WORD

From College of Information Science

WORD編集部は44年目も営業しています号

目次

BibTeX フォーマットをパースしたい

文　編集部　puripuri2100

1　はじめに

参考文献の情報をまとめるフォーマットとして理工系では BibTeX というものがよく使われている。たとえば図 1 にあるように、arXiv では論文情報を生成するときに BibTeX フォーマットにしか対応していない。

BibTeX データは通常 `bibtex`や `pbibtex`などのコマンドラインツールで tex 形式などに変換し、それを LaTeX などから読み込むことで参考文献部分を出力するという形で使われると理解してよい。

そのため、BibTeX データを解釈して他の形式に変換するソフトウェアを用意することで、LaTeX などの TeX 系列以外のソフトウェアでもその文献データを使用することができる。今回は具体的には SATySFi という組版ソフトウェアで BibTeX データを使えるようにするために、bib2saty というソフトウェアを作成することを考え、実装した。

より汎用的な情報とするために、BibTeX 形式のデータをパースして struct や enum などからなるデータ構造に落とし込むための方法について報告する。

2　BibTeX フォーマットの概要

BibTeX フォーマットは概ね以下のような書き方をする。

```
@article{id1,
  tag1 = "text",
  tag2 = {text},
  page = 2,
}
@comment {
  comment 1
  comment 2
  comment 3
}
@misc{id2,
  tag1 = "text",
  tag2 = {text},
}
```

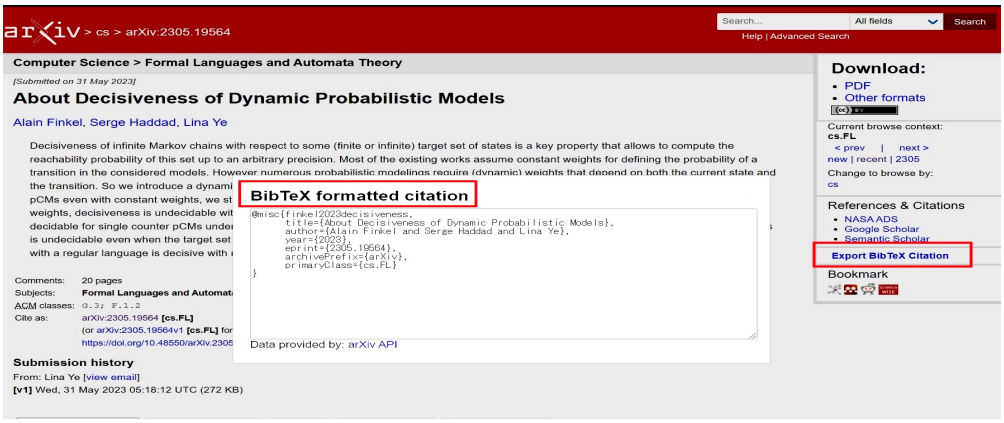

図 1　arXiv では BibTeX フォーマットでの情報しか自動で生成してくれない

`@`のあとには「資料の発表形態がなにか」を表す文字列を入れる（エントリと呼ばれる）。これは既に定まっているものが複数あるが、規約上は `[a-zA-Z]+`であれば何でも良い。具体的には

- `article`：雑誌
- `book`：書籍
- `phdthesis`：博士論文

などがある。これらのエントリの直後に波括弧で括られて詳細情報が書かれる。開き波括弧の直後にはこの参考文献に振られる ID を指定する。ここには日本語や空白などが含まれる場合もある。最後に、よくあるカンマ区切りの Key-Value 形式が与えられる。

ただし、例にあるように `@comment`の場合は波括弧内にどのような文字列を書いても良い。

BibTeX のより詳細なフォーマットについては公式サイトが詳しい[*1]。

3　BibTeX フォーマットの難しい点

ここまでだけを見ると、JSON に似た単純な記法だと思われるかもしれないが、BibTeX フォーマットを解析するうえで根本的に難しい点が存在する。それがテキスト（Key-Value 形式で書く部分で、各 key に対応する value のこと。`"text"`や `{text}`と書かれるもの。）の書き方である。このテキストには以下のような特徴がある。

- テキスト内で波括弧で括られた箇所は「加工禁止」となる
- テキスト同士の結合ができる
- テキスト内では LaTeX コマンドを使うことができる
- LaTeX コマンドの定義ができる

[*1] `http://www.bibtex.org/Format/`

それぞれについて詳しく述べる。

3.1 「加工禁止」について

通常は各論文誌で指定された引用文の書き方に合わせるためにテキストを処理する必要がある。しかし、その処理の過程で意図しないように変換されてしまう場合がある。これを防ぐために加工禁止を意図して波括弧でくくるのである。

例えば、複数著者は通常 andと語で自動で分割される。`{T. Hoge and E. foo}`と書いてあった場合、これは「T. Hoge」と「E. foo」の 2 つの名前であると判定する。しかし、ここで「F. And」さんが居た場合困ることが起きる。つまり、`{T. Hoge and F. And and E. foo}`と書いてあった場合は「T. Hoge」と「F.」と「E. foo」の 3 つ名前であると判定されるのである。しかし、「F. And」を波括弧でくくることで意図しない分割を防ぐことができる。つまり、`{T. Hoge and {F. And} and E. foo}`と書くことで意図通り「T. Hoge」と「F. And」・「E. foo」の 3 つの名前であると判定されるようになる。

この加工禁止のための波括弧は入れ子にすることができる。

3.2 テキストの定義と結合について

テキストを変数に束縛し、その変数を使いまわすことができる。例えば

```
@string {
  foo = "Foo",
  bar = "Bar"
}
```

とファイル内に書くと、fooという変数が Fooというテキストを、barという変数が Barというテキストをそれぞれ表すようになる。そのため、`author = foo`とすると、これは`author = "Foo"`としたのと同様の結果が得られる。

また、テキストを結合することができる。テキスト同士を #記号で繋ぐだけで実現できる。そのため、`"hoge" # "fuga"`とするとこれは `"hogefuga"`としたのと同様の結果となる。また、これは変数を参照しても良く `"hoge" # foo`とすると `"hogefoo"`としたのと同様の結果となる。

3.3 LaTeX コマンドの使用と定義について

さて、それとは別にテキスト内では LaTeX 記法を使うことができる（[*2] 参照)。コマンドのほか、数式やアクセント記号も使うことができる。

`title = {\textbf{title}}`のように強調を行う場合や `title = {The $20^{th}$}`のように数式を使うことで上付き文字を表現する場合がある。

アクセント記号の例としては、著者名に使われるものがあり、例えば `author={Dunkel, J\"{o}rn}`のようなものがある。これは著者のアイデンティティに関わるものであり対応

[*2] https://www.bibtex.org/SpecialSymbols/

することが望ましい。

さて、ここで使用する LaTeX コマンドをファイル内で定義することもできる。

```
1 @preamble {"\newcommand{\test}{test}"}
```

のようにすると、自動生成される tex ファイルの冒頭にこのテキストが展開され、`\test` コマンドが使用できるようになるという具合である。この時もテキストの結合や定義と参照の機能を使うことができるようになる。

