WORD

From College of Information Science

WORD編集長も大統領選挙から撤退します号

目次

あなたとドメイン、どんな物語がありますか？

文　編集部　ぬっこ (@nukkonukko11)

1　あなたとドメイン、どんな物語がありますか？

こんにちは、ぬっこです。全人類ドメインを持っていると思いますが、あなたは最初にドメインを取った時のことを覚えていますか？

僕は高 3（17 歳）の 5 月あたりに取ったと思います。元々ゲーム開発をしており、それより前は映像制作などをやっていたのでそのまとめのポートフォリオサイトが欲しく取得しました。当時（というか今も）web は全く詳しくなく手探り状態で静的なサイトを立ち上げ、その場合 AWS とかいうやつの S3 と CloudFront で良さげだという結論に至り、そのまま良い感じにやってくれる Route53 でドメインを取りました。

AWS はレジストラの中ではかなり料金が高く、mizuame.works は年間 36 ドル/約 5,400 円となっています。値段的に Cloudflare に移したほうが良い気がしているのですが、DNS 引き継ぐのがだるそうという理由でやっていません。メールサーバーをさくらで契約しており、いじりたくないというのもあります。

さて、今回は Twitter[*1] 上でドメインについてのアンケートを発射したところ、多くの方に反応を頂き、N=110 位のデータが集計できたのでその結果をご報告します。半手動でやっているため、集計ミスがあったり全員のお話を載せることができなかったりすることをお許し下さい。

1.1　Q1: あなたが初めてドメインを取得したのは何歳頃？

まず、大体何歳頃に取っているかをグラフにしました。ご覧の通り、かなり分散していますが 14-15 歳と 16-17 歳が一番多いですね。Twitter を見ていると中高生の方でドメインを取っている方をよく見かけるので個人的には納得の結果です。また、12-13 歳で取ったという方も居て驚きました。

1.2　Q2: レジストラはどこを利用していますか？

次にレジストラですが、ぶっちぎりで一位は Cloudflare でした。やはり卸値で売っていることもあり、最安なので使っている方が多いという結果になりました。

お名前.com が次に多いのですが、アンケートを見ていると “お名前がうるさいので移管しました” や “スパムのような営業メールに飽き飽きして別のドメインを取った” などの意見も見られました。持っている方でも移行したいと話している人がいました。広告でよく 1 年 0 円を謳っていることもあり最初は安いみたいですね。

[*1] 自称 X

1 年でレジストラを移せば 1 年間は無料で使えるため､そのような使い方をしている人もいるそうです。最安を攻めるならお名前.com で取得→ 1 年で Cloudflare に移行が正解なんでしょうか?

意外だったのは AWS で取っている人が少ないことですね。高いですし個人で使うメリットがあまりないのでしょうか？

1.3 Q3: そのレジストラを利用しているのはなぜですか？

Cloudflare は安いからという意見が一番多かったですが、それ以外にも “1 番は ZeroTrust を用いて簡単に認証を実装できるため” や “workers が使えるから” などの他の機能との親和性が高い事をあげている人もいました。

また、同時に VPS を借りている人からは “VPS と一緒に管理･請求されるから” という意

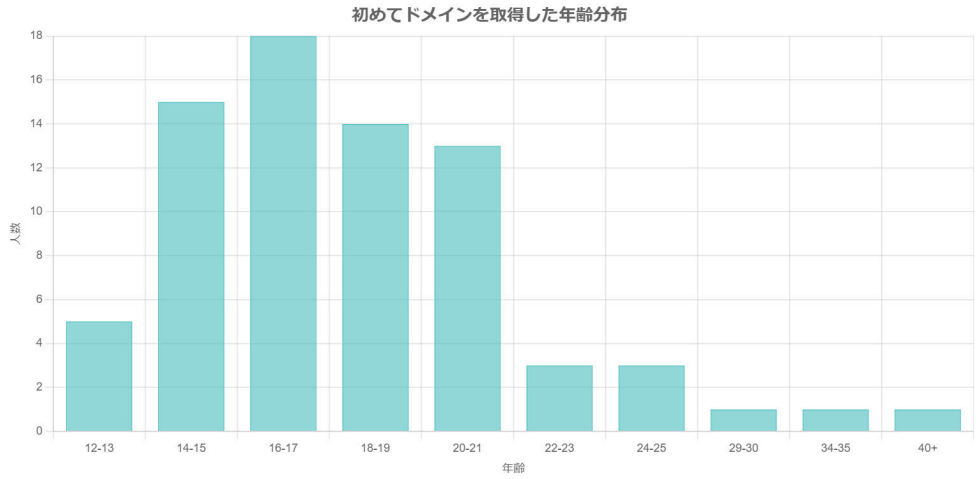


図 1　年齢分布

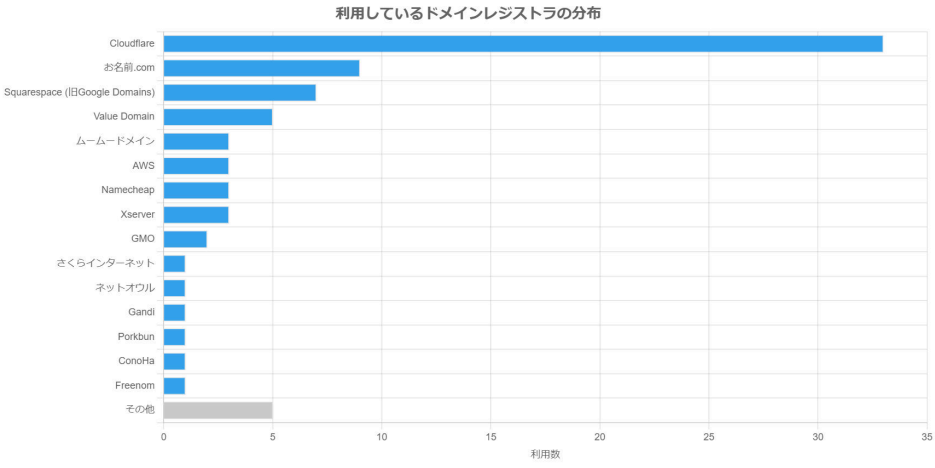


図 2　レジストラ分布

見もありました。あまり取り扱い事業者の少ない TLD の場合はその TLD を取得するために特定のレジストラを選んだといった意見もありました。

面白かったのは “当時 LINE Pay などもなく高校生でクレカも V プリカも作れなかった歳だったのでウェブマネー支払いができるレジストラを” とあり、クレカ支払いや PayPal 以外が可能なレジストラがあるのを初めて知りました。

1.4 Q4: いくつドメインを持っていますか？

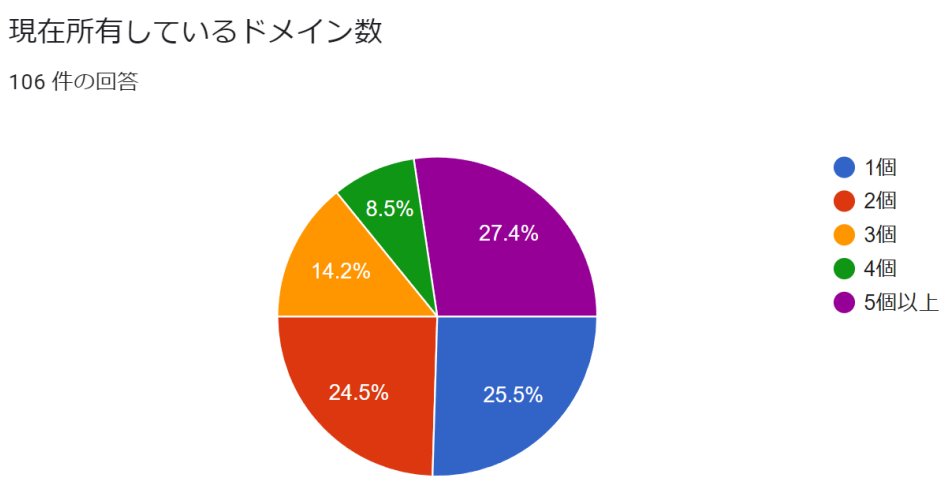


図 3　ドメイン所持個数

次にドメインの所有数ですが一番多かったのは 1 つ所有でした。そして、ほとんど同じ割合で 2 個所有でした。驚きなのが次に割合として多かったのは 5 個以上所有している人でした。そんなにドメインを持って何をやるのだろう? という疑問があるのですが、どうやら様々な団体の活動用に取っているケースがあるそうです。また、“遊び” や “コレクション”、“kuso.domains に乗せようと思った” などの理由もありました。

1.5 Q5: なぜドメインを取得しましたか？

話の続きになりますが、なぜそもそもドメインを取ったのかという話です。圧倒的に多いのはホームページ用、ブログ用、ポートフォリオ用ですね。また “URL 短縮用” や “Minecraft 用”,“Misskey 用” という理由も見られました。面白かったのは zip ドメインを持っている方は “zip が公開された時にすぐ閃いたから。” との意見もありました。zip ドメインは Twitter 上でも物議をかもしていた記憶があります。アンケートに答えてくださった方は “windows-installer.zip” というなかなかに良いドメインを持っているそうです。

