコードから型情報を取り出すために、一度コンパイルして DWARF 情報を利用しているのは、興味深い戦略ではないでしょうか。

cgo では C と Go の橋渡しをするコードがいくつも生成されて、うまく繋がるようにビルドされることがわかりました。しかし、実際に C と Go が連携して正しく動作するには、ABI の違いもあり一筋縄にはいきません。実行時のさまざまな処理が必要になります。

次回はランタイム編として、C と Go の間にどのような違いがあるのか、runtime.cgocall はどのようにその差を埋めているのかについて紹介したいと思います。

第 3 章

社内ゲームジャムに GitHub Issue だけで参加する

山口 大祐

3.1　はじめに

KLab では、年末年始の休暇を利用して社内ゲームジャムを開催しています。私はこれまで技術広報として運営側で協力してきましたが、普段の業務はサーバーサイドの API 実装やインフラ構築が中心で、UI 開発の経験がほとんどありません。

そんな私でもゲーム開発ができるなら、企画やデザインなど他の分野の人もゲームジャムに参加できるようになるのではないか――そう考えて、GitHub Issue だけでゲームジャムに参加してみることにしました。

コードは一行も自分で書かず、すべて GitHub 上で Issue を立てるだけで開発を進めます。本章では、その体験を通じて得られた知見とエピソードを紹介します。

3.2　つくったもの

会社のオフィスでフロア移転があった年だったので、オフィスの引っ越しをコンセプトにしたファミコン版ドンキーコング風のアクションゲームを作りました。

ゲームのタイトルは「運べ！ 引っ越し荷物」です。プレイヤーは水色のスーツを着た新入社員で、段ボール箱を抱えながら、宴会中の酔っぱらいが転がしている酒瓶の誘惑を避けつつ、光る扉のゴールを目指します。



図 3.1: タイトル画面

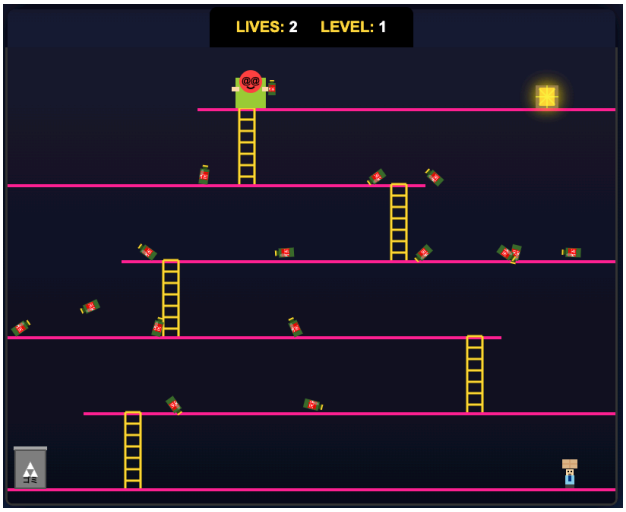


図 3.2: ゲームプレイ画面

ゲームは全 4 レベルで構成されています。

- LEVEL1：通常の移動とジャンプで酒瓶を避けながらゴールを目指す
- LEVEL2：移動速度が遅くなる代わりにジャンプ力が上がる
- LEVEL3：台車で荷物を運ぶためジャンプ禁止、はしごだけで攻略する
- LEVEL4：時間制限 20 秒のスピードラン

HTML5 の Canvas API を使ったブラウザゲームとして作成しました。外部ライブラリは使わず、Vanilla JavaScript のみで構成されています。音響も Web Audio API でプログラム生成しており、外部の音声ファイルは一切使っていません。

- 公開 URL：`https://klab-yamaguchi-da.github.io/GameJam2/`
- リポジトリ：`https://github.com/klab-yamaguchi-da/GameJam2`

このあとの解説で Issue と PR の概要を記載している部分があります。ですが、ぜひ実際どのような Issue から PR が作られているかリポジトリと見比べながら読んでみてください。

3.3 GitHub Copilot Coding Agent について

今回利用したのは、GitHub Copilot Coding Agent[*1] という機能です。

Visual Studio Code などの IDE の拡張機能として使う Agent モードとは異なり、GitHub 上で完結する仕組みです。Issue や Pull Request に Copilot をアサインすると、GitHub Actions 上で自動的にコーディングが行われ、Pull Request が作成されます。

具体的な流れは以下のとおりです。

1. GitHub Issue に実現したいことを自然言語で記載する
2. Issue に Copilot をアサインする
3. Copilot が自動でブランチを作成し、コードを変更し、Pull Request を作成する
4. 作成された Pull Request を確認し、必要に応じてコメントで修正を指示する
5. 問題なければマージする

Copilot は内部でコードの実行やテストも行えるため、ブラウザ上の HTML5 ゲームのようなシンプルな構成であれば、動作確認まで含めて自動で行ってくれます。今回の開発では、Pull Request にスクリーンショットまで添付してくれることが多く、マージ前の確認が容易でした。

[*1] `https://docs.github.com/ja/copilot/concepts/agents/coding-agent/about-coding-agent`

3.4 やったこと

開発の流れを時系列に沿って紹介します。全体で 41 件の Issue を作成し、41 件の Pull Request が生成されました。

環境整備

最初に、開発環境の整備を複数の Issue を使って段階的に指示しました。

１つ目は、基本的なリポジトリ内のファイル構成の整備です。

リスト 3.1: Issue #1

```
Title: 基本的なファイルの整備
概要:
GithubActions用のdispatchで起動する echo workflow
GithubPages用のdocsディレクトリと Hello を返す index.html と ☒ の絵文字の favicon
copilot 用の instructions ファイル
それぞれ作成してください
```

これにより、以下のような PR の中に、GitHub Pages でゲームを公開するための基本的なファイルや Actions などのリポジトリの基本的なファイル構成ができました。

リスト 3.2: PR #2

```
生成されたファイル一覧
- .github/copilot-instructions.md
- .github/workflows/echo.yml
- docs/favicon.ico
- docs/index.html
```

次に、Copilot 用の instructions ファイルに基本的なリポジトリの運用方針を記載するように指示しました。

3.4 やったこと

リスト 3.3: Issue #3

```
Title: instructions に基本的な方針を記載
概要:
このリポジトリは、社内ゲームジャム向けに web3 でゲームを作るリポジトリです。
docs/index.htmlにゲームを作ります。適切にファイル分割してください。
Copilotでの作業完了時に、作業内容を元にinstructionsを更新可能であれば更新するように設定→
してください。
README.mdも常に適切に更新するようにしてください。
このリポジトリは日本語で運用します。PullRequestの名前やコメントなどは日本語で記載してくだ→
さい。
```

このルールを最初に設定しておくことで、以降の Issue で毎回指示しなくても一貫した運用がされるようになります。Issue 概要を元に、以下のような PR 方針で instructions ファイルの更新と README.md の作成がされました。

リスト 3.4: PR #4

```
社内ゲームジャム向けWeb3ゲーム開発リポジトリの運用方針とガイドラインを整備。

## 変更内容

### `.github/copilot-instructions.md`
- **Web3ゲーム開発**の明記とセキュリティガイドライン追加
- **日本語運用**の明確化（PR、コミット、コメント全て日本語）
- **ファイル分割方針**: `docs/index.html`をエントリーポイントとし、JS/CSS/アセットを適→
切に分離
- **ドキュメント管理**:
  - README.mdを変更毎に更新
  - 作業完了時にinstructionsも更新（新パターン、構造変更、技術的決定事項）

### `README.md`
- プロジェクト概要、技術スタック、構造を追加
- 開発方針（ファイル構成、コーディングルール）を明記
- GitHub Pagesデプロイ情報とセットアップ手順を記載

Copilotと開発者が一貫した方針で開発できる基盤を整備。
```

その他、**このリポジトリで唯一の自分で直接コミットしたもの**として、社内の GitHub リポジトリに同期転送を行う GitHub Actions の Workflow を環境整備として設定しました。

ベースのゲーム構築

ベースとなるファミコン版のドンキーコング風ゲームについては、以下のような簡単な Issue だけで作成できました。

リスト 3.5: Issue #5

```
Title: ベースのゲーム
概要: ベースのゲームとして、ファミコン版のドンキーコングを作ってください
```

これだけで、プレイヤーの移動・ジャンプ・はしご登り、樽の自動生成と転がり、衝突判定、スコア・ライフシステム、画面遷移（タイトル→プレイ→クリア/ゲームオーバー）まで、一通り動くゲームが生成されました。コード量は約 1,000 行、8 ファイルに適切に分割された状態です。

PR #6 で生成されたコードは、設定値が `config.js` に集約されたデータ駆動の設計になっており、後のレベル追加やパラメータ調整がしやすい構造でした。

リスト 3.6: PR #6

```
生成されたファイル構成
- docs/index.html：ゲームのエントリーポイント（タイトル画面・Canvas要素）
- docs/css/style.css：ゲーム画面のスタイル定義
- docs/js/config.js：ゲーム設定の定数管理（物理パラメータ、キャラクター設定、レベル設計→
)
- docs/js/main.js：UI制御・画面遷移（タイトル→プレイ→クリア/ゲームオーバー）
- docs/js/game.js：ゲームループ、エンティティ管理、状態管理、スコア・ライフシステム
- docs/js/player.js：プレイヤーの移動・ジャンプ・はしご登り・重力・衝突判定
- docs/js/barrel.js：タルの自動生成・プラットフォーム移動・はしご降下・衝突判定
- docs/js/donkeykong.js：ドンキーコングとプリンセスのキャラクター描画・アニメーション
- README.md：プロジェクト説明の更新
```

3.4 やったこと

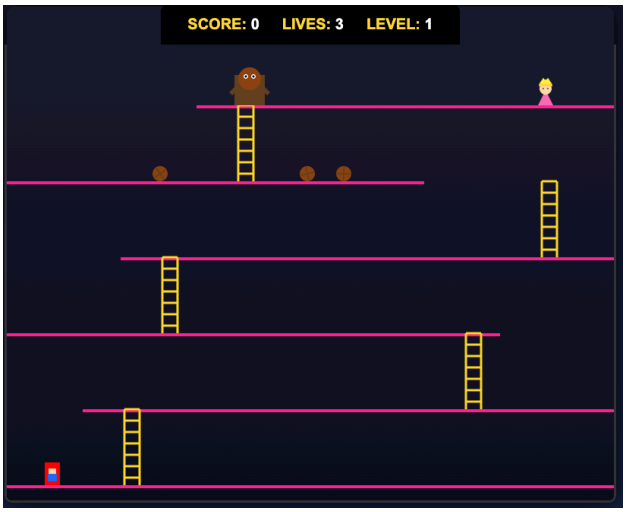


図 3.3: ドンキーコング風のベースゲーム

モチーフの変更

ドンキーコングそのままから、コンセプトにあわせてゲームのモチーフを変更しました。Issue #21 では以下のように指示しています。

リスト 3.7: Issue #21

```
Title: モチーフ変更
概要：
ゴリラのモチーフを赤い顔で黄緑の服の酔っぱらいに変更する。
樽を酒瓶に変更する。
姫を光る扉に変更する。
プレイヤーを水色のスーツの新入社員に変更する。プレイヤーは段ボール箱の荷物を運んでいる
```

これだけの指示で PR #24 が作成され、各キャラクターの描画メソッドが書き換わり、ゲームロジック（衝突判定や物理演算）はそのままに、見た目だけが変更されました。酔っぱらいの「@@」の目やへらへら笑顔、酒瓶の「酒」ラベルなど、Canvas API の描画コードで細かく表現してくれたのは予想外でした。

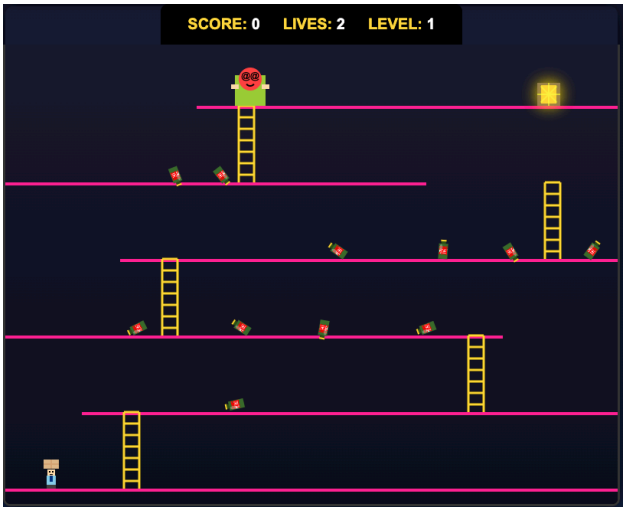


図 3.4: モチーフ変更後のゲーム

ステージの追加と調整

LEVEL1 のステージができた後は、ステージの追加と調整を繰り返しました。

LEVEL2 の追加

LEVEL1 がだいぶ固まってきた段階で次のステージである LEVEL2 の追加を指示しました。

リスト 3.8: Issue #39

```
Title: LEVEL2を作る
概要:
LEVEL１をクリアしたらLEVEL2が始まる。
開始の文面「次の荷物は軽いけどゆっくり運ばなきゃ」
設定：横移動速度を０．７倍。ジャンプ力を１．５倍。とする
```

