ACCESS
株式会社アクセス

ACCESS テックブック

ACCESSのエンジニアが語る秘伝のレシピ集

ACCESS技術書典同好会　著

ACCESS テックブック

ACCESS 技術書典同好会　著

2019-09-22 版　　発行

プロローグ

本書「ACCESS テックブック」は、株式会社 ACCESS に所属するエンジニアによって書かれた初の技術同人誌です。ACCESS はブラウザエンジン開発から、IoT、クラウド、人気スマホアプリ、ハードウェア、ネットワークソフトウェアと、とても幅広い技術分野に対して、それぞれ専門的な技術を持って開発をしています。そこに属するエンジニアは、深い知識を持ち、そしてマニアックな人たちばかりです。各章ははそれぞれ完結しているので、好きな章からお読みください。

株式会社 ACCESS Co-CTO 池内康樹

本書に関する問い合わせ先

https://twitter.com/ikeyasu

株式会社 ACCESS について

日本でインターネットの歩みが始まった 1980 年代、「すべてのモノをネットにつなぐ」という企業ビジョンとともに、株式会社 ACCESS は誕生しました。ACCESS は、このビジョンを DNA として成長し、インターネットの普及とともに「ネットにつなぐ技術」を進化させ続けてきました。IT 革命元年と呼ばれた 1999 年には、世界で初めて「携帯電話をネットにつなぐ技術」の実用化に成功。企業躍進の起爆剤となりました。さまざまなイノベーションを経て、IoT の時代がいよいよ幕を開けようとしています。創業時より思い描いていたビジョンが現実のものになろうとする今、ACCESS は「ネットにつなぐ」技術で、世界により豊かな社会と暮らしを創造し、人々の次の未来の実現を目指します。

https://www.access-company.com/recruit/

本書の電子版は、以下で配布予定です。

http://access-company.github.io/techbookfest

※ 各社の会社名、サービスおよび製品の名称は、それぞれの所有する商標または登録商標です。

目次

プロローグ **3**

第 1 章 クリーンアーキテクチャー **9**

1.1 はじめに . . . 9
1.2 アーキテクチャーを選定する目的 . . . 9
1.3 スマートフォンアプリ開発のアーキテクチャー . . . 9
1.4 重要な SOLID 原則 . . . 10
1.5 クリーンアーキテクチャーとは . . . 13
1.6 今回のアプリ設計 . . . 17
1.7 クリーンアーキテクチャーを導入した結果 . . . 19
1.8 まとめ . . . 22
1.9 謝辞 . . . 23
1.10 本書の Qiita 版 . . . 24
1.11 著者プロフィール . . . 24

第 2 章 HTTPS in Local Network **25**

2.1 はじめに . . . 25
2.2 問題定義 . . . 26
2.3 想定されている解法 . . . 27
2.4 おわりに、CG 参加のすすめ . . . 30
2.5 おまけ: 個人サークルの宣伝 . . . 30
2.6 謝辞 . . . 31

第 3 章 TVM **33**

3.1 Introduction . . . 33
3.2 What is TVM ? . . . 33
3.3 Other Deep Learning Compiler . . . 42
3.4 Optimization Detail . . . 43

3.5 AutoTVM 48
3.6 Tutorials 53
3.7 Reference 53
3.8 Link 54
3.9 Author 57

第 4 章 Web Frontend Boilerplate Battle 59
4.1 私を取り巻くウェブフロントエンド事情 60
4.2 boilerplate 実装という大きな壁 61
4.3 Introducing "igata" 62
4.4 igata の設計思想 64
4.5 igata の採用する技術 68
4.6 まとめ 92
4.7 Web Frontend Boilerplate Battle 92
4.8 完走した感想 93
4.9 参考・引用 93
4.10 著者 94

第 5 章 Elixir で brainf*ck した話 95
5.1 Agenda 95
5.2 動機 96
5.3 Brainf*ck とは? 96
5.4 閑話休題 97
5.5 まずやること 1: 車輪の再発明か否かチェック 97
5.6 まずやること 2: README を書く 98
5.7 コードを生成する 99
5.8 テスト 100
5.9 ハマりポイント 1: そもそも出来ない前処理 101
5.10 ハマりポイント 2: コンパイルエラーのテスト 102
5.11 ハマりポイント 3: `@spec` があるか確かめる 103
5.12 まとめ 104

第 6 章 Antikythera 105
6.1 Agenda 105
6.2 Antikythera Framework とは 105
6.3 コア機能と利点 106
6.4 Antikythera console 112

6.5　Antikythera の内部実装について 113
6.6　まとめ 117
6.7　著者 118

第 7 章　STM32 マイコンの Ethernet ドライバ　119
7.1　おことわり 119
7.2　話のさわりと目的 119
7.3　サンプルコードの必要性 120
7.4　相次ぐトラブル 121
7.5　マイコン向けプログラミングとは 126

あとがき　127

第 1 章

クリーンアーキテクチャー

1.1　はじめに

はじめまして、第 1 章を担当する K.Nagauchi です。2018 年に ACCESS に入社して以来、Android™と iOS のスマートフォンアプリを開発しています。

昨年から新規開発が始まった受託案件に、話題のクリーンアーキテクチャーを導入しました。本章では、そこで学んだ諸々をご紹介します。クリーンアーキテクチャー自体の解説もある程度含まれますが、ページ数の都合上、歴史背景や細かい部分までは説明していませんのでご容赦ください。

簡単ですが、案件の情報を下記に記載します。

プラットフォーム	Android/iOS
言語	Kotlin/Swift
開発チーム人数	3〜6 人

1.2　アーキテクチャーを選定する目的

なぜアーキテクチャーを選定するのか？ それは、求められるシステムの構築・保守に必要な人材を、最小限に抑えたいからです。「アーキテクチャーは上位レベル、設計は下位レベル」のように区別されることもありますが、両者の間に明確な境界はなく、上位から下位に至るまで決定の連続です。

1.3　スマートフォンアプリ開発のアーキテクチャー

MVVM、MVC

スマートフォンアプリ開発で代表的なアーキテクチャーとして、Google が Android に推奨している **MVVM**、Apple が iOS の Cocoa Application に採用している **MVC** が

あります。

では、Android、iOS 両方で同じアプリを作る場合はどうでしょうか。並行作業を楽にするため、少なくともビジネスロジック部分は共通化したいですよね。

Xamarin や Flutter、Kotlin MPP のような X-Platform 開発手法が近年は有名ですが、Native と比較すると保守できるエンジニアが少ない、Trouble shooting 事例が少ない、サポート打ち切りの可能性が上がるなどのリスクが伴います。

受託案件では、何か発生したとき迅速にアップデートできるほうがよいので、それらのリスクを考慮して採用せず、代わりにアーキテクチャーを明確に定めることにしました。

VIPER

MVC を用いると、View にも Model にも属さないコードが Controller に実装されがちで、気がつけば膨大な Controller と、小さな View & Model という構成になりがちです。

VIPER は、iOS で MVC に置き換わるとされているクリーンアーキテクチャーの一種で、単一責任の原則（SRP）に基づいています。**V**iew、**I**nteractor、**P**resenter、**E**ntity、**R**outer の略語です。

Android は View と Router の分離が難しいので適用には不向きですが、VIPE で組み上げた後に V の一部を Router 化するという工夫も存在します。

ですが、VIPER に執着するくらいなら、クリーンアーキテクチャー適用で充分では？と私達は考えました。こうして、案件にクリーンアーキテクチャーが選定されました。他のどんなアーキテクチャーと比較したのかは、章の後半で改めて説明します。

1.4　重要な SOLID 原則

本題に入る前に、先ほどチラッと出てきた「○○の原則」について、もう少し説明します。

- **SRP: 単一責任の原則**
  - 1 つのモジュールはたった 1 つの役割に対して責務を負う
- **OCP: オープン・クローズドの原則**
  - 拡張に対して開かれており、修正に対して閉じていること
  - つまり変更が発生したら既存のコードは修正せず、新しくコードを追加して対応すること
  - オブジェクト指向設計の核心で、再利用、保守、柔軟性のメリットが受けられる
- **LCP: リスコフの置換原則**
  - 基底クラスから派生したサブクラスは、常に基底クラスと置き換え可能である

こと

- **ISP: インターフェイス分離の原則**
  - クライアントが利用しないメソッドをインターフェースに含めるべきでない
  - クライアント毎に細かく分離すべき
- **DIP: 依存関係逆転の原則**
  - 抽象モジュールを具象モジュールに依存させるべきではない。具象を抽象に依存させること

これらのうち、DIP がクリーンアーキテクチャーにおいて特に重要な原則で、次いで SRP、OCP が重要と私達は考えています。

依存関係逆転の原則（DIP）

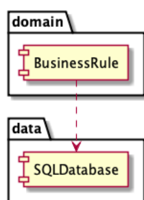

図 1.1: レガシーな構成

レガシーな階層アーキテクチャーでは、図 1.1 のように、ドメイン層 (ビジネスルール) から永続化層 (DB) へと直接参照が行われていました。

この構成は、抽象（BusinessRule）が具象（SQLDatabase）に依存していますが、明らかに DIP に違反しています。では、どうすればよいでしょうか。

違反を解決するには、図 1.2 のようにインターフェースを挟んだ構成にします。

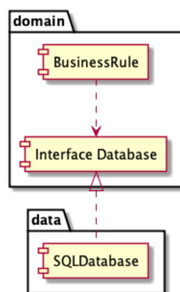

図 1.2: DIP を適用した構成

これだけで、具象（SQLDatabase）が抽象（BusinessRule）に依存するようになり、DIP に準拠させることができました。

インターフェースは、同じモジュール内にある抽象（BusinessRule）に適合した仕様になっています。具象（SQLDatabase）側を、抽象（BusinessRule）側のインターフェース仕様に合わせて組み立てる必要があります。

こうすることで、先ほどと依存関係が逆転し、柔軟なシステムになったと言えます。なぜでしょうか。

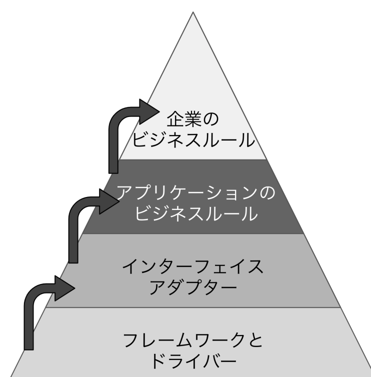

図 1.3: 上位レイヤーと下位レイヤー

技術は変わりやすく廃れやすいものです。例えば、使っていた OSS の更新が止まったので別の OSS に乗り換えたい、DB フレームワークのパフォーマンスが悪いので別のものに乗り換えたいといった場合、図 1.3 における企業のビジネスルールが OSS や DB に依存していると、乗り換えが容易ではありません。

一方で、図 1.3 の上位レイヤーのビジネスルールは、そう簡単には変わりません。例えば銀行の ATM 機の使い方は、短期間で頻繁に変わったりしないですよね。

変わりにくいものを変わりやすいものに依存させると、システム全体が不安定になります。そこで、依存関係を逆転させれば、抽象モジュールへの影響を最小限に抑えつつ、具象モジュール側を変更できるのです。

また、具象モジュールのモックを用意することで、単体テストの作成・実施も容易になります。

1.5　クリーンアーキテクチャーとは

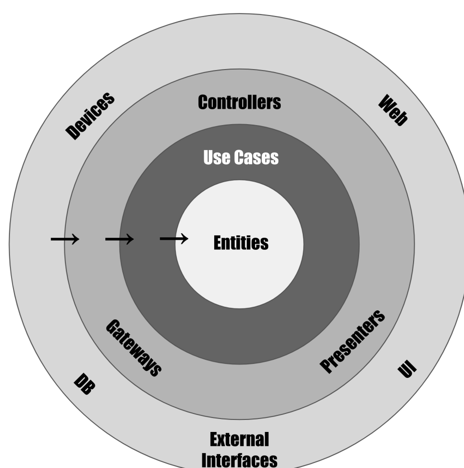

図 1.4: クリーンアーキテクチャー

図 1.4 の矢印は依存関係を示しています。依存性は内側だけに向かっていなければならない、というのがクリーンアーキテクチャーの最重要ルールです。

円の内側は、外側について何も知らず、特に外側で宣言された変数、関数、クラスなどの名前に、内側のコードから触れてはなりません。外側ほど変わりやすく、内側ほど変わりにくいからです。

円の中心のビジネスルールを基準とすると、以下の依存関係が成り立ちます。

- ビジネスルールはフレームワークに依存しない
- ビジネスルールは単体でテスト可能
- ビジネスルールは UI に依存しない
- ビジネスルールはデータベースに依存しない
- ビジネスルールは外界のインターフェイスに依存しない

図 1.4 は概要なので、上記原則が満たされていれば何層でも構いません。が、過剰な階層化は生産性を下げることになってしまうので、注意しましょう。

制御の流れ

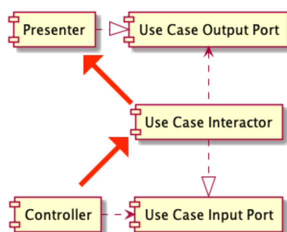

図 1.5: 制御の流れ

図 1.5 は、Controller から入力を行い、出力を Presenter で表示する制御の流れを示しています。

すべての制御は、Use Cases を介して行われるべきです。また、外側のコンポーネント同士（Controller と Presenter）は直接データをやり取りしないべきです。我々はそれを**環状道路の罠**と呼んでいます。

境界線を引くコツ

境界線を引くコツとしての 1 つ目は、重要なものとそうでないものの間です。UI、DB はそれぞれビジネスルールにとって重要でないので、間に線を引きます。

2 つ目は、変更の軸が変わる境目です。UI、Web とのインターフェースはビジネスルールと異なる理由・頻度で変更されるので、間に線を引きます。

また、先ほど説明した単一責任の原則 (SRP) も、境界線を引くための指針となります。

各コンポーネントのレベル

入出力からの距離で、コンポーネントのレベルを判断します。入出力に近いほど下位レベルとなります。下位レベルのコンポーネントを円の外側に、上位レベルのものを内側に配置し、依存関係逆転の法則 (DIP) によって、下位を上位に依存させます。

各コンポーネントの説明

Entities

Enterprise Business Rules(企業のビジネスルール) 層で、最重要ビジネスルールや、最重要ビジネスデータを指します。例えば、銀行の利子計算や、オンラインストアの商品販売などのように、企業 (組織) 内で共通するルール・データを記述する部分です。

企業のソフトウェアでないならば、アプリケーション固有のビジネスルール・データ (後述) の最重要部分となります。

DB や E-R モデルでいうところの「エンティティ」とは少し違うので注意しましょう。この層は、特定のデータベースには依存しません。「ルール」なので、何らかのメソッドを持ったクラスかもしれません。

Use Cases

アプリケーション固有のビジネスルール層です。ここで言うビジネスルールは、ソフトウェアが存在する理由であり、「手段」と混同してはなりません。データベースや UI など、下位の詳細には関わらないべきとされています。例えば、下記のような制御の流れを示すものがユースケースです。

```
1. テキストボックスに入力された宛先・件名・本文を受け取る
2. 宛先を検証する。宛先が適切なら次に進む
  a. 宛先が空か不正ならダイアログでエラーを通知して終了
3. 件名を検証する。件名が適切なら次に進む
  a. 件名が空なら警告を表示する
    i. ユーザが OK ボタンを押したら承認したら次に進む
    ii. ユーザがキャンセルボタンを押したら拒否したら終了
4. メールを送信する
```

Interface Adapters

ユースケースと外界 (UI やデータベース) とのデータ変換などを担う部分です。MVxx アーキテクチャーの xx 部分（View、ViewModel、Presenter、Controller など）は、ここに属します。Model はユースケースの集まりと考えられるため、より上位です。

エンティティとデータベースフォーマットとの間の変換もここで行います。SQL なら SQL 文、Realm なら Realm Object です。

Frameworks & Drivers

外部との境界で、フレームワークやツールを使う部分です。あまりコードを書かず、実際書くとしても Interface Adapters と一体になるでしょう。

メインコンポーネント

システムの入り口 (main 関数) を含む部分で、Android なら Application や MainActivity を含むモジュールです。最下層に属するので、唯一すべてのコンポーネントへの参照が許されます。

この層で、DI フレームワークを使って依存関係の注入を行います。

「詳細」なもの

詳細なものは、図 1.4 の円の内側に入り込まないよう気をつけます。

データベースは詳細です。データの保存にファイルを使うのか、RDBMS を使うのかは、アーキテクチャーとしては重要ではないからです。

Web も詳細です。Web は集中と分散の歴史をひたすら繰り返しており、非常に変わりやすいものです。また、入出力デバイスのひとつとも考えられます。

フレームワークも詳細です。フレームワークはとても便利ですが、そこに依存すると簡単に抜け出せなくなるからです。「フレームワークと結婚するな、フレームワークとは距離を置け」という言葉もあります。フレームワークを使うこと自体は禁止されていません。

ビジネスルールは本当に具象に依存すべきでないのか

すべての具象をユースケースから排除することは不可能です。例えば、String は具象ですが、String なしで開発することはできません。String は安定しているので、使っても問題ないと考えます。List などのコレクションも同様です。

微妙なラインにいるのが Rx です。Rx が今後も廃れる心配がないなら使ってよいと考えます。Android Clean Architecture のサンプルは RxJava を使っています。

優れたアーキテクチャーとは

優れたアーキテクチャーの原則を以下に紹介します。

- 開発しやすい
  - 適切に分割されたコンポーネントがあれば分担しやすい

- テストしやすい
  - ユースケースがフレームワークに依存しないので独立したテストができる
- 保守しやすい
  - 「洞窟探検」のコストを抑える
- 選択肢を残しておく
  - データベース、UI、プロトコルなど詳細に関する決定を先送りにできる
  - 「ソフト」とは、柔軟に変更できるという意味
  - 変更できることにソフトウェアの価値がある

クリーンアーキテクチャーまとめ

Clean Architecture とは、ユースケースをシステムの中心として捉え、ユースケース (上位の方針) とフレームワーク等 (下位の詳細) を分離し、依存関係の逆転によって下位を上位に依存 (プラグイン化) させ、コンポーネントごとのテストを容易にし、詳細部分の置き換えを容易にする設計手法です。

「速く進む唯一の方法は、うまく進むことである」(Robert C. Martin)

1.6　今回のアプリ設計

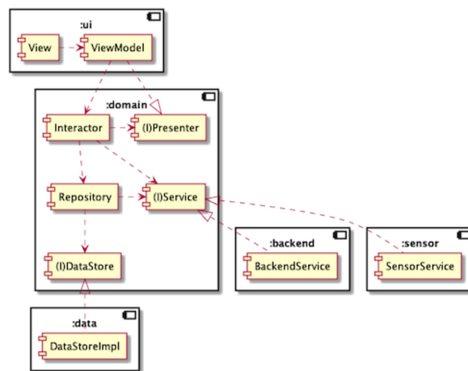

図 1.6: アプリの全体設計

ui モジュール

名前の通り、ユーザーインターフェイスを実装するモジュールです。OS に依存します。

View は、Android で言う Activity や Fragment です。ViewModel のデータを表示

し、ユーザの入力に応じて ViewModel にコマンドを発行するクラス/メソッド群です。

ViewModel は、Interactor から受け取ったデータを表示可能な形式に変換したり、View などからコマンドを受け取って Interactor のメソッドを呼び出すクラス/メソッド群です。

domain モジュール

ビジネスロジックを実装するモジュールです。**OS には依存しません**。

Interactor は、ユースケースを表すクラス群です。ViewModel から見た Model に相当します。ビジネスロジックはここに書きます。ユースケースごとにクラスを作成する決まりです。

Presenter は、処理したデータを ViewModel に渡すためのインターフェース群です。表示に適したデータ加工（ソートなど）は行いません。

Repository は、データへのアクセス手段を持つクラス群です。データがサーバーにあるかローカルにあるかを Interactor が意識しなくて済むように隠蔽します。サーバから取得したデータを DB に保存するなどの処理を行います。データの CRUD に専念し、複雑なロジックは書かない決まりです。

Service は、サーバや端末センサーにアクセスするためのインタフェースです。backend モジュールや sensor モジュールに具象クラスを持ちます。

DataStore は、ローカルのデータベースにアクセスするためのインタフェースです。data モジュールに具象クラスを持ちます。

Model(図では省略) は、エンティティ (ドメインモデル) の集まりです。アプリで扱うデータを単純なデータクラスで表現します。

data モジュール

データの永続化を行います。OS に依存します。今回は Realm を使用しました。

なお、全ての永続データを data モジュールに集めるとは限りません。UI 表示に関する設定は、ここではなく ui コンポーネント内で、より簡素な方法（XML など）で保存します。

backend モジュール

サーバーとの通信を行います。OS に依存します。今回は Retrofit/OkHttp を使用しました。

sensor モジュール

端末センサーからのデータ読み出しを行います。OS に依存します。

utility モジュール

ログ機能や String の拡張関数のような、ビジネスロジックと直接関係のないユーティリティをここに実装します。domain からも参照可能にするため、**OS には依存しません**。

OS 依存の API を使う場合は interface 化し、実装を backend、data、sensor、もしくはより下位レベルの app モジュールなどに持たせます。

app モジュール

図 1-6 には載せていませんが、メインのコンポーネントで、各コンポーネントの初期化などを行います。

DI コンテナ（Kodein/Swinject）はここで管理し、各クラスのコンストラクタやプロパティを通じて、依存性の注入を行います。

1.7　クリーンアーキテクチャーを導入した結果

クリーンアーキテクチャーを導入してどのように感じたか、開発が一区切りついたタイミングでメンバーにアンケートを取りました。

Q1: クリーンアーキテクチャーの理解はスムーズでしたか？

- そう思う (5pts)………………2 人
- ややそう思う (4pts)………………0 人
- どちらとも思わない (3pts)………………3 人
- ややそう思わない (2pts)………………1 人
- そう思わない (1pt)………………0 人

平均：3.50pts

Q2: クリーンアーキテクチャーを使ったことにより、保守しやすいコードが書けたと思いますか？

- そう思う (5pts)………………2 人

- ややそう思う (4pts)………………4 人
- どちらとも思わない (3pts)………………0 人
- ややそう思わない (2pts)………………0 人
- そう思わない (1pt)………………0 人

平均：4.33pts

Q3: クリーンアーキテクチャーを使ったことにより、質の高いコードが書けたと思いますか？

- そう思う (5pts)………………4 人
- ややそう思う (4pts)………………2 人
- どちらとも思わない (3pts)………………0 人
- ややそう思わない (2pts)………………0 人
- そう思わない (1pt)………………0 人

平均：4.67pts

Q4: クリーンアーキテクチャーを使ったことにより、開発時間が短縮できたと思いますか？

- そう思う (5pts)………………0 人
- ややそう思う (4pts)………………1 人
- どちらとも思わない (3pts)………………3 人
- ややそう思わない (2pts)………………1 人
- そう思わない (1pt)………………1 人

平均：2.67pts

よかった点（自由記入欄）

- 構造化の度合い
  - 各クラスの責務が小さくなり、内容を理解しやすかった
  - 各クラスの役割がある程度明確になった
  - それぞれのレイヤー毎に単体テストできた
  - 上手にモジュール分けされているので、この部分が未定だけどそこは stub にして他を先に実装するという判断がしやすい
- 一貫性・保守性

  - 指針が明確になったことで、誰が作っても同様の設計になり保守性が上がった
  - レビュー時にも「クリーンアーキテクチャーに基づいているかどうか」という明確な観点からコードを見ることが出来てやり易かった
  - ほぼほぼ SOLID 原則に従った実装になりやすかった
  - 仕様変更時の改修量が少なかった（例：サーバー API が急に変更になったが、backend の JSONObject クラス 1 つを修正するだけで済んだ）
  - ほとんどのクラスがコンパクトになった
  - モジュール毎に機能を分けて書いていき、それぞれに必要以上に依存関係を持たせないので保守がしやすい
  - 質の高いコーディング以外受け付けないので、強制的に質が高くなるところが良い
- 理解可能性
  - 修正箇所を見つけるのも意外と苦ではなかった
  - どの機能においても似たようなコーディングの仕方になるので、後から「この機能どうなっているんだっけ」と確認する時に探しやすかった

改善すべき点（自由記入欄）

- 構造化の度合い
  - ui モジュールの各クラスが肥大化した（View と Navigation を両方含んでいるからと思う）
  - Presenter と ViewModel を分離すべきだったかも
  - もしくは、Controller と Presenter の実装を分けるべきだった
  - いちいち UseCase(Interactor) を通さないといけない場合があるので、それが面倒
  - クリーンアーキテクチャの模範例に従いすぎずモジュールに柔軟性も持たせた方がよかったかも
  - ファイル数が多い
  - 仕様が読みづらくなるので、品質の高い仕様書（UML）を書くべき
- 理解可能性
  - アーキテクチャーの学習コストがかかった（選定に関わった人は習得が早いが、そうでない人はちょっと苦戦する傾向あり）
  - 敷居が高い。理解するまでに時間がかかった
  - どこに責務を持たせるか、人によって判断の違いが若干あり、その場合議論が起きやすい（レビューの長さが時々ボトルネックになっていた）