4 実装

ここまで BibTeX のフォーマットを確認したが、実際にこれを解析して SATySFi 形式に変換するソフトウェアを実装した。

今回は解析部分の実装について説明する。リポジトリは `https://github.com/puripuri2100/BibSATySFi` である。

4.1 方針

今回は完璧な解析を行うことは目指さず、ある程度の BibTeX フォーマットのファイルを解析できることを目標とした。具体的にサポートする記法を以下に示す。

- テキストの定義
- テキストの結合
- アクセント記号
- コメントの処理
- 各エントリの解析

また、サポートしない記法は以下に示す通りとなる。

- LaTeX コマンドの定義と使用
- 数式

字句解析器でアクセント記号などを含めたテキストの処理やエントリ中の各種記号のトークン化を行い、次に構文解析器でトークンを AST に変換する方法を取った。字句解析器を挟むことで、構文解析器では単純な BNF で表すことができるような構文のみを扱うことができるようになり、コードの読みやすさが向上する。

実装は OCaml で行った。字句解析器は Sedlex[*3] という字句解析器ジェネレータライブラリを用いて生成し、構文解析器は Menhir[*4] という構文解析器ジェネレータを用いた生成した。

[*3] `https://github.com/ocaml-community/sedlex`
[*4] `https://gitlab.inria.fr/fpottier/menhir`

4.2 字句解析

基本的には

- `,`
- `=`
- `#`
- `@[a-zA-Z]+`
- `\OE`
- `\'{A}`

というような記号類が出てきたときにこれらを該当するトークンや文字列に置き換えるだけの実装ではあるが、波括弧（`{`と `}`）は

- エントリの開始終了
- テキストの開始終了
- 加工不可の指定の開始終了

の 3 つの意味があり、単純なトークン化を行うだけでは構文解析の段階で躓く原因となりかねない。そこで、字句解析器に対して「モード」の概念を導入することでこれを解決した。

モードを

- トップレベルモード
- エントリモード
- テキストモード
- 加工不可モード

の 4 種類用意し、それぞれによって波括弧の持つ役割を切り替えていく。具体的には

- トップレベルモード
  - `{`：エントリモードの開始
  - `}`：エラー
- エントリモード
  - `{`：テキストモードの終了
  - `}`：エントリモードの終了
- テキストモード
  - `{`：加工不可モードの開始
  - `}`：テキストモードの終了
- 加工不可モード
  - `{`：加工不可モードの開始
  - `}`：加工不可モードの終了

という場合分けを行う。加工不可モードは入れ子が可能なため、今どの深さの加工不可モードであるかの情報も保持しておく必要がある。

4.3 構文解析

字句解析器によってテキストの複雑さが吸収されて綺麗なトークンに変換されると、以下の BNF like な記法で表されるような構文として BibTeX フォーマットを記述することができる。

```
<bib> := <entry> <bib> | null
<entry> :=
  ENTRY_TYPE "{" STRING "," <value_list> "}"
  | "@string" "{" <text_def_list> "}"
  | "@comment" "{" <dummy_value> "}"
  | "@preamble" "{" <dummy_value> "}"


<value_list> := <value> | <value> "," <value_list> | null
<value> := STRING "=" <text>


<text_def_list> :=
  STRING "=" <text>
  | STRING "=" <text> "," <text_def_list>
  | null

<text> := TEXT | TEXT "#" <text>
```

このうち、

- TEXT
- ENTRY_TYPE
- STRING
- "{"
- "}"
- "="
- "#"
- ","

は字句解析器によってトークン化されている。

これらを BNF like な記法で示した構文にしたがって

```
type cite_key = string


type text =
  | Text of string
  | RawText of string
  | Int of int
  | DefText of string


type entry_type =
  | Article
  | Book
  | Booklet
  | Conference
  | InBook
  | InCollection
  | InProceedings
  | Manual
  | MastersThesis
  | Misc
  | Online
  | PhDThesis
  | Proceedings
  | TechReport
  | UnPublished
  | OtherEntry of string


type entry = {
  entry_type : entry_type;
  cite       : cite_key;
  value_list : (string, (text list)) Hashtbl.t;
}
```

によって定義される AST に単純に変換することで構文解析が完了する。

5　おわりに

この記事では簡単ではあるが BibTeX フォーマットの解析方法を示した。

字句解析器に「現在のモードによって文字の持つ意味を切り替える」という機能を入れることで複雑な構文に対しても対応することができる。

今後は LaTeX コマンドへの対応を拡充していきたいと考えている。具体的には `\textbf` や `\textsc`などのフォントを切り替えるコマンドを検知して対応するユニコード文字に自動で置き換えるなどである。

また、SATySFi のライブラリの形で BibTeX パーサーを実装していきたいと考えている。この豊富な資産である BibTeX フォーマットに対応することができればユーザの体験が向上するであろう。

やはり俺の中国語組版はまちがっている

文　編集部　いなにわうどん

1　はじめに

ご無沙汰しております、いなにわうどんです。最近仕事の方で中国語組版に関わる機会があり、その際に少し違和感のある組版と遭遇しました。図 1 は感嘆符（！）を含む縦組の文章を示したものです。和文（図 1 左）では感嘆符が中央に配置されるのに対し、中文（図 1 右）では感嘆符が右側に配置され、日本語組版に親しんだ我々にとっては些か不自然に見えます。

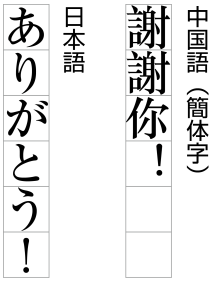

図 1　日本語組版と中国語組版の比較

文献を当たったところ、W3C の Chinese Layout Task Force [*1] から発行された Requirements for Chinese Text Layout（通称 CLReq）という技術ノートに約物[*2] の配置に関する記述がありました。CLReq 曰く、中国語の簡体字の文章において、図 1 のように感嘆符を右側に配置することは正当な組版規則であり[*3]、筆者の中国語組版に対する認識が誤っていたというわけです[*4]。

W3C[*5] からは、日本語組版の規則を示す日本語組版処理の要件（通称 JLReq）[*6] が公開されており、CLReq はこれの中文版と捉えることができます。CLReq を読み進めてみたところ、その他にも JLReq との差異を見出すことができたため、本稿ではその一部を紹介しながら興味深き中国語組版を紐解きます[*7]。

[*1] https://w3c.github.io/clreq/homepage/
[*2] 組版に使用する記号のこと。句読点、括弧類など
[*3] 後述しますが、繁体字の場合は日本語と同様に左右中央に配置します
[*4] タイトル回収
[*5] 原点を辿ると JLReq は JIS X 4051 を準拠とするため 1993 年まで遡ります
[*6] Requirements for Japanese Text Layout. https://www.w3.org/TR/jlreq/
[*7] 本稿では組版ルールに焦点を当て、具体的な実装までは言及しません。CLReq は英語 + 中国語で記述されているため、組版用語の日本語訳は筆者によるものです