これだけで PR #40 上で、レベルシステムの拡張、ストーリー画面の実装、新しいマップレイアウトの生成まで自動で行われました。`speedMultiplier` や `jumpMultiplier` といったパラメータが `config.js` に追加され、レベルごとに異なるゲーム性が実現されています。

3.4 やったこと



図 3.5: LEVEL2 の初期ゲーム画像

LEVEL3・4 の追加

Issue #43 では、LEVEL3（ジャンプ禁止＋台車モード）と LEVEL4（20 秒タイムリミット）を一つの Issue でまとめて依頼しました。LEVEL4 のカウントダウンタイマーは、残り 5 秒以下で赤色警告になる演出まで自動で実装されていました。

リスト 3.9: Issue #43

```
Title: LEVEL3,4を作る
概要:
LEVEL2をクリアしたらLEVEL3が始まる。
開始の文面「次の荷物は重いけど台車がある」
設定：横移動速度を１倍。ジャンプ力を不可。とする

LEVEL3をクリアしたらLEVEL4が始まる。
開始の文面「最後の荷物は急がなきゃ」
設定：横移動速度を１倍。ジャンプ力も１倍。ただし、２０秒のカウントダウンを左下に表示する
```

細かなステージ調整

ステージの調整は最も多くの Issue を費やした部分です。

- ゴミ箱の設置（Issue #27）：空き瓶が画面下に溜まる問題を解決するため、ゴミ箱を設置してボトルを自動回収する仕組みを追加
- はしごの位置調整（Issue #25, #79 など）：プラットフォーム間の接続が途切れている箇所の修正
- 酔っぱらいの追加配置（Issue #54, #69）：複数の酔っぱらいを配置して難易度を調整
- ゲームオーバー時の残機回復（Issue #49）：再チャレンジのたびに開始時のライフが 1 つ増える仕組みを追加

この他にもいくつかの調整 Issue と PR を通して、特に LEVEL2 と LEVEL3 は最終的に大きく変わりました。

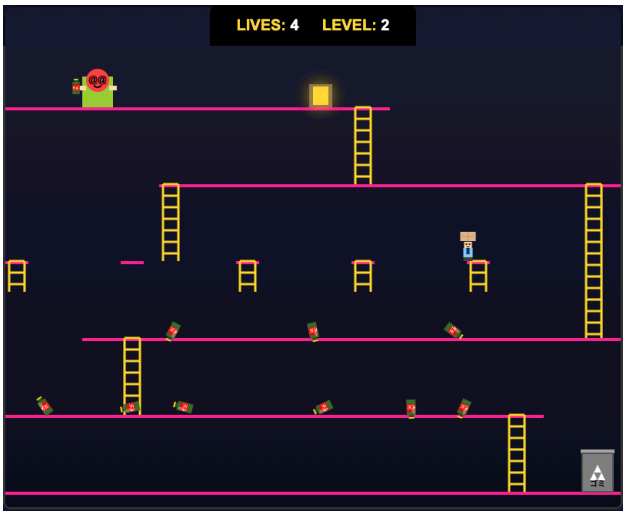


図 3.6: LEVEL2 の最終的なゲーム画像

3.4 やったこと

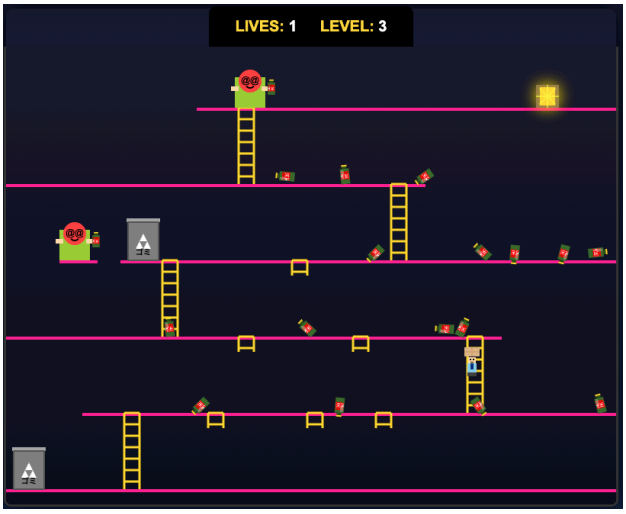


図 3.7: LEVEL3 の最終的なゲーム画像

音の追加

BGM や効果音も、以下のような簡単な Issue だけで追加できました。

リスト 3.10: Issue #65

```
Title: 音をつける
概要:
8ビット風の音を追加してください
* レベルBGM
* 歩行 SE
* ジャンプ SE
* 死亡 SE
* クリア SE
* ゲームクリアの２０秒ミュージック
```

結果として、Web Audio API を使って矩形波・三角波・サイン波を組み合わせた 8 ビット風サウンドがすべてプログラムで生成されました。BGM はチップチューン風のメロディで 8 秒ループ、ゲームクリア音楽は 20 秒のファンファーレが作成されています。外部の音声ファイルを一切使わず、すべて JavaScript で音を生成しているのは、Copilot が実行環境の制約を理解した上での判断だと思われます。

PR #66 では 456 行の `audio.js` が新規作成され、6 種類の音響効果が一度に実装されました。

リスト 3.11: PR #66

```
新規作成: docs/js/audio.js

矩形波・三角波・サイン波を使用した音源生成
BGMバッファのキャッシング機構
6種類の音響効果:
* レベルBGM（8秒ループ、チップチューン風メロディ）
* 歩行SE（800Hz矩形波）
* ジャンプSE（300Hz→800Hz上昇音）
* 死亡SE（600Hz→100Hz下降音、三角波）
* ステージクリアSE（C5-E5-G5-C6ファンファーレ）
* ゲームクリア音楽（20秒、メロディ+ハーモニー）
```

リスト 3.12: ステージクリアの SE データ箇所の抜粋

```
        // クリア音は4つの音符で構成（ファンファーレ風）
        const notes = [
            { freq: 523.25, start: 0.0, duration: 0.12 },    // C5
            { freq: 659.25, start: 0.12, duration: 0.12 },   // E5
            { freq: 783.99, start: 0.24, duration: 0.12 },   // G5
            { freq: 1046.50, start: 0.36, duration: 0.24 }   // C6
        ];
```

バグフィックス

こういった開発をしてくる中で、一見動いてそうに見えても多数のバグがありました。それらバグフィックスも Issue で対応できました。そのうちいくつかを紹介します。

- プレイヤーがはしごを登るときに x 座標がずれる問題（Issue #25）
- 転がる樽がステージの端で落下しない問題（Issue #13）
- プレイヤーがはしごの上でジャンプできなくなる問題（Issue #33）
- はしごを降りると下に突き抜ける問題（Issue #77）
- 上のステージからはしごを降りられない問題（Issue #55）
- LEVEL3 でジャンプが禁止されない問題（Issue #47）：||演算子が 0 を偽値として扱う JavaScript の罠

特に Issue #47 のバグは興味深いものでした。`jumpMultiplier: 0` を設定しているの

にジャンプできてしまう原因は、`levelData.jumpMultiplier || 1.0` という記述で 0 が false 値として扱われ、`1.0` にフォールバックしていたことでした。以下のような Issue だけで、Copilot は原因を特定し `||`を`??`（Nullish Coalescing 演算子）に変更することで正しく修正してくれました。

リスト 3.13: Issue #47

```
Title: LEVEL3はジャンプ禁止
概要:LEVEL3ではジャンプ力を0にしてください
```

リスト 3.14: PR #48

````
**説明:**
LEVEL3では`jumpMultiplier: 0`が設定されているが、`||`演算子が`0`を偽値として扱うため`1→
.0`にフォールバックし、ジャンプが無効化されていなかった。

## 変更内容
**game.js** - `||`を`??`に変更（2箇所）
```javascript
// Before: 0 falls back to 1.0
levelData.jumpMultiplier || 1.0

// After: 0 is preserved
levelData.jumpMultiplier ?? 1.0
```

**player.js** - ジャンプ条件に`jumpMultiplier > 0`を追加
```javascript
if (keys[' '] && this.isOnGround && this.jumpMultiplier > 0) {
    this.velocityY = -CONFIG.PLAYER.JUMP_POWER * this.jumpMultiplier;
    // ...
}
```
````

3.5　エピソード

微調整の誘惑との戦い

ステージのパラメータ調整は「はしごの x 座標を 390 にして」「酔っぱらいを右に 40 ピクセル移動して」のように、数値を直接指定して行いました。

正直なところ、`config.js` を開いて数値を直接書き換えてコミットを作ったほうが早いのは明らかです。しかし今回は「誰でもゲームジャムに参加できることの実験なので、Issue だけで開発する」という狙いがあったので、ひたすら Issue に座標を書き続けました。LEVEL3 の調整だけで Issue #45, #54, #69, #73, #79 と 5 件の Issue を費やしています。

並列 Issue の罠

GitHub Copilot Coding Agent は、複数の Issue に対して並列に Pull Request を作成します。これは一見効率的に見えますが、同じファイルを変更する Issue を並列に立てると、マージ時にコンフリクトが発生します。

実際に PR #61 では main ブランチとのコンフリクトが発生し、Close して新しい Issue としてやり直す必要がありました。コンフリクトの解消を Copilot に依頼することもできますが、別の PR の変更まで混ざってしまうこともあり（PR #18 のコメント参照）、結局 Close してやり直すほうが確実でした。

この経験から、同じファイルに触れる Issue は直列に進めるのが安全だとわかりました。

いつでもどこでも開発できる

GitHub Issue を作るだけなので、PC がなくてもスマートフォンからアイデアをメモする感覚で開発を進められます。実際に、正月の初売りの列に並びながら Issue を立てたりしていました。

Issue を立てておけば、Copilot が自動で Pull Request を作ってくれるので、後で結果を確認するだけです。休暇中に気軽に参加するスタイルとしては、この非同期性がとても合っていました。

プレイ会当日のライブデバッグ

年が明けてゲームの提出が出揃ったところで社内プレイ会を実施しました。当日、会場のディスプレイでゲームを動かしたところ、ゲームスピードが自宅と違うことに気づきました。原因はフレームレートがディスプレイのリフレッシュレートに依存していたことです。

その場で Issue #81「ディスプレイのフレームレートでゲームスピードが変わっているので固定する」を立てて放置しておいたところ、しばらくして PR #82 が自動で作成されました。

Copilot はアキュムレータベースの固定タイムステップゲームループを実装し、スパイラルオブデス防止やタブ切り替え対応まで含めた堅牢な修正を行ってくれました。

社内プレイ会の場で Issue を立てるだけでバグが修正されていたのは、ちょっとしたライブコーディングのようでした。

PR コメントでの追加指示

Issue に対して作成された Pull Request の内容を確認して、さらに微調整したい場合があります。そのような場合は、Pull Request 上で`@copilot` のメンション付きでコメントすることで、追加の修正を依頼できます。

たとえば PR #80 では、はしごの配置調整後に「下の酔っぱらいの位置を変更する。右に 40、小さい足場と一緒に移動させる」とコメントしたところ、同じ PR 内で追加コミットが作成されました。

PR #58 でも、レイアウト調整後に「移動速度を 0.5 に遅くする」と追加コメントして、同じ PR 内で対応してもらっています。

この仕組みにより、Issue を新しく立てなくても、PR の中で IDE のチャットで追加指示を出すように調整を進められました。

3.6 まとめ

GitHub Issue だけでゲーム開発ができるのか、という実験的な取り組みでしたが、実際にプレイ可能なゲームを完成させることができました。

- ドンキーコング風のオーソドックスなゲームであれば、1 つの Issue である程度動くものが生成される
- モチーフの変更、ステージ追加、音の追加など、自然言語の指示で開発を進められる
- JavaScript やゲーム開発の経験がなくても、ブラウザ上で動くゲームが作れる
- バグフィックスも、症状を記載するだけで原因の特定と修正まで行ってくれる

