J. Plasma Fusion Res. Vol.96, No.7 (2020) 379-387

サロン

Markdownによる原稿執筆のすゝめ

An Encouragement of Writing with Markdown

伊藤篤史
ITO Atsushi M.
核融合科学研究所 ヘリカル研究部 基礎物理シミュレーション研究系

（原稿受付：2020年4月18日）

1．はじめに

科学論文の執筆や普段の研究ノート，会議の議事録，レポートの作成など，原稿を書く際にどのようなファイルフォーマットを用いておられるだろうか．おそらくMicrosoft社のOffice Word（以下，MS Word）やLaTeXという名前が挙がるだろう．中には，クラウドと連携した各種のノートアプリを研究ノートに利用している方もおられるだろう．

ここでは，最近普及してきたMarkdown形式による原稿執筆を勧めたい．あわせて筆者の開発したMarkdownエディタ"NowType"に関しても紹介したい．

2．これまでの原稿執筆：MS WordとLaTeX

あらためて，原稿執筆においては文章を文字列データとして入力することがコンピュータ時代の必須作業であり，なんらかのテキストエディタを用いることになる．しかし，テキストを入力するだけでは読む側にとって可読性の高い状態にはならず，実用上は見た目を整えるレイアウト調整が必要である．文章中の特定の部分のフォントの種類，サイズ，色を変えたり，太字や斜体にしたりするだろう．これらをテキスト入力と同時に行えるアプリケーションがいわゆるリッチテキストエディタである．前述のMS Wordはおそらく最も良くできたリッチテキストエディタであり，フォントの変更に加えて，図やテーブルを任意の位置に挿入したり，余白等のマージンを整えたり，原稿のレイアウト調整としておおよそ想定される全てのことができる．なによりも，印刷レイアウトそのままの状態で文章を入力して原稿の執筆を行える，いわゆるWhat You See Is What You Get（WYSIWYG）としての完成度が最大の魅力だろう．

それでも科学論文の執筆においては，特に数式の入力にはあまり向いていないという印象がある．以前に比べれば最近は幾分ましになったが，やはり複雑な数式になるとレイアウト調整が十分に行えない．また，数式番号の付与とその文中引用にも課題が残る．また，ファイルフォーマットはバイナリ圧縮されており，開くにはMS Wordか互換アプリケーションが必要である（注釈：ファイルフォーマットの仕様は公開されているが，その複雑な仕様ゆえに互換アプリケーションの開発は現実的に大変難しいだろう）．

数式の入力を第一の優先事項にすると，LaTeXに一日の長がある．数式の表現とレイアウトに関してはほぼ満足できる．入力する数式構文に関してもLaTeX形式が科学の世界では標準的なものとして普及しており，一部はMS Wordの数式入力にも取り入れられている．数式番号の付与や，その文中引用も非常に簡単である．数式の多い理論研究の原稿を書く時にはLaTeXが良く使われているだろう．

LaTeXのファイルフォーマットはシンプルなテキスト形式である．よって，レイアウトに関する指定は，専用のコマンド文字列を文章と合わせて記述することになる．豊富なコマンドが存在するので様々なレイアウト調整が可能である．しかしながら，実際に理想のレイアウトを実現するには職人的な技術が必要と感じる．多くの人は何かしらのテンプレートを使用して文章を入力し，多少のカスタマイズをしてお茶を濁していることだろう．また，本文となる原稿とレイアウト用のコマンド文字列が入り混じってしまい，執筆時の可読性が落ちてしまう．最も厄介な点は，数式や文章レイアウトの見た目の確認をするためにテキスト形式のファイルからPDF形式等へコンパイルする作業が必要となることだ．コンパイルには数秒から場合によっては数分を要し，本来の原稿執筆の集中が途切れる．複雑な数式等でコンパイルエラーとなった際には一苦労だ．これはWYSIWYGなMS Word等と比べて大きな欠点である．これが理由で，個人的には数式が少ない場合にはLaTeXを使うことは少なくなった．

ところで，シンプルなテキスト形式の利点は大きい．LaTeXのコンパイル環境がインストールされていなくても（昨今のLaTeX環境のインストールには膨大な時間が掛かる），PCであれば必ず入っているテキストエディタがあれば最低限の中身の確認と編集ができる．さらに，文章の差分が容易に取得でき，例えばLinuxのdiffコマンドな

National Institute for Fusion Science, Toki, GIFU 509-5292, Japan
author's e-mail: ito.atsushi@nifs.ac.jp

どで変更箇所が簡単に確認できる．最近であればテキスト形式であることは，Git 等のバージョン管理ソフトを用いることで変更履歴の管理が容易になるという利点にもつながる．

