PDFlib GmbH

PDFlib, PDFlib+PDI, PPS

A library for generating PDF on the fly

Version 8.0.3

チュートリアル

Cobol・C・C++・Java・Perl・
PHP・Python・RPG・Ruby・Tcl エディション

PDFlib GmbH
Franziska-Bilek-Weg 9, 80339 München, Germany
www.pdflib.com
電話 *+49 • 89 • 452 33 84-0*
FAX +49 • 89 • 452 33 84-99

疑問点がおありの場合は *tech.groups.yahoo.com/group/pdflib* にある *PDFlib* メーリングリストとアーカイブをご覧ください。

ライセンス取得のための連絡先：*jp.sales@pdflib.com*
商用 *PDFlib* ライセンス保持者向けサポート：*jp.support@pdflib.com*（ライセンス番号をお知らせください）

この出版物およびここに含まれた情報はありのままに供給されるものであり、通知なく変更されることがあり、また、*PDFlib GmbH* による誓約として解釈されるべきものではありません。*PDFlib GmbH* はいかなる誤りや不正確に対しても責任や負担を全く負うものではなく、この出版物に関するいかなる類の（明示的・暗示的または法定に関わらず）保障をも行うものではなく、そして、いかなるそしてすべての売買可能性の保障と、特定の目的に対する適合性と、サードパーティの権利の侵害とを明白に否認します。

PDFlib と *PDFlib* ロゴは *PDFlib GmbH* の登録商標です。*PDFlib* ライセンス保持者は *PDFlib* の名称とロゴを彼らの製品の文書内で用いる権利を与えられます。ただし、これは必須ではありません。

Adobe・*Acrobat*・*PostScript*・*XMP* は *Adobe Systems Inc.* の商標です。*AIX*・*IBM*・*OS/390*・*WebSphere*・*iSeries*・*zSeries* は *International Business Machines Corporation* の商標です。*ActiveX*・*Microsoft*・*OpenType*・*Windows* は *Microsoft Corporation* の 商標です。*Apple*・*Macintosh*・*TrueType* は *Apple Computer, Inc.* の 商標です。*Unicode*・*Unicode* ロゴは *Unicode, Inc.* の 商標です。*Unix* は *The Open Group* の 商標です。*Java*・*Solaris* は *Sun Microsystems, Inc.* の 商標です。*HKS* は *HKS* 商標連合＝ *Hostmann-Steinberg*・*K+E Printing Inks*・*Schmincke* の登録商標です。他の企業の製品とサービス名は他の商標やサービスマークである場合があります。

ソフトウェアアプリケーションやユーザー向け文書で表示される *PANTONE®* カラーは *PANTONE* 定義規格と一致しない場合があります。正確な色については最新の *PANTONE Color Publication* をご覧ください。
PANTONE® およびその他の *Pantone, Inc.* の商標は *Pantone, Inc.* に帰属します。*© Pantone, Inc., 2003.*
Pantone, Inc. は *PDFlib GmbH* に対して *PDFlib* ソフトウェアとの組み合わせでのみ使用するための頒布ライセンスされた色データおよび／またはソフトウェアの著作権者です。*PANTONE* カラーデータおよび／またはソフトウェアは *PDFlib* ソフトウェアの実行の部分として以外に他のディスク上やメモリ内へ複製してはいけません。

PDFlib は以下のサードパーティソフトウェアの変更された部分を含んでいます。
ICClib、*Copyright © 1997-2002 Graeme W. Gill*
GIF 画像デコーダ、*Copyright © 1990-1994 David Koblas*
PNG 画像参照ライブラリ（*libpng*）、*Copyright © 1998-2004 Glenn Randers-Pehrson*
Zlib 圧縮ライブラリ、*Copyright © 1995-2002 Jean-loup Gailly and Mark Adler*
TIFFlib 画像ライブラリ、*Copyright © 1988-1997 Sam Leffler*、*Copyright © 1991-1997 Silicon Graphics, Inc.*
Eric Young の書いた *Cryptographic* ソフトウェア 、*Copyright © 1995-1998 Eric Young*（*eay@cryptsoft.com*）
Independent JPEG Group の *JPEG* ソフトウェア、*Copyright © 1991-1998, Thomas G. Lane*
Cryptographic ソフトウェア、*Copyright © 1998-2002 The OpenSSL Project*（*www.openssl.org*）
Expat XML パーサ、*Copyright © 1998, 1999, 2000 Thai Open Source Software Center Ltd*
ICU International Components for Unicode、*Copyright © 1995-2009 International Business Machines Corporation and others*
参照 *sRGB ICC* カラープロファイルデータ、*Copyright © 1998 Hewlett-Packard Company*

PDFlib は *RSA Security, Inc.* の *MD5* メッセージダイジェストアルゴリズムを含んでいます。

目次

o PDFlib ライセンスキーの適用

PDFlib GmbH が提供する PDFlib・PDFlib+PDI・PPS のバイナリバージョンはいずれも、商用ライセンスをお持ちでなくても、フル機能の評価版としてご利用いただけます。ただし未ライセンス版では、生成されるページすべてに、*www.pdflib.com* というデモスタンプが横断印字され、また、内蔵の pCOS インタフェースは小容量の文書に限られます（詳しくは PDFlib チュートリアルを参照）。PDFlib ライセンスの取得を前向きにご検討いただいている企業で、評価段階やプロトタイプのデモ期間中に評価制約の除去をご希望の場合は、*sales@pdflib.com* 宛に企業情報・プロジェクト内容を簡単にご説明いただければ、一時的なライセンスキーをご提供します（評価キーの提供をお断りする権利を私達は保持いたします。たとえば匿名によるご希望の場合等）。

PDFlib・PDFlib+PDI・PDFlib Personalization Server（PPS）は 1 つのパッケージとして頒布されてはいますが、それぞれ異なる製品であり、別々のライセンスキーを必要とすることに注意してください。PDFlib+PDI のライセンスキーは PDFlib に対しても有効ですが、その逆は無効であり、また、PPS のライセンスキーは PDFlib+PDI と PDFlib に対して有効です。すべてのライセンスキーはプラットフォーム依存であり、購入された対象のプラットフォームでしか使用できません。

PDFlib または PDI のライセンスキーをご購入いただいたら、それを適用してデモスタンプを除去してください。ライセンスキーを設定するにはいくつかの方法があります。以下にそれを解説します。

クックブック 完全なコードサンプルがクックブックの general/license_key **トピックにあります。**

Windows インストーラ Windows インストーラを使用する場合は、製品をインストールする際にライセンスキーを入力することができます。インストーラはライセンスキーをレジストリに追加します（後述）。

API 呼び出しで実行時にライセンスキーを適用 スクリプトやプログラムに行を追加して、ライセンスキーを実行時に設定するようにします。PDFlib オブジェクトをインスタンス化した直後に（すなわち、*PDF_new()* かそれと同等の呼び出しの後に）、*license* パラメタを設定する必要があります。具体的な文法は、使用するプログラミング言語によって異なります。

- C・Python の場合：

```
PDF_set_parameter(p, "license", "...あなたのライセンスキー ...")
```

- C++・Java・Ruby の場合：

```
p.set_parameter("license", "...あなたのライセンスキー ...")
```

- Perl・PHP の場合：

```
p->set_parameter("license", "...あなたのライセンスキー ...")
```

- RPG の場合：

```
c               callp     RPDF_set_parameter(p:%ucs2('license'):
c                                  %ucs2('...あなたのライセンスキー ...'))
```

- TCL の場合：

```
PDF_set_parameter $p, "license", "...あなたのライセンスキー ..."
```

ライセンスファイルの利用　実行時の呼び出しでライセンスキーを与えるのではなく、ライセンスキーをテキストファイルに入力しておくという方法もあります。次の形式に従ってください（PDFlib のディストリビューションにはすべて、ライセンスファイルテンプレート *licensekeys.txt* が入っているので、それをご利用いただくこともできます）。「#」キャラクタで始まる行は、注釈を内容としており、無視されます。2 行目は、ライセンスファイル自体のバージョン情報を内容としています：

```
# Licensing information for PDFlib GmbH products
PDFlib license file 1.0
PDFlib      8.0.3    ...あなたのライセンスキー ...
```

ライセンスファイル内には、複数の PDFlib GmbH 製品に対するライセンスキー群を、それぞれ別の行に記述することもできます。また、複数のプラットフォームに対するライセンスキー群を内容として持たせることにより、同一のライセンスファイルを複数のプラットフォームで共用することも可能です。ライセンスファイルは、下記の方法で設定することができます：

- *licensekeys.txt* という名前のファイルが、すべてのデフォルト位置で検索されます（11 ページ「デフォルトファイル検索パス」を参照）。
- *licensefile* パラメタを *set_parameter()* API 関数で設定することもできます：

  ```
  p.set_parameter($p, "licensefile", "/ファイルへの/パス/licensekeys.txt");
  ```

- ライセンスファイルを指し示す環境（シェル）変数を設定することもできます。Windows では、システムのコントロールパネルを使って、「**システム**」→「**詳細**」→「**環境変数**」を選択します。Unix では、以下のようなコマンドを適用します：

  ```
  export PDFLIBLICENSEFILE=/ファイルへの/パス/licensekeys.txt
  ```

- IBM i5/iSeries では、ライセンスファイルは下記のように指定できます（このコマンドは、スタートアッププログラム *QSTRUP* 内で指定することができ、すべての PDFlib GmbH 製品で動作します）：

  ```
  ADDENVVAR ENVVAR(PDFLIBLICENSEFILE) VALUE(<... パス ...>) LEVEL(*SYS)
  ```

レジストリにライセンスキー　Windows では、下記のレジストリキーにライセンスファイルの名前を書き込むという方法もあります：

```
HKLM\SOFTWARE\PDFlib\PDFLIBLICENSEFILE
```

または、下記のレジストリキーのうちのいずれか 1 つにライセンスキーを直接書き込むこともできます：

```
HKLM\SOFTWARE\PDFlib\PDFlib8\license
HKLM\SOFTWARE\PDFlib\PDFlib8\8.0.3\license
```

MSI インストーラは、インストール時に与えられたライセンスキーを、これらのエントリの末尾に書き込みます。

注 64 ビット Windows システム上でレジストリを手作業で扱う際の注意点：通常どおり、64 ビットの PDFlib バイナリは Windows レジストリの 64 ビットビューとともに動作し、64 ビットシステム上で動作する 32 ビットの PDFlib バイナリはレジストリの 32 ビットビューとともに動作します。32 ビット製品に対するレジストリキーを手作業で追加する必要があるときは、必ず 32 ビット版の *regedit* ツールを使用してください。これは「スタート」

→「ファイル名を指定して実行 …」のダイアログから下記によって起動することができます：

```
%systemroot%\syswow64\regedit
```

デフォルトファイル検索パス　Unix・Linux・Mac OS X・i5/iSeries システムでは、パス・ディレクトリ名を何も指定しなくても、デフォルトでいくつかのディレクトリでファイル群が検索されます。UPR ファイル（さらなる検索パスを含んでいる場合もある）を検索して読み込む前に、下記のディレクトリが検索されます：

```
<rootpath>/PDFlib/PDFlib/8.0/resource/cmap
<rootpath>/PDFlib/PDFlib/8.0/resource/codelist
<rootpath>/PDFlib/PDFlib/8.0/resource/glyphlst
<rootpath>/PDFlib/PDFlib/8.0/resource/fonts
<rootpath>/PDFlib/PDFlib/8.0/resource/icc
<rootpath>/PDFlib/PDFlib/8.0
<rootpath>/PDFlib/PDFlib
<rootpath>/PDFlib
```

Unix・Linux・Mac OS X では、***<rootpath>*** は、まず */usr/local* で、ついで HOME ディレクトリで置き換えられます。i5/iSeries では ***<rootpath>*** は空です。

ライセンス・リソースファイルのデフォルトファイル名　デフォルトで、下記のファイル名が、デフォルト検索パスディレクトリ群の中で検索されます：

```
licensekeys.txt          (ライセンスファイル)
pdflib.upr               (リソースファイル)
```

この機能を利用すると、環境変数やランタイムオプションを一切設定せずにライセンスファイルを扱うこともできます。

i5/iSeries・zSeries でのマルチシステムライセンスファイル　i5/iSeries・zSeries に対するライセンスキーはシステム固有ですので、複数のシステム間で共用することはできません。リソースの共有を実現し、複数のシステムで共用できる単一のライセンスファイルで作業を行うには、下記のライセンスファイル形式を用いて、複数のシステム固有キーを単一のファイル内に持たせることができます：

```
PDFlib license file 2.0
# Licensing information for PDFlib GmbH products
PDFlib      8.0.3   ...あなたのライセンスキー ...    ...マシン1のシリアル番号...
PDFlib      8.0.3   ...あなたのライセンスキー ...    ...マシン2のシリアル番号...
```

1 行目のバージョン番号が変わっている点と、複数のライセンスキーの後にそれぞれ、16 進 8 桁のシリアル番号（i5/iSeries の場合）か 16 進 4 桁の CPU ID（zSeries の場合）を付加している点に留意してください。

i5/iSeries でライセンスファイルを利用　i5/iSeries システムでは、ライセンスファイルは ASCII でエンコードされている必要があります（***asciifile*** パラメタ参照）。以下のコマンドは、環境変数 ***PDFLIBLICENSEFILE*** が適切なライセンスファイルを指し示すよう設定しています。

```
ADDENVVAR ENVVAR(PDFLIBLICENSEFILE) VALUE('/PDFLIB/7.0.3/licensefile.txt')
LEVEL(*SYS)
```

アップデートとアップグレード　アップデート（ある製品の古いバージョンからその同じ製品の新しいバージョンへの切り換え）またはアップグレード（PDFlib から PDFlib+PDI または PPS への、あるいは PDFlib+PDI から PPS への切り換え）を購入された場合は、そのアップデートかアップグレードに対して受け取った新しいライセンスキーを適用する必要があります。前の製品に対する古いライセンスキーはもう利用することができません。なお、ライセンスキーは、ある製品バージョンのなかのすべてのメンテナンスリリースに対して有効です。すなわち、ライセンスに関する限り、すべての *8.0.x* バージョンは同じに扱われます。

未ライセンスの評価版機能　いずれの機能も、ソフトウェアに対してライセンスキーを一切適用せずに完全に評価していただけます。しかし、ある特定製品に対する有効なライセンスキーの適用後は、それよりも高いカテゴリの機能は利用できなくなります。たとえば、有効な PDFlib のライセンスキーをインストールすると、PDI の機能は試用できなくなります。同様に、PDFlib+PDI のライセンスキーをインストールした後は、パーソナライゼーション機能（ブロック関数）は利用できなくなります。

ある製品に対するライセンスキーがすでにインストールされている時は、そのかわりにダミーのライセンス文字列「0」（数字のゼロ）を設定すれば、それよりも高い製品クラスの機能が試用できるようになります。そうすることにより、それ以前に無効にされた関数が有効になり、また、すべてのページを横断するデモスタンプが再び有効になります。

ライセンスオプション　PDFlib の 1 台または複数台のサーバ上での利用や、PDFlib を利用者自身の製品とともに再配布することに対しては、それぞれ異なったライセンスオプションがご利用いただけます。サポート契約やソースコード契約もご提供しております。ライセンスについての詳細情報とPDFlib購入申込フォームはPDFlibディストリビューションの中にあります。商用 PDFlib ライセンスのご購入に関心がある場合や、ご質問に関しては何でも、下記までご連絡ください。

PDFlib GmbH, Licensing Department
Franziska-Bilek-Weg 9, 80339 München, Germany
www.pdflib.com
電話 +49・89・452 33 84-0、FAX +49・89・452 33 84-99
ライセンスに関するお問い合わせ： *jp.sales@pdflib.com*
PDFlib ライセンスのサポート： *jp.support@pdflib.com*

1 導入

1.1 説明書とサンプルの紹介

PDFlib 製品の有効活用を支援するために、以下に挙げる資料を提供しています。

注 **Windows Vista・Windows 7 では、ミニサンプルとスタータサンプルはデフォルトで「Program Files」ディレクトリ内へインストールされます。Windows の保護スキームに従い、これらのサンプルによって作成される PDF 出力ファイルは「互換性ファイル」のもとでしか見えません。推奨される回避策 : サンプルの入ったフォルダをユーザーディレクトリへコピーします。**

すべての言語バインディング用のミニサンプル ミニサンプル（hello・image・pdfclock 等）は、どのパッケージにも、どの言語バインディングにも入っています。最小限のコードで、テキスト出力・画像・ベクタグラフィックの出力例を示しています。このミニサンプルを使えば簡単に、PDFlib が正しくインストールできているか試したり、PDFlib アプリケーションの書き方をさっと把握したりすることができます。

すべての言語バインディング用のスタータサンプル スタータサンプルは、すべてのパッケージに入っており、さまざまな言語バインディングで利用可能です。主要な用途で、汎用的な出発点として利用できます。簡単なテキスト・画像出力、テキストフロー・表組版、PDF/A・PDF/X 生成、その他さまざまな用途を網羅しています。このスタータサンプルを見れば、PDFlib 製品を使って特定の目的を達するための基本技法を知ることができます。このスタータサンプルを見てみることを強く推奨します。