一方で、限界や注意点もあります。

- ステージの座標調整のような微調整は、直接コードを書くよりも時間がかかる
- 並列に Issue を立てるとコンフリクトが発生しやすい
- これだけで最終的な製品品質のゲームを作るのは現実的ではない

とはいえ、プロトタイピングの段階では十分に活用できると感じました。特に社内ゲームジャムのようなお祭りイベントにおいて、企画やデザインなど普段コードを書かない分野の人が参加するための手段としては、大きな可能性を感じます。

思い立ったときに Issue をメモしておくだけで、後から結果を確認できる。この非同期で気軽なスタイルは、休暇期間中のゲームジャムにはちょうど良い参加の仕方でした。

第 4 章

Copilot 主導で進める Unity ゲーム開発

takeichi-a

4.1　はじめに

本章では、GitHub Copilot を全面的に活用して Unity のゲームを開発した経験について紹介します。

開発期間は約 2 週間、開発者は筆者 1 名のみです。使用した環境は VSCode 上の GitHub Copilot（Claude Sonnet 4.5）、ゲームエンジンは Unity 6.3 です。

開発したのは、ボードゲームを題材にしたターン制のゲームです。ゲームジャム[*1]への提出を目標に、設計方針の決定やタスクの指示は人間が行い、実際のコーディング作業はほぼすべて Copilot に任せるというスタイルで進めました。

社内で定期的にゲームジャムが開催されていることは以前から知っていましたが、毎回参加しそびれていました。そんな折、年末に友人と遊んだボードゲームが面白く、「これをいつでも遊べるようにデジタルゲーム化したい」と思い立ったのがきっかけです。ただし、まとまった開発時間を確保するのは難しい状況でした。そこで、最近有効に活用できると聞いていた Copilot の習熟も兼ねて、コーディングをすべて Copilot に任せる「完全 Copilot 進行」で挑戦することにしました。

本章ではまず実装した機能の全体像を示したうえで、各機能における設計の意図や Copilot との協業で工夫した点を解説します。最後に、このような「完全 Copilot 進行」で感じたメリットとデメリットについてもまとめます。

[*1] 短期間でゲームを制作するイベント。限られた時間の中で完成を目指すため、開発効率が重要になる

4.2 実装機能の全体像

ゲームルールについてはあまり深堀りしないですが、経済をテーマした拡大再生産系のゲームを参考にしています。

本プロジェクトで実装した主な機能は以下のとおりです。

- タイトルとインゲームの 2 シーン構成
- CPU 機能
- オンラインマルチ機能
- シーン・プレハブを生成するエディタ拡張
- 特定のゲーム状況を再現するセットアップツールの実装
- 完全イベント駆動のゲーム設計
- 複数のゲームモード構成（未完成）
- スクリプタブルオブジェクトを使ったゲームデータ設計
- Unity のログを自動で読み取る仕組み

以降の節では、それぞれの機能について詳しく説明します。

4.3 タイトルとインゲームの 2 シーン構成

ゲームを作り始める初期段階から、オンラインマルチプレイを想定していました。そのため、シーン構成は次の 2 つに絞り、初めから明確に分離して設計しました。

- **タイトルシーン**：ソロプレイまたはマルチプレイの募集を行う準備画面。参加人数の設定もここで行う
- **インゲームシーン**：オンライン・オフラインを問わず、共通のルールで動作するゲーム本体の画面

リザルト表示を 3 つ目のシーンとして分離する選択肢もありましたが、ゲーム全体の規模を考慮し、インゲームシーンに統合する形としました。オンラインとオフラインで共通のルールに基づいて動作するインゲームの構成を初期段階から意識していたことが、後のオンラインマルチ機能の実装をスムーズに進めるうえで役立ちました。

4.4 CPU 機能

「いつでも遊べるゲーム」にするためには、一人でも遊べる必要があります。そこで CPU プレイヤーの実装は、開発の早い段階から進めました。

設計上のポイントは、プレイヤーの操作をすべてインプット層[*2]を経由させることです。具体的には、PlayerData.isCPU フラグによってプレイヤーの種別を判定し、ゲーム進行のフェーズ処理内で分岐させています。

リスト 4.1: ActionPhase での人間/CPU 分岐

```
private void ActionPhase()
{
    if (AllPlayersNoWorkers())
    {
        ChangePhase(GamePhase.Payment);
        return;
    }
    if (CurrentPlayer != null && CurrentPlayer.isCPU)
    {
        ExecuteCPUTurn(CurrentPlayer); // CPU: 自動でターン実行
    }
    // 人間プレイヤーの場合はUI入力を待機
}
```

CPU の場合は即座に思考ルーチンが呼ばれ、人間プレイヤーの場合は UI 側からのアクション（ワーカーのドラッグ＆ドロップなど）を契機にターンが進行します。この方針を徹底したことで、人間も CPU もゲーム側の動作に差が生じない設計を実現しました。なお、インプット層を専用のインターフェースとして切り出す設計も検討しましたが、今回の規模ではフラグによる分岐で十分と判断し、シンプルな実装を優先しました。これは Copilot に対しても分岐の意図が伝わりやすく、「このプレイヤーが CPU なら自動で動かして」という指示だけで正しく実装できる利点がありました。

[*2] プレイヤーからの入力を抽象化し、ゲームロジックとの間に置く中間層。これにより入力元が人間でも CPU でも同一のインターフェースでゲームを動かせる

CPU の難易度設計

CPU のレベル調整はゲームロジックそのものの理解が求められるため、難易度が高い領域です。今回は 4 段階の難易度（Easy / Normal / Hard / Expert）を用意し、それぞれ異なる評価戦略を実装しました。

リスト 4.2: 難易度別の建物評価分岐

```
public static float EvaluateBuilding(
    BuildingInstance building, PlayerData player,
    int currentRound, int totalRounds,
    CPUDifficulty difficulty)
{
    switch (difficulty)
    {
        case CPUDifficulty.Normal:
            return EvaluateBuildingNormal(...);
        case CPUDifficulty.Hard:
            return EvaluateBuildingHard(...);
        case CPUDifficulty.Expert:
            return EvaluateBuildingExpert(...);
        default:
            return EvaluateBuildingNormal(...);
    }
}
```

各難易度の戦略は次のとおりです。

- **Easy**：ルールベース。基本的な優先順位に従い、一定確率でランダム行動を取る
- **Normal**：各効果を数値化したうえで、ランダム性を加味した選択を行う
- **Hard**：勝利点到達計算と残りラウンド数に応じた緊急度の重み付けを追加
- **Expert**：シナジー評価、コスト管理シミュレーション、終盤ブーストなどの高度な戦略を統合

Copilot にこれらの難易度別ロジックを実装させるには、ゲームルールを正確に伝える必要がありました。たとえば Expert 難易度では、「終盤 2 ラウンドではスコアを大幅にブーストして、残りラウンドの少なさに応じた戦略切替を入れて」といった解像度の指示が求められます。単純に「強い CPU を作って」とだけ伝えても、ゲーム特有の戦略は反映されません。この領域は人間側のゲーム理解が問われる部分でした。

4.5 オンラインマルチ機能

オンラインマルチ機能の実装には Unity 公式の Netcode for GameObjects を採用しました。この機能の実装では、Copilot への指示に特に気を配る必要がありました。

ルームの検索機能や同期の機能について、Copilot に実装を依頼すると、Unity 公式が推奨しない古い API を使った実装を提示してくることがありました。これは、Copilot の学習データに古い情報が多く含まれていることが原因と考えられます。この問題に対処するため、最終的には Unity 公式のサンプルプロジェクトをダウンロードし、リポジトリ直下に丸ごと配置して Copilot の参考資料として利用しました。具体的なコードを「教材」としてプロジェクト内に置くことで、Copilot がそのコードを参照し、最新の API に沿った実装が得られるようになりました。

ゲーム状態の同期には `NetworkVariable<T>`を使用し、サーバーからクライアントへの通知は `[ClientRpc]` で実装しています。

また、ルーム検索用の ID は Unity Services が生成する英数字のセッションコードをそのまま使わず、6 桁の数字コードに変換しています。

リスト 4.3: ルーム ID の 6 桁数字変換

```
private string GenerateNumericCode()
{
    return UnityEngine.Random.Range(100000, 999999).ToString();
}

// セッション作成時にプロパティとして保存
var sessionOptions = new SessionOptions
{
    Name = $"Room_{displayRoomCode}",
    SessionProperties = new Dictionary<string, SessionProperty>
    {
        { "NumericCode",
            new SessionProperty(displayRoomCode) }
    }
};
currentSession = await MultiplayerService.Instance
    .CreateSessionAsync(sessionOptions);
```

プレイヤーは英数字混じりのセッションコードではなく、覚えやすい 6 桁の数字を入力するだけでルームに参加できます。

4.6 シーン・プレハブを生成するエディタ拡張

Copilot との協業において、シーンやプレハブの作成を人間が担当するとボトルネックになることがわかりました。人間側が Copilot の意図をくみ取って事前に準備する必要が出てくるうえ、AI の作業ペースに人間が追いつけず、開発のスピードが落ちてしまいます。

この課題を解決するために、エディタ拡張のオブジェクト生成機能を活用しました。シーンやプレハブの作成、コンポーネントのアタッチまで、すべて Copilot にスクリプト経由で実行させる方式です。実装は `[MenuItem]` 属性を付けた static メソッドで構成されており、Unity メニューから実行できます。

リスト 4.4: プレハブの一括生成

```
[MenuItem("MyGame/Setup/Create All UI Prefabs")]
public static void CreateAllUIPrefabs()
{
    CreateHandPanelUIPrefab();
    CreateBuildingPanelPrefab();
    // ... 20種類以上のプレハブを生成
    AssetDatabase.SaveAssets();
    AssetDatabase.Refresh();
}
```

シーン生成側では、プレハブの生成後に `SerializedObject` でアセットを自動アサインし、シーンごと保存します。

リスト 4.5: シーン生成とアセットの自動アサイン

```
public static void CreateInGameScene()
{
    PrefabGenerator.CreateAllUIPrefabs();
    Scene scene = EditorSceneManager.NewScene(
        NewSceneSetup.DefaultGameObjects,
        NewSceneMode.Single);
    var so = new SerializedObject(gameMgr);
    so.FindProperty("roundDatabase")
        .objectReferenceValue = roundDatabase;
    so.ApplyModifiedProperties();
    EditorSceneManager.SaveScene(
        scene, "Assets/Scenes/InGame.unity");
}
```

`static partial class` を使って機能ごとにファイルを分割し、可読性を維持しています。人間の役割は Copilot の完了報告を確認し、メニューから実行ボタンを押すだけになりました。

Copilot が生成したエディタ拡張スクリプトの品質については、想定した UI と異なる出力になることもありましたが、再度指示を出せば修正できる程度の問題にとどまりました。これにより、人間が Unity エディタ上で手作業を行う必要がほぼなくなり、Copilot の作業速度を活かせる体制が整いました。

4.7 特定のゲーム状況を再現するセットアップツールの実装

ゲームの動作確認をする際に、毎回ランダムな状況が生成されると、いつまでたっても検証が終わりません。特定の状況を意図的に再現できるデバッグツールは必須です。そこで、ゲーム開始時の状態を指定できるセットアップツールを Copilot に作成させました。

セットアップツールの設定は ScriptableObject（`DefaultGameConfig`）で管理しています。Inspector から以下のようなパラメータを指定できます。

表 4.1: セットアップツールの主な設定項目

パラメータ	説明
gameMode	ゲームモードの選択
cpuPlayerCount	CPU 数（0〜3）
cpuDifficulty	Easy / Normal / Hard / Expert
debugStartRound	開始ラウンド番号
disableTurnOrderShuffle	手番順の固定

さらに、`DebugPlayerSettings` というネスト構造で、プレイヤーごとのステータスなどを細かく指定できます。これにより、「3 ラウンド目で特定の建物を持っている状態」といったピンポイントな状況を再現することが可能です。

今回のゲームはイベント駆動で設計しており、すべての進行が決まった挙動に従います。リアルタイム操作によるランダム性がないため、セットアップ時の初期状態さえ指定すれば、その後の挙動は決定的に再現できます。CPU の行動も期待値計算がメインであり、乱数の使用が少量に限られるため、この方式で十分な再現性が確保できました。

4.8 完全イベント駆動のゲーム設計

題材としたボードゲームがターン制であるため、リアルタイムに状態を進行・監視する複雑さは必要ありません。この特性を活かし、ゲーム設計には完全なイベント駆動モデルを採用しました。

