LaTeX をベースとしたウェブサイトについての検討

2019-04-28 Sun

動機

LaTeX をベースとしたウェブサイトについて検討したい。

つまり、イメージとしては、ある URL にアクセスするといくつかの.pdf ファイルへのリンクが並んでいて、あるリンクをクリックすると、その.pdf ファイルが閲覧できる。要するに単に、**.pdf ファイルがあるウェブサイト**である。

また、そのそれぞれの.pdf ファイルを作成する上では、**LaTeX を活用したい**のである。つまり、テキストエディタなどで.tex ファイルを編集し、何らかの LaTeX 処理系を用いてそれを.pdf ファイルにコンパイルし、オンラインのサーバ上にその.pdf ファイルを配置し、.html ファイルにおいてその.pdf ファイルへのリンクを記載すればよい。

よいのだが、.tex ファイルを少し編集するたびに手作業でそれを行うのはあまり現実的ではないので、うまい**自動化が考えられないだろうか**、という論点をこの文書で考える。

そこまでして LaTeX を使うか

という問題がそもそもある。たいていの問題について、**Markdown + MathJax あたり**が美しい解決なのではないか。(.tex ファイルを用いた) LaTeX が有用な場合があるとしても、それを過度に広げることは悪いデザインなのではないか。時流に逆行しているのではないか。

そしてそもそも、個人的には、Markdown や MathJax にすらほとんど用はなく、**plain text でたいていは足りている**。plain text は、メタなコマンドがないので、なんでもコピペできて便利だ。思考のツールとして効率的だ。plain text への信仰を捨てて LaTeX を信仰する合理性はあるのか。

ただ、私が思うのは、悲しいかな、**数学と学術論文**が世界の頂点だということである。そしてそこでは英語が標準語だということである。そして、いわゆる「サピア＝ウォーフの仮説」もまた、残念だが事実だ。数式の記法を用いずして、数学的に考えることはできない。私自身は、数学や学術論文の世界にまで踏み込みたいというまでの欲はない。しかし、世俗的な安寧のない者が、知的向上の道まで閉ざされてしまえば、そこに一種の死を感じるというまでである。私は、考えるという営みを生活の一部として行っていきたい。そして、思考は記法に制約されるが、思考のツールとしての記法として、LaTeX が当面において王だと考えられるのである。

私にとっての LaTeX の優位性は、**単層のレイヤーとしてのその完結性**にある。それ以下でないことと同じだけ、それ以上でないことが重要である。例えば、HTML は素晴らしい言語であり、汎用性がある。CSS や

JavaScript、あるいはサーバサイドの PHP などとも連関して、動的でインタラクティブに情報を表現することができる。しかしそれは、用途によっては行きすぎていないか。.pdf ファイルならば、それを移動するときに、他のファイルへの依存性を考える必要がないし、ウィンドウサイズが変わった際の表現の変化を考える必要もない。ユーザからの入力を取らない、定数関数だ。

例えば、すでに触れた、Markdown + MathJax は強力だろう。あるいは、Jupyter Notebook というものが、多くの用途で人気がある。しかし一つの不安は、それが数百年後も同様に人気か、という点にある。Markdown から HTML へのコンパイラの仕様は変化しないか、MathJax による数式の表示のあり方は変化しないか。一方で、LaTeX は、要素技術の複合体というよりは、一つのまとまりである。よく見れば要素技術の複合体だが、**枯れた部分が多く**、その変化が遅い。

賢い人々は様々なツールをその時々の必要性によって使いこなすのだろうが、私は、考える必要のあることを**最小限に単純化**したい。LaTeX を重視するなら、なんでもかんでも LaTeX で済ませられないか、を考えるのだ。言語の壁の問題などもあり、LaTeX を用いることがどの程度、時代の変化に強いか、大いに不安はあるが、使ってみたいと思うのである。

もとより、LaTeX は**印刷のため**に強力だろう。というより、HTML が不思議と印刷に弱いというべきかもしれない。ウェブブラウザによって結果は大きく異なり、特に、A4 なら A4 でその出力結果をソースファイルから完全にコントロールするようなことは、全くできまい。個人的にさほど印刷が必要かというと、そうでもないのだが、用紙のサイズや文字の大きさが固定されてこそ、細かいレイアウトを考える意味がある、という面はある。