1.8 まとめ

クリーンアーキテクチャーの苦手なポイントとは

もし既に MVC などの他アーキテクチャーを使っていたら、移行がそれなりに大変です。また、一部の開発者にとっては理解が困難かもしれません。

スクラムでやる程度の規模の開発には向いていますが、個人開発だとやりすぎ感があり、反対に規模が大きくなっても、Entity 部分の肥大化に伴う構成の複雑化、保守性や可読性の低下などが問題となり、アーキテクチャーの強みが活かせないと感じました。が、各クラスが小さくなるというメリットは、大規模になっても活きます。

Entity と言う呼び方がわかりづらい、ドメイン駆動設計の Entity とは意味合いが微妙に違って混乱する…という意見も多数寄せられました。

データのコピーや変換による速度低下・オーバーヘッドは問題にならないか

異なるモジュールにデータを渡せばコピーや変換が発生します。その際の処理低下やオーバーヘッドですが、繰り返し何度も変換するわけではないので、許容範囲と感じました。

例えば data から domain なら、DB に依存する型から domain で扱える簡素な型への変換が必要ですが、domain から UI には変換は必要なく、描画処理で吸収すればよいです。そうすると 1 回の変換で済むので、目に見える速度低下には繋がりません。

すごく長いリストを扱うと、トータルで何割かのオーバーヘッドが起きて問題となるかもしれません。が、そのせいでアーキテクチャーを諦める前に、リストの最大数を制限する仕様としたり、言語のアンチパターンを取らないなど、他の努力でカバーできると考えられます。

また、理由があれば json 形式のまま backend から UI へ渡してよいなど、柔軟な対応をすることもあります。

他に検討したアーキテクチャーは？

先ほど挙げた MVC、MVP、MVVM から、MVW、Flux、Redux、Angular あたりを検討しました。

何故クリーンアーキテクチャーを選んだのか

スマートフォンアプリのクロスプラットフォーム開発に向いており、iOS の MVC のように特定のモジュールが肥大化する恐れも無いと考えたからです。

また、マネジメント層からは開発スピードより品質に重きを置いてほしいと言われており、単体テストしやすいアーキテクチャーとして選びました。

ただし、正直なところ、じっくりと検討したわけではありませんでした。エンジニア招集から開発着手まであまり日数が無く、1つ1つのアーキテクチャーをじっくり選定するより、最後はチームメンバーの得意不得意や特性を踏まえて、直感に頼るのも大事だと思いました。

1.9 謝辞

本書を書くにあたって、以下の本や記事を参考にしました。

- Clean Architecture 達人に学ぶソフトウェアの構造と設計 Robert C.Martin(著), 角 征典 (訳), 高木 正弘 (訳)
- The Clean Code Blog by Robert C. Martin (Uncle Bob) - Clean Coder Blog
  - https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- MVVM architecture, ViewModel and LiveData (Part 1) by Hazem Saleh - ProAndroidDev
  - https://proandroiddev.com/mvvm-architecture-viewmodel-and-livedata-part-1-604f50cda1
- Model-View-Controller by Apple - Documentation Archive
  - https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html
- iOS Project Architecture: Using VIPER by Pedro Henrique Peralta - Cheesecake Labs
  - https://cheesecakelabs.com/blog/ios-project-architecture-using-viper/
- Using the VIPER architecture on Android by Marcio Granzotto Rodrigues | Cheesecake Labs
  - https://cheesecakelabs.com/blog/using-viper-architecture-android/
- 単一責任の原則（SRP） by gomi_ningen - Qiita
  - https://qiita.com/gomi_ningen/items/02c42e2487d035f9c3c8
- オープン・クローズドの原則の重要性について by riki(Rikitake) - Eureka Engineering - Medium

  - https://medium.com/eureka-engineering/go-open-closed-principle-977f1b5d3db0

以上の著者の皆様と、クリーンアーキテクチャーの要点資料作成やご指導をくださった弊社の油井真斗さん、アンケートにご協力くださった同じ案件の皆様、Qiita 版を含めて本記事をレビューしてくださった皆様に、この場を借りて感謝を申し上げます。

- Android は Google LLC の商標です。
- iOS は Apple Inc の商標です。

1.10 本書の Qiita 版

- クリーンアーキテクチャーでスマホアプリ開発した感想（勉強会用）by knagauchi - Qiita
  - https://qiita.com/knagauchi/items/8c9fb93520b56fd3a564

1.11 著者プロフィール

図 1.7: 著者近影

長内 健作 （ナガウチ ケンサク） @tonionagauzzi

ソフトウェアエンジニア。2010 年から他社での Android アプリ開発を経て、2018 年 2 月に株式会社 ACCESS に入社。現在では iOS も含めたマルチプラットフォームのアプリ開発を手がける傍ら、スマートデバイス・スペシャリストとしてトレンド技術の発掘や新人研修の講師などを担当している。

個人で制作した Android アプリ「強制停止チェーン」は 2012 年に 10 万ダウンロードを記録（現在は諸事情で公開を停止）。2019 年 6 月 14 日、クリーンアーキテクチャーの記事で Qiita の日間トレンド 3 位に浮上。そういった経緯で本書を執筆。

第 2 章

HTTPS in Local Network

2.1 はじめに

2019 年 3 月まで ACCESS 社員だった sylph01[*1]です。現在は別の W3C 会員企業に所属しています。

ブラウザベンダとして知られる ACCESS は W3C 会員企業で、かつてはモバイル向け HTML や EPUB の標準化活動を W3C で行っていました。今回は ACCESS 在籍時から継続して活動している W3C の Community Group である HTTPS in Local Network Community Group の活動を紹介しようと思います。

W3C の全体会議（のうち技術を扱うもの）である TPAC(Technical Plenary and Advisory Committee) は今年は福岡で開催されます[*2]。HTTPS in Local Network CG は日本人メンバーが非常に多く参加しており（というよりも大多数が日本人だし日本発のグループ）、今回の TPAC を期に CG レポートを発表し議論を大きく進展させることを目論んでいます。本章ではそもそもこのグループがどのような問題を扱っているか、これまでの経緯、そして想定されている解法の提案について紹介します。

なお、本文中には TLS や PKI に関する用語が説明なく用いられる場合がありますのでご了承ください。市販されている本では『プロフェッショナル SSL/TLS』(Ivan Ristić 著、齋藤孝道 監訳, ラムダノート, 2017) がこの分野の説明では一番網羅的に解説しておりおすすめです。

[*1] Twitter: `@s01`, GitHub: `@sylph01`

[*2] 2019 年 9 月 16-20 日なので、本原稿の執筆時にはまだ開催されておらず、本原稿が世の中に出る頃には既に終わっているという、絶妙にネタにしにくいタイミング…！

2.2 問題定義

政府などによる大規模な盗聴が明らかになったことから、現代の Web においては常時 HTTPS 化が叫ばれています。HTTPS(TLS) は暗号化と接続先の認証のために証明書に依存しており、この証明書は発行主体の名前を利用するため DNS、特にグローバルなインフラとしての DNS に依存しています。一方で Internet of Things(IoT) の世界観ではローカルネットワークのデバイスがサーバーとして振る舞うことも増え、これらへの接続も HTTPS で行いたい、というユースケースが出てきます。ところで、ローカルネットワークに対してはグローバルな DNS に対して有効な名前を払い出さないことが普通です。ローカルなデバイスにいちいち名前をつけるコストも高い、それを DNS に登録するコストも高ければ、ローカルネットワーク内にどのようなデバイスがあるかということが DNS から明らかになってしまうことにはプライバシー上の問題もあります。このため、グローバルに有効な名前がない以上ローカルネットワークのデバイスに有効な証明書を払い出すことができず、結果としてローカルネットワークには HTTPS がない、というのが今までの Web PKI の前提でした。

ローカルネットワークに対して HTTPS が欲しいといった場合には、自分で自己署名用の認証局を立てて自己署名証明書を払い出し、その証明書を使うマシンすべてに自分の認証局のルート証明書をインストールする、ということが一般的でした。一般的、とはいっても、システム管理者がこれらのコンセプトを完全に理解し、グローバルな運用をせず限られた範囲で運用をすることが前提の世界観での「一般的」なので、エンドユーザーがこのような仕組みを構築することはまずありません。また、エンドユーザーにルート証明書をインストールさせるモデルでローカルネットワークへの HTTPS 化を実現してしまうと、エンドユーザーはこの行為のリスクを理解しないままに OK ボタンを押すことが増えてしまい、結果として攻撃者が盗聴用の証明書のインストールをエンドユーザーに行わせることも容易になってしまい、「信頼された認証局のみが有効な証明書を払い出すことができる」という Web PKI の大前提を壊すことになってしまいます。

HTTPS in Local Network CG では、IoT の進展によってエンドユーザーの家庭内でも HTTPS をしゃべるローカルネットワークのデバイスが登場した際に、これらのデバイスへの HTTPS でのアクセスを、現状の PKI の枠組みの中で、あるいはそれをローカルネットワークに限定した拡張を行うことで提供する、ということを目的としています[*3]。

[*3] draft 版の charter は `https://httpslocal.github.io/cg-charter/` に、ユースケースの記述は `https://github.com/httpslocal/usecases` にあります。

2.3　想定されている解法

現在想定されている解決方法において、トレードオフとなるポイントには以下のような点があります。

- 現状の Web PKI の仕組みやユーザーエージェントの実装に手を入れる必要があるか
- インターネット接続がない状態でも動作するか
  - 証明書の検証のためにインターネット接続があることを前提としているか、を意味します
- デバイスのドメイン名が DNS 経由で公開されるかされないか
- デバイスがインターネットから reachable であるか否か

ある解決方法を導入するにあたって、現状の PKI に独自の拡張を入れることになった場合、それが現状の PKI の仕組みに脆弱性をもたらさないか適切に検証を行う必要があるため非常にコストが高く、可能であればそれは回避したいです。これはブラウザの実装に手を入れる必要が出てきた場合も同様です。

デバイスのドメイン名がインターネットに公開されるか否か、またそれがインターネットから reachable になるか否か、はデバイスオーナーのプライバシー（と場合によってはデバイスのセキュリティ）上の性質です。現在 CG で考慮されているような問題を既に実装したものとして Mozilla の Things Gateway というものがありますが、これは Mozilla がデバイスに DNS 名を付与することで証明書を払い出すことを可能にしている仕組みで、結果として DNS レコード上にデバイス名が現れ、外部からデバイスがアクセス可能になる、という性質があります。特に外部からデバイスがアクセス可能になるという点は非常に気をつけなければならない点で、エンドユーザーのデバイスが勝手にインターネットにつながってしまう方法をデフォルトにするのは危険性が高いと考えられます。

ここでは TPAC 2019 にて CG が解決方法として提案する予定である 2 つの方法を紹介します。

自己署名証明書を UA 経由で信用する

あれ？ さっきは「自己署名証明書は解ではない」と言っていなかったっけ？…というのは、今までの PKI の仕組みに基づく「自己署名証明書」では、「証明書エラーを無視して受け入れる」か「ルート証明書をインストールする」かの方法しか取れず、どちらにせよ常態化するとユーザーに信頼の判断を押し付けセキュリティリスクを生み出すために解ではないとしていました。提案されている方法では、「ローカルドメイン」に対してのみ自

己署名証明書を無条件に（強い表示の出る）証明書エラーとはせず、信頼・不信頼の判断をするユーザーインターフェースを用意する、という方法によって、ローカルネットワーク内の自己署名証明書を持つデバイスに対して HTTPS ができるようにします。もちろんこの方法はユーザーエージェントの実装に手を入れる必要が出てくるため実装コストは高いですが、CA が末端のデバイスに対して適切に認証を行い証明書を払い出す際に発生する運用コスト・プライバシー上の懸念を解決できます。

この方法には 2 つのバリエーションが提案されています。

Mozilla の Martin Thomson 氏の提案[*4]では、ローカルネットワークのデバイスに接続した際に表示する警告画面に「デバイスの証明書の公開鍵に基づく (Gravatar のような) アイコン」と「接続先の名前」を表示し、その接続先に接続したことがあるかどうか、その上で接続を受け入れるかどうか、を表示する UI を出す内容を提案しています。ローカルネットワークのデバイスに名前を振る場合にランダムな名前を振れば名前付与の問題は解決するように見えますが、そうした場合ユーザーは (以下の名前は一例です) `huavj8w96w3a2629hj63mcr98qvxb7f4.local` は知っているデバイスか、`yekn34szudbnv2vfk3j59z8nzb6sfmbi.local` は危険なデバイスであるか、を識別しなくてはいけなくなります。この提案ではユーザー可読であるようにアイコンでデバイスを識別しているようにしている点が特徴的であるといえます。また、本提案では証明書に基づくアイコンを表示するためにサーバーの公開鍵情報 (SPKI) のハッシュを利用するため、SPKI ハッシュを origin に含められるように origin を拡張する、という提案を行っています。origin の構成要素を変更することは Same Origin Policy の土台を変更することになるので多くのアクターへの交渉が必要になりますが、その中でも主要ブラウザベンダーからこの提案が出てきたということは注目に値することでしょう。

東芝の安次富大介氏の提案[*5]では、WebAuthn のトラストモデルを応用することでユーザーがデバイスの証明書に信頼を与えられる方法を説明しています。WebAuthn では FIDO サーバーと Authenticator の間にはデバイスアテステーションに基づく信頼関係が構築されており、Relying Party(RP) と FIDO サーバーの間にも RP が FIDO サーバーを信頼することで Authenticator の信頼の権威を FIDO サーバーに求められるようになっています。最後に、RP のアプリケーションに対して Authenticator を使用できるようにするためにはユーザーが WebAuthn API の利用を許可する手続きが必要になります。これと同様の仕組みを、FIDO サーバーをプライベート認証局（ここでは特にデバイスベンダーの認証局を想定している)、Authenticator を各デバイスに見立て、WebAuthn API の代わりに生の公開鍵 (raw public key) あるいは自己署名証明書を含む形の fetch API にユーザーが許可を与えることで、RP が提供するアプリケーションが cross-origin

[*4] `https://github.com/httpslocal/proposals/issues/1`
[*5] `https://github.com/httpslocal/proposals/issues/2`

でデバイスに HTTPS リクエストをしようとした場合にユーザーの許可を求められるようにしよう、というのがこの提案の内容になります。文字だけで説明するのにはかなり限界がありそうな内容なので、図表を伴った昨年の TPAC 2018 での発表スライド[*6]を見ながら追うと対応関係がわかりやすいのではと思います。

Name Constraints

Name Constraints は既に存在する CA 証明書の拡張の一つで、CA あるいはその下位 CA がどの IP アドレスや DNS 名などに対して証明書を発行してよいかの制限をかけるものです。これの使いみちには CA が特定の名前空間に限定して中間 CA に中間 CA として証明書を発行する能力を与える使い方や、内部ネットワークに限定した Name Constraints を持つ自己発行の CA 証明書をインストールすることで内部ネットワークへの通信は自己署名証明書でできるようになる一方で外部ネットワークへの盗聴目的で自己発行の CA 証明書が利用されないようにできる、といった使い方があります。Name Constraints を用いる提案では、ブラウザに既にルート証明書の入っている CA が、特定ドメイン名のサブドメインに対してのみ発行能力を限定した下位 CA を Name Constraints を用いて作成し、デバイスベンダーは自身のドメイン名のサブドメインの証明書を作成してデバイスにインストールする、という手順で、ルート証明書につながるトラストチェーンを持つ証明書を払い出すことを提案しています。

既に存在する拡張で実現できるならもう解決しているのでは？ と思われるかもしれませんが、まずひとつに、ブラウザの対応状況がまだ追いついていない、つまり Name Constraints は拡張であるため、Name Constraints の付与されている証明書で制約外の証明書を発行してしまった場合でも非対応ブラウザはそれに気づくことなく受け入れてしまいます。Netflix の運営する BetterTLS という Web サイトではブラウザが Name Constraints に対応しているかをテストできるような仕組みを用意して Name Constraints の普及を促進していますが、モバイルブラウザを中心に[*7]まだ対応していないものも見られ、実際に機能が浸透しきるには時間が必要であることは事実でしょう。また、この方法では CA に運用コストを押し付けることになってしまいます。ドメインごとに中間 CA を新しく作るので、CA に付随する監査や鍵管理の業務がその分だけ増えることになります。もちろんその中間 CA 業務をドメイン管理者側がやってもよいのですが、今まで CA がやっていたような高いセキュリティ・信頼性の求められる業務を突然デバイスベンダーができるようになるかというとその限りではないでしょう。

[*6] `https://github.com/httpslocal/group` 以下、`20181025.w3c-tpac.httpslocal.proposal.pdf`
[*7] 別に NetFront Browser に限ってではないらしい

提案方法のまとめ

これらの 2 つの方法は以下のように捉え直すことができます。

「証明書エラーを無視して受け入れる」か「ルート証明書をインストールする」方法でローカルネットワークへ HTTPS を実現した場合、エンドユーザーは実質すべての証明書を無条件に受け入れるようになってしまう、ということを説明しましたが、

- UA 経由で証明書を信用する方法では「特定 1 つの証明書のみを」信用することを選択する
- Name Constraints では「すべてのドメインに対して発行が可能なルート証明書を受け入れる」代わりに「CA が特定のドメインに限定した証明書を発行する」

ということによって、それぞれすべての証明書を受け入れる代わりに、ユーザーあるいは CA が影響範囲を限定することでローカルネットワークに HTTPS を導入できるのではないか、としている、といえるでしょう。

2.4 おわりに、CG 参加のすすめ

いかがでしたか？（まとめサイトにありがちな締め文句）

ところで、W3C の活動の多くは会員限定となっていますが、Community Group は W3C 会員資格がなくても参加することができます。W3C の Web サイトにアクセスし個人アカウントを作ることで[*8]Community Group の活動に参加することができます。Community Group では W3C の公式の Working Group になる前のアイデアを会員外も含めた多くの人の手を使って進めるため、先進的なアイデアを実現すべく集まっているグループが多く存在します。みなさんも興味を持った Community Group があれば参加してみてはいかがでしょうか？

2.5 おまけ: 個人サークルの宣伝

sylph01 の個人活動名義のサークル "Cryptic Command" は今回の技術書典 7 はサークル参加しておりませんが[*9]、過去の出版物として以下のようなものがあります:

- "DNSDNS Resolution": DNS 技術のうち、セキュリティ・プライバシーにフォーカスした最新動向を扱った本。セキュアな DNS 技術で自由なインターネットを取

[*8] W3C 会員企業に所属している場合は手続きが異なり、会社の代表者 (AC Rep) に依頼をしてユーザー登録とグループへの参加を申請することになります

[*9] さすがに TPAC から帰って即技術書典はけっこうギャンブル度が高い

り戻そう！

- "Computer Science Head-Start Guide 2018 Edition": 中高生や専門でない学生をメインターゲットにした、コンピュータ科学以前のコンピュータ入門書。
- "バージョン管理 with Pijul": バージョン管理は Git だけじゃない！ 圏論に裏打ちされたつらくないバージョン管理、体験してみませんか？
- "Dark Depths of SMTP": 商業で SMTP の本が出てないのでおそらくこれが最新です。メールサーバーの運用と迷惑メール対策（とその対策）技術を解説しています。

上記の各出版物についてはサークルの Web サイト `https://cryptic-command.net/` をご参照ください。

2.6 謝辞

本記事執筆にあたって、HTTPS in Local Network CG メンバーの皆様にレビューをいただきました。この場をお借りして改めて感謝いたします。

第 3 章

TVM

3.1　Introduction

本項目では、
「TVM っていう単語をたまに聞くけど、何だろう？」
とか、
「Deep Learning の処理を遅いんだよ！ もっと高速化出来ないのか！」
など、思ってる方に向けて、TVM という OSS についての説明をつらつら記載していきます。

TVM のドキュメントやソースコード、論文などを参考にしていますが、誤っている部分があれば、ご指摘いただけると幸いです。

また、本記事のベースを作成した際に参考にした TVM のコミットハッシュは下記です。
（古めのコミット情報となりますが、一部最新情報も記載しています）

- https://github.com/dmlc/tvm/commit/a7c90ee579e0e830e349f372f3783a0bd31ac605

3.2　What is TVM ?

TVM はワシントン大学発の「DNN 向けの最適化を行うフレームワーク（コンパイラ）」である。
Apache-2.0 ライセンスで OSS として、github 上にソースコードが公開されている。
そのため、様々な開発者が上記ソースの修正、機能追加を行っている。
※ DNN = Deep Neural Network の略

複数の種類のフレームワーク（TensorFlow, MXNet, ONNX..）から作成された DNN モデル情報を、複数のチップセット向け（x86_CPU, GPU, ARM_CPU, FGPA..）に最

適化を行うことが可能である。
チップセット向けに最適化を行う際に、AutoTVM を用いて、チューニング処理を行っていることが特徴的である。
※上記チューニングにより、事前に最適なチューニングパラメーターを決定していなくても、自動で最適なパラメーターを検出してくれる。

What is DeepLearning Compiler ?

DNN の学習・推論処理はパフォーマンスを出すために、GPU/TPU などのチップセットを載せるケースが多い。
サーバー側での演算実施を行う場合は前述したスペックを有するマシン構成を構築できるが、エッジ・クライアントサイドでの学習・推論処理を行う場合、CPU のみが搭載されたマシンでの処理が必要になるケースも少なくない。

そこで、TVM を含めた「DeepLearning Compiler」(= DNN 専用のコンパイラ)による最適化処理を行い、低スペックなマシンでの推論処理のパフォーマンスをより向上させる必要がある。

下記のように以前は各フレームワークごとにチップセット向けのバイナリを個別に作成していたが、DeepLearning Compiler を一段追加することにより、最適化処理を各チップセットに出力できるような構成になっている。

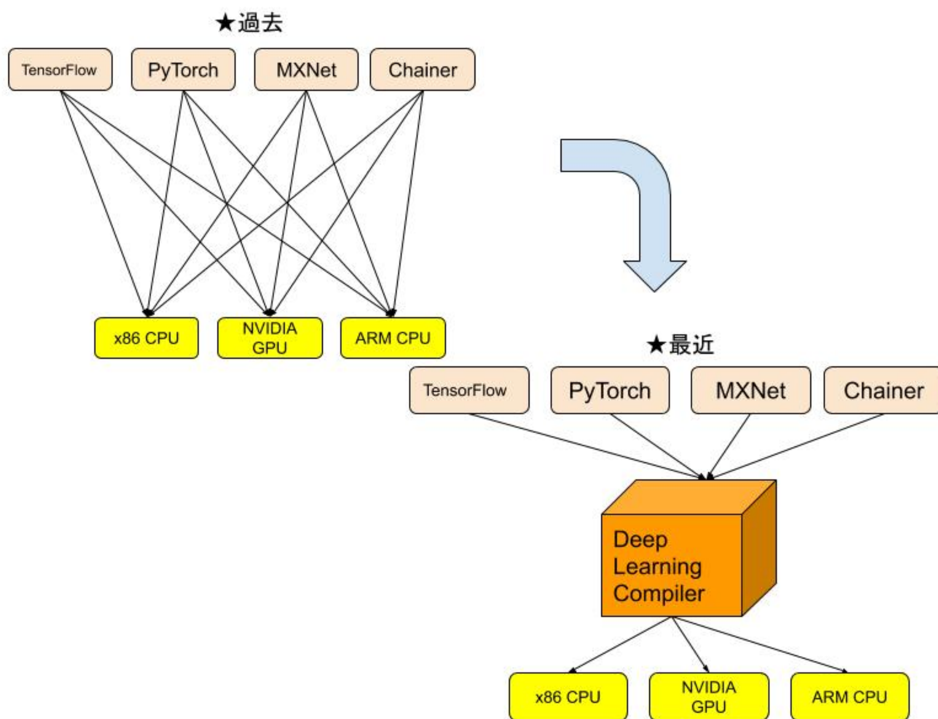

図 3.1: DeepLearning Compiler

What optimization is run in TVM ?