具体的には、以下のルールを徹底しています。

- Start や Awake で他のオブジェクトを参照しない
- Update 関数でゲームの状態を変更しない
- ゲームの進行はすべてコールバック経由で次のステップへ遷移させる

ゲーム進行の制御は C#の event と delegate を中心に構築しています。フェーズ変更・ラウンド変更・ターン変更はすべてイベントとして発火され、UI 側がそれを購読して描画を更新する疎結合設計です。

リスト 4.6: イベント定義とフェーズ変更の発火

```
// GameManager.cs - イベント定義
public event GamePhaseChangedHandler OnGamePhaseChanged;
public event RoundChangedHandler    OnRoundChanged;
public event TurnChangedHandler     OnTurnChanged;
public event GameEndedHandler       OnGameEnded;

// フェーズ変更時にイベント発火
public void ChangePhase(GamePhase newPhase)
{
    currentPhase.Value = newPhase;
    ChangePhaseClientRpc(newPhase,
        GetPhaseNameJapanese(newPhase));
    OnGamePhaseChanged?.Invoke(newPhase);
}
```

Start や Awake を使わない初期化については、コルーチンを用いた明示的な依存順制御で対応しています。各マネージャーのスポーン完了を while ループで待機してからゲームを開始する方式です。

リスト 4.7: マネージャー初期化の待機パターン

```
private IEnumerator InitializeGameSequence()
{
    while (GameManager.Instance == null
        || !GameManager.Instance.IsSpawned)
        yield return null;
    while (CardManager.Instance == null
        || !CardManager.Instance.IsSpawned)
        yield return null;
    while (BuildingManager.Instance == null
        || !BuildingManager.Instance.IsSpawned)
        yield return null;

    GameManager.Instance.StartGame();
}
```

この方針により、ゲームの状態遷移が明示的になり、Copilot に対する指示もシンプルに保てました。「どのイベントが発生したら何をするか」を伝えるだけで、Copilot が適切な実装を行えるためです。また、デバッグの際にも処理の流れを追いやすくなるという副次的なメリットがありました。

ただし、Copilot がこのルールに完全に従い続けるわけではありません。何度かルールに違反するコードが生成されることがありました。その場合は違反に気づいた時点で指摘し、修正を依頼しました。また、ある程度の実装がひと段落つくたびに、ルール違反がないかを Copilot に探させるようにしていました。

4.9 複数のゲームモード構成

題材にしたボードゲームには 3 つのゲームモードが存在しており、それらすべてを遊べるように準備を進めていました。しかし、各モードのデータ入力が間に合わず、ゲームジャムへの提出時点では 1 モードのみの対応にとどまりました。

複数モードに対応するにあたっては、共通のシステムを前提としたデータ構造を設計しました。ゲームの基本的な進行ロジックは共通化し、カードごとに利用可能なモードを `List<GameMode>` で管理することで、モードに応じたカードの出し分けをデータだけで制御できるようにしています。そのため、データさえ揃えばモードの追加は可能な状態になっていますが、限られた期間の中での優先順位判断として、まずは 1 モードの完成度を高めることを選択しました。

4.10　スクリプタブルオブジェクトを使ったゲームデータ設計

ゲームのデータ管理には ScriptableObject[*3]を採用しました。

当初、コードベースでデータを管理する方法も検討しましたが、この方式には Copilot との協業において問題がありました。AI の操作と人間の操作の両方でスクリプトファイルが変更されるため、Unity のコンパイルが頻繁に走り、大きな時間のロスが発生します。本プロジェクトでは人間がコードを直接編集する場面をそもそも想定していなかったため、この問題は顕著でした。

テキストベースのデータ管理と ScriptableObject を比較検討した結果、コード側からデータ定義をカスタムでき、Unity エディタ上での編集も容易な ScriptableObject を採用しました。たとえばカード 1 枚のデータは以下のような構造で定義しています。

リスト 4.8: カードデータの ScriptableObject 定義

```
[CreateAssetMenu(menuName = "MyGame/CardData")]
public class CardData : ScriptableObject
{
    public string cardId;
    public string cardName;
    public CardType cardType;
    public List<GameMode> availableInModes;
    public int buildCost;
    public int sellPrice;
    public MaxWorkerCapacity maxWorkers;
    public bool isShared;
    public List<CardEffect> effects;
}
```

カード 1 枚が 1 つの ScriptableObject ファイルに対応し、`CardDatabase` が実行時に `Resources.LoadAll<CardData>()` で全件を収集する仕組みです。データの持ち方や構成は人間が設計し、実際の ScriptableObject の作成やデータの入力作業は Copilot に委ねる形で分業しています。

[*3] Unity が提供するデータコンテナの仕組み。スクリプトから独立したアセットとしてデータを保持できるため、データの変更がコンパイルを引き起こさない

4.11 Unity のログを自動で読み取る仕組み

Copilot にゲームの不具合を修正させるには、エラー情報を正確に伝える必要があります。しかし、Unity のコンソールに表示されるエラーをひとつひとつ人間が読み取って Copilot に伝えるのは、手間がかかります。

MCP[*4]を使えばこの問題を解決できる可能性はありましたが、今回は環境セットアップに時間をかけたくなかったため、よりシンプルな方法を採用しました。Unity の `Application.logMessageReceived` を利用し、すべてのログをプロジェクト内の Markdown ファイルに自動で書き出す `GameLogger` を実装したのです。

リスト 4.9: GameLogger によるログ自動書き出し

```
public static class GameLogger
{
    private static readonly string LogFilePath =
        Path.Combine(Application.dataPath, "UNITY_CONSOLE_PLAY_LOG.md");

    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    private static void Initialize()
    {
        File.WriteAllText(LogFilePath, "");
        Application.logMessageReceived += OnLogReceived;
    }

    private static void OnLogReceived(
        string logString, string stackTrace, LogType type)
    {
        string prefix = type switch {
            LogType.Error => "[ERROR]", LogType.Warning => "[WARNING]",
            _ => "[LOG]" };
        WriteLog($"{prefix} {logString}");
    }
}
```

`[RuntimeInitializeOnLoadMethod]` 属性により Play 開始直後に自動で初期化され、セッション中のすべてのログがファイルに蓄積されます。Copilot はこのファイルを読み取ることで、エラーの検知や動作不備の発見を自律的に行えるようになりました。

この仕組みにより、人間がエラー内容を転記する手間がなくなり、Copilot が自らログを確認して問題を特定・修正するワークフローが実現しました。

[*4] Model Context Protocol。AI アシスタントが外部のツールやデータソースと連携するためのプロトコル

4.12 Copilot 主導の開発で感じたこと

最後に、本プロジェクトを通じて感じた、Copilot 主導で開発を進めることのメリットとデメリットをまとめます。

メリット

- **指示を出すだけでゲームが形になる**：設計方針やタスクの内容を言語化して伝えるだけで、Copilot が実装を進めてくれます
- **未知の技術もカバーできる**：自分が詳しくない領域でも、やりたいことを伝えれば実装してくれるため、学習コストを抑えながら開発を進められます
- **他作業との並行が容易**：定期的に画面を確認して指示を出すだけなので、人間側は他の作業と並行して進められます
- **参考資料の理解が速い**：サンプルプロジェクトや公式ドキュメントを提示すると、Copilot がすぐに内容を理解して実装に取り込んでくれます

デメリット

- **不要な防御的コードの追加**：try-catch や後方互換性のための処理など、今回のプロジェクトでは不要なコードを勝手に追加し、コードの見通しが悪くなることがあります
- **情報が少ない技術への対応が弱い**：ドキュメントや事例が少ない機能については、なかなか正しい実装にたどり着けないことがあります
- **誤ったエラー原因の提示**：エラーの原因を見当違いの箇所に求めることがあり、修正が遠回りになるケースがあります
- **指示の精度が求められる**：Copilot はプロジェクト全体を完全に把握しているわけではないため、適切な指示を行わないと仕様書や既存実装を無視して新たなコードを生成し始めることがあります

4.13 おわりに

本章では、Copilot を主体とした Unity ゲーム開発の実践例を紹介しました。

Copilot との協業では、AI が作業しやすい設計を人間側が意識して作ることが重要です。イベント駆動による明確な状態遷移、インプット層の抽象化、ScriptableObject によるコードとデータの分離、ログファイルを介したエラー情報の共有など、いずれも Copilot が自律的に作業を進められるようにするための工夫でした。

手動で開発を行った場合、年末年始のすべての時間を注いでも間に合うかわからない規模のものが、2 週間の Copilot 主導開発で形になりました。

一方で、Copilot には苦手な領域や誤りもあるため、人間側が設計の舵取りや品質の確認を怠らないことも欠かせません。AI に任せる部分と人間が担う部分を明確にし、それぞれの強みを活かすことが、Copilot 主導の開発を成功させる鍵だと感じました。

第 5 章

Docker Compose + Ansible + Grafana Cloud による快適自宅サーバ管理

Yoshio HANAWA / @hnw

5.1 はじめに

読者の皆さんは、自宅サーバをどれくらい真面目に管理できているでしょうか。

筆者の自宅では 3 台の Raspberry Pi が動いているのですが、先日までは無法地帯でした。自作デーモンや Slack ボットを手動でデプロイしており、バージョン管理やバックアップの仕組みもありません。

こうした前時代的な運用で特に困るのが OS アップデートです。稼働中のサービスを壊してしまう恐怖から移行に踏み切れず、7 年前の OS をだましだまし使い続けていました。

先日、「さすがにこのままでは良くない！」と一念発起し、自宅サーバ運用を Docker Compose + Ansible + Grafana Cloud というモダンな環境に移行しました。

当初、筆者はこれらのツールを「個人の自宅環境にはオーバースペックだ」と捉えていました。企業の大規模なインフラ管理ならともかく、構成がバラバラな個人の自宅サーバに導入しても、学習コストが高い割にメリットが小さいと考えていたのです。しかし、実際に運用を始めてみると、自宅サーバ運用もこなせる現実的な選択肢だとわかりました。

本稿では、この 3 つのツールを自宅でどう活用したか順に紹介します。

5.2 Docker Compose：設定とデータを一元化するコンテナ管理基盤

Docker と Docker Compose は開発環境や業務システムでは定番ですが、自宅サーバで必要かは意見が分かれるツールでしょう。筆者自身、自宅サーバをコンテナで管理するメリットをあまり感じていませんでした。チームでの環境共有や大規模システムのデプロイには確かに有効ですが、管理者が 1 人しかいない自宅サーバなら、OS 直下にツールを入れて、限界が来たらクリーンインストールする運用でも何とか回ってしまうからです。

ところが、実際に Raspberry Pi 上で環境を作ってみると、Docker が提供する「コンテナによる環境分離」の機能よりも、Docker Compose の「構成情報をひとまとめにできる」特性が自分にとっては重要だとわかりました。

「設定とデータが 1 つのディレクトリに収まる」ことの価値

自宅サーバ運用で厄介なのは、「このアプリがどこに何を置いているか」が数年後に分からなくなることです。

OS に直接ソフトウェアを入れると、設定ファイルやデータはシステム全体に散らばります。その状態が長く続くと、「ここを直したら別の何かが壊れるのではないか」という不安がつきまとい、別のサーバに引っ越すときの移行漏れが起こりやすくなります。

Docker Compose を導入すると、コンテナの設定は `compose.yaml` に、設定ファイルや永続データは指定したディレクトリに集約できます。筆者は `/opt/stacks/<アプリ名>/` を単位にしており、「このアプリの影響範囲はこのディレクトリだけ」と判断できるようになりました。

この状態には、運用上かなり大きな意味があります。

- 引っ越しが簡単になる: アプリを別のマシンへ移すときは、ディレクトリを丸ごとコピーして起動すればよい。
- メンテ対象が明確になる: 動作中アプリの設定ファイルやデータの場所がすぐに分かる。
- 自動化しやすくなる: デプロイ作業が「特定ディレクトリ以下を単純コピー」になり、後述する Ansible で扱いやすくなる。

注意点: 自作アプリのコンテナ化はやはり重い

一方、移行作業でもっとも手がかかったのは自作アプリのコンテナ化でした。

たとえば Go で書いた Slack ボットやセンサー読み取りデーモンなら、OS 直下にシングルバイナリを置いて動かす方がはるかに簡単です。これをコンテナ化しようとすると、それまで考えなくてよかったポイントで悩むことになります。

- 設定の受け渡し: `//go:embed` でバイナリに同梱していた設定を、環境変数から読むように変更する必要がある。
- 最小ベースイメージへの対応: `distroless` などを使うと、`os/exec` で呼んでいた外部コマンドがそのままでは使えない。
- ハードウェアアクセス: I2C や BLE を使う場合は、`devices` のマウントや権限設定を考える必要がある。