プログラミングのアルゴリズムを考えるにあたっても、コードそのものには結局は表現力がない。仕様全てをコードそのものに表現するというのは一般的に一つの夢だろうと思うが、不可能だろう。人間用のドキュメンテーションと実装との乖離は、将来に渡ってコストになるはずだ。人間用の記法において仕様定義を熟慮してから、やや可読性を軽視した実装をすることが、常に最も強力だろうと考えられる。結局常に、人間の思考活動においては、純粋に**人間用の記法が中心となる**のだ。その一つの理由は、一つの具体について複数の抽象が有用でありうることにあるかもしれない。

GitHub を用いるべきか

GitHub を用いてはどうだろうか、と思った。オフラインなローカルとオンラインなリモートを同期させる方法としては、奇妙な依存性がなく、とても筋がいい。

しかし一つ思い浮かぶ問題は、Git はテキストデータのためのものであって、.pdf は基本的にバイナリだろうということである。よって、.tex に微細な変更があった場合の.pdf の更新について、効率的に差分を表示することはできず、素朴に git を用いたならば、望まないデータ容量の消費が起こるのではないか。

それについては、何らかの力づくな方法で、Git に格納されている過去のデータを破壊して削除していけばよいのではないか。それが可能かは未調査である。おそらくできるのではないか。しかし、Git や GitHub をこのように使うことは基本的に、とても筋が悪い。

そうして、Git リポジトリの docs ディレクトリを GitHub Pages という機能でウェブサイトとして公開すれば、目的を果たせるのではないか。

なお、.tex ファイルが更新された際には、プログラムによって.pdf ファイルとのバージョンの違いを検出し、.tex ファイルをコンパイルして、.pdf ファイルを適切に配置する。そのようなプログラムの開発を検討しようというのが、この文書の趣旨である。実装には、Python 3 言語を使うつもりだ。

なお、GitHub のリポジトリの容量は 1 GB であり、1 ファイルの容量は 50 MB 未満であるべきであるらしい。

ディレクトリ構成

Git リポジトリのルートに、docs ディレクトリと tex ディレクトリを置く。docs ディレクトリはウェブサイトとして公開するためのディレクトリである。tex ディレクトリは、.tex ファイルらを配置するためのディレクトリである。

.tex ファイルは、コンパイルの際に中間ファイルを生成するなどすることが多い。よって、1 つの.tex ファイルにはそれ専用のディレクトリを与えることが実用的だろう。よって、1 つの.tex ファイルが 1 つのディレクトリに入っていることを前提に考える。.tex ファイルのファイル名は原則として tex.tex とするが、任意であっていい。しかし、ウェブサイトに配置される.pdf ファイルについては、.tex ファイルを含むディレクトリの名称に、拡張子.pdf をつけたものとする。それが実用上便利だろうと思うからである。

.tex ファイルのあるディレクトリに、実用上、出力されるであろう.pdf ファイルおよび中間生成物は、.gitignore を利用して可能な限り無視することを原則とする。

直下に 1 つ以上の.tex ファイルを含むディレクトリを、「leaf ディレクトリ」と呼ぶことにする。leaf ディレクトリではないディレクトリを「branch ディレクトリ」と呼ぶことにする。graph theory における leaf node と branch node という呼称を援用してのことである。tex ディレクトリにおいて、leaf ディレクトリは、branch ディレクトリによって階層的に配置されていていい。branch ディレクトリの階層構造は、docs ディレクトリの内部において反映される。

中心的な役割を果たすプログラムを、compile_to_website.py と呼ぶことにする。compile_to_website.py は、tex ディレクトリの直下に配置される。compile_to_website.py の実行と実行時を、単に実行や実行時と呼ぶことにする。

