オープンソースカンファレンス 2011 Nagoya セミナーテキスト

僕と契約して翻訳者になってよ！
～アプリケーション UI 翻訳入門～

今回のセミナーでは、何を取り扱うのか

ひとくちに翻訳といってもいろいろあります。ここでは、大きく「ドキュメント翻訳」と「メッセージ翻訳」の二つに分類し、後者のメッセージ翻訳を取り上げます。

ドキュメント翻訳

仕様書やマニュアル類など、普通の文章で書かれた文書です。様々な種類のものがあり、文書構造、更新頻度、管理用ファイル形式、公開用ファイル形式などによって作業方法が異なってきます。

例：オンラインマニュアル (NetBSD の wizd(8))

```
WIZD(8)                 NetBSD System Manager's Manual                 WIZD(8)

NAME
     wizd -- automatically correct typographical errors in man pages

SYNOPSIS
     wizd

DESCRIPTION
     wizd automatically checks and corrects spelling errors, usage problems
     with mdoc(7) macros, and other typographical errors in man pages.

     wizd is invoked by any cvs(1) commit to a man page.  A standalone mode is
     also available by sending mail to <wiz@NetBSD.org>.
```

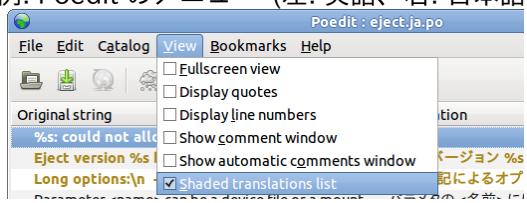
日本語に翻訳した例

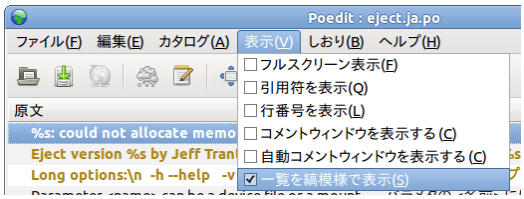
```
WIZD(8)                 NetBSD システム管理者マニュアル                WIZD(8)

名称
     wizd -- マニュアルページにおける誤字等の自動修正

書式
     wizd

解説
     wizd は、マニュアルページ中の綴りの誤り、mdoc(7) マクロの使い方の問題、
     その他の印字にまつわる問題を自動的に確認し修正します。

     wizd は、マニュアルページが cvs(1) commit される度に起動されます。
     <wiz@NetBSD.org> にメールを送れば、スタンドアローンモードを使うことも できます。
```

メッセージ翻訳

ソフトウェアのユーザーインターフェースなど、短いメッセージの集合体です。

例: Poedit のメニュー (左: 英語、右: 日本語訳)





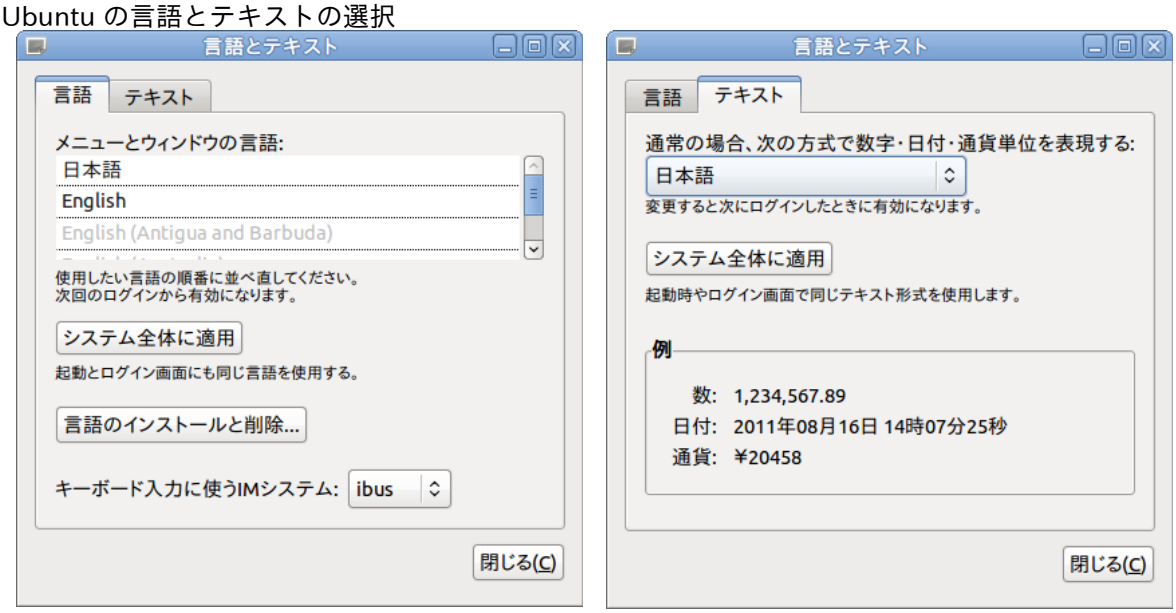
Linux の日本語対応の例

かつてはソフトウェアの日本語対応が不十分だったりまったくなかったりしたので、JPerl や Nemacs といった「日本語化」されたソフトウェアがいろいろ作られました。これらは日本語に特化したものなので、オリジナルの Perl や GNU Emacs とは別物です。

Linux ディストリビューションも日本語対応が十分ではなかったので、JE という日本語に特化したパッケージ集が作られました。Plamo や Vine のような「日本発」のディストリビューションでも、日本語環境の充実が大きな売りのひとつとなっていました。