3．Markdown の利点

ここまで取り上げた MS Word や LaTeX に対して，Markdown の特徴を見ていこう．Markdown が採用しているファイルフォーマットは LaTeX と同じく，テキスト形式である．よって，LaTeX の項で挙げたテキスト形式の利点はそのままである．

Markdown の場合，本文となるテキスト以外にレイアウトに関する記述は LaTeX ほど豊富ではない．フォントの種類や色，図の配置場所など，“見た目”に関して指示を行う方法は一切ない．段落，見出し，箇条書き（リスト），表（テーブル）といった文章の“構造”を指示するための記法だけが存在する．構造の指示のための構文は非常に簡単であり，本文中に多種多様なコマンドが混ざらないので本文を視認しやすい．このことから Markdown は軽量マークアップ言語と呼ばれる．これに対して LaTeX は重量級のマークアップ言語といえるだろう．

“見た目”に関する記述ができないことは，それ単体では印刷物を成型（変換・コンパイル）できないことを意味する．しかし言い換えれば，本文テキストの内容と見出しや段落などの構造さえ変えなければ，どのようなフォントでどのような配置で表示してもよいということである．ゴシック体が好きな人もいれば明朝体が好きな人もいるだろう．背景の色はライトかダークか，見出しには数字をつけるか否か，お洒落なデザインかシックなデザインか，好みは人それぞれあるだろう．Markdown のプレビューでは，見た目に関する情報は別途“スタイル”や“テーマ”として用意しておいて，ユーザーごとに好みのスタイルで表示することができる．

LaTeX であればこれらの見た目に関する項目もテキスト中に記述してしまうので，誰がコンパイルしても同じ見た目になる．MS Word も同様の見た目に関する情報をファイルの中に埋め込んでいるので，誰がファイルを開いても同じ表示となる．それも一つの利点だが，むしろ Markdown の様に“見た目”に関する構文がないことは，ユーザーの好みの多様性を主眼に置くと利点となる．

原稿を提出される側にとっても，本文と構造に関するデータだけを受け取って，フォントや配置に関しては出版物等に合わせて適切なスタイルを適用すればよい．MS Word や LaTeX で指定と異なるスタイルのファイルが提出されて，手作業で治すのは手間がかかるだろう．

このように，文章の構造と見た目を切り離して管理する方法は，Web ページを記述するための Hyper Text Markup Language（HTML）と Cascading Style Sheets（CSS）が代表的なものだ．HTML も CSS もどちらもテキスト形式のファイルである．本文のテキストと文章の構造は HTML ファイル内にタグを使って記述し，フォントなどの見た目に関する指示は CSS ファイルに記述する．そして CSS ファイルを差し替えれば，いとも簡単に見た目が変わる．例えば，CSS を変えることでデザイン部分はユーザーの好みや時代を反映したモダンなものに定期的に更新しつつも，本文の内容は殆ど手を付けずに文章コンテンツを長生きさせることができる．

しかし，本文と文章構造を記述する側の HTML においてタグを用いた記述というのは文章の構造を把握しやすい反面，手作業での記述が容易とは言えない．またタグの種類も規格が更新されるごとに増え続けるので覚えきれない．原稿執筆のフォーマットとして HTML は向いているとは言えない．

そこで，より原稿執筆に向いた HTML に変わる形式として John Gruber[1]によって提案されたものが Markdown 形式である．HTML に比べると文章構造の種類はより限定されていてできることも限られてくる．しかし，その覚えることの少なさが原稿執筆にはちょうど良い．

また，軽量マークアップ言語ゆえにスタイルを適用した“印刷状態”への変換が非常に速い．LaTeX のコンパイルの様に長時間待つ必要はなく，ほぼリアルタイムで変換できる．ほとんどの場合は John Gruber の提案通り HTML に変換され，その上で CSS ファイルによるスタイル指定を反映して画面にレンダリングされることになる．このリアルタイムな変換の特徴を生かして，Markdown エディターの多くはウィンドウを左右に分割し，左側のテキストエディターで Markdown を編集し，右側のプレビュー用ウィンドウに CSS が適用された印刷状態の文章を表示する．左側で編集するとリアルタイムで右側画面の印刷状態に反映される．この様なことができるのは，最近の Web ブラウザ技術の進化によるところも大きい．実際にエディタやビューアを開発するプログラマにとっては，Markdown から HTML への変換さえ行えれば，HTML の解釈と CSS の適用はブラウザエンジン側で行いレンダリングしてくれるので楽なものだ．

とはいえ，後述する NowType の開発では WYSIWYG を実現するために大変苦労したことは言及しておく．

4．具体的な Markdown の記述例

それでは，Markdown の記法を簡単に紹介しよう．まず，ファイルフォーマットはテキスト形式だ．拡張子は `.md` である．`.md` ファイルを適当なテキストエディタや Markdown エディタで開き，文章をテキストで入力すればそれが本文となる．この時，決められたマークアップ方法によって文章の構造を指定する．以下にその例を示すが，同時に対応する HTML タグ要素も挙げるので理解の助けにしてほしい．