実行時の処理

実行されると、プログラムはまず、tex ディレクトリを再帰的に走査する。そして、leaf ディレクトリらのパスを得る。

次に、docs ディレクトリを走査する。これにより、docs ディレクトリの現状を知る。

compile_to_website.py は、同じディレクトリにある log.txt ファイルを (もしあれば) 参照する。log.txt ファイルは、.tex の更新差分を考えるための資料である。

もしも log.txt ファイルが存在しなければ、docs ディレクトリの中のファイルを全て削除して、全ての.tex ファイルをコンパイルし、docs ディレクトリの中をゼロから構築しなおす。

対応する leaf ディレクトリが存在しない.pdf ファイルは削除する。

存在する.pdf ファイルのソースファイルである.tex ファイルのハッシュが、log.txt に記載されているはずである。記載されており、なおかつハッシュが変わっていないときのみ、当該.pdf ファイルを、「更新不要 (uptodate)」と判断する。更新不要である leaf ディレクトリ以外について更新 (update) する。

update するとは、.tex ファイルをコンパイルし、.pdf ファイルを docs ディレクトリの下位に配置することである。(とりあえずの仕様として、) leaf ディレクトリ内にすでに.pdf ファイルが存在するときにも改めてコンパイルする。

更新処理が終わったならば、現在の tex ディレクトリと docs ディレクトリの状況から、log.txt ファイルの内容をゼロから作り直す。(同じファイルのハッシュを 2 回計算することによる計算量の増加は、許容していいと見込んでいる。)　その内容は、.pdf のパスから.tex のハッシュへの連想配列である。

log.txt ファイルの更新処理が終わったならば、.html ファイルらを作成する。

Git データの初期化について

上で触れたように、.pdf ファイルはバイナリファイルであるから、Git に適さない。頻繁に更新すると無駄に容量が増加すると思われる。これを避けるためには、.pdf について、最新のデータ以外は削除したい。

しかしながら、それを行うごく簡単な方法は見当たらなかった。しかし、Git リポジトリ全体を初期化してしまう方法は、シンプルで確実そうなものが紹介されていた。よって、Git リポジトリ全体を初期化する方法で、下の Python コードとして実装した。

この方法の短所の一つとして、Git によって本来は差分が記録されてよいはずの.tex ファイルなど、一切のファイルについて、差分やバックアップのようなものは一切得られないということである。データを喪失するリスクがあるだろう。

参照されているファイルについて

.tex ファイルの更新のみチェックするという考えでいたが、\verbatiminput などとして参照しているファイルが更新されていても無視してしまうのは問題な気がしてきた。未解決。input 対象は必ず input ディレクトリ以下にあるという前提にして、処理すればよいだろうか。