PDFlib チュートリアル *PDFlib* チュートリアル（本説明書）は、どのパッケージにもある 1 個の PDF 文書で、重要なプログラミング概念をより詳しく、小さなコード例を用いて説明しています。コードをスタータサンプルより拡張していくにあたっては、この PDFlib チュートリアル内の関連する内容を知っておく必要があります。

注 **本 PDFlib チュートリアルでは、作成例はたいてい Java 言語で示されています（29 ページ「2 章 PDFlib の言語バインディング」の言語ごとの作成例と、別途その旨特記するいくつかの C 特有の作成例を除いて）。具体的な文法は言語ごとに異なりますが、PDFlib プログラミングの基本概念は、対応するどの言語バインディングについても同じです。**

PDFlib API リファレンス *PDFlib API* リファレンスは、どのパッケージにもある 1 個の PDF 文書で、PDFlib アプリケーションプログラミングインタフェース（API）を構成するすべての関数・引数・オプションを簡潔に記述しています。引数の詳細や対応するオプション、入力条件、その他従うべきプログラミング規則を調べるには、この PDFlib API リファレンスが絶対の規範です。これ以外の参照文書はどれも必ずしも完全ではありません。たとえば Javadoc の PDFlib API 一覧や、*php.net* での PDFlib 関数一覧は不完全です。PDFlib を使用する際はかならず、*PDFlib API* リファレンスの完全な内容を参照してください。

pCOS パスリファレンス pCOS インタフェースを利用すると、PDF 文書からさまざまな特性を取得することができます。pCOS は PDFlib+PDI・PPS に内蔵されています。pCOS パスリファレンスは、PDF 文書内の個々の対象を指し示してそれぞれの値を取得するために用いられるパス文法の説明を内容としています。

PDFlib クックブック　*PDFlib クックブック*は、いろいろな課題を達成するための PDFlib コードを集めています。クックブックの例はたいてい Java 言語で書かれていますが、簡単に他のプログラミング言語に合わせることができます。なぜなら PDFlib API は、対応するどの言語バインディングについてもほぼ等価だからです。この PDFlib クックブックには、サンプルプログラムがどんどん追加されています。Web 上で下記 URL で利用可能です:

```
www.pdflib.com/pdflib-cookbook/
```

pCOS クックブック　*pCOS クックブック*は、PDFlib+PDI と PPS に含まれている pCOS インタフェースのコードを集めています。pCOS インタフェースを利用すると、PDF 文書からさまざまな特性を取得することができます。下記 URL で利用可能です：

```
www.pdflib.com/pcos-cookbook/
```

TET クックブック　PDFlib TET（Text Extraction Toolkit）は、PDF 文書からテキストや画像を抽出するための別製品です。これを PDFlib+PDI と組み合わると、PDF 文書をその内容に応じて処理することができます。***TET* クックブック**は、TET のためのコードを集めています。この中には、TET と PDFlib+PDI を組み合わせた利用例が含まれています。たとえば、ページ上のテキストに応じて Web リンクやしおりを追加したり、検索単語をハイライトしたり、テキストに応じて文書を分割したり、目次を生成したりなどです。TET クックブックは下記 URL で利用可能です：

```
www.pdflib.com/tet-cookbook/
```

1.2 PDFlib プログラミング

PDFlib とは何か　PDFlib は、Adobe の Portable Document Format（PDF）形式のファイル生成を可能にする開発コンポーネントです。PDFlib は、開発したプログラムに対するバックエンドとして働きます。アプリケーションプログラマーの役割としては、ただ処理させたいデータを持ってくればよく、後の仕事は PDFlib が全部引き継いで、そのデータを視覚化した PDF 出力を生成します。そのデータを視覚的に表現する PDF 出力の生成処理は PDFlib がすべて請け負います。PDFlib を使えば、PDF の実際の内部構造を見ることなしに、さまざまなやり方で出力を組版することができます。PDFlib のバイナリパッケージでは、以下の製品がまとめて 1 個のバイナリに入っています。

- PDFlib。テキストやベクトルグラフィック・画像・ハイパーテキスト要素を含んだ PDF 出力を作成するために必要なあらゆる機能を持ちます。PDFlib は、一行・複数行のテキストや画像の配置、表の作成のための強力な組版機能をそなえています。
- PDFlib+PDI。PDFlib の全機能に加え、既存 PDF 文書内のページを取り込んだ出力が生成できる PDF 取り込みライブラリ（PDI）と、取り込み文書から任意の PDF オブジェクト（ページ上の全フォント一覧やメタデータ等さまざまな）を取得するための pCOS インタフェースを含んでいます。
- PDFlib Personalization Server（PPS）。PDFlib+PDI に加え、PDFlib ブロックに自動流し込みを行う機能も持ちます。ブロックとは、ページ上のプレースホルダであり、その中にテキストや画像や PDF ページを流し込めるものです。ブロックは、Adobe Acrobat 用 PDFlib Block Plugin（Mac 版・Windows 版あり）を用いて対話的に作成することができ、その中に、PPS を用いて自動的に流し込みを行います。このプラグインも PPS に含まれています。

PDFlib はどのように利用できるのか　PDFlib はさまざまなプラットフォーム上で利用可能です。Unix・Windows・Mac のいずれでも利用することができ、また、IBM i5/iSeries・zSeries といった EBCDIC ベースのシステムでも使えます。PDFlib は C 言語で書かれていますが、それ以外にもさまざまな言語やプログラミング環境から呼び出すことが可能です。このような言語や環境を言語バインディングと呼びます。PDFlib の言語バインディングは、インターネットとスタンドアロン両方の、現在広く使用されているあらゆるアプリケーション開発言語をカバーしています。PDFlib の API（アプリケーションプログラミングインタフェース）はわかりやすく、また、どのバインディングにおいても共通した形を持っています。現在、対応しているバインディングは以下のとおりです。

- COM。Visual Basic や、VBScript か JScript による Active Server Pages、Borland Delphi、Windows Script Host などの環境で利用されます
- ANSI C
- ANSI C++
- Cobol（IBM zSeries）
- Java。J2EE サーブレット・JSP を含みます
- .NET。C# や VB.NET、ASP.NET などの環境で利用されます
- PHP
- Perl
- Python
- REALbasic
- RPG（IBM i5/iSeries）
- Ruby。Ruby on Rails を含みます
- Tcl

PDFlib は何に使えるのか　PDFlib の利用目的としてまず挙げられるのは、自分のソフトウェア内や Web サーバ上で PDF を動的に作成することです。Web サーバ上で HTML ページを動的に生成するのと同じように、PDFlib プログラムを使って PDF を動的に生成させるようにすれば、その中にユーザーからの入力を反映させたり、Web サーバ上のデータベースから抽出したデータなどの動的データを反映させたりすることができます。PDFlib を用いたアプローチには以下のような利点を挙げることができます。

- PDFlib はデータ生成アプリケーションに直接組み込めます。
- この直接処理の採用により、PDFlib は PDF 生成手段として最速であり、Web 用途に最適です。
- PDFlib のスレッドセーフ性と堅牢なメモリ・エラー処理は、高パフォーマンスなサーバアプリケーションの運用に対応します。
- PDFlib はさまざまなオペレーティングシステム、さまざまな開発環境で利用できます。

PDFlib 活用の必要条件　PDFlib を使えば、PDF の実際の仕様を知らずに PDF を生成できます。PDFlib は PDF の技術的な中身をなるべくユーザーから隠していますが、PDF に関する一般的理解はあるにこしたことはありません。PDFlib を最大限活用しようとするアプリケーションプログラマーは、PostScript（ひいては PDF）の基本的グラフィックモデルをひととおり理解していることが理想です。とはいえ、アプリケーションプログラマーとして相応の経験があり、画面表示や印刷用の何らかのグラフィック API の取扱経験があるならば、PDFlib の API についてもそう障害なく会得できると思われます。

1.3 PDFlib/PDFlib+PDI/PPS 8 の新機能

PDFlib/PDFlib+PDI/PPS 8 と Block Plugin 4 の主な新機能・改良機能を以下に挙げます。これ以外にも多くの新機能があります:詳しくは PDFlib API リファレンスを参照してください。

1.3.1 Acrobat 9 対応の PDF 機能

PDFlib は、Acrobat 9（技術的には : PDF 1.7 Adobe 拡張レベル 3）に従ったさまざまな PDF 機能に対応しています。

外部グラフィック内容　PDF 文書のページは、他の PDF 文書のページへの参照を含むことができます。これを参照 XObject といいます。参照元ファイルはプレースホルダだけを含みます。たとえば画像の低解像度版や、実際のページ内容がないと記した単なる注記などです。この技法を使えば、繰り返される内容（トランザクション印刷等）を何度も送る必要がありません。参照 XObject は PDF/X-5g・PDF/X-5pg の重要な構成要素です。

レイヤーバリアント　レイヤーバリアント（レイヤー設定ともいいます）は、レイヤーのグループととらえることができます。レイヤーをグループ化することにより、印刷が安全になります。なぜなら、ユーザーがうっかり間違ったセットのレイヤーを有効・無効にしてしまう（たとえば特定の言語のレイヤーを有効にしながら、全言語共用の画像レイヤーを有効にし忘れる）ことを防げるからです。この理由からレイヤーバリアントは、PDF/X-4・PDF/X-5 規格ではレイヤー使用の基本となっています。

PDF ポートフォリオ　PDF ポートフォリオは、PDF やその他の文書を 1 個の実体にまとめたもので、これは Acrobat 9 で便利に利用することができます。ファイル添付の整頓にフォルダ構造を用いていなければ、できた PDF コレクションは Acrobat 8 でも利用可能です。PDFlib ではまた、PDF ポートフォリオ内のファイル添付の整頓を実現する定義済み・カスタムメタデータフィールドを使うこともできます。新しいアクションを用いれば、埋め込んだ文書のページへ直接飛ぶしおりを作ることも可能です。

地理参照付き PDF　GeoPDF は、ページ全体に対する、またはページ上の個別の地図に対する、地理参照情報を含みます。Acrobat 9 は、GeoPDF を操作するためのさまざまな機能を提供しています。PDFlib を使えば、画像およびページの一部または全体に対して地理空間参照データを割り当てることができます。

AES-256 暗号化と Unicode パスワード　PDFlib は、セキュリティ向上のため AES-256 暗号化に対応しています。AES-256 は、Acrobat 9 で導入されており、Unicode パスワードも使うことができます。

Acrobat 9 文書の取り込み　PDFlib+PDI・PPS で、Acrobat 9 文書を取り込んで処理することができます。pCOS インタフェースも Acrobat 9 文書を分析できます。

1.3.2 フォント処理とテキスト出力

複雑用字系のシェーピングと双方向テキスト組版　単純用字系（ラテン字等）は、キャラクタが左から右へ順番に置かれる用字系です。複雑用字系では、テキストのシェーピング（コンテキストに応じて適切なグリフ形を選ぶ）や、キャラクタの並べ替えや、テキストを右から左へ組むための追加処理が必要になります。PDFlib は、アラビア・ヘブライ・デーヴァナーガリー・タイ用字系を含むさまざまな用字系に対する複雑用字系出力が可能です。

予備フォント　予備フォントは、さまざまなフォントとエンコーディング関連の制約を扱うための強力なしくみです。フォントを混ぜて組み合わせたり、足りないグリフを他のフォントから持って来たり、エンコーディングを拡張したりすることが可能です。予備フォントは、合成フォントのデザインの相違に対応するために個別のグリフの大きさを自動的に調整することができます。

OpenType レイアウト機能　OpenType レイアウト機能は、フォントファイル内の追加テーブルの形で、OpenType フォントに賢さを追加します。これらのテーブルは、合字・スモールキャピタル・スウォッシュキャラクタといった高度なタイポグラフィ機能を記述します。また、半角・全角・プロポーショナルグリフ・異体字、その他さまざまな高度な日中韓テキスト出力が可能です。

文書間でフォントを保持　フォントと関連データを、生成文書の完了後もメモリに保持しておくことができます。これにより、次の文書でこのフォントを再度解析する必要がなくなり、フォントのサブセット化といった文書個別の処理だけが行われるので、パフォーマンスが向上します。

日中韓外字キャラクタのための SING フォント　外字という日本語の用語は、広く使われて（苗字・地名等で）いながらもどのエンコーディング規格にも含まれていないカスタムキャラクタを指します。Adobe の SING フォントアーキテクチャ（**グリフレット**）は、日中韓テキストにおける外字問題を解決します。PDFlib は、SING フォントと、関連する Microsoft の EUDC フォント（エンドユーザー定義フォント）の概念に対応しています。**予備フォント**機能を利用すれば、SING・EUDC フォントを既存フォントに合成することも可能です。

フォントエンジンを再設計　PDFlib のフォントエンジンが再設計されて最適化され、さまざまな Unicode・エンコーディング関連の利点が得られるとともに、全体的なパフォーマンス向上、メモリ使用量の低減を実現しました。この再設計により、いくつかの制約が取り除かれることができ、既存の機能の性能が拡張されています。たとえば、Type 1・Type 3 フォントで 256 個を超えるグリフを指定したり、スウォッシュキャラクタをグリフ名で指定したりすることができるようになりました。

画像クリッピングパスのまわりにテキストを回り込み　テキストフロー組版エンジンは、テキストを任意のパスに回り込ませ、また、取り込み TIFF・JPEG 画像のクリッピングパスを用いることもできます。このようにして、複数行テキストを画像に回り込ませることができます。

パス上のテキスト　テキストを、直線分・曲線・円弧を任意に混ぜた任意のベクトルパス上に配置することができます。このパスはプログラムで構築することも可能です。ある

いは、TIFF・JPEG 画像のクリッピングパスを抽出してテキストパスとして用いることもできます。

1.3.3 PDFlib Block Plugin と PDFlib Personalization Server（PPS）

PDFlib Block Plugin は、PDFlib Personalization Server（PPS）でブロック流し込み（パーソナライゼーション）を行うための PDF 文書の作成に使います。

PPS ブロック処理を Acrobat でプレビュー　このプラグインは、PPS によるブロック流し込み処理のプレビューを Acrobat で直接生成することができます。迅速なプレビューにより、デザイナーは、自分のブロック文書をサーバへ送って処理させる前に、その文書への PPS 流し込みの結果をすぐに吟味することができます。プレビュー PDF は、デバッグと開発の支援として、可能なエラーメッセージを持ったしおり・レイヤー・注釈を内容として持ちます。このプレビュー機能は開発サイクルを速めるとともに、PDFlib の諸機能を試すためのインタラクティブなテストフレームワークとして利用することもできます。

ブロックコンテナの PDF/A・PDF/X 状態を複製　PDF/A または PDF/X 文書に基づくブロックプレビューを生成する際には、Block Plugin は、規格同定・出力インテント・メタデータ等、その規格の関連するあらゆる側面を複製することができます。PDF/A・PDF/X 複製モードにおけるブロック流し込み操作が、選ばれている規格に違反するであろう場合は（たとえばデフォルト画像が RGB 色空間を用いていながら、文書に適切な出力インテントが含まれていない場合等）、エラーメッセージが表示されます。このため、ユーザーは起こりうる規格違反をワークフローの非常に早い段階で捕捉することができます。

再設計されたユーザーインタフェースとスナップグリッド　PDFlib Block Plugin のユーザーインタフェースは、さまざまな既存の、そして新しいブロックプロパティの利用を実現するために再構築されました。

新機能スナップグリッドは、ブロックをデザインラスタに従ってすばやくレイアウトするのに有用です。

ブロックプロパティを追加　Block Plugin と PPS に新しいブロックプロパティがさらに追加されました。たとえば、ブロック内に配置されるテキスト・画像・PDF 内容の透過を指定することができます。

PDFlib 8 の諸機能をブロックで活用　複雑用字系のテキスト出力やOpenTypeレイアウト機能といった PDFlib 8 の関連する新機能を、ブロックプロパティで直接活用できます。たとえば、ブロックにアラビア語やヒンディーのテキストを流し込むことが可能です。

1.3.4 その他の重要な機能

再利用可能なパスオブジェクト　パスオブジェクトは、いずれのページからも独立に構築して、描線・塗り・クリッピングのために複数回用いることができます。パスオブジェクトはまた、回り込み形状として用いたり（テキストを不規則な形状の領域内に組む）、テキストをパス上に配置するために用いたりすることができます。

PDF/X-4・PDF/X-5　PDFlib は、グラフィックアート業界のための PDF/X-4・PDF/X-5 規格に従った出力を生成します。従来の PDF/X-1・PDF/X-3 規格と異なり、これらはより新しい PDF バージョンに基づいており、透過とレイヤーを許容します。PDF/X-4p・PDF/X-5pg では、外部的に参照された ICC プロファイルを出力インテントとして用いることができます。PDF/X-5g・PDF/X-5pg は、外部グラフィック内容の使用に対応しています。

TIFF・PNG 画像のアルファチャンネル　PDFlib は、TIFF・PNG 画像を取り込む際、画像の透過（アルファチャンネル）に従います。アルファチャンネルは、なめらかな遷移を生み出し、画像を背景に溶け込ませるのに利用することができます。

JBIG2 圧縮画像　JBIG2 は、白黒画像のための高効率な画像圧縮形式です。PDFlib は、単ページ・複数ページの JBIG2 画像を取り込み、その圧縮の利点を生成 PDF 出力内で維持します。