このようにアプリごとに直す箇所が違うため、率直に言って面倒です。

ただし、この種の定型作業は生成 AI との相性がよい領域でもあります。`Dockerfile` や `compose.yaml` のたたき台を作らせ、GitHub Actions による CI/CD と組み合わせれば、「コードを push するとイメージが自動でビルドされる」状態まで比較的素直に持っていけます。初期投資は必要ですが、長く運用するつもりなら十分に投資回収できると感じました。

■コラム: Raspberry Pi 3 のコンテナ環境は意外と実用的

今回の発見の 1 つは、Raspberry Pi 3 でもコンテナ環境が実用に耐えることです。当初は「メモリ 1GB の Pi 3 で Docker は厳しいだろう」と思っていたのですが、今のところトラブルなく動作しています。

筆者は cloudflared（Cloudflare トンネル）や Dockge（管理 UI）のビルド済みコンテナを利用していますが、快適に利用できています。

ただし、コンテナを動かすなら 64bit 版の OS を選ぶことを強くお勧めします。最近は 32bit 版を提供しないコンテナイメージも増えているため、64bit SoC を積んだ Pi 3 なら 64bit OS を入れておく方が扱いやすくなります。

5.3 Ansible：大規模環境でなくても便利に使える IaC ツール

Ansible は IaC（Infrastructure as Code）を実現する定番ツールで、複数のサーバに同じ設定を冪等に適用できるのが強みです。ただ、自宅サーバに持ち込むと途端に投資対効果が怪しくなります。筆者が特に気になったのは、次の 2 点でした。

1 つ目は動作の遅さです。Ansible は SSH 接続のたびに Python を起動して処理するため、シェルスクリプトより体感的に遅く、試行錯誤の多い個人サーバ環境ではストレスを感じることがあります。

2 つ目は記法の冗長さです。たとえば自作デーモンを動かす際、シェルスクリプトなら次のように設定ファイルをコピーして起動コマンドを叩くだけです。

```
cp my-daemon.service /etc/systemd/system/
systemctl enable --now my-daemon
```

しかし Ansible の場合、リスト 5.1 のようにファイル配置と systemd の有効化を組み込みモジュールの仕様にしたがって記述しなければなりません。

リスト 5.1: 冗長な Playbook の例

```
- name: Copy systemd service file
  ansible.builtin.copy:
    src: my-daemon.service
    dest: /etc/systemd/system/my-daemon.service
    mode: '0644'

- name: Enable and start my-daemon
  ansible.builtin.systemd:
    daemon_reload: true
    name: my-daemon
    enabled: true
    state: started
```

シェルスクリプトなら何も見ずに書ける処理でも、Ansible ではモジュールの仕様を都度確認する必要があり、慣れないうちは小さな変更でも手間がかかります。また、行数も数倍になってしまいます。

「少量の設定を管理したいだけなら、シェルスクリプトの方がまし」というのが以前の筆者

の印象でした。この違和感を減らし、自宅でも運用し続けられる形にするため、次の 4 つの指針に基づいて Ansible 対応を進めました。

指針 1: 定型作業は Ansible Galaxy に任せる

Ansible の冗長さがもっとも気になるのは、Docker などのミドルウェアをインストールする定型作業です。シェルスクリプトなら数行で済む処理でも、Ansible で正しく書こうとすると、GPG 鍵の追加やリポジトリ登録をモジュールごとに丁寧に記述することになります。

しかも、こうしたコードは一度書けば終わりではありません。数年後に OS を入れ替えると、パッケージ配布元や鍵の登録方法が変わっていて、そのままでは動かないことが珍しくありません。

生成 AI に最新手順で書き直させることもできますが、数ヶ月から数年に一度しか触らない自宅サーバ運用では、最終的にレビューするのは自分自身です。そこに毎回認知負荷をかけるより、定型作業は最初からコミュニティの資産に乗る方が楽です。

そこで筆者は、Ansible Galaxy [*1] に公開されているロールを積極的に使うことにしました。

第三者のロールをそのまま実行することには当然リスクがあります。ただ、たとえば `geerlingguy.docker` のように広く使われているロールなら、多数の利用者による検証と継続的なメンテナンスが期待できます。個人と AI の 1 対 1 で毎回レビューするより、こちらの方が長期運用では安心だと感じました。

指針 2: ホストの構成情報を 1 つのファイルに集約する

Ansible を運用していると、「どのホストに何のロール（機能）を持たせているか」「その際、どの変数を上書きしているか」といった構成情報が、`site.yml` や `group_vars` などあちこちに散らばりがちです。数ヶ月に一度しかインフラを触らない自宅サーバ運用では、この全容を思い出すだけで一苦労です。

そこで筆者は、「このホストに何のロールを持たせ、どう設定するか」というマクロな構成情報を、すべてホストごとの変数ファイル（`host_vars/<ホスト名>/main.yml`）ひとつに集約する方針をとりました。

具体的には、筆者が日常的に編集するファイルは リスト 5.2 のような純粋な「データの配列」になっています。

[*1] Ansible のロールやコレクションを公開・共有するプラットフォーム。 https://galaxy.ansible.com/

リスト 5.2: ホストごとの設定ファイルの記述例（ロール部分）

```
# 1. ホストに適用するロール
apply_roles:
  - raspberrypi
  - common
  - name: grafana.grafana.alloy
    become: true
  - name: geerlingguy.docker
    become: true
  - docker_apps
```

このシンプルな配列だけでインフラを構成できるよう、プログラム的な処理はすべて「基盤コード」に押し込んでいます。筆者の環境の `site.yml` は リスト 5.3 のように変数 `apply_roles` のすべてのロールを動的に読み込む仕組みになっています。

リスト 5.3: ロールの動的読み込み基盤

```
- hosts: all
  tasks:
    - name: Apply roles dynamically
      ansible.builtin.include_role:
        name: "{{ role_item.name | default(role_item) }}"
        apply:
          become: "{{ role_item.become | default(false) }}"
      loop: "{{ apply_roles | default([]) }}"
      loop_control:
        loop_var: role_item
      tags:
        - always
```

特定のホストにコミュニティのロールを追加する際も、既存のロールを割り当てる際も、リスト 5.2 の `apply_roles` 配列に 1 行書き足すだけです。複雑さを基盤に寄せ、「1 つのファイルを見れば今のホストの状態がすべて分かる」状態にしたことで、運用時の認知的負荷が大きく下がりました。

指針 3: コンテナ追加を「データとファイルの配置」だけで完結させる

インフラ構成は 1 ファイルにまとまりましたが、Ansible にはもう 1 つ越えるべき壁があります。それがタスク記述における複雑さです。

Ansible で新しい機能を追加する場合、リスト 5.1 のような冗長な記述が必要になったり、条件分岐（when）や実行結果の保持（register）どを駆使してタスクを書くことになります。筆者はこれを「YAML でのプログラミング」と呼んでいますが、この独特な文法をたまにしか触らない環境で都度思い出すのは、モチベーションを下げる原因になります。

新たなアプリを簡単に導入したいけれども、「YAML プログラミング」はやりたくありません。そこで考えたのが、コンテナに特化した汎用処理です。ここで、前章で紹介した Docker Compose によるディレクトリ集約の考え方が活きてきます。

アプリが Docker Compose によってディレクトリ単位にまとまっていれば、デプロイ作業は「特定ディレクトリを丸ごとコピーしてコンテナを起動する」という単一の定型処理に落とし込めます。つまり、全コンテナ共通の Ansible タスクを組むことができます。

そこで筆者は、リスト 5.4 のように変数ファイルに docker_apps というコンテナ専用の配列を定義し、アプリ追加時の Ansible タスクの記述を不要にしました。

リスト 5.4: ホストごとの設定ファイルの記述例（コンテナ部分）

```
# 2. ホストで動かすコンテナ
docker_apps:
  - cadvisor
  - dockge
  - switchbot-actions # ← 新しいコンテナを追加するときは、名前を書くだけ
```

新しいアプリをデプロイする際、筆者は Ansible のタスクを 1 行も書きません。compose.yaml や .env といった設定ファイルをコンテナ用のディレクトリに作成し、上記の docker_apps 配列にアプリ名を 1 行書き足すだけです。

あとは自作の共通処理（_app_loop.yml）が、ディレクトリを自動で探索し、見つかったファイルをサーバに配置し、コンテナを再起動してくれます。

「Docker Compose で管理単位を揃え、Ansible 側は配列へのデータ追加とファイル配置だけで済ませる」。この 2 つのツールの組み合わせによる単純化こそ、自宅サーバで Ansible 運用を続ける最大のコツだと筆者は考えています。

指針 4: 部分実行でストレスを軽減する

残念ながら、 Ansible は実行時間の観点で遅いツールです。1 行変更するたびに Playbook 全体を流して数分待つのは、やはりつらいものがあります。

そこで、今回のプロジェクトではロール単位・コンテナ単位で部分実行できるようにしました。Ansible の -t（--tags）オプションを使えば、必要なタスクだけを狙い撃ちできます。

```
# docker_apps ロール（コンテナ群）だけ実行
ansible-playbook site.yml -t docker_apps

# 1 つのコンテナ（switchbot-actions）だけ更新
ansible-playbook site.yml -t switchbot-actions
```

試行錯誤しているときでも、対象コンテナの設定変更を 20 秒程度で再適用できれば十分実用になります。

ここで効いてくるのが、前節の基盤処理です。ロール名やコンテナ名に対応するタグを動的に付けているため、新たなロールやコンテナを追加した際も自動的に適用されます。「複雑さを基盤に寄せる」設計が、待ち時間の短縮にも活きています。

■コラム: IaC のハードルを下げる秘密情報管理：SOPS + Touch ID

Ansible によるサーバ設定では外部サービスのトークンや鍵を扱うことが多いですが、平文のまま Git 管理するわけにはいきません。そこで筆者は `community.sops` ロールを使い、SOPS + age で秘密情報管理をすることにしました [*2]。

SOPS は YAML や JSON のキーを残したまま値だけを暗号化できるツールで、 age はそのバックエンドになる暗号化ツールです。

age の大きな魅力はプラグイン機構です。`age-plugin-se` を使うと、macOS の Secure Enclave を鍵として利用できます。これにより、Touch ID をタップするだけで暗号化ファイルを復号できます。パスワードを覚える必要がなく、鍵が Mac 本体の外に出ることもないため、セキュリティと利便性を両立できます。

Ansible からは、SOPS で暗号化したファイルを `community.sops.load_vars` で直接読み込めます。このタスクを冒頭で一度実行しておけば、その後のロール全体で復号済みの値を変数として使えます。

```
- name: Load SOPS secrets
  community.sops.load_vars:
    file: vars/secrets.sops.yml
  tags: always
```

Touch ID だけで復号できる体験は想像以上に快適です。暗号化の存在を強く意識せずに秘密情報を運用できるので、デプロイ時の心理的ハードルが下がると感じています。

5.4 Grafana Cloud：自宅サーバ・自作アプリでこそ活きる可視化基盤

Grafana Cloud は Grafana Labs が提供するクラウド型のオブザーバビリティプラットフォームです。メトリクス（Prometheus 互換）、ログ（Loki）、トレース（Tempo）をまとめて扱えます。海外の SRE・インフラ界隈では広く使われています。

採用のきっかけは単純で、無料枠が大きかったからです。

Grafana Cloud の Free Tier では、ログが月 50GB、メトリクスは 1 万系列、トレースは月 50GB まで使えます [*3]。自宅の数台の Raspberry Pi からログとメトリクスを送り続けても、無料枠を超えることはまずありません。

無料枠目当ての消極的な選定でしたが、使ってみて多くの強みをもつサービスだと気づきました。以下では、3 つの魅力を紹介します。

魅力 1: Integration が最初の壁を下げてくれる

Grafana Cloud の大きな利点は、公式 Integration が充実していることです。Linux、Docker、各種データベースなどに対応した Integration が用意されており、「どのメトリクスを集め、どのダッシュボードで見ればよいか」というノウハウが最初からパッケージ化されています。

筆者は Linux server integration と Docker integration を使っており、インストール手順に従うだけで Grafana Alloy の設定とダッシュボードをそのままセットアップできました。生成される Alloy 設定には、監視に必要なメトリクスだけを絞り込むフィルタリングも含まれています。下記はその抜粋です。

リスト 5.5: Linux server integration で提供される Alloy 設定（抜粋）

```
prometheus.relabel "integrations_node_exporter" {
  forward_to = [prometheus.remote_write.metrics_service.receiver]

  rule {
    source_labels = ["__name__"]
    // CPU・メモリ・ディスク・ネットワーク等、必要なメトリクスが 100 件以上列挙済み
```