compile_to_website.py

```
#!/usr/bin/python3
# tex ディレクトリ以下にある.tex ファイルらをコンパイルして docs ディレクトリ以下に構成する。
import hashlib
import json
import os
import shutil
import subprocess
import sys

ROOT = os.path.abspath("../../..")  # tex および docs ディレクトリの親ディレクトリ
os.chdir(ROOT)                      # カレントディレクトリを変更する


# log.txt ファイルは、.pdf のパスから.tex ディレクトリのハッシュへの辞書である。
# すでに最新の.tex ディレクトリを改めてコンパイルしないためのキャッシュである。
def write_log(d):
```

```
    """辞書 d を JSON 形式でファイル tex/log.txt に書き込む。"""
    path = os.path.join("tex", "log.txt")  # e.g. tex/log.txt
    s = json.dumps(d)                      # JSON 形式の文字列にする
    with open(path, "w") as f:             # ファイル log.txt を開く
        f.write(s)                         # 書き込む


if "--clear" in sys.argv:  # コマンドライン引数に--clear があればキャッシュを初期化する
    write_log({})


def f():
    """ファイル tex/log.tex を読み込んで辞書として返す。"""
    path = os.path.join("tex", "log.txt")
    if not os.path.isfile(path):           # log.txt がなければ作る
        print(path + "がありません。作成します。")
        write_log({})
    with open(path) as f:
        s = f.read()
    return json.loads(s)


log = f()

# .tex ファイルのあるディレクトリを葉ディレクトリとして集める。
tex_dirs = []
for root, dirs, files in os.walk("tex"):
    for file in files:
        _, ext = os.path.splitext(file)
        if ext == ".tex":
            tex_dirs.append(root)
            break
print(".tex ファイルのあるディレクトリは以下の{}個です。{}".format(len(tex_dirs), tex_dirs))

# 既存の.pdf ファイルらのパスを集める。
pdfs = set()
count = 0
for root, dirs, files in os.walk("docs"):
    for file in files:
        _, ext = os.path.splitext(file)
```

```
        path = os.path.join(root, file)
        if ext == ".pdf":
            pdfs.add(path)
        if ext == ".html":
            os.remove(path)
            count += 1
print("{}個の HTML ファイルを削除しました。".format(count))
print("既存の.pdf ファイルは以下の{}個です。{}".format(len(pdfs), pdfs))


def call(xs):
    """xs の内容を print() してから、xs を subprocess.call() する。"""
    print("    #####BEGIN#####", " ".join(xs))
    subprocess.call(xs)
    print("    #####END#####", xs[0])


def compile(path):
    """.tex ファイル path をコンパイルする。"""
    dirname = os.path.dirname(path)  # .pdf ファイルや中間ファイルを生成するディレクトリ
    # call(["ptex2pdf", "-l", "-u", "-ot", '"--shell-escape -synctex=1"',
    #       path, "-output-directory", dirname])
    os.chdir(dirname)
    basename = os.path.basename(path)
    call(["ptex2pdf", "-l", "-u", "-ot",
          '"--shell-escape -synctex=1"', basename])
    os.chdir(ROOT)


def get_named(path_pdf):
    """コンパイル直後の.pdf ファイル path_pdf に対応する docs 以下での名前を返す。"""
    path_pdf = os.path.normpath(path_pdf)
    xs = path_pdf.split(os.sep)
    xs = ["docs"] + xs[1:-2] + [xs[-2] + ".pdf"]
    path_named = os.path.join(*xs)
    return path_named


def get_hash(path):
    """ファイル path の SHA1 を返す。"""
```

```
    with open(path, "rb") as f:
        bytes = f.read()
    return hashlib.sha1(bytes).hexdigest()


def get_hash_tex(path_tex):
    """.tex ファイル path_tex およびその参照ディレクトリについてハッシュを返す。"""
    hash_tex = get_hash(path_tex)
    dirname = os.path.dirname(path_tex)
    dir_input = os.path.join(dirname, "input")
    if os.path.isdir(dir_input):  # 参照ディレクトリがあるか
        hashes = []               # ファイルの SHA1 を集める
        for root, dirs, files in os.walk(dir_input):
            for file in files:    # それぞれのファイルについて
                path = os.path.join(root, file)
                hashes.append(get_hash(path))
        hash_tex = "-".join([hash_tex] + hashes)  # 安易に連結する
    return hash_tex


n_update = 0
n_uptodate = 0
updated = []
log_new = {}
pdfs_new = set()
for tex_dir in tex_dirs:
    listdir = os.listdir(tex_dir)  # 葉ディレクトリの中のファイルら
    for relative in listdir:
        path = os.path.join(tex_dir, relative)
        if os.path.isdir(path):  # ディレクトリは無視する
            continue
        root, ext = os.path.splitext(path)
        if ext == ".tex":  # 必ずある
            path_tex = path
            path_pdf = root + ".pdf"     # コンパイルで作られる.pdf の名前
            path_named = get_named(path_pdf)
            hash_tex = get_hash_tex(path_tex)
            break  # 最初に見つかった.tex ファイルのみ処理する
    if path_named in log:                # 以前にコンパイルされたことがある
        if log[path_named] == hash_tex:  # 前回のコンパイルから更新がない
```

```
            uptodate = True               # すでに最新だ
        else:
            uptodate = False
    else:
        uptodate = False
    if uptodate:
        n_uptodate += 1
    else:
        compile(path_tex)
        dirname = os.path.dirname(path_named)
        os.makedirs(dirname, exist_ok=True)
        shutil.copyfile(path_pdf, path_named)  # docs ディレクトリの下にコピーする
        n_update += 1
        updated.append(os.path.basename(path_named))
    log_new[path_named] = hash_tex  # .pdf に対応する.tex の SHA1 を記録する
    pdfs_new.add(path_named)        # 対応する.tex ファイルがある.pdf ファイルを記録する
write_log(log_new)
print(".tex ファイル{}個中{}個について.pdf ファイルを更新しました。{}"
      .format(n_update+n_uptodate, n_update, updated))
# print(".tex ファイルに対応する.pdf ファイルは以下の{}個です。{}"
#      .format(len(pdfs_new), pdfs_new));
pdfs_delete = pdfs - pdfs_new
print("削除された.pdf ファイルは以下の{}個です。{}"
      .format(len(pdfs_delete), pdfs_delete))

for pdf in pdfs_delete:
    print("PDF ファイルを削除します。" + pdf)
    os.remove(pdf)

for root, dirs, files in os.walk("docs", topdown=False):
    dirs = [dir for dir in dirs if os.path.isdir(os.path.join(root, dir))]
    is_leaf = dirs == []
    if is_leaf:
        has_pdf = False
        for file in files:
            _, ext = os.path.splitext(file)
            if ext == ".pdf":
                has_pdf = True
        if not has_pdf:
            for file in files:
```

```
                print("ファイルを削除します。", os.path.join(root, file))
                os.remove(os.path.join(root, file))
            print("ディレクトリを削除します。", root)
            os.rmdir(root)

# HTML ファイルらを作成する。
count = 0
for root, dirs, files in os.walk("docs"):
    dirs, files = sorted(dirs), sorted(files)
    path = os.path.join(root, "index.html")
    s = """<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>{}</h1>
    <ul>
""".format(root)
    for dir in dirs:
        s += ('<li><a href="{}/index.html">{}/index.html</a></li>\n'
              .format(dir, dir))
    for file in files:
        s += '<li><a href="{}">{}</a></li>\n'.format(file, file)
    s += """    </ul>
  </body>
</html>"""
    with open(path, "w") as f:
        f.write(s)
    count += 1
print("{}個の HTML ファイルを作成しました。".format(count))
```