圧縮オブジェクトストリーム・相互参照ストリーム　PDFlib は、圧縮オブジェクトストリームと相互参照ストリームを生成します。これらの方式は、生成 PDF 文書全体のファイルサイズを低減するとともに、旧来の相互参照テーブルを持つ PDF に課せられる従来の 10 GB の制約を超えて飛ぶことを PDF 文書で可能にします。10 GB はとんでもなく大きなデータと思われるかもしれませんが、トランザクション印刷の分野ではこの限界に達する応用例が増えてきています。PDFlib ユーザーがこの領域の PDF 文書を生成したいシナリオは今後ますます増えていくと我々は予測しています。

内蔵 PANTONE® Goe™ カラーライブラリ　PDFlib は、コート紙・上質紙用の新色 2058 色と新カラー命名方式を持つ PANTONE® Goe™ カラーライブラリに対応しています。この Goe™ カラーライブラリは Pantone, Inc. によって 2008 年に導入されたものです。

既存関数の改良　以下に、PDFlib 8 における既存機能の最も重要な改良を挙げます：

- 画像の詳細を *PDF_info_image( )* で取得
- PPS と Block Plugin：新しい PDFlib の諸機能を PDFlib ブロックを通じて利用可能とする追加のブロックプロパティ
- Unix システム上での Unicode ファイル名
- 表組版機能：パスオブジェクト・注釈・フォームフィールドを表セル内に配置
- テキストフロー：追加の組版制御オプション、高度な言語個別の改行
- 影付きテキスト
- 取り込み画像内の XMP メタデータを保持
- *PDF_info_font( )* におけるさまざまな改良
- しおり作成のための追加のオプション
- C++ バインディングにおける設定可能な文字列データ型（***wstring*** で Unicode 対応等）

1.4 PDFlib の機能一覧

表 1.1 に、PDF lib の主な PDF 生成機能を挙げます。PDFlib 8 の新機能・改良機能には注釈を付しています。

表 1.1 PDFlib の機能一覧

分類	機能
PDF 出力	PDF 文書をディスクファイルまたは（Web サーバで）直接メモリ上へ生成
	大容量出力、任意の PDF ファイルサイズ（10 GB 超も可）
	ページの一時停止 / 再開・挿入機能で、ページを順番によらず生成
各種 PDF	PDF 1.3 ～ PDF 1.7ext3[1]（Acrobat 4 ～ 9）。ISO 32000-1（＝ PDF 1.7）も
	線形化（Web 最適化）PDF による、Web 上でのバイトサービング
	タグ付き PDF によるアクセシビリティ・リフロー
	マーク付き内容によるアプリケーション独自データ・代替テキストをタグ付けなしで追加[1]
ISO 規格	ISO 15930：PDF/X。グラフィックアート業界向け[1] ISO 19005：PDF/A。アーカイビング用 ISO 32000：PDF 1.7 の標準化バージョン[1]
グラフィック	一般的な基本ベクトルグラフィック要素：線・曲線・円弧・楕円[1]・矩形等
	スムーズシェーディング（カラーブレンド）・パターン塗り / 描線
	透過（不透明）・ブレンドモード
	外部グラフィック内容（参照 XObject）による可変データ印刷[1]
	再利用可能パスオブジェクト、クリッピングパスを画像から取り込み[1]
レイヤー	表示を切り替えられるオプショナルなページ内容
	注釈・フォームフィールドをレイヤー上に配置可
	レイヤーのロック、表示倍率による自動表示など
	PDF/X-4・PDF/X-5 のレイヤーバリアント[1]（レイヤーのグループ化で誤印刷防止）
フォント	TrueType（TTF・TTC）・PostScript Type 1 フォント（PFB・PFA、Mac では LWFN も）
	PostScript/TrueType のアウトラインを持つ OpenType フォント（TTF・OTF）
	欧文・日中韓テキスト出力のための何ダースもの OpenType レイアウト機能に対応。例：合字・スモールキャピタル・オールドタイプ数字・スウォッシュキャラクタ・簡体 / 繁体・縦書き字体[1]
	Windows・Mac システムにインストールされているフォント（「ホストフォント」）の直接利用
	あらゆる種類のフォントの埋め込み。TrueType・OpenType・Type 3 フォントのサブセット化
	ユーザー定義（Type 3）フォントによるビットマップフォントやカスタムロゴ
	EUDC・SING[1] フォント（グリフレット）で、日中韓外字キャラクタ
	予備フォント（足りないグリフを補助のフォントから取得）[1]
	フォントを文書間で保持し、パフォーマンス向上[1]

表 1.1 PDFlib の機能一覧

分類	機能
テキスト出力	さまざまなフォントでテキスト出力。テキストに下線・上線・取り消し線
	フォント内のグリフを数値・Unicode 値・グリフ名[1] で指定可能
	カーニングで、文字間隔を改善
	テキストを太字化・斜体化・影付き[1]
	パス上にテキストを生成[1]
	日中韓フォントでプロポーショナル幅[1]
	見つからないグリフの代替設定
国際化	ページ内容・インタラクティブ要素・ファイル名[1] に Unicode 文字列。UTF-8・UTF-16・UTF-32 形式
	多様な 8 ビット・レガシマルチバイト日中韓エンコーディング（Shift-JIS・Big5 等）に対応
	システムからコードページを取得（Windows・IBM i5/iSeries・zSeries）
	日本語・中国語・韓国語テキストの標準・カスタム日中韓フォント・CMap
	日本語・中国語・韓国語テキストの縦書き
	アラビア語・タイ語・デーヴァナーガリー等の複雑用字系のキャラクタシェーピング[1]
	アラビア語・ヘブライ語等の右書き用字系の双方向テキスト組版[1]
	Unicode 情報を PDF 内に埋め込み、Acrobat での適切なテキスト抽出を促す
画像	BMP・GIF・PNG・TIFF・JBIG2[1]・JPEG・JPEG 2000[1]・CCITT ラスタ画像の埋め込み
	画像ファイル形式の自動検出
	画像情報の取得（ピクセルサイズ・解像度・ICC プロファイル・クリッピングパスなど）[1]
	TIFF・JPEG 画像内のクリッピングパスを解釈
	TIFF・PNG 画像内のアルファチャンネル（透過）を解釈[1]
	画像マスク（透過画像に色を適用）、スポットカラーによる画像着色
色	グレースケール・RGB（数値・16 進列・HTML 色名）・CMYK・CIE $L^*a^*b^*$ カラー
	PANTONE® カラー（PANTONE® Goe™ も）[1]・HKS® カラー対応内蔵
	ユーザー定義スポットカラー
カラーマネジメント	ICC プロファイルによる ICC 準拠カラー。ICC 4 プロファイル対応[1]
	テキスト・図形・ラスタ画像のレンダリングインテント
	デフォルトグレー・RGB・CMYK カラースペースで、デバイス独自カラーを再対応付け
	PDF/A・PDF/X の出力インテントに ICC プロファイル
アーカイビング	PDF/A-1a・PDF/A-1b（ISO 19005-1）
	PDF/A-1 の XMP 拡張スキーマ
グラフィックアート	PDF/X-1a・PDF/X-3・PDF/X-4[1]・PDF/X-4p[1]・PDF/X-5p[1]・PDF/X-5pg[1]（ISO 15930）

表 1.1 PDFlib の機能一覧

分類	機能
	埋め込み / 外部参照[1] 出力インテント ICC プロファイル
	PDF/X-5p・PDF/X-5pg の外部グラフィック内容（参照ページ）[1]
	取り込み画像の OPI 1.3・OPI 2.0 情報を生成
	分版情報（PlateColor）
	テキストのヌキ・ノセなどの設定
テキストフロー組版	テキストを、1 個ないし複数の矩形内に、または任意形状領域内に組版。ハイフネーション（ユーザー定義ハイフネーション位置が必要）・フォント / 色変更・揃え方式・タブ・リーダ・制御コマンドを指定可能。テキストを画像に回り込み
	言語独自処理で高度な改行
	柔軟な画像貼り付け・組版
	画像または画像のクリッピングパス[1] にテキストを回り込み
表組版	さまざまなユーザー設定に従って表行・表列のサイズを自動計算して配置する表組版機能。複数ページにわたる表も可能。
	表セル内に、一行 / 複数行テキスト・画像・PDF ページ・パスオブジェクト・注釈・フォームフィールドを配置可能
	表セルを、枠線・背景色オプションを指定して組版可能
	柔軟なスタンプ機能
	貼り付け画像等各種オブジェクトの座標を参照する範囲枠の概念
セキュリティ	RC4（40/128 ビット）または AES 暗号化アルゴリズム（128/256[1] ビット）で PDF 出力を暗号化
	Unicode パスワード[1]
	権限設定の指定（印刷不可・コピー不可等）
インタラクティブ要素	フォームフィールドの生成。すべてのフィールドオプションと JavaScript を設定可能
	フィールド群からバーコードを生成
	しおり・注釈・ページを開く / 閉じる等各種イベントに対するアクションの生成
	しおりの生成。さまざまなオプション・制御を設定可能
	ページ遷移効果（シェード・モザイク等）
	PDF リンク・起動リンク（他の文書種別）・Web リンク等、あらゆる種類の PDF 注釈を生成可能
	リンク・しおり・文書を開くアクションに名前付き移動先
	ページラベル（ページのシンボリック名）の生成
マルチメディア	PDF 内に 3D アニメーションを埋め込み
GeoPDF	地理空間参照情報を持つ PDF の生成[1]
タグ付き PDF	タグ付き PDF・構造情報の生成による、アクセシビリティ・ページリフロー・内容再利用改善。リンク等各種注釈を文書構造へ入れ込み可能
メタデータ	文書情報：標準フィールド（タイトル・サブタイトル・作成者・キーワード）・ユーザー定義フィールド

表 1.1 PDFlib の機能一覧

分類	機能
	文書情報フィールドから、またはクライアントが与えた XMP ストリームから XMP メタデータを生成
	TIFF・JPEG・JPEG 2000 画像内の XMP 画像メタデータを処理 [1]
プログラミング	Cobol・COM・C・C++[1]・Java・.NET・Perl・PHP・Python・REALbasic・RPG・Ruby・Tcl の言語バインディング
	仮想ファイルシステムによるインメモリ処理（データベースの画像処理など）

1. PDFlib 8 の新機能 / 大幅改良機能

1.5 PDFlib+PDI の追加機能

PDFlib+PDI・PPS には、表 1.1 に示した基本的な PDF 生成機能に加えて、表 1.2 に挙げる機能があります。

表 1.2 PDFlib+PDI の追加機能一覧

分類	機能
PDF 入力（PDI）	既存 PDF 文書からページを取り込み
	PDF 1.7 拡張レベル 3（Acrobat 9）までのすべてのバージョンの PDF を取り込み可能[1]
	すべての PDF 標準暗号化アルゴリズムによる暗号化文書を取り込み可能（要マスターパスワード）[1]
	取り込みページの情報を取得[1]
	取り込みページのページ寸法を複製（BleedBox・TrimBox・CropBox 等）[1]
	複数の取り込み PDF 文書にわたる冗長なオブジェクト（同一フォント等）を削除
	異常な入力 PDF 文書を修復[1]
	取り込み PDF 文書から PDF/A・PDF/X 出力インテントを複製
pCOS インタフェース	pCOS インタフェースで、取り込み PDF 文書の詳細を取得[1]

1. PDFlib+PDI 8 の新機能 / 大幅改良機能

1.6 PPS の追加機能

表 1.3 に、PDFlib Personalization Server（PPS）でのみ利用可能な機能を挙げます（表 1.1 に示した基本的な PDF 生成機能と表 1.2 に示した PDF 取り込み機能に加えて）。

表 1.3 PDFlib Personalization Server（PPS）の追加機能一覧

分類	機能
可変データ処理（PPS）	PDFlib ブロックにテキスト・画像・PDF データを流し込んで PDF をパーソナライズ
PDFlib Block Plugin	PDFlib Block Plugin で、Windows・Mac 版 Acrobat 上で PDFlib ブロックを対話的に作成
	再設計されたユーザーインタフェース[1]
	PPS ブロック流し込みを Acrobat 上でプレビュー[1]
	Acrobat 上でブロックを対話的に作成・編集する際のスナップグリッド[1]
	ブロックコンテナの PDF/X・PDF/A プロパティを複製[1]
	PDF フォームフィールドを PDFlib ブロックに変換して、自動流し込み可能に
	テキストフローブロックを連結して、ブロックであふれたテキストを次のブロックへ
	ブロックプラグインに PANTONE®・HKS® スポットカラー名一覧を内蔵[1]

1. PDFlib Personalization Server 8 の新機能 / 大幅改良機能

1.7 製品別機能一覧

表 1.4 は、PDFlib ファミリの各種製品の機能一覧です。

表 1.4 製品別機能一覧

機能	API 関数・引数・オプション	PDFlib	PDFlib+PDI	PPS
基本的な PDF 生成	以下に挙げるもの以外すべて	○	○	○
線形化（Web 最適化）PDF	PDF_end_document( ) の linearize オプション	○[1]	○	○
PDF の最適化（効果的でないクライアントコードと、最適化されていない取り込み PDF 文書に対してのみ意味を持ちます）	PDF_end_document( ) の optimize オプション	○[1]	○	○
参照 PDF、PDF/X-5g・PDF/X-5pg	PDF_begin_template_ext( )・PDF_open_pdi_page( ) の reference オプション	○[1]	○	○
ポートフォリオ作成のために PDF 文書を解析	PDF_add_portfolio_file( ) の password オプション	○[1]	○	○
PDF 取り込み（PDI）	すべての PDI 関数	—	○	○
既存 PDF から pCOS で情報を取得	すべての pCOS 関数	—	○	○
ブロックによる可変データ処理・パーソナライゼーション	ブロック流し込みのためのすべての PPS 関数	—	—	○
Acrobat 用 PDFlib Block Plugin	PPS で利用するための PDFlib ブロックを対話的に作成	—	—	○

1. この機能には内部的に PDI を必要とするので、PDFlib ソースコードパッケージでは利用できません

2 PDFlib の言語バインディング

注 スタータサンプルに目を通されることを強く推奨します。すべての PDFlib パッケージに入っています。アプリケーション開発の出発点として有用です。PDFlib プログラミングの重要な点を多数網羅しています。

2.1 各言語バインディングにおけるデータ型

この説明書では、さまざまな言語バインディングにおける関数 / メソッドのプロトタイプを示しています。言語バインディングによる主な違いは、オブジェクト指向言語バインディングでは PDFlib メソッドは名前に *PDF_* 接頭辞がないのに対し、それ以外の言語バインディングではすべての関数名に *PDF_* 接頭辞がついていることです。また、非オブジェクト指向言語バインディングではすべての関数に 1 番目の引数として PDF コンテキスト引数を与える必要があります。これに対し、オブジェクト指向言語バインディングは言語ラッパが作成するオブジェクトの中に PDF コンテキストを隠しています。

表 2.1 に、すべての言語バインディングにおける PDF 文書型と文字列文書型の使い方を挙げます。テキストと文字列の扱いについて詳しくは ***PDFlib* チュートリアル**を参照してください。データ型 *integer*・*long*・*double* については、すべてのバインディングでこれらの型には明らかな対応づけがありますので、ふれていません。

表 2.1 各言語バインディングにおけるデータ型一覧

言語バインディング	p 引数と PDF_ 接頭辞？	文字列データ型	バイナリデータ型
C	あり	const char * [1]	const char *
C++	なし	デフォルトで std::wstring[2]	const char *
Cobol	あり[3]	STRING	STRING
Java	なし	String	byte[ ]
Perl	なし	string	string
PHP	なし	string	string
Python	あり	string	string
RPG	あり	string、ただし x'00' を加える必要あり	data
Ruby	なし	string	string
Tcl	あり	string	byte 配列

1. C 言語の NULL 文字列値と空文字列は同等と見なされます。
2. C++ API を、std::basic_string テンプレートのインスタンス化を通じてカスタマイズすることもできます。たとえば、API を std::string に切り替えて、古いアプリケーションとの互換を実現することができます。あるいはユーザー定義のデータ型を、API 内で用いられる文字列型の基盤として用いることもできます（36 ページ「2.5 C++ バインディング」参照）。
3. Cobol プログラムでは PDFlib 関数の短縮名を用いる必要があります。

2.2 Cobol バインディング

Cobol 用 PDFlib API では、標準の C 名は使用できず、かわりに省略形の関数名を用います。その短縮関数名はここには記さないので、相互参照一覧（*xref.txt*）を別途参照してください。たとえば *PDF_load_font()* のかわりに短縮形 *PDLODFNT* を用いる必要があります。

Cobol で書かれた PDFlib のクライアントは PDFLBCOB オブジェクトに静的にリンクされています。このオブジェクトは、PDLBDLCB Load Module（DLL）を動的にロードします。この DLL は、PDNEW（*PDF_new()* に対応）への最初の呼び出しの際に PDFlib Load Module（DLL）を動的にロードします。PDFlib の内部構造がメモリに新規に割り当てられ、そのインスタンスハンドルが *P* 引数に格納されます。この引数は、以後呼び出しごとに指定する必要があります。

PDLBDLCB ロードモジュールは、8 文字の Cobol 関数とコア PDFlib ルーチンとの間のインタフェースを提供するものです。このモジュールはまた、PDFlib の非同期な例外処理と、Cobol が期待する一本槍な「関数ごとに戻り値を調べる」方式との間の対応づけを提供するものでもあります。