1.6 Q6: なぜその TLD にしましたか？

次に TLD は何を取得したかですが、一番多かったのは.com でした。続いて .dev、.net、.jp でした。.com と .net、.jp は有名なのでわかるのですが.dev が 2 番目に多いあたりエンジ

ニアっぽいなと思いました。個人的面白ドメインは.zip、.wtf、. みんな ですかね。

その TLD にした理由においては.com や.net では “ポピュラー”、“一般的だから”、“名実ともに一番強いので”、“権威” などといった意見が見られました。.dev を取っている方の意見としては “パソカタ人間が最初に取る独自ドメインといったらこれだから” や “ソフトウェアエンジニアっぽいから” などの意見がありました。. みんなを取っている方は “かわいいから”、.zip を取っている方は “ .zip を取れたから。xslx.zip / docx.zip が一番欲しかった。”、.wtf を持っている方からは “面白いので” との意見を頂きました。また、2LD[*2] と TLD と合わせて読むことで何らかの言葉にしている人もいました。

[*2]（2LD は Second-Level Domain の略で、例えば example.com の “example” の部分を指します。）

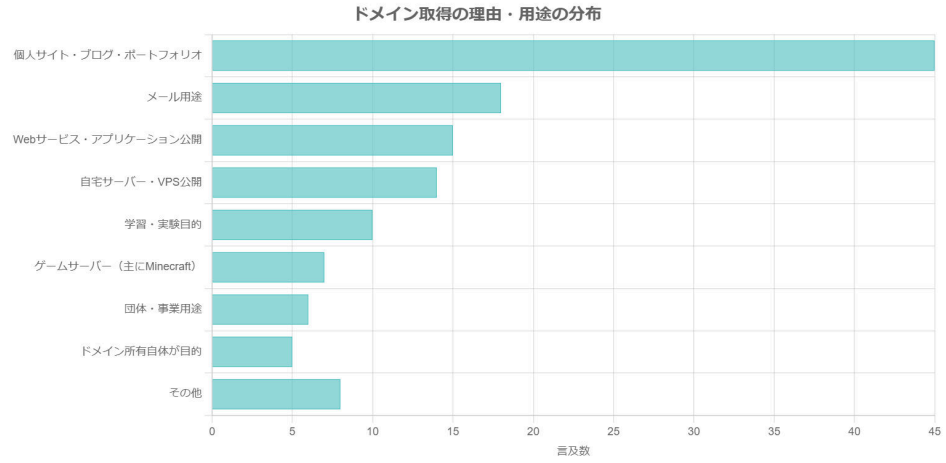


図 4　ドメイン取得の理由

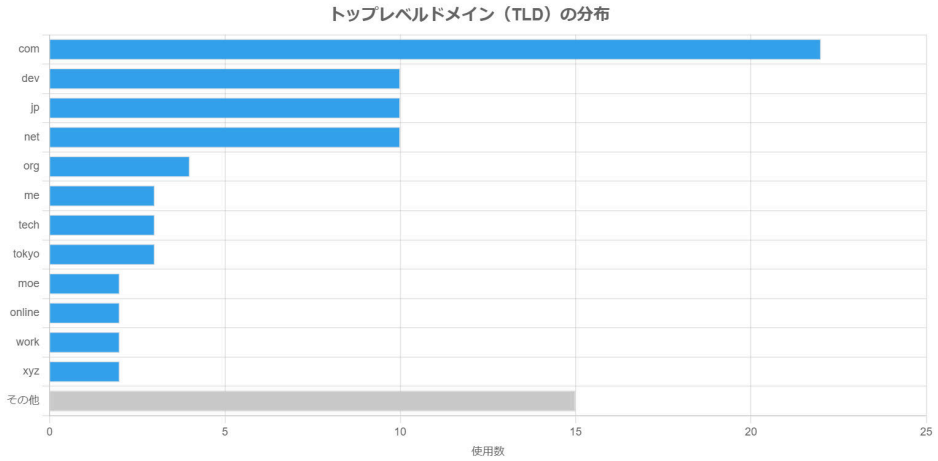


図 5　TLD の分布

1.7　Q7: 維持費は年間いくらですか？

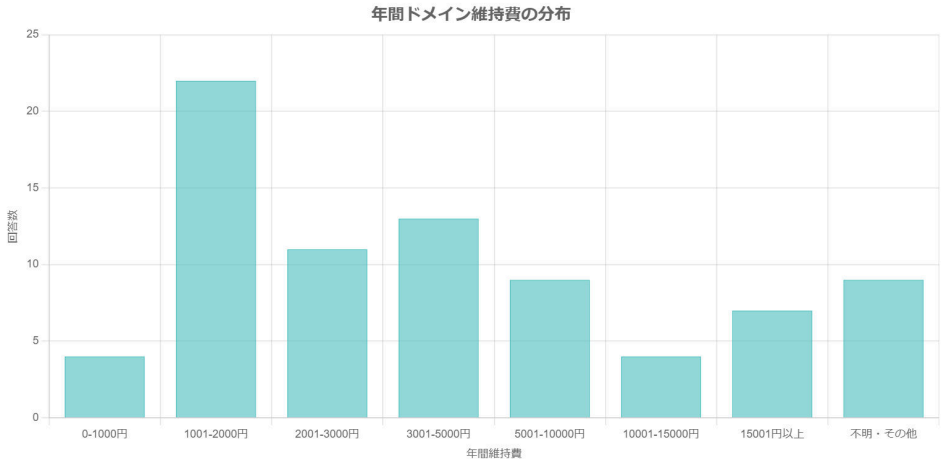


図 6　年間維持費分布

1 ドル 150 円として計算しました。1 ドメイン 1,001 円から 2,000 円がボリュームゾーンとなりました（図 6）。複数ドメインを持っている人は基本 1 万円超えで恐ろしいと思っていたのですが、よくよく考えたら僕も今年 AWS からドメインだけで 72 ドル請求されていました。

1.8　いかがでしたか？

Cloudflare が本当にレジストラとして一強なのが見て取れました。ユニークなドメインを取っている方もいて、見ていて興味深かったです。

アンケートにご協力いただいた皆様へ

本記事の作成にあたり、以下の方々からアンケートへのご回答をいただきました。心より感謝申し上げます。（ハンドルネーム・敬称略）

eee, kichi2004, ikeji, Rona, nandenjin, n4mlz, tarctarx, ふろど, cely_chan, いなにわうどん, Mamimal, ゆー, Cra2yPierr0t, Lex, onokatio, yhara, namachan10777, honahuku, ぼいど, オオタガオ, jitenshap, わたすけ, Jugesuke, さらだぼぉる, Moneto, walnuts1018, ksatoshi, もぐもぐ, s3i7h, Ogawa3427, よね/Yone, こるく, @ssakuo_jp, じあえんそ, motorailgun, 間瀬 BB, You&I, Flast, wt, mimifuwa, apis, おかし, esehehelp, hidori, たなぼた, yude, らて, ふぁ, ゆき, とみすけ, ちゅるり, セツナ, 西川岬希, T マート店長, じゃがびぃ, tomo0611_dev, みずさわ, Chi, choko1229, n01e0, reishoku (Twitter: @reishoku___), ゆずは, まつ, daken, かたぎりあまね, Soprano, Miriel @mirielnet, Tskikoh, 雷, かずっち, チズチズ, huggy, Ryoga, 橘いおね, ami, 岬, aida0710, Naxii, kazubu, めがね, とめっくす, むとう, .Hiro K Matthews, わりこま

変な JavaScript のバグ発表ドラゴンが変な JavaScript のバグを発表します。

文　編集部　Sosuke Suzuki

筆者は JavaScript が好きなので、JavaScript の中でも、Safari の JavaScript の処理系である JavaScriptCore に存在していた特有のバグを発表します。本記事で発表するバグはすべて修正されています。

0.1　前提

JavaScript は仕様と実装が明確に分離されたプログラミング言語です。JavaScript の仕様は、ECMA International の技術委員会の一つである TC39 によって策定されている ECMA262 です。それらの仕様に従う形で、Chrome や Safari、Firefox の開発者が自分たちの JavaScript エンジンを開発します。

明確な仕様が存在しているので、実装がそれに従っていない場合はバグとして扱われます。本記事で発表するバグはすべてそういう種類のバグです。

0.2　Object.groupBy と Map.groupBy の第一引数に String を渡すと TypeError が throw されるやつ

ES2024 から標準化された `Object.groupBy`と `Map.groupBy`は第一引数にイテラブルを受けとり、第二引数に受け取った関数を使ってグルーピングする関数です。

```
const result = Object.groupBy([1, 2, 3, 4], (element) =>
  element % 2 === 0 ? "偶数" : "奇数"
);
console.log(result); // { "偶数": [2, 4], "奇数": [1, 3] }
```

この例では `Object.grpupBy`を使っています。`Map.groupBy`も引数の形は同じですが返り値が Map になります。

仕様によると[*1]、この `Object.groupBy`と `Map.groupBy`は、第一引数に任意のイテラブルを受け取ることができます。しかし JavaScriptCore の実装では、第一引数がオブジェクト型ではないとわかった時点で TypeError を throw するようになっていました。

つまり、オブジェクト型ではないかつイテラブルである型の値が、不正に TypeError として扱われてしまうことになります。この「オブジェクト型ではないかつイテラブルである型」には String が該当します。