また，筆者の開発した Markdown エディタである NowType[2]はフリーソフトウェアとして公開している．インストールして使用することもできるが，Web ページ[2]に行けばブラウザ上で気軽に試すことができるようにもしてあるので是非触ってみてほしい．

4.1 段落

一般的に Markdown では文章の纏まりは段落となり，連

続しない改行は無視される．連続した改行すなわち空行があると次の段落との区切りとなる．対応するHTMLタグは P である．

4.2 見出し

行の最初の文字列がシャープ # と半角スペース _ で始まる時，続く文字列は段落ではなく見出しとみなされる．# の数によって，見出しのレベルを 6 段階に設定できる．次の例では見出しと段落の記述を示している．

```
# 見出し 1（Title）

ここが段落の文章になる改行しても無視される
空行がある為この文章は第二段落となる

## 見出し 2（Chapter）

### 見出し 3（Section）

#### 見出し 4（Subsection）

##### 見出し 5

###### 見出し 6

段落の文章
```

ここでシャープの数による 6 段階の見出しは，ちょうどHTML の H1 タグから H6 タグに相当する．

4.3 リスト

Markdown では，二種類の箇条書き（リスト）として番号無しリストと番号付きリストが使える．

番号無しリストの記述は次の様にする．行の最初の文字がマイナス記号 - と半角スペース _ で始まると，続く文章をリストの要素とみなす．同様の行を連続して記述することでひと纏まりのリストとみなされる．また，リストの子要素を追加する場合にはマイナス記号 - の前にインデントとなる半角スペースを 2 つ以上入力すると，前の行のリストの子リスト要素になる．

番号無しリストの例を示す．

```
- リスト A
- リスト B
  - 子リスト B-a
    - 孫リスト B-a-a
- リスト C
```

これは次のように表示される．

- リスト A
- リスト B
  - 子リスト B-a
    * 孫リスト B-a-a
- リスト C

ちょうどHTML では UL タグと LI タグに相当する．

ただし，Markdown の表記においてはリスト要素の次の行に空行が無い場合に，続く文章の解釈が異なる．筆者の開発している NowType で読み込む場合は，先頭のインデントとなる半角スペースが 2 つ以上存在する場合はリスト内の要素とみなし，存在しない場合は新しい段落とみなす．つまり，次の Markdown テキスト

```
- リストの文章 1 です．
  行の頭に半角スペースが二つあるので，リストの
  内部要素です．
- リストの文章 2 です．
行の頭に半角スペースが無いので新しい段落です．
```

に対して，NowType では次の様に表示される．

- リストの文章 1 です．行の頭に半角スペースが二つあるので，リストの内部要素です．
- リストの文章 2 です．

行の頭に半角スペースが無いので新しい段落です．

この辺りは，Markdown のエディタ・ビューア毎に解釈が異なり，いわゆる Markdown の方言といわれる曖昧な部分である．

番号付きリストの記述は次のようにする．行の最初の文字が 数値，ピリオド . ，半角スペース _ の順で始まった場合に，続く文章をリストの要素とみなす．その行を連続して記述することでひと纏まりのリストとみなされる．また，二行目以降のリスト要素においては，数値 の前に 2 つ以上の半角スペースをインデントとして入力すると，前の行のリストの子リスト要素になる．

番号付きリストの例を示す．

```
1. リスト A
1. リスト B
    1. 子リスト B-a
      1. 孫リスト B-a-a
1. リスト C
```

ここで数値に意味はなく，連番である必要はない．逆に何かしら意味のある数値を書いても表示には反映されない．表示の際には自動で連番になり，次のように表示される．

1. リスト A
2. リスト B
   (a) 子リスト B-a
       i. 孫リスト B-a-a
3. リスト C

ちょうどHTMLでは `OL` タグと `LI` タグに相当する．

4.4 強調

Markdownでは，文字列をアスタリスク `*` で囲むと，囲んだ部分が強調表示される．アスタリスクの数により強調度合いは三段階に分かれる．通常それらは斜体，太字，斜体+太字で表示される．例えば，`*italic*`，`**bold**`，`***italic+bold***`と記入すると，*italic*，**bold**，***italic+bold*** と表示される．これら三段階の強調は，ちょうどHTMLの `EM`，`STRONG`，両方のネストに対応している．ただし実際の表示はCSSの指定によって変えられるので，エディタ・ビューアによって異なる場合がある．