他の DeepLeanring Compiler でも似たような最適化を行なっているが、TVM では主に以下の最適化処理を行なっている。

- High-Level Optimization(グラフ最適化)
  - グラフレベルの最適化処理を実施する。
  - 基本的に HW 非依存な最適化処理となっている。
  - 例） グラフ融合（Graph Fusion)、不必要なノードの削除（delete dead code, pruning)、レイアウト変換
- Low-Level Optimization（primitive な演算最適化）
  - HW に依存した演算レベルでの最適化処理を実施する。
  - HW 依存な最適化処理となる。(TVM では、最適化処理の多くは Halide から由来している）
  - 本部分の最適化を行う際に一部は HW 毎に、AutoTVM でのチューニング処

理を実施することで、更に高速化が可能となる。
  - 例） ループ処理の最適化、ベクトル化（SIMD の利用）、並列処理の実施、レイアウト変換

TVM Infrastructure

TVM の構造はこちら (※ 1) に記載がある。
※以下のような図である。

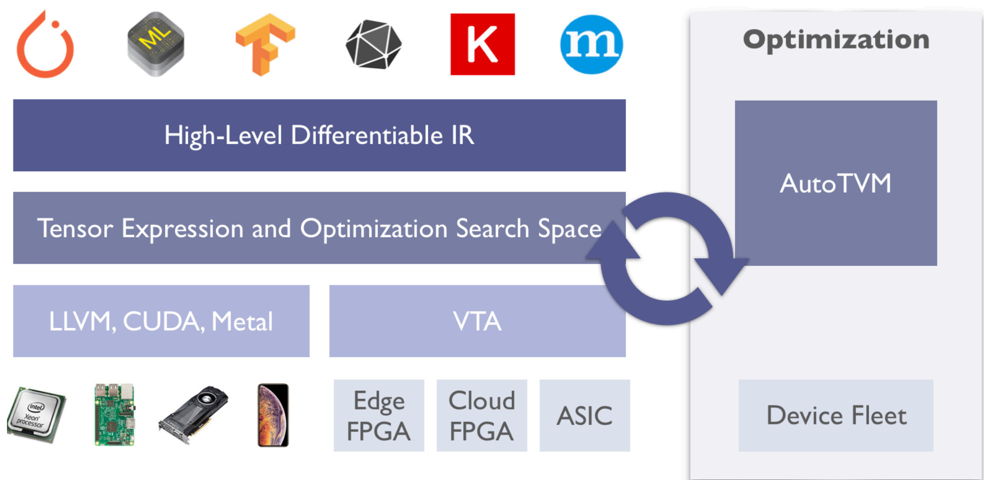

図 3.2: TVM Infrastructure

What kind of IR is used in TVM ?

IR とは Intermediate Representation の略であり、中間表現のことである。
TVM では内部での処理時に以下の二つの IR を利用している。

- High-Level IR
  - グラフ情報を表現する IR のことで、以下の 2 つの IR が利用可能。
  - NNVM（第一世代）
  - Relay（第二世代）
- Low-Level IR
  - 最終的な演算手順を表現する IR のこと。Halide IR(後述) をベースにして、本 IR は構築されている。

- この IR を各チップセット向けの言語や IR に変換する。例えば、x86_CPU であれば、LLVM IR へ変換する。

・IR が二種類ある件については、こちらの FAQ(※ 2) にも記載がある。
High-Level IR について、もう少し下記に詳細を記述する。

- NNVM(第一世代 IR)
  - 従来の DNN フレームワークと同様に計算グラフを表現出来るように、Data Flow 型の表現方法を採用している。
  - 現時点でも NNVM 自体は利用可能だが、TVM の公式サイトのドキュメントからは NNVM のチュートリアルが削除されつつある。
  - Relay が対応しきれていない部分については、まだドキュメントが残っているケースが多い。
- Relay(第二世代 IR)
  - 来の計算グラフの表現方法に加え、Control Flow に関する情報を組み込めるようになっている。(関数型言語のような表現方法となった。)
  - 開発途中であり、NNVM で対応していたフレームワークが一部利用できない。
  - 現状では、NNVM ではなく Relay の利用が推奨されているため、チュートリアルは Relay ベースなものが多い。

What kind of dnn framework is supported ?

TVM では、各種 DeepLearning Framework のモデルフォーマットに対応している。
対応している Framework 一覧を以下に記載する。

- TensorFlow
- Tensorflow Lite
- Keras
- MXNet
- CoreML
- Caffe2
- Darknet
- ONNX

TVM での対応フォーマットを見る場合、High-Level IR ごとに異なるため、詳細を見る場合は、High-Level IR の実装を確認した方が良い。
※下記に各 IR の Framework の変換処理を格納しているディレクトリのリンクを記載

する。

- Relay
  - https://github.com/dmlc/tvm/tree/master/python/tvm/relay/frontend
- NNVM
  - https://github.com/dmlc/tvm/tree/master/nnvm/python/nnvm/frontend

What kind of chipset is supported ?

TVM で対応しているチップセットを下記に記載する。
（対応チップセットは他にも存在する）

- x86 CPU
- NVIDIA GPU
- Mobile GPU
- ARM CPU
- FPGA

どのチップセット向けにコードを作成できるかを見れば、対応チップセットが把握できる。
こちらの codegen のディレクトリ (※ 3) の中を確認すれば良い。

What kind of programming language is supported ?

TVM を利用する際にラッパーとなる API が各言語ごとに用意されている。
言語に応じて、用意されているラッパーのレベルが異なるが、下記の言語がサポートされている。

- Python
  - 実装部分のリンクはこちら（※ 4）
  - TVM では、他の DNN フレームワークと同様に Python のラッパー API が最も充実されている。
  - TVM の公式ドキュメントにも Python API の内容は充実している。
  - API の利用については、チュートリアルが参考になる
- C++
  - TVM の下回り部分の実装は基本的に C++ で実装されている。
  - そのため、直接 TVM の API を呼び出すことで TVM の機能が利用可能である。

- golang
  - 実装部分のリンクはこちら（※ 5）
  - こちらのサンプルコード（※ 6）に mobilenet モデルを読み込んで実行するサンプルがある。
- Rust
  - 実装部分のリンクはこちら（※ 7）
- Java
  - 実装部分のリンクはこちら（※ 8）

Deply

各環境向けに TVM でビルドしたモジュールをデプロイすることも可能。
詳細はこちら (※ 9) を参照してほしい。

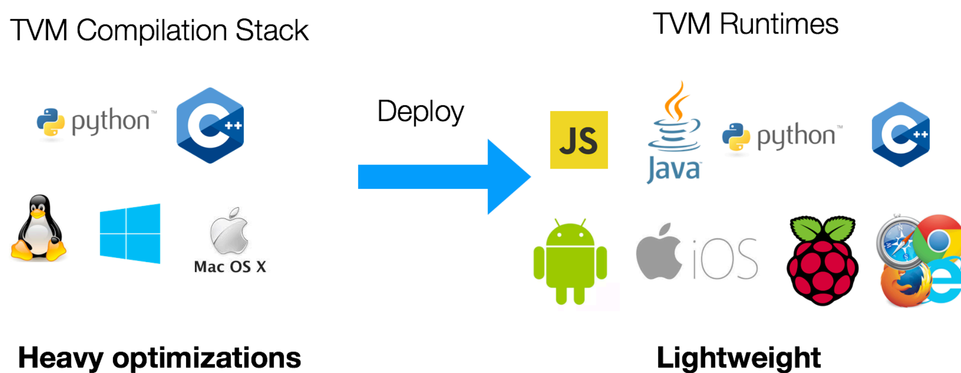

図 3.3: TVM Deply

- TVM Compilation Stack でコンパイル処理（各種最適化処理を実施）を行い、HW に合わせて、Runtime ライブラリを作成する。
- Runtime ライブラリを利用して、各種 HW に合わせて、API を組み込み、推論処理を実施する。

こちら (※ 10) のチュートリアルに記載があるように、ターゲット向けの Runtime ライブラリを作成し、RPC を利用して、ターゲット上で推論処理を実施することが可能。

What is HalideIR ?

TVM の low-level の処理を見る上で、Halide のアルゴリズムの理解がとても大切になるため、概要を記載する。

- TVM では各チップセット向けに最適化したコードを渡す際に Halide ライクな最適化処理を実施しているためである。

- Halide の概要についてはこちら (※ 11) の資料が参考になる。

Halide は「画像処理向けの DSL（Domein Specific Language)」である。最も大きな特徴は「アルゴリズム」と「スケジュール」を分離している点にある。
スケジュールの分離により、マルチプラットフォームの対応を行いやすくなっている。

- アルゴリズム
  - 演算の本質的な処理部分を記載。
  - HW 非依存な記載内容となる。(= HW 毎に共通の記載内容)
- スケジュール
  - 具体的にどのように演算するか？ を記載。
  - HW 依存な記載内容となるため、HW 毎の実装が必要になる。
  - HW 毎の最適化はここで行う。

具体的な例を下記に記載する。
※下記の例は 3*3 のフィルタ（重みが全て 1）を利用して畳み込み、平均プーリングを適用するケースである。

- 通常の書き方

```
void box_filter_3x3(const Image in, Image &blury) {
  Image blurx(in.width(), in.height());

  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)
      blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)
      blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
}
```

```
```

- Halide の書き方

```
Func box_filter_3x3(Func in) {
  Func blurx, blury;
  Var x, y;

  // アルゴリズム部分
  blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
  blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;

  // スケジュール部分
  blury.tile(x, y, xi, yi, 256, 32).vectorize(xi, 8).parallel(y);
  blurx.compute_at(blury, x).store_at(blury, x).vectorize(x, 8);

  return blury;
}
```

書き方を比較してみると、アルゴリズムとスケジュールが分離されていることが分かる。
アルゴリズムとスケジュールが分離されているため、例えば2つのチップセット向けにコードを対応させる場合、以下のようになる。

```
Func box_filter_3x3(Func in) {
  Func blurx, blury;
  Var x, y;

  // アルゴリズム部分
  blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
  blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;

  // マルチプラットフォーム向け（スケジュール部分）
  if (target.has_gpu_feature()) {
    Var tx, ty;
    blury.gpu_tile(x, y, tx, ty, 32, 8);
  } else {
    blury.tile(x, y, xi, yi, 256, 32).vectorize(xi, 8).parallel(y);
    blurx.compute_at(blury, x).store_at(blury, x).vectorize(x, 8);
  }

  return blury;
```

```
}
```

Use Debugger

TVM ではモデル情報のデバッグ機能（= Debugger）が存在する。

デバッガーを利用して推論処理をすると、
「最終的なオペレーター情報（どのように結合されたか？ どのようなレイアウトになったか？」や
「各オペレーターの演算時間」を確認することが出来る。
※具体的な利用手順はこちら (※ 12) を参照して欲しい。

debug 機能を ON にした状態で graph（= TVM でビルドされたモデルバイナリ）を作成することにより、
推論処理時にこちら (※ 13) のようなデバッグ情報が出力される。

本デバッグ機能は以下の点から、使いやすいと考えている。

- どの演算処理で時間がかかっているか？ が分かる点
- どのような最適化（結合処理など）が行われているか？ が分かる点

3.3 Other Deep Learning Compiler

TVM 以外にどのような DNN 向けのコンパイラがあるかを下記に記載する。
※下記は代表的なコンパイラであり、他にも DNN 向けの最適化コンパイラは存在する。
※各最適化コンパイラが別の最適化コンパイラを利用するケースも多くある。（例. OpenVINO では MKL-DNN を利用）

OpenVINO

Intel で開発されている最適化コンパイラ。

様々な DNN 用のフレームワークに対応しているコンパイラ。
Intel 製の CPU で動作を見た場合、爆速で推論処理が実施可能。
バックエンドは、CPU 向けでは MKL-DNN(※ 14)、GPU 向けでは clDNN(※ 15) を利用している。

公式サイトはこちら (※ 16)。リポジトリはこちら (※ 17)。

Tensorflow XLA

Google で開発されている最適化コンパイラ。
Tensorflow の一部機能として提供されている。
公式サイトはこちら (※ 18)。リポジトリはこちら (※ 19)。

Glow(Pytorch)

Facebook で開発されている最適化コンパイラ。
PyTorch の一部の機能として提供されている。
ここ (※ 20) を見ると、Caffe2/PyTorch で使うことを前提としていそうで、他の Framework からは使えないと思われる。
公式サイトはこちら (※ 21)。リポジトリはこちら (※ 22)。

ONNXRuntime

Microsoft で開発されている最適化コンパイラ。
バックエンドとしては、CPU 向けに MKL-DNN/nGraph が利用でき、GPU 向けには NVIDIA TensorRT が利用できる。
ONNX のみの対応で、IF としては Python、C#、C++ が提供されているが、Python がメイン。
リポジトリはこちら (※ 23)。

3.4 Optimization Detail

TVM では前述したように二つの最適化処理を行なっている。
本項ではもう少し最適化処理の内容を細かく確認していく。

High-Level Optimization

DNN のモデル自体を改変し、最適化を行っている。
具体的な最適化処理を以下に記載する。

- High-Level な最適化については、IR（NNVM or Relay）によって、微妙に異なっている。

- Relay・NNVM 共に build_module.py の build 関数の処理を追えば、最適化処理の内容が分かる。

最適化レベル（= 0 から 3）に応じて、実施される最適化処理が異なる。
0 があまり最適化されず、3 が最も最適化されている。
以下の表にどの最適化が、どの最適化レベルで実施されるかをまとめた。
※各最適化の詳細は後述する。

最適化名	対応 IR	最適化レベル
SimplifyInference	Relay, NNVM	0
OpFusion	Relay, NNVM	1
PrecomputePrune	NNVM	2
FoldConstant	Relay	2
FoldScaleAxis	Relay, NNVM	3
AlterOpLayout	Relay, NNVM	3
CombineParallelConv2D	Relay	3
CanonicalizeOps	Relay	3
EliminateCommonSubexpr	Relay	3

SimplifyInference

通常の推論処理とあまり変わらないが、以下の処理を追加で実行している。

- Dropout レイヤーの削除（推論時は不要）
- Batch Normalization レイヤーを [scale × data + shift] の演算に変換

こちら (※ 24) のテストコードが参考になる。

OpFusion

Operator 同士を結合する処理を行い、演算の高速化を図っている。
オペレーター融合についてはいくつかルール（オペレーターの種別によって、融合方法が変わる）があるため、下記に詳細を記載する。
(NNVM と Relay では実装箇所が異なる)

■実装箇所

- Relay : fuse_ops.cc(※ 25)
- NNVM : graph_fuse.cc(※ 26)

■オペレーター種別　各オペレーター毎に、大雑把に種別が設定されている。
NNVM、Relay でオペレーター種別が微妙に異なっている。

- NNVM の定義はこちら (※ 27)
- Relay はこちら (※ 28)

OperatorPattern	Operator (example)
Elementwise	Relu
Injective	flatten
Opaque	softmax、lrn
Complex Oprater（OUT_ELEMWISE_FUSABLE）	Conv、pooling、matmul、dense
Communication（COMM_REDUCE）	Sum
Broadcast	broadcast_add、elemwise_mul

■融合するためのルール　前述したオペレーター種別に応じて、融合するためのルールが決まっている。
ただし、NNVM と Relay で融合処理が異なっており、下記に NNVM での融合処理をまとめる。

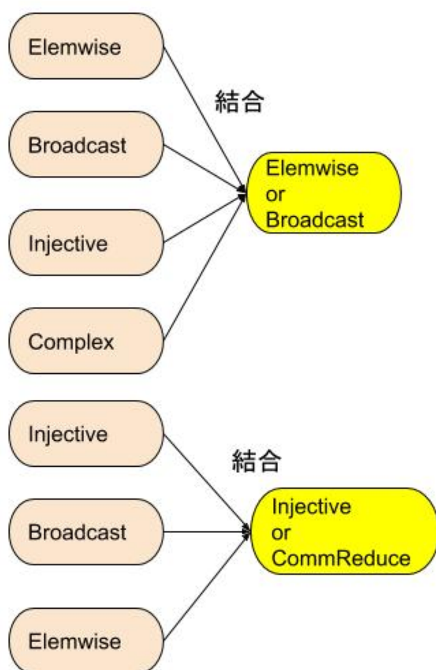

図 3.4: NNVM Operator fusion pattern

PrecomputePrune

OpFusion を行う前の前処理を行っている。

- 特定のノードを削除したりしているようだが、詳細は不明。
- Python 側のコードはこちら (※ 29)。C++ 側のコードはこちら (※ 30)。(Python 側から C++ 側のコードを呼び出している)

FoldConstant

定数項をまとめることにより、演算処理をシンプルにし、最適化をしやすくする。

- こちら (※ 31) の資料が分かりやすい。
- fold_constant のテストコード (※ 32) をみると、イメージが付きやすい。

FoldScaleAxis

Scale（乗算処理）を畳み込み対応。
Scale を行う順番を変更したりして、演算回数を減らすのが目的。

乗算処理（element-wise）において、対象となる Tensor のサイズが小さい方が効率が良いため、conv2d/dense を含むケースで本処理は効力を発揮する。

AlterOpLayout

Operator の形状を変換する対応。(NCHW -> NCHWc にしたりなど)

HW に合わせた形状変換を行うことにより、高速化が図れる。
本処理については Low-Level Optimize 的な処理といえる。

(TODO：詳細を追記)

CombineParallelConv2D

同一形状（入力形状、カーネルの高さ・幅・入力チャネル数が同一）の convolution2d が並列に合った場合、一つの convolution2d として表現（= 結合）する対応。
※結果を処理する場合は、出力値を slice することにより可能。

まとめて演算することにより、高速化が見込める可能性がある。
こちらのテストコード (※ 33) が参考になる

CanonicalizeOps

複雑なオペレーターをシンプルなオペレーター同士に分割したりして、シンプル化する処理。

- 現状は Bias_Add を Expand Dims + Broadcast Add に変換する処理のみが入っている。

- コードはこちら (※ 34)。テストコード (※ 35) も参考になる。

EliminateCommonSubexpr

全く同一の演算処理が並列に走っている場合、共通化して演算処理を減らす処理。

- テストコード (※ 36) が分かりやすい。

- こちら (※ 37) の Tensorflow の最適化処理の説明資料も分かりやすい

Low-Level Optimization(CPU)

TVM ではチップセットに応じて、チューニング処理が変わるが、本資料では CPU 向けでの最適化内容についてまとめていく。
こちら (※ 38) のチュートリアルが参考になる。

CPU 向けの最適化では以下の 2 点を実現するために各種最適化を行なっている。

- キャッシュヒット率の向上
- SIMD 命令を利用した演算の高速化

上記を実現するために下記のような処理を行なっている。
（詳細は後述する）

- Blocking（tiling）
- Vectorization
- Loop Permutation
- Array Packing
- Write cache for blocks
- Parallel

Blocking(tiling)

データを別のグループ（配列）に分割して、処理を実施する。

CPU アーキテクチャで保持するキャッシュメモリに合わせて、ブロックサイズを指定することにより、キャッシュヒット率が向上し、データの読み書きが高速になる。
（NCHW -> NCHWc の対応とは別。）

Vectorization

ベクトル演算（SIMD 命令）が出来るようにするために、指定したブロック（通常はループの最も内側）をベクトル化する。

上記により、SIMD 命令（single instruction multiple data：一つの命令で複数データをまとめて処理）が実行でき、高速化に繋がる。

Loop Permutation

ループの処理順序を変える。
（例えば、abcd という配列があった場合、bcad の要素順序でループを回す）

上記により、CPU のキャッシュメモリサイズに応じて、実行順序を変えることにより、キャッシュヒット率が向上する。

Write cache for blocks

ブロックに分割した際にブロックごとの処理を最終的にまとめる必要があるが、各ブロックの結果を別メモリに書き込むとシーケンシャル（連番）ではなくなる。（アクセスが遅くなる）
そのため、キャッシュに書き込み、シーケンシャルな順序にしてしまう。

Parallel

指定したブロックの処理を並列に実行させる。
並列実行により、処理速度の向上が見込める。

3.5 AutoTVM

ここでは TVM の目玉機能である「AutoTVM」について、情報をまとめる。

What is AutoTVM ?

AutoTVM は「チップセット毎に推論処理を最も高速化させるパラメーターを自動で検出する機能（= Tuning)」を持つ。

つまり、前述した Low-Level Optimize に関連していると言える。

具体的には以下の機能（= 最適なパラメーターを検索する機能）を持つ。

- Local Search
  - モデルのオペレーター毎に最適なパフォーマンスを行えるパラメーターを検索する。

  - データのレイアウト変換を行うため、Layout Transform オペレーターが追加される。
- Global Search
  - モデル全体（= graph）で最も最適なパフォーマンスになるパラメーターを検索する。
  - Local Search で検出したデータレイアウトを元に、最終的にどのようなデータレイアウトで処理をするのが最適化を決定する。

各種 Tuning 処理自体はターゲットとなる端末で実行する必要がある。
どのパラメーターパターンが最適か？ を実測して（= コスト計算）決定するためである。

Local Search

モデルに含まれる各オペレーター毎に最適なパラメーター（= config space）を検索する。

Optimize Operator by Local Search

Local Search により、オペレーター毎に最適なパラメーター（tiling, loop, unroll..etc）を検出し、グラフに適用した場合、以下のような変換処理となる。
データの配列が変わるため、レイアウト変換を行うオペレーター（= Layout Transform）を追加する必要がある。（下記の図は TVM の論文 (※ 39) である Figure2 から抜粋）

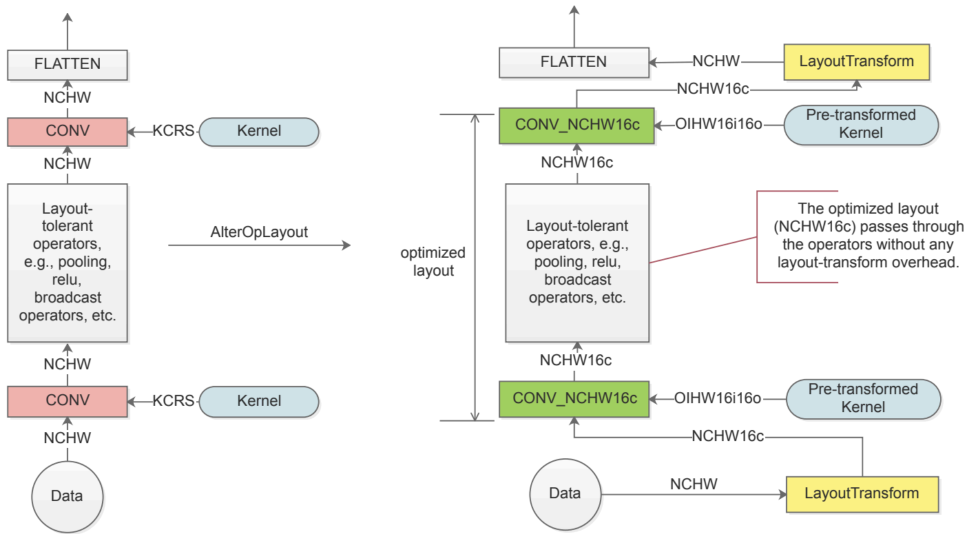

図 3.5: Simple Convolution Transform

What is Config Space ?

Config Space は各オペレーターにおいて、「チューニング対象となりうるパターンをまとめたデータ」と言える。

例えば、「行列の積」を求めるオペレーターについて、具体的に config_space を考えてみる。※下記の例はこちら (※ 40) のチュートリアルより抜粋。

```
@autotvm.template
def matmul(N, L, M, dtype):
    A = tvm.placeholder((N, L), name='A', dtype=dtype)
    B = tvm.placeholder((L, M), name='B', dtype=dtype)

    k = tvm.reduce_axis((0, L), name='k')
    C = tvm.compute((N, M), lambda i, j: tvm.sum(A[i, k] * B[k, j], axis=k),
        name='C')
    s = tvm.create_schedule(C.op)

    # schedule
    y, x = s[C].op.axis
    k = s[C].op.reduce_axis[0]

    ##### define space begin #####
    cfg = autotvm.get_config()
```

```
    cfg.define_split("tile_y", y, num_outputs=2)
    cfg.define_split("tile_x", x, num_outputs=2)
    ##### define space end #####

    # schedule according to config
    yo, yi = cfg["tile_y"].apply(s, C, y)
    xo, xi = cfg["tile_x"].apply(s, C, x)

    s[C].reorder(yo, xo, k, yi, xi)

    return s, [A, B, C]
```

define_split は Blocking を行う際に「どのように分割するか？」をパラメーターとしてチューニングしたい際に指定する。
num_outputs は出力する値の数となっている。（例．3,12 など）

ここで仮に N, L, M = 512 とすると、tile_y, tile_x のパターン数はそれぞれ以下となる。

- tile_x：10 パターン
  - (1, 512), (2, 256), (4, 128), (8, 64), (16, 32) ..
- tile_y：10 パターン
  - (1, 512), (2, 256), (4, 128), (8, 64), (16, 32) ..

