portable dumper: アーキテクチャに依存しない Emacs の起動時間短縮手法

永野 圭一郎, 林 芳樹

要旨

本稿では、我々が GNU Emacs を対象に開発した portable dumper を解説する。これは、既存手法 (unexec) と同程度に Emacs の起動を高速化しながら、unexec の問題点である移植性の低さを改善したものである。起動の高速化は、既存手法と同様に、起動に必須の Lisp ファイル群を、あらかじめ解釈済みの状態で保存しておくことによって実現する。我々はその際に、データの保存および復元において、プラットホーム依存の知識を避けることによって高移植性を実現した。この portable dumper が既存手法と同程度に Emacs の起動を高速化すること、そして GNU/Linux および Windows を含む多数の platform で同一のコードが動作すること、以上 2 点を実験により確かめた。

1 はじめに

我々が設計・実装した portable dumper とは、大抵の C 処理系でサポートされているようなライブラリ/システムコール呼び出しと、プラットホームに依存しない知識のみを用いて、Lisp ファイル群を、解釈済みの状態で保存し、高速に再読み込みする手法である。[*1]

従来同じ目的には、 unexec という手法が用いられてきた。これは「動作中のプロセスのメモリ内容から直接、実行ファイルを生成する」という、極めて強くプラットホームに依存するものであった。我々は GNU Emacs[1] の Microsoft Windows への移植のひとつである Meadow[2] の開発にたずさわっており、この部分の Windows での実装が完全ではないことに不満を持ち、抜本的な改善策を必要とする事情があった。

portable dumper は、既に XEmacs に同じ目的の機構が存在する。我々の貢献は、XEmacs と同等以上に多くのユーザに使われている GNU Emacs に対して設計・実装を与えたことである。本論文では、XEmacs の実装と我々のそれとの比較も議論する。

本論文では portable dumper の実装の解説、実験による評価、および別のソフトウェアに採用された類似手法について述べる。

2 背景

本節では Emacs のビルド過程にある loadup , dump という処理に関する解説を行う。その上で、既存手法である unexec の解説をする。

2.1 loadup と dump

テキストエディタ Emacs は、Lisp インタプリタ部分のみが C で記述されている。ファイル操作や多言語関連をはじめとする、ユーザが直接触れる編集の基本コマンドの多くが、C よりも扱いやすい独自の言語 Emacs Lisp で記述されている。

テキストエディタとしての通常の操作に必須な基本プログラム群を、Lisp ファイルから読み込み解釈する処理を、読み込み内容を指定するファイルの名前にならって以下 loadup と呼称する。この基本 Lisp ファイルの数は 80 にも及ぶ[*2]。

この基本ファイルを全て、Emacs 起動時に毎回読み込み解釈し直すのは時間的に高価である[*3]。そのため、loadup が済んだ直後の Emacs の状態を保存し、次回からの起動を高速化する機構が、Emacs には備わっている。この「Lisp ファイル解釈後の状態を保存する処理」を以下 dump と呼ぶこととする[*4]。

dump は Emacs のビルドの際に、Makefile に従って自動的に行われる。我々が普段使用している実行ファ

[*1] 誤解を受けやすいところであるが、我々の意図は、解釈済み Lisp ファイルの保存と復元を行うプログラムを可搬にするということであって、保存された Lisp ファイル (dump ファイル) の可搬性は考慮していない。すなわち我々は、プラットホーム A と別のプラットホーム B とで同じコードによって保存と復元ができるということを目指しているのであって、プラットホーム A で作成した dump ファイルをそのまま別のプラットホーム B で使用できる、というレヴェルの可搬性を、特に目的とはしていない。