注 PDLBDLCB と PDFLIB が、STEPLIB の使用を通じて COBOL プログラムから利用可能になっている必要があります。

データ型 PDFlib API リファレンス内で使用されている各種データ型は、以下の例のように Cobol データ型に対応づける必要があります。

```
05  PDFLIB-A4-WIDTH         USAGE COMP-1 VALUE 5.95E+2. // float
05  WS-INT                  PIC S9(9) BINARY.           // int
05  WS-FLOAT                COMP-1.                     // float
05  WS-STRING               PIC X(128).                 // const char *
05  P                       PIC S9(9) BINARY.           // long *
05  RETURN-RC               PIC S9(9) BINARY.           // int *
```

PDFlib API に渡される Cobol 文字列はすべて、LOW-VALUES (NULL) 終端子のために 1 バイト多く格納領域を持たせて定義する必要があります。

戻り値 PDFlib API のさまざまな関数の戻り値は、参照渡しの *ret* という引数を追加しておくことによって、そこに格納されてきます。各関数呼び出しの結果がこの引数に代入されるのです。戻り値がゼロの場合は、その関数呼び出しが正常に実行されたことを意味します。それ以外の値は何らかのエラーの発生を示しており、その場合 PDF の生成は続行できません。

戻り値を何も返さない関数（戻り値型 void の C 関数）については、この追加引数は用いられません。

エラー処理 PDFlib の例外処理は Cobol 言語バインディングでは利用できません。そのかわりすべての API 関数は、エラーを示す戻りコード（*rc*）引数という追加の引数に対応しています。*rc* 引数は参照渡しであり、問題発生の通知に用いられます。非ゼロ値は関数呼び出し失敗を示します。

2.3 COM バインディング

（この節は、PDFlib チュートリアルの COM・.NET・REALbasic エディションにのみ掲載しています）

2.4 C バインディング

PDFlib は、C 言語にいくつかの C++ モジュールを加えたもので書かれています。PDFlib の C バインディングを利用するには、静的または共有ライブラリ（Windows・MVS 上の DLL）を使うことができ、中心となる PDFlib インクルードファイル ***pdflib.h*** を PDFlib のクライアントのソースモジュールにインクルードする必要があります。あるいは、***pdflibdl.h*** を用いて PDFlib DLL を実行時に動的に読み込ませることもできます（詳しくは次項参照）。

注 PDFlib の C バインディングを用いたアプリケーションは、C++ コンパイラでリンクを行う必要があります。PDFlib ライブラリでは、C++ で実装された部分をいくつかインクルードしているためです。C リンカを用いると、必要な C++ サポートライブラリに対してアプリケーションが明示的にリンクされていない限り、未解決の外部参照エラーが出る可能性があります。

実行時に読み込まれる DLL として PDFlib を利用　たいていのクライアントでは、PDFlib を、静的に結合したライブラリとして用いるか、リンク時に結合されるダイナミックライブラリとして用いると考えられますが、それ以外の利用法として、PDFlib DLL を実行時に読み込み、全 API 関数へのポインタを動的に取得することもできます。この方法は特に、PDFlib DLL を必要時にのみ読み込みたい場合に有用であり、また、MVS 上でも有用です。なぜなら MVS では通例、ライブラリは PDFlib に明示的にリンクされず、DLL として実行時に必要に応じて読み込まれるからです。PDFlib ではこのような動的な利用法を可能にするためのしくみを特に設けてあります。それは以下の方法にしたがって利用します。

- ***pdflib.h*** のかわりに ***pdflibdl.h*** をインクルードする。
- ***PDF_new()*** ・ ***PDF_delete()*** のかわりに ***PDF_new_dl()*** ・ ***PDF_delete_dl()*** を用いる。
- ***PDF_TRY()*** ・ ***PDF_CATCH()*** のかわりに ***PDF_TRY_DL()*** ・ ***PDF_CATCH_DL()*** を用いる。
- それ以外のすべての PDFlib 呼び出しについては関数ポインタを用いる。
- ***PDF_get_opaque()*** は使ってはいけません。
- 追加のモジュール ***pdflibdl.c*** をコンパイルして、アプリケーションをそれにリンクさせる。

注 PDFlib DLL の実行時読み込みは、限られたプラットフォームでのみ対応しています。

C でのエラー処理　PDFlib は、try ～ catch 節による構造化例外処理に対応しています。ですので、C と C++ のクライアントでは、PDFlib で発生した例外をキャッチして、その例外に対して適切に反応することができます。catch 節でクライアントは、問題の正しい性質を記した文字列と、一意な例外番号と、例外を発生させた PDFlib API 関数の名前とを得ることができます。例外処理を持つ PDFlib C クライアントプログラムの一般的構造は次のようになります。

```
PDF_TRY(p)
{
        ...いろいろなPDFlib命令...
}
PDF_CATCH(p)
{
        printf("PDFlib exception occurred in hello sample:\n");
        printf("[%d] %s: %s\n",
            PDF_get_errnum(p), PDF_get_apiname(p), PDF_get_errmsg(p));
        PDF_delete(p);
        return(2);
```

```
}

PDF_delete(p);
```

PDF_TRY/PDF_CATCH はトリッキーなプリプロセッサマクロとして実装されています。このうちのどちらかをうっかり入れ忘れると、コンパイラから出るエラーメッセージは原因理解が困難なものになってしまう可能性があります。マクロは上記と正確に同じように用いるようにしてください。*TRY*節と*CATCH*節の間には一切コードを入れてはいけません（*PDF_CATCH()* 以外）。

catch 節の重要な仕事は、*PDF_delete()* と PDFlib オブジェクトへのポインタとを用いて PDFlib の内部構造を解放することです。*PDF_delete()* は必要に応じて出力ファイルも閉じます。致命的例外の後は PDF 文書は使用不能であり、不完全・不整合な状態で取り残されます。もちろん、例外発生時にどのような動作をさせるのが適切かはアプリケーションに依存します。

C と C++ のクライアントで例外をキャッチしない場合、例外発生時のデフォルト動作は、適切なメッセージを標準エラー出力などに出力して、致命的エラーならば抜けるというものです。PDF 出力ファイルは未完成の状態で残されます！ これはライブラリのルーチンとして適当とは言いかねるので、エラー処理が重要なアプリケーションでは PDFlib の例外処理機能を活用することを強く推奨します。ユーザー定義の catch 節ではたとえば、エラーメッセージを GUI ダイアログボックスに表示し、停止以外の処置をとることができるでしょう。

volatile 変数　*PDF_TRY()* ブロックと *PDF_CATCH()* ブロックの両方で使う変数については特に注意が必要です。1 つのブロックから別のブロックへ制御が移ることについてコンパイラは知らないので、このような場合には不適切なコードを生成してしまう可能性があります（レジスタ変数の最適化等）。幸い、こうした問題を避けるには次のような簡単な規則があります。

注 *PDF_TRY()* ブロックと *PDF_CATCH()* ブロックの両方で使う変数は、volatile 宣言するべきです。

volatile キーワードを使うと、変数に最適化（危険をはらむ）を適用してはいけないということをコンパイラに知らせることができます。

try/catch ブロックの入れ子と例外の再 throw　*PDF_TRY()* は、任意の深さの入れ子にすることが可能です。エラー処理を入れ子にした場合、以下のように内側の catch ブロックで例外を再 throw することによって外側の catch ブロックをアクティブにすることができます。

```
PDF_TRY(p)                          /* 外側のtryブロック */
{
        /* ... */

        PDF_TRY(p)                  /* 内側のtryブロック */
        {
            /* ... */
        }
        PDF_CATCH(p)                /* 内側のcatchブロック */
        {
            /* エラーをクリーンアップ */
            PDF_RETHROW(p);
        }
```

```
        /* ... */
}
PDF_CATCH(p)                        /* 外側のcatchブロック */
{
        /* さらにエラーをクリーンアップ */
        PDF_delete(p);
}
```

内側のエラーハンドラで *PDF_RETHROW()* を呼び出すと、プログラムの実行はただちに外側の *PDF_CATCH()* ブロックの最初のステートメントへ移ります。

try ブロックを未完了で抜ける　*PDF_TRY()* ブロックを、たとえば return ステートメントによって抜けたい場合、すなわちそれに対応する *PDF_CATCH()* マクロへの呼び出しをバイパスする場合は、以下のように *PDF_EXIT_TRY()* マクロを使って例外機構に通知する必要があります。このマクロから try ブロック末尾までの間、他のライブラリ関数は一切呼び出してはいけません。

```
PDF_TRY(p)
{
        /* ... */

        if (error_condition)
        {
            PDF_EXIT_TRY(p);
            return -1;
        }
}
PDF_CATCH(p)
{
        /* エラーをクリーンアップ */
        PDF_RETHROW(p);
}
```

C でのメモリ管理　最大限の柔軟性を持たせるため、PDFlib 内部のメモリ管理ルーチン（標準の C の *malloc/free* に基づいている）は、クライアントが作成した外部のプロシージャで置き換えることができます。このプロシージャは、PDFlib 内部のすべてのメモリ割り当てとメモリ解放の際に呼び出されます。メモリ管理ルーチンをインストールするには、*PDF_new2()* を呼び出せば、PDFlib 内部のルーチンのかわりに用いられるようになります。*PDF_new2()* の引数には以下の全ルーチンを指定（あるいは何も指定しない）する必要があります。

- 割り当てルーチン
- 解放（割り当て解除）ルーチン
- 再割り当てルーチン。以前に割り当てルーチンで割り当てられたメモリブロックを拡げるために用いられます。

こうしたルーチンは、標準の C の *malloc/free/realloc* と同じ意味を持たなければなりませんが、実装内容は任意に選ぶことができます。どのルーチンにも、呼び出し側 PDFlib オブジェクトへのポインタが与えられます。この規則の唯一の例外として、割り当てルーチンへの一番最初の呼び出しは PDF ポインタとして NULL を与えます。そのため、クライアントで作成するメモリ割り当てルーチンは、NULL の PDF ポインタを扱えるように作っておく必要があります。

PDF_get_opaque() 関数を用いると、アプリケーション固有の void ポインタを PDFlib オブジェクトから取り出すことができます。この void ポインタ自体は *PDF_new2()* への呼び出しでクライアントによって指定されます。void ポインタは、マルチスレッドのアプリケーションが PDFlib オブジェクト内のスレッド固有データやクラス固有データへのポインタを保持しておいてメモリ管理やエラー処理に利用したい場合に有用です。

C 言語バインディングでの Unicode C 言語バインディングのクライアントでは、テキストが null キャラクタを含む可能性がある場合には標準のテキスト出力関数（*PDF_show()*・*PDF_show_xy()*・*PDF_continue_text()*）を使用しないよう注意する必要があります。そのような場合にはかわりに関数 *PDF_show2()* 等を用いなければならず、また、文字列長は別途与える必要があります。これ以外の言語バインディングではこのことに注意する必要はありません。なぜなら PDFlib 言語ラッパがそもそも内部的に *PDF_show2()* 等を呼び出すようになっているからです。

2.5 C++ バインディング

注 C++ で書かれた .NET アプリケーションについては、C++ バインディングを通じてではなく、PDFlib .NET DLL を直接利用することを推奨します（例外として、クロスプラットフォームアプリケーションの場合は C++ バインディングを使う必要があります）。PDFlib ディストリビューションには、この組み合わせを演示する .NET CLI で使う C++ サンプルコードがあります。

pdflib.h C ヘッダファイルに加えて、C++ のためのオブジェクト指向のラッパが PDFlib クライアントのために提供されています。これは *pdflib.hpp* ヘッダファイルを必要とします。このヘッダファイルは *pdflib.h* をインクルードしています。*pdflib.hpp* はテンプレートベースの実装を内容として持っていますので、対応する *.cpp* モジュールは必要ありません。C++ オブジェクトラッパを用いると、すべての PDFlib 関数名の *PDF_* 接頭辞は、よりオブジェクト指向的な表現に置き換わります。

実行時に読み込まれる DLL として PDFlib を利用 C 言語バインディングと同様、C++ バインディングでは、PDFlib を自分のアプリケーションに実行時に結合させることもできます（32 ページ「実行時に読み込まれる DLL として PDFlib を利用」を参照）。動的読み込みは、*pdflib.hpp* をインクルードするアプリケーションモジュールをコンパイルする際に、下記のようにして有効にすることができます：

```
#define PDFCPP_DL    1
```

これに加えて、追加モジュール *pdflibdl.c* をコンパイルして、できるオブジェクトファイルに自分のアプリケーションをリンクさせる必要があります。動的読み込みの詳細はこの PDFlib オブジェクト内に隠蔽されていますので、これは C++ API に影響を与えません：すべてのメソッド呼び出しは、動的読み込みが有効にされているか否かにかかわらず同じに見えます。

注 実行時の DLL 読み込みは、一部のプラットフォームでのみ利用できます。

C++ での文字列処理 PDFlib 8 では、新たな Unicode 対応 C++ バインディングが導入されました。新しいテンプレートベースのアプローチにより、文字列処理に関して下記の利用パターンに対応しています：

- C++標準ライブラリ型*std::wstring*の文字列が基本文字列型として用いられます。これは UTF-16 または UTF-32 でエンコードされた Unicode キャラクタ群を保持することができます。これが PDFlib 8 のデフォルト動作であり、カスタムデータ型（次項参照）が wstring よりも顕著な利点を提供するのでない限り、新しいアプリケーションにおいて推奨されるアプローチです。
- 文字列処理にカスタム（ユーザー定義）データ型を用いることもできます。ただし、そのカスタムデータ型が *basic_string* クラステンプレートのインスタンス化であり、かつユーザーが提供する変換メソッドを通じて Unicode への変換と Unicode からの変換ができるものである必要があります。例として、UTF-8 文字列のためのカスタム文字列型実装が PDFlib ディストリビューションに含まれています。
- プレーン C++ 文字列を用いることもできます。これは、PDFlib 7 までのバージョンを利用して開発された既存 C++ アプリケーションとの互換のためです。この互換方式は既存アプリケーションのためにのみ意図されているものです（後述のソースコード互換性に関する記述を参照してください）。

新しいインタフェースでは、PDFlib メソッドに受け渡される文字列と、PDFlib メソッドから受け取る文字列はすべて、ネイティブ wstring であると前提します。*wchar_t* データ型のサイズに応じて、wstring は UTF-16（2 バイトキャラクタ群）か UTF-32（4 バイトキャラクタ群）でエンコードされた Unicode 文字列を内容として持つと見なされます。ソースコード内のリテラル文字列は、ワイド文字列であることを示すために接頭辞 *L* をつける必要があります。リテラル内の Unicode キャラクタは *\u*・*\U* 文法で作成できます。この文法は ISO C++ 規格に含まれていますが、これに対応していないコンパイラもあります。その場合にはリテラル Unicode キャラクタは 16 進キャラクタで作成する必要があります。

アプリケーションを新しい C++ バインディングに合わせる　PDFlib 7 までのバージョンを利用して開発された既存 C++ アプリケーションは、下記のようにして PDFlib 8 に合わせることができます：

- PDFlib C++ クラスは *pdflib* 名前空間に属するようになりましたので、クラス名は修飾する必要があります。*pdflib::PDFlib* 構造を避けるため、クライアントアプリケーションは PDFlib メソッドを使う前に下記を追加する必要があります：

```
using namespace pdflib;
```

- アプリケーションの文字列処理を wstring に切り替えます。これは外部情報源からのデータも含みます。しかし、ソースコード内の文字列リテラル（オプションリストを含む）にも *L* 接頭辞をつける必要があります。例：

```
const wstring imagefile = L"nesrin.jpg";
image = p.load_image(L"auto", imagefile, L"");
```

- PDFlib エラーメッセージと例外文字列（*PDFlib・PDFlibException* クラスの *get_errmsg()* メソッド）を処理する際には、適切な wstring 対応メソッド（*wcerr* 等）を用いる必要があります。
- Unicode 非対応言語においてのみ必要な PDFlib メソッドの呼び出しは除去します。特に下記を除去します：

```
p.set_parameter("hypertextencoding", "host");
```

- PDFlib C++ バインディングにおいて、*pdflib.cpp* モジュールは必要なくなりました。PDFlib ディストリビューションにはこのモジュールのダミー実装が含まれていますが、PDFlib アプリケーションのビルド処理からはこれは除去する必要があります。

レガシアプリケーションのフルソースコード互換　新しい C++ バインディングは、アプリケーションレベルのソースコード互換を念頭に設計されていますが、クライアントアプリケーションは再コンパイルする必要があります。レガシアプリケーションにおいてフルソースコード互換を実現するには下記の方法があります：

- *pdflib.hpp* をインクルードする前に、wstring ベースのインタフェースを下記のように無効化します：

```
#define PDFCPP_PDFLIB_WSTRING 0
```

- *pdflib.hpp* をインクルードする前に、PDFlib 名前空間を下記のように無効化します：

```
#define PDFCPP_USE_PDFLIB_NAMESPACE 0
```

C++ でのエラー処理　PDFlib API 関数は、エラー発生時には C++ 例外を発生させます。この例外はクライアントコード内で C++ の *try* ～ *catch* 節を用いてキャッチされる必要があ

ります。詳細なエラー情報提供のため、PDFlib　　クラスにはパブリッククラス *PDFlib::Exception* があります。このパブリッククラスは、詳しいエラーメッセージ取得のためのメソッドと、例外番号取得のためのメソッドと、例外を発生させた PDFlib API 関数の名前を取得するためのメソッドとを公開しています。