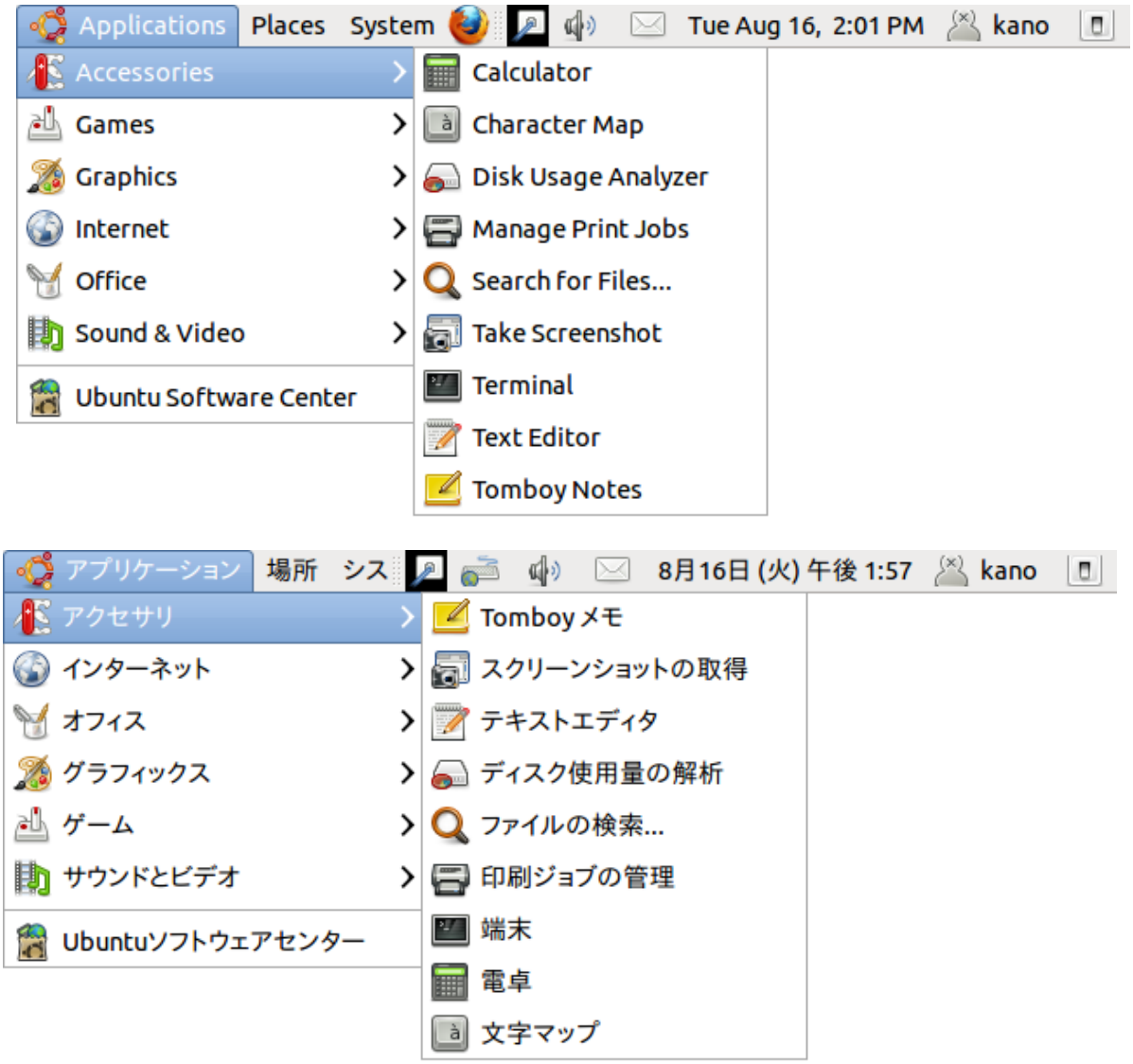
最近の Linux ディストリビューションでは、「日本発」のものでなくても、日本語がそれなりに使えるものが多くあります。たとえば、Ubuntu のソフトウェアパッケージには日本語のデータを処理できるものが多くありますし、Ubuntu の設定で日本語を選択すると、各種メッセージやヘルプ等は日本語で表示されるようになります[1]。

Ubuntu の言語とテキストの選択



1 それでも十分ではないかもしれないので、日本語 Remix といった取り組みがあったりします。

言語選択によって、メッセージが英語表示になったり日本語表示になったりする



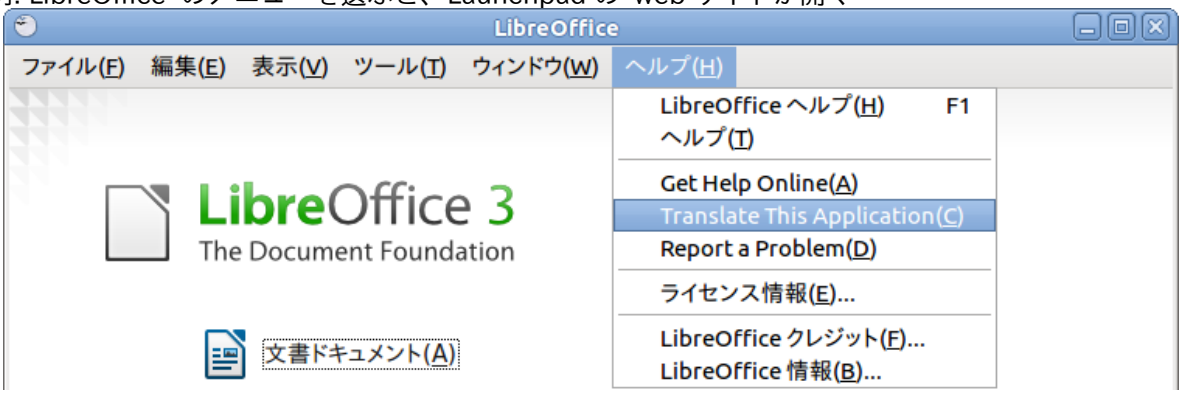
誰が翻訳しているのか

日本語を解さない欧米人が、日本語対応を勝手に作ってくれるわけはないので、日本語を解する人が翻訳したりしているわけです。多くのオープンソースソフトウェアでは、インターネット上のボランティアがコードを書いていますが、これと同様に翻訳もボランティアがやっています[2]。

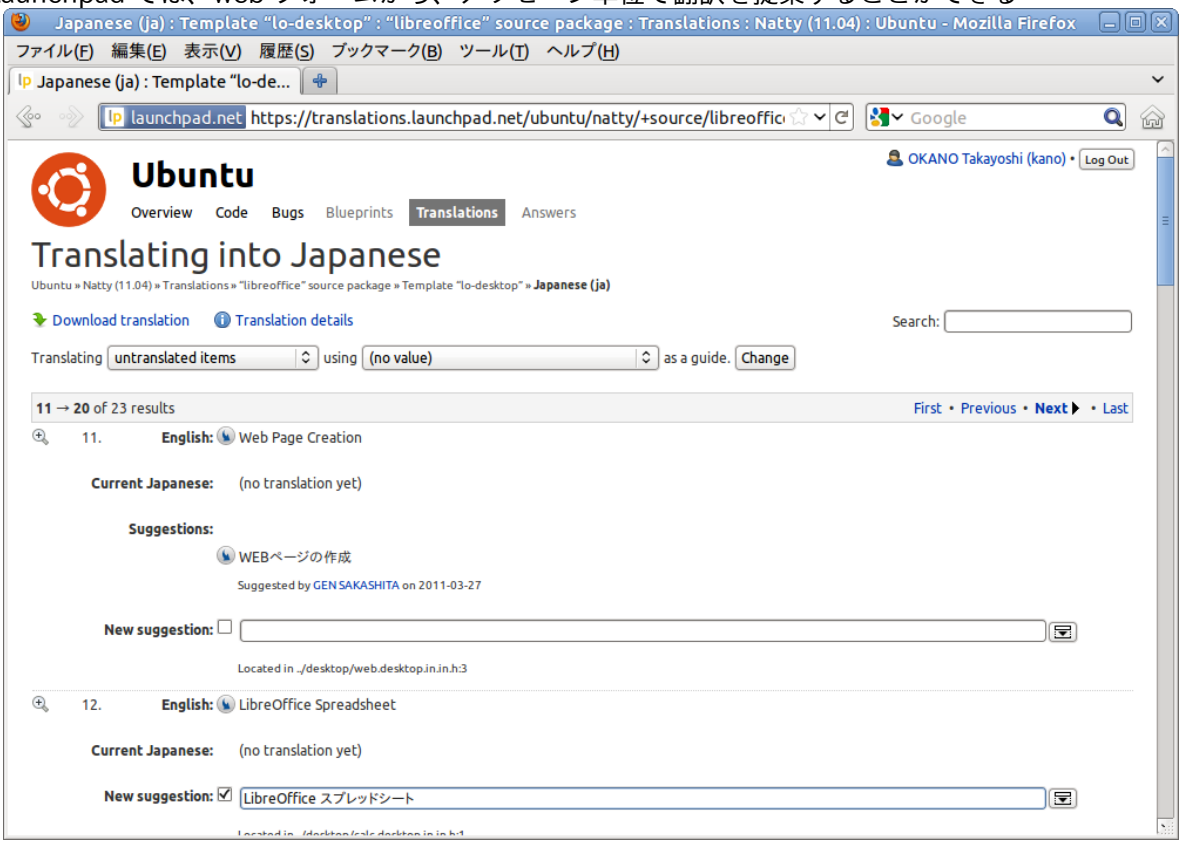
Ubuntu では、「このアプリケーションを翻訳する…」というメニューを選ぶと、Launchpad という web サイトが開くようになっています。ここに訳文を送信すると、査読されたうえでソフトウェアパッケージに取り込まれます。特別なツールがなくても、インターネット接続と web ブラウザさえあれば、誰でも翻訳作業に参加できるというわけです。