よって、Config Space のトータルのパターン数（= length）は、10 × 10 = 100 パターンとなる。
（この 100 というのは、パターンの上限数（= 全パターン）となる。）

上記の例のように、各オペレーター・次元数毎に網羅されるパターン数が確定される。

また、ターゲットとなるチップセット（CPU、GPU、ARM CPU など）に応じて、Config Space の作り方は異なる。

Task

モデルの各オペレーター毎に最適なパラメーター検索が Local Search であるが、1 オペレーターで検索する処理を「Task」として扱っている。（AutoTVM 内部で、Task と呼んでいる）

Convolution、Dense（行列の乗算処理）など、演算処理に応じて、タスクが異なる。

- Config Space はパターン数を表すため、パターン数が多ければ、それだけチューニングにかかる時間も増大する。
- Task の数 = チューニング対象となっているオペレーターの数

Target Operator

Local Search でチューニング対象となるオペレーターを下記に記載する。

- convolution2d
- deconvolution2d
- dense

Tuner

どのパラメーターからチューニングを実施するかを各種 Tuner を用いて決定している。探索アルゴリズムとも言えるが、全探索をする場合は最終的なパラメーター探索結果は同一になる。
early stopping を指定している場合は、検知するパラメーターに差が出る可能性がある。

- Random
- GridSearch
- GA（genetic algorithm）
- XGBoost

ここのチュートリアル (※ 41) を見ると、チューニングパターン数（= Config Space の長さ）が 10^9（10 億パターン）を超える場合は XGBoost を Tuner として指定するのが良いとされている。（効率的に最適なパターンを見つけてくれる）

Global Search

モデル全体（= graph）で最も最適なパフォーマンスになるパラメーターを検索する。

具体的にどのような検索処理をやっているかは、こちら (※ 42) の RFC や TVM の論文 (※ 39) の [3.3.2 Global Search] を参考にしてほしい。

Adjust Data Layout

Local Search で各オペレーター毎に最適なレイアウトを検出するが、グラフ全体で見ると、本当に最適なレイアウトとは限らない。

つまり、Layout Transform を実際にはどの程度行うか？ を全体の処理速度をみて、検討することになる。

処理のイメージ図は下記。
(下記の図は TVM の論文 (※ 39) である Figure3 より抜粋)

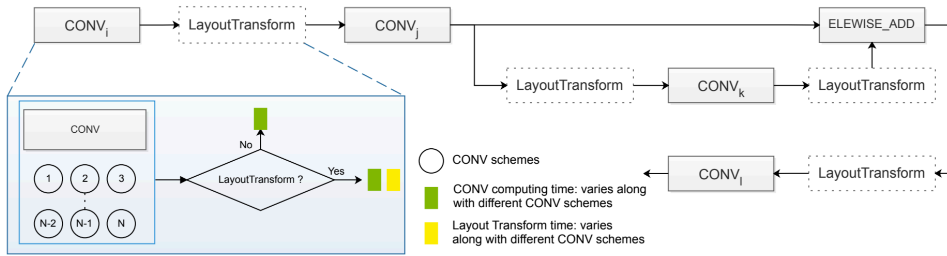

図 3.6: TVM Global Search

Tuner

以下の Tuner を利用して、グラフレベルの最適化構造を見つける。

- DPTuner
- PBQPTuner

3.6 Tutorials

TVM を試しに使ってみたい場合、まずはチュートリアルを試すことをお勧めする。
https://docs.tvm.ai/tutorials/ がチュートリアルのトップページになる。
下記にいくつかチュートリアルの情報を記載する。

Compile Models

ONNX や Tensorflow のモデルフォーマットなどから、TVM にモデルを読み込ませる事が可能。
例えば、ONNX であれば、こちらのコード（※ 43）を参考にすれば良い。

Deply Android

Android へのデプロイを行うチュートリアルもある。
https://docs.tvm.ai/tutorials/frontend/deploy_model_on_android.html

3.7 Reference

- TVM Document

  - https://docs.tvm.ai/index.html
- TVM Source Code
  - https://github.com/dmlc/tvm

3.8 Link

- ※ 1：TVM about
  - https://tvm.ai/about
- ※ 2：TVM FAQ
  - https://docs.tvm.ai/faq.html
- ※ 3：Codegen directory
  - https://github.com/dmlc/tvm/tree/master/src/codegen
- ※ 4：Python API directory
  - https://github.com/dmlc/tvm/tree/master/python/tvm
- ※ 5：golang API directory
  - https://github.com/dmlc/tvm/tree/master/golang
- ※ 6：golang sample code
  - https://github.com/dmlc/tvm/blob/master/golang/sample/complex.go
- ※ 7：Rust API directory
  - https://github.com/dmlc/tvm/tree/master/rust/frontend
- ※ 8：Java API directory
  - https://github.com/dmlc/tvm/tree/master/jvm
- ※ 9：TVM Deploy information
  - https://docs.tvm.ai/deploy/index.html
- ※ 10：cross compile and rpc
  - https://docs.tvm.ai/tutorials/cross_compilation_and_rpc.html
- ※ 11：「Halide による画像処理プログラミング入門」
  - https://www.slideshare.net/fixstars/halide-82788728
- ※ 12：TVM Debugger
  - https://docs.tvm.ai/dev/debugger.html
- ※ 13：Debugger sample output
  - https://docs.tvm.ai/dev/debugger.html#sample-output
- ※ 14：MKL-DNN
  - https://github.com/intel/mkl-dnn
- ※ 15：clDNN

  - https://github.com/intel/clDNN
- ※ 16 : OpenVINO toolkit
  - https://software.intel.com/en-us/openvino-toolkit
- ※ 17 : OpenVINO github repository
  - https://github.com/opencv/dldt
- ※ 18 : Tensorflow XLA
  - https://www.tensorflow.org/xla
- ※ 19 : Tensorflow XLA github repository
  - https://github.com/tensorflow/tensorflow/tree/master/tensorflow/compiler/xla
- ※ 20 : Glow onnx IF document
  - https://github.com/pytorch/glow/blob/39491b588d7a986e44a298045267b64fd2ce73a8/docs/Onnxifi.md
- ※ 21 : Glow
  - https://ai.facebook.com/tools/glow
- ※ 22 : Glow github repository
  - https://github.com/pytorch/glow
- ※ 23 : ONNXRuntime github repository
  - https://github.com/Microsoft/onnxruntime
- ※ 24 : test_pass_simplify_inference
  - https://github.com/dmlc/tvm/blob/e470f8eaa28daa27a1bd15d38150bafe4f1e5a96/tests/python/relay/test_pass_simplify_inference.py#L4-L43
- ※ 25 : relay fuse_ops
  - https://github.com/dmlc/tvm/blob/eae76b3c3b189197f79b79fcccd6a2348640e6a0/src/relay/pass/fuse_ops.cc#L612-L683
- ※ 26 : nnvm graph_fuse
  - https://github.com/dmlc/tvm/blob/master/nnvm/src/compiler/graph_fuse.cc
- ※ 27 : nnvm operator attribute types
  - https://github.com/dmlc/tvm/blob/cffb4fba03ea582417e2630bd163bca773756af6/nnvm/include/nnvm/compiler/op_attr_types.h#L48-L65
- ※ 28 : relay operator attribute types
  - https://github.com/dmlc/tvm/blob/eef35a57d95c650e490b168f1f585d9ec00412ee/include/tvm/relay/op_attr_types.h#L36-L57
- ※ 29 : nnvm build module
  - https://github.com/dmlc/tvm/blob/master/nnvm/python/nnvm/compiler

/build_module.py#L389-L424

- ※ 30：nnvm precompute_prune
  - https://github.com/dmlc/tvm/blob/f721a645cd77661a8284559d676ef20f91eaa771/nnvm/src/compiler/precompute_prune.cc
- ※ 31：ウェブブラウザ向け深層学習モデル高速実行フレームワーク「WebDNN」
  - https://speakerdeck.com/kiikurage/uebuburauzaxiang-keshen-ceng-xue-xi-moderugao-su-shi-xing-huremuwaku-webdnn?slide=30
- ※ 32：relay test_pass_fold_constant
  - https://github.com/dmlc/tvm/blob/ee8058069a41a80845b952a371c5beb2570f6d24/tests/python/relay/test_pass_fold_constant.py#L6-L31
- ※ 33：relay test_pass_combine_parallel_conv2d
  - https://github.com/dmlc/tvm/blob/990521dd441c3ef064d2e4302b83f050c8b9236b/tests/python/relay/test_pass_combine_parallel_conv2d.py#L5-L49
- ※ 34：relay canonicalize_ops
  - https://github.com/dmlc/tvm/blob/97ca4031f3ec0563baa3a974b2f39d651592fdea/src/relay/pass/canonicalize_ops.cc
- ※ 35：relay test_pass_alter_op_layout
  - https://github.com/dmlc/tvm/blob/f81e2873d15a87097e34b2081186636e552e279a/tests/python/relay/test_pass_alter_op_layout.py#L263-L310
- ※ 36：relay test_pass_eliminate_common_subexpr
  - https://github.com/dmlc/tvm/blob/1ca0393e828c2c6e32d9da0beb87582df9158fc6/tests/python/relay/test_pass_eliminate_common_subexpr.py#L7-L28
- ※ 37：Tensorflow グラフ最適化処理
  - https://www.slideshare.net/ssuser5f01c2/tensorflow-134544864
- ※ 38：TVM Tutorials optimize gemm
  - https://docs.tvm.ai/tutorials/optimize/opt_gemm.html#sphx-glr-tutorials-optimize-opt-gemm-py
- ※ 39：TVM paper
  - https://arxiv.org/pdf/1809.02697.pdf
- ※ 40：AutoTVM use-better-space-definition-api
  - https://docs.tvm.ai/tutorials/autotvm/tune_simple_template.html#use-better-space-definition-api
- ※ 41：AutoTVM auto-tuners-in-tvm

  - https://docs.tvm.ai/tutorials/autotvm/tune_simple_template.html#auto-tuners-in-tvm
- ※ 42：GraphTuning RFC
  - https://github.com/dmlc/tvm/issues/1585
- ※ 43：TVM ONNX Tutorials
  - https://docs.tvm.ai/tutorials/frontend/from_onnx.html#sphx-glr-tutorials-frontend-from-onnx-py

3.9 Author

- 著者名：坂本 将太
- Twitter：https://twitter.com/kurama554101
- Github：https://github.com/kurama554101

第 4 章

Web Frontend Boilerplate Battle

はじめまして! @diescake と申します。本書を手に取っていただきありがとうございます。

私は株式会社 ACCESS でウェブフロントエンドエンジニアとして活動していて、最近はもっぱら React + Redux + TypeScript でウェブアプリケーションの開発を中心に行っています。一方、趣味では、大喜利エンジニア (笑) などと自称していて、新しいガジェットやサービスが登場するとすぐに飛びつき、斜め上の活用方法を模索したりしています。技術的な話よりも、くだらないことをツイートしている頻度が高いのですが、もしよろしければ Twitter フォローして貰えると嬉しいです!

図 4.1: diescake の Twitter アカウント

さて、大喜利エンジニアなどと名乗りを上げた後で何ではあるが、今回はウェブフロントエンドにおける boilerplate という真面目な題材を取り上げる。個人開発ではあるが、ACCESS の商用案件でも利用している React ベースの boilerplate である igata を紹介する。

igata の紹介を通じて、ウェブフロントエンドの各要素で導入している技術とその選定

理由、運用する際の知見やノウハウなど、共有していければと思う。

また、私は比較的 ACCESS のウェブフロントエンドの分野を横断してウォッチしている立場ではあるが、結局のところ、現場の 1 人のエンジニアでしかないため、ACCESS に関する話は、あくまで、私個人の意見・感想であることに注意いただきたい。

4.1 私を取り巻くウェブフロントエンド事情

現在、私は IoT 事業部の中で主に B2B 向けの請負開発を行う部署に所属している。従って、開発するウェブアプリケーションも IoT ソリューションの一部として提供されるものであることが多い。具体的には、IoT 機器を取り扱い監視するための管理画面やダッシュボードを SPA(Single Page Application) で新規構築したり、既存のウェブアプリケーションをリプレイスするといった内容で非常に引き合いが多い。

一方、引き合いの多さに対して、ウェブフロントエンドエンジニアは不足している状況にある。それは、ACCESS に限った話ではなく、友人や懇親会で情報交換をする限りでは、他社でも同様の傾向があり、買い手市場の中でも特にウェブフロントエンドエンジニアは採用が難しいと話す人もいる。

また、ウェブフロントエンドという分野は歴史が浅い。日進月歩であり、新しい技術、ライブラリ、ツール、サービス、概念、果てはプログラミング言語までが登場し、プラットフォームであるブラウザ自体も大きく進化を続けている。そのため、一口にウェブフロントエンドの開発経験があるからといって、現場で採用する技術に対してノウハウを持ち、即戦力になるかといえばそうとも限らない。(勿論、他分野において優秀なエンジニアはすぐにキャッチアップし即戦力となるが、それはそれで希少な人材である)

従って、請負開発という観点では、非常に仕事が多い分野ではあるものの、慢性的にエンジニアが不足していて、かつ、性質上、協力会社のエンジニアにも頼りにくい。そんな状況である。

4.2 boilerplate 実装という大きな壁

とはいえ、引き合いがある以上、会社は仕事を取ってくる。そうすると、ウェブフロントエンド開発経験者が極端に少ないチームが結成されることもある。こういった開発体制では、まず最初に大きな壁にぶつかることになる。それが boilerplate (ボイラープレート) の実装である。

boilerplate という言葉は聞き慣れない人もいるかもしれない。これは、いわゆる開発を開始するためのベースとなるソースコードや、ディレクトリ構成、ツール類が整備された環境のことである。明確な定義があるわけではないが、一般的にビルドの設定や、フレームワーク、各種ライブラリ、静的解析ツール類が導入済みで、実装の参考となるサンプルアプリケーションが動作する状態であることが多い。

こういった boilerplate が必要なのは、ウェブフロントエンド開発に限った話ではない。他分野においてもアーキテクトが適切な設計を行い、相応しい環境を事前に準備する必要があるだろう。しかし、ウェブフロントエンドほど選択肢が多岐に渡ることは稀だ。先に述べた通り、言語、フレームワーク、ライブラリといった選択肢が非常に豊富であり、刻一刻とベストプラクティスも変化している。さらに、選択したそれらが適切に機能するように、ビルド関連の設定にも気を払う必要がある。実際、未経験でいきなりこの作業を行うことは不可能だろう。

驚くほどの余談であるが、ダークソウル 3 ではゲーム冒頭、チュートリアルに相当するエリアの締めとして「灰の審判者グンダ」というボスが登場する。このボスは前作をクリアしている筆者でも心が折れそうになる強さで、多くのプレイヤーを篩いにかけた。大抵のゲームでは、攻略が難しいと感じたら、一旦撤退してレベルを上げたり、装備を新調して再挑戦するといった手段があるだろう。また、一旦諦めて他のルートへ進むというのも良い。しかし、このボス戦はあくまでチュートリアルの一貫なので、そういった選択肢は用意されていない。繰り替えし挑戦し、パターンを覚えて、プレイヤースキルを上げて攻略するしかないのだ。

ウェブフロントエンドで boilerplate の話をするときは、そんなダークソウル 3 のチュートリアルのことを思い出す。

さて、話を戻すと、自力で boilerplate を構築せずとも、先人が公開している boilerplate は多数存在するため、その恩恵に与るという方法もあるし、実際多くのエンジニアはそうするだろう。しかし、偶然目に留まったそれが十分に保守されているとは限らないし、クオリティもピンキリであり商用利用するにはリスクも大きい。自分の希望通りの boilerplate が見つかることは稀だろう。

ここで、本格的な開発が始まる前に boilerplate として準備しておきたい項目を簡単にあげておく。1 つ 1 つが重要な要素であり、開発効率や品質に直結するだろう。

- 言語、フレームワークの選択
- コンパイラの導入とビルド設定 (dev/prod)
- ディレクトリ構成の決定
- 各種ライブラリの選択とサンプル実装
- テストツールの選択とサンプル実装
- スタイルの実装手段の選択とサンプル実装
- 静的解析ツール、フォーマッタの導入とルール設定
- デバッグ用ローカルサーバ準備
- コーディング規約、PR テンプレートの準備
- etc etc etc……

ところで、React の場合であれば、Facebook 公式が提供する create-react-app という CLI ツールがある。いくつかのコマンドを実行するだけで、簡単にベースコードを生成できる便利な代物であるが、生成されるコードについては賛否があるため、後の項目で触れる。

4.3 Introducing "igata"

図 4.2: igata のロゴ

- diescake/igata
  https://github.com/diescake/igata

そこで、私が個人で開発している React ベースの boilerplate である igata を紹介する。話の流れから igata はこれらの問題をスマートに解決するエレガントなソリューションのように思えるかもしれないが何のことはない。有象無象に公開されている boilerplate の 1 つである。

単に、私がゼロベースで実装した経験から、ノウハウを共有するための題材としてちょうど良かったということと、ACCESS 社内でも利用されていることから、社名を掲げて

頒布する本の内容としても悪くないと思ったのである。

由来

ところで igata という名称の由来は以下の通りである。

```
'boilerplate'
|> toJapanese // 鋳型
|> toAlphabet // igata
```

お分かりいただけたであろうか。単に boilerplate は和訳しただけである。pipeline operator は未だに stage-1 proposal で進捗を見せていないが、私が最も導入を心待ちにしている仕様の 1 つだ。

- tc39/proposal-pipeline-operator
  https://github.com/tc39/proposal-pipeline-operator

元々の開発動機

もう少し自分語りが続く。igata はウェブフロントエンド関連のライブラリやツールを試すための自分用の boilerplate を目的として作りはじめた。情報収集の折に、ふと面白そうな技術情報を見かけても、それを試すための環境がないと気軽に実験ができないからだ。この際に、ベースコードの準備から入ると深遠な yak shaving に嵌りがちで、それはそれで勉強にはなるが、当初の目的が達成されないということがしばしばあった。

同時に「ぜんぜんわからない。俺は雰囲気で webpack を書いている」という状態も問題だと思っていたため、ゼロベースで boilerplate を実装することで、これらベースレイヤーの処理に関する理解が深まることも期待していた。

ACCESS 社内案件への転用

一方、仕事では自分が担当する案件とは別に、主に部内を横断して、ウェブフロントエンド関連案件の上流レビューをする機会に恵まれ、この領域の経験者不足による辛みも感じていた。こういった案件では、フォローのために中途半端にコードレビューだけ参画しても頭を抱えることが多く、経験不足をカバーできる boilerplate の仕組みとサンプル実装が非常に重要であることを感じた。

そこで、自分用にメンテを続けていた igata を、社内向けにも活用できるのではと思い

社内勉強会やヒアリングを受けた際に紹介。現在、ACCESS のいくつかの商用案件で導入され、開発が進行している。

4.4 igata の設計思想

続いて、設計思想というほど高尚な話ではないのだが、igata を実装、メンテナンスする上で意識している特徴を挙げる。

メジャーで無難、リーズナブルな選択を

boilerplate としては特に先進的な仕組みや、チャレンジングな機能を取り入れたりしない。

ライブラリに関しては、選択肢が非常に豊富であるため、私の好みも多分に反映されてはいるが、ググれば情報が潤沢に手に入るメジャーなライブラリを中心に選択している。何かトラブルがあったときに、GitHub の issue を覗いたり、エラーコードをググることで有用な情報にリーチしやすくなるため、非常に重要である。

また、あらゆる問題解決をライブラリに任せ、実装をブラックボックス化してしまうのは良いことではない。これは明確な線引がない上に個人差も強く加減が難しいのだが、素朴に実装して済むことであれば、その方がより簡潔で見通しの良いコードになるだろう。

ライブラリは常に最新に追随

依存ライブラリ、具体的には package.json の dependencies と devDependencies は常に最新に追随している。

これは boilerplate として最も重要な要素の 1 つであるが、同時にウェブ上に公開されている多くの boilerplate が実現できていない点でもある。サービス運用フェーズでは、闇雲にライブラリをアップデートすることは好ましくないが、開発開始時点で古くなっていることは大きな損失だ。特に機能面の改良だけではなく、検出された脆弱性が修正されている場合もあるので注意が必要だ。

ちなみに、igata では Dependabot により継続的なアップデートを取り込んでいる。

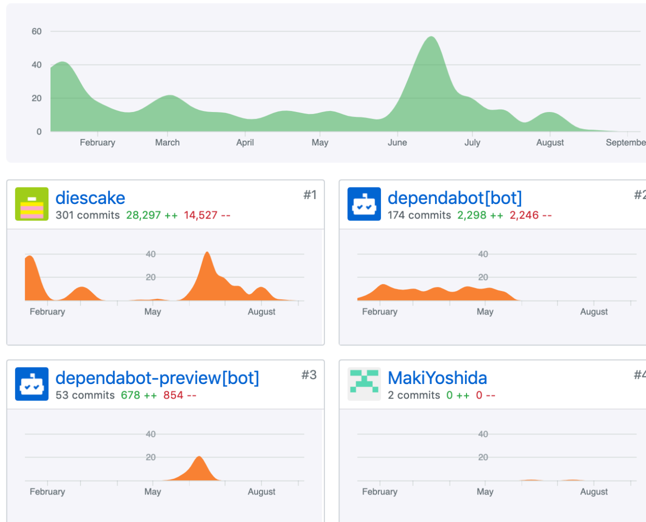

図 4.3: Dependabot

ご覧の通り igata の 4 割近いコミットは彼の功績であり、bot なしでの保守は考えられない。今年の春に Dependabot が GitHub 社に買収されて以来、プライベートリポジトリでも無償で利用可能となったため、社内リポジトリでも活用している。

現在開発中の案件では、ウォータフォールに近い開発スタイルであるため、システムテスト直前までは Dependabot で継続的に最新ライブラリに追随し、システムテストに入る直前でライブラリのバージョンを一旦フリーズ。以降は脆弱性対応の取り込みのみに絞る運用を考えている。

汎用的なサンプル実装を提供する

これは若干 boilerplate の範疇を超えるものであるが、ウェブアプリケーションを開発する上で大抵実装が必要になる機能というものがある。例えば、ログイン、セッション管理、プログレスバー、モーダルダイアログ、API 周りのデータフロー、エラー処理 (準備中) といった機能や仕組みで、そのまま利用可能なサンプル実装を提供している。これらのサンプル実装は後で紹介する。

create-react-app を利用しない

boilerplate の説明で触れた create-react-app について、私の関わる案件や igata では利用しない方針を取っている。

正直にいえば create-react-app と真直ぐ向き合い、より良く使う方法を模索したこと

があるわけではなく、なぜ create-react-app で作られたプロジェクトではダメなのか？と言われると論理的な説明に苦しい。実際、社外では商用案件で利用しているという話もたまに聞く。

ただ、create-react-app によって生成されたプロジェクトを eject した際の、package.json と webpack のあまりに混沌とした様に耐えられなかった。不要な依存ライブラリや webpack の設定も多く、ブラックボックス要素が強いまま扱うことの是非も気になった。個々のライブラリ単体で見ればシンプルな仕様であっても、create-react-app としてまとめられてしまうと、トラブルシューティングのために調査すべき情報が増えるし、影響範囲も大きくなる。

また、仕事に適用する際の懸念としては、請負開発では顧客の要件に対して技術的に課題があったとしても調整が難しい場合があり、小回りが効かないというのはそれ自体がリスクであるという事情がある。

実際のところ boilerplate という問題を攻略する上で create-react-app ってどうなのよ？　という話はとても興味がある。詳しい人がいたらぜひ話を聞かせて欲しい。

igata の変更を継続的に取り込む方法は考慮しない

igata をベースとして新しく作られたリポジトリから、igata の変更を継続的に取り込む方法は考慮しない。厳密には、取り込めれば良いと思っているが、スマートな解決法がなく断念しているという説明が正しい。もしこれを実現するのであれば、劣化版のオレオレ create-react-app に近い形になってしまうだろうか？　と考えたことはあるが、いずれにしても igata の開発において、派生先リポジトリを考慮しつつ実装を進めることは避けたいという気持ち故の方針である。

幸い、請負開発の案件では、非常に短期で軽量な開発である PoC(Proof of Concept) と呼ばれる実証実験であったり、納品後のメンテナンスは顧客側で実施するというケースも多く、継続的な機能追加や、パッケージのバージョンアップが求められることは少ない。この辺りは、自社サービスとは異なる、請負開発特有の事情とも言えるかもしれない。

また、igata をベースに新しくリポジトリを作る際は、GitHub にはテンプレートという機能を利用する。

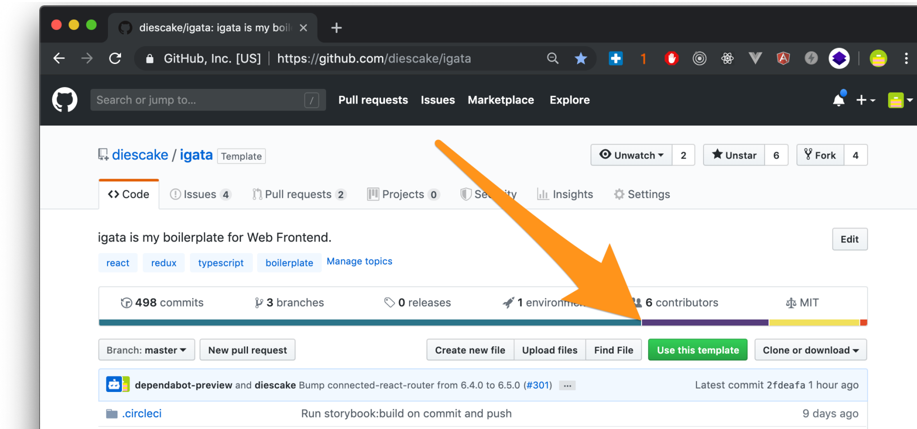

図 4.4: GitHub の template 機能

この機能によって、とあるリポジトリをテンプレートとして新たにリポジトリを作成できる。

テンプレートとフォークの違いについては、公式ドキュメントの「Creating a repository from a template」を参照して欲しいが、一言でいうと、テンプレート元の情報は一切引き継がずに master のソースコードだけがハードコピーされた新規リポジトリが生成されるイメージである。ちなみに、テンプレート元から派生先を辿ることはできない。