1.1 CLReq の構成

CLReq は以下の 4 章＋付録から成立し、JLReq を踏襲した構成となっています。本稿ではこのうち 1–3 章を取り上げます。

1. 序論：Introduction
2. 中国語組版の基本：Basics of Chinese Composition
3. 行の組版処理：Line composition
4. 見出し・注・図版・表・表現の配置処理：Positioning of Headings, Notes, Illustrations, Tables and Expressions

2 序論

CLReq 第 1 章では序論として以下に掲げる中国語組版の特徴が示されています。以下の 2, 3. は日本語とも共通した特性です。

1. 中国語は繁体字・簡体字と 2 つの字体を有する。独自の地域標準（中国本土、台湾、香港、マカオ、シンガポール、マレーシア等）として、異体字やストロークの数、配置規則等が異なる場合もある
2. 縦組・横組と 2 つの書字方向を持つ
3. 漢字や約物は 1:1 の正方形で設計される

その後は CLReq の編集方針として、主に欧文組版や JLReq との差分について焦点を置いたことや、主に書籍を対象とすることなどが言及されています。

3 中国語組版の基本

CLReq 第 2 章では中国語組版の基本的な事項が記されています。ここでは中国語組版に特徴的な項目のみを取り上げます。

3.1 使用文字種

中国語を構成する文字は漢字・句読点・英数字・ラテン文字等が混在しています。主な部分を占める漢字には繁体字（Traditional Chinese）と簡体字（Simplified Chinese）が存在し、それぞれ以下の地域で使用されています[*8]。

繁体字　台湾、香港、マカオ
簡体字　中国本土、シンガポール、マレーシア

[*8]同一言語に 2 つの文字体系が存在する背景には、1950 年代に中華人民共和国で推められた「文字改革」政策によって簡体字が誕生したという歴史があります

3.2 組体裁

組体裁は、文字と文字に間隔を入れないベタ組み（密排、solid setting）を原則としますが、以下の場合は等間隔のスペースを入れる（疏排、loose setting）[*9] ことがあります。

- 文字数の少ないランニングヘッド（柱）
- 文字数が少ない図表キャプション（図表とのバランスを取るため）
- 一行に数文字程度しかない詩
- 子供を主な読者とする出版物

後述する禁則処理等によって行長に過不足が生じた場合には、日本語同様に行頭と行末を合わせる均等揃えも一般に行われます。加えて、雑誌の見出し等の文字組みでは、詰め組みも行われています[*10]。図 2 に中国語組版での文字組みの例を示します。

図 2　中国語の組体裁

3.3 書体

日本語では明朝体、ゴシック体が主に選定されますが、中国語では次の 4 書体（図 3）が代表的です。宋朝体は日本でも存在する[*11] ものの、あまり馴染み深くはない気がします。

明朝体（宋体、Song**）**
: 本文や見出し等に幅広く使用される。

楷書体（楷体、Kai**）**
: 公文書や教科書等に使用される（台湾の公文書はすべて楷書体）。見出しや会話文等、他と区別すべき部分に使用される。

ゴシック体（黒体、Hei**）**
: 見出しや看板等に使用される。印刷書体として採用される機会は極めて稀だったが、時代の変化とともに徐々に登場頻度を高めつつある。

[*9] 日本語では等間隔に文字を空けて組むことが少なく、適切な訳語が見当たらないため、本稿では便宜上「均等空け」と呼ぶことにします

[*10] CLReq 中では、文字を均等に詰めた（1 歯詰め等）図が掲載されています。漢字は基本的に字面が均一であるため、プロポーショナル詰めではなく、均等詰めだけでも十分な効果が得られるのかもしれません

[*11] 写研からは 1965 年に石井宋朝体（`https://archive.sha-ken.co.jp/typeface/075-0-LS/`）が発表されています

宋朝体（仿宋体、Fangsong）

明朝体と楷書体の中間に位置する書体で、副題や引用文等に使用される。中国本土の公文書は宋朝体で記述される。

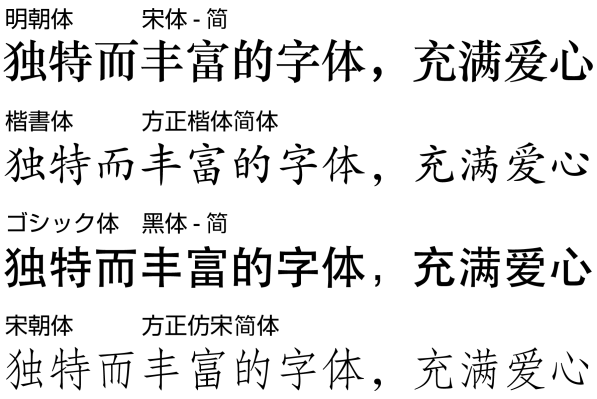

図 3　中国語組版で主に使用される書体

3.4　サイズ表記

印刷技術の変遷に伴い、サイズ表記として活版時代には号（hào）と呼ばれる単位が、写植時代には日本でもおなじみ級数[*12]が、現在は DTP の普及により DTP ポイントが導入されました。しかし、現在でも号での指定が主流なようです。また、行長は文字サイズの整数倍とし、行間は文字サイズに対して 50–100% 程度で指定します。

3.5　書字方向

中国語では、縦組・横組の 2 つの書字方向が存在しますが、これらの使用場面や頻度は地域によって異なります。

台湾・香港

縦組・横組の両方が使用される。特に台湾では、政府機関による頒布物、教材、自然科学系の書籍は主に横組が、詩や小説などの文学作品には縦組が採用される。

中国本土

ほとんどの出版物が横組で組まれる。従来、中国語の出版物は縦組で構成されていたものの、翻訳出版物の増加や横組を主に扱うワープロの影響を受けて、横組が主流となったとされる。

縦組の文章に英数字が混在する場合は、日本語同様に 1 文字ずつ正立、全体として 90 度回転、縦中横[*13]のいずれかで処理されます。漢字と英数字が隣接する場合は四分アキを挿

[*12] 写真植字で導入された単位。1 級（Q）= 1 歯（H）= 0.25 mm

[*13] 縦組の行中に、数字等を横組で配置すること

入します[*14][*15]。また、全体の書字方向が縦組の場合でも、図表・キャプション・見出し等に限っては横組とする場合があります。

4　行の組版処理

CLReq 第 3 章は行組版処理をトピックとした章です。約物、注・ルビ、段落調整処理、行調整処理に関心を分けて第 3 章の内容を概説していきます。

4.1　約物の扱い

句読点

文末には句点「。」（IDEOGRAPHIC FULL STOP、U+3002）を、節等の文章の区切りには全角カンマ「,」（FULLWIDTH COMMA、U+FF0C）を挿入します。日本語における読点[*16]「、」（IDEOGRAPHIC COMMA、U+3001）は、3 つ以上の項目からなるリストの要素を区切る目的で主に使用されます。テン「、」とカンマ「,」を区別する点が特徴的です。ただし日本語同様、学術書等の欧文を多く含む文章では全角ピリオド、全角／半角カンマを句読点として使用することもあります[*17]。

配置方法も独特で、簡体字（中国本土）では日本語と同様に四隅に配置するのに対し、繁体字（台湾・香港）では文字枠の中央に句読点を配置します（図 4）。これらの配置の違いは Unicode 上では区別されません。「はじめに」で述べた感嘆符「！」や疑問符「？」は縦組時に配置が異なり、簡体字では右側に、繁体字では中央に配置されます。