[*2] Emacs 21.2 の場合

[*3] 詳細は評価の節で実験により検証している。

[*4] dump を行う Lisp 関数名はdump-emacs である。

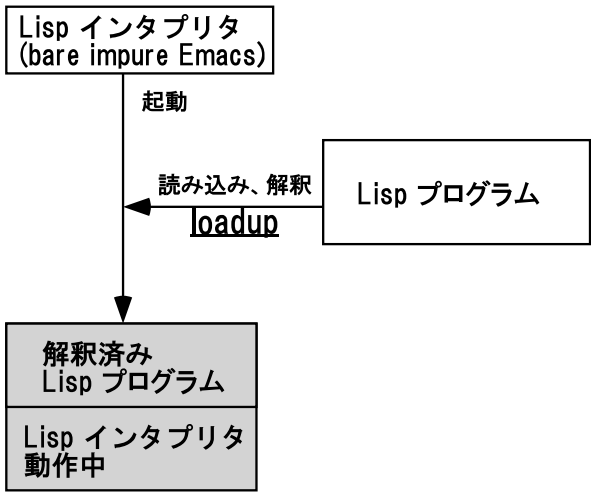

図 1: loadup と dump の関係

イルemacs は、この dump が済んだ状態のものである。loadup 以前の、C で記述された部分のみの状態の Emacs は temacs という実行ファイルであり、これには「bare impure Emacs[*5]」という名が付いている。

loadup と dump の関係を図 1 に示す。

我々が提案する portable dumper は、dump 手法のひとつである。

2.2 unexec

我々の提案以前の既存の dump 手法を、その関数名にならい unexec と呼ぶ。

unexec とは、「プロセスのメモリ内容から、直接実行ファイルを作成する」という手法である。具体的には、Lisp インタプリタの実行ファイルの中に、 loadup 直後のメモリの内容をデータセクションとして追加した、新たな実行ファイルを作成する。これによって、次回実行時からは、再解釈の手間なく loadup 直後の状態を即座に復元できる。これを図 2 に示す。

この手法は、プロセスのメモリの使用法へあまり関係なく適用できるという利点を持つものの、以下のような問題を抱えている。

- メモリの内容から直接実行ファイルを生成するコードは、極めて強くアーキテクチャに依存する。特にバイナリ実行ファイルの書式に強く依存し、可搬性がない。開発者は、新しい CPU・OS への移植の際に、 unexec 用のコードを書き足さなければならない。
- そもそも、このような操作を公式にサポートしていないプラットホームが多い。いつ不可能になっても不思議はない手法だと言える。

[*5] 裸の Emacs

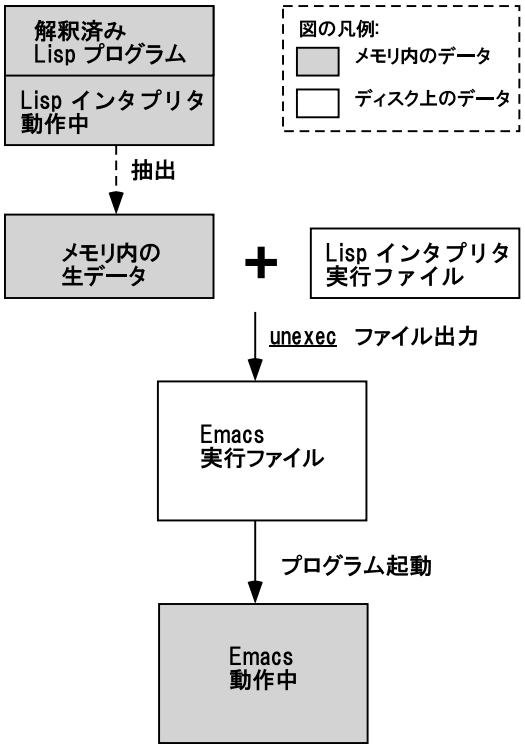

図 2: unexec

3 実装

本節では portable dumper の実装について解説する。

我々が設計・実装した portable dumper とは、fwrite やread, mmap といった可搬性の高い関数のみを用いて、loadup した Lisp ファイル群を、解釈済みの状態で特定のファイルに出力し、次回起動時には dump ファイルの読み込みのみで再解釈の手間なく loadup 直後の状態を復元する手法である。概要を図 3 に示す。

unexec では、プロセスのメモリ内容を単純に書き出せば事足りた。移植性を重視しながら可能な限り処理を簡略化するために、我々は、Emacs の Lisp プログラム解釈後の状態を、データとして見た際の特徴に注目する。すなわち、Lisp プログラムに限っては、ガベージコレクションのマーク段階と同様に、Lisp のデータから別のデータへの参照を次々と全てたどることによって、十分