- Creating a repository from a template
  https://help.github.com/ja/articles/creating-a-repository-from-a-template)

4.5　igata の採用する技術

ここからは実際に igata の実装について触れていく。

TypeScript

React に限らず、昨今のウェブフロントエンド開発において、フルスクラッチで実装を始めるのであれば TypeScript を採用しない理由はないだろう。例に漏れず igata でも TypeScript を採用している。

ここでは、TypeScript を採用することによる恩恵について詳しくは触れないが、先のウェブフロントエンドを取り巻く事情で触れたように、この分野の開発経験が十分ではないエンジニアや、持ち帰り作業中心の協力会社エンジニアとチームを組むことも多い。こういった体制では、コードレビューに至る前に多くの問題を拾い上げてくれる。この効果は絶大である。

ありがちな問題として、リードエンジニアに作業が集中しコードレビューに忙殺されてしまうと、開発全体が円滑に進まなくなってしまうことがある。各員に負担を分散し Pull Request の初回品質が向上することはチームの開発を円滑に進めるという意味でも非常に大きな役割を果たす。

そういった事情もあり、TypeScript のルールも比較的厳しめに設定している。

```
// tsconfig.json
{
  "strict": true, // If enable this option, following 7 options are enable.
  // "noImplicitAny": true,
  // "noImplicitThis": true,
  // "alwaysStrict": true,
  // "strictBindCallReply": true,
  // "strictNullChecks": true,
  // "strictFunctionTypes": true,
  // "strictPropertyInitialization": true,

  "noImplicitReturns": true,
  "noUnusedLocals": true,
}
```

これらルールについては賛否両論あり、日々議論されていることは認識しているが、ウェブアプリケーションをフルスクラッチで実装するのであれば、このくらいのルールを遵守することは、難しくないと考えている。

一方で明示的な any を利用することは許容している。データフロー上問題が発生し得ない any と型パズル力の不足や時間的な都合で推論を機能させることができなかった場合の any を区別できるように、FIXME_any という型を用意している。

```
type FIXME_any = any
```

FIXME_any の定義は上記の通りで、完全に any と同様に機能するが、もし可能であれば修正したいという意思を表している。これは、PR の時点でコードレビュアが解決しても良いし、要点を切り出した上で、余暇の型パズルとして勤しんでも良いし、Slack や Twitter に投げて神へ祈りを捧げるのも良いだろう。

ところで、TypeScript 公式ウェブサイト上で展開されている Playground は TypeScript 3.6 のバージョンアップと共にリニューアルされ、Playground 中に記述したコードの内容がリアルタイムに URL に反映され、気軽にシェアできるようになった。

例えば、以下 URL を開くと、続くソースコードが表示される。(下記 URL はレイアウトの都合上意図的に改行を追加しているため注意)

```
https://www.typescriptlang.org/play/index.html#code/
MYewdgzgLgBFCGAjANgUxgXhgbwFAxiWAC4d8CYx4BbVUgcgAtVlkR6Aacg+Aczp
j0ATAAZO5AL5cCAE1QAzUngqUaA+gHcQAJ2QzxK5gEtejKAwCMAdjHSYU8qaNLuq
2gygBPAA6oIwbSNvKAMKDVQTMwYANhEAa156SVwJXFwAenSYAFVTGABCfNxQSFh+
KAA1eGQAV1QAMW0QagAVJDRMGAAK0NRPC1JezxB5OB9UEbh21A4YIaFSeDBPAEpM
AD4plFQAbSGLAF09vqEDtPKq2oam1umu+iJ0QT5UejXMmFQAD19gWAB5ADSuAu1T
qjWabW29zk8ieTAiphC7yy31+AOBoKuENu0PoTnhLzeMA+aNQfxgAFFtE1tLggA
```

```
const table = {
  abc: {
    name: 'hello',
    age: '20'
  },
  def: {
    name: 'world',
    height: '170'
  },
  ghi: {
    name: 'typescript',
    weight: '60kg'
  }
}

// Ugh !!
const getValueFromTable = (key1: any, key2: any) => table[key1][key2]

getValueFromTable('abc', 'age') // expect OK
getValueFromTable('def', 'height') // expect OK
getValueFromTable('ghi', 'age') // expect Error
```

これは実際に社内で共有した型パズルである。key1 に応じて条件が変化する key2 の推論がうまくいかず悩んでいた。このように非常に手軽に共有でき、非常に便利なのでオススメである。

React

igata では SPA を実現するための View ライブラリとして React を採用している。

私が好んで React を利用している最も大きな理由は、つまらない話かもしれないが最初に触れたライブラリが Angular や Vue ではなく React であったからという理由が大きい。ACCESS においても React が採用されることが多く、趣味と仕事でノウハウを集中できるという理由もある。

ちなみに、ACCESS としては React の利用を強制しているわけではないため、担当エンジニアの裁量によって AngularJS (JS 時代) や Vue を採用する案件もあるが、現在は React に落ち着いている印象である。私の観測範囲の案件に関していえば、Angular を利用するほど大規模なウェブアプリケーションを開発する機会は少なく、現場のエンジニアが流動的であることから Vue を採用した場合は、交通整理が大きな負荷になることが懸念されるため、結局のところ React が妥当なのかもしれない。

さて、React について触れると igata では Class Components ではなく React Hooks を利用した Function Components の利用を推奨している。

これは公式推奨であることも理由の 1 つだが、ソースコードが明瞭で簡潔に記述でき、バックエンド開発では Elixir を活用する社内エンジニアとも思考の相性が良いのではと考えている。

また、React コンポーネント内では、何らかの理由でイベントリスナーを実装しなければならないことがある。Class Components の場合は、ライフサイクルメソッドとして componentDidMount と componentWillUnmount を利用し、Function Components の場合は、React Hooks で useEffect を利用して実現することになる。

どちらの方法でも完全に同等の機能を実現できるが、ライフサイクルメソッドの場合「いつ、何をする」という記述の流れを取るのに対し、React Hooks の場合は「何を、いつする」という流れを取るため、とある機能に着目したときに処理が分離せず見通しが良くなる。

実際にサンプルコードを用意したので見比べてみるとわかりやすい。

```
// Class Components
export class PomeranianCC extends React.Component {
  bark = () => console.log('わんわん')
  meow = () => console.log('にゃーん')

  componentDidMount() {
    window.addEventListener('click', this.bark)
    window.addEventListener('click', this.meow)
  }

  componentWillUnmount() {
    window.removeEventListener('click', this.bark)
    window.removeEventListener('click', this.meow)
  }

  render() {
    return <h1>Hello, pomeranian !!</h1>
  }
}
```

```
// Function Components with React Hooks
export const PomeranianFC: FC = () => {
  useEffect(() => {
    const bark = () => console.log('わんわん')
    window.addEventListener('click', bark)
    return () => window.removeEventListener('click', bark)
  }, [])
```

```
  useEffect(() => {
    const meow = () => console.log('にゃーん')
    window.addEventListener('click', meow)
    return () => window.removeEventListener('click', meow)
  }, [])

  return <h1>Hello, pomeranian !!</h1>
}
```

上記はいずれも、コンポーネントがマウントされた際に、2 つのイベントリスナーを張り、アンマウントされた際に、イベントリスナーを開放する処理を実装している。実際には、イベントリスナーを張る対象は、特定の DOM ノード (ref, useRef を利用する) になることがほとんどだと思うが、ここでは単純化し widow オブジェクトを対象としている。

Class Components の場合は、ライフサイクル中心の考え方であるため、機能に着目した時に処理が分離されてしまう。Function Components で React Hooks を利用した場合は、機能に着目しているため、処理が集約される。

また、ここでイベントリスナーが増えていくケースはあまり考えたくないが、仮に無数に増えたとしても、処理が集約されているためイベントリスナーの開放漏れに繋がる可能性は低いだろう。

加えて、以下のように実装を外出しして共通化することも可能だ。汎用的な処理であれば、別ファイルに実装し import しても良い。非常に強力である。

```
const onClickWindowEffect = (message: string) => {
  const f = () => console.log(message)
  window.addEventListener('click', f)
  return () => window.removeEventListener('click', f)
}

// Function Components with React Hooks
export const PomeranianFC: FC = () => {
  useEffect(() => onClickWindowEffect('わんわん'), [])
  useEffect(() => onClickWindowEffect('にゃーん'), [])

  return <h1>Hello, pomeranian !!</h1>
}
```

Redux

React に対して Redux には否定的な意見を多く聞くが、とはいえ、現時点では Redux が最も無難な選択なのは間違いないだろう。(Relay はどうなんだろうか？　)

React Hooks が登場して以来は、脱 Redux の機運も高まっており、Redux における問題点の解決を目指したコンパクトなアイデアを見かける機会が増えてきたが、Redux には一日の長があり、Redux を前提としたライブラリが充実し、ノウハウも蓄積されている。商用向けウェブアプリケーションの開発であれば、このアドバンテージは無視できないだろう。

実際開発では React 自体に関する悩みよりも、Redux フレームワークの上で制御するデータフローやエラー処理、副作用の扱い、ディレクトリ構成など、ウェブアプリケーション全体のアーキテクチャに起因する悩みの方が圧倒的に多い。これは、同時に Redux を利用する上でのノウハウとして蓄積されているため、気軽に他フレームワークを選択しづらいという気持ちもある。

本稿でも Redux に関連した話は長くなる。

コンポーネントの分離とディレクトリ構成

Redux 公式ドキュメントの Presentational and Container Components に従う。

- Usage with React
  https://redux.js.org/basics/usage-with-react

要点を簡単に説明すると、Redux store へアクセス (connect) し、ロジックの責務を負う Container Components と、マークアップやスタイルなどの、UI に関する責務を負う Presentational Components を分けて管理するというアプローチである。

Redux では store にアプリケーションの状態を一元的に管理するが、これに対してあらゆるコンポーネントから自由にアクセスしてしまうと、データフローが単一方向とはいえ、影響範囲が読みづらくなるため、制限を設けて整理しようという話である。

igata でも原則この方式に従っていて、コンポーネント設計を行う上で、Redux store へアクセス可能な Container Components をどう決めるかというのは大きな関心ごとの 1 つである。

一方で、責務の分類については厳格には行っていない。Container Components であっても、マークアップを構築し、CSS によるスタイルの調整を行ってしまうことが多い。これについては、私自身が、ルールを遵守する難易度に対して得られるメリットが薄いと感じているためであるが、単に横着しているだけかもしれない。正直、あまり熟考できていない要素の 1 つである。

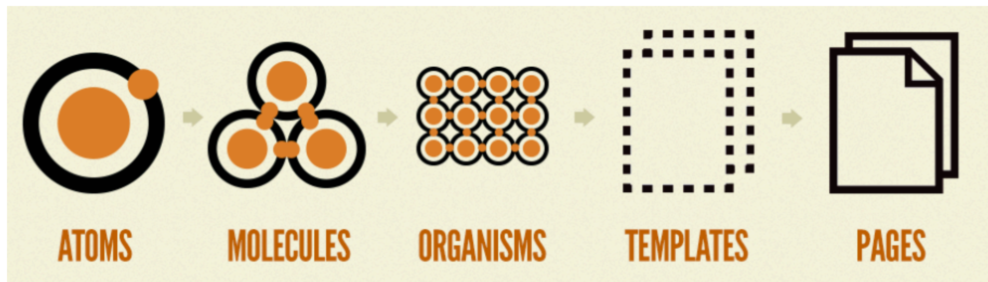

図 4.5: Atomic Design

また、ディレクトリ構成に関していえば、他に Atomic Design という選択肢もある。Presentational and Container Components というアプローチでは、Redux Store のデータを中心に捉え Container Components をどう決めるかという考えが先行するため、トップダウンの設計になりがちである。

一方、Atomic Design の場合は逆に、最小要素である Atoms をどう定義し、組み合わせることでページを構成するかというアプローチを取るため、自然とボトムアップの設計になりやすい印象がある。Storybook といったスタイルガイドを中心に開発する際も Atomic Design は向いているだろう。

ただし、コンポーネントに対する分類が増えることと、その基準が明確ではない点もあるため、開発メンバー間で認識を統一し、適切に分類できるかという疑問がある。この認識が揃わずに開発を進めると、コードレビュー時に度々ストレスを伴うことになるし、整理されず混沌としてしまった場合は、逆にコードリーディングの理解の妨げになってしまうのではないかと懸念していた。

勿論、これは Atomic Design というアプローチを否定するものではない。開発メンバー間で認識を合わせ、適切にコンポーネントを分類できれば、コンポーネントの再利用性が高まり、有効なアプローチだと思うし、特にウェブアプリケーションの規模が大きくなるほど、その恩恵を受けやすくなるのではないかと思う。

connect vs useSelector, useDispatch

React Redux v7.1.0 より、Hooks に対応し、useSelector, useDispatch という API が公開された。従来の HOC である connect API を利用する方式よりも、簡潔で直感的な記述ができるようになった。

- React Redux/Hooks
  https://react-redux.js.org/next/api/hooks

igata は connect を利用する方式のままであるが、これは意図して選択しているわけで

はなく、単に検討が進んでいないだけである。

上記ドキュメントによると connect の第 1 引数を利用 (mapStateToProps) した場合と useSelector は完全互換ではなく、いくつか動作が異なる点があると記載がある。この辺りを仕様をキャッチアップしてから移行したい。

非同期・副作用の処理

Redux において副作用をどう扱うか、というのは最も大きな関心ごとの 1 つであり、その代表である Web API の扱いが、ウェブアプリケーションにおいて重要であることを疑う人はいないだろう。

Redux では副作用を扱うための middleware としては、redux-thunk と redux-saga のいずれかが利用されることが多い。redux-thunk は非常にシンプルで薄い middleware で、actionCreator のレイヤーで主に async-await を中心に利用して非同期処理を実装することになる。一方 redux-saga は saga の用意する豊富な API に加えて generators を利用して非同期処理を実装していくことになる。igata では redux-saga を選択している。

最近のウェブフロントエンドでは generators に触れる機会は限られているため、極端に言えば redux-saga だけのために generators を学ぶことになるエンジニアもいるだろう。また、redux-saga は強力な API を豊富に提供しているが、それ故に扱いには慣れが必要になる。

これは、今までに述べた igata の設計思想とは反するように思えるかもしれないが、ビジネスロジックが集中する重要なレイヤーであるため、習得の初期コストを抑えるよりも、独立性の高さや、テストの書きやすさを重視した。

また、ハイブリッドアプリとしてウェブアプリケーションを実装する場合、Native の用意した JavaScript API を利用してデータのやり取りを行うこととなるが、これらのやりとりも非同期処理であり、コールバックによってデータの受信を待ち受けることになる。redux-saga であれば、saga のレイヤーで Native からのイベントを待ち受け、任意の action を dispatch (put) する実装が容易に実現できる。Redux 単体や redux-thunk ではどう実装すべきか悩む処理を、saga のレイヤーに記述すると、Redux のフローに自然に合流させることができるのが大きな魅力だ。

最後に redux-saga API は非常に充実していて、定期的にドキュメントを見直すと新しい発見があり、ここぞというときに力を発揮してくれる。習得コストは小さくないが、リターンは大きいと考える。

Actions

Redux における ActionTypes 及び actionCreators に関しては、ライブラリ類は利用せず素朴に実装する方針とした。

以前は Redux 公式の提供する redux-actions を採用していたが、記述するソースコー

ド量は削減できるものの、その反面、Actions と Reducer との繋がりが分かりづらく、全容を理解しづらい状況になっていた。特に TypeScript と組み合わせた際の推論について理解が及ばず、混乱することが多かったように思う。

とはいえ、TypeScript となると、Reducer 向けの Payload の型や、各種コンポーネント向けの Dispatcher の型、など類似の記述が多く、現場のフィードバックとコントリビュートがあり、推論で用意する仕組みが導入された。

```
// action types
export const Type = {
  LOGIN: 'LOGIN/LOGIN',
  LOGIN_SUCCESS: 'LOGIN/LOGIN_SUCCESS'
} as const

// action creators
export const login = (id: string, password: string) => ({
  type: Type.LOGIN,
  payload: { id, password }
})

export const loginSuccess = ({ token, userId }: LoginState) => ({
  type: Type.LOGIN_SUCCESS,
  payload: { token, userId }
})

export type LoginAction =
    CreateActionTypes<Omit<typeof import('./login'), 'Type'>>
export type LoginDispatcher =
    CreateDispatcherTypes<Omit<typeof import('./login'), 'Type'>>
```

CreateActionTypes と CreateDispatcherTypes に import した自身のファイルを渡すことで、ActionType と ActionCreator の実装を参照し LoginAction と LoginDispatcher を導く。

import 側のコンポーネントと Reducer の実装は以下の通りである。

```
// React コンポーネント
interface DispatchProps {
  readonly login: LoginDispatcher['login']
}

// 実装省略
```

```
// Reducer
export const loginReducer: Reducer<LoginState, LoginAction> = (
  state: LoginState = defaultState,
  action: LoginAction
) => {
  switch (action.type) {
    case Type.LOGIN:
      return state

    case Type.LOGIN_SUCCESS: {
      return { ...action.payload }
    }
    // 実装省略
  }
}
```

Routings

react-router を採用している。他に選択肢はないように思う。

また、これに加えて react-router の Redux binding である connected-react-router を導入しているが、igata から外す方針を考えている。あまり恩恵を受けられていないというのもあるが、connected-react-router と react-router それぞれの役割を分けて理解する必要があり、トラブルシューティングも複雑化しているように思う。

特に、connected-react-router の問題なのに、react-router のドキュメントを調べてしまうということはありがちだ。

Styles

Styles については、主に CSS 周辺の話であるが、これは React に関わらず、昨今のウェブフロントエンドにおいて、最もホットで話題が尽きない分野の 1 つである。

CSS

CSS については、大きく分けて検討すべきカテゴリが 2 つある。

1 つは、AltCSS と呼ばれる CSS プリプロセッサを利用するかどうか。具体的には、LESS, Sass, Stylus, PostCSS などがある。(PostCSS は位置づけに悩むが、一応ノミネート)

2 つ目は、CSS をどう記述し DOM ノードに適用するか。React では 3 つのスタイルがある。

- グローバルに CSS を読み込む従来の方法
- CSS Modules
- CSS in JS

igata では CSS Modules + Sass を採用している。

最近では styled-components や emotion といった CSS in JS の採用が伸びているため、保守的な選択ではあるが、従来より培ってきた HTML + CSS の開発経験をほぼそのまま生かすことができるという理由は大きい。その上で、CSS Modules では、CSS のクラス名の名前空間を分離できるため、BEM や OOCSS、SMACS といった命名規則を抑える必要がないというのが大きな理由だ。

ちなみに、Sass を導入していることに大きな理由はない。過去の開発経験を踏襲して導入しているが、CSS Modules 扱う場合はメリットが薄れており、旨味は薄れているように思う。

CSS フレームワーク

CSS フレームワークは意図して導入していない。

これは請負案件という性質が大きいのかもしれないが、社内外デザイナーが綿密な UI デザインを用意し、顧客との合意の後、実装に入るというケースは多い。この場合 UI デザインを忠実に実現することが求められるため、CSS フレームワークの恩恵を受けづらく、むしろ、マークアップが複雑化したり、フレームワークの仕様を理解していないと、予期せぬ影響を及ぼしてしまう可能性もある。

一方、PoC 案件など、機能以外の実装が重視されていないケースでは UI デザインが用意されず、実装者に任されている場合がある。こういった場合は、bootstrap や bulma などの CSS フレームワークを導入する。

余談であるが、UI デザインに関しては、PoC 案件では重視されないことも多い。UI をどう実装するかは、完全にエンジニアに一任されることもあるが、いざ実装してデモを行うと色々と修正要望を受けることが非常に多い。見積もりや進め方には注意したい点である。

アニメーション

react-transition-group を導入している。

アニメーション関連はライブラリは非常に豊富で話題としても目立つのだが、CSS Modules と相性が良い手段は限られてくる。というのは、CSS Modules を採用している以上は、アニメーションも CSS で実現し実装を集約したいからである。スタイル全般が CSS として別ファイルに実装されていて、アニメーションの実装だけがロジック側にあるというのは、少々厳しい気持ちになる。

従って逆にいえば CSS では実現が不可能な高度なアニメーションが必要なのであれば、アニメーションのライブラリだけでカバーするのではなく、スタイルからして CSS in JS を選択した方が、JavaScript 側へ処理を集約できるため、筋が良いのではないかと推測する。

また、この react-transition-group は非常に扱いやすく、感銘を受けたライブラリの 1 つである。あるオブジェクトがアニメーションするということは、ロジックとしては考慮すべき状態が増えることになるが、これを開発者が意識しないで済むように扱ってくれる。これにより、ロジックとアニメーションの実装が完全に分離できる。

Linters と Formatters

静的解析ツールとして ESLint と Stylelint, フォーマッターとして Prettier を導入している。

以前は TypeScript といえば TSLint が定番であったが、本年の 1 月より ESLint が TypeScript をサポートし始めたことで移行が進んだ。現在は TypeScript であっても ESLint を採用することが多い。ESLint はドキュメントが非常に充実していて、各種 Rule に対する具体例も多い。リリースに伴って破壊的変更がある場合も、移行手段のサポートが非常に手厚く安心感がある。

加えて、フォーマッターとして Prettier を導入している。これは ESLint ではカバーできないソースコードの整形を行うツールで、JavaScript や TypeScript だけでなく、JSON, YAML, Markdown など、プロジェクト内の大抵のファイルを一定のルールに沿って整形できる。

しかし、JavaScript や TypeScript ファイルに対して適用する場合は ESLint と衝突しないよう注意が必要である。具体的には、ESLint のルールのうちフォーマットに関するルールを無効化した上で、ESLint の実行完了後に Prettier をフックしてコードを整形することになる。これは以下の ESLint 用のプラグインとライブラリを導入することで対応できる。

- eslint-config-prettier
- eslint-plugin-prettier

さらに、Visual Studio Code (VSCode) において、ファイル保存時に処理する必要があることを考えると、VSCode 本体と VSCode の ESLint Extensions, Prettier Extensions の設定が必要となり、これもまた非常に複雑である。幸運にも、試行錯誤に結果自分の環境では機能しているが、正しい設定の組み合わせを理解していない。どこかで整理した上で、.vscode のワークスペース設定に追記したい。

また、これらの ESLint, Stylelint, Prettier といったツールは実行漏れがないように、

コミットの際に自動で適用する仕組みを導入している。以下のライブラリで実現している。

- husky
- lint-staged

ツールを完備していても、実行しなければ意味がない。自動化しておけば、運用ルールを改めて周知する必要はないし、ヒューマンエラー防止にもなるため、負担軽減になる。

最後に、本運用に関する Tips を 1 つ紹介する。

この仕組みによって、何らかの理由で作業を中断し WIP コミットしようとしてもツールが走ってしまうため、実装が不十分である場合はコミットに失敗してしまう。

つまり、Git stash を利用するしかないのだが、これにも 1 つ問題がある。新規にファイルを追加していた場合は Untracked files として扱われるが、これは git stash の対象とはならない。この場合は、一度 Untracked files を git add で staged の状態にしてから git stash することで、まとめて stash が可能だ。地味ではあるが、実装作業中にコードレビューをするためにブランチを切り替えるという機会は多く、覚えておくと役に立つだろう。

■コラム: 設定ファイルのフォーマットはどれを選ぶべき？

先に静的解析ツールの話をしたが、いずれも設定ファイルを準備する必要がある。ルールを指定するためだ。そして、これら設定ファイルが必要になるのは、静的解析ツールだけではない。CircleCI や GitHub Actions のような CI 系ツールにも必要であるし、親の顔より見た package.json も npm の設定ファイルと見ることもできるだろう。

このとき、ツールによっては設定ファイルのフォーマットにいくつか選択肢が用意されている。例えば、ESLint は、以下のいずれのファイルであっても認識可能だ。

- .eslintrc.js
- .eslintrc.yaml
- .eslintrc.yml
- .eslintrc.json
- .eslintrc
- package.json (package.json 内に eslintConfig プロパティで記述できる)