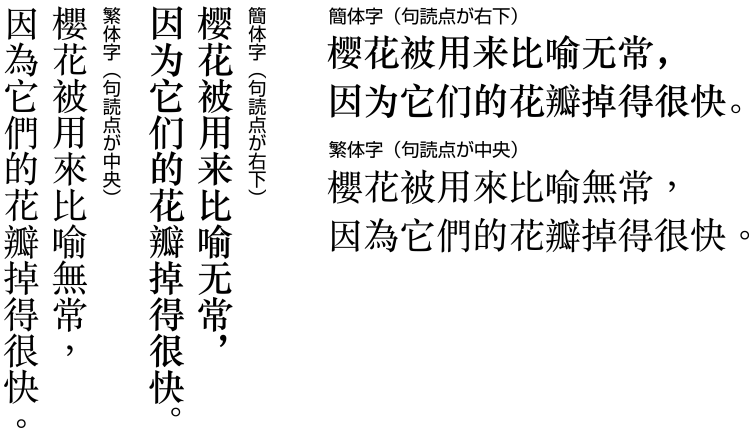

図 4　簡体字・繁体字での句読点の配置

[*14] 最近 Twitter でも議論を呼んだ話題ですが、CLReq には四分アキの代わりに半角スペース（SPACE、U+0020）を挿入することがあると明示されています

[*15] 行長に過不足が生じる場合は二分までスペースを広げることが可能で、こうすることで前後の漢字をベタ組みとして組み上げることが可能となります

[*16] CLReq では secondary comma と呼称されています

[*17] o（オー）や O（ゼロ）と句点の混同を避けるためだそうです

引用符、ダッシュ

縦組の場合は鉤括弧、横組の場合はクォーテーションと使い分けがなされます。鉤括弧の使い方にも地域差があり、台湾では鉤括弧（「」）をまず使用し、鉤括弧の中で二重鉤括弧（『』）を使用します。一方の中国本土は逆で、二重鉤括弧またはダブルクォーテーション「“”」を適用してから、鉤括弧ないしシングルクォーテーション（‘’）を使用します。ダッシュや三点リーダを使用する場合は倍角にします[*18]。

その他約物類

圏点

横組の圏点は文字の下側に、縦組の圏点は文字の右側に配置します。

中黒

中黒「・」の幅は台湾・香港では全角、中国本土では半角（二分）です。

書籍名・固有名詞

書籍名を表す場合は、低い波線「﹏」（WAVY LOW LINE、U+FE4F）を文字の下（縦組では左）に配置するか、二重山括弧「《》」で囲みます。固有名詞や人名の下には下線を引きます。圏点とこれらの線は、横組の場合は片面設定（single-sided setting、必ず下側に付されることから）と、縦組の場合は両面設定（double-sided setting、左右いずれかに付されることから）と呼ばれます。

故人の表現

人名を黒い罫線で囲むことで故人を表現します[*19]。最近亡くなった方を表すために用いられ、広く知られた故人には適用されません。

4.2　注、ルビの扱い

注音符号、ピンイン

中国語にも行間注（Chinese interlinear annotation）と呼ばれるルビが存在し、漢字の発音や意味を示すために使用されます。ただし、教育目的として文章全体にルビを振ること（総ルビ）が中心で、日本での用例のように難読漢字のみにルビを振ることは少ないようです。

台湾では発音を示すのに注音符号（Zhuyin）が用いられ、縦組・横組いずれの場合も文字の右側に密着して配置するのが特徴的です。注音符号と親文字のサイズの比率は 3:10 を基本とし、中付き[*20] にします（図 5）[*21]。

一方の中国本土ではピンイン（拼音、Pinyin）が公式に採用され、横組の文章に限定して使用します。文字を基準に発音を示す場合は文字の上側に置き、単語を基準とする場合は

[*18] ダッシュに関しては「—」（EM DASH、U+2014）を 2 つ続けることが多いですが、本来は「⸺」（TWO-EM DASH、U+2E3A）の使用が推奨されています

[*19] 寡聞なもので†最強†などと適当な使い方ばかりしていたのですが、欧米では短剣符「†」を故人の表現に使用するとのことです

[*20] 親文字に対してルビを中央揃えにすること

[*21] CLReq ではその他に多くの配置規則が紹介されていますが、本稿では割愛します

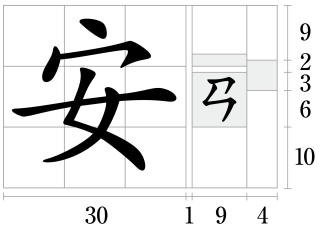

図 5　注音記号の配置（CLReq に登場する図を基に筆者作成）

２字下げ

校服和教科書不得擅自更改。這很有趣，但我繼續度過一些空閒時間。我只是想弄清楚如何表現才能假裝冷靜。如果這就是愛，那麼日常世界就脆弱到足以失去孤獨的力量。突出的感情仍然膽小，每個人都會受到傷害。

凸排

香　港：中華人民共和國南部的一個特別行政區。它由九龍半島、香港島和附屬小島組成。

図 6　2 字下げと凸排

上側ないし下側に配置します。ピンインのサイズは親文字の半分で、これは日本語のルビの影響を受けたものです。注音符号、ピンインを併記することも可能で、この場合は注音符号を右側に、ピンインを下側（縦組みであれば左側）に表記します。いわゆる両ルビです。

二言語注釈、行間注

ライトノベル等で外来語をそのまま表現する際には、二言語注釈（bilingual annotations）が使用されます。表現としては、中国語に対して外来語のルビを振るかその逆のいずれかで、日本語のルビと同じ方向に配置します。また、行間注（interlinear comments）として行間にコメントが付される場合もあります[*22]。

4.3　段落の調整処理

字下げ

日本語では段落先頭を 1 字下げにしますが、中国語では 2 字下げが基本です。ただし雑誌のように多段組の出版物では 1 字空けを採用する場合もあります[*23]。この他に凸排（tupai、itemization）と呼ばれる 2 行目以降を字下げとする配置も存在します（図 6）。一行目の頭に人名・項目名 + コロン（：）を置く場合に使用され、3 文字 + コロン分の計 4 em を字下げ幅として確保します。

[*22]注が複数行に亘ることもあるようで、後注のような位置づけなのかもしれません
[*23]最初の段落の字下げを行わなかったり、すべての段落の字下げを無視したりする指定も可能です

ウィドウ、オーファン

組版にはウィドウ（widow）、オーファン（orphan）なる概念が存在し、CLReq では次の通りに言及されています：In the tradition of Chinese composition, an orphan does not make a line, nor does a widow make a page.