4.5 コードブロック

強調と同様に文字列をバッククォートで囲むとコードブロックになる．コードブロックではインライン表示とディスプレイ表示（独立行表示）が使える．文字列を1つずつのバッククォートで囲むと，`inline_code` のようにインライン表示になる．一方で文字列を3つずつのバッククォートで囲んだ場合は，

```
display style code block
function test(a, b){
 return a + b;
}
```

の様にディスプレイ表示になる．

これらのコードブロックは，ちょうどHTMLの `PRE` タグや `CODE` タグに相当する．

4.6 数式

当初のMarkdownには数式を記述する方法は存在しなかったようだが，現在では殆どのエディタ・ビューアで数式のための拡張記法として，LaTeX形式の数式記述がサポートされている．

文字列の協調やコードブロックと同じように，文字列をダラー `$` で囲むと数式ブロックとなる．数式ブロック内ではLaTeX表記が使える．数式ブロックでもコードブロックと同じようにインライン表示とディスプレイ表示（独立行表示）が使える．両端を1つずつのダラー `$` で囲んだ数式文字列は，$f(x)=x^2+ax+b$ のようにインライン数式になる．一方で，両端を2つずつのダラー `$$` で囲んだ数式文字列は，

$$f(x)=\int g(x,y)\,dy \tag{1}$$

の様にディスプレイ数式になる．

TeX表記の数式文字列をHTMLにレンダリングする部分は，MathJax[3]やKaTeX[4]といったライブラリが一般的に使われる．どちらのライブラリを利用しているかでサポートされているTeX表記の範囲には若干の違いがあるが，殆どの場合には問題がない．ただし，LaTeXのような数式番号の付与とその参照には独自の作り込みが必要であるので，全てのビューアで同様に表示させることは難しい．

4.7 テーブル

Markdownではテーブルも利用できる．テーブル表記においては，行の区切りを改行で表し，列の区切りをバーティカルバー `|` で表す．テーブルをMarkdownテキストで表現した例は次のようになる．

```
|項目1|項目2|項目3|
|---|---|---|
|りんご|100円|3個|
|みかん|50円|10個|
|いちご||売り切れ|
```

ここで，一行目は見出し（ヘッダー）になる．二行目は見出しとデータ行の区切り線であり．多くのビューアにおいて，ヘッダー行と二行目の区切り線の行の存在がテーブルと認識する判定基準になる．

また，二行目の区切り線は列ごとの左寄せ・中央寄せ・右寄せの指示にも使われる．バーティカルバーの間の半角文字は最低3つ必要で，`---` もしくは `:--` と記述すれば，その列は左寄せになる．`--:` と記述すれば右寄せ，`:-:` と記述すれば中央寄せになる．

三行目以降はデータ行である．上記の例の場合は次のように表示される．

項目1	項目2	項目3
りんご	100円	3個
みかん	50円	10個
いちご		売り切れ

ちょうどHTMLでは `TABLE` タグに相当する．

4.8 URLリンク

MarkdownではURLリンクを記述できる．次のようなテキスト文字列を入力すれば，

```
[表示する文字列](index.html)
```

“表示する文字列”が文章中にインライン表示されるとともに，リンクになる．

これはちょうどHTMLの `A` タグに相当する．

4.9 画像とキャプション

Markdownでは画像ファイルも張り込める．先のURLリンクの括弧 `[` の前にエクスクラメーションマーク `!` を追加すると画像へのリンクと認識される．また，URLリンク同様にインラインブロックになる．例として次のようなMarkdownテキストを考える．

```
画像！[ALT属性として登録する文字列](icon.min.png)も表示させられます
```

これは次のような文章としてとして表示される．

画像  も表示させられます

HTML では IMG タグに相当する．

ただし，文章ファイル自体の保存形式はテキスト形式であるので，画像データを埋め込めるわけではない．あくまで画像ファイルへのパスを記述し，レンダリング時に画像ファイルを読み込んで表示させることになる．よって画像ファイルは原稿の Markdown と同じディレクトリかサブディレクトリに格納しておくのが良い．

また，多くのビューアでは Markdown を HTML へ変換することで表示を行っている．そのため，サポートされる画像ファイルフォーマットとしては JPG，PNG，GIF，SVG などWebブラウザで表示できるものに限定される．論文等の印刷物で度々用いられるEPS形式等は表示できないことが多いので事前に変換しておくことが望ましい．ベクター画像形式である EPS の変換先としては，同じくベクター画像形式である SVG が良いだろう．

加えて，Markdownエディタ・ビューアにはWebブラウザ上で動作するものも存在する．Web ブラウザ上で動作する場合には，セキュリティーの観点からローカルファイル操作に制限があり，画像ファイルの表示は上手くいかない場合がある．