[*1] https://tc39.es/ecma262/#sec-groupby

以下のコードは JavaScriptCore では TypeError になっていました。

```
const result = Object.groupBy("abcd", (element) => element < "c" ? "前半" : "後半");
console.log(result); // { "前半": ["a", "b"], "奇数": ["c", "d"] }
```

これでは困るので修正しました[*2]。

0.3 RegExp.prototype[@@split] を呼び出しても RegExp.$+ らが更新されないやつ

JavaScript には標準化されていないがほぼすべてのブラウザに実装されている機能がいくつか存在します。その代表的なものが RegExp Legacy Features[*3] です。

これはグローバルの RegExpというオブジェクト上に存在するいくつかのプロパティにまつわる機能の総称で、たとえば RegExp.$+や RegExp.$&などがあります。これらのプロパティは、RegExp オブジェクトに対して特定の操作をしたときに更新されます。

ご覧の通り、普通に意味がわからないし標準化されていないので、使用は推奨されません。

```
console.log(RegExp["$&"]); // $1
/foo/.exec("foo");
console.log(RegExp["$&"]); // "foo"
```

RegExp.prototype[@@split]を呼び出したときにも、この RegExp Legacy Features のプロパティたちは更新されるべきなのですが、更新されないというバグがありました。

これはあんまり困りませんが、バグではあるので修正しました[*4]。

ちなみに、この RegExp Legacy Features という機能は、ブラウザに実装されているのに標準化されていないのはおかしいということで、標準化が進んでいます[*5]。

0.4 RegExp.prototype[@@matchAll] が正規表現の v フラグを考慮しないやつ

ES2023 から正規表現に新しく v フラグというフラグが導入されました。これは u フラグのスーパーセットとなるフラグで、このフラグを有効にすると、u フラグの全ての機能に加えて集合記法と文字列に対する Unicode プロパティが使えるようになります。

[*2] https://commits.webkit.org/276736@main
[*3] https://github.com/tc39/proposal-regexp-legacy-features
[*4] https://commits.webkit.org/277559@main
[*5] https://github.com/tc39/proposal-regexp-legacy-features

v フラグは u フラグのスーパーセットなので、少なくとも u フラグができることはできないといけません。しかし RegExp.prototype[@@matchAll] という関数は、v フラグをつけた RegExp オブジェクトに適用しても、フラグがない状態として扱うようになっていました。

```
function doMatchAll(regex) {
  // 吉 is a surrogate pair
  const text = "吉a吉";
  const matches = [...RegExp.prototype[Symbol.matchAll].call(
      regex, text)];
  return matches.map((match) => match.index);
}

// uもvもなし
console.log(doMatchAll(/(?:)/g)); // [0, 1, 2, 3, 4, 5]
// uあり
console.log(doMatchAll(/(?:)/gu)); // [0, 2, 3, 5]
// vあり
console.log(doMatchAll(/(?:)/gv)); // [0, 2, 3, 5] になるべきなのに [0, 1, 2, 3, 4, 5] になる
```

これはたまに困る人がいそうなので、修正しました[*6]。

0.5 論理代入式の左辺が関数呼び出しのときに SyntaxError ではなく ReferenceError が起こるやつ

ES2021 から論理代入演算子という演算子が追加されました。

```
foo ||= 3;
bar &&= 4;
baz ??= 5;
```

この演算子の左辺がもし関数呼び出しだったとき、仕様によると[*7]SyntaxError として扱う必要があります。しかし、JavaScriptCore では ReferenceError として扱われていました。

```
// SyntaxErrorであるべきなのにReferenceErrorになってしまっていた
foo() ||= 3;
```

[*6] https://commits.webkit.org/277160@main

[*7] https://tc39.es/ecma262/#sec-assignment-operators-static-semantics-early-errors

「たかがエラーの種類かよ」と思われるかもしれませんがこの２つには大きな違いがあります。ReferenceError は try-catch で握りつぶせるのに対して、SyntaxError を握り潰すことはできません。以下のコードは、もし `foo() ||= 3` が ReferenceError を throw するのなら、全体としては成功します。

```
try {
  foo() ||= 3;
} catch {}
```

代入演算子の左辺に関数呼び出しを書く人間はおそらくいないため、困ることはないと思いますがバグなので修正しました[*8]。

ちなみに、通常の代入演算子の左辺が関数呼び出しだった場合は、ReferenceError を throw するべきだと仕様によって定められています。理想的には SyntaxError であるべきなのですが、歴史的経緯でそのようになっており、JavaScript は後方互換性を重視するプログラミング言語なので変えられずにいるようです。

一方で、2021 年に新しく追加された論理代入演算子は、そういった後方互換性を気にする必要がないので、左辺に関数呼び出しが来た場合には SyntaxError にするものとして標準化されました。

0.6 RegExp.prototype[@@split] を呼び出したときに、this の hasIndices と dotAll がゲッターであっても、その関数が呼び出されないやつ

JavaScript のプロパティアクセスでは、もしそのプロパティがゲッターだった場合、ゲッター関数が呼び出されることになります。

```
const foo = {
  get property() {
    console.log("hello");
    return "property";
  },
};

console.log(foo.property); // "hello" のあとに "property" と表示される
```

`Object.defineProperty`を使うと、すでに存在するオブジェクトの特定のプロパティをゲッターとして上書きすることができます。次の例では、RegExp にもとから存在している `hasIndices`と `dotAll`というプロパティをゲッターとして上書きしています。

[*8] https://commits.webkit.org/277536@main

```
const re = /abc/;

let hasIndicesCount = 0;
Object.defineProperty(re, "hasIndices", {
  get() {
    hasIndicesCount++;
  },
});

let dotAllCount = 0;
Object.defineProperty(re, "dotAll", {
  get() {
    dotAllCount++;
  },
});
```

そして、この状態で RegExp.prototype[@@split]などの一部の組み込み関数を呼び出した場合、内部的に hasIndicesと dotAllへのプロパティアクセスが発生するという仕様になっています[*9]（実際にはこれ以外にもいくつかのプロパティにアクセスする）。

しかし、JavaScriptCore では hasIndicesと dotAllをゲッターフィールドにした状態で RegExp.prototype[@@split]を呼び出しても、それらのゲッター関数が呼び出されないというバグがありました。

```
re[Symbol.split]("abc");

console.log(hasIndicesCount); // 1 であるべきなのに 0 のまま
console.log(dotAllCount); // 1 であるべきなのに 0 のまま
```

これは別に困らないと思いますがバグなので修正しました[*10]。

なぜこのようなバグがあったのかというと、最適化のためです。関連する RegExp のプロパティがもとのものから更新されていない場合に限って高速パスで実行するという実装になっていたのですが、そのチェックが hasIndices と dotAll のみ漏れていたためこのようになったようです。

0.7 おわりに

変な JavaScript のバグがまた出てきたその時は発表したい。

[*9] https://tc39.es/ecma262/multipage/text-processing.html#sec-regexp.prototype-@@split

[*10] https://commits.webkit.org/279905@main

大学のレポート、Typst で書いてみませんか

文　編集部　Ryoga, Yu

1　はじめに

みなさんは大学のレポート課題は何で書いていますか？ 多くの人は LaTeX や Microsoft Word を使っているのではないでしょうか。LaTeX は数式の表記や論文のフォーマットに優れていますが、慣れるまでに時間がかかり、何より環境構築が大変で、特に初心者にとってはハードルが高いかもしれません。一方、Microsoft Word は使いやすい反面、複雑な数式や専門的なレイアウトには限界があります。そこで今回は、新たな選択肢として Typst という組版エンジンを紹介します。直感的な書き方と強力な機能を兼ね備えた文書作成ツールで、LaTeX の代替としても十分に活用できます。

2　Typst の特徴

まず、非常に新しい組版エンジンであるというところが特徴として挙げられます。2019 年から開発がスタートしており、処理系は 2023 年 3 月にオープンソース化されました。また、Rust で開発がされています。文書作成には独自の Typst 言語（拡張子：`.typ`）を用いて記述するのですが、Markdown などの軽量マークアップ言語に近い構文でありながら、複雑な処理も記述でき、表現力に長けています。

筆者はここ最近のレポートを Typst で書いています。そして、実際に使ってみて感じた Typst の素晴らしい点をまとめると、

- 環境構築が非常に簡単
- コンパイルが爆速
- 公式ドキュメントが充実している
- シンプルな文法
- エラーメッセージが分かりやすい
- 強力なスクリプト機能
- 強力なプラグイン

などが挙げられます。それぞれ順に詳しく述べます。

2.1　環境構築が非常に簡単

Homebrew が入っている環境なら `brew install typst` だけで入ります。Windows の環境なら `winget install --id Typst.Typst` を打つだけでインストールが完了します。LaTeX の環境構築に比べ圧倒的に簡単ですし、ディスク容量も大きくありません。また、`file.typ` というファイルから PDF にコンパイルしたい場合は、`typst compile`