ウィドウは段落の最後行が次ページの先頭行に位置する現象を指し、オーファンは段落の最終行に 1 字だけが残る現象として定義されています[*24]。ウィドウおよびオーファンは、それぞれ以下の操作を通じて回避することができます。

■ウィドウの回避

- ウィドウを前ページに移動させ、版面を超えるように配置する
- 前ページの最終行を次のページに移動させ、ウィドウと合体させる
- 文字数を減らす
- 文字数を増やして、次ページに跨る文章が 2 行以上となるように調整する

■オーファンの回避

- 最終行の 1 行前の最終文字を押し出す
- 段落の文字数を減らす
- 最終行の文字数を増やす

4.4 行の調整処理

禁則処理

禁則処理に関しては、中国本土では GB/T 15834—2011 として国家標準規格が制定されているほか、台湾・香港でも慣習的なルールに基づいて処理が行われます。ただし台湾・香港の新聞組版では禁則処理を行わない場合もあります。また、ぶら下がり[*25] は行われません[*26]。

行末調整と文字組みアキ量設定

複数行にわたる中国語の文章では、最終行を除いて両端揃えを基本とします。したがって欧文の混在、禁則処理、約物の連続等を要因として行長に過不足が生じた場合に、行の調整を目的として以下の処理が行われます。

- 欧文スペースや約物のアキ量を詰める（いわゆる追い込み）
- 分離禁止文字でない文字間を空ける（いわゆる追い出し）

一般には詰める処理を最初に試み、それだけでは調整不能だった場合に空ける処理を行

[*24] 日本語組版や欧文組版の文脈では、オーファンはページの最終行に次段落の先頭行が残ることを指すため（出典：`https://note.morisawa.co.jp/n/n245d0ff3241a`）、中国語組版におけるオーファンの定義とは乖離があるようです。CLReq ではウィドウとオーファンが共存するケースがあるとも解説されています

[*25] 行末に句読点が位置する場合、その句読点を版面から飛び出して配置すること。「ぶら下げ」とも

[*26] 前述の通り繁体字では句読点を中央に配置するため、ぶら下がりを行うことで体裁の悪化が懸念されます

います。約物に関しては、中国本土や香港では、連続する約物や行末・行頭に位置する約物のアキ量を調節しますが、台湾ではベタ打ちとするケースが多いようです。追い込み・追い出し処理を行う際の手順を以下に示します。

■追い込みの順序

- 行末の約物の後をベタ（アキなし）にする
- 欧文スペースを最大で四分まで詰める
- 中黒の前後を最大でベタまで詰める
- 括弧の前後を最大でベタまで詰める
- カンマ・読点・セミコロンの前後を最大でベタまで詰める
- 漢字–欧文間スペースを最大で八分まで詰める*27
- 句点・感嘆符・疑問符を最大でベタまで詰める*28

■追い出しの順序

- 欧文スペースを最大で二分まで空ける
- 漢字–欧文間スペースを空ける*29
- 分離禁止文字を除く文字間を均等に空ける

5　むすびにかえて

日本語との共通点も多い中国語組版ですが、その組版規則に目を通すとやはり細やかな違いが見えてくるものです。特に繁体字と簡体字にみられる組版ルールの差異は、広大な面積や膨大な話者、様々な歴史的経緯を有する中国語ならではと感じました。本稿で紹介した組版規則は CLReq の一部ですので、興味を持たれた方はぜひ原文をお読みいただければと思います*30。

6　参考文献

- W3C Working Group: Requirements for Chinese Text Layout, 2023, `https://www.w3.org/TR/clreq/`.
- W3C Working Group: Requirements for Japanese Text Layout, 2020, `https://www.w3.org/TR/jlreq/`.
- 株式会社モリサワ: MORISAWA PASSPORT 英中韓組版ルールブック, `https://www.morisawa.co.jp/fonts/multilingual/typesetting/`.

*27 スペースは四分で固定とし、アキ調整が許容されないこともあります
*28 文章の切れ目を表すことから、これらのアキ調整を許容しない組版スタイルもあります
*29 前述の通りスペースを四分に固定するスタイルや、三分までしか空けることを許さないスタイルも存在します
*30 記事執筆のために CLReq を読破したところ DeepL 無料版の制限に達してしまったが Cookie を削除したら回復した

Solo5 Hack ことはじめ

文　編集部　yuseiito

1　何

こんにちは、yuseiito です。梅雨らしからぬ日が続いておりますが、いかがお過ごしでしょうか。本稿では、最近私が楽しく触っている Solo5 という OSS のミドルウェアについて紹介したいと思います。

紹介に入る前に、Solo5 の理解の前提となる Unikernel という概念について解説したのち、実際の Solo5 のコードを読み解いて実装の理解を試みます。そしてそれを踏まえて、最後に自身で KVM 環境で用いる hypervisor call を新たに追加する改造を行うことに挑戦します。

この一連の手順を通して、OSS のミドルウェアのコードを読み解く手順や、改造を施すことを通して実装を理解するフローについて、あまり経験のない方にも追体験していただけると嬉しく思います。

2　予備知識 ― Unikernel

Unikernel は、OS カーネルに相当する部分を Library OS として実装し、それらをハイパーバイザ上で動かす、一風変わった OS の形態です。

Library OS は、プログラミング言語のライブラリとして OS 機能を提供する、特殊な OS の形態です。汎用 OS では割り込みによって実現されるシステムコールを、純粋な関数呼び出しとして記述します。こういった設計にすることで、アプリケーションと OS が完全に一体化[*1] されるため、汎用 OS に比べて以下のようなメリットが享受できます。

- 当該アプリケーションに必要な極小の OS 機能のみ選択してビルドできる。[*2]
- イメージサイズが小さく、起動が高速
- コード量が少なく、セキュリティ性を高めやすい

一方で、Library OS ではあるデバイスに特化した OS 機能とアプリケーションが完全に一体化してしまうため、ハードウェア間でのバイナリ互換性はほぼなくなってしまいます。

そこで、これらのメリットを享受しつつ、ハードウェア間でのバイナリ互換性を保つために、ハイパーバイザ上で動く仮想マシンの OS としてこれを採用しようというのが、Unikernel のアイデアです。

Unikernel では、照準を特に利用対象をサーバ、特にクラウドなどのマルチテナントサー

[*1] 静的リンク

[*2] この選択は明示的にせずとも「使用されている関数のみを含める」という簡素なルールに基けばよいため、Link Time Optimization などを利用すれば容易に実現できる

バに合わせています。こういった分野で従来用いられてきた方法として、大きく分けてコンテナと VM の 2 つがありますが、ホストカーネルを共有するコンテナと比べて、Unikernel はマシン間でホスト OS を共有しないため、より強い隔離が実現されます。また、汎用 OS を利用した VM に比べても、巨大な汎用 OS を読み込むわけではない分パフォーマンスが大きく改善します。

こういった特性は、特に Function as a Service や IoT などの分野で有用であると考えられており、研究が行われています。

3 Solo5

前章で、Unikernel はハイパーバイザの上で動かすことでハードウェア互換性の問題を回避していると述べました。

しかし、世の中には様々なハイパーバイザが存在します。つまり、本当に portable なプログラムを得るには、ハイパーバイザ間の互換性を吸収するレイヤが必要です。そのためのミドルウェアが、今回のテーマとなる Solo5 です。

Solo5 は、単一のコードを Xen, KVM, muen といった各種実行環境で動かすことができます。

そのために、Unikernel 側に対して Solo5 の共通インタフェースを提供するための **bindings** と、各種ハイパーバイザを Solo5 の共通インタフェースに変換するための **tenders** が用意されています。