ところで，画像にキャプション（figure caption）を加えたいという要望もあるだろう．しかし，Markdown においてはキャプションの記述に関する標準的な記法は無い．よって様々な拡張記法が提案されており，ビューア毎に採用されているものが異なる．

例えば括弧 [] 内部の文字列をキャプションとして解釈するビューアがある．しかしその場合，キャプション中にインライン数式などを記述しようとすると，括弧 [] の内部に特殊な文字が並ぶことになり，IMG タグの ALT 属性として登録するのは相応しくないと考えられる．最悪のケースとして表示用の HTML が破壊される．

そこで筆者の開発した NowType では，次の条件を満たした時に画像をインライン表示ではなくディスプレイ表示と見なし，キャプションを追加することとした．

- 画像用の Markdown テキスト ![Alt](path) が段落の先頭にある場合，この画像はディスプレイ表示とする．
- ディスプレイ表示と見なした場合，画像に続く文章・要素はキャプションと見なす．キャプションの終端は空行とする．

例として，次の Markdown テキストを考える．

```
段落 1 の文章
（空行）
![図 1](icon256x256.png)Fig1． ここに書いた文章が caption とみなされる．
数式$x=y$や**強調表示**も記述できる．
（空行）
段落 2 の文章
```

これは，HTML としては FIGURE タグと FIGCAPTION タグを使って次のように変換される．

```
<p>段落 1 の文章</p>
<figure>
<img alt='Fig1' src='icon256x256.png'>
<figcaption>
図 1 ここに書いた文章が caption とみなされる．
数式<span (generated by KaTeX)>x=y</span>や
<strong>強調表示</strong>も記述できる．
</figcaption>
</figure>
<p>段落 2 の文章</p>
```

実際に表示される図とキャプションも**図 1** として掲載しておく．

4.10 引用（citation）と注釈（footnote）

Markdown では引用（citation）と注釈（footnote）に関しては，それらを統合したような機能がある．文章内で引用をする時には，括弧 [] とハット記号（サーカムフレックス） ^ を用いて

```
Markdown エディタ NowType は Ito によって作られた[^1]
```

のように記述する．

そして引用元を記述する時には，同様の記号にコロン : と半角スペース _ を加え，行の先頭に配置して次のように書く．

図 1 ここに書いた文章が caption とみなされる．数式 $x = y$ や**強調表示**も記述できる．

```
[^1]: Atsushi M. Ito, プラズマ・核融合学会誌,
vol. X, No. Y, 2020.
```

これらは，HTMLの A タグを利用して実装されることが多いが，実際にこれがどのように表示されるかはビューアに依存するので注意が必要である．

4.11 その他

おそらく原稿執筆に必要な「文章構造」に関しては，上記でだいたい事が足りるだろう．上記以外にもMarkdownで記述できることはあるが，エディタやビューア毎の独自拡張である可能性が高いので注意が必要である．

プログラマ界隈では，GitHubの投稿ページで採用されている"GitHub Flavored Markdown Spec"[5] が一般的に成りつつある．また，Markdown特有の方言を無くすために標準化をめざした"CommonMark"[6]という規格も提案されている．ただし，両者ともに学術論文の執筆となると少しできることが不足している．そのため，筆者の開発したNowTypeでは学術論文執筆に必要最低限の機能を揃えるため，一部独自の拡張構文を追加している．

5．Markdownから別のファイル形式への変換

さて，Markdownで執筆した原稿を他者に渡す際には，Markdownの普及が十分でないことと，表示の互換性が課題である．最も良い方法はPDFに変換して受け渡すことである．多くのエディタ・ビューアはPDF出力を備えている．

しかし，出版物の原稿の場合には，MS WordやLaTeXといった出版社指定のファイルフォーマットで提出する必要がある．このような場合には，Markdownで書かれたファイルを変換することになる．変換ツールとしては"Pandoc"[7]が良く使われる．Pandocで変換できるファイルフォーマットは非常に種類が豊富であるので，マニアックな要望にも対応できるかもしれない．

6．入力支援とNowType

ここまでに述べたように，Markdownはテキスト形式とシンプルな構文を利点としたファイルフォーマットである．上記の指示構文は理解しやすく実際に使ってみると慣れるのも早いだろう．一方で，やはりある程度の長さの文章を執筆する時にはプレーンなテキストエディタでの執筆作業は疲れるという人も多いのではないか．個人的にも，WYSIWYGなエディタで書いた原稿を確認しながら執筆を進めていく方がやりやすい．

数式に関しては，入力したLaTeX構文が意図したものになっているかどうか，できるだけ迅速に数式へ整形して確認したい．そこで，リアルタイムでMarkdownを整形しながら原稿執筆ができるWYSIWYGエディタを作ることがNowType[2]開発の主な目的である．数式のレンダリングにはKaTeX[4]を利用したが，数式番号の付与などで独自の改良を加えてある．