では、どのファイルフォーマットを選択すればよいだろうか？

私の場合は YAML を選択することが多い。そしてなぜ YAML が良いのかといえば、コメントが書きやすいということに尽きる。開発ツールが充実し、多機能化する

と同時に、設定ファイルに記述する内容も増え、複雑化している。例えば、以下は igata の package.json の一部を抜粋したものである。(説明の都合上、一部並べ替えている)

```
// 前略
"@typescript-eslint/eslint-plugin": "^2.2.0",
"@typescript-eslint/parser": "^2.2.0",
"babel-loader": "^8.0.6",
"ts-loader": "^6.1.0",
"clean-webpack-plugin": "^3.0.0",
"copy-webpack-plugin": "^5.0.4",
"css-loader": "^3.2.0",
"dotenv-webpack": "^1.7.0",
// 後略
```

この依存ライブラリを見ると TypeScript でありながら babel でもコンパイルする構成を取っているようにみえる。しかし、実際には babel-loader は storybook のビルドにのみ利用していて、igata 本体では利用していない。igata をテンプレートとして利用し、まず不要な機能を削ろうとして babel-loader や @babel/core を見かけたら困惑するだろう。

ESLint に関していえば、何かしらの理由で特定のルールを無効化する機会があるだろう。できれば、その特殊な事情である「何かしら」はコメントに残したい。

JSON ファイルでは、こういった重要な情報をコメントとして残すことができないのは大きな問題である。(一応、1 つのプロパティとしてコメントを記述することは可能であるが、気は進まないだろう……)

勿論 YAML ではなく JS でも良い。ただ、単純なオブジェクトで済む場合は、YAML の方がインデントなくトップレベルにプロパティを記述できる点が筆者の好みである。webpack のように変数やシンプルなロジックが書きたい場合に JS を選択している。

それにしても package.json 内の情報は増える一方だというのに改善が進まないのは残念というほかはない。

Tests

Jest と Storybook を導入している。

ただし、正直なところ boilerplate として仕組みを提供しているものの、私の周りの案件ではテストをしっかり書くといった機会は少ない現状である。決して、軽視しているわけではないのだが、長期保守に至る案件が少ないという事情と、未経験者も多い状況でテストまで手が回っていないというのが実情である。

若干言い訳のようで苦しいが、TypeScript のルールを比較的厳しめに設定していることと、先に述べた ESLint, Stylelint といった静的解析ツールである程度の品質を保っている。

また、redux-saga の項で触れたが、この限られた時間とリソースの中、1 箇所テストを書くレイヤーを選ぶなら、ビジネスロジックが中心となる redux-saga が最も費用対効果が優れていると考えている。従って、igata として、まずは redux-saga のレイヤーに対して効果的にテストを書く仕組みを導入していきたい。

文言の扱い

ウェブアプリケーション上に表示する文言については、1 つのファイルに集約し、import して参照する方式を取っている。

これ自体は一般的で珍しくないが、1 つポイントがあるとすれば、この文言を記述するファイルは JSON や YAML ではなく TypeScript として実装している点だろう。

直感的には、定数データは JSON, YAML 形式を選択しがちであるが、TypeScript ファイルとして実装することで、推論の恩恵を受けることができる。これは、参照元でプロパティ名をサジェストして入力できるし、typo によるエラーも検出できる。この恩恵は非常に大きい。

また、文言を返すシンプルな関数を登録できる点も便利だ。具体的なユースケースとしては、以下のような変数をサンドイッチした文言を表示したいケースがある。

```
選択した項目の合計は◯件です。
```

◯ 件の部分は変数によって動的に決まる値である。この場合、文言を定数として用意するのであれば、"選択した項目の合計は" と "件です。" という、2 つの文字列を定義する必要がある。こういった場合、以下のように文言を返す関数を実装してしまう方がスッキリする。

```
(n: number) => ‘選択した項目の合計は${n}件です。‘
```

　また、非常に簡易的ではあるが、ブラウザの言語設定 (navigator.languages) を参照し、読み込むファイルを切り替えることで i18n 対応を実現している。先に述べた関数が登録できるメリットは i18n 向けにはより重要になる。日本語と英語では、ある変数をサンドイッチするときの並びが異なる可能性があるからだ。(例えば、敬称や金額の表示など)

　最後に、i18n 対応する場合は言語ごとに ja.ts や en.ts のように文言実装が分かれることになるが、一方にしか存在しないプロパティがある場合は、参照するプロパティの型が `string | undefined` と推論されるため、事前に不備を検出できる。

　ちなみに、以下の様に export する文言オブジェクトは const assertion を指定すると VSCode 上ではマウスオーバーで変数の中身が確認できるため、非常に快適である。

```
// ja.ts
export default {
  todoApp: {
    title: ’TODO アプリ’,
    name: ’TOP’,
    newTodo: ’TODO を追加’,
    loginMessage: (userId: string) => ‘ログインユーザ: ${userId}‘
  },
  footer: {
    twitter: {
      label: ’Twitter’,
      url: ’https://twitter.com/diescake’
    },
    github: {
      label: ’GitHub’,
      url: ’https://github.com/diescake/igata’
    }
  }
} as const // const assertion
```

```
7      return (
6        <div className={style.container  (property) title: "TODO アプリ" | "TODO Application"
5          <Header title={words.todoApp.title} userId={props.userId} icon={faListAlt} />
4
3          <div>
```

図 4.6: const assertion によって、マウスオーバーで文言の内容が表示される

上記のポップアップを表示した環境には、ja.ts と en.ts の 2 つの文言ファイルが存在していたため、推論の結果双方の文字列が Union Types で表示されている。

ビルド

webpack

今更触れる必要もないくらいのデファクトスタンダードであるが、igata も webpack を採用している。設定ファイルは以下の通り 3 つのファイルに分割している。

- webpack.prod.js
- webpack.dev.js
- webpack.common.js

共通処理を webpack.common.js に集約し、それぞれ webpack.prod.js, webpack.dev.js から import して参照する形式を取る。

```
"scripts": {
  "start": "webpack-dev-server --config webpack.dev.js",
  "build": "webpack --config webpack.prod.js"
},
```

package.json に記述された npm script からはそれぞれ webpack.dev.js と webpack.prod.js を設定ファイルのエントリポイントとして指定する。また、細かく触れないが、webpack 周り、特に webpack-dev-server にはいくつか嵌りどころがあり解決している。(`historyApiFallback` や `0.0.0.0` など)

yarn vs npm

igata では yarn を採用している。(とはいえ、npm を利用しても問題が発生するわけではない)

最近では特にそれ程大きな違いはないと感じるが、コマンドラインインタフェース (CLI) という観点で npm に比べて yarn がより直感的であり、嵌りどころが少なく感じるため採用を継続している。

例えば、yarn でも npm でもそれぞれ、ローカルインストールするライブラリのバージョンを厳格に扱うための lock ファイルを持つ。

- package-lock.json
- yarn.lock

これによって、異なる環境でも完全に同バージョンのライブラリをインストールすることが可能になるのだが、npm install コマンドは、ロックファイル (package-lock.json) を参照せずに package.json を参照してライブラリのインストールを行ってしまうため、npm ci コマンドを実行する必要がある。

```
$ npm ci
```

yarn の場合はいつも通り yarn コマンドを実行すればロックファイル（yarn.lock) を参照してくれる。

```
$ yarn
```

npm の場合、運用を考えれば、npm install コマンドよりも npm ci コマンドを実行する機会の方が多くなるはずだが、私の印象では npm ci コマンドはそれほど浸透しておらず npm install が軽快に叩かれているような印象がある。

VSCode 向けサポート

昨今のウェブフロントエンド開発では開発環境として VSCode を利用するユーザを多く見かけるようになった。私自身もその 1 人であるが、特に TypeScript 自体が Microsoft の手掛ける OSS であるということもあり、アップデートが早く親和性が高いというのは、大きな理由の 1 つだろう。

さて、igata でも VSCode ユーザ向けにいくつかのワークスペース設定を用意している。さほど充実しているわけではないのだが、推奨 Extensions のリストと、Chrome を利用したデバッガの設定、一部エディタの共通設定を提供している。

ところで余談だが、ウェブフロントエンドに限らず、未経験者と仕事をする上でペアプログラミングを行うことは、キャッチアップを早めるために非常に有効だと感じている。特にアドバイスを行う場が、テキストベースの質問ドリブンだけであったり、Pull Request 上が中心となると、互いに精神衛生上あまり良くないだろう。

VSCode には Live Share というペアプログラミングにうってつけの機能がある。

- Visual Studio Live Share
  https://visualstudio.microsoft.com/services/live-share/

これは一方がホスト側となり、もう一方がクライアントとして接続することで、リアルタイムにソースコードを共有し編集ができるという代物である。双方自分の使い流れた環境で、ペアプログラミングを行うことができる。

また、実践できてはいないが、1:1 だけではなく多人数でも利用可能なので、モブプログラミングも可能だ。

DevOps

DevOps という括りが適切なのか判断できないが、igata をメンテナンスしたり、業務で活用しているサービスをいくつか紹介する。

Dependabot

好きです。

Pull Reminders と Pull Assigners

こちらも Dependabot と同様、本年の 6 月に GitHub 社に買収されたサービスで、現在はプライベートリポジトリに対しても無償で利用可能となっている。

Pull Reminders は Pull Request に動きがあり、Reviewers や Assignees に自身が設定された際に Slack に通知を行ってくれるサービスである。GitHub の通知を Slack のチャンネルに飛ばしている人は多いと思うが、Pull Reminders はチャンネルへの通知だけではなく、bot が直接 direct message を飛ばしてくれるため、他の通知に紛れて見落とすことが少ない。

一方 Pull Assigners は Pull Request に対して、あるルールに沿ってレビュアを選出してアサインできるサービスである。1 つの Pull Request に対してレビュアにアサインする人数を規定したり、その割り振りを特定のチームの中から、単純な持ち回り (ラウンドロビン) で決めるのか、あるいは、負荷を分散するために、過去のレビュー担当回数を見るのか、といった条件を選定できる。

現在、私の案件では、1 つの Pull Request に対して、リードエンジニアのチームから 1 名、その他担当エンジニアのチームから 1 名を選出した合計 2 名をレビュアにアサインする仕組みを取っている。ただし、Pull Assigners 単体では生成された Pull Request に対して自動でレビュアを選定することはできないため、.github/CODEOWNERS と組み合わせて運用している。

```
/* .github/CODEOWNERS */
```

```
/web/ @access-company/webapp-lead-reviewer @access-company/our-awesome-project
```

- About code owners
  https://help.github.com/ja/articles/about-code-owners

ただ、Pull Assigners がガチャを並列に処理してしまう影響なのか、若干不安定で手動で再実行する機会は多い。加えて、Pull Assigners は Reviewers にアサインできるが、なぜか Assignees には適用してくれない。色々と惜しい。

今後は GitHub Actions を活用したソリューションがたくさん登場してフローが整備されていくことと思う。自分でも大喜利するぞ。

Netlify

信頼と実績の Netlify である。あまりに多用していて足を向けて眠れない。

SPA をホスティングしようと思うと、成果物自体は静的であるが、成果物を生成するためにビルドが必要になる。例えば、GitHub Pages にホスティングしようと思うと、リポジトリにビルドした成果物を合わせてコミットする必要があり、精神衛生上とてもよろしくない。(GitHub Actions で master をビルド、別ブランチにプッシュする運用はできるかもしれない)

こういった場合は、Netlify を利用するのがオススメである。ほぼ、ゼロコンフィグでビルドからデプロイを実現してくれる。さらに、個々の Pull Request に対してもビルド&デプロイが自動で走り、Pull Request ごとにホスティングしてくれる。まさに、到れり尽くせりである。

1 点注意が必要なのは、SPA のルーティングで browser history を利用している場合は、404 を / に fallback する仕組みが必要で、これは、netlify 向けの設定ファイル (_redirects) を成果物中に用意する必要がある。

- Redirect & Rewrite Rules
  https://www.netlify.com/docs/redirects/

サンプル実装の紹介

最後になったが igata のサンプル実装のいくつかを紹介する。これらは、実際に社内の商用案件で利用している機能がほとんどで、定期的に追加している。

余談であるが、私が個人的に行っている運用で、業務中に実装する機能が igata にも欲しい機能だと考えた場合は、先に igata で実装してから、その成果を案件側に取り入れる

という方法を取ることがある。これは、請負契約上、成果物は顧客権利物となるため、有用な汎用機能を実現したとしても横展開できないためである。

ちなみに、igata 向けに機能実装を行っている時間は控除し勤務時間外としている。私は ACCESS では完全裁量労働の立場なので、これによって給与や評価が変化しないというのがポイントである。

ログインとセッション管理

主に B2B 向けに提供するウェブアプリケーションは、ほぼログイン機能が必要となるため用意した。

ログイン ID, パスワードを入力するフォームがあり、submit によって POST リクエストを投げる。ログインに成功したら、レスポンスに含まれるトークンを Redux store と localStorage に保持する。

また、ウェブアプリケーションの初期化時に、localStorage にトークンが保持されていれば、Redux store に読み出す。Redux には Single Source of Truth というポリシーがあるが、これに沿って Redux store 以外の記憶領域へのアクセスは最低限に留めている。(URL に付随するクエリは双方向であるため、このポリシーに反しやすく設計によく悩む)

セッションと各種 path に対するアクセス権の関係は Authenticated というコンポーネントを用意することで、視覚的にわかりやすく分割している。

```
<Switch>
  <Route exact path="/login" component={Login} />
  <Route exact path="/terms-of-service" component={TermsOfService} />
  <Authenticated>
    <Switch>
      <Route exact path="/" component={TodoApp} />
      <Route exact path="/board" component={Board} />
      <Route exact path="/profile" component={Profile} />
    </Switch>
  </Authenticated>
</Switch>
```

上記コードでは、/login と /terms-of-service には、トークン (セッション) を持たずにアクセスできるが、Authenticated コンポーネントの子ノードにあたる /, /board, /profile へアクセスした場合は、ログイン画面にリダイレクトする処理が実装されている。

axios ラッパーの実装

先の項目で Redux における副作用を redux-saga で扱うという話をしたが、実際に API を扱うとなると、単純な実装の重複が目立ちラッパーの実装が必要であることに気づくだろう。igata では以下の機能を持つ axios のラッパーレイヤーを実装している。

- Authorization ヘッダとトークンの自動付加
- 401 Unauthorized 受信時のリダイレクト処理
- プログレスインジケータを扱うための、通信状態を Redux Store へ put する処理

ウェブアプリケーション中では、ほぼ全てのリクエスト (勿論 XMLHttpRequest 相当のみ) で Authorization ヘッダを付加することになるためデフォルトで付加し、ログインなど付加しないケースにフラグを指定するといった形式を取る。

また、全てのレスポンスで 401 Unauthorized を受信した場合は、(大抵の場合) セッションが無効であることを意味するため、ログイン画面へのリダイレクト処理を用意している。

最後に、新しいリクエストを送信、レスポンスを受信したタイミングでそれぞれ put (dispatch) することで、現在通信中のリクエスト数を Redux store に保持している。これは非常に簡易的な機能であるが、後に触れるプログレスインジケータの実装に利用している。

API レイヤーのデータフロー制御

続いてもう 1 つ重要なのはレスポンスのデータフローである。レスポンスに対して行うべき処理は多く、特にエラーによる処理の分岐が多いため、データフローを形式的に整理しておかないと、場当たり的なエラー処理となりバグの温床となりがちである。

- レスポンスのバリデーションと型の付与
- エラーレスポンスの処理
- レスポンスの読み替え

API 経由で取得したレスポンスには当然ながら型が付与されていない。従って、TypeScript 中で安全に扱うためには、仕様に準じた型を定義して割り当てる必要があるが、このとき、ランタイムにレスポンスを検証するのであれば、バリデーション処理を実装する必要がある。

```
const isLoginResponse = (props: any): props is LoginResponse => {
  try {
```

```
    const { token, user_id } = props
    return typeof token === 'string' && typeof user_id === 'string'
  } catch (e) {
    console.error(e)
    return false
  }
}
```

上記 isLoginResponse 関数では、TypeScript の Type Guard という仕組みを利用し、ログイン API のレスポンスを検証している関数である。この処理の結果 true が返ると、入力に渡された props は晴れて LoginResponse であると認められたことになる。

ただし、このバリデーションは常に厳密に行う必要はない。サーバの API 仕様が固まっていて、十分に信頼できる場合は手を抜いても良いだろう。頻繁にリクエストが発生する API の場合は validation 処理自体が負荷となりユーザ体験を損なう可能性もある。

```
const mapResponseToState = (res: LoginResponse): LoginState => ({
  token: res.token,
  userId: res.user_id
})
```

上記 mapResponseToState 関数はレスポンスの読み替えを行っている関数である。isLoginResponse 関数によって、検証されたレスポンスが引数に渡される。この関数が最も機能するケースとしては、snake_case で記述されたプロパティ名を camelCase にリネームする例である。

JavaScript, TypeScript は慣例的に変数名を camelCase で定義することが多い。従って、snake_case で記述された JSON データを Redux store に格納することは避けるべきである。上記の例では user_id を userId にリネームしている。

また、機械的に命名規則に従ってリネームを行うだけであれば、変換ライブラリを噛ませば良いが、それ以外にも要素の取捨選択を行ったり、何らかの理由で構成の微調整、プロパティ名を変更したいことがあるため、自前で関数を用意する方式を取った。変換ライブラリを利用しても、型は手動で記述する必要があるのでアンバランスにも感じたのも理由の 1 つだ。

プログレスインジケータ

よくある簡易的なプログレスインジケータを用意している。

厳密にいえば、疑似プログレスインジケータであり、何らかのリクエストを行った際

に、じわじわプログレスバーが伸び出し、レスポンスが帰ってきたタイミングで、Flush させて帳尻を合わせる。本プログレスインジケータは CSS で実装しているため、必要に応じてスタイルの調整を行うことができる。

また、一般的なウェブサイトの開発をイメージしていると見落とされがちであるが、SPA ではプログレスのようなユーザインタラクションは、全て自前で実装する必要がある。開発環境ではサーバからのレスポンスが十分早いことが多く UX 上の問題として発覚しづらいが、実サーバと結合して動作確認をすると、クオリティが非常に低く見えることから、後追いで対応が必要になることは多い。

こういったケースでは、ウェブアプリケーション最上部に張り付く形で表示されるプログレスインジケータの採用がオススメである。UI への影響が小さいため、後からでも導入しやすく、さらに、以下のようなメジャーなウェブアプリケーションでも採用されているため、納得感のある交渉ができるだろう。

- instagram
- GitHub
- note
- Google+

その他雑多な実装

その他にも細かい実装は色々と用意されている。

いくつかあげると例えば、History API (react-router) による画面遷移時に、スクロールの位置を最上部に戻す実装がある。

SPA における画面遷移は再読み込みが行われないため、画面遷移してもスクロールバーの位置が保持される。例えば、テーブルのページネーションを例にあげると、テーブルを最下部までスクロールした後に、次ページへ移ったら、最下部にいるというのは直感的ではないだろう。

他には、モーダルダイアログを表示するための、簡易的なコンポーネントも持つ。モーダルダイアログは、ウェブアプリケーションを実装する上で、高い頻度で利用される要素の 1 つである。

正直現在あまり実装が充実している状態ではないのだが、用途に応じたダイアログコンポーネント (Alert, Confirm など) を別枠で用意したり、呼び出し元コンポーネントには直接ロジックは埋め込まず、dispatch のみで制御できる仕組みとか、整理して導入できればと思う。

SPA の実装経験がないと、見落とされがちなポイントというのは非常に多いが、こういった要素は可能な限り boilerplate として潰しておけば幸せになれそうな気がする。

幸せになりたい。

4.6 まとめ

というわけで、長きに渡って React ベースの boilerplate である igata の紹介を通して、知見やノウハウを共有してきた。各項目では、思いつくまま、五月雨的に思考を吐き出してきたので、体系的にまとまっているとはいえず、わかりやすいとも思えないが、それでも少しでも得るものがあったのなら非常に幸いであり、本望である。

また、React ベースの boilerplate 難民の方は igata を触って貰えると嬉しいが、それだけではなく、ぜひ一度自分でもゼロベースで boilerplate を実装してみて欲しい。ベースコードに導入される技術への理解が深まるだけではなく、ウェブアプリケーション全体を俯瞰して、ある機能をどう実現するべきか、と思考する癖が身につき、大きな成長に繋がるだろう。

「なっちゃいない」
「・・・え？」
「漫然とライブラリを導入するな」
「何を前にし———————何を解決しようとしているのか意識しろ」
「それが、ライブラリを利用する者に課せられた責任———————義務としれ」
「ッッ！！」

今までも述べてきた通り、ウェブフロントエンドの世界は日進月歩であり、日々変化を続けているため、そのトレンドを追い続けることは簡単ではない。しかし、ベースコードと、アーキテクチャに対する正しい理解と、問題をスマートに解決するために頭を捻り続けた経験は、大きくパラダイムシフトしたとしても、無駄になることはないだろう。

4.7 Web Frontend Boilerplate Battle

最後に、本章のタイトルである Web Frontend Boilerplate Battle について回収しておく。これは有名なエクストリームスポーツである Gentoo Linux Install Battle から取ったものである。この種の Xxxx Xxxx Battle といったフレーズは定期的に登場し Ryzen SEGV Battle も記憶に新しい。

さて、このネタに共通するのは、本来の目的から逸脱し、その過程、あるいは重要ではない要素に強くフォーカスし盛り上がってしまうという点である。ありふれたつまらない言い方をするなら「手段が目的となっている」というフレーズが当てはまるケースも多いだろう。

今回、私は半年近く igata という boilerplate の実装を続けてきたが、そう。

「まだ、ユーザに何も価値を提供していないのである!!」

本来であれば、boilerplate のことなど意識せずに、何らかの問題を解決したり、新しい価値を届けるためのウェブアプリケーションの実装に注力したいし、そうあるべきのはずだ。

Web Frontend Boilerplate Battle にはそういった皮肉の意がちょっとだけ込められている。ちゃんとウェブサービス作ってくぞ。

4.8　完走した感想

Qiita 記事を書くような感覚でライトに書き始めたものの、気がついたら自分でも驚く程の分量となり、こんなに語りたい内容があったのかと本気で驚いた。

本稿を執筆する中で、各所で自分の理解の浅さが身に染みたが、正直それよりも igata を実装する過程で得たノウハウや知見は結構あるものだ、と感心する方が勝った。(自画自賛)

igata として不足していて改善すべき点に気づくことも多かった。これは、今まで通り気ままに対応していければと思う。

また、雑な紹介になってしまった項目も多くあり申し訳ない気持ちである。特に、全体的に文字ばかりで分かりづらいので、サンプル実装のソースコードを例に解説するべきだと理解はしているのだが、残念ながら時間切れである。

この先は君自身の目で確かめてくれ！

4.9　参考・引用

- 実践 TypeScript ~ BFF と Next.js & Nuxt.js の型定義~
  https://www.amazon.co.jp/dp/483996937X
- りあクト！ 第 2 版 TypeScript で始めるつらくない React 開発
  https://oukayuka.booth.pm/items/1312652
- atomic design
  http://bradfrost.com/blog/post/atomic-web-design/
- Gentoo Linux Install Battle
  https://dic.nicovideo.jp/a/gentoo%20linux
- DARK SOULS III
  https://www.darksouls.jp/detail_ds3_tffe.html
- グラップラー刃牙 1 巻
  https://www.amazon.co.jp/dp/4253053092
- 範馬刃牙 30 巻

https://www.amazon.co.jp/dp/B00EF33KXW

- しあわせアフロ田中 4 巻
  https://www.amazon.co.jp/dp/4091877338

4.10　著者

図 4.7: 唐突に酒樽を担いで喜びを見せる diescake

- 著者名: 近藤 大介 (diescake)
- Twitter: https://twitter.com/diescake
- GitHub: https://github.com/diescake
- Qiita: https://qiita.com/diescake

第 5 章

Elixir で brainf*ck した話

「せっかく Elixir をやるんだからマクロを書いてみたい」っていう人の背中を押して沼に突き落としたい。

執筆時点での Elixir のバージョンは 1.9.1。

作ったもの: https://github.com/SekiT/brainfux

5.1 Agenda

- 動機
- Brainf*ck とは?
- まずやること
  - 車輪の再発明か否かチェック
  - README を書く
- コードを生成する
  - 立ちはだかる壁 `[]`
  - コードを生成するのをやめる
  - 最小限のコードを生成する
- テスト
- ハマりポイント
  - そもそも出来ない前処理
  - コンパイルエラーのテスト
  - `@spec` があるか確かめる
- まとめ

5.2 動機

- Metaprogramming Elixir を読んでマクロを書いてみたくなった
  - https://www.amazon.co.jp/Metaprogramming-Elixir-Write-Less-Code/dp/1680500414
- それとは別に、衝動的に brainf*ck したくなった (某アニメの影響。脳〇いいよね)

ここから導き出される結論は……
「Brainf*ck から Elixir のコードを生成するマクロを書こう！」

5.3 Brainf*ck とは?

https://www.wikiwand.com/ja/Brainfuck
`+-><,.[]` の 8 つの文字からなるプログラミング言語。
Zero fill された配列とポインタがあり、