[*2] Ansible には標準で「Ansible Vault」という暗号化機能がありますが、本稿ではより便利な SOPS を紹介します

[*3] ただし Free Tier のデータ保存期間は 14 日間です。無料枠の詳細は変更される可能性があるため、最新情報は公式サイトを確認してください

```
    regex  = "up|node_cpu_seconds_total|node_memory_MemAvailable_bytes|node_disk→
_io_time_seconds_total|..."
    action = "keep"
  }
}
```

「何を収集すべきか」を一から調べる必要がなく、無料枠に収まるよう絞り込まれた設定がそのまま手に入るのは大きな利点です。また、 図 5.1 のようなダッシュボードが最初から提供されるため、自分でゼロから作る必要がありません。

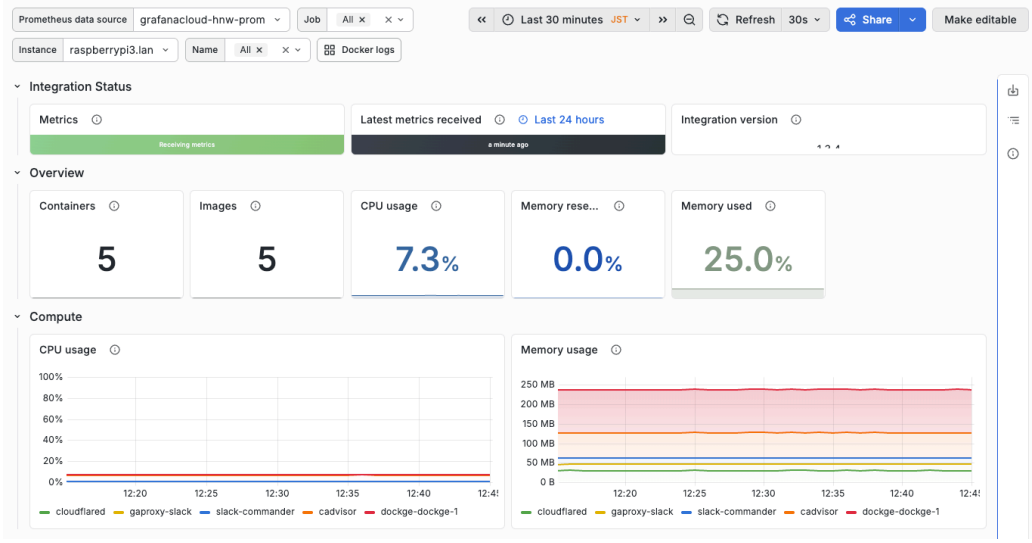


図 5.1: 筆者の利用している Docker integration のダッシュボード

魅力 2: 自作アプリのパフォーマンスを可視化できる

Grafana が役に立つのは、システム標準のメトリクス監視だけではありません。自作プログラムに計測コードを仕込むと、「なんとなく遅い」という感覚を数字で議論できるようになります。

筆者は、洗面所の SwitchBot 人感センサーが反応するとスマート電球を点灯させる仕組みを自宅で動かしています（図 5.2）。便利なのですが、使い続けるうちに点灯の遅さがストレスになってきました。

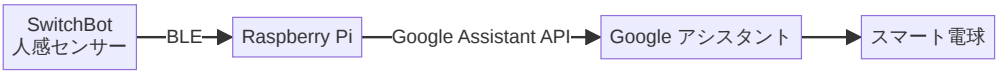


図 5.2: 筆者自作の洗面所の電気をつけるシステム

当初は、BLE と Google Assistant API のどちらがボトルネックなのか分かりませんでした。そこで各処理の所要時間をメトリクスとして記録してみたところ [*4]、BLE では時々取りこぼしが起き [*5]、Google Assistant API は想像以上に遅いことが分かりました。これをきっかけにそれぞれのボトルネックを改善し、実用的な点灯速度まで改善できました。

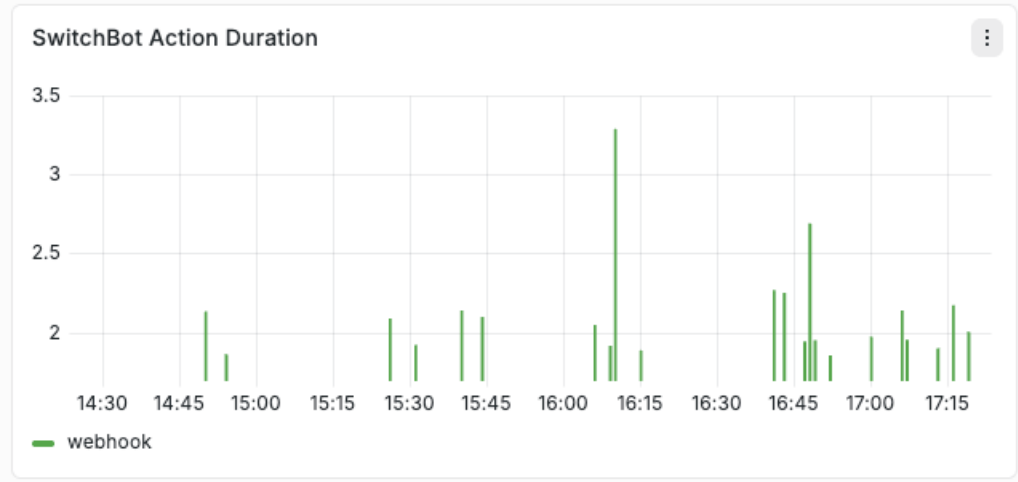


図 5.3: Google Assistant API の実行時間を可視化したもの

「なんとなく遅い」をその場で計測・可視化できる環境があると、改善の打ち手が一気に具体的になります。本稿では紹介しませんでしたが、トレースの可視化もかなり強力です。個人環境でのオブザーバビリティ基盤の活用事例はまだ多くない印象ですが、自宅サーバでも十分に元が取れると感じました。

[*4] 本来ならトレースを取るべき状況でしたが、作業時点ではトレース収集の設定をしていませんでした

[*5] BLE チップの世代を上げれば安定性を改善できるのではないかと考え、Pi 3 から Pi 4 にコンテナごと引っ越しました。実際に安定性が改善しました。

魅力 3: デファクトスタンダードなので AI に質問しやすい

Grafana や Prometheus は高機能なぶん、最初の学習コストは低くありません。ダッシュボードの設定項目は多く、メトリクスの可視化に使うクエリ言語の PromQL も独特です。

ただし、これらのツールが世界中で使われているデファクトスタンダードであることは大きな強みです。たとえば「このグラフの 2 系統だけを積み上げ表示したい」と AI に聞けば、ダッシュボードの調整方法がかなりの精度で返ってきます。PromQL のたたき台もすぐ得られます。自前の監視スクリプトやマイナーなツールでは、この恩恵は受けにくいでしょう。

また、自作アプリでメトリクスを出したいと思ったときも、各言語の Prometheus exporter ライブラリが充実しており、サンプル実装もネット上にたくさん見つかります。標準的な基盤を選ぶほど、実装と運用の両面で AI の力を引き出しやすくなる、という意味でも Grafana Cloud は扱いやすい選択肢だといえるでしょう。

5.5 おわりに

自宅サーバ運用に必要だったのは、企業向けのノウハウをそのまま持ち込むことではありませんでした。 Ansible と Docker Compose を組み合わせることで「できるだけ Ansible タスクを書かない」ことが自宅で Ansible を運用し続けるコツだと筆者は考えています。

今回導入した 3 つのツールはすべて想像以上に有用でした。特に大きかったのは、「OS 更新やマシン引っ越しが怖くなくなったこと」と「自宅サーバだけでなく、自作アプリケーションの問題も深掘りできること」です。自宅サーバは仕事のインフラほど厳密でなくても回りますが、再現性と可観測性が少し入るだけで、運用の気楽さは大きく変わります。

なお、本稿で紹介した Ansible のリポジトリは GitHub で公開しています [*6]。具体的な設定の参考にしてください。

[*6] https://github.com/hnw/home-ansible/

第 6 章

ルータ沼にはまってきた —— J4125 と OpenWrt で挑むトリプルスタック構築記

yamada-masa

6.1　発端：1000 円の検証機から始まった「ルータ沼」

「安定した高速インターネットの実現と、自宅サーバの公開」。これが今回の沼への入口でした。

やりたいことはシンプルに次の二つでした。

1. **PPPoE（固定 IP）でサーバを外に公開すること**
2. **DS-Lite（MAP-E 系 / transix 経由）で日常の通信を高速化すること**

有名なのは YAMAHA のルーターで、中古でも余裕でこれができると言われています。ただし中古でも RTX830 で 2 万円程度と「それなりの値段」です。そこで私が思ったのは「自前で同じことができないか」でした。ネットをうろうろしていると、OpenWrt というファームウェアを使えば PPPoE と DS-Lite を同一ルータで共存させられると分かりました。同僚が VyOS で自宅ルータを組んでいるという話も聞いていたし、なんとなくいけそうな気がしました。

まず手元の古いルータ `WN-AX1167GR` に試してみました。ファームウェアを OpenWrt の `factory.bin` に書き換えたあと再起動を待ちましたが、起動してくれることはありませんでした。やり直そうと、元のファームウェアを復旧しようと試みました。WN-AX1167GR がネッ

トワーク内の TFTP サーバからファームを取得して起動する挙動を利用して、ノート PC に `tftpd64` を入れて直結してみたのですが、ノート PC 側にアクセスが来た形跡すらなかったのでした。見事に文鎮化しました。早速一回目の挫折でした。

気を取り直して二回目。今度は生成 AI と壁打ちしながら機種選定を進めました。TP-Link の ArcherC7 が候補に挙がりましたが、同じ型番でもハードウェアバージョンで全く別物という地雷があり、価格が折り合うものが見つかりませんでした。そこで目に留まったのが **Archer C6 V2** でした。オークションで **1000 円程度**で入手できました。

この 1000 円が、沼の底へ導く入場料となりました。

6.2 敗北：Archer C6 が教えてくれた「CPU の壁」

インストールと初期設定

C6 V2 へのインストールは拍子抜けするほど素直でした。OpenWrt のファームウェアセレクターから対象イメージを取得し[*1]——

純正ファームの Web コンソールから更新を実行してしばらく待つと、あっさり成功しました。無線もデフォルトで OFF になっていて、技適問題も問題なしと判断しました。

続いて最小限の初期設定を投入します。

リスト 6.1: 日本語化パッケージのインストール

```
# 日本語化
opkg update
opkg install luci-i18n-base-ja
opkg install luci-i18n-firewall-ja
opkg install luci-i18n-opkg-ja
```

`Network -> Interfaces -> wan` を PPPoE に変更して ID/PASS を入力、Save & Apply で接続完了です。この時点で WAN は PPPoE でネットにつながりました。

続いて DS-Lite を投入します。

リスト 6.2: DS-Lite のインストールと設定

```
opkg update
opkg install ds-lite
```

[*1] https://firmware-selector.openwrt.org/?version=23.05.5&target=ath79%2Fgeneric&id=tplink_archer-c6-v2

```
opkg install kmod-ipip
opkg install sit

uci set network.dslite='interface'
uci set network.dslite.proto='dslite'
uci set network.dslite.peeraddr='aftr.example.ne.jp'  # プロバイダから提供されるAFT→
Rアドレス
uci set network.dslite.mtu='1280'
uci set network.dslite.mru='1280'

uci set network.wan6='interface'
uci set network.wan6.proto='dhcpv6'
uci set network.wan6.ifname='@wan'
uci set network.dslite.metric='10'
uci set network.wan.metric='20'
uci commit network
/etc/init.d/network restart
```

AFTR アドレスは DS-Lite を契約すると自動で降ってきます。tcpdump で拾うとよいです（生成 AI 情報でしたがこれは当たっていました）。

ソフトウェアオフロードもかけておきます。

リスト 6.3: ソフトウェアオフロードの有効化（C6）

```
uci set firewall.@defaults[0].flow_offloading='1'
uci set firewall.@defaults[0].flow_offloading_hw='0'
uci commit firewall
/etc/init.d/firewall restart
```