`file.typ` というコマンドを打つだけです。

さらに、Typst には Web アプリ (`https://typst.app`) も存在し、これ以上に手軽に使うことも可能です。LaTeX でいうところの Overleaf のようなものを公式が提供しており、インストール不要でブラウザから直接文書作成が行えます。この Web アプリはリアルタイムでのプレビュー機能を備えており、どこでもすぐに作業を開始できる点が非常に便利です。リアルタイムでのプレビュー機能は Overleaf のそれに比べて反映が爆速です。さらに Web アプリのすごいところは共同作業にも強力なサポートを提供していることです。リアルタイムでの共同編集が可能で、チームメンバーと同時に文書を作成・編集できるため、グループワークでのレポート作成にも最適です。コメント機能やバージョン管理も備えており、編集履歴を追跡しながら円滑に作業を進めることができます。

2.2 コンパイルが爆速

これは筆者が最も感動した点です。LaTeX に比べ圧倒的に高速であり、ほぼリアルタイムでコンパイルが可能です。

いわゆるホットリロードも可能で、`typst watch file.typ` のようなコマンドを打ち込めばソースファイルを監視し、変更時に自動的に再コンパイルすることもできます。ちなみに、Typst には増分コンパイル機能があるため、毎回最初からコンパイルするよりも高速に動作するらしいです。

2.3 公式ドキュメントが充実している

公式ドキュメント (`https://typst.app/docs/`) にはチュートリアルから言語のリファレンスまで全てがまとまっています。

特に「Guide for LaTeX users」という記事を読めば LaTeX ユーザが Typst に移行するために必要なことが全てまとまっています。LaTeX に対応する Typst のコードが示されています。

問題点を挙げるとするならば現在（2024 年 7 月）はドキュメントが日本語化されておらず、英語を読む必要があるくらいです。

2.4 シンプルな文法

example.png という画像を中央に配置し、キャプションとして「example」をつけ、その後 2 乗和の公式を表示するコードで LaTeX と Typst の両者を比較してみます。

LaTeX

```latex
\documentclass{article}

\usepackage{graphicx}
\usepackage{amsmath}
\usepackage{amsfonts}

```

```
\begin{document}

\begin{figure}
\centering
\includegraphics[width=100mm]{example.png}
\caption{example}
\end{figure}

\[
\sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6}
\]

\begin{center}
二乗和の公式
\end{center}

\end{document}
```

Typst

```
#figure(
  image("example.png"),
  caption: "example"
)

$
sum_(i=0)^n i^2 = (n(n+1)(2n+1))/6
$
#align(center)[二乗和の公式]
```

このように Typst の方が圧倒的にシンプルにわかりやすく書けます。

Typst ではマークアップする方法として関数を使用します。また、特定の関数については専用の構文が存在し、ショートコードのようになっています。例えば特定の文字を太字にしたいとき、strong 関数により`#strong[example]` とかけますが、`*example*` のような専用の構文を用いても書けます。

なお、構文と対応する関数の網羅的な一覧は「Syntax – Typst Documentation」(`https://typst.app/docs/reference/syntax/`) に記されています。

構文を使って書くとかなりシンプルに記述できますが、対応する関数を使って書くことで冗長ではあるが柔軟な表現が可能です。

また、set 関数や show 関数によりデフォルトの挙動を変更することもでき、例えば #set list(indent: 0.5em) と書くことで、箇条書きのインデント幅を変更することができます。

数式まわりも特徴的です。Typst の数式記法は LaTeX のそれと互換性がないため、LaTeX ユーザにとっては慣れるまで時間がかかるかもしれませんが、圧倒的にシンプルに、バックスラッシュだらけにならない文法で記述することができます。この例には載せていませんが、数式中におけるカッコの大きさを自動的に出し分けてくれるため、\left(... \right) のように括弧の大きさを調節する必要がないです。

ところで、LaTeX を使って文書を作成しているとき、文法やその他のミスによってエラーが発生しますが、そのエラーメッセージがしばしば難解で、何が問題なのかがわからなかった経験はありませんか？ 一方、Typst のエラーメッセージは LaTeX よりは分かりやすいため、ストレスなく書けます。

2.5 強力なスクリプティング機能

Typst には強力なスクリプティング機能があり、柔軟な表現が可能です。Typst はチューリングもにっこりのチューリング完全であり、ほとんど何でもできます。

例として 1 から 100 までの和を計算することを考えてみましょう。実は、LaTex でも ifthen パッケージなどを使うことによって次のように書くことができます。

```
\usepackage{ifthen}

\newcounter{S}
\newcounter{K}
\newcommand{\my_sum}[2]{%
  \setcounter{S}{#1}
  \setcounter{K}{#1}
  \whiledo{\value{K} < #2}{%
    \stepcounter{K}
    \addtocounter{S}{\value{K}}
  }%
  \the\value{S}%
}

\begin{document}
  \[
    \sum_{i=1}^{100} i=\my_sum{1}{100}
  \]
\end{document}
```

バックスラッシュや文字がたくさんでみにくいですね。とても実用はできません。一方 Typst はこうです。

```
#let my_sum(n) = range(1,n+1).sum()
$
sum_(i=1)^(100) i = #my_sum(100)
$
```

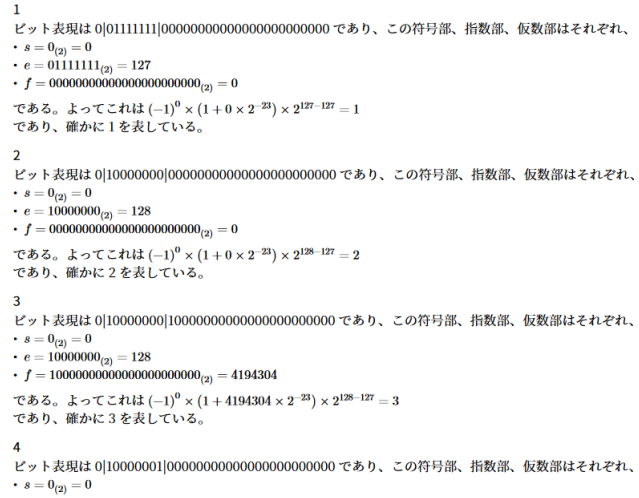
ここまでシンプルに記述できます。

でもこんなの使わないじゃん、と思うかもしれませんが、同じような文章を値を変えて何度も書きたい、といったことはよくあるでしょう。実際、筆者はコンピュータとプログラミングという授業のレポート課題で次のようなコードを書きました。浮動小数点数の内部表現について説明する課題です（一部表現を変えています）。

```
#let bin2int(bin) = {
  if bin.len() == 0 {
    return 0
  }
  return bin2int(bin.slice(0, count: bin.len() - 1)) * 2 + int(bin.last())
}

#let explain(n, repr) = {
  let splitted = repr.split("|")
  let (s, e, f) = splitted.map(v => bin2int(v))
  let result = calc.pow(-1, s) * (1 + f / calc.pow(2, 23)) * calc.pow(2, e - 127)
  assert(n == result)

  [=== #n]
  [ビット表現は #repr であり、この符号部、指数部、仮数部はそれぞれ、\
  - $s = #splitted.at(0) _((2)) = #s$ \
  - $e = #splitted.at(1) _((2)) = #e$ \
  - $f = #splitted.at(2) _((2)) = #f$ \
  である。よってこれは
  $(-1)^#s times (1+#f times 2^(-23)) times 2^(#e - 127) = #result$ \
  であり、確かに #n を表している。]
```


1
ビット表現は 0|01111111|00000000000000000000000 であり、この符号部、指数部、仮数部はそれぞれ、
- $s = 0_{(2)} = 0$
- $e = 01111111_{(2)} = 127$
- $f = 00000000000000000000000_{(2)} = 0$

である。よってこれは $(-1)^0 \times (1 + 0 \times 2^{-23}) \times 2^{127-127} = 1$
であり、確かに 1 を表している。

2
ビット表現は 0|10000000|00000000000000000000000 であり、この符号部、指数部、仮数部はそれぞれ、
- $s = 0_{(2)} = 0$
- $e = 10000000_{(2)} = 128$
- $f = 00000000000000000000000_{(2)} = 0$

である。よってこれは $(-1)^0 \times (1 + 0 \times 2^{-23}) \times 2^{128-127} = 2$
であり、確かに 2 を表している。

3
ビット表現は 0|10000000|10000000000000000000000 であり、この符号部、指数部、仮数部はそれぞれ、
- $s = 0_{(2)} = 0$
- $e = 10000000_{(2)} = 128$
- $f = 10000000000000000000000_{(2)} = 4194304$

である。よってこれは $(-1)^0 \times (1 + 4194304 \times 2^{-23}) \times 2^{128-127} = 3$
であり、確かに 3 を表している。

4
ビット表現は 0|10000001|00000000000000000000000 であり、この符号部、指数部、仮数部はそれぞれ、
- $s = 0_{(2)} = 0$

図 1　得られる出力

```
}

#explain(1, "0|01111111|00000000000000000000000")
#explain(2, "0|10000000|00000000000000000000000")
#explain(3, "0|10000000|10000000000000000000000")
#explain(4, "0|10000001|00000000000000000000000")
#explain(5, "0|10000001|01000000000000000000000")
#explain(6, "0|10000001|10000000000000000000000")
#explain(7, "0|10000001|11000000000000000000000")
```

これにより以下のような出力（[図 1]）を得ることができます。

これを LaTex で書こうと思うと、それぞれに面倒な計算と検算が必要だったり、表現を少し変えるたびに何か所も修正が必要だったりと手間が掛かると思います。実用的なコード量で処理を自動化したり、繰り返したりできる、それが Typst の魅力です。