また，強調表示，コードブロック，箇条書き，テーブル，図，リンクの挿入など，上記で挙げた指示構文に関しての入力支援機能を持たせた．

出力機能として，PDF形式とLaTeX（LuaLaTeX）形式へ変換して出力する機能も持たせてある．前述のPandoc等の外部ツールを使わずに変換することで，NowType独自のMarkdown表現にも開発者の責任の範疇として対応させている．

後述のWeb技術をベースにしているので，Operating System（OS）に依存しないクロスプラットフォームなアプリケーションになっている．さらに，PC上のデスクトップアプリケーションとして動作するスタンドアローン版に加えて，Webブラウザ上で動作するWeb版がある．これらはNowTypeのWebサイトおよびMicrosoft Storeにて配布している．

7．Markdownエディタ開発

7.1 2020年現在のGUIアプリ開発

ここでは，Markdownエディタ NowTypeの開発を経て筆者が得た経験を共有すべく，2020年現在のアプリケーション（以下，アプリ）開発について触れておきたい．

筆者は普段の研究では計算機シミュレーションを行っている．シミュレーションの為のプログラムは何らかの数値を計算することが目的であり，見た目は必要ない．よって，入力を受けとって出力として数値や文字列を返すCommand Line Interface（CLI）アプリが中心となる．一方で，今回のエディタ開発のような場合には，OS上で表示されるウィンドウ，ボタン，メニュー，チェックボックスなど，操作する上で便利な“見た目”を持ったGraphical User Interface（GUI）アプリを作ることになる．CUIアプリとGUIアプリの開発はかなり作法が異なる．CUIアプリの開発経験が十分にある方でも，GUIアプリの開発には敷居の高さを感じてなかなか取り組めないこともあるだろう．その敷居を跨ぐきっかけとして，この文章が役立てば幸いである．

WindowsでのGUIアプリ開発を振り返ると，Windows 7以前まではOSの提供するWin32APIを呼ぶことでウィンドウやボタンを実装していた．言語はVisual C++，Visual Basic，Visual C#など色々なものがあったが，Win32APIを直接呼び出すか，もしくはMFCなどのラッパーライブラリを利用していた．その後，Windowsの進化と共に，.NET Frameworkや，Windows 10に合わせて作られたWinRTが登場したが，なかなかWin32API時代の資産から乗り換えることができないことが多いようだ．その精神的な障壁の一つは，これらがWindows OSのみで動くものであり，クロスプラットフォームが基本となりつつある昨今ではあえて乗り換えるメリットが少ないと感じることではないだろうか．

アプリのクロスプラットフォーム化の波は，まさしくスマートフォンやタブレットの普及でiOSやAndroidの利用者が増えたことによるものだが，それに合わせて台頭してきたものがブラウザで動くWebアプリケーション（以下，Webアプリ）である．今やPCデバイスやスマートフォン

はインターネットに繋がることが当たり前で，インターネットを閲覧するためのWebブラウザはほぼ必ず搭載されている．あわせて，ブラウザで表示するHTMLは標準規格が整備され，Webページは OSやブラウザに依らず，同じ見た目，同じ動作が保証されていることになっている（ただし，実際にWebアプリ開発を行うと，OS間の違いやブラウザ間の違い，標準規格の穴に頭を抱えることになる）．

よって，Webブラウザ上で動くアプリケーションを構築できれば，それは自然とクロスプラットフォームアプリになる．ブラウザ上での表示であるので，Webページと同様にボタンやラベルなどの「構造」はHTMLで，色やフォントなどの見た目はCSSで記述すればよい．ボタンを押したときの各種の動作はJavaScriptで記述する．また最近では，Webページとして構築されたアプリをラップして単体（スタンドアローン）のアプリとして構築する仕組みとして，Electron[8]やNW.js[9]といったツールが存在する．さらには，ブラウザのブックマークとキャッシュを利用してオフラインでもWebページを閲覧できるようにさせることで，より簡易的な疑似アプリとして動作させるProgressive Web Apps（PWA）といったものまで登場した．これらの登場により，Webページさえ作れば，それを各OS向けの単体アプリとして簡単にリリースでき，クロスプラットフォームで動かすことができる．さらに前者のElectronやNW.jsの場合には，Webブラウザ上で動作することに起因したローカルファイル操作に関するセキュリティー制限も，Node.js[10]との連携によってかなり緩和される．単体のアプリを動作させていながら，その実はバックエンドでNode.jsサーバーがローカルファイル操作等を行い，フロントエンド側のWebページと通信することで実現される．この辺りはセキュリティーの面も含めてElectronのライブラリに必要な機能が整備されているので心配はいらない．