PDFlib ルーチンが発生させる C++ 準拠の例外は規則どおりに動作します。PDFlib で発生した例外をキャッチするコードの抜粋を以下に挙げます。

```
try {
        ...いろいろなPDFlib命令...
catch (PDFlib::Exception &ex) {
        wcerr << L"PDFlib exception occurred in hello sample: " << endl
              << L"[" << ex.get_errnum() << L"] " << ex.get_apiname()
              << L": " << ex.get_errmsg() << endl;
}
```

C++ でのメモリ管理　C++ バインディングでクライアントが作成したメモリ管理は、C 言語バインディングの場合と同様に動作します。

PDFlib コンストラクタは、オプションのエラーハンドラと、オプションのメモリ管理プロシージャ、オプションの void ポインタ引数を受け入れます。*pdflib.hpp* ではデフォルトで NULL 引数が与えられており、これは PDFlib の内部エラーを引き起こし、メモリ管理ルーチンを起動させます。メモリ管理関数はすべて C 関数でなければならず、C++ メソッドであってはなりません。

2.6 Java バインディング

Java には、ネイティブ言語コードを Java プログラムに接続できるポータブルな仕組みがそなわっています。これを Java Native Interface（JNI）といいます。JNI は、ネイティブな C や C++ のルーチンを Java コード内から呼び出せるプログラミング方式を提供します。逆の呼び出しも可能です。各 C ルーチンは、Java VM で利用可能とするには、適切なコードによるラッパを必要とします。成果物であるライブラリは、Java VM に読み込まれるには、共有オブジェクトか動的オブジェクトとして生成される必要があります。

PDFlib は、ライブラリを Java から利用可能にする JNI ラッパコードを提供します。この技法により、共有ライブラリを Java VM から読み込んで、PDFlib を Java に接続させることができます。ライブラリの実際の読み込みは、***pdflib*** Java クラス中の静的メンバ関数を介して実現されます。このため、Java クライアントでは共有ライブラリの取り扱いにいちいちわずらわされることがありません。

PDFlib は安定かつこなれていますので、ネイティブ PDFlib ライブラリを Java VM に接続しても、Java アプリケーションに安定性やセキュリティ上の制約が加わることはありません。むしろネイティブな実装による速度向上の利点があります。

PDFlib Java 版のインストール　PDFlibバインディングが動作するには、Java VMが、PDFlib Java ラッパと PDFlib Java パッケージを利用できるようにする必要があります。PDFlib は次のパッケージ名を持つ Java パッケージとしてまとめられています。

```
com.pdflib.pdflib
```

このパッケージは ***pdflib.jar*** ファイルの中にあって、***pdflib*** というただ 1 つのクラスを持っています。このパッケージをアプリケーションで利用可能にするには、***pdflib.jar*** を ***CLASSPATH*** 環境変数に追加する必要があります。または、Java コンパイラとランタイムへの呼び出しの中に ***-classpath pdflib.jar*** オプションを加えるか、これと等価の手順を Java IDE 中で踏む必要があります。JDK では、Java VM が既知のディレクトリを検索するように設定することができます。具体的には、***java.library.path*** プロパティにディレクトリの名前を設定します。たとえば次のように記述します。

```
java -Djava.library.path=. pdfclock
```

このプロパティの値を知るには次のようにします。

```
System.out.println(System.getProperty("java.library.path"));
```

上記に加え、プラットフォームによって以下の手順を行う必要があります。

- Unix : ***libpdf_java.so*** ライブラリ（Mac OS X では ***libpdf_java.jnilib***）は、共有ライブラリのためのデフォルト格納場所のうちのいずれかに置くか、適切に設定したディレクトリに置く必要があります。
- Windows : ***pdf_java.dll*** ライブラリは、Windows のシステムディレクトリに置くか、PATH 環境変数で列挙されたディレクトリに置く必要があります。

J2EE アプリケーションサーバとサーブレットコンテナで PDFlib を使う　PDFlib は、サーバサイドの Java アプリケーションに完全に適合しています。PDFlib ディストリビューションには、PDFlib を J2EE 環境で利用するためのサンプルコードと設定が含まれています。下記の設定上の注意点に留意してください :

- サーバがネイティブライブラリをさがしに行くディレクトリはベンダーによって異なります。通常その候補となる場所は、システムディレクトリや、基盤となっている Java

VM に固有のディレクトリや、サーバディレクトリです。サーバベンダーが提供するマニュアルを参照してください。

- アプリケーションサーバとサーブレットコンテナはしばしば特別なクラスローダを用いていますが、それには制約があったり、専用のクラスパスを利用していたりする場合があります。サーバのなかには、特別なクラスパスを定義することによって PDFlib パッケージが確実に発見されるようにしなければならないものがあります。

PDFlib の各サーブレットエンジン・各アプリケーションサーバでの利用については、詳細は PDFlib ディストリビューションの中の J2EE ディレクトリ内の追加のマニュアルを参照してください。

Java でのエラー処理 Java バインディングは、PDFlib のエラーをネイティブ Java 例外に翻訳する特別なエラーハンドラをインストールします。例外が起きると PDFlib は次のクラスのネイティブ Java 例外を発生させます。

```
PDFlibException
```

Java 例外は以下のように通常の try/catch で処理できます。

```
try {

...いろいろなPDFlib命令...

} catch (PDFlibException e) {
            System.err.print("PDFlib exception occurred in hello sample:\n");
            System.err.print("[" + e.get_errnum() + "] " + e.get_apiname() +
                        ": " + e.get_errmsg() + "\n");

} catch (Exception e) {
        System.err.println(e.getMessage());

} finally {
        if (p != null) {
            p.delete();                     /* PDFlibオブジェクトを削除 */
        }
}
```

PDFlib は適切な ***throws*** 節を宣言しますので、クライアントのコードでは、起こりうるすべての PDFlib 例外をキャッチするか、それらを自分で宣言しなければなりません。

Unicode とレガシエンコーディングの変換 PDFlib ユーザーの便宜のため、ここでいくつか便利な文字列変換方法を示します。詳しくは Java の説明書を参照してください。下記のコンストラクタはバイト配列から、そのプラットフォームのデフォルトエンコーディングを用いて Unicode 文字列を生成します。

```
String(byte[] bytes)
```

下記のコンストラクタはバイト配列から、***enc*** 引数で与えられたエンコーディング（SJIS・UTF8・UTF-16 等）を用いて Unicode 文字列を生成します。

```
String(byte[] bytes, String enc)
```

下記の String クラスのメソッドは Unicode 文字列を、***enc*** 引数で指定されたエンコーディングに従った文字列に変換します。

```
byte[] getBytes(String enc)
```

PDFlib の Javadoc 仕様書　PDFlib パッケージには、PDFlib の Javadoc 仕様書が含まれています。この Javadoc は、すべての PDFlib API メソッドの短い記述しか含んでいませんので、詳しくは PDFlib API リファレンスを参照してください。

PDFlib の Javadoc を Eclipse で設定するには下記のように操作します：

- パッケージ・エクスプローラーで Java プロジェクトを右クリックし、**「*Javadoc* ロケーション」**を選択します。
- **「ブラウズ ...」**をクリックして、Javadoc が置かれているパス（PDFlib パッケージに含まれています）を選びます。

この手順をふめば、**「*Java* 参照」**パースペクティブや**「ヘルプ」**メニュー等で PDFlib の Javadoc をブラウズできるようになります。

Groovy での PDFlib の利用　PDFlib の Java バインディングは、Groovy 言語でも使うことができます。その API 呼び出しは Java の呼び出しと同等であり、オブジェクトのインスタンス化だけが若干違います。PDFlib を Groovy で使う簡単な例が、PDFlib ディストリビューションに含まれています。

2.7 .NET バインディング

（この節は、PDFlib チュートリアルの COM・.NET・REALbasic エディションにのみ掲載しています）

2.8 Perl バインディング

Perl[1] 用 PDFlib ラッパは、C ラッパファイル 1 個と Perl パッケージモジュール 2 個で構成されています。このモジュールのうち 1 個は各 PDFlib API 関数の Perl 版を提供し、もう 1 個は PDFlib オブジェクトのためのものです。C モジュールは、Perl インタプリタが実行時に読み込む共有ライブラリをビルドするために使われます。この処理には Perl パッケージモジュールも使用されます。Perl スクリプトでは ***use*** ステートメントで共有ライブラリモジュールを参照します。

PDFlib Perl 版のインストール　Perl の拡張機構は共有ライブラリを、実行時に DynaLoader モジュールを通じて読み込みます。Perl の実行形式が、共有ライブラリ対応つきでコンパイルされている必要があります（大多数の Perl の設定ではそうなっています）。

PDFlib バインディングが動作するためには、Perl インタプリタが PDFlib Perl ラッパとモジュール ***pdflib_pl.pm・PDFlib/PDFlib.pm*** を利用できるようになっている必要があります。以下に述べるプラットフォーム依存な方式を用いることもできますし、次のように ***-I*** コマンドラインオプションを用いて Perl の ***@INC*** モジュール検索パスにディレクトリを追加することもできます。

```
perl -I/path/to/pdflib hello.pl
```

Unix　Perl は、***pdflib_pl.so***（Mac OS X では ***pdflib_pl.bundle***）・***pdflib_pl.pm・PDFlib/PDFlib.pm*** を、カレントディレクトリで検索するか、下記の Perl コマンドで出力されるディレクトリで検索します：

```
perl -e 'use Config; print $Config{sitearchexp};'
```

Perl は ***auto/pdflib_pl*** サブディレクトリも検索します。上記コマンドの一般的な出力は次のようになります。

```
/usr/lib/perl5/site_perl/5.8/i686-linux
```

Windows　PDFlib は、Perl 5 の Windows への ActiveState ポートに対応しています。これは ActivePerl[2] とも呼ばれます。DLL ***pdflib_pl.dll*** とモジュール ***pdflib_pl.pm・PDFlib/PDFlib.pm*** が、カレントディレクトリで検索されるか、下記の Perl コマンドで出力されるディレクトリで検索されます：

```
perl -e "use Config; print $Config{sitearchexp};"
```

上記コマンドの一般的な出力は次のようになります。

```
C:\Program Files\Perl5.8\site\lib
```

Perl でのエラー処理　Perl バインディングは、PDFlib のエラーをネイティブ Perl 例外に翻訳する特別なエラーハンドラをインストールします。Perl の例外は、適切な言語構造を適用することにより取り扱うことができます。すなわち、問題の起こりそうな箇所を次のように挟みます。

```
eval {
        ...いろいろなPDFlib命令...
```

1. *www.perl.com* を参照。
2. *www.activestate.com* を参照。

```
};
if ($@) {
        die("$0: PDFlib Exception occurred:\n$@");
```

文字列処理の複数の方式　アプリケーションでの必要に応じて、UTF-8・UTF-16・レガシエンコーディングのいずれで処理を行うこともできます。以下のコードはこの 3 つの場合すべての作成例です。どの例も同じ日本語の出力を生成しますが、それぞれ違う形式の入力文字列を受け取っています。

1 つ目の例は Unicode UTF-8 で処理を行い、*Unicode::String* モジュールを使用しています。このモジュールは最新の Perl ディストリビューションに含まれており、CPAN で入手可能です。Perl は内部処理を UTF-8 で行うので、明示的な UTF-8 変換は不要です。

```
use Unicode::String qw(utf8 utf16 uhex);
...
$p->set_parameter("textformat", "utf8");
$font = $p->load_font("Arial Unicode MS", "unicode", "");
$p->setfont($font, 24.0);
$p->set_text_pos(50, 700);
$p->show(uhex("U+65E5 U+672C U+8A9E"));
```

2 つ目の例は Unicode UTF-16 で処理を行っています。バイト順序はリトルエンディアンです。

```
$p->set_parameter("textformat", "utf16le");
$font = $p->load_font("Arial Unicode MS", "unicode", "");
$p->setfont($font, 24.0);
$p->set_text_pos(50, 700);
$p->show("\xE5\x65\x2C\x67\x9E\x8A");
```

3 つ目の例は Shift-JIS で処理を行っています。Windows 以外では、文字列変換のために *goms-RKSJ-H* CMap を利用できる必要があります。

```
$p->set_parameter("SearchPath", "../../../resource/cmap");
$font = $p->load_font("Arial Unicode MS", "cp932", "");
$p->setfont($font, 24.0);
$p->set_text_pos(50, 700);
$p->show("\x93\xFA\x96\x7B\x8C\xEA");
```

Unicode とレガシエンコーディングの変換　PDFlib ユーザーの便宜のために、ここで便利な文字列変換方法を示します。詳しくは Perl の説明書を参照してください。下記のコンストラクタはバイト配列から UTF-16 Unicode 文字列を生成します：

```
$logos="\x{039b}\x{03bf}\x{03b3}\x{03bf}\x{03c3}\x{0020}" ;
```

下記のコンストラクタは Unicode キャラクタ名から Unicode 文字列を生成します：

```
$delta = "\N{GREEK CAPITAL LETTER DELTA}";
```

Encode モジュールは多くのエンコーディングに対応しており、そのエンコーディング間変換のためのインタフェースを持っています：

```
use Encode 'decode';
$data = decode("iso-8859-3", $data);    # レガシからUTF-8へ変換
```

2.9 PHP バインディング

注 PDFlib を PHP[1] で使う際のさまざまな方式やオプションに関する詳細な情報は、PHP 用の PDFlib のローダブルモジュールを用いるべきかどうかといった質問を含めて、*PDFlib-in-PHP-HowTo.pdf* 文書に記載してあります。この文書はディストリビューションパッケージに含まれており、また、PDFlib ウェブサイトにも掲載してあります。

PDFlib PHP 版のインストール PHP を設定して、外部の PDFlib ライブラリについて PHP が認識する必要があります。以下の 2 通りの選択肢があります。

- *php.ini* に以下の行のうちのいずれかを追加する。

```
extension=libpdf_php.so        ; Unix・Mac OS X用
extension=libpdf_php.dll       ; Windows用
```

PHP は、ライブラリを検索するとき、Unix では *php.ini* の中の *extension_dir* 変数で指定されたディレクトリの中を捜し、Windows ではその他に標準のシステムディレクトリの中も捜します。どのバージョンの PHP PDFlib バインディングがインストールしてあるかを確認するには、次のような一行の PHP スクリプトを用います。

```
<?phpinfo()?>
```

そうすると、カレントの PHP の設定に関して、長い情報ページが表示されます。このページの中で *pdf* という見出しのセクションを見てください。このセクションに *PDFlib GmbH Binary Version* と（PDFlib のバージョン番号とともに）書いてあれば、サポートされている新しい PDFlib ラッパを使用しているということです。サポートされていない古いラッパの場合はかわりに *PDFlib GmbH Version* と表示されます。

- スクリプトの先頭に以下の行のうちのいずれかを書いて PDFlib を実行時に読み込む。

```
dl("libpdf_php.so");           Unix用
dl("libpdf_php.dll");          Windows用
```

PHP に合わせた PDFlib 関数のエラー戻り値 PHP では、関数内でのエラー発生時に値 0（FALSE）を返す方式が用いられているので、PDFlib 関数はすべて、エラー発生時に -1 でなく 0 を返すよう変更してあります。この変更については、PDFlib API リファレンスの関数解説に記してあります。ただし 55 ページ「3 章 PDFlib 文書を作成」のコード抜粋例では、エラー時に -1 を返す通常の PDFlib の方式を用いているので、読む際には注意してください。

PHP のファイルの取り扱い PDF や画像などのディスク上のパスの指定の無いファイル名や相対パスのファイル名は、PHP の Unix 版と Windows 版とでは次のように異なった取り扱いを受けます。

- Unix システム上の PHP は、パス部分をまったく持たないファイルを、スクリプトが置かれているディレクトリの中で検索します。
- Windows 上の PHP は、パス部分をまったく持たないファイルを、PHP DLL が置かれているディレクトリの中でのみ検索します。

プラットフォームに依存しないファイル名の取り扱いを行うには、SearchPath 機能の利用を強く推奨します（59 ページ「3.1.3 リソース設定とファイル検索」参照）。

1. *www.php.net* を参照。

PHP での例外処理　PHP は構造化例外処理に対応しているため、PDFlib 例外は PHP 例外として発生します。PDFlib はクラス *PDFlibException* の例外を発生させます。このクラスは PHP の標準の Exception クラスを派生させたものです。以下のように、標準的な *try* ～ *catch* 技法を用いて PDFlib 例外を扱うことができます。

```
try {

...いろいろなPDFlib命令...

} catch (PDFlibException $e) {
    print "PDFlib exception occurred:\n";
    print "[" . $e->get_errnum() . "] " . $e->get_apiname() . ": "
                $e->get_errmsg() . "\n";
}
catch (Exception $e) {
    print $e;
}
```

Unicode とレガシエンコーディングの変換　*iconv* モジュールを用いて文字列変換ができます。詳しくは PHP の説明書を参照してください。

Eclipse と Zend Studio による PDFlib 開発　PHP Development Tools（PDT）[1] は Eclipse と Zend Studio による PHP 開発に対応しています。PDT は、下記に示す手順で、状況依存ヘルプに対応するよう設定することができます。

PDFlib を Eclipse 設定に追加して、すべての PHP プロジェクトに知られるようにします:

- 「*Window*」→「*Preferences*」→「*PHP*」→「*PHP Libraries*」→「*New...*」を選択してウィザードを起動します。
- 「*User library name*」に *PDFlib* と入力し、「*Add External folder...*」をクリックしてフォルダ *bind\php\Eclipse PDT* を選びます。

既存または新規の PHP プロジェクトで、PDFlib ライブラリへの参照を下記のように追加することができます :

- PHP Explorer で PHP プロジェクトを右クリックし、「*Include Path*」→「*Configure Include Path...*」を選択します。
- 「*Libraries*」タブへ行き、「*Add Library...*」をクリックし、「*User Library*」→「*PDFlib*」を選択します。

これらの手順をふめば、PHP Explorer ビューの *PHP Include Path/PDFlib/PDFlib* ノード配下で PDFlib メソッドの一覧を閲覧できるようになります。新しい PHP コードを書いている時、Eclipse は、すべての PDFlib メソッドに対するコード補完と状況依存ヘルプで支援します。

1. *www.eclipse.org/pdt* を参照。

2.10 Python バインディング

PDFlib Python 版のインストール　Python[1] の拡張機構は共有ライブラリを実行時に読み込むことによって動作します。PDFlib バインディングを動作させるには、以下のように、Python インタプリタが PDFlib Python ラッパを利用できるようにする必要があります。このラッパは、PYTHONPATH 環境変数に列挙されたディレクトリ内で検索されます。Python ラッパの名前はプラットフォームによって異なります：

- Unix・Mac OS X：*pdflib_py.so*
- Windows：*pdflib_py.pyd*

Python でのエラー処理　Python バインディングは、PDFlib エラーを Python 例外に翻訳する特別なエラーハンドラをインストールします。Python 例外は次のように通常の try/catch で取り扱うことができます。

```
try:
        ...いろいろなPDFlib命令...
except PDFlibException:
        print 'PDFlib Exception caught!'
```

1. *www.python.org* を参照。

2.11 REALbasic バインディング

（この節は、PDFlib チュートリアルの COM・.NET・REALbasic エディションにのみ掲載しています）

2.12 RPG バインディング

PDFlib は、PDFlib 関数を埋め込まれた ILE-RPG プログラムをコンパイルするために必要なすべてのプロトタイプといくつかの有用な定数とを定義した */copy* モジュールを提供しています。

Unicode 文字列処理　PDFlib が提供する関数はすべて、可変長の Unicode 文字列を引数として用いているので、*%UCS2* ビルトイン関数を使ってシングルバイト文字列を Unicode 文字列に変換する必要があります。PDFlib の関数が返す文字列はすべて可変長の Unicode 文字列です。これらの Unicode 文字列をシングルバイト文字列に変換するには *%CHAR* ビルトイン関数を使います。

注 *%CHAR*・*%UCS2* 関数は、カレントジョブの CCSID を使って文字列を Unicode と変換します。PDFlib に同梱の例では CCSID 37（US EBCDIC）をベースとしています。この例をこれ以外のコードページで走らせると、オプションリスト内のいくつかの特殊キャラクタ（*{ [ ] }* 等）が正しく翻訳されないことがあります。

すべての文字列は可変長文字列として渡されるので、文字列長を明示的に指定する必要のあるさまざまな関数では *length* 引数を渡してはいけません（可変長文字列の長さは、文字列の先頭 2 バイトに格納されています）。

PDFlib を利用している RPG プログラムのコンパイルとバインド　PDFlib 関数を RPG から利用するには、コンパイル済みの PDFLIB・PDFLIB_RPG サービスプログラムが必要です。PDFlib の定義をコンパイル時にインクルードするには、*/copy* メンバの名前を次のように ILE-RPG プログラムの *D* スペックの中で指定する必要があります。

```
d/copy QRPGLESRC,PDFLIB
```

PDFlib ソースファイルライブラリがライブラリリストの先頭にない場合は、次のようにそのライブラリも指定しなければなりません。

```
d/copy PDFsrclib/QRPGLESRC,PDFLIB
```

ILE-RPG プログラムのコンパイルを開始する前に、PDFlib に同梱の PDFLIB・PDFLIB_RPG サービスプログラムを含むバインドディレクトリを作成しておく必要があります。次の例は、ライブラリ PDFLIB の中の PDFLIB というバインドディレクトリを作成したい場合の指定です。

```
CRTBNDDIR BNDDIR(PDFLIB/PDFLIB) TEXT('PDFlib Binding Directory')
```

バインドディレクトリを作成した後は、PDFLIB・PDFLIB_RPG サービスプログラムをバインドディレクトリに追加する必要があります。次の例は、さきに作成したバインドディレクトリにライブラリ PDFLIB の中のサービスプログラム PDFLIB を追加したい場合の指定です。

```
ADDBNDDIRE BNDDIR(PDFLIB/PDFLIB) OBJ((PDFLIB/PDFLIB *SRVPGM))
ADDBNDDIRE BNDDIR(PDFLIB/PDFLIB) OBJ((PDFLIB/PDFLIB_RPG *SRVPGM))
```

これで、次のように *CRTBNDRPG* コマンド（または PDM の中のオプション 14）を用いてプログラムをコンパイルできるようになりました。

```
CRTBNDRPG PGM(PDFLIB/HELLO) SRCFILE(PDFLIB/QRPGLESRC) SRCMBR(*PGM) DFTACTGRP(*NO)
BNDDIR(PDFLIB/PDFLIB)
```

RPG でのエラー処理　ILE-RPG で書かれた PDFlib クライアントでは、ILE-RPG が提供する *monitor/on-error/endmon* エラー処理機構を利用することがでいます。あるいは、ILE-RPG の ***PSSR*** グローバルエラー処理サブルーチンを用いて例外をモニタする方法もあります。ジョブログにはエラー番号と、失敗した関数、および例外の原因が示されます。PDFlib は呼び出し側プログラムへエスケープメッセージを送ります。

```
c     eval      p=PDF_new
*
c     monitor
*
c     eval      doc=PDF_begin_document(p:%ucs2('/tmp/my.pdf'):docoptlist)
:
:
*     エラー処理
c     on-error
*     このエラーについて何かします
*     PDFlibオブジェクトの解放を忘れないように
c     callp     PDF_delete(p)
c     endmon
```

2.13 Ruby バインディング

PDFlib Ruby 版のインストール　Ruby[1] の拡張機構、共有ライブラリを動作時に読み込むことによって動作します。PDFlib バインディングが動作するには、Ruby 用の PDFlib 拡張ライブラリの場所を Ruby インタプリタが知っていて利用できる必要があります。たいていの場合、このライブラリ（Windows・Unix では ***PDFlib.so***、Mac OS X では ***PDFlib.bundle***）がインストールされる場所は、ローカル ruby インストールディレクトリの ***site_ruby*** ブランチの中であり、すなわち、次のような名前のディレクトリの中にインストールされます。

```
/usr/local/lib/ruby/site_ruby/<rubyバージョン>/
```

ただし Ruby は、これ以外のディレクトリへも拡張を探しに行きます。これらのディレクトリの一覧を取得するには、次の ruby コールを用いることができます。

```
ruby -e "puts $:"
```

この一覧はたいていの場合、カレントディレクトリを含んでいますので、テスト目的においては、PDFlib 拡張ライブラリとスクリプトを同じディレクトリに入れるだけでよいでしょう。

Ruby でのエラー処理　Ruby バインディングは、PDFlib 例外をネイティブ Ruby 例外に翻訳する特別なエラーハンドラをインストールします。こうした Ruby 例外は、以下のような通常の ***rescue*** 技法で扱うことができます。

```
begin
        ...いろいろなPDFlib命令...
rescue  PDFlibException => pe
  print "PDFlib exception occurred in hello sample:\n"
  print "[" + pe.get_errnum.to_s + "] " + pe.get_apiname + ": " + pe.get_errmsg + "\n"
```

Ruby on Rails　Ruby on Rails[2] は、Ruby による Web 開発を実現するオープンソースフレームワークです。Ruby 用 PDFlib 拡張は Ruby on Rails で利用可能です。作成例をパッケージ内に収めてあります。Ruby on Rails 用 PDFlib 作成例を実行するには以下の手順に従ってください。

- Ruby をインストールします。
- Ruby on Rails をインストールします。
- Ruby on Rails 用サンプルを含む Ruby 版 PDFlib パッケージをアンパックします。
- ***bind/ruby/RubyOnRails*** ディレクトリへ移動し、次のように Ruby ウェブサーバを起動します。

  ```
  ruby script/server
  ```

- ブラウザで ***http://localhost:3000*** を開きます。

PDFlib サンプルのコードは ***app/controllers/pdflib_controller.rb*** にあります。

ローカルに PDFlib をインストール　PDFlib を Ruby on Rails だけで使いたいにもかかわらず、PDFlib を Ruby で一般利用できるようグローバルにインストールすることができない場合は、Rails ツリー内の ***vendors*** ディレクトリの中に PDFlib をローカルにインストール

1. ***www.ruby-lang.org/en*** を参照。
2. ***www.rubyonrails.org*** を参照。

するという方法があります。この方法は特に、Ruby 拡張を一般利用できるようインストールする権限を持たない場合に、それでも PDFlib を Rails で扱いたいというときに役立ちます。

2.14 Tcl バインディング

PDFlib Tcl 版のインストール　Tcl [1] の拡張機構は共有ライブラリを実行時に読み込むことによって動作します。PDFlib バインディングを動作させるには、Tcl シェルが PDFlib Tcl ラッパ共有ライブラリとパッケージインデックスファイル *pkgIndex.tcl* を利用できるようにする必要があります。あるディレクトリ内のライブラリを利用できるようにするには、スクリプト内で次のような指定を用います（この方法は、PDFlib をインストールするためのルート権限を持たないマシン上で PDFlib を利用したい場合に有用です）。

```
lappend auto_path /path/to/pdflib
```

- Unix：ライブラリ *pdflib_tcl.so*（Mac OS X では *pdflib_tcl.dylib*）は、共有ライブラリのデフォルトの場所のうちの 1 つに置かれているか、適切に設定されたディレクトリに置かれている必要があります。通常、*pkgIndex.tcl* と *pdflib_tcl.so* は両方とも次のディレクトリに置かれています。

```
/usr/lib/tcl8.4/pdflib
```

- Windows：ファイル *pkgIndex.tcl* と *pdflib_tcl.dll* は以下のディレクトリ内で検索されます。

```
C:\Program Files\Tcl\lib\pdflib
C:\Program Files\Tcl\lib\tcl8.3\pdflib
```

Tcl でのエラー処理　Tcl バインディングは、PDFlib エラーをネイティブ Tcl 例外に翻訳する特別なエラーハンドラをインストールします。Tcl 例外は次のように通常の try ～ catch 技法で処理できます。

```
if [ catch { ...いろいろなPDFlib命令... } result ] {
        puts stderr "Exception caught!"
        puts stderr $result
}
```

1. *www.tcl.tk* を参照。

3 PDFlib 文書を作成

3.1 PDFlib プログラミングの一般的特徴

クックブック プログラミングの一般的な諸側面に関するコードサンプルが PDFlib クックブックの general カテゴリにあります。

3.1.1 例外処理

ある種のエラーは、多くの言語で例外と呼ばれています。この呼び方は理にかなっています。そうしたエラーはまさに例外であって、プログラムの動作中にそんなに頻繁には起こらないだろうと予想される類のものだからです。一般に採られる方針としては、エラーが頻繁に起きそうな関数呼び出しについては従来型のエラー伝達方式（すなわち、-1 等の特殊な戻り値でエラーを示す）を用いるけれども、まれにしか起こらないエラーで、こんな特例にまで場合分けを増やしてコードをややこしくするのもどうかと思われるような場合については例外方式を用いる、というのが普通です。PDFlib のやり方もまさにこれと同じです。たとえば、かなりの頻度でエラーが起きそうな操作としては、次のようなものが挙げられます。

- 出力ファイルを、パーミッションがないのに開こうとする
- PDF を読み込みたいとき、間違ったファイル名で開こうとする
- 画像ファイルが壊れているのに開こうとする

PDFlib では、このようなエラーを特殊な戻り値で示します。それぞれの値は PDFlib API リファレンスに示してあります（たいていは -1。ただし PHP バインディングでは 0）。エラー時に -1 を返すと解説に記されている関数についてはすべて、アプリケーション開発者の側でこのエラーコードをチェックする必要があります。

これに対して、以下に挙げるようなエラーは、もっと深刻かもしれない、しかしさほど頻繁には起こらないエラーです。

- 仮想メモリ不足
- スコープ違反（たとえば文書を開く前に閉じようとする）
- PDFlib API 関数に対する引数指定の誤り（たとえば円を負の半径で描こうとする）、またはオプション指定の誤り

PDFlib がこのような状況を認識したときには、特殊なエラー戻り値が呼び出し側に渡るのではなく、例外が発生します。知っておくべき重要なこととして、例外が発生すると、それまで生成させていた PDF 文書は完了できなくなります。例外の後で安全に呼べるメソッドは *PDF_delete()*・*PDF_get_apiname()*・*PDF_get_errnum()*・*PDF_get_errmsg()* だけです。それ以外の PDFlib のメソッドを例外の後で呼んだ場合の結果は予測不能です。例外の中には以下の情報が含まれています。

- 一意なエラー番号。
- 例外を起こした PDFlib API 関数の名前。
- 問題の詳細などを述べたテキスト。

関数呼び出しによる問題発生の原因情報を得る さきに述べたように、生成中の PDF 出力文書は、例外が起こったらかならず放棄しなければなりません。しかしクライアントによっては、プログラムの流れを調整したり、別のデータを与えたりすることで、文書を存続させることを望む場合もあるかもしれません。たとえば、あるフォントの読み込みがで

きないとき、たいていのクライアントは文書を放棄してしまうでしょうが、クライアントによっては、別のフォントを使って何とかしたいという場合もありえます。そのようなときは、問題がより詳しく記されたエラーメッセージを得たい場合もあるでしょう。そのような場合には、*PDF_get_errnum()*・*PDF_get_errmsg()*・*PDF_get_apiname()* 関数を、問題発生した関数呼び出しの直後に呼び出すことができます。具体的には、エラー値 -1（PHP では 0）を返してきた関数呼び出しの直後に呼び出します。

エラーポリシー　PDFlib はエラー状況を検出すると、いくつかの戦略のうちの 1 つに従って反応します。これは *errorpolicy* パラメタで設定できます。エラーコードを返す可能性のある関数はすべて、*errorpolicy* オプションにも対応しています。以下のエラーポリシーに対応しています。

- *errorpolicy=legacy*：この設定は非推奨ですが、以前のバージョンの PDFlib と互換な動作を保証します。すなわち例外やエラー戻り値を、*fontwarning*・*imagewarning* といったパラメタやオプションで制御する方式です。これは、PDFlib 6 とのソースコード互換性を要するアプリケーションについてのみ推奨します。新しいアプリケーションでは使用するべきではありません。*legacy* 設定はデフォルトのエラーポリシーです。
- *errorpolicy=return*：エラー状況が検出されたとき、いかなる警告パラメタ・オプションにもよらず、各関数がエラー値 -1（PHP では 0）を返します。アプリケーション開発者はこの戻り値を調べて問題を発見し、その問題に対してアプリケーションごとに適切な方法で対処する必要があります。このアプローチではエラー処理に統一的なアプローチができるので、推奨します。
- *errorpolicy=exception*：エラー状況が検出されたとき、例外を発生させます。ただし、出力文書は例外の後には使えなくなります。この方式を使うと、エラー条件分岐をさぼって手軽にプログラミングができますが、そのかわり問題が起きると出力文書は、たとえそれがアプリケーション側で対処できる問題であっても失われてしまいます。

以下のコードに、例外処理に関する各種の戦略を示します。各作成例は、存在するかしないかわからないフォントを読み込もうとしています。

errorpolicy=return のときは、戻り値がエラーかどうかを調べる必要があります。それが失敗を示しているときは、以下のように失敗の原因を取得することで、状況を適切にさばくことができるでしょう。

```
font = p.load_font("MyFontName", "unicode", "errorpolicy=return");
if (font == -1)
{
        /* フォントハンドルが無効。しかしPDF出力は継続可能。*/
        errmsg = p.get_errmsg();
        /* 別のフォントを試すか諦める */
        ...
}
/* フォントハンドルは有効。継続 */
```

errorpolicy=exception のときは以下のように、エラーが発生したら文書は放棄しなければなりません。

```
font = p.load_font("MyFontName", "unicode", "errorpolicy=exception");
/* 例外が発生しなければフォントハンドルは有効。
 * 例外が発生したら、PDF出力は継続できない
 */
```

クックブック　完全なコードサンプルがクックブックの general/error_handling トピックにあります。

警告　問題の状況によっては、PDFlib がそれを内部的に検出できても、例外を発生させてプログラムの流れをさえぎる正当な理由にはならないものもあります。以前のバージョンの PDFlib は、致命的でない例外という概念に対応していて、それは無効にもできましたが、PDFlib 7 は致命的でない状況では、例外を一切発生しなくなりました。かわりに、状況の説明をログ記録します（ログ記録を有効にしてあれば）。ログ記録は下記のように有効にできます。

```
p.set_parameter("logging", "filename=private.log");
```