2 形態はプロジェクトにより様々です。閉じたメンバーだけで作業しているプロジェクトもありますし、企業が業務として作業することもあります。このあたりもコードと同様です。

例: LibreOffice のメニューを選ぶと、Launchpad の web サイトが開く



Launchpad では、web フォームから、メッセージ単位で翻訳を提案することができる



このように、Ubuntu では誰でも翻訳に参加しやすい仕組みが用意されていますが[3]、とっつきやすい仕組みを用意していない翻訳プロジェクトもあります[4]。

また、Ubuntu にしても、web インターフェースの裏側には別の仕組みがあります。今回のセミナーでは、このあたりの裏側の仕組みを少しだけ覗いてみることにします。

---

3 eject コマンドのような GUI のメニューを持たないソフトウェアでは、このようなわかりやすい誘導がありませんが、Launchpad のサイトに手動でアクセスすれば、翻訳の提案を web ブラウザを使っておこなうことができます。

4 それどころか、どこで翻訳しているかもよくわからないプロジェクトもあったりします。

国際化と地域化

現在、私たちのまわりでは舶来のソフトウェアが数多く使われています。これらは舶来品であるにもかかわらず、前述の Ubuntu の例のように、何も考えずとも日本語が使えるようになっています。

もちろん、何も考えなくてもいいのは利用者側の話であって、何も考えずとも日本語が使えるようにするために開発者側では多くの努力がおこなわれています。

英語圏で ASCII だけを扱うことしか考えていない場合は、8 ビット目が立った文字やエスケープ文字[5]を処理できなかったり、マルチバイト文字を適切に処理できなかったりします[6]。日本語を扱うためにはこれらを修正する必要があります。しかし、単に日本語のテキストを処理できればよいというわけではありません。文字集合、エンコーディング、フォント、文字方向、年月日・時刻の表示、通貨の表示、小数点等の表示……といったもの (ロカール) を、それぞれの地域に合わせる必要があります。

これらに加えて、メッセージを地域に合わせること (ようはメッセージ翻訳) も必要です。

地域化

ソフトウェアをある地域に合わせることを、地域化 (Localization: L と n の間に 10 文字あるので L10n と略します) といいます。

昔の舶来ソフトウェアは地域化への配慮などというものはされていなかったので、Jperl や Nemacs のように個別に地域化をおこなっていました。翻訳以外にも必要なことはいろいろある (というか、むしろ翻訳のほうが優先度が低い) のですが、ここではメッセージの翻訳に絞って説明します。

メッセージ翻訳の例

例: JNetHack

日本人なら誰でもプレイしたことがあるであろう JNetHack を例に取ります。

JNetHack は、Rogue ライクゲームの NetHack を日本語化したものです。NetHack のメッセージはすべてハードコードされているので、JNetHack では、ハードコードされたメッセージを直接翻訳する形をとっています。以下は NetHack 3.4.3 と、JNetHack 3.4.3-0.1.0 の src/allmain.c です。

nethack/src/allmain.c

```
    pline(new_game ? "%s %s, welcome to NetHack!  You are a%s %s %s."
                   : "%s %s, the%s %s %s, welcome back to NetHack!",
          Hello((struct monst *) 0), plname, buf, urace.adj,
          (currentgend && urole.name.f) ? urole.name.f : urole.name.m);
```

5 エスケープ文字は ASCII 印字可能文字ですが、エスケープ文字を扱えない端末があったりするせいで BNEWS ではエスケープ文字を通さず、ISO-2022-JP が使えませんでした。仕方がないのでローマ字で書いたり ^ と [に分けて配送したりしました。そしてできたのが JUNET 記念日。

6 とりあえず 8 ビット クリーンにして EUC-JP を使えば、なんとなく動いたので、それで誤魔化したりもしていました。

jnethack/src/allmain.c

```
#if 0 /*JP*/
    pline(new_game ? "%s %s, welcome to NetHack!  You are a%s %s %s."
                   : "%s %s, the%s %s %s, welcome back to NetHack!",
          Hello((struct monst *) 0), plname, buf, urace.adj,
          (currentgend && urole.name.f) ? urole.name.f : urole.name.m);
#else
    if(new_game){
      pline("%s, NetHackの世界へ！このゲームではあなたは%sの%s(%s)だ. ",
            Hello((struct monst *) 0, TRUE), urace.j,
            (currentgend && urole.jname.f) ? urole.jname.f : urole.jname.m,
            buf);
    } else {
      pline("%s, NetHackの世界へ！あなたは%sの%sだ！",
            Hello((struct monst *) 0, TRUE), urace.j,
            (currentgend && urole.jname.f) ? urole.jname.f : urole.jname.m);
    }
#endif
```

この手法には、気持ち悪いといったことのほかにも、以下のような問題点があります。

- 他の地域向けに地域化しようとすると、同じ作業を 0 からやらなければいけない
- 原文が更新されたときに、L10n 版を更新するのが、ものすごく面倒
- 翻訳を前提とした構成になっていないので、うまく翻訳できないことがあったりする