ユーザは、対象とするハイパーバイザ向けの binding をアプリケーションに組み込んで Unikernel をビルドし、実行時には tenders を通してハイパーバイザを呼び出すことで、インタフェースの違いをすることなく Unikernel を取り扱うことができます。

4 Solo5 における Hypercall の実装

Solo5 での Hypervisor call は、対象とするハイパーバイザおよびアーキテクチャによって実装が異なっています。[*3]

ここでは、KVM を対象とした x86_64 アーキテクチャにおける実装を見ていきながら、適宜他の環境についても補足します。

そもそも、Solo5 において、hypervisor call は以下のように呼ばれます。ここでは、`solo5_console_write` が hypervisor call にあたります。

```
1 #include "solo5.h"
2
3 int solo5_main(const struct solo5_start_info *si){
4     solo5_console_write("Hello, world\n", 13);
5     return SOLO5_EXIT_SUCCESS;
6 }
```

[*3] この差異を吸収するのが Solo5 の目的なので当然ではある

これに対応する KVM 向けの実装は、bindings/hvt/console.c にある以下の部分です。hvt というのは、KVM 向けの tender のことで、Hardware Virtualized Tender の略です。

```
1  #include "bindings.h"
2
3  int platform_puts(const char *buf, int n){
4      struct hvt_hc_puts str;
5      str.data = (char *)buf;
6      str.len = n;
7      hvt_do_hypercall(HVT_HYPERCALL_PUTS, &str);
8      return str.len;
9  }
10
11 void solo5_console_write(const char *buf, size_t size){
12     (void)platform_puts(buf, size);
13 }
```

texttthvt_do_hypercall は、x86_64 向けには以下のように実装されています。(include/hvt_abi.h)

```
1  static inline void hvt_do_hypercall(int n, volatile void *arg
       ){
2    // (中略)
3      __asm__ __volatile__("outl %0, %1"
4              :
5              : "a" ((uint32_t)((uint64_t)arg)),
6                "d" ((uint16_t)(HVT_HYPERCALL_PIO_BASE + n))
7              : "memory");
8  }
```

実装は、単一の outl 命令です。outl 命令は、x86_64 の I/O ポートに対して書き込みを行う命令です。つまり、solo 5 における x86_64 KVM 環境の hypervisor call は、I/O ポートへの書き込みとして表現されています。

では、この IO ポートへの書き込みは、どのようにしてホスト Linux へと伝播するのでしょう。先に説明した通り、Solo5 において、ハイパーバイザを Solo5 向けインタフェースに変換するのが tender の役目でした。ですから、我々の求める実装は tender にありそうです。

実際、tenders/hvt/hvt_kvm_x86_64.c にあります。IO ポートへの書き込みは、KVM の exit, 特に KVM_EXIT_IO の状態として、ホスト Linux へと伝播されます。

そして、その際に書き込まれたポート番号から hypervisor call 番号を特定し、書き込まれたアドレスに存在する構造体の内容から hypervisor call の引数を取得します。

```
int hvt_vcpu_loop(struct hvt *hvt){
    struct hvt_b *hvb = hvt->b;
    int ret;
    while (1) {
        ret = ioctl(hvb->vcpufd, KVM_RUN, NULL);
        //  (中略)
        struct kvm_run *run = hvb->vcpurun;
        switch (run->exit_reason) {
        case KVM_EXIT_IO: {
        // (中略)
            hvt_hypercall_fn_t fn = hvt_core_hypercalls[nr];
            if (fn == NULL)
                errx(1, "Invalid guest hypercall: num=%d", nr
                    );

            hvt_gpa_t gpa = *(uint32_t *)((uint8_t *)run + run
                ->io.data_offset);
            fn(hvt, gpa);
            break;
        }

        case KVM_EXIT_FAIL_ENTRY:
        // 以下、略
}
```

ここで呼ばれる hvt_core_hypercalls は、hypervisor call の実体の関数ポインタを整列したものです。

ここで、tenders/hvt/hvt_core.c の記述を見てみましょう。

```
static int setup(struct hvt *hvt, struct mft *mft){
    if (waitsetfd == -1)
        setup_waitset();

    assert(hvt_core_register_hypercall(HVT_HYPERCALL_WALLTIME,
```

```
6                   hypercall_walltime) == 0);
7       assert(hvt_core_register_hypercall(HVT_HYPERCALL_PUTS,
8                   hypercall_puts) == 0);
9       assert(hvt_core_register_hypercall(HVT_HYPERCALL_POLL,
10                  hypercall_poll) == 0);
11
12      return 0;
13  }
```

ここで、hvt_core_register_hypercall で、hypervisor call 番号と実体の関数ポインタを登録しています。

例えば、今追っている hypercall_puts は、以下のように実装されています。

```
1   static void hypercall_puts(struct hvt *hvt, hvt_gpa_t gpa){
2       struct hvt_hc_puts *p =
3           HVT_CHECKED_GPA_P(hvt, gpa, sizeof (struct hvt_hc_puts
                ));
4       int rc = write(1, HVT_CHECKED_GPA_P(hvt, p->data, p->len),
            p->len);
5       assert(rc >= 0);
6   }
```

はい、ついに辿り着きました。file descriptor 1、すなわち標準出力への書き込みが行われています。

ちなみに、ここまで適当にはぐらかしてきた hypercall 番号は、include/hvt_abi.h に定義されています。

```
1   enum hvt_hypercall {
2       /* HVT_HYPERCALL_RESERVED=0 */
3       HVT_HYPERCALL_WALLTIME=1,
4       HVT_HYPERCALL_PUTS,
5       HVT_HYPERCALL_POLL,
6       HVT_HYPERCALL_BLOCK_WRITE,
7       HVT_HYPERCALL_BLOCK_READ,
8       HVT_HYPERCALL_NET_WRITE,
9       HVT_HYPERCALL_NET_READ,
10      HVT_HYPERCALL_HALT,
11      HVT_HYPERCALL_MAX
12  };
```

5 Hypervisor call を追加する

では、ここまでみてきた実装を踏まえて、hypervisor call を追加してみます。今回は、ホストを確かに呼べていることを確認するために、ホストのホスト名[*4] を取得する hypervisor call を追加します。

まず、include/hvt_abi.h に、新しい hypervisor call 番号を追加します。

```
enum hvt_hypercall {
  // (中略)
    HVT_HYPERCALL_HALT,
    HVT_HYPERCALL_HOSTINFO, // NEW!
    HVT_HYPERCALL_MAX
};
```

さらに、同じファイルに、新しい hypervisor call の引数の構造体を定義します。

```
struct hvt_hc_hostinfo {
  HVT_GUEST_PTR(const char*) data;
  size_t len;
};
```