7.2 HTMLの動的編集

それでは実際にWebアプリとしてリッチテキストエディタを作る方法について述べる．リッチテキストエディタではなくプレーンなテキストエディタであれば，HTMLのBODY要素の中にTEXTAREA要素を設けてやればよい．殆どのブラウザはカット・コピー・ペーストやUndo, Redoといった編集操作にも標準で対応しているのでプログラムも最小限で良い．ブラウザがサポートしていれば，入力した文章のスペルチェック機能すら搭載されている．

しかしリッチテキストエディタを作るとなると，この様に単純ではない．例えばある部分だけ太字にしたい場合，対象の文字列をSTRONGタグやSPANタグで囲み，フォントの変わるようなCSSを適用させるという操作が必要になる．しかし，TEXTAREA要素の子として別のHTML要素は持たせられない．よって，このような用途ではTEXTAREA要素は利用できない．

このようなリッチなテキストの編集をWebで実現する機能として，HTML5からはcontentEditable属性というものが追加された．通常キーボード入力で編集を受け付けるHTML要素はTEXTAREA要素かINPUT要素くらいであった．ところが，プロパティーとして `contentEditable="true"` を設定すると，デフォルトでは編集を受け付けないDIV要素等であってもキーボード入力を受け付けて文字列が編集可能な要素にすることができる．

Markdownエディタを WYSIWIGとして実現させるには，ユーザーの入力に合わせて，動的にHTML要素を生成していく必要がある．例えば，シャープ `#` と半角スペース `_` をユーザーが入力したことを検知したら，以後の文字列を“見出し”とみなしてHTMLのH1要素を動的に生成させる．このような動的なHTML要素の生成はJavaScriptを利用して行う．HTMLにおける要素は開始と終了のあるタグで挟むことで表現され，タグを入れ子にすれば要素の中に子要素を持たすことができる．このような要素の親子関係はDocument Object Model（DOM）として規定されており，最近のJavaScriptではDOMを直接操作するAPIが用意されている．DOM操作のAPIは殆どのモダンブラウザでサポートされており，それらを使ってHTML要素の生成や消滅を行えばよい．

ただし，DOMをJavaScriptで操作すると，ブラウザが標準で用意してくれているUndo操作のための編集履歴と実際のHTML要素の状態に齟齬が生じる．よって，Undo操作に関する全ての処理もJavaScriptで独自構築する必要がある．

7.3 クロスプラットフォームの実際

ここまでの内容を読んで，HTMLと多少のJavaScriptが扱えれば，Electronでラップしてクロスプラットフォームアプリが簡単に作れるという印象を持っていただけただろうか．興味を持った方は是非チャレンジしてほしい．

一方で，Markdownエディタ開発を通じて筆者が感じた実際上の難しさにも触れておきたい．

常に計算をし続けているシミュレーションのようなCUIアプリと違い，GUIアプリではユーザーの操作に端を発して何らかの処理を行う，いわゆる“イベント駆動型”の処理を行う．ユーザーのクリック操作やキー入力に合わせてイベントが発火するので，それをJavaScriptでフックして目的の処理を行うようにコードを記述する．この時，フックしなかった場合にブラウザがデフォルトで行う動作をというものもあるが，それをキャンセルしてやることもできる．例えば，TEXTAREAにキー入力を行うと，フックしなかった時はキーに合わせた文字が追加されるが，フックしてデフォルト動作をキャンセルしてやれば文字が追加されなくなる．合わせてDOM操作のAPIで何らかのHTML要素を追加するようにすれば，ユーザーのキー入力によってHTML要素が追加されたように振る舞う．

しかしここで問題となるのが，イベントの種類が多く，ブラウザによってサポートされているイベントが完全には統一されていないということである．さらに，イベントの発生順序もブラウザによって異なる．もっと言えば，同じブラウザであっても異なるOS上では動作も異なる．よって実際の開発では，想定される主要ブラウザおよび主要

OSの全てで動作確認を行う必要がある．場合によっては，ブラウザやOSごとに処理を書き分けてやる必要がある．印象として，クリック関連のイベントはさほどブラウザ間の差異がないが，キー入力関連には大きな差がある．

キー入力に関連して，半角英数以外の日本語のような多言語入力の制御も難しい．半角英数以外の入力では，いわゆるInput Method Editor（IME）による入力を行うが，このIMEの制御はブラウザというよりOS側で制御している部分が大きい．そのためか，上記で触れたようなイベント発火時にフックしてデフォルト処理による文字入力をキャンセルするという行為が，IME入力に関しては行えない．また，ブラウザやOSによってJavaScriptから取得できるIME関連イベントのプロパティーに違いがあり，ものによっては必要な情報が取得できない．主要ブラウザのエンジンが英語圏の開発者によって作られているので仕方のないことかもしれないが，IME入力に関しては，Webの標準規格としてもまだ成立していないようである．