最後の問題の例としては、src/lock.c の以下の処理があります。unlock という英単語と、英語メッセージの構造 (“unlocking ...” の冒頭 2 文字を削れば“locking ...” にできる) に依存した処理となっているため、JNetHack では処理を変えるとともに愚痴を書いています。

nethack/src/lock.c

```
/* produce an occupation string appropriate for the current activity */
STATIC_OVL const char *
lock_action()
{
     /* "unlocking"+2 == "locking" */
     static const char *actions[] = {
            /* [0] */    "unlocking the door",
            /* [1] */    "unlocking the chest",
            /* [2] */    "unlocking the box",
            /* [3] */    "picking the lock"
     };

     /* if the target is currently unlocked, we're trying to lock it now */
     if (xlock.door && !(xlock.door->doormask & D_LOCKED))
            return actions[0]+2;      /* "locking the door" */
     else if (xlock.box && !xlock.box->olocked)
            return xlock.box->otyp == CHEST ? actions[1]+2 : actions[2]+2;
     /* otherwise we're trying to unlock it */
     else if (xlock.picktyp == LOCK_PICK)
            return actions[3];  /* "picking the lock" */
#ifdef TOURIST
     else if (xlock.picktyp == CREDIT_CARD)
            return actions[3];  /* same as lock_pick */
#endif
     else if (xlock.door)
            return actions[0];  /* "unlocking the door" */
     else
            return xlock.box->otyp == CHEST ? actions[1] : actions[2];
```

```
}
```

jnethack/src/lock.c

```
/* produce an occupation string appropriate for the current activity */
STATIC_OVL const char *
lock_action()
{
     /* "unlocking"+2 == "locking" */
     static const char *actions[] = {
/*
** 英語は un をつけるだけで逆の意味になるが，日本語はそうはいかない.
** 誰だ？こんな数バイトけちるコード書いたやつは？
*/
#if 0 /*JP*/
            /* [0] */    "unlocking the door",
            /* [1] */    "unlocking the chest",
            /* [2] */    "unlocking the box",
            /* [3] */    "picking the lock"
#else
            /* [0] */    "扉の鍵をはずす",
            /* [1] */    "宝箱の鍵をはずす",
            /* [2] */    "箱の鍵をはずす",
            /* [3] */    "鍵をはずす"
#endif /*JP*/
     };

     /* if the target is currently unlocked, we're trying to lock it now */
     if (xlock.door && !(xlock.door->doormask & D_LOCKED))
#if 0 /*JP*/
            return actions[0]+2;      /* "locking the door" */
#else
            return "扉に鍵をかける";
#endif
     else if (xlock.box && !xlock.box->olocked)
/*JP
            return xlock.box->otyp == CHEST ? actions[1]+2 : actions[2]+2;
*/
            return xlock.box->otyp == CHEST ? "宝箱に鍵をかける" : "箱に鍵をかける";
     /* otherwise we're trying to unlock it */
     else if (xlock.picktyp == LOCK_PICK)
            return actions[3];  /* "picking the lock" */
#ifdef TOURIST
     else if (xlock.picktyp == CREDIT_CARD)
            return actions[3];  /* same as lock_pick */
#endif
     else if (xlock.door)
            return actions[0];  /* "unlocking the door" */
     else
            return xlock.box->otyp == CHEST ? actions[1] : actions[2];
}
```

メッセージ分離

メッセージをハードコードした場合の問題を解消するために、ソフトウェア本体からメッセージを分離できるようにするという手法があります。

たとえば、Rogue ライクゲームのなかには、メッセージ分離型ローグ・クローン[7]というのがあり

7 Tim Stoehr による Rogue-clone II を、太田@リコーさんが日本語化したものが Rogue-clone 日本語版

ます。これは、以下のように、すべてのメッセージに通し番号を割り振り、英語と日本語それぞれの「メッセージカタログ」を用意しています。起動時に、メッセージカタログをコマンドライン引数で指定することで、ひとつのプログラムで英語メッセージと日本語メッセージを切り替えることができます[8]。

rogue_s/main.c (ゲーム開始時のメッセージ出力)

```
                sprintf(buf,
                      mesg[10],
                      nick_name);
                message(buf, 0);
```

rogue_s/mesg_E(英語メッセージカタログ[9])

```
10   "Hello %s, welcome to the Dungeons of Doom..." "やあ、%s。 運命の洞窟へようこそ..."
```

rogue_s/mesg (日本語メッセージカタログ)

```
10   "やあ、%s。 運命の洞窟へようこそ..."
```

メッセージの分離には、以下のような利点があります。

・すでにメッセージが分離されているので、あとは翻訳するだけ
・原文が更新されたら、そのメッセージだけ翻訳を更新すればよい
・コードが読めない人でも翻訳に参加しやすい[10]

メッセージ分離型ローグ・クローンでは、独自形式のメッセージカタログを作成しており、すべての処理を自前でおこなっています。しかし、メッセージ翻訳は多くのソフトウェアでおこなわれるものなので、メッセージ分離の手法が共通化されていたほうが何かと便利でしょう。ということで、実際に共通化がおこなわれています。

tcsh の例 (catgets を利用したメッセージ分離)

身近な例では tcsh があります。メッセージカタログを用意することでメッセージを各言語に対応させられるほか、日本語でもいろいろな方言や口調に切り替えることができます[11]。

tcsh のメッセージ翻訳では catgets を利用しています。メッセージカタログの形式は、メッセージ分離型ローグ・クローンと似ています。具体的には以下のような形です。

(jrogue)、それを伊藤@京大さんがメッセージ分離したのがメッセージ分離版ローグ・クローン (rogue_s)

8 それだけでなく、fj Rogue や Leaf Rogue といった派生品もあります。

9 日本語メッセージが併記されていますが、これはプログラムからは読み込まれません。

10 実際には、どのような状況で表示されるメッセージかをソースを見て確認する必要があったりしますが。

11 tcsh 日本語オリジナルカタログ
http://www2.wbs.ne.jp/%7Eznc/tcsh/catalogs/

tcsh-6.17.00/nls/C/set1 (英語メッセージカタログ)

```
14 Command not found
```

tcsh-6.17.00/nls/ja/set1 (日本語メッセージカタログ)

```
14 コマンドが見つかりません
```

メッセージの呼び出しには catgets(3) を使うのですが、独自のマクロと関数を介していてアレなので、ソースは省略します。

翻訳への配慮

言語によって文法 (語順、複数形の扱い等) が異なりますので、原文でそこが配慮されていないと、うまく翻訳できない場合があります。GNU make は gettext (次節で説明します) を使ってメッセージを分離していますが、かつての GNU make では、「Entering directory `/foo/bar'」の訳が「入ります ディレクトリ `/foo/bar'」という不自然な日本語になっていました。これは原文の語順を変えられない構成になっていたからです。この問題は現在では解消されており、日本語訳は「ディレクトリ `/foo/bar' に入ります」となっています[12]。

原作者は、そこまで配慮してメッセージを書く必要があるということです[13]。

make-3.79.1/main.c (一つの文を構成する "Entering" と "directory `%s'¥n" が別メッセージとなっており、出力の順序は変えられない)

```
/* Write a message indicating that we've just entered or
   left (according to ENTERING) the current directory.  */

void
log_working_directory (entering)
    int entering;
{
  static int entered = 0;
  char *msg = entering ? _("Entering") : _("Leaving");

  /* Print nothing without the flag.  Don't print the entering message
     again if we already have.  Don't print the leaving message if we
     haven't printed the entering message.  */
  if (! print_directory_flag || entering == entered)
    return;

  entered = entering;

  if (print_data_base_flag)
    fputs ("# ", stdout);

  if (makelevel == 0)
```