次に、tenders/hvt/hvt_core.c に、新しい hypervisor call の実体を追加します。なお、ファイル先頭に #include <fnctl.h> を追加しておきます。

```
static void hypercall_hostinfo(struct hvt *hvt,hvt_gpa_t gpa){
    struct hvt_hc_hostinfo *p = HVT_CHECKED_GPA_P(hvt, gpa,
        sizeof (struct hvt_hc_hostinfo));
    int fd = open("/proc/sys/kernel/hostname",O_RDONLY);
    if(fd==-1) err(1,"Failed to open /proc/sys/kernel/hostname
        ");
    char* buf = (char*) HVT_CHECKED_GPA_P(hvt,p->data,p->len);
    if(read(fd,buf,p->len) == -1) err(1,"Failed to read /proc/
        sys/kernel/hostname");
    close(fd);
}
```

tender 側の改造の最後に、tenders/hvt/hvt_core.c の setup 関数に、新しい hypervisor call の登録を追加します。

[*4] まぎらわしい。ホスト OS の hostname のこと。

```
static int setup(struct hvt *hvt, struct mft *mft){
    if (waitsetfd == -1)
        setup_waitset();
    //  (中 略)
    assert(hvt_core_register_hypercall(HVT_HYPERCALL_HOSTINFO,
                hypercall_hostinfo) ==0); // HERE!

    return 0;
}
```

最後に少し、bindings も変更しておきましょう。include 以下のファイルは bindings と tender で共有されている宣言のため、bindings 側の改造の大部分は実は tenders の改造と同時に完了しています。

変更すべきは、実際に呼び出す部分だけです。bindings は、hvt_do_hypercall を呼び出していたのでした。ここでは、[*5] bindings/hvt/console.c に追記します。

```
void solo5_hostinfo(const char *buf, size_t size){
  struct hvt_hc_hostinfo m = {
    .len = size,
    .data = (char*) buf
  }

  hvt_do_hypercall(HVT_HYPERCALL_HOSTINFO, &m);
}
```

最後に、この関数のプロトタイプを include/solo5.h に追加します。

```
void solo5_hostinfo(const char *buf, size_t size); // ADD
```

これで改造が終わりました。docs/building.md に従って、改造した tender や bindings をビルドして、以下の unikernel を動かしてみましょう。

```
#include "solo5.h"
#include "bindings/lib.c" // strlen()

static void puts(const char *s){
  solo5_console_write(s, strlen(s));
```

[*5] 本来は新規ファイルに書き込むべきだが、ズボラな私が Makefile の改変を避けるため。

```
}

int solo5_app_main(char *cmdline){
  char buf[256];
  solo5_hostinfo(buf,256);
  puts("Host machine's name:\t");
  puts(buf);
  return SOLO5_EXIT_SUCCESS;
}
```

実行すると、以下のような出力が得られるはずです（sebastian は私の開発マシンのホスト名です。[*6]）。

```
sebastian% make run
../../toolchain/bin/x86_64-solo5-none-static-cc -o -z solo5-
    abi=hvt kernel.hvt manifest.c test_hostinfo.c
../../tenders/hvt/solo5-hvt --mem=2 -- kernel.hvt
             |      ___|
  __|  _ \  |  _ \ __ \
\__ \ (   | | (   |  ) |
____/\___/ _|\___/____/
Solo5: Bindings version v0.7.3-4-g4c570de-dirty
Solo5: Memory map: 2 MB addressable:
Solo5:   reserved @ (0x0 - 0xfffff)
Solo5:       text @ (0x100000 - 0x103fff)
Solo5:     rodata @ (0x104000 - 0x105fff)
Solo5:       data @ (0x106000 - 0x10afff)
Solo5:       heap >= 0x10b000 < stack < 0x200000
Host machine's name:    sebastian
Solo5: solo5_exit(0) called
```

確かに、hostname が取得できていますね。

6 まとめ

本稿では、Unikernel についてごく簡単に紹介したのち、Solo5 という Unikernel ミドルウェアの実装の解読を試み、最後に簡単な改造を試みました。

[*6]余談だが、私はディズニー映画のファンで、利用しているマシンのホスト名を慣習としてディズニーキャラクタの名前にしている。sebastian は、「リトルマーメイド」に出てくるカニである。人間の世界は最低だよ。

なお、わかりやすさを優先したため、若干無理な改造を行っています。本来、ホスト OS のホスト名が知りたいのなら直接 `/proc/sys/kernel/hostname` をマウントして読み出せば良いですし、そもそも hypercall が追加したい場合はこのように直接書き足すのではなく Solo5 に準備された module 機能を使うべきです。[*7]

しかし、このような簡素なインラインの改造を行うことは、ソフトウェアの構造を理解する上で非常に有効な手段であり、OSS を読む速度を上げる効果的な方法であると言って良いでしょう。ちょっとした オープンソースソフトウェアを読み解く感覚や、それに対して改造を施すことで理解を深める感覚が伝わっていれば幸いです。

[*7] 実際、ブロックデバイスの操作は `block` モジュールに、`net` モジュールにそれぞれ実装されており、core 側には含まれていない。

生活が崩壊しているオタクのための手抜き生活術（食事編）

文　編集部　maetin

1　はじめに

こんにちは、maetin です。突然ですが皆さんは生活（食事、部屋の片付け等）が終わっていますか？ 周りのオタクを見ていると、パソカタをする気持ちはあっても食事や居住空間の整理など生存に必要な行為がおろそかになっている人が多いなぁと感じます。パソカタが得意な人が生活のリソースを削ってパソカタ業をすることで様々な成果を出しているという一面もあるので一概に否定をすることはできないですが、それで体調や精神を崩すことでパソカタ業に支障をきたしている人も多いのではないでしょうか。そこで今回は怠惰な自分が実際に行っている自分のリソースと相談しながらやる生活の手抜き術、特に食事を紹介しようと思います。

2　食事編

食事って実は面倒ですよね、分かります。いつの間にかお腹が減っていて集中の妨げになるからと適当な菓子なり清涼飲料水で腹を満たしてしまう人も多いのではないでしょうか。良くないですね。そういった時、私は以下の面を考慮して手抜きをしています。

- 食事の用意
- 食べる手間・時間
- 食器等の片付け

2.1　食事の用意

食事の用意は、自分で作るか買うかの二択になります。自分で作る場合は、食材の買い出し、調理、後片付けという手間がかかります。また、買う場合は、買いに行く手間とお金がかかります。どちらも面倒ですね。自分が食事の用意をするときはその日の食事の用意にかけられるリソース順に以下のように考えます。

1. 家にある食材で適当に炒めもの等を作る
2. 冷凍ご飯を温めてレトルトカレーや納豆などと一緒に食べる
3. BASE BREAD を適当にトッピングして食べる
4. 外食やコンビニで買ってくる
5. Uber Eats や出前館で注文する

家にある食材で適当に炒めもの等を作る

これは一番気持ちに余裕がある日か、家にある食材が腐りそうなときにやります。野菜が取れたり安くて美味しい飯が食べれたりというメリットがありますが、欠点は一人暮らしで料理をするとバカの量を作ってしまい、ドカ食いになってしまう場合があることです。