更に、Typst ファイル内で `import` することによって、コンパイル時に自動で解決され、手動でインストールをする必要なくすぐに使うことができます。

例えば、codelst (`https://typst.app/universe/package/codelst`) というプラグインを使用することにより、コードブロックを綺麗に表示することができます。もっと不思議なものには、pyrunner (`https://typst.app/universe/package/pyrunner`) という Python のコードを実行することができるプラグインもあり、Typst Universe 上でいろいろと探してみるだけでも面白いです。

2.6 おわりに

Typst は最近出てきたということもあり、まだまだ発展途上なところが多いです。特に日本語まわりのレンダリングについては LaTeX の方が比較的綺麗ですし、日本語の文献も多いです。しかし、大学の授業レポートくらいならかなり実用的ですし、いろいろなことをシンプルなコードで実現することができます。何より爆速で、時間を節約することができます。ぜひ一度、Typst を試してみて、その便利さと使いやすさを実感してみてください。

Tauri + React で Markdown を描画する

文　編集部 （北野｜金子）尚樹（puripuri2100）

1　Tauri とは

Tauri[*1] とは GUI アプリのフレームワークです。UI 部分を Web フロントエンドで構築しメインプロセスを Rust で構築するという構成になっています。Web アプリのフロントエンドとバックエンドをひとまとめにしてローカル環境で動く実行ファイルにしてしまおうというコンセプトの下作られています。

GUI の表現に特化し、多くのリソースが投入されて発展してきた Web フロントエンドの技術をそのまま使えるという点において従来の GUI 記述フレームワークよりも学習コストが低く、資産の流用も容易です。また、メインプロセスに使用する Rust は現在とても勢いのある言語で多くのライブラリが存在し優秀な開発ツールが揃っています。

UI 部分とメインプロセスの間の通信は JSON 形式で行われる。Web フレームワークは JavaScript で動くためデータ構造をそのまま受け取ることができ、Rust でもデータ構造を JSON 形式に自動で変換するライブラリが存在するため苦労なくデータのやり取りができます。

2　Tauri で Markdown を描画したい

今回自分が作成していたアプリではメモ欄を機能として組み込んでいました。そのメモではテキストを単に p タグで表示していましたが、だんだんと箇条書きやリンク、数式の表示などをサポートしたくなってきました。

そこで考えたのが Markdown を入力してもらい、それを解析して描画するようにすることでした。CommonMark[*2] と呼ばれる Markdown の最小構成のような仕様ではリンクや強調表現や箇条書きなどのみの対応ですが、GitHub Flavored Markdown[*3] と呼ばれる、GitHub のみで使える拡張仕様では表やタスクリストなどもサポートしています。そのほかにも数式を埋め込める拡張などもあり、Markdown の方言を集めることでとても強い表現力を手に入れることができ、Markdown 記法の解析と描画を解決策として採用することとしました。

3　仕様

今回の解析と描画の目的は「便利なメモ機能」です。そのため、メモには過剰すぎる機能やインジェクションの恐れのある記法をサポートしないようにします。しかし、その一方

[*1] https://tauri.app/
[*2] https://commonmark.org/
[*3] https://github.github.com/gfm/

で必要な機能はサポートする必要があります。

今回の実装に当たって必ず実装する機能と絶対に実装しない機能は以下のようにしました。

- 必ず実装する機能
  - 箇条書き
  - 強調
  - 打消し
  - リンク
  - 数式
  - コード
  - 引用
- 絶対に実装しない機能
  - HTML タグ直入力
  - メタデータ入力
  - 脚注
  - WARNING や CAUTION などの装飾

4 採用技術

UI を担当する Web フレームワークには React[*4] を用いました。React は「HTML タグって attributes や children が引数で返り値が Element の関数とみなせるよね？」という関数型言語オタクの香りがしてとても好みだからです[*5]。

Markdown の解析は Rust の pulldown-cmark[*6] というライブラリを用いました。解析できる記法が多く、解析するかどうかも含めて柔軟に切り替えることができるためです。

5 解析の手順

入力された Markdown テキストを pulldown-cmark で解析し、得られたストリームデータから typescript と React で表現するためのデータ型に変換する関数を定義します。

5.1 文章構造に対応するデータ型の定義

Markdown で記述する文章の構造は大きく、縦方向に積まれる「ブロック」と横方向に伸びる「インライン」の 2 種類に分けられます。

「ブロック」を構成する要素としては以下のものがあります。

- 段落
- 横線

[*4] `https://react.dev/`

[*5] 「宣言的 UI」や「UI = f(state)」と表現されるらしい

[*6] `https://github.com/pulldown-cmark/pulldown-cmark`

- 節見出し
- 箇条書き
- 引用
- コード
- 表

「インライン」を構成する要素としては以下のものがあります。

- テキスト
- インラインコード
- 数式
- 強調
- 打ち消し
- 改行
- リンク
- 画像

このように列挙してみると納得感があるかと思います。これらの要素の構成要素を考えた上で Rust でのデータ構造として記述すると次のような再帰構造の列挙型になります。

```
#[derive(Debug, Clone, Serialize)]
#[serde(tag = "type", rename_all = "snake_case")]
pub enum MarkdonwInline {
    Text {
        string: String,
    },
    InlineCode {
        string: String,
    },
    InlineMath {
        string: String,
    },
    DisplayMath {
        string: String,
    },
    Emphasis {
        text: Vec<MarkdonwInline>,
    },
    Strong {
```

```
        text: Vec<MarkdonwInline>,
    },
    Strike {
        text: Vec<MarkdonwInline>,
    },
    Br,
    Link {
        link: String,
        text: Vec<MarkdonwInline>,
    },
    Image {
        link: String,
        string: String,
    },
}

#[derive(Debug, Clone, Serialize)]
#[serde(tag = "type", rename_all = "snake_case")]
pub enum MarkdonwBlock {
    Paragraph {
        text: Vec<MarkdonwInline>,
    },
    Rule,
    Heading {
        level: usize,
        text: Vec<MarkdonwInline>,
    },
    BlockCode {
        lang: Option<String>,
        code: String,
    },
    Quote{ quote_children: Vec<MarkdonwBlock>},
    Ol {
        start: u64,
        children: Vec<MarkdonwList>,
    },
    Ul {
```

```
        children: Vec<MarkdonwList>,
    },
}

#[derive(Debug, Clone, Serialize)]
#[serde(tag = "type", rename_all = "snake_case")]
pub enum MarkdonwList {
    Ol {
        start: u64,
        children: Vec<MarkdonwList>,
    },
    Ul {
        children: Vec<MarkdonwList>,
    },
    Item {
        check: Option<bool>,
        text: Vec<MarkdonwInline>,
    },
}
```

このとき列挙型を定義するときに記述している

```
#[derive(Debug, Clone, Serialize)]
#[serde(tag = "type", rename_all = "snake_case")]
```

という 2 行のおまじないの意味を説明します。

まず 1 行目ですが、これは定義したデータ構造に便利な機能を自動で実装するためのマクロです。左から順に次のような機能を自動で実装してくれます。

Debug　要素の中身を文字列化してプリントできるようにする。
Clone　deep copy ができるようになる。
Serialize　様々なフォーマットに変換するための統一的なインターフェースを実装する。

次に 2 行目です。これは前述の Serialize 機能の自動実装の設定です。”rename_all” は要素名を Rust の CamlCase から TypeScript でよく使われる snake_case に自動変換させる指定を行っています。”tag” は列挙型を JSON フォーマットに落とし込む際の要素の名前とデータの取り扱い方法の指定を行っています。特に指定がなければ通常は

```
{
  "text": {"string": ""},
```

```
  "inline_code": {"string": ""},
  ...
}
```

という風に変換されますが、これでは TypeScript 側で受け取るときに取り扱いが難しいため、要素名を別でくくりだすようにしています。これにより

```
{
  {"type": "text", "string": ""},
  {"type": "inline_code", "string": ""},
  ...
}
```

という風に変換されるようになり、”type” の値で処理を切り替えられるようになります。これに対応してこのデータ構造を受け取る TypeScript の型は次のようになります。実際に取り扱う際には”type” の値でどの要素に値が入っているかが確実にわかるため、型の恩恵をある程度得ながら処理を進めることができるようになります。

```
export type markdown_inline = {
  type: 'text' | 'inline_code' | 'inline_math' | 'display_math
      ' | 'emphasis' | 'strong' | 'strike' | 'br' | 'link' | '
      image',
  string: string | null,
  text: markdown_inline[] | null,
  link: string | null
}

export type markdown_block = {
  type: 'paragraph' | 'rule' | 'heading' | 'block_code' | '
      quote' | 'ol' | 'ul',
  text: markdown_inline[] | null,
  start: number | null,
  level: number | null,
  lang: string | null,
  code: string | null,
  children: markdown_list[] | null,
  quote_children: markdown_block[] | null,
}

```

```
export type markdown_list = {
  type: 'ol' | 'ul' | 'item',
  children: markdown_list[] | null,
  text: markdown_inline[] | null,
  check: boolean | null,
  start: number | null,
}
```