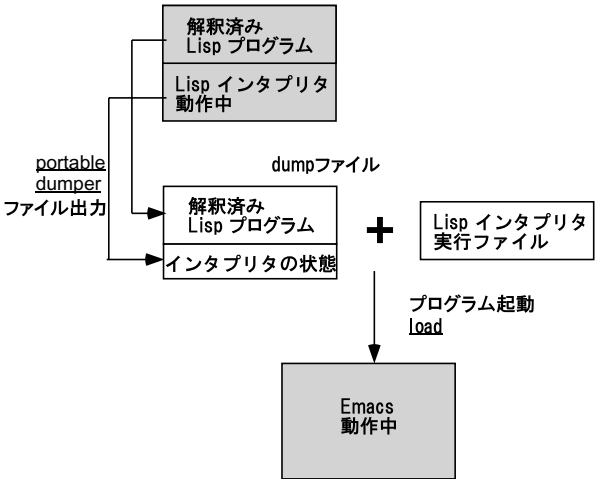

図 3: portable dumper

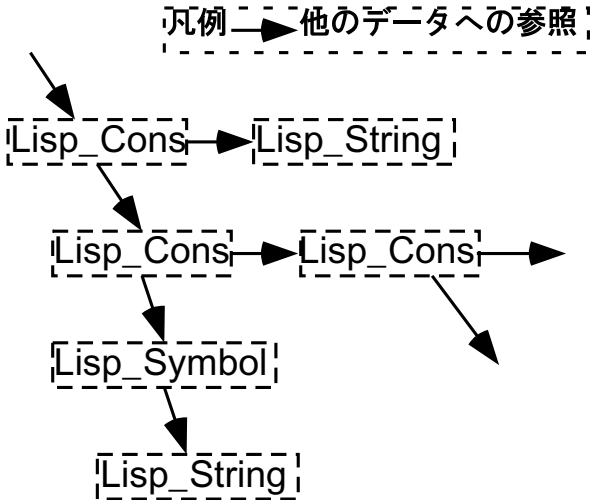

図 4: メモリ上での Lisp プログラムの模式図

条件を満たすデータに容易に到達することができる。

よって我々は、保存すべきデータを「Lisp プログラム」と「Lisp インタプリタの状態」とに分離して扱う。Lisp プログラムとは、 loadup 時に解釈したプログラムのことである。Lisp インタプリタの状態とは、Lisp プログラムの解釈時に更新されるインタプリタの内部状態を指し、C のグローバル変数に相当する。以下のこの順に解説を進める。

3.1 Lisp プログラムの保存と復元

3.1.1 保存するデータの同定

Lisp プログラムの保存における我々の注目は、前述の通り、ガベージコレクションのマーク段階と同様に、Lisp のデータから別のデータへの参照を次々と全てたどることによって、十分なデータに到達することができるというものである。

インタプリタにより解釈された Lisp 式は、プロセスのヒープ内部で、文字列なら struct Lisp_String 、cons セルならば struct Lisp_Cons といった、データ型ごとの構造体の形で保持されている。

Emacs の Lisp インタプリタは、マークアンドスイープのガベージコレクション機構を備えている。これは、(1) ある決まったルートセットから他のデータへの参照を次々と辿ることで、今後使用される可能性がある全てのデータに印を付けことができ (マーク)、(2) データをたどり終わった時点で印の付いていないデータは、今後も使用される可能性がないため、占有しているメモリ領域を解放することができる (スイープ)、というものである。メモリ上の Lisp プログラムが参照によって結び付けられているさまを図 4 に示す。

我々はこのマーク段階と同等の走査によって、loadup 直後の状態を復元するために十分な Lisp プログラムデータ全てを同定することができる。ただし、ここでは Lisp プログラムデータに直接印を付けるのではなく、到達できたデータとそのサイズを配列に記録することとし、また 2 重到達の検出にはハッシュ表を用いることとした。このハッシュ表にはデータ型ごとに参照とサイズを記録してあり、後述のデータ再配置の際のアドレス再計算にも用いる。

3.1.2 ファイルへの保存

以上の手順で把握した十分なデータを、最小のファイルアクセスで、またロード後には可能な限り無変更で、メモリ空間に復元できるように、調整した形でファイルへ出力する。我々の意図は、復元を、1 度の mmap で dump ファイルを直接メモリ空間に配置することによって行うものである。

復元を高速に完了するために、 dump ファイルのサイズは可能な限り小さいことが望ましい。そのためここでは、必ずしもメモリ上に連続して確保されてはいるとは限らないデータを、保存時に一列に再配置 (リロケーション) する。その際、データの位置するメモリアドレスが変化するため、データ中にある他のデータへの参照を更新しなければならない。(図 5)