push_after_delete_everything.py

```
#!/usr/bin/python3
# GitHub リポジトリの履歴情報を全て削除して完全に初期化する。
import os
import re
import subprocess
```

```
test_url = "https://github.com/example/example.git"
test_email = "you@example.com"
test_name = "Your Name"

ROOT = os.path.abspath("../../..")
os.chdir(ROOT)


def f():
    """設定ファイル.git/config から項目 url、email、name の値を抜き出して返す。"""
    path = ".git"
    b = os.path.isdir(path)
    if not b:
        print("エラーです。ディレクトリ「" + path + "」がありません。終了します。")
        exit()

    path = os.path.join(".git", "config")
    b = os.path.isfile(path)
    if not b:
        print("エラーです。ファイル「" + path + "」がありません。終了します。")
        exit()
    with open(path) as f:
        s = f.read()

    m = re.search("url = (https://github.com/.*\.git)", s)
    if not m:
        print("エラーです。.git/config ファイルに url がありません。終了します。")
        exit()
    url = m.group(1)
    print("url:", url)

    m = re.search("email = (.+)", s)
    if not m:
        print("エラーです。.git/config ファイルに email がありません。終了します。")
        exit()
    email = m.group(1)
    print("email:", email)

    m = re.search("name = (.+)", s)
```

```
    if not m:
        print("エラーです。.git/config ファイルに name がありません。終了します。")
        exit()
    name = m.group(1)
    print("name:", name)
    return url, email, name


url, email, name = f()


# https://stackoverflow.com/questions/9683279/
#     make-the-current-commit-the-only-initial-commit-in-a-git-repository
def f(xs):
    """xs の内容を print() してから、xs を subprocess.call() する。"""
    print("#", " ".join(xs))
    subprocess.call(xs)

f(["git", "pull"])
f(["du", "-sh"])
f(["rm", "-rf", ".git"])
f(["du", "-sh"])
f(["git", "init"])
f(["git", "add", "."])
f(["git", "config", "user.email", email])
f(["git", "config", "user.name", name])
f(["git", "commit", "-m", "Initial commit"])
f(["git", "remote", "add", "origin", url])
f(["git", "push", "-u", "--force", "origin", "master"])
```