しかし欠点を補ってあまりあるほど食事の自由度が増えるため、ぜひやるべきでしょう。

私がよく行うのは気が向いたときにスーパーなどに行き肉や野菜を大量に買いだめて、冷凍しておくことで気が向いたときにいつでも肉と野菜を焼肉のタレで炒めたものを食べるということです。こうしておくことで、食材が腐るかどうかという認知コストと食材を買いに行く手間を減らしつつ、程よい頻度で野菜を摂取することができます。また、料理をする際に炊飯器で 3 合ほどご飯を炊いておくと、余ったご飯を冷凍しておくことで次に述べる食事の用意コストも一緒に下げることができて一石二鳥です。

冷凍ご飯を温めてレトルトカレーや納豆などと一緒に食べる

これは程よく手抜きができてかつ温かいごはんを食べながら食費の節約ができる方法です。前章で述べたようなご飯を冷凍しておくと、解凍する手間とレトルトを用意する手間だけで温かい食事ができ、非常に楽です。また、ご飯に合わせるものとしても

- レトルトカレー
- 納豆
- サバ缶等
- 冷凍の親子丼等
- 冷凍の惣菜

など様々な付け合せのレパートリーがあり、飽きが来ることも殆ど無いです。やはり日本人のソウルフード・ご飯は最強ですね。さらに、これに合わせて気軽に野菜分を摂取したくなった場合は、以下のものをジップロックに入れて揉むことでお手軽チョレギサラダもどきが作れます。食物繊維も摂れて最高ですね。

- キャベツ 1/8 玉
- ごま油 15 cc
- 塩適量
- だしの素適量

このように冷凍のご飯をストックしておくことで、食事の用意コストを下げることができるのでぜひおすすめです。ご飯を炊く行為がそもそも面倒な場合はパックご飯で代替することもできますが、パックご飯は自炊感がなくなり生活への自信を削ることになるので自分としてはあまりやっていません。

BASE BREAD を適当にトッピングして食べる

BASE BREAD[*1] という完全栄養食を導入することで栄養を取りつつ食事の用意コストを下げることができます。しかしデメリットとしては、食事の用意コストは下がるものの味がある程度落ちてしまうことです。そこで様々なアレンジ方法を用いて飽きを遅らせ、長期的に食べ続けることができるようにしています。私がよく行うのは以下のようなアレンジです。

- マヨを塗ってベーコンと一緒に焼く
- ツナマヨを作成し挟んで焼く
- 焼いた後ジャムを塗る
- レトルトスープを付け合わせる

マヨやジャムは冷蔵庫に常備しておくと良いでしょう。

外食やコンビニで買ってくる

いよいよ手抜き度が上がってきましたが、外食や惣菜を買う際にもある程度考えなければならないことがあります。外食を利用するメリットは

- 食事の用意コストが 0
- おいしい

ということですが、デメリットとしては

- 高い
- 栄養が偏りがち

ということがあります。なので、外食をする際には

- 外食に行くたびにジャンルを変えるようにする
- 野菜を多めに摂る
- 週に何回までかを決めておく

などをするとデメリットをある程度抑えられるでしょう。

Uber Eats や出前館で注文する

これは完全に最後の手段です。外食に比べて更に高くなる一方で外食と同じクオリティのご飯が食べられるため非常に便利ですが、非常に便利が故に使いすぎて破産したり太ったりしてしまうなどのリスクが大きいです。使う際は節度を持って使いましょう。

[*1] https://shop.basefood.co.jp/

2.2　食べる手間・時間

作業中で食べる時間や手間が惜しいということもあるでしょう。そんなときは以下のような方法を用いて食べる手間・時間を減らしましょう。

1. BASE BREAD
2. プロテイン
3. 納豆
4. カロリーメイト
5. 菓子類

できるだけ上のものを食べると良いですが、食事の手間が惜しいほどになっている場面では気遣いをするのも難しいでしょうから、食べないよりはお菓子でも食べたほうが良いと思われます。

2.3　食器等の片付け

食洗機を買いましょう。2 万円程度で 1 人暮らし用タンク式の食洗機が買えます。筆者が買ったのはラクア mini[*2] という機種で、一人分の食器等を洗うのであれば十分すぎるほどの性能を持っています。食洗機を買ったり設置したりするのが難しい場合は紙皿・紙コップといった使い捨ての容器を使うことでも大幅に片付けの手間を減らすことができます。

3　まとめ

いかがでしたか？ 意外なことなんですが食事は人間の生活において非常に重要な要素です。食事をおろそかにすることで必須栄養素が足らず睡眠の質が悪くなったり集中力が下がったりすることもあります。そのため、上手な手の抜き方を覚えることで栄養と睡眠の質を保ちながら生活のコストを下げることができます。みなさんも良い生活を心がけましょう！

（機会があれば掃除編も出そうかと思います）

[*2] https://www.thanko.jp/view/item/000000003922

編集後記

編集部員たちの記事をまとめてみたら、奇数ページになってしまいました。空白ページを作るのもなんだかな、と思ったので、普段はあまり書かれないのですが、珍しく編集後記を書くことにしました。

とはいえ、あまり述べることもないので、WORD の編集について少し書いてみようと思います。WORD は、例年「引っ越し準備号」「入学祝い号」「研究室紹介号」の 3 つの特別号と、数刊の通常号を発行しています。通常号は、毎回その時々の時事やミームに関連したタイトルをつけることが慣例となっているため、タイトルの決定には苦労します。本号では、「WORD 編集部は 44 年目も営業しています号」と題して、編集部にもファンの多いあの名店が 43 年の歴史に幕を閉じたことにちなみました。実は、WORD は本当に今年で 44 年を迎えています。皆様のご購読に御礼申し上げるとともに、これからも WORD へのご支援をお願いする所存です。

毎号の編集にあたっては、1 号分発行できそうなネタが各部員に集まってきた頃を見計らって編集長が記事募集の呼びかけを行い、同時に GitHub 上にリポジトリを立てます。そこで、各部員が記事を執筆し、Pull Request として提出します。記事が出揃ったら、編集部員の集まる編集会議で記事の校正作業（赤入れ）を行い、修正ののち印刷に入ります。

こういった手順で執筆から印刷、製本まで部員の手で行っていますから、発行には短くても数週の時間がかかります。特に、製本作業は部員が一冊ずつ手作業で行っているので、情報科学類らしからぬ、ちまちまとした作業が続きます。

そんなわけで、いまデータでお読みの方も、ぜひ一度印刷物を手にとって読んでみていただけると嬉しく思います。簡素な B5 紙のホチキスどめですが、WORD の伝統的な形態で、むしろ面白がっていただけるのではと思います。

また、編集部では編集部員を募集しています。こんなちまちまとした作業にみんなで取り組むことに関心のある人は、ぜひ編集部にお越しください。word@coins.tsukuba.ac.jp でいつでもお待ちしております。

さて、もうすぐ夏本番。つくばの森の夏は虫が多いこともあり、虫が得意でない私には少し苦痛な季節ではありますが、夏に出かける予定を考えると気分が上がります。それまでの雨の季節、お体に気をつけて、WORD とともにお過ごしください。

文責　編集長　伊藤祐聖

情報科学類誌

WORD

From College of Information Science

WORD編集部は44年目も営業しています号

発行者　情報科学類長

編集長　伊藤祐聖

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）

2023年7月7日　初版第1刷発行　（256部）