我々はファイルへの保存をデータ型ごとに行うことにした。保存対象のデータおよびそのサイズは前段階で全て配列に記録されている。そのためデータ領域開始アド

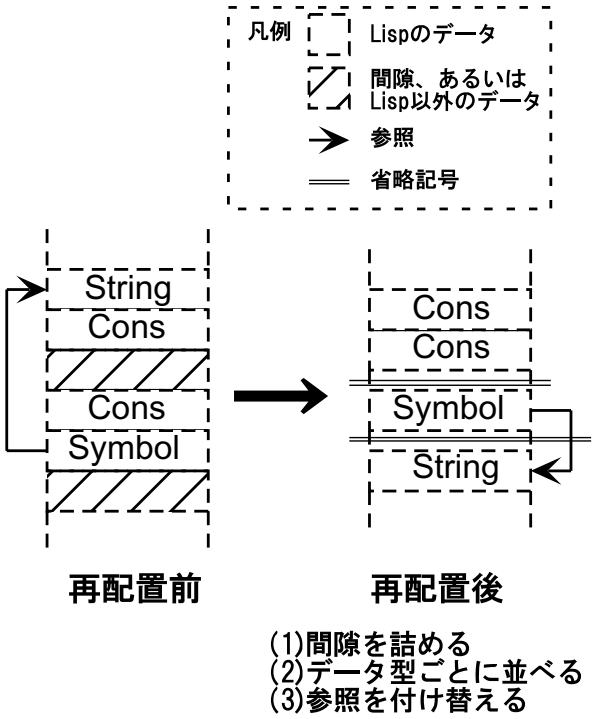

図 5: 再配置

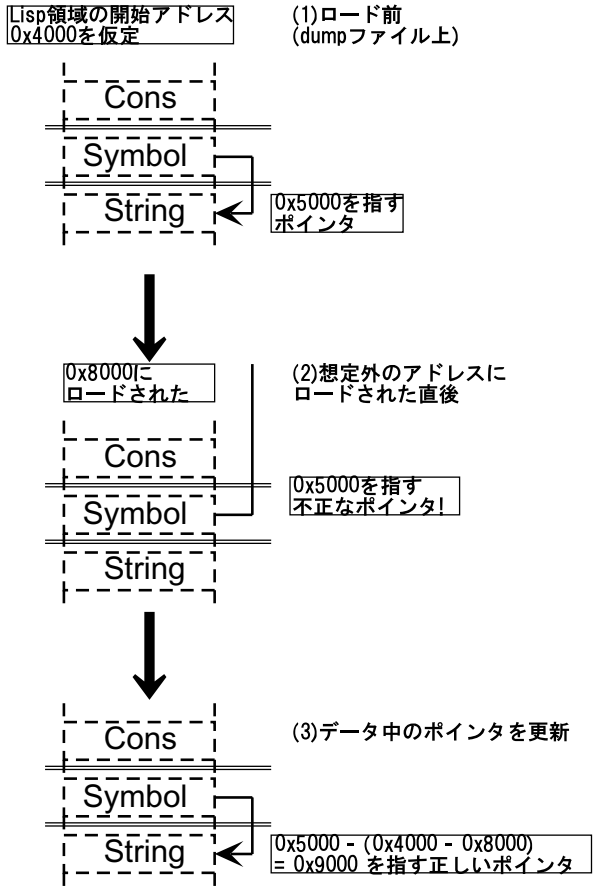

図 6: 想定開始アドレス以外に読み込まれた場合の参照の更新

レスを適当に与えることによって、再配置後のデータアドレスを、実際に保存を行う前に計算することができる。これでデータ保存時の、参照の更新が可能である。

3.1.3 復元

上述の手法により、復元は、理想的には mmap により dump ファイルの内容を直接メモリ空間に配置することで完了する。

ただし、想定したアドレスにファイルをマップできなかった場合、および mmap の実行が不可能なプラットホームにおいて malloc によるメモリ確保で代用した場合、保存時に想定していたメモリ領域開始アドレスとのずれが生じ、データ中の参照が不正となる。この場合は全データを走査し、想定アドレスと実際の割り当てアドレスとの差異でポインタを更新する必要がある。(図 6)