12 GNU make 3.79 には「TAB のつもりですか?」というメッセージに腹が立つという問題もありました。

13 GNU make の例のように、配慮されていない事例はままあります。こういうのはバグとして報告しましょう。

```
    printf ("%s: %s ", program, msg);
  else
    printf ("%s[%u]: %s ", program, makelevel, msg);

  if (starting_directory == 0)
    puts (_("an unknown directory"));
  else
    printf (_("directory `%s'¥n"), starting_directory);
}
```

make-3.80/main.c (翻訳に配慮し、一つの文は一つのメッセージとなった)

```
  /* Use entire sentences to give the translators a fighting chance.  */

  if (makelevel == 0)
    if (starting_directory == 0)
      if (entering)
        printf (_("%s: Entering an unknown directory"), program);
      else
        printf (_("%s: Leaving an unknown directory"), program);
    else
      if (entering)
        printf (_("%s: Entering directory `%s'¥n"),
                program, starting_directory);
      else
        printf (_("%s: Leaving directory `%s'¥n"),
                program, starting_directory);
  else
    if (starting_directory == 0)
      if (entering)
        printf (_("%s[%u]: Entering an unknown directory"),
                program, makelevel);
      else
        printf (_("%s[%u]: Leaving an unknown directory"),
                program, makelevel);
    else
      if (entering)
        printf (_("%s[%u]: Entering directory `%s'¥n"),
                program, makelevel, starting_directory);
      else
        printf (_("%s[%u]: Leaving directory `%s'¥n"),
                program, makelevel, starting_directory);
```

国際化

メッセージ分離は、メッセージの地域化を容易にするための手法のひとつです。前述のとおり、地域化においては、メッセージ以外にもさまざまなもの (ロカール) をその地域や言語に合わせる必要が

あります。

ロカールに依存する部分を本体から分離して、複数のロカールに対応できるようにすることを国際化 (Internationalization : i と n の間に 18 文字あるので i18n と略します) といいます。何も考えずに舶来のソフトウェアを日本語環境で使うことができるのは、そのソフトウェアが国際化されているからです。

gettext のメッセージカタログ

前述のとおり、メッセージ分離にもいくつもの手法があります。共通化されているものとしては、XPG による catgets と、Sun による gettext があります。catgets は現在では POSIX 標準となっています。

ここでは現在それなりに普及しているような気がする gettext による地域化を取り上げます。gettext はもとは Sun の独自実装でしたが、現在は GNU gettext が事実上の標準となっています[14]。

gettext の超概要[15]

gettext では

- 原文と訳文をセットにしたメッセージカタログを用意しておく
- 言語によって変える必要のあるメッセージは、gettext(3) を介して出力する
- メッセージの ID には、(catgets のような番号ではなく) メッセージ原文を用いる。

という方法を取ります。

具体的には、

```
printf ("Hello, World!\n");
```

のようなコードのメッセージを分離する場合は

```
printf (gettext("Hello, World!\n"));
```

のように書くことになります。

これに対して、以下のようなメッセージカタログを用意して原文と訳を対応づけることで、日本語環境では日本語のメッセージを表示することができます。

```
msgid "Hello, World!\n"
msgstr "こんにちは、世界!\n"
```

なお、いちいち gettext() とか書くのは面倒なので、通常はマクロを定義して _() と書きます。

```
printf (_("Hello, World!\n"));
```

メッセージカタログのファイルは、Portable Object の頭文字をとって PO ファイルといい、ファイル名の末尾も通常は .po です。

xgettext(1) というコマンドにより、ソースからメッセージを全部取り出して、メッセージカタログの雛形 (Portable Object Template - POT ファイル) を作ることができます。これは以下のように、訳文が空文字列になった状態になります。

---

14 GNU というからには GPL (一部 LGPL) なので、他の実装もあります。たとえば NetBSD のライブラリーでは、Citrus プロジェクトによる実装が使われています。

15 しっかりした説明は、GNU gettext のマニュアルなどを参照してください。
http://www.gnu.org/software/gettext/manual/gettext.html (info gettext でも同じものを参照可能)

```
# src/hello.c: 69
msgid "Hello, World!\n"
msgstr ""
```

翻訳する場合は、この雛形に訳文を埋めていけばよいのです。

「#」で始まる行はコメントです。メッセージがどのソースファイルの何行目にあるかを xgettext(1) が自動的にコメントに含めてくれるので、ソースを確認する際に便利です。

PO ファイルと MO ファイル

PO ファイルは人間が読むことのできるテキストファイルですが、ソフトウェアがこれをそのまま使うわけではありません。ソフトウェアに組み込んで利用する際にはバイナリーファイルである MO (Machine Object) ファイルを使います。

通常、ソフトウェアのソース配布物には PO ファイルが含まれており、バイナリー配布物には MO ファイルが含まれています。動画編集ソフト PiTiVi を例にとると、PiTiVi 0.13.4 のソース[16]では、pitivi-0.13.4/po ディレクトリーに各言語の PO ファイルがあります (日本語用は ja.po)。そして、NetBSD のバイナリーパッケージ[17] pitivi-0.13.4nb5 では、share/locale ディレクトリー以下に各言語の MO ファイルがあります (日本語用は ja/LC_MESSAGES/pitivi.mo)。

PO から MO へのコンパイルは msgfmt(1) というコマンドで、MO から PO への逆コンパイルは msgunfmt(1) というコマンドでおこなうことができます。

翻訳作業はもっぱら PO ファイルを使っておこない、実際にソフトウェアに組み込む際にだけ MO ファイルを使うことになります。

新規に翻訳を始める場合

gettext を利用しているソフトウェアでは、たいていはリリース版のソース配布物の中に POT ファイルと各言語の PO ファイルが含まれています[18]。

すでに日本語用の PO ファイルが存在する場合は、そのファイルの未訳のメッセージを翻訳するなどして、すべてのメッセージを翻訳済の状態にします(次節参照)。

日本語用の PO ファイルが存在しない場合は、POT ファイルをもとに全部翻訳すればよいです。

たとえば GNOME ならば、GNOME 翻訳プロジェクトの web サイトに POT ファイルや、最新の原文にあわせた PO ファイルが用意されているので、ダウンロードします。

PO ファイルは単なるテキストファイルですから、テキストエディターさえあれば編集することができます。たとえば GNOME 標準の gedit とか、KDE 標準の KEdit などでも編集できます[19]。

---

16 ftp://ftp.gnome.org/pub/GNOME/sources/pitivi/0.13/

17 ftp://ftp.NetBSD.org/pub/pkgsrc/current/pkgsrc/multimedia/pitivi/README.html

18 大規模なソフトウェアではメッセージカタログも大きくなるので、言語別のメッセージカタログを別に配布することもあります。

19 こういうときは「vi でも Emacs でも」というところでしょうが、Emacs には PO モードがあるので、あえてこのように書きました。しかし gedit にも gedit-pomode というプラグインがあったりします。