- `+`, `-`: ポインタの指す先の値をインクリメント/デクリメント
- `>`, `<`: ポインタをインクリメント/デクリメント
- `,`: 入力から一文字読んでポインタの指す先に格納
- `.`: ポインタの指す先の値を出力
- `[`: ポインタの指す先が 0 なら対応する `]` の先にジャンプ
- `]`: 対応する `[` にジャンプ

他の文字はコメントとして無視される。

Brainf*ck のプログラムの例

- `,[.,]`: echo するプログラム (入力をそのまま出力する)
- `,>,[-<+>]<.`: 2 つの入力を足し算するプログラム
- "Hello, world!" を出力するプログラム:

```
+++++++++
[
  >++++++++
  >+++++++++++
  >+++++
  <<<-
```

```
]
>.
>++.+++++++..+++.
>-.------------.
<++++++++.--------.+++.------.--------.
>+.
```

- 以下解説 (興味のある方向け)
  - "Hello, world!" の各文字の文字コードは 72, 101, 108, 108, 111, 44, 32, 119, 111, 114, 108, 100, 33
  - 9 * 8 = 72, 9 * 11 = 99, 9 * 5 = 45 をループを使って予め作っておく
  - 72 を出力
  - 99 + 2 = 101 を出力、101 + 7 = 108 を 2 回出力、108 + 3 = 111 を出力
  - 45 - 1 = 44 を出力、44 - 12 = 32 を出力
  - 111 + 8 = 119 を出力、119 - 8 = 111 を出力、111 + 3 = 114 を出力、114 - 6 = 108 を出力、108 - 8 = 100 を出力
  - 32 + 1 = 33 を出力

このように、文字コードを直接いじくらないといけない上に、数字や文字がコードに登場しないので brain が f*ck される。

Brainf*ck についてもう少し

本来は難読言語として作られたのではなく、コンパイラが小さくなるように作られた言語。
8 種類の命令だけでプログラムが書けるため、文字を差し替えたりハフマン符号化したりして色々なクソ言語が遊びで作られた。

5.4 閑話休題

「Brainf*ck から Elixir のコードを生成するマクロを書こう！」
ということでいよいよ作り始めようと思う、がその前に……

5.5 まずやること 1: 車輪の再発明か否かチェック

まずは既存のパッケージと被ってないかチェック (hex.pm で検索) すると……
exfuck: https://github.com/shiroyasha/exfuck

awesome-elixir 入りしている brainf*ck **インタープリタ**
=> コンパイラじゃないので被ってない！ セーフ！ (強弁)

5.6 まずやること 2: README を書く

やってない人が多いが、README を最初に書くと良い:

- 何をしたいか、どう使ってほしいかを明確にできる
- ある程度客観的になれる (要る・要らないの判断が出来る)
- README がテストコードになってくれる
  - 使い方の例を書くことで、自然とテストコードになる

こんな感じにしたい

```
use Brainfux

# defbf というマクロで関数定義

defmodule Sample do
  defbf hello_world """
    +++++++++
    [
      >++++++++
      >+++++++++++
      >+++++
      <<<-
    ]
    >.
    >++.+++++++..+++.
    >-.------------.
    <++++++++.--------.+++.------.--------.
    >+.
  """

  defbf echo ",[.,]"
end

Sample.hello_world()
# => "Hello, world!"

Sample.echo("foo")
# => "foo"
```

```
# bfn というマクロで無名関数定義

(bfn ",[+.,]").("HAL")
# => "IBM"
```

5.7　コードを生成する

作るものは決まった。作り始めよう。

安直に、brainf*ck のコードを 1 文字ずつ対応する Elixir のコードに変換していくことを考える。

マシンの状態を表す map を `%{tape: [], pointer: 0, input: "", output: ""}` みたいな感じで作っておいて、

- `+` => `|> Map.update!(:tape, fn [head | tail] -> [head + 1 | tail] end)`
- `>` => `|> Map.update!(:pointer, &(&1 + 1))`

という感じに変換すれば良さそう？

立ちはだかる壁 []

しかし `[` と `]` に対応するような Elixir のコードは無い。

いわゆる `while` に相当するものが必要なわけだが、標準には無い。マクロで定義するにしても

```
while (mutable な何か) do
  ...
end
```

という形になり、（mutable な何か）を実現するとしたら軽量プロセスを立ち上げて……ということになるが、`[` が登場するたびにプロセスを立ち上げてたらキリがない。

それなら関数の再帰呼び出しで実現するか？ ということになるが、やはり `[` が登場するたびに関数定義していたらキリがない。

従って、予めどこかで関数を定義しておいて、生成するコードではその関数を呼び出すだけにするしかない。

それならいっそ……

コード生成するのをやめる

インタープリタを予め作っておいて、生成するコードはそれを呼び出すだけにしてしまえば良い。

(そもそも長いコードをマクロで生成するのは bad practice であった)

しかしそれではインタープリタを作るのと同じになってしまうので、コンパイル時にいくつか処理を挟み込むことにする。こんなことをやりたい:

- コンパイルエラーを吐く ([] の閉じ忘れ)
- コードの最適化
- 使われる配列のサイズを計算する
- 無限ループの検出

最小限のコードを生成する

ということで以下のような感じになった:

```
defmodule Brainfux do
  defmacro defbf(name, raw_code) do
    # 諸々のチェックや最適化をここでコンパイル時に行う
    code = Brainfux.Preprosessor.process!(raw_code)
    quote do
      @spec unquote(name)(input :: String.t) :: String.t
      def unquote(name)(input \\ "") do
        state = %{input: input, output: "", ...(略) }
        %{output: output} = Brainfux.Executor.execute(state, unquote(code))
        output
      end
    end
  end
end
```

(実際にはもうちょっと色々やった。`bfn` で無名関数を定義する時と `do` ブロックの内容が同じなので共通化したり状態を表すために struct を定義したり)

5.8 テスト

テストを書こう。マクロのテストは、実際にそのマクロを使って結果を確かめるだけで良い:

```
defmodule BrainfuxTest do
  use ExUnit.Case

  test "defbf defines module function" do
    defmodule DefBfTest do
      use Brainfux

      defbf hello_world """
        (中略)
      """
    end

    assert DefBfTest.hello_world()       == "Hello, world!"
    assert DefBfTest.hello_world("foo")  == "Hello, world!"
  end
end
```

ほとんど README に書いたそのまま。**ドキュメントはテストになってくれる。** というかテストを書くつもりでドキュメントを書くといいのかもしれない。

コードの最適化が出来てるかどうかのテストは、最適化する処理を別のモジュール/関数に分けているので、それの単体テストとしてやれば良い。

マクロで生成するコードを短くしたので、テストもしやすくなったのである。

5.9　ハマりポイント 1: そもそも出来ない前処理

やりたいこととして

- コンパイルエラーを吐く (`[]` の閉じ忘れ)
- コードの最適化
- 使われる配列のサイズを計算する
- 無限ループの検出

を挙げたが、後半の 2 つは理論上不可能であった。以下のような反例がある:

- `,[[>]+[<]>-]`
  - 入力された数値ぶんだけ 1 を並べるプログラム
  - 入力に依存して使われる配列のサイズが変わるため、事前に把握することは不可能
  - 以下解説
  - `,` で入力を受け取り、`[>]` で 0 が現れるまで右に移動

    - + でそこを 1 にし、 [<] で最初の入力の場所の左の 0 まで移動
    - > で入力のある場所に戻り、- でデクリメント。ここが 0 になるまで以上を繰り返す
- ,---[+]
    - 入力された値が 3 以下なら終了するが、4 以上なら無限ループ
    - つまり入力に依存して無限ループするかどうか変わる

無限ループの検出は、そもそもこれが出来たらチューリングマシンの停止性問題が解けてしまうので理論上不可能であった……

5.10　ハマりポイント 2: コンパイルエラーのテスト

「コンパイルエラーになること」のテストは、テスト自体がコンパイルエラーになってしまうことがある。例えば

```
assert_raise CompileError, ~S( Unexpected "]" at position: 1), fn ->
  use Brainfux
  bfn ",].,["
end
```

ではダメで、 defmodule を挟む必要がある:

```
assert_raise CompileError, ~S( Unexpected "]" at position: 1), fn ->
  defmodule EndNotStartedBracket do
    use Brainfux
    def echo(str) do
      (bfn ",].,[").(str)
    end
  end
end
```

fn の中のマクロは基本的に fn を作った時点で呼ばれるが、 defmodule はその関数が呼ばれるまで評価されないのである:

```
# 適当にマクロを定義
defmodule MyMacro do
```

```
  defmacro one do
    IO.puts("macro invoked")
    quote(do: 1)
  end
end

import MyMacro

fun1 = fn -> one end
# => "macro invoked" が出力される

fun1.()
# => 1

fun2 = fn ->
  defmodule Fun2 do
    IO.puts("Fun2 defined")
  end
end
# => この段階ではまだ IO.puts は呼ばれない

fun2.()
# => ここで "Fun2 defined" が出力される
```

5.11 ハマりポイント 3: `@spec` があるか確かめる

今回のマクロでは `@spec` の行も生成している。これがちゃんと追加されていることをテストしたい。

一見 `Code.Typespec.fetch_specs/1` で良さそうだが、これは BEAM ファイルから情報を抜き出すため、テストコード中で `defmodule` したモジュールからは情報を取れない。

`defmodule` 中に `@spec` の中身を expose する関数を定義するという力技でなんとか解決した:

```
defmodule DefBfTypeSpecTest do
  use Brainfux
  defbf echo ",[.,]"

  # Getter for typespecs
  spec = Module.get_attribute(__MODULE__, :spec) |> Macro.escape()
  def specs, do: unquote(spec)
end
```

```
test "defbf adds typespec" do
  [{:spec, expr, _env}] = DefBfTypeSpecTest.specs()

  assert Macro.to_string(expr) == "echo(String.t()) :: String.t()"
end
```

5.12 まとめ

- 長いコードをマクロで生成するのはやめよう
- コンパイルエラーが起こることのテストは注意が必要
- `@spec` があることのテストは工夫が必要
- マクロは楽しい

第 6 章

Antikythera

弊社で開発を進めている OSS である Antikythera Framework
(`https://github.com/access-company/antikythera`)
について記述します。なお、この章の内容は skirino(`https://github.com/skirino`)
の発表資料: `https://skirino.github.io/slides/antikythera_framework.html#/`
をもとに記述しております。

また、詳細については (`https://hexdocs.pm/antikythera/api-reference.html`)
を参照ください。

6.1 Agenda

- Antikythera Framework とは
  - 簡単な経緯と Antikythera を開発するに至った動機について
- コア機能と利点
  - 他のフレームワークとの比較
- Antikythera の内部実装について
  - いくつかの design decision についての pros/cons のご紹介 Core features and benefits

6.2 Antikythera Framework とは

Antikythera Framework とは一言で言うと「Elixir Framework for in-house PaaS」です。フレームワークの詳細に入る前に、弊社で Antikythera Framework が開発される前に存在したいくつかの問題についてご紹介します。そして、その問題を受けて、Antikythera が採用したアプローチを紹介するという流れで記述します。

弊社では多くの Web サービスを開発する案件が存在しています。しかし、それらの開

発において下記のような特徴をもっておりました。

- 複数の Web サービスが異なる言語を用いて開発されている。
- 統一的な方式はなく異なる手法で運用されている。
- いくつかの Web サービスはインフラコストが非常に大きい

これにより、以下のことが課題として上がっていました。

- (1) 技術スタックとして統一性がない
  - 課題
    - コードを再利用する機会がほとんどない
    - 開発者が異なる案件の開発に携わるときに学習コストが大きい
  - 求められるもの
    - サーバサイド開発において共通の開発言語
- (2) 運用方式に統一性がない
  - 課題
    - script や知見を team をまたがって共有する機会がない
    - 各 team は日々の業務に忙殺されて"toil"を自動化する余裕がない
  - 求められるもの
    - 標準化/自動化された運用
      - 例えばデプロイ
- (3) インフラコストの増加
  - 課題
    - 冗長化やバッファのために、CPU リソースなどは多めに確保する必要がある
    - 監視や管理のためのコンピュータリソースが別途必要
  - 求められるもの
    - 全てのサービスで共有される一つのプール領域

上記の課題に対して我々が取ったアプローチが複数の web サービスを単独の ErlangVM cluster(Antikythera) でホスティングするという方法です。

※今後、Antikythera 上で動くそれぞれの web service を"gear"と呼称します。この gear は他の gear に依存することも可能です。

6.3 コア機能と利点

以下では、Antikythera の基本的な機能を紹介し、それにより得られるメリットについて解説します。

基本方針

全ての (gear の)beam file を同じ ErlangVM 上でロードすることを前提としています。これらの gear に割り当てられるコンピュータリソースは Antikythera 本体 (Elixir で記述) により制御されます。

アーキテクチャ

アーキテクチャは下記のようになっています。Load balancer によって Web リクエストは各 ErlangVM 上のノードに分散され、各ノード上で動く Antikythera によってリクエストが処理されます。詳しくは後述します。

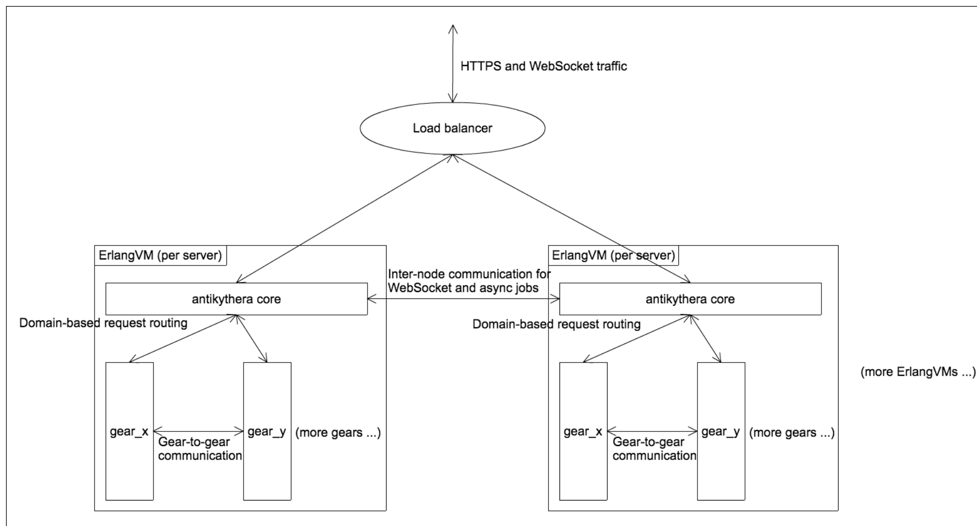

図 6.1: architecture

コア機能

マルチサービス向けのプラットフォームとしての機能

Antikythera は前述したように様々なサービス (gear で実装されるもの) が共存する形となっているプラットフォームです。それを実現するための機能として下記を提供しております。

- リソースの制御とアイソレーション

  - gear は他の gear の影響を受けて動作が阻害されてはならないため、各 gear が利用できるコンピュータリソースを制御、アイソレーションしております。
- 運用の自動化
  - hot code upgrade によりデプロイを行います。これにより、ある gear のデプロイのために他の gear が影響を受けることを減らしています。
- ビルトインされた関数群
  - ロギングやメトリクスレポートなどが備わっており、各 gear は特別な設定をすることなく最小限の監視機構を得ることができます。

Web フレームワークとしての機能

Web サービスをホスティングするために必要な下記の機能を提供しています。

- domain based routing
  - 各 gear には自動で個別のドメインが割り当てられ、web リクエストは domain に応じてどの gear がリクエストを受け取るかが決定します。
- path based routing
  - gear 内のどの module のどの関数が実行されるかは web リクエストの path に基づいて決定します。
- Phoenix(`https://phoenixframework.org/`) で提供されているものに似た関数群。

path based routing のイメージ図は下記となります。

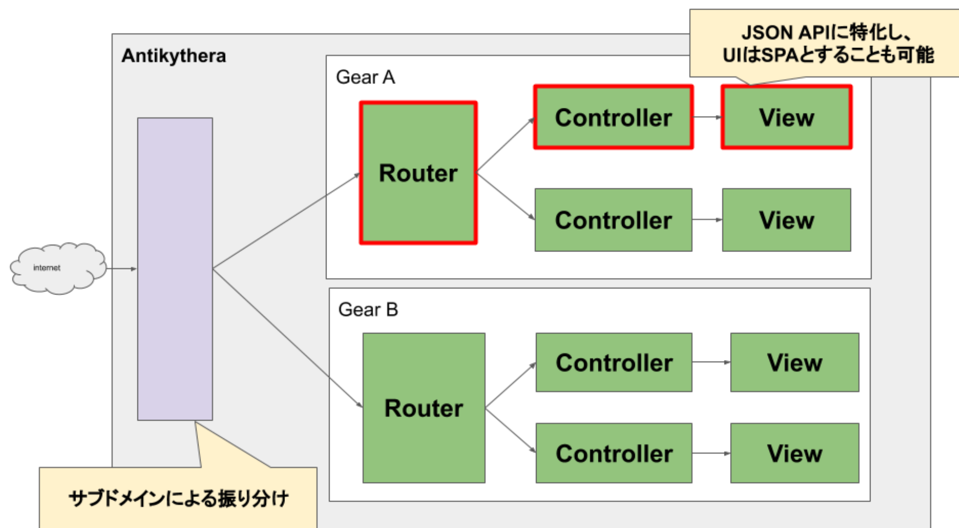

図 6.2: Path Based Routing

非同期処理機能

非同期処理は AsyncJob と呼ばれる仕組みにより提供されています。非同期処理は Antikythera 内で管理される job queue に登録することで、非同期で worker が queue から job を取り出して処理します。この仕組みにおいて特徴的な点は下記です。

- ビルトインされた分散 job queue
  - job queue は Antikythera のメモリ上で実現され、複数のノードで冗長化されます
  - これにより Antikythera は外部の queue に対して polling を行うことなく、Antikythera 内で処理を完結し、また worker に job を push する方式をとれるので無駄な polling 処理が発生しなくなります。
- 個々の job は任意の Elixir コードを実行可能です。
- one-off または定期的な job のスケジューリングが可能
  - 定期的な job の実行ルールは cron 記法で記述可能

利点

Antikythera は前述した弊社での問題に対して多くの点を解決することができる長所をもっています。具体的には下記です。

- 言語の統一
  - コードを再利用することが容易です。
- clsuter 管理の集約
  - gear 開発者はサーバ管理や dependency 管理から解き放たれます。
- インフラコストの削減
  - バッファのためのインフラコストを個々の案件で確保するのではなく、Antikythera cluster としてのバッファのみを確保すればよくなります。
  - 弊社の Prod 環境では AWS 上で構築され、25 個の gear を動かしています。
  - これらのサービスを C5.xlarge インスタンス 3 台のみで実現しています。
- その他の利点
  - gear 開発者はインフラに対しての運用が必要なくなります。(インフラ管理は Antikythera 本体の開発チームのみが担当します。)
  - 少ないコストで始めることができるのでプロトタイプ作成に最適です。

まとめると、もともと弊社で課題であった異なるサービス間での技術スタックの統一性の欠如やコードの再利用性の問題を Antikythera としてまとめ上げることにより解決しています。また同時にそれらが Antikythera という同じ cluster 上で動くことにより監視機構についても統一化が可能になり、冗長化のためのオーバーヘッドを削減することによりインフラコストの削減を可能にしています。一方で複数のサービスを同じ Cluster 上で動かすことから「マルチサービス向けのプラットフォームとしての機能」で記述した「リソースの制御とアイソレーション」が重要になります。これをどのように実現しているかについては後述の「Antikythera の内部実装について」で述べます。

以下では Antikythera と他のフレームワークを比較することで Antikythera の特徴的な性質について述べます。

他のプロダクトとの比較

Phoenix との比較

Phoenix も Antikythera も Elixir ベースのフレームワークです。Phoenix は多数のユーザを持つような一つの Web サービスを開発することを想定したフレームワークです。一方で Antikythera は複数のサービスに対する新しい実行モデルを提供するフレームワークです。

micro-services との比較

micro-services も Antikythera もサービスレベルでコードをセパレートしています。Antikythera は 1 つの ErlangVM 上でサービスを共存させることにより管理するた

めのサーバを削減し、運用を標準化することを目的としています。また、gear 間の communication は erlangVM 内の message processing によって実現されるためオーバヘッドが小さく済みます。Antikythera 内の gear は同じ PC 内でリソースを共有していることからより効率的にリソースを共有できます。

便利なツール群

gear は下記の機能を利用することができます。

- gear config
- logging
- alerting
- metrics reporting
- antikythera console

Gear Config

それぞれの gear に対して定義できる任意の `map` です。主に API Key などの認証情報や動的な設定を保持することができます。大本のデータは暗号化された JSON ファイルとして管理され、そのデータは ETS 上によりキャッシュされます。これにより、これらのデータに gear は実行時に高速にアクセスすることができます。

Logging

それぞれの gear は専用のロガープロセスを持っています。下記のログは自動でログに出力されます。また各 gear は任意のログを出力できます。出力されたログは AWS の S3 にアップロードされることで永続化され、gear 開発者用画面からダウンロードすることが可能です。

- HTTP リクエストを受け取った時とレスポンスを返す時
- WebSocket の接続と切断時
- AsyncJob の実行と終了時
- エラー発生時の stack trace

Alerting

例外が発生したときは Antikythera は予め登録された gear 開発者の ML にアラートメールを送信します。(大量のエラーが発生した場合はまとめて送ります。) これによりすばやくエラーに気づき、修正することが可能になります。

Metrics Reporting

様々な metrics が自動で収集され、metrics stroge(pluggble な実装となっているので選択可能です) に保持されます。収集される metrics の例としては下記です。

- Http response time(avg, 95%, max)
- AsyncJob の実行時間 (avg, 95%, max)
- executor pool の実行中/待機中の数
- ErlangVM の metrics

6.4 Antikythera console

OSS 部分ではありませんが、弊社内では gear 開発者の用の管理画面として Antikythera console という Web UI を提供しています。Antikythera console では下記のような各種 gear の設定を変更や確認ができます。- gear config の設定- log ファイルの取得- metrics の確認

この画面自体も gear で実現されています。下記にこの管理画面を記載します。

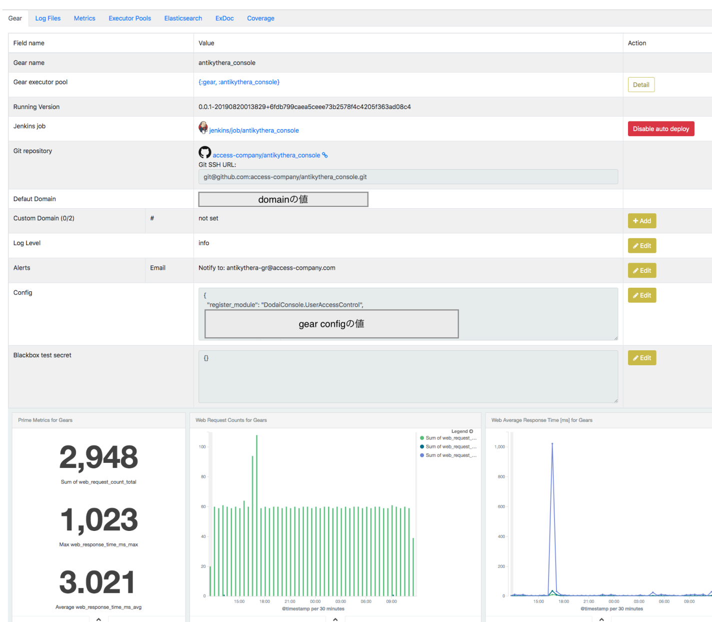

図 6.3: Antikythera Console の TOP 画面

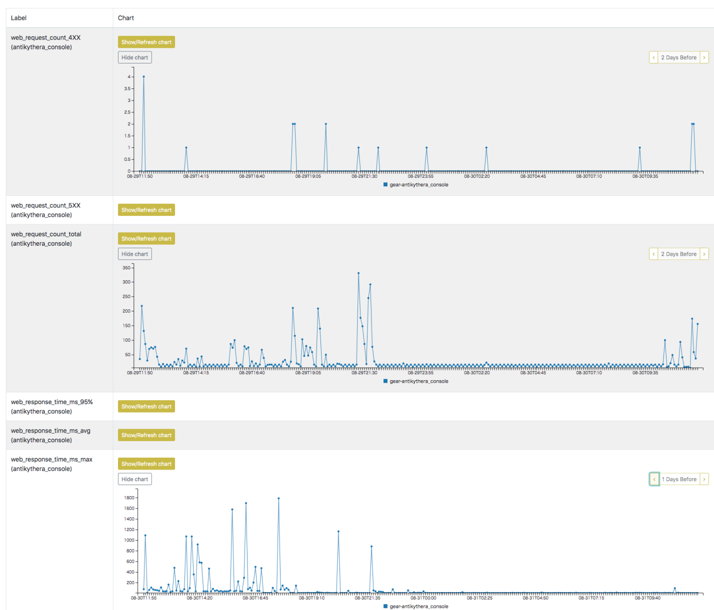

図 6.4: Antikythera Console の metrics 画面

なお、この管理画面の gear については OSS として公開予定です。

6.5 Antikythera の内部実装について

ここからは個々の Antikythera の機能がどのように実現されているかやデザインデシジョンについて記述します。

Architecture

Http request handling

- http 1.1 server としては cowbody(`https://github.com/ninenines/cowboy`) を利用しています。
- cowboy の router を利用して、domain based routing を実現できます。
- path based routing は macro により生成された function clauses とのマッチングにより実現しています。