5.2 解析する関数の定義

pulldown-cmakr ライブラリで Markdown 文字列を解析すると、次のような定義の列挙型[*7][*8] がストリームデータとして得られます。

```
pub enum Event<'a> {
    Start(Tag<'a>),
    End(TagEnd),
    Text(CowStr<'a>),
    Code(CowStr<'a>),
    InlineMath(CowStr<'a>),
    DisplayMath(CowStr<'a>),
    Html(CowStr<'a>),
    InlineHtml(CowStr<'a>),
    FootnoteReference(CowStr<'a>),
    SoftBreak,
    HardBreak,
    Rule,
    TaskListMarker(bool),
}

pub enum Tag<'a> {
    Paragraph,
    Heading {
        level: HeadingLevel,
        id: Option<CowStr<'a>>,
        classes: Vec<CowStr<'a>>,
        attrs: Vec<(CowStr<'a>, Option<CowStr<'a>>)>,
```

[*7] https://docs.rs/pulldown-cmark/0.11.0/pulldown_cmark/enum.Event.html
[*8] https://docs.rs/pulldown-cmark/0.11.0/pulldown_cmark/enum.Tag.html

```
    },
    BlockQuote(Option<BlockQuoteKind>),
    CodeBlock(CodeBlockKind<'a>),
    HtmlBlock,
    List(Option<u64>),
    Item,
    FootnoteDefinition(CowStr<'a>),
    Table(Vec<Alignment>),
    TableHead,
    TableRow,
    TableCell,
    Emphasis,
    Strong,
    Strikethrough,
    Link {
        link_type: LinkType,
        dest_url: CowStr<'a>,
        title: CowStr<'a>,
        id: CowStr<'a>,
    },
    Image {
        link_type: LinkType,
        dest_url: CowStr<'a>,
        title: CowStr<'a>,
        id: CowStr<'a>,
    },
    MetadataBlock(MetadataBlockKind),
}

pub enum TagEnd {
  ...
}
```

例えば

```
# 最初の章

数式：$x^2 = y^2$
```

```
4 コード：`println!("Hi!")`
```

という Markdown テキストは次のような配列に変換されます。

```
1  [
2    Event::Start(Tag::Heading{level: 1}),
3    Event::Text("最初の章"),
4    Event::End(TagEnd::Heading{1}),
5    Event::Start(Tag::Paragraph),
6    Event::Text("数式："),
7    Event::InlineMath("x^2 = y^2"),
8    Event::HardBreak,
9    Event::Text("コード："),
10   Event::Code("println!("Hi!")"),
11   Event::End(TagEnd::Paragraph),
12 ]
```

このように開始と終了が明示的に与えられその間にコンテンツが挟まる配列構造を、前述したような再帰的な木構造に変換する関数を実装することになります。これは解析結果の配列から値を先頭から取り出し木構造を構築する再帰関数を複数定義して相互に呼びあうことで容易に実装することができます。

例えば横方向に伸びるインライン要素を解析し配列にする関数は次のように定義することができます。Text や InlineMath のような単純な値であればそれぞれに対応する列挙型の要素に紐づけて配列に追加します。強調のような入れ子になっている要素であれば parse_markdown_inline を再帰的に呼び出し、返り値を要素に紐づけて配列に追加します。このとき、終了条件となるタグを与えるようにすることで強調や打消しなどのいくつかの構造に汎用的に適用することができます。

```
1  fn parse_markdown_inline(
2    iter: &mut Parser<DefaultBrokenLinkCallback>,
3    end: TagEnd
4  ) -> Vec<MarkdonwInline> {
5    let mut v = Vec::new();
6    while let Some(event) = iter.next() {
7      match event {
8        Event::End(tag_end) => {
9          if tag_end == end {
10           break;
11         }
```

```
      }
      Event::HardBreak => v.push(MarkdonwInline::Br),
      Event::Text(text) => v.push(MarkdonwInline::Text {
        string: format!("{text}"),
      }),
      Event::Code(text) => v.push(MarkdonwInline::InlineCode {
        string: format!("{text}"),
      }),
      Event::InlineMath(text) => v.push(MarkdonwInline::
          InlineMath {
        string: format!("{text}"),
      }),
      Event::DisplayMath(math) => v.push(MarkdonwInline::
          DisplayMath {
        string: format!("{math}"),
      }),
      Event::InlineHtml(text) => v.push(MarkdonwInline::Text {
        string: format!("{text}"),
      }),
      Event::Html(text) => v.push(MarkdonwInline::Text {
        string: format!("{text}"),
      }),
      Event::Start(Tag::Emphasis) => {
        let inline = parse_inline(iter, TagEnd::Emphasis);
        v.push(MarkdonwInline::Emphasis { text: inline });
      }
      ...
      _ => ()
    }
  }
  v
}
```

同様にブロックの解析は次のように定義できます。

```
  fn parse_block(
    iter: &mut Parser<DefaultBrokenLinkCallback>,
    end: Option<TagEnd>,
```

```
) -> Vec<MarkdonwBlock> {
  let mut v = Vec::new();
  while let Some(event) = iter.next() {
    match event {
      Event::End(e) => {
        if Some(e) == end {
          break;
        }
      }
      Event::Start(Tag::Paragraph) => {
        let para = parse_inline(iter, TagEnd::Paragraph);
        v.push(MarkdonwBlock::Paragraph { text: para });
      }
      Event::Start(Tag::HtmlBlock) => {
        let para = parse_inline(iter, TagEnd::HtmlBlock);
        v.push(MarkdonwBlock::Paragraph { text: para });
      }
      Event::Start(Tag::BlockQuote(_)) => {
        let blocks = parse_block(iter, Some(TagEnd::BlockQuote
            ));
        v.push(MarkdonwBlock::Quote {
          quote_children: blocks,
        });
      }
      Event::Start(Tag::Heading { level, .. }) => {
        let text = parse_inline(iter, TagEnd::Heading(level));
        v.push(MarkdonwBlock::Heading {
          level: level as usize,
          text,
        });
      }
      ...
      _ => (),
    }
  }
  v
}
```

その他にも箇条書きなどを解析する関数を定義して相互に呼び出していきます。このように、相互に再帰する関数を定義して呼び出し合うことで木構造を構成することができます。

6 描画

解析で得られたデータを次は React で描画します。得られたデータは木構造で渡されるので、描画側も同様に再帰関数を定義し相互に呼びあうことで実装できます。

多くの要素は素の HTML で表現することができますが、コードブロックと数式については追加のライブラリを使用することが適切だと思います。今回、数式については react-katex ライブラリ[*9] を、コードブロックについては react-code-blocks ライブラリ[*10] をそれぞれ使いました。

6.1 実装

簡単に実装すると描画用関数は次のようになります。列挙型の名前を type 要素で保持しているため、そこで条件分岐を行って中身のデータを対応するタグにするだけになります。

```
1  function md_inline(md: markdown_inline) {
2    if (md.type == "text") {
3      return <>{md.string}</>;
4    } else if (md.type == "br") {
5      return <br />;
6    } else if (md.type == "inline_code") {
7      return <code>{md.string}</code>;
8    } else if (md.type == "inline_math") {
9      return <InlineMath>{md.string}</InlineMath>;
10   } else if (md.type == "display_math") {
11     return <BlockMath>{md.string}</BlockMath>;
12   } else if (md.type == "emphasis") {
13     return <em>{md.text?.map(md_inline)}</em>;
14   } else if (md.type == "strong") {
15     return <strong>{md.text?.map(md_inline)}</strong>;
16   } else if (md.type == "strike") {
17     return <s>{md.text?.map(md_inline)}</s>;
18   } else if (md.type == "link" && md.link) {
19     return <a href={md.link}>{md.text?.map(md_inline)}</a>;
20   } else if (md.type == "image" && md.link && md.string) {
21     return <img width="60%" alt={md.string} src={md.link} />;
```

[*9] https://github.com/talyssonoc/react-katex

[*10] https://github.com/rajinwonderland/react-code-blocks

```
  }
  return null;
}

function md_block(md: markdown_block): ReactElement {
  if (md.type == "paragraph") {
    return <p>{md.text?.map(md_inline)}</p>;
  } else if (md.type == "rule") {
    return <Line />;
  } else if (md.type == "heading" && md.level && md.text) {
    // TODO
    return <strong>{md.text.map(md_inline)}</strong>;
  } else if (md.type == "block_code" && md.code) {
    return (
      <CopyBlock
        text={md.code}
        language={md.lang ? md.lang : "text"}
        theme={github}
        showLineNumbers={false}
      />
    );
  } else if (md.type == "quote") {
    return <blockquote>{md.quote_children?.map(md_block)}</
       blockquote>;
  } else if (md.type == "ol" && md.start) {
    return <ol start={md.start}>{md.children?.map(md_list)}</
       ol>;
  } else if (md.type == "ul") {
    return <ul>{md.children?.map(md_list)}</ul>;
  }
  return null;
}
```

6.2 実行結果

Markdown を図 1 のように入力すると、図 2 のように描画されるようになります。数式や箇条書き、コードブロックのシンタックスハイライトも綺麗に描画されていることがわかります。