3.2 インタプリタの状態の保存と復元

Lisp プログラム解釈済みの状態を復元するには、loadup 中に更新されたインタプリタの内部状態を保存しておく必要がある。これはコードのレヴェルでは、C のグローバル変数を保存、復元することに相当する。

この部分は現在は単純に、保存する必要がある変数をひとつひとつ人手で同定し、保存・復元のためのコードを書くという実装にしている。

4 評価

以上の解説に従って実装した portable dumper の、実験による評価を行う。この手法における我々の意図は、(1) 起動時間の短縮および (2) コードの可搬性の向上である。よって、この 2 点について以下順に評価を進める。

4.1 起動時間の評価

portable dumper によって dump されたファイルから Emacs を起動するのに必要な時間を、 unexec を用いた場合、および高速化手法を用いない場合と比較するのが表 1 である。実験は Pentium III 600MHz および 1GB のメモリを搭載したマシンで、Debian GNU/Linux[*6]上で行った。手順は、まずそれぞれの条件で Emacs をビルドし、emacs -batch -q -f kill-emacs[*7]を数回実行して使用されるファイルをディスクキャッシュに入れたのち、同じ

[*6] Linux kernel 2.4.15-pre6, GCC 2.95.4 (Debian prerelease)

[*7] Emacs を起動し、必要な初期化を完了した時点で即座に終了させるコマンドライン

高速化 (dump) 手法	起動に要した時間
高速化なし	1.43sec
unexec	0.0639sec
portable dumper (1)	0.0950sec
portable dumper (2)	0.113sec

表 1: 起動時間計測実験 結果

portable dumper に関して、 dump ファイルからの復元が mmap のみで完了する (参照の更新が必要ない) 条件を (1)、mmap 後に Lisp プログラム中の参照の更新が必要な条件での計測を (2) としている。

コマンドラインを連続 100 回実行して合計時間の平均を取った。

我々の考えでは、実験結果は以下の事項を意味する。

- **高速化手法は依然、必要である。** 実験環境は計算機として決して低速ではないスペックであるにもかかわらず、高速化手法を用いない場合には、対話的アプリケーションとして問題が感じられるほどに起動に時間を要している。
- **unexec には、portable dumper に対する速度的優位はない。** portable dumper は unexec と比較して、起動時に dump ファイルを読む I/O のコストが存在する分、数字の上では低速となっている。だがその差は高々 0.05 秒であり、問題とするにはあたらない。

4.2 可搬性の評価

我々は、以下のプラットホーム全てで、同一の portable dumper コードが動作することを確認した。我々の意図通りに、OS やプロセッサの違いを越えて動作していることが見て取れる。

- GNU/Linux (kernel 2.2/2.4)
- FreeBSD (RELEASE-4.5)
- Solaris (2.6/2.7/2.8 on SPARC, 2.8 on Intel)
- Microsoft Windows (Windows 98, Windows 2000)[*8]

[*8] ただし 2002 年 8 月現在、Windows の実装は不完全である。Microsoft Windows でファイルをメモリマップするには、mmap ではなく独自の API (MapViewOfFileEx) を用いる必要があるが、そのためのコードは準備中である。この実験では、mmap が使えない場合のコード、すなわち malloc でメモリ領域を確保し、そこに dump ファイルの内容を読み込む手法で動作させている。

5 他のソフトウェアにおける類似手法

本節では、他のソフトウェアに採用された、我々の portable dumper と同様のアプローチを紹介する。

6 XEmacs

XEmacs[3] の portable dumper 実装 [4] は Olivier Galibert 氏により作成された。その概略は我々のそれとほぼ同様、 loadup 後に内部データをファイルに出力し、次回起動時からはそのファイルを用いることで loadup 直後の状態を即座に復元する、というものである。

Emacs と XEmacs の、起動高速化手法の観点からの最大の差異は、XEmacs の方が内部データ型の種類が多いことである。XEmacs の Lisp エンジンは Emacs のそれと比較して多数の型を持ち、また新たな C の構造体を追加することにも抵抗がない。portable dumper は unexec と異なり、ソフトウェアの内部構造に依存してしまう。よって扱うべきデータの種類が多いことは直接、portable dumper の実装を複雑にすることに繋がる。

この問題に対する彼らのアプローチは、個々のデータ型について**description** というアドヴァイスを定義することである。この description には、その型のデータが置かれたメモリ領域に関する仕様が書かれている。図 7 に、Lisp の文字列型の description を例として挙げる。この仕様を見ることで、portable dumper が必要とする、データのサイズや他のデータへの参照の位置を得ることができる。