スピードテストの結果

構成が整ったので、DS-Lite で繋がっている PC 側から Google のスピードテストを走らせました。

上り：約 200Mbps 下り約 150Mbps

既存ルータで上りが 100Mbps だったのが 200Mbps に改善したはよかったものの、下りが 150Mbps で頭打ちというのは結構な期待外れでした。

計測中の top コマンドを実行してみると、なんとなく答えがわかりました。

リスト 6.4: スピードテスト中の top コマンド出力

```
Mem: 47256K used, 74776K free, 1012K shrd, 0K buff, 20528K cached
CPU:   8% usr   0% sys   0% nic   0% idle   0% io   0% irq  91% sirq
Load average: 0.28 0.25 0.21 2/55 14971
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
   10     2 root     RW       0   0%  83% [ksoftirqd/0]
```

`ksoftirqd/0` が **83%** で張り付き、sirq が **91%** に達しています。生成 AI に見せると「DS-Lite のカプセル化処理（IPv4-over-IPv6）がソフトウェアで処理されており、CPU が完全に飽和している」という診断が返ってきました。

つまり 150Mbps という数字は「C6 の CPU が毎秒処理できる DS-Lite パケット量の物理的な上限」だったのでした。ハードウェアレベルで DS-Lite に対応していないとこうなるのだなと思い知りました。CPU って大事なんですね。

1000 円の価値

古いルータで速度を出そうというのが甘い考えでした。とはいえ C6 は勉強代として、1000 円の価値がありました。PPPoE と DS-Lite が共存できること、ボトルネックが CPU にあること——この 2 点がよく理解できたことです。

次のステップとしては「もっと CPU 性能の高い機体を用意する」という結論に至りました。そしてふと、タンスの肥やしになっている **J4125 のミニ PC** が目に入りました。

6.3 転換：J4125 への移行と「OpenWrt ビルド」の洗礼

ロースペックミニ PC がよみがえる

J4125 は Intel Celeron ファミリーの省電力 x86 SoC です。C6 のチップとは比べ物にならない処理能力があるはずで、「これならさすがにいけるだろう」と思って、OpenWrt の x86 版を入れてルータ化してみることにしました。

最初の壁：NIC がひとつもない

ルータにするためには WAN と LAN の少なくとも 2 ポートが必要です。ところがこの J4125 機、**オンボードの NIC がひとつもない**という構成でした。

そこで USB NIC を 2 本用意することにしました。調べたところ **rtl8153 チップ搭載の USB-LAN アダプタ**が OpenWrt でサポートされているとのことだったので、2 本調達しま

した。

「これで準備完了、あとは標準 ROM を書き込むだけ」——そう思っていました。

第二の壁：標準 ROM にドライバが入っていない

OpenWrt の x86 公式イメージをそのまま書き込んで起動してみると、USB NIC が**1 本も認識しない**という状態でした。

`dmesg` を確認しても rtl8152/8153 のドライバロードを示すメッセージが見当たりませんでした。x86 向けの標準ビルドには `kmod-usb-net-rtl8152` は含まれていないことがわかりました。

NIC がゼロの状態では SSH すらできず、キーボードとディスプレイで操作はできる状態でしたが、ネットワークにつながらないルータを前に途方にくれました。しかし当時の私は調子に乗っていたので大して調べもせずなんか OpenWRT を自分でビルドしたらいいんじゃないかという結論になりました。

WSL2 (Fedora) でのビルド環境構築

Windows 機の WSL2 で Fedora 環境を立ち上げてビルドすることにしました。OS は完全に著者の個人的好みなので、こだわりがなければ Ubuntu などの Linux をお勧めします。

リスト 6.5: WSL2 Fedora 環境のセットアップ

```
wsl --install -d FedoraLinux-43
sudo dnf update -y
sudo dnf install -y gcc gcc-c++ make automake autoconf \
    ncurses-devel zlib-devel gawk flex bison gettext \
    wget git rsync python3 patch perl perl-FindBin unzip
```

ソースを取得してブランチを切ります。

リスト 6.6: OpenWrt ソース取得とブランチ切り替え

```
git clone https://git.openwrt.org/openwrt/openwrt.git
cd openwrt

git branch -a | grep openwrt-23
git checkout -b openwrt-23.05 remotes/origin/openwrt-23.05

./scripts/feeds update -a
```

```
./scripts/feeds install -a
```

menuconfig でモジュールを選ぶ

`make menuconfig` を実行して、必要なカーネルモジュールを選んでいきます。
今回の構成で必須になったのは以下のとおりです。
まずはターゲット設定ですが、これは次のとおりです。

表 6.1: ターゲット設定

項目	設定値
Target System	x86
Subtarget	x86_64
Target Profile	Generic

USB NIC（rtl8153）に必要なモジュールは次のとおりです。

リスト 6.7: USB NIC（rtl8153）用モジュール

```
kmod-usb2
kmod-usb3
kmod-usb-net
kmod-usb-net-rtl8152   <- 8152 を選ぶと 8153 もサポートされます
```

DS-Lite 関連モジュールは次のとおりです。

リスト 6.8: DS-Lite 関連モジュール

```
kmod-ip6-tunnel
kmod-sit
kmod-ipip
```

`kmod-sit` と `kmod-ipip` はなくても動くケースもあったようですが、念のため入れておきました。LuCI（Web コンソール）も結局あると便利なので含めています。

ビルド実行

PATH の大文字小文字の区別で失敗するケースがあるので、PATH を上書きしてからビルドします。また、失敗時に原因をすぐ確認できるよう `V=s` を付けておくのがポイントです。ビルド時間は多少遅くなりますが、失敗したらデバッグ情報付加して再ビルドする羽目になるので最初からデバッグ情報を見れるようにしておくのが良いという結論になりました。

リスト 6.9: ビルド実行コマンド

```
PATH=/usr/local/bin:/usr/bin:/bin make -j$(nproc) V=s
```

初回ビルドは相当時間がかかりますが、成功すると以下のイメージが生成されます。

リスト 6.10: 生成される ROM イメージ名

```
openwrt-x86-64-generic-squashfs-combined-efi.img
```

これを Windows 側にコピーしておきます。

罠：make target/linux/clean を忘れるとハマる

menuconfig でカーネルモジュールの設定を**変更したあとに再ビルドする場合**、必ずこのコマンドを先に実行する必要があります。絶対です。

リスト 6.11: カーネルキャッシュのクリア

```
make target/linux/clean
```

これを忘れると、前回ビルドのキャッシュが残ったままで新しい設定がカーネルに反映されません。「rtl8152 を追加したはずなのにドライバが入っていない」という状態が出来上がります。エラーメッセージを元に生成 AI と相談したのですが、ハルシネーションにハルシネーションを重ねる事態になってしまい、原因特定にかなり時間がかかりました。`V=s` で詳細ログを出しながら追いかけてようやく原因に気づくことができました。

menuconfig でカーネルモジュールを変えたら、必ず `make target/linux/clean` してか

らビルドし直す——基本中の基本でした。C でビルドする界隈では当たり前ですよね、数十年ぶりで忘れていました。

x86 実機へのインストール

ROM はライブメディアでそのままインストールできないタイプなので、軽量 Linux から `dd` でインストールします。

1. **Alpine Linux** を rufus で USB メモリに焼く
2. **OpenWrt のイメージ**も別の USB メモリに rufus で焼く
3. Alpine を USB 起動して、root でパスワードなしログイン
4. `setup-keymap` で 106 キーボードを有効化
5. `fdisk -l` でインストール先のディスクとメディアを確認
6. `dd` でイメージを書き込む

リスト 6.12: dd によるインストール

```
mount /dev/sdb1 /mnt
dd if=/mnt/openwrt-x86-64-generic-squashfs-combined-efi.img \
   of=/dev/sda bs=4M conv=fsync
sync
```

再起動すると、今度こそ USB NIC が認識されました。`ip link` に `eth0` `eth1` が並んだとき、ちょいと感動すら覚えました。

6.4　実装：DS-Lite × PPPoE × IPv6 リレーの「全部盛り」

目指した構成の全体像

USB NIC が認識されたので、本番構成の実装が可能になりました。最終的に目指した構成は次のとおりです。やりたい事を全部詰め込んだ「全部盛り」です。

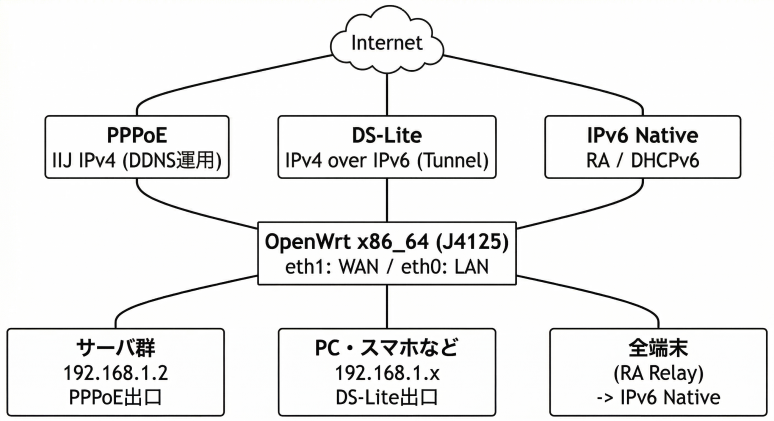


図 6.1: ネットワーク全体構成図

ポイントは経路の**使い分け**です。

- **一般端末（PC・スマホなど）** → DS-Lite 経由でインターネットへ（高速・低遅延）
- **自宅サーバ（192.168.1.2, 192.168.1.3）** → PPPoE 経由で固定 IP から出る（外部公開用）
- **IPv6** → HGW なし環境でリレー設定し、地域誤認を抑制する

DNS が宅内の一般端末を下記のような経路に誘導します 1. IPv6 のサーバがあるときは IPv6 でアクセス 2. IPv6 のサーバがないときは DS-Lite でアクセス

PPPoE と DS-Lite の基本設定

まずはネットワークインターフェースをまとめて設定します。

リスト 6.13: ネットワーク基本設定

```
uci batch << 'EOF'
set network.lan=device='br-lan'
set network.lan.proto='static'
set network.lan.ipaddr='192.168.1.1'
set network.lan.netmask='255.255.255.0'
set network.lan.ip6assign='60'

set network.wan=device='eth1'
set network.wan.proto='pppoe'
set network.wan.metric='20'
set network.wan.username='XXXXXXX'
```

```
set network.wan.password='XXXXXXX'

set network.dslite=interface
set network.dslite.proto='dslite'
set network.dslite.peeraddr='XXXX:XXXX::XXXX:XXX'
set network.dslite.mtu='1280'
set network.dslite.metric='10'
set network.dslite.encaplimit='ignore'

set network.wan6=interface
set network.wan6.proto='dhcpv6'
set network.wan6.device='eth1'
set network.wan6.reqaddress='try'
set network.wan6.reqprefix='auto'
commit network
EOF

/etc/init.d/network restart
```

metric の値がルーティングの優先度を決めます。DS-Lite を 10、PPPoE を 20 にすることで、デフォルトでは DS-Lite 経由になります。サーバだけは後述のポリシールーティングで PPPoE 側に強制します。

ファイアウォールに DS-Lite ゾーンを追加します。

リスト 6.14: ファイアウォール DS-Lite ゾーン設定

```
uci batch << 'EOF'
add firewall.zone
set firewall.@zone[-1].name='dslite'
set firewall.@zone[-1].network='dslite'
set firewall.@zone[-1].input='ACCEPT'
set firewall.@zone[-1].output='ACCEPT'
set firewall.@zone[-1].forward='ACCEPT'
set firewall.@zone[-1].masq='1'
set firewall.@zone[-1].mtu_fix='1'

add firewall.forwarding
set firewall.@forwarding[-1].src='lan'
set firewall.@forwarding[-1].dest='dslite'

commit firewall
EOF

/etc/init.d/firewall restart
```

ポリシーベースルーティング：サーバだけ PPPoE から出す

自宅サーバを固定 IP（PPPoE 側）から外に公開するには、IP アドレスを見てルーティングテーブルを切り替える「ポリシーベースルーティング」が必要です。

まずルーティングテーブルに pppoe という名前のテーブルを追加します。

リスト 6.15: ルーティングテーブルへの pppoe 追加

```
echo "100 pppoe" >> /etc/iproute2/rt_tables
```

続いてサーバの経路に PPPoE を強制するルールを作成します。

リスト 6.16: ホットプラグ・スクリプトによるルーティング設定

```
cat << 'EOF' > /etc/hotplug.d/iface/99-pppoe-routing
[ "$ACTION" = "ifup" ] && [ "$INTERFACE" = "wan" ] || exit 0
ip route add default dev pppoe-wan table pppoe
ip rule add from 192.168.1.2 table pppoe
ip rule add from 192.168.1.3 table pppoe
EOF

# 実行権限を付与
chmod +x /etc/hotplug.d/iface/99-pppoe-routing
```