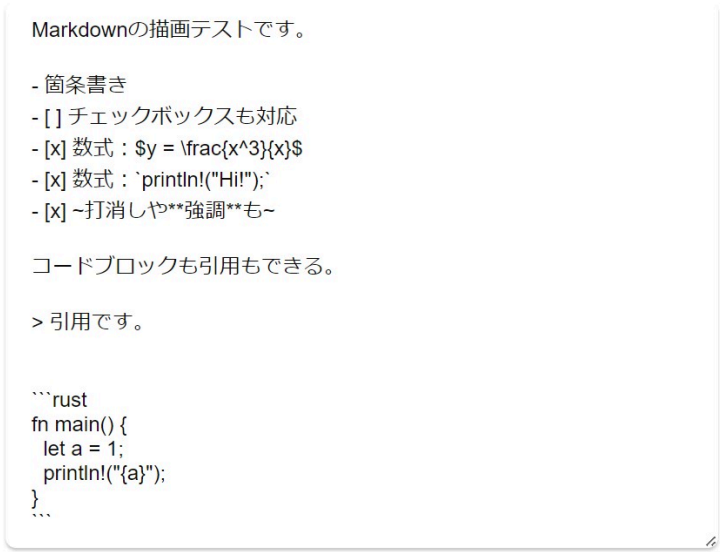


図 1　Markdown 形式の入力

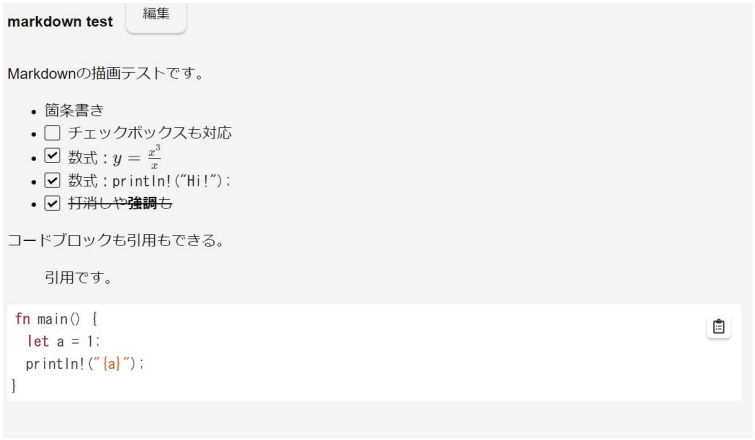


図 2　Markdown の表示

7　おわりに

Markdown は普段使いするときによく使うような要素を短い記法で表現することができて便利であり、皆さんも普段よく使っているかと思います。しかし、Markdown は構文が複雑でありサポートされている機能が多いため一から描画しようとするととても大変になります。

今回 Rust と React ライブラリの力を使うことで簡単に GUI アプリケーションに組み込めるようになりました。これで GUI アプリにおいてのユーザー体験が向上することが期待できます。

みなさんもこれを参考にぜひ Tauri でアプリケーション開発をしてみてください。

Linux デスクトップ元年であるところの 2024 年

文　編集部　間瀬 BB

はじめまして, **W O RD** 編集部の間瀬 BB と申します. 私は 2024 年度に AC 入試にて筑波大学に入学させていただいた者なのですが, **W O RD** を始め, 様々な楽しい空間やおもしろ人間の方々に囲まれ, まだ 4 ヶ月ほどですが日々大変楽しい大学生活を送らせていただいております.[*1]

さて, 話は変わりますが, 皆様はﾊﾟｼｮﾝｺｶﾀｶﾀ[*2] していらっしゃられますでしょうか? おそらく **W O RD** を読まれていらっしゃる方はその殆どが日常的にﾊﾟｼｮﾝｺに触れ合っていらっしゃると思われます.

そのような方に向け, 今回は筆者が大学入学数カ月後から 7 月ぐらいまでに試した 2 つの変な GNU/Linux[*3] ディストリビューションをメイン PC にデスクトップ環境としてインストールし, 数週間使ったレビューをお届けしたいと思います.

1　筆者の Linux 遍歴

筆者は前々より Raspberry Pi やヤフ〇クで買ってきたラックサーバなどでの OS は基本的に Ubuntu を用いてきておりました. その流れで 2021 年よりメイン PC として Ubuntu をヤフ〇クで買ってきた ThinkPad X260 に導入して長いこと使っていたりしました.

図 1　あれ...

ただ, なんと破壊が発生し少し衝撃を加えるとフリーズして死ぬ[*4] という現象により, X260 くんはその生涯を終えました.

[*1] できれば, 単位にも囲まれたいですね (?)(厳しそう)

[*2] パションコをカタカタする意

[*3] GNU/Linux と書かないと牛の大群に突撃されるらしいという風のうわさを聞いたのでこのような記述をしています. 以後は Linux とします

[*4] メモリのスロットの半田がイカれたっぽい: 接点復活剤をバカの量吹いてみたり, Reddit に同じ感じの症状の人が修理を依頼したらコレの修理方法はメモリを押さえるスポンジを追加すると書いてあるサービスマンマニュアルを修理員が見せてくれたとかいうスレがあったりして真似をして少し改善はしたが, 完全解決とは行かなかった.

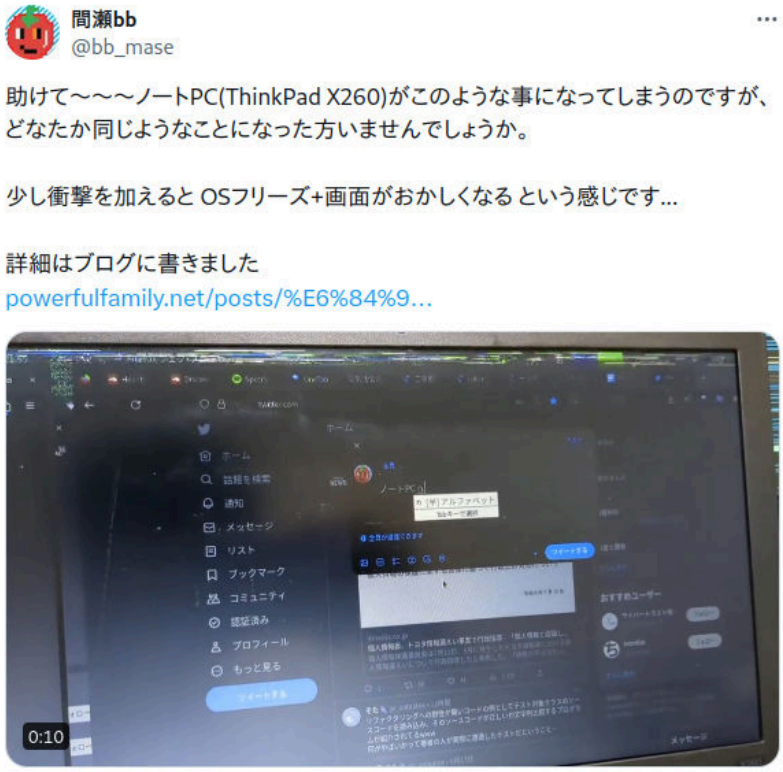


図 2　当時の助けを求めるツイート

そんなこんなでまた PC をヤフ〇クで仕入れてきたことによって, 前述のツイートもあり Windows + WSL2(Ubuntu) で慎ましくやっていたのですが, 結局開発は WSL2 の上の Neovim でやってたりするので Windows 側にあるファイルを開くのが面倒だったり, WSL2 は NAT の下にいるので外に公開したいときに謎の激長コマンドを打ち込みポートフォワード設定を入れないといけなかったり[*5], 筆者がネットワーク系を良く触るのもあり, 静的ルート追加して〜と `ip`コマンドのノリで追加したいなとなったときに謎の `ROUTE.exe`等々をしばかないといけなかったり[*6] などなど, 徐々に「Windows 君さあ〜」となる頻度が高くなり不満が溜まってきました.

また, **WORD** にいらっしゃる 23 生の方々は様々な Linux デスクトップ環境を使っていたりしていたりしており, いいな〜Linux, Windows 滅ぼすか〜の気持ちが更に高まっていきました.

さて, Linux にするにはするで今まで使ってきた Ubuntu とは別のものを使っていきたい

[*5] 結局使用するのが嫌すぎて一度も使用したことがない
[*6] なんか設定でネットワークモード的なのをいじればホストと同じ感じにできるとかも把握しており, 試してみたが私が期待した通りの動作をしなかった記憶があります

なというのと, Ubuntu デスクトップ版は GNOME が強制されるということで別のデスクトップ環境を使いたいなとも思いました. 最初は *I use Arch btw*[*7] になっちゃうか〜となっていたのですが, 悪い 24 生の Linux オタクにそそのかされて, 別の Linux ディストリビューションにしようとなりました.

2　実際にやってみた

ここでは, 入れ方をコマンド一つ一つ丁寧に書いても良いのですが, それらは過去の **WORD** 及びインターネットに非常に沢山の資料があるため, ここではインストールの感じや, 詰まった点, デスクトップ環境 (Sway) インストール等々をしてしばらく使ってみた感想を書き記し, 今後 Linux デスクトップ環境を導入したいと考える方, Arch Linux 以外の選択肢を試してみたい等々の方に向けてのものとします.

2.1　Gentoo Linux を入れてみた

読者の皆様は Gentoo Linux を使ったことありますか? 私はあります.

まず, Ubuntu などの Linux ディストリビューションとは対称的に, Gentoo Linux は非常にパッケージインストールの選択性が高いディストリビューションです.