警告に関しては以下のアプローチを推奨します。

- 開発局面では警告のログ記録を有効にして、そのログファイル内のすべての警告メッセージを注意深く研究します。こうした警告は、コードやデータにひそむ問題を示している可能性がありますから、その原因を理解ないし除去するよう努めなければなりません。
- 運用局面では警告のログ記録を無効にして、問題が起きたときだけ再び有効にします。

3.1.2 PDFlib 仮想ファイルシステム（PVF）

クックブック　完全なコードサンプルがクックブックの general/starter_pvf **トピックにあります。**

ディスク上のファイルとは別に、クライアントは、***PDFlib Virtual File System***（PVF）というしくみを利用することもできます。これを用いると、メモリ内のデータを直接供給することができるので、ディスクのファイルを扱う必要がまったくありません。これには速度向上という利点があります。PVF はデータベースから取り出したデータ等のように、ファイルとしてディスク上に存在していない時にも活用できます。またそれ以外にも、クライアントの必要とするデータが、何らかの処理の結果としてメモリ上にすでに存在している場合に対して一般に有用です。

PVF の基本コンセプトは、仮想の読み取り専用ファイルに名前を付けて、それを通常のファイル名とまったく同じように、あらゆる API 関数で使用できるようにするというものです。UPR の設定ファイルでも使用することができます。仮想ファイル名は、クライアントが任意に名づけることができます。もちろん、仮想ファイル名は、通常のディスクのファイルと名前衝突が起こらないようなものにしなければなりません。そのため、以下のような体系的な命名規則を推奨します（***filename*** は、クライアントが名づける部分で、各カテゴリ内で一意な名前です）。また、ファイル名の拡張子についても、標準に従うことを推奨します。

- ラスタ画像ファイル：***/pvf/image/filename***
- フォントのアウトラインやメトリックのファイル（実際のフォント名をファイル名の先頭部分として用いることを推奨）：***/pvf/font/filename***
- ICC プロファイル：***/pvf/iccprofile/filename***
- エンコーディングとコードページ：***/pvf/codepage/filename***
- PDF 文書：***/pvf/pdf/filename***

ファイル名を指定されて捜す時、PDFlib はまず、その指定されたファイル名が既知の仮想ファイルのなかにあるかどうかを調べます。その後に、指定されたファイルをディスク上で開こうとします。

仮想ファイルの継続期間　仮想ファイルでデータが提供されたときには、それをすぐに消費する関数もあるでしょうし、そのファイルの一部分だけをまず読んで、残りの部分は後の時点で使うという関数もあるでしょう。そのため、各仮想ファイルの継続期間については細心の注意を払う必要があります。PDFlib は、それぞれの仮想ファイルに内部ロック

をかけておき、その内容がもう必要なくなった時にはじめてロックを外します。そのデータをただちにコピーしておくようクライアントが PDFlib に要求した場合（*PDF_create_pvf()* で *copy* オプションを指定）を除いては、仮想ファイルの内容をクライアントが変更・削除・解放することが許されるのは、PDFlib がロックが外した後に限られます。PDFlib は、*PDF_delete()* が実行されたときには、自動的にすべての仮想ファイルを削除します。しかし、ファイルの実際の内容（仮想ファイルの元データ）はつねにクライアントが解放しなければいけません。

さまざまな手法 PVF では、仮想ファイルのための必要メモリの管理手法は複数ありえます。どの手法でも考慮の対処とするべきことは、PDFlib は、仮想ファイル名を用いた API 呼び出しが終わった後でもまだその仮想ファイルの内容へのアクセスを必要とするかもしれないということと、*PDF_close()* の後であればもうアクセスを必要とすることは全くないということです。留意する必要があるのは、*PDF_delete_pvf()* を呼んでも実際のファイルの内容が解放されるわけではなく（*copy* オプションを指定した場合は例外）、その PVF ファイル名を管理していたデータ構造が解放されるだけだということです。よって、以下のような方式があります。

- メモリ使用を最小にする手法 : API 呼び出しで仮想ファイル名を用いたら、その直後に *PDF_delete_pvf()* を呼ぶのがよいでしょう。そして、*PDF_close()* の後にまた *PDF_delete_pvf()* を呼ぶことを推奨します。この 2 度目の呼び出しがどうして必要かというと、最初の呼び出しの時点ではまだ PDFlib からデータへアクセスが必要だったかもしれず、そのような場合はその時点では仮想ファイルのロック解除が拒否されてしまっているためです。しかし、最初の呼び出しですでにデータが解放できている場合もあるわけで、そのような場合でも 2 度目の呼び出しは何も害にはなりません。クライアントがファイルの内容を解放できるのは *PDF_delete_pvf()* が成功した時だけなのです。
- 仮想ファイルを再利用して速度を向上させる手法 : クライアントによっては、同じデータ（たとえばフォント定義）を複数の出力文書に対して使いまわしたいこともあるでしょう。そのような場合には、同じファイル内容に対して何度も作成・削除を繰り返すのは賢明ではありません。その仮想ファイルを使ってほかにもまだ PDF 出力文書を生成したいうちは、*PDF_delete_pvf()* は呼び出さないでおくのがよいでしょう。
- 手抜きプログラミング : メモリ使用量が気にならなければ、クライアントが *PDF_delete_pvf()* を全然呼ばないようにしてもかまいません。この場合 PDFlib は、*PDF_delete()* が呼ばれた時点で、すべての開かれたままの仮想ファイルを内部的に削除します。

どの場合でも、クライアントがそれぞれのデータを解放できるのは、*PDF_delete_pvf()* が成功裏に帰ってきた時か、*PDF_delete()* の後だけです。

PDF 出力を仮想ファイルに生成 PVF は、ユーザーデータを PDFlib に与えるだけでなく、PDFlib が生成した PDF 文書データを保持することもできます。これは、*PDF_begin_document()* に *createpvf* オプションを与えることによって実現できます。その PVF ファイル名は以後、別の PDFlib API 関数に与えることができます。これはたとえば、PDF 文書を PDF ポートフォリオへ入れ込むために生成するときに有用です。PDFlib が生成した PVF データを直接取得することはできませんので、メモリから PDF データを取り出すには、能動または受動インコア PDF 生成を用います（63 ページ「3.1.4 PDF 文書をメモリ内に生成」参照）。

3.1.3 リソース設定とファイル検索

高度な応用アプリケーションでは、PDFlib にさまざまなリソースを利用させる必要があります。たとえばフォントファイル・エンコーディング定義・ICC カラープロファイルなどです。PDFlib のリソース管理を、プラットフォームに依存しない、カスタマイズの容易なものにするためには、設定ファイルを作成しておくことができます。設定ファイルの中には、利用可能なさまざまなリソースとそれぞれのディスクファイル名を記述しておくのです。このような静的な設定ファイルだけではなくて、*PDF_set_parameter()* によるリソースの追加を用いた動的な実行時設定を行うことも可能です。設定ファイルに関しては、*Unix PostScript Resource*（UPR）という簡単なテキスト形式を採用しました。この形式は、Display PostScript の時代に生み出されたもので、今でもいくつかのシステムで使われています。ただし元の UPR 形式を多少拡張しました。この、PDFlib で用いる形の UPR ファイル形式については後述します。*makepsres* というユーティリティもあります（X Window System の一部として頒布されることが多い）。このユーティリティを使うと、PostScript フォントファイルとメトリックファイルから自動的に UPR ファイルを生成させることができます。

リソースのカテゴリ　PDFlib の対応しているリソースカテゴリを表 3.1 に挙げます。これ以外のリソースカテゴリは無視されます。その値は名前文字列として扱われます。すなわち、ASCII か UTF-8（BOM つき）でエンコーディングすることができます。Unicode 値は、*HostFont* リソースで各国語のフォント名を指定するのに有用でしょう。

表 3.1 PDFlib で使えるリソースカテゴリ

カテゴリ	形式	説明
SearchPath	値	データファイルのあるディレクトリの相対または絶対パス名
CMap	キー = 値	日中韓エンコーディングのための CMap ファイル
FontAFM	キー = 値	AFM 形式の PostScript フォントメトリックファイル
FontPFM	キー = 値	PFM 形式の PostScript フォントメトリックファイル
FontOutline	キー = 値	PostScript か TrueType か OpenType のフォントアウトラインファイル
Encoding	キー = 値	8 ビットエンコーディングまたはコードページのテーブルを持ったテキストファイル
HostFont	キー = 値	システムにインストールされているフォントの名前
ICCProfile	キー = 値	ICC カラープロファイル名
StandardOutputIntent	キー = 値	PDF/X の標準出力条件名（PDFlib 内蔵のものに加えて、完全な一覧は PDFlib API リファレンスを参照）

UPR ファイル形式　UPR ファイルはテキストファイルであり、その構造は非常に単純で、テキストエディタで簡単に書くことができますし、自動生成させることもできます。まず、その文法を見てみましょう。

- それぞれの行は最大 1023 キャラクタまで。
- 行末のバックスラッシュキャラクタ「\」は、行終端をキャンセルします。これは行を延長したいときに使えます。
- パーセント「%」キャラクタは、行末までの注釈を開始させます。行データの一部である（すなわち注釈を開始させない）パーセントキャラクタは、直前にバックスラッシュキャラクタを付けて保護する必要があります。

- 行終端を保護するバックスラッシュと、パーセントキャラクタを保護するバックスラッシュキャラクタとの直前のバックスラッシュキャラクタ群は、行データの一部であるなら二重にする必要があります。
- ピリオド「.」を単独で用いると、セクションの終了を意味します。
- すべての記述において、大文字・小文字が区別されます。
- スペースは、リソース名中とファイル名中をのぞくあらゆる箇所で無視されます。
- リソースの名前を値は、等号「=」を一切含んではいけません。
- 1 個のリソースが複数回定義された場合は、最後の定義がそれ以前の定義を上書きします。

UPR ファイルは以下の部分から成っています。

- ファイルの種類を示すおまじない行。次の形をとります。

```
PS-Resources-1.0
```

- ファイル中で記述されるすべてのリソースカテゴリを一覧にしたセクション。省略可能です。各行に 1 つずつリソースカテゴリを記述します。この一覧は、ピリオド 1 個だけの行によって終了します。利用可能なリソースカテゴリについては後述します。この省略可能なセクションが存在しない場合であっても、1 個のピリオドキャラクタは存在する必要があります。
- ファイルのはじめに挙げられたリソースカテゴリそれぞれについてセクションが 1 つずつ。各セクションは、リソースカテゴリを示す 1 行で始まり、その後に、利用可能なリソースを記述する行が任意の行数つづきます。この一覧は、ピリオド 1 個だけの行によって終了します。各リソースデータ行にはリソースの名前を書きます（等号はクォートする必要があります）。そのリソースがファイル名を必要とする場合には、等号の後にこの名前を付け加える必要があります。リソース記述に挙げられたファイルを PDFlib がさがす時には ***SearchPath***（以下を参照）が適用されます。

ファイル検索と SearchPath リソースカテゴリ　PDFlib はさまざまなデータアイテムをディスク上のファイルから読み込みます。たとえばラスタ画像・フォントアウトライン・フォントメトリック情報・エンコーディング定義・PDF 文書・ICC カラープロファイルなどです。相対パス名でも絶対パス名でもない、パス指定をまったくつけないファイル名を用いることもできます。***SearchPath*** リソースカテゴリを使うと、必要なデータファイルのあるディレクトリのパス名の一覧を指定することができます。何かファイルを開かなければならないとき、PDFlib は、まずそのままのファイル名でファイルを開こうとします。この試みが失敗すると、PDFlib は、***SearchPath*** リソースカテゴリで指定されたディレクトリ群の中でそのファイルが開けないかどうか、成功するまで一つ一つ試します。***SearchPath*** は複数指定することができ、逆順に検索されます（後の時点で設定されたパスほど、もっと早くに設定されたものよりも先に検索される）。この機能を使うと、PDFlib のアプリケーションを、プラットフォーム依存なファイルシステム体系から解き放つことができます。検索パス項目は下記のように設定できます。

```
p.set_parameter("SearchPath", "/パス/パス/ディレクトリ1");
p.set_parameter("SearchPath", "/パス/パス/ディレクトリ2");
```

この検索を無効にするには、フルパスによる指定を PDFlib 関数の中に書きます。なお、***SearchPath*** リソースカテゴリの機能は以下のようにプラットフォーム依存になっています。

- Windows の場合、PDFlib はレジストリ項目から読み取った記述で ***SearchPath*** リソースカテゴリを初期化します。下記のレジストリ項目にパス名のリストをセミコロン「;」で区切って指定することが可能です。これらは下記の順序で検索されます。

```
HKLM\SOFTWARE\PDFlib\PDFlib8\8.0.3\SearchPath
HKLM\SOFTWARE\PDFlib\PDFlib8\SearchPath
HKLM\SOFTWARE\PDFlib\SearchPath
```

- IBM iSeries では、***SearchPath*** リソースカテゴリは以下の値で初期化されます。

```
/PDFlib/PDFlib/8.0/resource/icc
/PDFlib/PDFlib/8.0/resource/fonts
/PDFlib/PDFlib/8.0/resource/cmap
/PDFlib/PDFlib/8.0
/PDFlib/PDFlib
/PDFlib
```

  これらの項目の最後のものはとくに、複数の製品に対するライセンスファイルを格納するために有用です。
- MVS つき IBM zSeries システム群では、SearchPath 機能には対応していません。
- OpenVMS では、論理名を ***SearchPath*** として与えることができます。

サンプル UPR ファイル　UPR 設定ファイルの作成例を以下に挙げます。

```
PS-Resources-1.0
.
SearchPath
/usr/local/lib/fonts
C:/psfonts/pfm
C:/psfonts
/users/kurt/my_images
.
FontAFM
Code-128=Code_128.afm
.
FontPFM
Corporate-Bold=corpb___.pfm

Mistral=c:/psfonts/pfm/mist____.pfm
.
FontOutline
Code-128=Code_128.pfa
ArialMT=Arial.ttf
.
HostFont
Wingdings=Wingdings
.
Encoding
myencoding=myencoding.enc
.
ICCProfile
highspeedprinter=cmykhighspeed.icc
.
```

UPR リソースファイルの検索　組み込みリソース（たとえば PDF コアフォント・組み込みエンコーディング・sRGB ICC プロファイル）やシステムリソース（たとえばホストフォ

ント）だけが使われる場合には、UPR 設定ファイルは必要とはされません。なぜなら、とりたてて設定がなくても、必要なリソースをすべて PDFlib が見つけられるからです。

それ以外のリソースを使いたい場合は、そのリソースを指定するために、***PDF_set_parameter()***（後述）を呼び出すか、UPR リソースファイルに記述します。このファイルを、PDFlib は、最初のリソースが要求された時に自動的に読み込みます。その過程は詳しくは下記のとおりです：

- Unix システム・Mac OS X システム・i5/iSeries では、パス・ディレクトリ名を一切指定しなくても、デフォルトでいくつかのディレクトリでライセンス・リソースファイルが検索されます。UPR ファイルを検索して読み取る前に、下記のディレクトリが検索されます（この順に）：

  ```
  <rootpath>/PDFlib/PDFlib/8.0/resource/icc
  <rootpath>/PDFlib/PDFlib/8.0/resource/fonts
  <rootpath>/PDFlib/PDFlib/8.0/resource/cmap
  <rootpath>/PDFlib/PDFlib/8.0
  <rootpath>/PDFlib/PDFlib
  <rootpath>/PDFlib
  ```

  Unix システムと Mac OS X では、***<rootpath>*** は、まず ***/usr/local*** で、ついで HOME ディレクトリで置き換えられます。i5/iSeries では ***<rootpath>*** はは空です。この機能を利用すれば、環境変数やランタイム引数を一切指定せずにライセンスファイル・UPR ファイル・リソースを取り扱うこともできます。
- 環境変数 ***PDFLIBRESOURCE*** が定義されていれば、その値を PDFlib は、読み込むべき UPR ファイルの名前とします。このファイルが読み込めないときは例外が発生します。
- 環境変数 ***PDFLIBRESOURCE*** が定義されていない場合、PDFlib は下記の名前のファイルを開こうとします：

  ```
  upr（MVSの場合。データセットが期待される）
  pdflib/<バージョン>/fonts/pdflib.upr（IBM i5/iSeriesの場合）
  pdflib.upr（Windows・Unix・その他すべてのシステムの場合）
  ```

  このファイルが読み込めないときは例外は発生しません。
- Windows では上記以外に PDFlib は下記のレジストリ項目を読み込もうとします。下記の順に検索されます：

  ```
  HKLM\Software\PDFlib\PDFlib8\8.0.3\resourcefile
  HKLM\Software\PDFlib\PDFlib8\resourcefile
  HKLM\Software\PDFlib\resourcefile
  ```

  そしてこれらの項目の値を、用いるべきリソースファイルの名前とします。このファイルが読み込めないときは例外が発生します。
  64 ビット Windows システム上でレジストリを手作業で扱う際の注意点：通常どおり、64 ビットの PDFlib バイナリは Windows レジストリの 64 ビットビューとともに動作し、64 ビットシステム上で動作する 32 ビットの PDFlib バイナリはレジストリの 32 ビットビューとともに動作します。32 ビット製品に対するレジストリキーを手作業で追加する必要があるときは、必ず 32 ビット版の ***regedit*** ツールを使用してください。これは **「スタート」→「ファイル名を指定して実行 ...」**のダイアログから下記によって起動することができます：

  ```
  %systemroot%\syswow64\regedit
  ```