全部翻訳し終わったら、しかるべき宛先に提出しましょう。GNOME なら GNOME 翻訳プロジェクトです。これで、うまくいけば、次のリリースからは日本語メッセージが使えるようになるわけです。

原文が更新された場合

原文が更新された場合は、

- 原文が変わっていない場合は、既存の訳をそのまま使う
- 原文が少し変わった場合は、適宜、訳を修正する
- 新しい原文が追加された場合は、一から訳す (似たようなメッセージが別にあれば、それをもとに修正してもよい)

という作業をすることになります。個々のメッセージについて、手作業でこんなことをしていては死んでしまうので、そのためのツールとして msgmerge(1) が用意されています。

msgmerge(1) を使うと、以下のようにメッセージカタログを更新 (新しい POT ファイルを古い PO ファイルにマージ) できます。

- 原文が変わっていない場合は、訳がそのまま使われる→そのままでよい
- 原文が少し変わった場合は、fuzzy マークがつく→適宜、訳を修正し、fuzzy マークを外す
- 新しい原文が追加された場合は、未訳となる→一から訳す (似たようなメッセージが別にあれば、fuzzy マークつきでそのメッセージの訳が使われるので、それをもとに修正する)

マージ前の PO ファイル

```
# src/hoge.c: 69
msgid "About..."
msgstr "このプログラムについて..."

# src/tara.c: 69
msgid "File does not exist!"
msgstr "ファイルが存在しません!"
```

新しい POT ファイル

```
# src/hoge.c: 69
msgid "About..."
msgstr ""

# src/tara.c: 69
msgid "File does not exist"
msgstr ""

# src/tara.c: 79
#, c-format
msgid "Fatal error: %s"
msgstr ""
```

原文に変更なし

原文に少し変更あり

メッセージの追加

マージ後の PO ファイル

```
# src/hoge.c: 69
msgid "About..."
```

```
msgstr "このプログラムについて..."

# src/tara.c: 69
#, fuzzy
#| msgid "File does not exist!"
msgid "File does not exist"
msgstr "ファイルが存在しません!"

# src/tara.c: 79
#, c-format
msgid "Fatal error: %s"
msgstr ""
```

以前の訳がそのまま使われる

以前の訳に、fuzzy マークがつく

似たメッセージがなければ、訳は空になる

マージ後の PO ファイルを見て、訳が空になっている箇所があれば翻訳し、fuzzy マークがあれば訳を確認・修正すれば、新しい POT ファイルに対応する PO ファイルができあがります。

翻訳したメッセージをソフトウェアに組み込んでみる

PO ファイルの翻訳手順は前述のとおりですが、適切に翻訳できているか確認するために、実際にソフトウェアに組み込んでメッセージを表示させてみる必要があります。

ソフトウェアに組み込むのは PO ファイルではなく、PO ファイルをコンパイルした MO ファイルです。msgfmt(1) を使って PO ファイルから MO ファイルを作ります。

```
$ msgfmt foo.ja.po -o foo.mo
```

MO ファイルを、しかるべき場所に置きます。場所は OS や対象ソフトウェアによって異なりますが、Filesystem Hierarchy Standard[20] 的には /usr/share/locale の下とされているので、Linux ディストリビューションによってはそのあたりにあるかもしれません[21]。以下の例は、NetBSD のパッケージシステムによりインストールされたソフトウェアでのものです。

```
# cp foo.mo /usr/pkg/share/locale/ja/LC_MESSAGES/
```

これで、対象ソフトウェアを実行すれば、翻訳したメッセージが反映されるようになります。

20 http://www.pathname.com/fhs/

21 ないかもしれません。locate LC_MESSAGES すれば何か見つかるかもしれません。

翻訳支援ツール

お好みのテキストエディターで編集するだけでも翻訳はできますが、ふつうのテキストエディターで作業するのはけっこう面倒です。

こんなこともあろうかと、便利な翻訳支援ツールが用意されています。

PO エディター概要

ふつうのテキストエディターでの作業には、以下のような問題があります。

- 処理の必要があるメッセージを探しにくい。fuzzy のものは「, fuzzy」で検索すればよいが、未訳のものを探すのは面倒
- 訳語や言い回しを統一したいが、メッセージが多いと大変

そこで、PO ファイルの編集に特化したエディターを使うと便利です。PO エディターによって機能は異なりますが、たとえば以下のような機能があったりします。

- 未訳、fuzzy、翻訳済みのメッセージ数がすぐわかる
- 処理の必要のあるメッセージにすぐたどり着ける
- 翻訳メモリーにより、既成の訳の流用や、訳語や言い回しの統一がしやすい

ここでは Poedit を例に挙げて説明します。

Poedit の機能

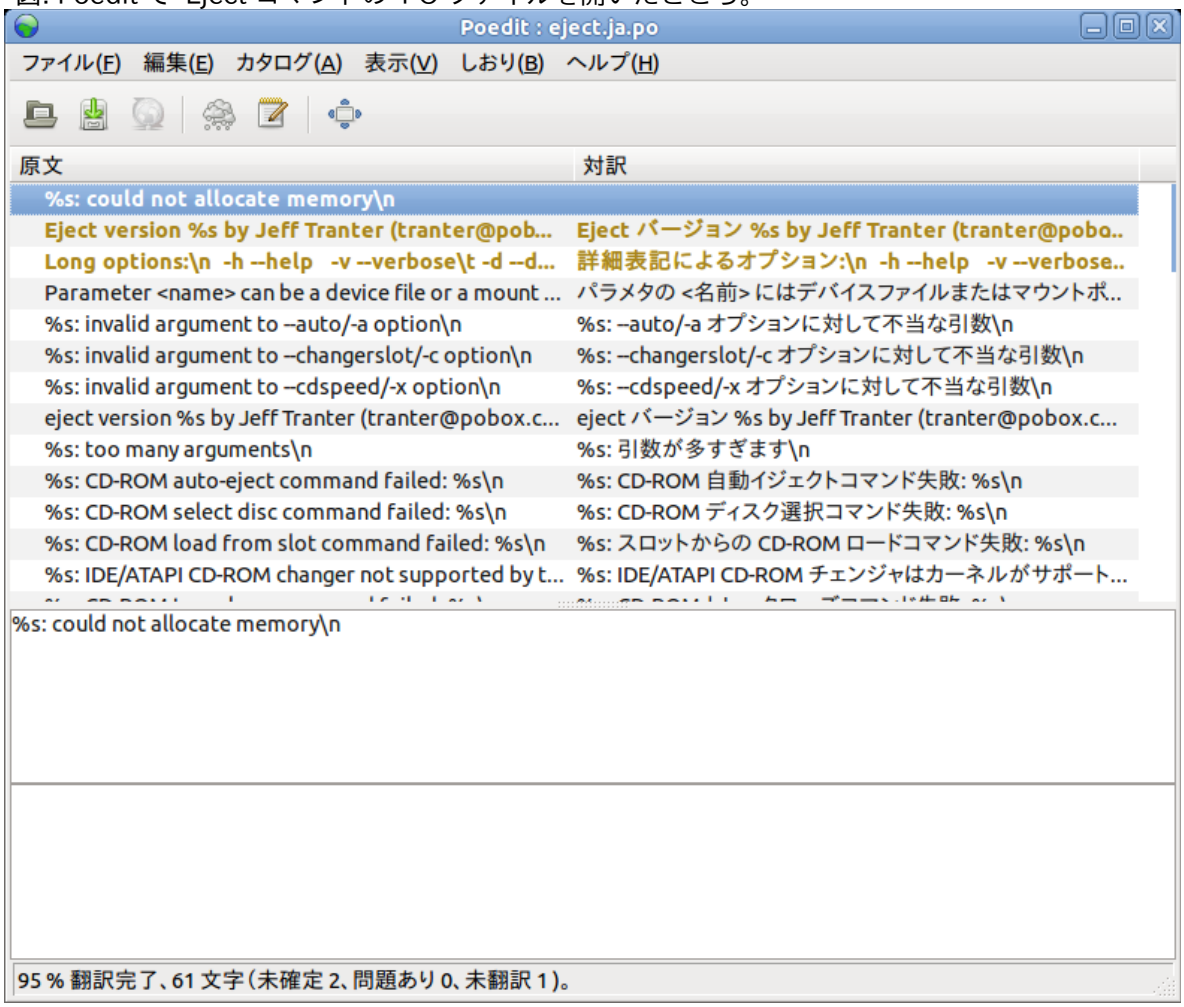
Poedit の web サイトは http://www.poedit.net/ です。