それだけでは Arch Linux とほぼ同様かと思われるのですが, そもそもソフトウェアを手元の環境でビルドしたほうが最適化うんぬん/ソフトウェアを最新に保つ上でええやろ! という思想らしく, なんとソフトウェアのビルドからカスタマイズができるのです.

それは裏を返すと, 手元でマシンパワーを使ってビルドをしないといけないということでもあるのです.

インストール

Gentoo は Arch Linux 等と同じく Live USB を焼いても GUI もしくは TUI が立ち上がるなんて事はなく, そこにあるのはただのシェルになります. Live USB にあるバイナリを用いてディスクのフォーマットや必要パッケージのダウンロードを対象ディスクに対して手動で行っていくというスタイルです.

ただ, Gentoo 自体をインストールするのは基本的な Linux の事前知識があることもありそこそこ簡単でした.[*8] Gentoo Linux 公式 Wiki には Gentoo Handbook というのが存在し, それの日本語版も英語版と情報の大きな差なしで公開がなされています. これのインストー

[*7] 日本語意訳 : あー?オレの OS? ***Arch Linux*** だけど?(暗黒微笑)

[*8] え, あなた一発で出来たんですか?

ルマニュアルに従えば, 誰でもインストールが可能と考えます.

また, マニュアルの最初には以下の記述があります.

> 選択もまた Gentoo の設計原則のひとつです。Gentoo のインストール中、ハンドブック全体を通して選択は明らかにされます。
>
> https://wiki.gentoo.org/wiki/Handbook:AMD64/Installation/About/ja より

Handbook には様々な考えられるであろう選択のすべてが書いてあり, その選択全てについて細かな方法/それを選択すると何が良いのかが記述されています. これによりそれぞれのユーザーが自分の考えに合ったオレオレ Gentoo をインストールできるようになっています.

さて, Gentoo 自体はインストールできたのですが, ここから Wayland + Sway をインストールする必要があります. いままで Handbook のコマンドを右から左に写していっていたため, いきなり見放された感があります. が, こんなことでヤイヤイ言っていたらこの先 Gentoo 世界では生きていけません. なんとかググって…ということもなく, Gentoo の Wiki に Sway という記事が立っている (しかも日本語もあり) のでそれを参考にしつつ自分が入れたいアプリケーションを入れていきます.

Portage というパッケージマネージャも非常に強力で, USE フラグで自身の環境を宣言することによって (Wayland を使うぜ! PipeWire を使うぜ! など) 他のパッケージを導入する際にそれへのインテグレーションに必要なものをインストールしたりしてくれます. デスクトップ環境構築には非常にもってこいな機能と感じます.

もちろん, 宣言を追加したあとでも既存のパッケージにそれへのインテグレーション等が存在する場合は全体アップデートを回すことによってインストールされます.

> portage はソフトウェア管理における、Gentoo の最も特筆すべき技術革新のひとつです。その高い柔軟性と膨大な量の機能により、Linux で利用可能な最高のソフトウェア管理ツールであると見なされることもしばしばです。
>
> https://wiki.gentoo.org/wiki/Handbook:AMD64/Working/Portage/ja より

こんなことも書いてあります.

それはそうと, さらっと Gentoo をインストールするとかアプリケーションをインストールするとか言っているのですが, なんと手元でビルドする必要があります. もちろん, Linux カーネルもです. 無限の時間が全然余裕で溶けます.[*9]

もちろん, Binary Package という尊厳破壊を積極的に使用することも可能なのですが, 全てのパッケージにおいて対応しているわけでは全然ないのでビルドはどうしても走ります. しかもノート PC なので下手なデスクトップよりも時間が…

n 回の敗北後[*10], 優勝を目指して **WORD** 編集室で逐一なんのビルドが始まったかをブロードキャストしていると[*11] ここで先輩の一言

「君 Gentoo やめたほうがいいよ, 単位落とすよ w」

春 B 期末に近づくにつれこの言葉の現実味が帯びてきました.
ここで私は Gentoo を諦める, つまり今まで自分の思想のためだけにﾁｸﾅﾐ大学から供給された無限の電気を使いビルドした成果物をすべて破棄することを決定しました. 逆 SDGs に大貢献です.

2.2 OpenSUSE Linux を入れてみた

となり, 白羽の矢が立ったのが OpenSUSE Linux です.

はい, 使い慣れている Ubuntu ではありません. なぜかというと, *I use OpenSUSE btw*[*12] をやりたいから, というのは半分事実ですが半分冗談です. Debian とかいうやつを使えばデスクトップ環境を自由に選びつつ Ubuntu っぽく行けるのですが, ローリングリリース方式である Tumbleweed というのがあるのと, それによってレポジトリを追加しなくてもある程度最新のパッケージが落ちてくるという点に引かれました.

インストール

落としてきた ISO ファイルを焼き, ブートをすると! なんと, GUI が生えてきて, それに従うだけとなっております. ただ, インストーラの言語設定を日本語にすると豆腐まみれになり大エスパー大会が始まるためそこは玉に瑕ですね.

[*9] 手元のモバイル端末による Twitter 等々が非常に捗ります
[*10] ここまでに 2~3 週間かかった記憶
[*11] 「依存めちゃくちゃありすぎワロタ」→「あ゛!GCC のビルド始まった!!!」→「あ゛あ゛今度は LLVM かよ!!!!」→
[*12] 日本語意訳 : あー?オレの OS? ***OpenSUSE*** だけど?(暗黒微笑)

デスクトップ環境をその場でインストールすることも可能なのですが, Sway は対応していなそうなのでパス. CUI を立ち上げおもむろに `zypper install sway`をします.

そして, Sway を実行すると…なんだこれ!

なんかいっぱいパッケージ入るんだね とか思っていたら, Gentoo とは違いデスクトップ環境を構成するランチャーや音関係などなどが全部依存として入っていました.

まあ, Gentoo をやっていたので少しビビリましたが全部が入るのも悪くないですよね.

ただ, 依存ということは「でも, オレこのターミナルエミュレーター使いたくないんだよな〜(Gentoo で自分が好きなのを発見している)」となると…

```
zypper rm alacritty
```

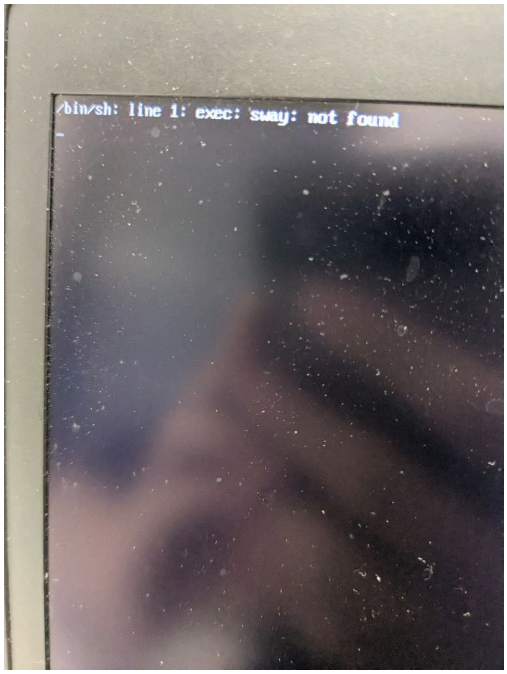


図 3

```
/bin/sh: line 1: exec: sway: not found
```

おい!(OS 再インストール)

といった感じで, Portage のカスタマイズ性, 最新性も欲しいな〜となりつつも, ビルドはもうこりごりだし〜という気持ちになり, 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…?となってしまいました.

結局の所, 適当に Sway や周辺ソフトウェア等々をインストールをし, それ使い初めて 1 週間ぐらいのタイミングで記事を書いています. とりあえずパソコンがないと色々支障があるのでチマチマとカスタマイズしつつ使っている感じとなっております. 今のところ先述のパッケージマネージャーの所以外は特に不満はない感じで安定しており意外と OpenSUSE 悪くないなといった感じです.

3　おわりに

逆張りって, 実はしないほうが良いらしい.

編集後記

北野尚樹（編集長）

皆さま暑い夏をいかがお過ごしでしょうか。WORD 編集部は今日も元気に冷房の効いた編集室でワイワイ作業をしています。

夏休みという期間はイベントが盛りだくさんであり編集部のメンバーも思い思いの活動を過ごしているようで、セキュリティキャンプに参加する人[*1] やインターンに行く人、趣味のソフトウェア作成に打ち込んでいる人、11 月に控えている雙峰祭の準備に余念がない人など、様々な楽しみがあることが部屋にいるだけでも伝わってきます。

こんな暑い夏こそ WORD 編集室に入り、涼しい部屋で多様なメンバーと楽しい大学生活を送りませんか？ まずは部屋の呼び鈴を押すところから。お待ちしております。

さて、私事ではありますが、この度結婚をし苗字を北野に変えることとなりました。旧姓の金子に変えて今号から北野として表記いたしますのでよろしくお願いいたします。また、これに伴う苦労は何としても次号でぶちまけたいと思っております。ご期待ください。

[*1] 受講者としてではなく運営側として参加する人がいるのも WORD 編集部のメンバーの凄さではないでしょうか

情報科学類誌

WORD

From College of Information Science

WORD編集長も大統領選挙戦から撤退します号

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7月23日　初版第1刷発行　（256部）