これで 192.168.1.2 と 192.168.1.3 からの送信パケットは自動的に pppoe テーブルを参照し、PPPoE インターフェース経由でインターネットに出ていくようになります。それ以外の宅内端末は metric の低い DS-Lite がデフォルト経路として使われます。

ポート開放（80/443）も追加します。

リスト 6.17: ポートフォワード設定 80/443

```
uci batch << 'EOF'
add firewall.redirect
set firewall.@redirect[-1].name='http80'
set firewall.@redirect[-1].src='wan'
set firewall.@redirect[-1].src_dport='80'
set firewall.@redirect[-1].dest='lan'
```

```
set firewall.@redirect[-1].dest_ip='192.168.1.2'
set firewall.@redirect[-1].dest_port='80'
set firewall.@redirect[-1].proto='tcp'

add firewall.redirect
set firewall.@redirect[-1].name='tls443'
set firewall.@redirect[-1].src='wan'
set firewall.@redirect[-1].src_dport='443'
set firewall.@redirect[-1].dest='lan'
set firewall.@redirect[-1].dest_ip='192.168.1.2'
set firewall.@redirect[-1].dest_port='443'
set firewall.@redirect[-1].proto='tcp'

commit firewall
EOF

/etc/init.d/firewall restart
```

IPv6 リレー：なぜ必要だったのか

DS-Lite はその仕組み上、**IPv4 通信がすべて同一グローバル IP を共有するトンネル**を通りますので、Geolocation サービスが「地域」を誤判定しやすいという問題があります。広告やアクセス元がはるか離れた地域のものになる「地域誤認」が発生しやすいです。よくあるのが居住地域と関係のない YouTube 広告が表示されてもやっとするという感じですね (大した問題ではないといえばそうです)。

これを抑制するために IPv6 での通信経路を確保します。IPv6 は経路として準備しておけば、上流の DNS が v4 より v6 を優先するので、宅内の経路もそれによって DS-Lite より v6 が優先になります。これを利用して地域誤認を抑制しようということです。IPv6 は個別にグローバルアドレスが割り当てられるので Geolocation 精度が高く、地域誤認を大幅に減らせます。

私のインターネット契約は **HGW（ホームゲートウェイ）なし、ONU 直結**です。HGW なし環境では/64 のプレフィックスしか配布されないため、LAN 側に IPv6 アドレスを配るには **RA リレー**が必要になります。この前提を元に IPv6 を開通させる設定を投入していきます。

リスト 6.18: IPv6 RA リレー設定

```
# wan6 側：マスターとして DHCPv6 を受け取る
uci set dhcp.wan6=dhcp
uci set dhcp.wan6.interface='wan6'
uci set dhcp.wan6.ignore='0'
uci set dhcp.wan6.master='1'

# LAN 側：RA・DHCPv6・NDP をリレーする
uci set dhcp.lan.ra='relay'
uci set dhcp.lan.dhcpv6='relay'
uci set dhcp.lan.ndp='relay'

uci commit dhcp
/etc/init.d/dnsmasq restart
```

これで宅内端末に IPv6 グローバルアドレスが降ってくるようになりました。

ソフトウェアオフロードと DNS の整備

パケット転送の高速化のため、ソフトウェアオフロードも有効にしておきます。

リスト 6.19: J4125 でのソフトウェアオフロード設定

```
uci set firewall.@defaults[0].flow_offloading='1'
uci commit firewall
/etc/init.d/firewall restart
```

宅内サーバへのアクセスをルータが正しく処理できるよう、ローカル DNS も整備します。単に v4 の DNS サーバになるだけでは名前解決が v6 優先で他の DNS サーバが優先になるため、このルータの DNS サーバを優先するように強制する設定も入れます。

リスト 6.20: ローカル DNS の設定

```
uci add_list dhcp.lan.dhcp_option="6,192.168.1.1"
uci commit dhcp
/etc/init.d/dnsmasq restart

# 独自ドメインを宅内サーバに向ける
uci add dhcp domain
uci set dhcp.@domain[-1].name='example.example.com'
```

```
uci set dhcp.@domain[-1].ip='192.168.1.2'
uci commit dhcp
/etc/init.d/dnsmasq restart

# このルータのDNSサーバを強制する設定
# --- 1. 配布する DNS アドレスを指定（IPv4 & IPv6）---
# クライアントに自分自身（OpenWrt）を DNS として教える
uci delete dhcp.lan.dns
uci add_list dhcp.lan.dns='192.168.1.1'
uci add_list dhcp.lan.dns="$(ip -6 addr show br-lan | awk '/inet6 fe80/ {print $→
2}' | cut -d/ -f1 | head -n 1)"

# --- 2. IPv6 / RA の配布権限を掌握する ---
# DNS広告はOpenWrtが生成し（ra_dns=1）、リレーではなくハイブリッドで行う
uci set dhcp.lan.dns_service='0'
uci set dhcp.lan.ra_dns='1'
uci set dhcp.lan.dhcpv6='hybrid'
uci set dhcp.lan.ra='hybrid'
uci set dhcp.lan.ndp='hybrid'
uci set dhcp.lan.ra_management='1'
uci set dhcp.lan.ra_preference='high'

# --- 3. 実行と保存 ---
uci commit dhcp
/etc/init.d/odhcpd restart
```

MTU の問題が発生したのでこれも対処します。

リスト 6.21: MTU 対応

```
# 大きなパケットが詰まる問題を解決
uci set firewall.@zone[1].mtu_fix='1'
uci delete firewall.@defaults[0].mtu_fix 2>/dev/null

uci commit firewall
/etc/init.d/firewall restart
```

これでネットワーク図に描いた「全部盛り」の構成が完成しました。

6.5　凱旋：圧倒的な数値と、残された課題

C6 vs J4125 ——性能の暴力

設定完了後、スピードテストを走らせました。結果を C6 当時と並べると一目瞭然です。

表 6.2: Archer C6 V2 と J4125 の性能比較

測定項目	Archer C6 V2	J4125
DS-Lite 下り	150 Mbps	600 Mbps
PPPoE 下り	—	400 Mbps
IPv6 下り	—	600 Mbps
CPU 使用率（計測中）	sirq 91% / ksoftirqd 83%	12%

C6 は DS-Lite 処理だけで CPU が完全飽和していたのに対し、J4125 は 600 Mbps を叩き出しながらも CPU 使用率が**わずか 12%** でした。正直オーバースペックという感じで、やりすぎたかな感がありました。

省電力モードに落としても十分すぎる性能だったため、BIOS の省電力設定を有効にして消費電力も抑えることにしました。

運用上のリアルな副作用

圧倒的な性能を手に入れたのはよかったのですが、移行後にいくつかの問題が出てきました。とはいえこれらは「性能を得た代償」としては軽微なものでした。ルータとしての本分——パケットを速く処理すること——において　J4125 は完璧でスキはないです。

問題 1：停電復旧時に自動起動しない

一般的な x86 マシンは停電からの復旧時に自動起動しない設定がデフォルトです。ルータが復旧していないせいでインターネットに繋がらない、という事態が発生しそうでした。BIOS の「AC Power Recovery」設定を「Last State」に変更することで対処しました。

問題 2：メッシュ Wi-Fi が使えなくなった

もともと使っていた親機はメッシュ Wi-Fi に対応していましたが、AP モードに切り替えるとメッシュ機能が無効になります。メッシュから通常の中継器へと格下げになりました。ですが体感でわかるほどの速度低下はまだないので「実害が出たら対応しよう」という気持ちで

いったん保留にしています。

6.6 結論：ルータの速度は「無線」ではなく「CPU」で決まる

あなたのルータが遅い理由、もしかしたら Wi-Fi 規格でも回線品質でもなく、**ルータ内蔵の CPU がもろもろ足りないだけ**かもしれません。

私もそんなことは考えたこともありませんでした。「Wi-Fi 6 対応の新しいルータを買えば速くなる」と信じていたし、「Ethernet の規格が Gigabit に対応していれば問題ない」と思い込んでいたのでした。ルータの CPU 性能やどんな通信方式にハードウェア対応しているかなんてパッケージや製品情報からほとんど読み取れないですよね。

ところが現実は全然違いました。パケットは NIC に届いてから、**CPU がソフトウェアでルーティング・カプセル化・ファイアウォール処理**をしたうえで別の NIC から送り出されます。この処理能力の差が、そのままスループットの差になって現れます。ハードウェアとして対応しているか、対応していない場合は CPU の性能が不足していないことが非常に重要でした。

Archer C6 のチップは DS-Lite のカプセル化処理だけで飽和し、150 Mbps が限界でした。一方 J4125 は同じ処理を CPU 使用率 12% でこなし、600 Mbps を超えました。使ったのは 1000 円の検証機とタンスの肥やしの J4125 という手段でしたが、結果はしっかり出せたと思います。

これからの展望

スマホの買い替えサイクルが来れば、そのうち我が家にも Wi-Fi 6E や Wi-Fi 7 のアクセスポイントが手元に来ることになります。無線側の帯域がさらに広がるわけですが、**ルータは J4125 のまま**で問題なさそうです。CPU 使用率 12% でしか動いていないルータに、まだ 88% の余白があります。有線 Gigabit の帯域を余裕で処理できる心臓がある限り、AP をどれだけ新しくしても足を引っ張ることはないはずです。とはいえさきに有線の速度が 1Gbps 超えてしまいそうですが。その際には 2.5Gbps までは NIC を新調することでおそらく速度を維持できるのではないかと考えています。

1000 円の検証機が教えてくれたことは「ルータ沼の本質はアンテナでも規格でもなく、パケットを処理する CPU 性能にある」ということでした。無線 AP は無線 AP で好きに選べばいいということですね。

沼にはまってみていろいろと勉強になりました。

6.7 付記：フルビルド不要だったという話

最後に、オチをひとつ添えておきます。

今回のビルド作業の目的は、突き詰めると「kmod-usb-net-rtl8152 をカーネルに組み込むこと」だけでした。後から調べると、**必ずしも OpenWrt をフルビルドする必要はなかった**ということが分かりました。

OpenWrt には.ipk 形式のパッケージが用意されており、カーネルモジュール単体もこの形式で配布されています。つまり、以下のような手順でも解決できたのでした。

1. 別の USB メモリに対象バージョンの kmod-usb-net-rtl8152_*.ipk（および依存パッケージ）をあらかじめコピーしておく
2. OpenWrt 起動後に USB メモリをマウントして opkg install でパッケージを手動インストールする
3. NIC が認識されれば、あとは通常の設定作業に進める

ビルド環境の構築から初回コンパイルまで数時間以上を要したことを考えると、この方法のほうがはるかに手軽です。これから同じ状況に挑戦される方には、まずこちらの方法を試されることをお勧めします。ビルドを選択した時は本当に調子に乗っていて、Linux の一般的知識があればわかりそうなこと (deb や rpm を外部ストレージからコピーしてインストールする) がすっかり抜けていたので基本的な知識をいつでも有効活用する力というのは技術力として大切な点だと思いました。

とはいえ、フルビルドをしたからこそ menuconfig の細かい挙動や make target/linux/clean の罠など、カーネルビルドの内部事情を身をもって学べたのも事実です。あえてこの沼にはまりたい人は一緒にはまってみましょう。

付録 A

執筆者・スタッフコメント

第 1 章 Shunsuke Ito / @fgshun

数年ぶりにコーヒードリッパーを買ってみた。コーヒーがおいしくなった……気がする。

第 2 章 Daisuke Makiuchi / @makki_d

眼鏡っ娘が好きです

第 3 章 Daisuke Yamaguchi / @_gutio_

ぽこポケが面白すぎて執筆が進まず大変でした orz

第 4 章 Ayato Takeichi

AI でのゲーム製作も楽しい！

第 5 章 Yoshio Hanawa / @hnw

花粉シーズン終了したらジョギングで健康維持していきたい（と思っているうちに梅雨が来るパターン）

第 6 章 Masato Yamada / @thunder_hey

興味本位でルータをいじり始めたらこんなところまで来てしまいました。

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。毎年ちょっとずつ改善入れてるおかげか色々進行しやすくなってる気がする

lee-d

表紙のイラストを担当しました。いつもとは少し違う作業でしで、悩みもありましたが試行錯誤しながら楽しく制作できました。

既刊・電子版ダウンロード

https://www.klab.com/jp/blog/tech/2026/tbf20.html



KLab Tech Book Vol. 17

2026 年 4 月 11 日　技術書典 20 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

(C) 2026 KLab 技術書サークル


KLab Tech Book Vol.17