modify_preamble.py

```
#!/usr/bin/python3
# .tex ファイルの通称プリアンブルについて自動処理する。

import os
import re

ROOT = os.path.abspath("../../..")  # tex および docs ディレクトリの親ディレクトリ
```

```
os.chdir(ROOT)


templates = {}
templates["1"] = r"""
\documentclass[uplatex,dvipdfmx]{jsarticle} \usepackage{amsmath,amssymb,bm}
\usepackage{verbatim} \usepackage{svg} \usepackage{graphicx}
\usepackage[dvipdfmx]{hyperref} \usepackage{pxjahyper}
\usepackage{paracol} \usepackage{threeparttable}
\usepackage{seqsplit}"""[1:]
pattern = r"%TEMPLATE-BEGIN (.*?)\n(.*?)\n%TEMPLATE-END"
compiled = re.compile(pattern, re.MULTILINE | re.DOTALL)
sub = r"%TEMPLATE-BEGIN {}\n{}\n%TEMPLATE-END"


def is_tex(path):
    """ファイル path が実在して（拡張子が）.tex ファイルなら真を返す。"""
    isfile = os.path.isfile(path)       # 存在するか
    root, ext = os.path.splitext(path)  # 拡張子を得る
    return isfile and ext == ".tex"


def has_template(path):
    """.tex ファイル path にテンプレートの記載がなければ偽を返す。
    もしあればテンプレート名、内容、マッチ全体を返す。"""
    with open(path) as f:
        s = f.read()
    match = compiled.search(s)  # テンプレートの記載を探す
    if match is None:           # テンプレートの記載がない
        return False
    name = match.group(1)       # テンプレート名
    text = match.group(2)       # テンプレートの既存の内容
    return name, text, match.group()


def is_uptodate(text, name):
    """テンプレート内容 text がテンプレート名 name のものとして最新なら真を返す。"""
    return templates[name] == text
```

```
def update_template(path, name):
    """.tex ファイル path をテンプレート名 name のテンプレートに更新する。"""
    with open(path) as f:
        src = f.read()
    text = templates[name]
    new = sub.format(name, text)
    dst = re.sub(compiled, new, src)
    with open(path, "w") as f:
        f.write(dst)
    inc = len(dst) - len(src)  # increase.
    print("{}のテンプレートを更新しました。文字数の差は{}です。".format(path, inc))


def do_file(path):
    """ファイル path について必要ならテンプレートを更新する。"""
    if not is_tex(path):         # .tex ファイルではない
        return False
    response = has_template(path)
    if response is False:        # テンプレート部分がない
        return False
    name, text, match = response
    if is_uptodate(text, name):  # テンプレート内容がすでに最新だ
        return "uptodate"
    update_template(path, name)  # ファイルデータを更新する
    return "updated"


def get_paths_tex():
    """存在する.tex ファイルのパスのリストを返す。"""
    paths = []
    for root, dirs, files in os.walk("tex"):  # 各ディレクトリについて
        for file in files:                    # ディレクトリ直下の各ファイルについて
            _, ext = os.path.splitext(file)
            if ext == ".tex":                 # 拡張子が.tex である
                path = os.path.join(root, file)
                paths.append(path)
    return paths


def main():
```

```
    """存在する.tex ファイルらのテンプレートを更新する。"""
    paths = get_paths_tex()
    print("{}個の.tex ファイルがあります。".format(len(paths)))
    n_update = 0
    n_uptodate = 0
    for path in paths:
        response = do_file(path)
        if response == "updated":
            n_update += 1
        elif response == "uptodate":
            n_uptodate += 1
    print("{}個のファイルはすでに最新です。".format(n_uptodate))
    print("{}個のファイルを更新しました。".format(n_update))


if __name__ == "__main__":
    main()
```