Gear-to-gear(g2g) communication

gear のコードの処理は web request の処理もしくは他の gear からの呼び出しによって実行されます。この際にこの 2 つの IF として同じ形をとる仕組みを採用しました。つまり、gear は別 gear のコード呼び出す時に Antikythera が提供する G2g の仕組みを使って呼び出しますが、Antikythera はその呼出を http like なものとして G2g に受け渡します。これにより gear 側は web request の処理であっても G2g の処理であっても同じコードで実現することが可能であることになります。

このことは Antikythera において gear を記載するということは直ちに他の gear への処理の公開を可能にする下地ができるということを意味しています。つまり、Antikythera 内での gear 間においてコードの共有を加速することができます。

Clustering

Antikythera は複数の ErlangVM が起動している node により cluster を形成しています。cluster を維持するため、それぞれの node は定期的に下記を実施しています。

- node のリストを取得
  - infrastructure の API(例: AWS の auto scaling group から EC2 のリストを取得する) を利用して現在の node のリストを取得しています。
  - cluster の member は上記により取得されてた node のリストとして管理されます。
- cluster にまたがった process registry としては syn (`https://github.com/ostinelli/syn`) を利用しています。

EAL

EAL とは Environment Abstraction Layer の略です。これは、Elixir の behaviours を利用して、pluggable なインフラ依存の機能を提供するレイヤーになります。これにより、Antikythera は特定の cloud provider に依存せず様々なクラウド上で動作することを目指しています。実際の実装はコンパイル時に決まります。詳細は AntikytheraEal.ImplChooser

`https://github.com/access-company/antikythera/blob/master/eal/util/impl_chooser.ex` を参照ください。

弊社では AWS 上で Antikythera を運用しており、behaviours の AWS 実装として `antikythera_aws`(`https://github.com/access-company/antikythera_aws`) を提供しています。

Resource Control

前述したように Antikythera では複数のサービスを同じ Cluster 上動作することから、いかに「コンピュータリソースの制御とアイソレーション」が実現するかが重要になります。ここで Erlang の process は実行コンテキストやメモリは完全に他の Erlang process から切り離されています。そのため、実行可能な Erlang process の数を用いてリソースを制御することが可能になります。(こういった特徴があるからこそ私達は elixir を採用しました。)

Executor pool

process 数を管理する上での制御単位です。Executor pool には下記の 3 種類が存在します。

- web request handler
- websocket connections
- asyncjob runners

たとえば web request handler として executor pool は各 gear あたり、5 個割り当てられています。そのため、各 gear は 5 つの web request を同時に処理することができます。なお、この制限は Antikythera cluster の各 node あたりの制限であり、node の数が増えれば cluster 全体として処理できる並列数が変わります。(同時に処理できると書きましたが、実際には Erlang scheduler がプリエンプティブに process の実行をスケジューリングします。)

アイソレーション

全ての gear が同じ ErlangVM 上で動いている関係上、それらは完全にはアイソレーションされていません。つまり、gear は他の gear のコードを実行できますし、他の gear の秘密情報を取得できます。(ただし、コンパイル時に静的なチェックで他の gear のコードを呼び出していないかなどのチェックは行っております。) だからこそ、Antikythera を"in-house PasS"として我々は運用しています。つまり、社内の web サービスのみを hosting することにより信頼できないコードが入り込むことを禁止しています。

その他の制限

リソースを共有する仕組み上、executor pool における処理では下記の制限を設けています。

- process における heap memory の最大値
- web request 処理や job の最大実行時間
- AsyncJob queue のアクセスに対する rate limit

Deplendency 管理

全ての gear は同じ ErlangVM 上で動作しています。そのため、それらは同じ library の version と設定を使う必要があります。deps/config を管理するためには"Antikythera instance"が必要になります。

Antikythera Instance

特定の Antikythera cluster の deps/config を管理するための mix project です。(必然的に Antikythera に依存しています。) 全ての gear は Antikythera Instance に依存していますので、gear は同じ deps/configs に依存することになります。

Deployment model

Antikythera Instance は OTP release としてパッケージされ、release されます。それぞれの gear は OTP application としてパッケージされ、release されます。(gear は OTP release としては含まれません) Antikythera の各 node は (なんらかの build システムで生成された)OTP release や OTP application の存在を監視しており、新しい version を見つけた場合はそれらを使って自分自身を update します。

AsyncJob

AsyncJob Queue

AsyncJob の queue は Antikythera の memory 上で保持されます。必然的に永続化の方法について検討する必要がありますが、Antikythera はそれらを複数のノードで分散して保持することで冗長化しています。具体的には raft(`https://raft.github.io/raft.pdf`) という分散アルゴリズムにより冗長化を制御しています。この実装は `raft_fleet`(`https://github.com/skirino/raft_fleet`) によって実現されています。

なお、Queue の永続化の方法として DB を選択するという案もありました。しかし私達は DB に対する過度な polling をさけ push ベースのインタラクションを実現するために上記の方法を選択しました。

AsyncJob Worker

ErlangVM 上に存在します。重要な点として、worker 専用のノードを作成していない点です。全てのノードを等価にすることでクラスター管理をシンプルにしています。

静的解析

Antikytheras では複数のサービスが同居する PF であるため、本質的には他のサービスのコードに対して別のサービスのコードにアクセスできるようになっています。第一義的には in-house PaaS という位置づけであり、悪意のあるコードがない前提での運用となりますが、意図しないアクセスが発生してしまう可能性はあります。Antikythera では mix compiler(`https://hexdocs.pm/antikythera/Mix.Tasks.Compile.GearStaticAnalysis.html`) により gear code の規格適合性をチェックしています。例えば、module のネーミングルールや禁止された関数を実行していないかです。各 gear の module は gear 名の module で始めるルールとなっています。ある gear のコード上にその gear 名の module でない module に対する関数呼び出しがある場合は、他の gear のコードへの不正なアクセスとしてコードコンパイル時にエラーにします。(他の gear のコードに依存する場合は Antikythera が用意している G2g の枠組みで実施します。)

6.6 まとめ

Antikythera フレームワークは in-house PasS として、企業内で多くのサービスを構築しているような状況に向いたフレームワークです。Antikythera を使うことにより、コスト面、運用面や社内の技術統一による知見の共有などにおいて多くのメリットを享受することができます。

弊社では多くの gear を Antikythera 上でホスティングすることにより、gear 開発者はアプリケーション開発にリソースを集中することができ、Antikythera 以前に発生していた多くの問題が解決され、効率的に開発を進めることが可能になりました。例えば、Elixir/Erlang の update は Antikythera チームが一括して適用、検証、デプロイを行います。gear 開発者はそのことにおいて追加作業が発生することは非常に稀です。(Elixir の後方互換性がない update が行われた場合は発生する可能性がありますが、Antikythera の本体部分の修正で巻き取れることがほとんどです。)

今後、Antikythera ではさらに gear 開発を加速することができる機能やパフォーマンス向上の仕組みを順次追加する予定です。それにより Antikythera 上で全ての gear が一度にそのメリットを享受することができます。

もし、Antikythera に興味がある方は Users Mailing List

https://groups.google.com/forum/#!forum/antikythera_users までお問い合わせください。

6.7 著者

高橋正樹

第 7 章

STM32 マイコンの Ethernet ドライバ

7.1　おことわり

この記事で解説するコードは、ドキュメントに載せる目的で公開されたサンプルコードではなく、ST マイクロエレクトロニクス社が開発者向けに実用目的で提供しているコードです。ライセンス承諾なしに再掲載できません。

この記事ではコードの処理の概要を日本語で解説します。コードを読みたい方は下記 URL から、ライセンスを確認の上、コードをダウンロードいただくようお願いいたします。

STM32CubeF7 Firmware Package
https://www.st.com/ja/embedded-software/stm32cubef7.html

LwIP TCP/IP stack demonstration for STM32F2x7 microcontrollers(STSW-STM32060)
https://www.st.com/ja/embedded-software/stsw-stm32060.html

System Workbench for STM32
http://www.openstm32.org/

7.2　話のさわりと目的

初めまして。株式会社 ACCESS に務める三原克大と申します。2003 年入社 (勤続 16 年強) で、直近 9 年強は組み込み家電向けプロプライエタリ (クローズドコード)Web ブラウザ NetFront Browser のメンテナンスを担当していました。また最近では上記より HTTP/TLS 通信ライブラリと JavaScript インタプリタのみ取り出した NetFront Agent の作成も行なっていました。

NetFront Browser/NetFront Agent は、その出自から、プラットフォームにモダン OS を必要としません。RTOS でタスク管理するボードにも載ります。僕の実績としては以前にμ ITRON に商用 TCP/IP スタックを組み合わせたボードに載せました。

それでは NetFront Agent のデモ用に、もっと入手しやすい STM32 マイコン搭載ボードに載せてみよう！ という話になりました。チーム内に STM32F746G-Discovery https://www.st.com/ja/evaluation-tools/32f746gdiscovery.html があったものですから、それをターゲットに作業開始。しかし予算がないので開発環境は無料のもののみ使用することとなり System Workbench for STM32 で開発を始めました。

といっても、まずボードで基本的なネットワーク通信ができなければいけません。移植の前にネットワークを動かすためにドライバを実装しようとして、つまずき、あとでサンプルコードを見たら必要なものが全て書いてあった……というのが顛末です。

組み込み向け開発でマイコンとライブラリの仕様を誤解するとまるで動かないという失敗を追体験いただくのが、この記事の目的です。

7.3 サンプルコードの必要性

System Workbench for STM32 には、STM32 の各種マイコンに対応できる形で、次のものが付属します。

STM32FXXXxx HAL Drivers
STM32 マイコンのデバイス (主に I/O) に一貫したプログラミングインタフェースを定義するため、ST マイクロエレクトロニクス社が作成した、デバイス操作を抽象化するライブラリです。

FreeRTOS
名前の通りのオープンソース RTOS です。現在の System Workbench for STM32 には 2017 年に Amazon が買収して MIT ライセンスとした Ver.10 が付属します。

LwIP
組み込む機器向けオープンソース TCP/IP スタックです。ROM/RAM それぞれ数十キロバイトで動作します。

そして STM32F746G-Discovery ボードには、STM32F746NGH6 というマイコンが載っていて、有線 Ethernet の NIC が搭載されています。

これだけあればネットワーク機能は完備、と思うかもしれませんが、そうではありません。先ほど、STM32 の各種マイコンに対応できる形で、と書きました。マイコン毎に異なる仕様に依存したコードが System Workbench for STM32 には含まれていないのです。

具体的には下記の機能を実現するコードがありません。

- LwIP が HAL を叩いてパケットを送る機能
- パケット受信を割り込みで認識して、LwIP の処理を開始する機能
- TCP/IP のプロトコル処理を担当する RTOS のタスク

これらを「改めて」コーディングしなければボードに付属の NIC で TCP/IP 通信できません。

でも、そんなものどうやってコーディングするんだ? という話です。

初めてボードで開発する人のために、TCP/IP 通信できるためには何を実装すればいいのかを示すサンプルコードが必要となるのです。

ネットではサンプルコードを STM32CubeMX というツールで作っている記事がよく見つかります。
https://www.st.com/ja/development-tools/stm32cubemx.html
それらの記事には、よく整った GUI のスクリーンショットが載っています。

そのスクリーンショットを見て、きっと有償に違いない、と勘違いしたのが道を逸れた最初の間違いです。その勘違いを前提に、無償のサンプルコードを探し始めました。

最初に見つけたのが、型番違いのマイコンである、STM32F2x7 を対象としたサンプルコードのパッケージの LwIP TCP/IP stack demonstration for STM32F2x7 microcontrollers でした。

I/O 操作は HAL Drivers で抽象化されているからコンパイルエラーを取り除くだけで動くだろう、と勘違いしたのが次の間違いです。

コンパイルエラーを取り除いてバイナリイメージを作ってボードに焼き込んでも、ちっとも動きません。

途方にくれたところで、ボードに乗っているマイコンに対応する正しいサンプルコードの STM32CubeF7 Firmware Package を見つけました。そこには正解がちゃんと書いてあったのです。

この経験をネタにしてやる! と思った結果が、この記事です。よく言えば、開発の工程を再体験できた、のかな?

7.4　相次ぐトラブル

マイコンとライブラリの仕様を間違えると何が起きるのか。ここから、起きたトラブルと、それに対する正解を見ていきます。

Tick カウンタが進まなくてタイムアウトが終わらない!

サンプルコードのブートストラップから TCP/IP 通信に至る流れは、次の図のようになっていました。

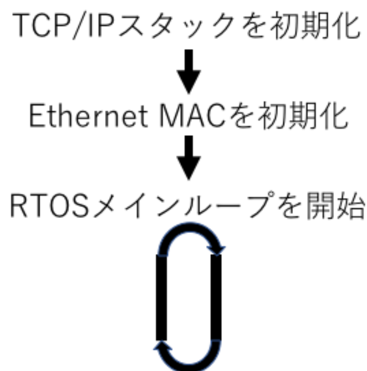

図 7.1: サンプルコードのブートストラップ

ボードで走らせると、TCP/IP 向け RTOS タスクを作成した後に Ethernet MAC を初期化する段になって沈黙しました。

デバッガで確認すると、Ethernet MAC を初期化を指示して一定時間の待ちに入ったところで、Tick カウンタが全く進まずタイムアウトが終わらないのです!

ここで HAL Drivers の Tick カウンタと FreeRTOS の関係を説明する必要があります。

HAL Drivers で Ethernet MAC を初期化する際、タイムアウトは HAL Drivers の Tick カウンタで測ります。

HAL Drivers の Tick カウンタは他のソフトウェアに非依存です。デフォルト設定では SysTick を使用して ms 単位でカウントしています。

ところが FreeRTOS は SysTick を自らのタイマのために占有します。

そのため、**FreeRTOS を使う場合、SysTick 以外のタイマを HAL Drivers が独自に持つことが強く推奨されています。**

HAL Drivers には他のタイマを使うサンプルコードも提供されています。ここで STM32F746NGH6 の TIM を使用するサンプルコードがあったので、それを有効化しました。

ところが、です。TIM を使った Tick カウンタが動作することを確認して、いざ本番のコードを走らせると、Ethernet MAC を初期化する段になって Tick カウンタが進まない、というか、**TIM の割り込みが入らない**…… なぜだ?

あれこれ探すと、FreeRTOS のソースコード

/FreeRTOS

/Source
/tasks.c
の関数 vTaskStartScheduler() に不吉なコメントが……

転載できないので要約を日本語で書くと、タスクスケジューラが起動する前あるいは起動処理中に割り込みが発生することを避けるため、@<strong>{FreeRTOS のタスクやセマフォ等のリソースを生成するとタスクスケジューラの起動が完了するまでシステム全体で割り込みを禁止するのです}。

先ほどのブートストラップでは、Ethernet MAC を初期化する前に TCP/IP 処理の RTOS タスクとセマフォを生成していました。はい、これでタスクスケジューラを動かすまで割り込みが入らなくなるのです。

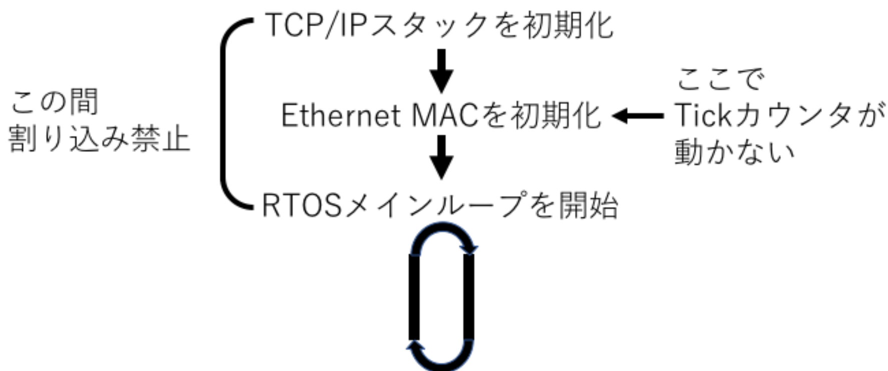

図 7.2: どうして Tick カウンタが進まないのか

Ethernet MAC を初期化したらパケットを受信する可能性もあるので TCP/IP スタックは先に初期化する方がいいです。しかし TCP/IP スタックに関係する RTOS タスクとセマフォを生成すると Ethernet MAC を初期化できません。

これ、なんのパズルですか……

少し考えたとき、これが答えかな、という感触はあり、たしかに動いたのですが、正解である STM32CubeF7 Firmware Package を見たらきちんとそう書いてありました。

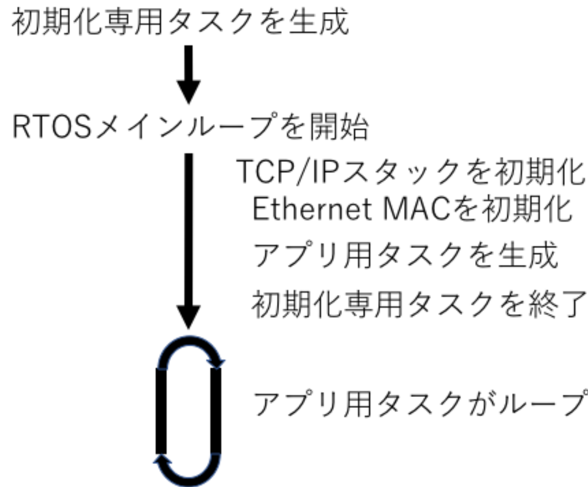

図 7.3: 正解: Tick カウンタが進むブートストラップ

TCP/IP スタックと Ethernet MAC を初期化するコードを FreeRTOS のタスクにして、タスクスケジューラを起動させてから実行するんですね。FreeRTOS は RTOS としては高機能で、タスクスケジューラから呼び出されたタスクが動的にタスクを登録できます。なのでアプリケーションのタスクを後から登録できるんです。

他のマイコンボードに詳しい人から、その手順はおかしい! とツッコミを受けました。組み込み開発に詳しい人が見たら、普通はそう思うでしょう。でも FreeRTOS を使うと、それしか通れる道がなかったのです。

システムは動いているのに Ethernet MAC だけ割り込みが入らない!

いろいろ障害を乗り越え、TCP/IP スタックを初期化し、Ethernet MAC を初期化し、アプリケーションの RTOS のタスクが動くところまで来ました。

じゃあ動作を確認しよう、と対向 PC で ping コマンドを打ちます。

すると応答がありません……

ボード側はソースコードが全て公開されていてデバッガの配下で動いています。そのため挙動は全て見えます。デバッガで一つ一つ確認すると、パケットが届いているはずなのに Ethernet MAC の IRQ ハンドラが呼び出されていない……

ここで Arm Cortex-M シリーズ CPU の割り込みハンドラを説明する必要があります。

多くの CPU で割り込みハンドラを CPU から呼び出してもらうには、ベクタテーブルをコーディングし、最初に呼ばれるコードをアセンブラで書いてメイン処理を行う C 言

語関数に引き渡す必要があります。

それが Cortex-M シリーズ CPU では開発者の負担を減らす仕様改変が行われました。ベクタテーブルはコンパイラが自動生成します。ベクタテーブルに登録されるのは C 言語関数です、アセンブラコードは必要ありません。ある IRQ 番号に割り当てられたシンボルが "FOO_IRQn" だとすると、コンパイラは "FOO_IRQHandler()" という名前の C 言語関数を探索してベクタテーブルに登録。ボードで実行したとき、割り込み発生時には FOO_IRQHandler() が直接呼び出されるのです。本当に楽になりました。

STM32F746NGH6 マイコンで Ethernet MAC がパケットを受信した場合の IRQ 番号に割り当てられたシンボルは "ETH_IRQn" です。すると割り込みハンドラは C 言語関数 ETH_IRQHandler() です。それがパケットが NIC に届き LED が点滅しても ETH_IRQHandler() が呼び出されません。

ためしに TIM の割り込みハンドラを確認すると定期的に割り込みが入っています。**システムでは割り込みが禁止されていないのに Ethernet MAC の割り込みだけ入らない**のです。

これは本当に参りました。いろいろ探してもヒントが見つかりません。お手上げでした。

参っていたところに STM32CubeF7 Firmware Package を見つけて正解を見たら、ファイル
/Projects
/STM32746G-Discovery
/Applications
/LwIP
/LwIP_HTTP_Server_Netconn_RTOS
/Src
/ethernetif.c
にコメントがありました。

Ethetnet MAC で DMA が受信パケットを格納するメモリはメモリキャッシュが無効化されていなければいけないと。

これは、僕が最初に参照したサンプルコードの対象マイコン STM32F2x7 シリーズと、STM32F746G-Discovery に搭載されている STM32F746NGH6 マイコンの CPU コアの違いが影響します。

STM32F2x7 シリーズは Cortex-M3 コア、STM32F746NGH6 マイコンが属する STM32F74xxx シリーズは Cortex-M7 with FPU コアを搭載します。Cortex-M7 with FPU コアには Cortex-M3 コアにはないデータキャッシュメモリがあります。

そういった CPU コアの仕様を理解した上で、Ethernet MAC が DMA を受信するメモリアドレス領域はデータキャッシュが無効化しなければいけないのです。

単純にリファレンスマニュアルを読んだだけでは書いてなかったような気がするのですが、僕が見落としたのでしょうか。まったく**サンプルコードがドキュメント**です。

メモリキャッシュを無効化すると一口に言っても、その制御は単純な C 言語プログラミングの範囲を超えます。

実際に STM32CubeF7 Firmware Package で行なっているのは

- リンカスクリプトで Ethernet MAC が受信パケットを格納するためだけのセクションを定義する。
- データを格納する C 言語配列に対して配置するセクションを指定する。その指定法は C 言語標準仕様にはありません。ARM CC、IAR C コンパイラ、GCC、全部記述方法が異なり、コンパイラが定義するマクロを参照して条件コンパイル。
- MPU 初期化関数で当該セクションに対してデータキャッシュを無効化

まったくの総合芸術です。CPU コアも開発環境も隅から隅まで知っていないと実装できません。

これを一から実装するのは無理でした……

7.5　マイコン向けプログラミングとは

ここまでお読みになられた読者は、マイコンでネットワーク通信するのはなんと面倒臭いのかと思われたでしょう。

というか、**ここまで読んでいただけるだけで奇特な心の広いお人です。**

基礎的機能が既存コードとして存在しないマイコンボードでのシステム開発は、「当たり前のこと」を実現するだけでも相当に手間取ります。

かつてのガラケーなどの開発においては、プログラマが死ぬ思いをして基礎的機能を実装していました。今まで記したことは、その、ほんの初歩です。

今は CPU とメモリが良くなり、組み込み機器でも Linux が載るようになりました。メーカが作ったドライバが最初からついていて、POSIX API に沿ってプログラミングすれば大抵のことができます。Linux の普及は組み込みプログラミングを大きく変えました。

それでも、と言わなければいけない時がある。どうしても非力なハードウェアで動かさなければいけない時がある。その時はプログラマが頑張るしかない。

どうしてもそれでなければいけない用途のために頑張る。それがマイコンでプログラミングするということです。

でもマイコン好きな方々から見れば So What? (それがどうした?) ですよね。

やりたいから、やるんです。

あとがき

「ACCESS テックブック」をお楽しみ頂けたでしょうか。ACCESS では一緒に働けるエンジニアを大募集しております。職種は、本書のように、スマホアプリ、WebFront、React、クラウド、Elixir、Deep learning、組み込み、C++ などまさに様々。いずれも本書のようなマニアックな話ができるスペシャリストと一緒に仕事ができます。

興味のある方は、是非、以下からご応募下さい。

https://www.access-company.com/recruit/

ACCESS テックブック

2019 年 9 月 22 日　初版第 1 刷　発行
著　者　ACCESS 技術書典同好会
印刷所　日光企画

ACCESSテックブック

株式会社ACCESSは、ブラウザエンジン開発、IoT、クラウド、人気スマホアプリ、ハードウェア、ネットワークソフトウェアと、とても幅広い技術分野に対して、それぞれ専門的な技術を持って開発をしています。本書では、ACCESSのエンジニアの、深く、マニアックな話をまとめました。アプリアーキテクチャー、国際標準化、Deep Learning、React、Elixir、マイコンと、てんでばらばらの内容ですが、全てACCESSの業務と何らかの関わりのある内容です。各章ははそれぞれ完結しているので、好きな章からお楽しみください。

発行：株式会社ACCESS