例にもれず，Markdownエディタの開発でも日本語IME入力の制御に大変苦労した．最終的にIME入力に関しては，キー入力イベントをフックするのではなく，`MutationObserver`というDOMの変更を検知するAPIを利用することでなんとか制御するに至った．`MutationObserver`自体も新しいAPIということもあってブラウザのサポート状況が心配されたが，幸い現状の主要なモダンブラウザの全てで動作を確認することできた．

7.4 アプリの配布に関する課題

とはいえ，クロスプラットフォームアプリの開発は，一昔前に比べればかなり簡単になったという印象を受ける．一方で，開発したアプリの配布に関しては寧ろハードルが上がっているように感じる．これは最近のセキュリティー重視の観点から，配布されているアプリのインストールや実行に関して制限が掛けられるためである．インターネット上で配布されているアプリは，作者の意図に反して第三者による改ざんが行われている可能性があるので，安易に実行してはいけないとされている．そのため，最近のOSでは標準機能として「信頼されてないアプリ」の実行を制限する機能が搭載されている．Windows 10のSmartscreenやMac OSのGatekeeperがこれにあたる．また，アンチウィルスソフトによってはさらに厳しい警告が表示される．最悪の場合はアプリを配布してもユーザーのPCでは実行してもらうことができない．Webからのダウンロードだけでなく，メールやUSBメモリーを介して受け渡したアプリにも同様の制限が掛かる．

これらの検知をクリアして「信頼されたアプリ」にするためには，「コード署名」と呼ばれるものをアプリに同梱する必要がある．さらにそのコード署名が本物であることを証明する「コード署名証明書」を然るべき機関で発行しなければならない．OSやアンチウィルスソフトがアプリ付属のコード署名が本物かどうか確認を行うことで「信頼されたアプリ」として認められ，無事にインストールや実行ができるようになる．

このコード署名の問題はセキュリティーの観点からは理解できるが，「コード署名証明書」の発行には年間数万円の費用が掛かることが問題である．今回の様に学術研究目的で開発したアプリの配布や，個人が趣味で開発したアプリなど，フリーソフトを無料配布する場合にはこの発行費は悩みの種である．また，殆どの場合は法人としてしか発行を受け付けていないのも問題である．

この状況への一つの解として，Windows 10向けのアプリであればMicrosoft Storeで配布することを紹介しておく．開発したアプリをWebサイトから申請し，無事に認定されればMicrosoft Storeで配布することができる．この際，コード署名とその証明書をMicrosoft社が無料で発行してくれるということが何より重要な点である．アプリの申請権を得るためには開発者登録費として1800円程度必要となるが，初回の一度きりでよい．先のコード署名証明書の発行費が毎年掛かることに比べたら遥かに安い．さらに，開発者になればアプリはいくつでも申請することができる．今回開発したNowTypeもMicrosoft Storeで配布している．実際にやってみると申請から認定までの作業はそれほど難しくない．アプリの公開範囲は広く全ユーザー向けにしても良いし，指定したユーザーにだけ限定公開することもできるので，閉じた研究用途でも大丈夫だろう．

Mac OSの場合も，2020年2月以降はApple社の認定を受けたアプリ以外は実行不可能となった．詳細は割愛するが，こちらもApp Storeに申請して認定されればコード署名証明書をApple社が発行してくれるようである．ただし，開発者登録費として毎年11800円が必要だ．

また，将来的には先に上げたPWAによってコード署名の問題は解決するかもしれないが，この辺りはWebとPCを取り巻く各団体のせめぎ合いがありそうだ．

8．おわりに

本稿では，Markdown形式による執筆方法を紹介した．また，当初の予想以上に苦労の連続であったものの，何とか使えるレベルのMarkdownエディタを作ることができた．まだまだ必要な機能やバグもたくさんあると思うが，積極的に対応していきたいので報告してほしい．読者の日々の研究活動において，原稿執筆が少しでも楽しいものになれば幸いである．

謝　辞

NowTypeは現在行っている研究遂行の為のサポートツールとして開発しました．本研究はJSPS科研費JP19H01882，JP19K21870，JP19H01874の助成を受けたものです．また，NowTypeの仕様を決めるにあたり，核融合科学研究所の高山有道助教から有意義なコメントをいただきました．ここに感謝申し上げます．

参 考 文 献

［1］ https://daringfireball.net/projects/markdown/
［2］ https://atsushi-m-ito.github.io/nowtype/
［3］ https://www.mathjax.org
［4］ https://katex.org
［5］ https://github.github.com/gfm/
［6］ https://commonmark.org
［7］ https://pandoc.org
［8］ https://www.electronjs.org
［9］ https://nwjs.io
［10］ https://nodejs.org