以下に主な機能を掲げますが、詳細は Poedit のドキュメントを参照してください。

進捗管理

PO ファイルを開くと、メッセージが一覧表示されますが、メッセージの状態 (翻訳済・未訳・fuzzy) 別に表示スタイルが変わっており、どのメッセージがどの状態にあるかが一目瞭然です。また、進捗率と未確定メッセージ数が常時表示されます。

図: Poedit で Eject コマンドの PO ファイルを開いたところ。



個々の PO ファイルの翻訳状況を把握できるだけでなく、「カタログマネージャー」に複数の PO ファイルを登録することにより、プロジェクト全体の翻訳状況を把握することができます。

メッセージ編集支援

PO ファイルでは、メッセージはプログラムソースから抽出された順に並んでいますが、Poedit では翻訳状態の順に並び替えて表示されます。

メッセージによってはプログラムソースを確認しないと訳しにくいものもあります。Poedit ではソースの該当箇所を即座に参照することができます。

翻訳メモリー

翻訳メモリーとは、その名のとおり、既存の原文と翻訳の組み合わせを蓄積したデータベースです。

Poedit では、未訳メッセージに対して、翻訳メモリーに同一または類似の原文がある場合は既存の訳を流用する形で自動翻訳をすることができます。

翻訳メモリーを使うことで、翻訳作業の手間が減らせることに加え、プロジェクト内の訳語や文体の統一がしやすくなるという利点もあります。

プログラムソースとの同期

PO ファイルよりプログラムソースが新しい場合、プログラムソースからメッセージを抽出し、メッセージカタログにマージすることができます (通常は xgettext(1) や msgmerge(1) を使っておこなう作業を、メニューから一発でやってくれる)。

その他の PO エディター

たとえば以下のようなものがありますので、使いやすいものを使えばよいでしょう。

Emacs の PO モード (po-mode.el)

http://www.emacswiki.org/emacs/PoMode

GNU gettext に含まれている

OmegaT

http://omegat.org/

PO のみならず多くのファイル形式に対応した翻訳支援ツール

Gtranslator

http://projects.gnome.org/gtranslator/

GNOME の PO エディター

Lokalize

http://www.kde.org/applications/development/lokalize/

KDE の PO エディター

web インターフェース

PO エディターを使えば作業をかなり効率化できますが、とっつきにくいという方がいらっしゃるかもしれません。PO ファイルの存在を意識することなく翻訳できるツールもあります。

Launchpad Translations

https://translations.launchpad.net/

Launchpad は Ubuntu の Canonical が管理しているソフトウェア共同開発プラットフォームです。このなかには翻訳支援ツール Translations もあります。

Ubuntu のアプリケーションには「このアプリケーションを翻訳する…」「問題を報告する…」というメニューがありますが、テキスト冒頭で紹介したとおり、これらを選択するとインターネット経由で Launchpad にアクセスします。

翻訳に際して po ファイルの知識は不要で、メッセージ単位で翻訳を提出することができます。

Ubuntu の翻訳はこの Launchpad を使っておこなわれていますが、Ubuntu 以外の翻訳にも使う

ことができ、いくつかの翻訳プロジェクトが利用しています。Launchpad そのものは OSS でないので、使いたければ Canonical のサイトを使うことになります。

Pootle

http://translate.sourceforge.net/wiki/

翻訳支援に特化したツールです。LaunchPad 同様に、メッセージ単位で翻訳をすることができる web システムです。

Pootle そのものが OSS なので、自分でサービスを立ち上げることもできます。

PO 以外の形式のファイルの翻訳

gettext は多くのソフトウェアで使われており、各種ツールも充実しています。しかし、gettext や PO ファイルを使っていないソフトウェアもあります。たとえば PHP を使った web アプリケーションでは、メッセージカタログが PHP の配列になっているものがあります[22]。

このような PO 以外の形式のメッセージカタログを直接翻訳しようとすると、gettext や PO エディター等を使うことはできません。しかし、メッセージカタログの変換をしてくれる便利なツールがありますので、このようなツールを使って PO ファイルに変換すれば、充実したツールを使うことができます。

また、ドキュメント翻訳では、原文はメッセージカタログの形式とはなっていないのがふつうです[23]。しかし、これも PO ファイルの形式に変換して作業する方法があります。

ITS tool

GNOME では[24]、XML で書かれたヘルプファイルを、ITS tool[25] というツールを使って PO ファイルに変換し、この PO ファイルを翻訳しています。これにより、翻訳作業者は元のファイル形式が PO であるか XML であるかにかかわらず、PO ファイルだけを使って同じ手順で作業することができます。

もちろん、翻訳に問題がないか確認するためには、PO ファイルを読み書きするだけでなく、実際のソフトウェアに組み込んで表示を確認するといった作業も必要になります。こういうことをやろうとすると、元のファイル形式に変換したりするので、意識することになりますが。

また、ソフトウェアのユーザーインターフェースのメッセージと、マニュアル類のようなドキュメントでは、翻訳そのものも変わってきます。ユーザーインターフェースは、表示スペースが限られることもあって、非常に簡潔なメッセージになることがあります。どのような状況でのメッセージであるかによって訳は変わりますし、メッセージの長さを抑えるために工夫が必要になったりもします。