XEmacs の実装では、このような手法によって、データ型の複雑さを portable dumper エンジンから分離することに成功している。

7 TEX

組版ソフトウェア LATEX[5] は、ベースとなる TEX というプログラムに、plain および LATEX というマクロパッケージを加えたものである。TEX プログラムはビルド時にコンパイルされてバイナリの実行ファイルとなる。マクロパッケージは利用者が記述するマクロと全く同じ物で、大量のテキストファイルである。

プログラム起動の度に全マクロを解釈し直すのは高価であるため、TEX は Emacs と同様、解釈済みのマクロを保存しておき、次回からの起動を高速化する機構を備えている [6]。

```
struct lrecord_description string_description[] = {
  { XD_BYTECOUNT,         offsetof (Lisp_String, size) },   ← sizeというメンバにBytecount (メモリのサイズ)が入っている
  { XD_OPAQUE_DATA_PTR,   offsetof (Lisp_String, data),     ← dataというメンバに、特に構造をもたないデータへのポインタがある
                          XD_INDIRECT(0, 1) },                そのサイズはこのdescriptionの0番目に挙げたメンバ (size) の値+1である
  { XD_LISP_OBJECT,       offsetof (Lisp_String, plist) },  ← plistというメンバに他のLispのデータへの参照が入っている
  { XD_END }
};
```

図 7: XEmacs における Lisp の文字列型の description

TEX システムのビルドで生成されるバイナリ実行ファイルには、`initex` と `virtex` の 2 種類がある。 `initex` は、TEX のビルド時に、必要なマクロパッケージを全て解釈して、そのメモリイメージをフォーマットファイル[*9]という形で保存する。現在利用者が扱う TEX プログラムは `virtex` のほうであり、これは `initex` により作成されたフォーマットファイルをロードすることで、マクロを即座に復元できる。このフォーマットファイルは、portable dumper における `dump` ファイルに相当する。

現在は以上のような利用形態が普通である。しかし以前は、 `preloaded` という実行ファイルが用いられていた時期もあった。 `preloaded` とは、フォーマットファイルの内容を含んでいる実行ファイルであり、そのためプログラム起動時に、フォーマットファイル読み込みのコストがかからないというのが利点である。

`preloaded` の作成手順は、 `virtex` にフォーマットファイルをロードさせた直後にコアダンプさせ、core ファイルを `undump` というプログラムに通すことで実行ファイルとする、というものだ。これは Emacs における `unexec` に相当する。

`preloaded` 実行ファイルの使用は、現在では推奨されていない。計算機の高速化により、フォーマットファイルのロードが十分短時間で行えるようになった、というのがその理由である。

計算機の発展により `preloaded` は時代遅れとなったと断ずる TEX コミュニティの立場は、我々の portable dumper を支持するものであると考える。

8 おわりに

GNU Emacs に対する portable dumper の実装を紹介し、実験によって、依然必要な高速化手法のなかでも我々のアプローチが移植性の面で優れていることを明らかにした。

我々は、この完成した portable dumper のコードの著作権を FSF へ寄贈する書面へのサインを既に済ませており、近々 Emacs 本体に組み込まれてリリースされるよう作業中である。また、Emacs 21 ベースの Meadow 2 (現在開発中) では、2002 年 4 月リリースのヴァージョン 1.99 Alpha 2 において portable dumper を組み込み、`unexec` に代わり標準の高速化手法とした。双方とも広く皆様のお手元へ届けられる日は近い。

謝辞

The Meadow Team の皆様、特に宮下尚 (himi) 氏に深く感謝致します。

参考文献

[1] GNU Emacs. `http://www.gnu.org/software/emacs/emacs.html`.

[2] 宮下尚. Meadow. available at `ftp://ftp.m17n.org/pub/mule/Windows/`.

[3] XEmacs. `http://www.xemacs.org/`.

[4] XEmacs Internals Manual. 13 Dumping. `http://www.xemacs.org/Documentation/21.5/html/internals_13.html`.

[5] TEX. `http://www.tug.org/`.

[6] Web2c manual. 3.5.2 memory dumps. `http://www.tug.org/web2c/manual/web2c_3.html#SEC16`.

[*9] ファイル名の末尾が.fmt のファイル