より汎用的なツール

ITS tool は XML と PO の変換をするツールですが、さらに多くの形式に対応したより汎用的な

---

22 PHP でも gettext を使うことができますが、標準状態では有効になっていません。
23 プレインテキストだったり roff だったり TeX だったり HTML だったり XML だったり……
24 KDE では同様のことを po2xml というツールを使っておこなっています。
25 http://itstool.org/

ツールとして、po4a[26] や translate toolkit[27] があります。

po4a という名前は po for anything に由来しており、さまざまなファイルと PO ファイルの変換をするツールです。roff や XML など、メッセージカタログの形式をとっていないファイルも、PO に変換して、PO と同じ手順で翻訳作業をすることができます。

translate toolkit は、翻訳関連のファイル形式を変換するツールです。po4a のように、メッセージカタログの形式をとっていないファイルを PO に変換することができるほか、各種のメッセージカタログ (PO のほか、XLIFF や PHP 配列等)、翻訳メモリーなどのファイル形式変換をすることができます。

また、翻訳用のエディターのなかには、さまざまな種類のファイルを直接入出力することができるものもあります。前述の OmegaT は、さまざまなファイル形式に対応しているため、各種のファイル形式に対して、同じ手順で翻訳作業をすることができます。

GNOME 翻訳ステータス (damned-lies)

前述のとおり、GNOME での翻訳作業は PO ファイルを使っておこなっています。そして、成果物や進捗の管理には、GNOME 翻訳ステータス (damned-lies) という web システムを使っています。

翻訳プロジェクトの活動の場としてはメーリングリストがありますが、個々の翻訳の提案や査読は damned-lies 上でおこなっています。

成果物は PO ファイル単位で登録するようになっているので、手元で PO ファイルを編集する必要があります。

damned-lies の使い方などの詳細は、「GNOME 日本語翻訳チーム参加者ガイド」をご覧ください。

26 http://po4a.alioth.debian.org/

27 http://translate.sourceforge.net/wiki/toolkit/index

翻訳プロジェクトの抱える問題

人手が足りない

数ある翻訳プロジェクトが共通して抱える問題が、人手不足です。

活発なソフトウェア開発プロジェクトでは、原文が絶えず更新されるので、翻訳も更新し続ける必要があります。翻訳する人手が足りないために原文に追従できなかったり、査読する人手が足りないために品質が確保できなかったりします。

また、翻訳そのもの以外にも、成果物の管理や上流との調整など、必要な作業はいろいろあり、これにもそれなりに手間がかかります。

翻訳が維持できなかった事例

人気のある統合デスクトップ環境である KDE では、日本語翻訳率が低下したことがあり、2004 年にリリースされた KDE 3.2.0 で日本語メッセージの配布が中止されました。翻訳率が低く原文と翻訳メッセージが混在するぐらいならば、ないほうがマシという判断をされることもあるわけです。なお、その後は翻訳率が改善し、KDE 3.2.1 以降では日本語メッセージが配布されています。

査読が維持できなかった事例

査読に関する事例としては FreeBSD があります。jp.FreeBSD.org では古くからオンラインマニュアルの翻訳をしていましたが、査読が十分できなくなったことから、翻訳プロジェクトとしての日本語マニュアル配布は FreeBSD 5.4-RELEASE 用を最後におこなわれていません。これ以降のものについては、翻訳プロジェクトのメンバーの方が個人的に日本語マニュアルを作成・配布しています。

フィードバックがない

誤字や誤訳等があっても、そのことを翻訳プロジェクトに報告してくれる利用者の方はあまりいないようです。

翻訳者は原文を読んで理解したうえで翻訳をしていますし、査読者も原文を読んでいることが多いので、「訳文だけ読んでもわかりにくい」といった問題には気づきにくいものです。また、翻訳に限らず、自分で書いた文章のアラは見えにくいものですので、他の方からの指摘は大変ありがたいものです。

些細なことであっても[28]、報告すると歓迎されるものと思います。

参加するにあたっては

空気を読む努力も

人手が足りないのですから、ふつうに考えれば参加者は歓迎されるでしょう。

---

28 そのことを強調するため、FreeBSD jpman や Linux JM では「1 バイトの更新でも大歓迎です」と書いています。

ただし、プロジェクトごとに作業の進め方は異なりますし、用語や文体のルールも異なります。メッセージ翻訳などでは大胆な意訳も許容されるでしょうが、仕様書のような厳密さを求められるドキュメントの翻訳では直訳に近い翻訳をしなければならない場合もあります。

本格的に翻訳プロジェクトに関わるならば、(メーリングリストやリポジトリーのログを見るなど)ある程度様子を見てからにしたほうがよいでしょう。

また、プロジェクトとして品質の低い翻訳を受け入れるわけにはいかないので、提出した翻訳が採用されなかったり、跡形もなく修正されたりすることもありえます。ダメ出しをするのは品質向上のためであって、貢献したいという人を追い出そうとしているわけではないので、そういうものだと思って臆せず参加してください。

誤字や誤訳の指摘といった場合は、そういったことをあまり気にする必要はないので、軽い気持ちで参加するなら、そのようなことから始めるのもよいかもしれません。

身近なプロジェクトに参加してみよう

自分が興味がない作業をしてもつまらないでしょうし適切な翻訳をしづらいでしょうから、現に使っているソフトウェアなど、まずは身近なプロジェクトに参加してみてはどうでしょうか。

オープンソースカンファレンスに参加している団体の大半は、「翻訳」を表に出していませんが、主体的に翻訳をしていたり、メンバーに翻訳者がいたりするなど、翻訳に関わりのある団体が多くあります。

Doc-ja Archive Project は、翻訳プロジェクト参加者が自発的に参加しているだけの緩やかな集まりであり、翻訳プロジェクトを組織化するといった性質のものではありませんので、個々の翻訳プロジェクトの状況を的確に把握しているわけではありませんが、可能な範囲で調べた結果を「観光ガイド(仮)」として配布しています。翻訳に関心をお持ちの皆さんの参考になれば幸いです。

参考文献

- GNU gettext マニュアル
  http://www.gnu.org/software/gettext/manual/
- GNOME 日本語翻訳チーム参加者ガイド
  http://www.gnome.gr.jp/l10n/gnomeja-guide/gnomeja-guide.html
- 翻訳ガイド（アプリケーション編） - Ubuntu Japanese Wiki
  https://wiki.ubuntulinux.jp/Develop/TranslationGuide

本テキストで例としてとりあげたソフトウェアの配布元

- wizd(8) (マニュアルのみ)
  http://www.NetBSD.org/
- NetHack
  http://www.nethack.org/
- JnetHack
  http://sourceforge.jp/projects/jnethack/
- Rogue Clone II
  http://rogueclone.sourceforge.net/
- メッセージ分離型ローグ・クローン
  news:fj.sources
- tcsh
  http://www.tcsh.org/
- GNU make
  http://www.gnu.org/software/make/

奥付

Doc-ja Archive Project
2011 年 8 月 20 日発行

文責: 岡野孝悌