注 64 ビット Windows システム上でレジストリを手作業で扱う際の注意点：通常どおり、64 ビットの PDFlib バイナリは Windows レジストリの 64 ビットビューとともに動作し、64 ビットシステム上で動作する 32 ビットの PDFlib バイナリはレジストリの 32 ビットビューとともに動作します。32 ビット製品に対するレジストリキーを手作業で追加する必要があるときは、必ず 32 ビット版の *regedit* ツールを使用してください。これは「スタート」→「ファイル名を指定して実行 ...」のダイアログから下記によって起動することができます：

- クライアント側で ***resourcefile*** パラメタを明示的に設定することでリソースファイルを PDFlib に実行時に読み込ませることもできます。次のように記述します。

  ```
  p.set_parameter("resourcefile", "/パス/パス/pdflib.upr");
  ```

  この呼び出しは任意の回数繰り返すことができます。その場合、リソース記述が蓄積されていきます。

リソースを実行時に設定　UPR ファイルを使った設定だけではなく、ソースコード中で *PDF_set_parameter()* 関数を使って直接個々のリソースを設定することもできます。この関数はカテゴリ名とそれに対するリソース記述をとります。リソース記述の部分は、UPR リソースファイルでこのカテゴリのセクションに書くのと同じように書きます。たとえば次のようになります。

```
p.set_parameter("FontAFM", "Foobar-Bold=foobb___.afm");
p.set_parameter("FontOutline", "Foobar-Bold=foobb___.pfa");
```

注 フォント設定の詳しい説明は 128 ページ「5.4.3 フォントを検索」を参照してください。

リソース値の取得　リソース項目は設定するだけでなく、*PDF_get_parameter()* を使って取得することもできます。カテゴリ名をキーとして、リスト内の番号を修飾子として指定します。たとえば下記の呼び出し

```
s = p.get_parameter("SearchPath", n);
```

は、SearchPath リスト内の ***n*** 番目の項目を取得します。要求されたカテゴリに対して利用可能な項目の数よりも ***n*** が大きいときは、空文字列が返されます。返された文字列は、何らかの API 関数が次に呼び出されるまで有効です。

3.1.4 PDF 文書をメモリ内に生成

ファイル上に PDF 文書を生成するだけではなく、PDFlib を使ってメモリ内に直接 PDF を生成させることもできます（インコア生成といいます）。この技法は、何らディスクベースの入出力が伴わないために速度の面で利点がありますし、PDF 文書をたとえば直接 HTTP で流したりすることもできます。Web 管理者が聞いて特に喜びそうなのは、自分のサーバにテンポラリ PDF ファイルを散らかされずに済むということでしょう。

生成されるデータは、定期的に少しずつ集めることもできますし（たとえば各ページができるごとに）、最後にまるごと PDF 文書になってからひとかたまりで取り出すこともできます（*PDF_end_document()* の後で）。PDF データの細切れでの生成および消費にはいくつかの利点があります。第一に、データ全体がメモリ内に収まる必要がないので、メモリ必要量が小さくて済みます。第二に、この方式では速度も上がる可能性があります。なぜなら、遅い接続でデータを伝送している場合でも、最初の一切れを送り出している間

に、次の一切れがもう生成中だからです。ただし、生成されるデータの総量は文書全体ができあがるまでわかりません。

createpvf オプションを用いると、PDF データをディスクファイルへ書き込むことなく、メモリ内へ生成し、その後それを PDFlib に渡すことができます（58 ページ「PDF 出力を仮想ファイルに生成」参照）。

能動インコア PDF 生成インタフェース PDF データをメモリ内に生成するには、*PDF_begin_document()* で空のファイル名を指定し、*PDF_get_buffer()* でデータを取り出します。

```
p.begin_document("", "");
...文書を作成...
p.end_document("");

buf = p.get_buffer();
... バッファ内のPDFデータを利用 ...
p.delete();
```

注 バッファ内の PDF データはバイナリデータとして扱う必要があります。

これは「能動」モードと捉えられます。なぜなら、バッファ内容をいつ取り出したいかをクライアント側で決めているからです。能動モードはすべての対応言語バインディングで利用可能です。

注 C と C++ のクライアントでは、返ってきたバッファを解放してはいけません。

受動インコア PDF 生成インタフェース 「受動」モードは、C と C++ の言語バインディングでのみ利用可能です。この場合、ユーザーは、1 つのコールバック関数をインストールします（*PDF_open_document_callback()* を通じて）。この関数は、PDFlib によって、予測不可能なさまざまな時点で呼ばれます。PDF データが消費されるのを待っている時にはいつでも呼ばれることになります。放出（ライブラリからクライアントへの PDF データの転送）に関するタイミング上の、ひいてはバッファ容量上の束縛条件は、クライアント側で設定することができるので、柔軟性が非常に高くなっています。環境によって、PDF 文書をまるごと一度に取り出すのが好都合な場合もあるでしょうし、複数のかたまりに分けるのがよい場合もあるでしょうし、たくさんのコマ切れに分けて PDFlib の内部文書バッファ量を抑えるのが望ましい場合もあるでしょう。放出の手法を設定するには、*PDF_open_document_callback()* で *flush* オプションの値を指定します。

3.1.5 大容量 PDF 文書

多くのユーザーはギガバイト単位の PDF 文書を扱う必要には迫られないでしょうが、業務アプリケーションのなかには、大量の請求書や明細などを含む文書を作成したり処理したりする必要があるものがあります。PDFlib 自体は生成する文書のサイズにいかなる制約も設けていませんが、PDF Reference やいくつかの PDF 規格によって課せられるいくつかの制限があります：

- 2 GB ファイルサイズ制限：PDF/A-1 規格では、ファイルサイズを 2 GB までに制限しています。1 文書がこの制限よりも大きくなるときは、PDFlib は PDF/A-1・PDF/X-4・PDF/X-5 モードでは例外を発生させます。それ以外の場合であれば 2 GB を超える文書を作成できます。
- 10 GB ファイルサイズ制限：PDF 文書は伝統的に、相互参照テーブルによって内部的に、10 進 10 桁すなわち 10^{10}-1 バイトまでに制限されてきました。これはおよそ 9.3 GB に

あたります。しかし、圧縮オブジェクトストリームを用いればこの制約は超えることができます。10 GB　を超える出力文書を作成しようとするときは、*PDF_begin_document()* で *objectstreams={other}* オプションを設定する必要があります。これにはPDF 1.5 以上が必要です。圧縮オブジェクトストリームはいずれにせよファイル全体のサイズを低減させますが、*objectstreams* 実装の一部である圧縮相互参照ストリームはもはや 10 進 10 桁の制限下には置かれず、したがって 10 GB を超える PDF 文書の作成を可能にします。

- オブジェクトの数：一文書内のオブジェクトの数は全般的には PDF によって制限されていませんが、PDF/A-1・PDF/X-4・PDF/X-5 規格では、一文書内の間接オブジェクトの数を 8,388,607 個に制限しています。一文書がこの制限を超えるオブジェクトを必要とするときは、PDFlib は PDF/A-1・PDF/X-4・PDF/X-5 モードでは例外を発生させます。それ以外の場合であればもっとオブジェクトの多い文書を作成できます。生成されたオブジェクトの数は PDFlib で直接取得することはできませんが、クライアントアプリケーションは画像ハンドルを再利用し、画像を**ページ**スコープの外で読み込むことによって、いくらかオブジェクトを減らすことが可能です。PDF 内のオブジェクトの数は、ページ内容の複雑さや、相互参照要素の数などに依存します。シンプルな内容の大容量文書は通常ページあたり 4 ～ 10 個のオブジェクトを持ちますので、100 ～ 200 万ページ程度の文書であればこのオブジェクト制限を超えずに作成することができます。

3.1.6 PDFlib を EBCDIC ベースのプラットフォームで利用

PDF ファイル形式の中のオペレータと制御構造は ASCII ベースであり、z/OS・USS・MVS オペレーティングシステムを搭載した IBM iSeries・zSeries といった EBCDIC ベースのプラットフォームでは正しく動作しません（ただし zLinux は ASCII ベースですのでこの限りではありません）。しかし、特別なメインフレームバージョンの PDFlib を利用すれば、ASCII ベースの PDF オペレータと EBCDIC の（またはそれ以外の）テキスト出力とを混ぜることができます。こうした EBCDIC セーフなバージョンの PDFlib は、さまざまなオペレーティングシステムやマシンアーキテクチャで利用可能です。

PDFlib のさまざまな機能を EBCDIC ベースのプラットフォームで活用するためには、以下に挙げるものは EBCDIC テキスト形式で与えられることが期待されます（より具体的には、iSeries ではコードページ 037 で、zSeries ではコードページ 1047 で）。

- PFA フォントファイル・UPR 設定ファイル・AFM フォントメトリックファイル
- エンコーディングファイル・コードページファイル
- PDFlib 関数の文字列引数
- 入出力ファイル名
- 環境変数（実行環境が対応している場合）
- PDFlib のエラーメッセージも EBCDIC 形式で生成されます（Java 以外）。

ASCII 形式の入力テキストファイル（PFA・UPR・AFM・エンコーディング）を使いたい場合は、*asciifile* パラメタを *true* に設定します（デフォルトは *false*）。すると PDFlib は、これらのファイルが ASCII エンコーディングで書かれていると期待するようになります。ただしその場合でも、文字列引数はやはり EBCDIC エンコーディングであると期待されます。

これに対し、以下のものはつねにバイナリモードで取り扱う必要があります（すなわち、いかなる変換も行ってはいけません）。

- PDF 入出力ファイル
- PFB フォントアウトラインファイル・PFM フォントメトリックファイル

- TrueType・OpenType フォントファイル
- 画像ファイル・ICC プロファイル

3.2 ページ記述

3.2.1 座標系

PDF のデフォルト座標系が PDFlib の内部では用いられています。デフォルト座標系（デフォルトユーザースペースともいう）では、ページの左下隅に原点があり、DTP ポイントを単位として用いています。

```
1 pt = 1/72 inch = 25.4/72 mm = 0.3528 mm
```

1 番目の座標は右へ向かって増加し、2 番目の座標は上へ向かって増加します。PDFlib のクライアントプログラムでは、このデフォルトユーザースペースを回転・拡縮・並行移動・斜形化させることによって、新しいユーザー座標を作ることもできます。こうした変形に対応する関数はそれぞれ *PDF_rotate()*・*PDF_scale()*・*PDF_translate()*・*PDF_skew()* です。座標系を変更した場合、グラフィック・テキスト関数の中の座標はすべて新しい座標系に従って指定しなければなりません。座標系は、各ページの最初でデフォルト座標系に再設定されます。

メートル座標の利用　メートル座標を利用したい場合は簡単で、座標系を拡大すれば利用できます。拡大率は、上記の DTP ポイントの定義より導かれます。

```
p.scale(28.3465, 28.3465);
```

この呼び出しの後は、PDFlib はすべての座標をセンチメートル単位として解釈します（インタラクティブ機能については例外、後述）。これは *72 ÷ 2.54 = 28.3465* だからです。

関連する機能として、*PDF_begin/end_page_ext()* で *userunit* オプションを指定して（PDF 1.6）ページ全体に対する拡縮倍率を与えることもできます。ただしユーザー座標は、Acrobat での最終的なページ表示に対してのみ効力を持つものであり、PDFlib で座標の拡縮を行うものではありません。

クックブック **完全なコードサンプルがクックブックの** general/metric_topdown_coordinates **トピックにあります。**

インタラクティブ要素の座標　インタラクティブ関数には、作成したいテキスト注釈・リンク・ファイル注釈の矩形の座標を与える必要があります。PDF では、ハイパーテキストの関数のための座標はつねにデフォルト座標系で記述されていると見なされます。ユーザー座標系（変形されているかもしれない）で記述されていると見なされることはありません。これは非常にやっかいですので、PDFlib には、ユーザー座標が指定されてもそれを PDF が認める形式に自動変換する機能があります。この自動変換を有効にするには、次のように、*usercoordinates* パラメタを *true* に設定します。

```
p.set_parameter("usercoordinates", "true");
```

リンク・フィールドの矩形としては、その縁がページの縁に平行なものにしか PDF では対応していないので、拡縮・回転・並行移動・斜形化によって座標系が変形しているときには与えられた矩形は形を調整しなければなりません。このような場合、PDFlib では、その四角形を含み、かつ縁がページの縁と平行な最小の矩形を計算します。そしてこれをデフォルト座標に変換し、その結果の値を、与えられた座標のかわりに用います。

要するに大局的に言ってどんな効果があるかといえば、*usercoordinates* パラメタが *true* に設定されていれば、ページ内容に対してもインタラクティブ要素に対しても同じ座標系が使えるということです。

座標の表示　PDFlib のユーザーが PDF の座標系を扱うのを支援するために、PDFlib のディストリビューションには *grid.pdf* という PDF ファイルが含まれています。この PDF ファイルは、よく使われるいくつかのページサイズの座標を表示するものです。望みのサイズのページを何か透明なものに印刷すれば、PDFlib での開発のために有用な道具になるかもしれません。

ページ座標は、Acrobat では下記のようにして表示できます：

- カーソル座標を表示するには下記を用います：
  Acrobat X：**「表示」→「表示切り替え」→「カーソル座標」**
  Acrobat 9：**「表示」→「カーソル座標」**
  Acrobat 8：**「表示」→「ナビゲーションパネル」→「情報」**
- 座標は、Acrobat で現在選択されている単位で表示されます。表示単位を変更するには、Acrobat 8/9/X では次のように操作します：**「編集」→「環境設定」**（**→「一般 ...」**）→ **「単位とガイド」**を選択して、ポイント・インチ・ミリ・パイカ・センチメートルのうちのいずれかを選びます。

ただし、表示される座標系はページの左上隅が原点であり、PDF のデフォルトである左下隅の原点とは異なるので注意が必要です。Acrobat の座標表示と合わせた座標系を選ぶ方法については 69 ページ「下向き座標の利用」を参照してください。

PDF のプリントアウトでページの縦横の長さが正しくないように見えても惑わされないようにしましょう。なぜ正しく出ないのか、よくある原因は以下のとおりです。

- Acrobat の印刷ダイアログの**「ページの拡大 / 縮小 :」**オプションの設定が**「なし」**以外になっているために、印刷出力が拡縮されている。
- 非 PostScript プリンタドライバの場合は、印刷するものの大きさが正確に再現されるとは限らない。

オブジェクトの回転　重要なことは、一度ページ上に描いたものは変更ができないということです。PDFlib には、回転・並行移動・拡縮・斜形化の関数がありますが、こうした関数は、すでに存在しているものに対しては効力を持たず、それ以降に描かれるものに対してだけ効力があります。

テキスト・画像・取り込み PDF ページを回転させるのは簡単で、*PDF_fit_textline()*・*PDF_fit_textflow()*・*PDF_fit_image()*・*PDF_fit_pdi_page()* で *rotate* オプションを指定します。こうしたオブジェクトをそれぞれのはめ込み枠内で 90 度の倍数だけ回転させるのは、これらの関数の *orientate* オプションで可能です。下記の例は傾き 45 度のテキストを生成します。

```
p.fit_textline("回転テキスト", 50.0, 700.0, "rotate=45");
```

クックブック　完全なコードサンプルがクックブックの text_output/rotated_text トピックにあります。

ベクトルグラフィックの回転は、一般の変形関数 *PDF_translate()*・*PDF_rotate()* を利用すれば可能です。下記の例は、左下隅を *(200, 100)* に持つ回転矩形を作成します。描きたい矩形の隅へ座標原点を変更し、座標系を回転させて、矩形を *(0, 0)* に配置しています。以下のように save と restore ではさむことにより、縦置きテキストの作成を完了した後、簡単に元の座標系に戻ってオブジェクトの配置を継続できます。

```
p.save();
        p.translate(200, 100);              /* 原点を矩形の隅へ移動*/
        p.rotate(45.0);                     /* 座標を回転させる */
        p.rect(0.0, 0.0, 75.0, 25.0);       /* 回転された矩形を描く */
```

```
        p.stroke();
p.restore();
```

下向き座標の利用　PDF の上向き座標系とは違って、グラフィック環境のなかには下向き座標を用いているものもあるので、そちらを採用したい開発者もいるでしょう。そのような座標系は PDFlib の変換関数で簡単に設定することができます。ところが、この変換はテキスト出力に対しても効力を持つので（テキストが簡単に上下ひっくり返ります）、テキストが裏返しになってしまわないようにするには、ほかにも何らかの呼び出しを行うことが必要になります。

下向き座標が簡単に利用できるようにするため、PDFlib では、ある特殊なモードに対応しています。このモードでは、すべての関連する座標に対してそれぞれ異なる解釈が適用されます。この *topdown* 機能は、PDFlib ユーザーが下向き座標系でごく自然に作業が行えるようにするために設けられています。具体的には、ページの左下隅に原点 *(0, 0)* があって *y* 座標が上向きに増加するデフォルト PDF 座標系を扱うのではなく、ページの左上隅に原点があって *y* 座標が下向きに増加する修正座標系を用います。ページでこの下向き座標系を利用するには、次のように *PDF_begin_page_ext( )* で *topdown* オプションを指定します。

```
p.begin_page_ext(595.0, 842.0, "topdown");
```

説明の完全を期するため、下向き座標系を設定した場合の効果を以下に詳しく挙げます。

次のような「絶対座標」は、通常と何も変わらないやり方でユーザー座標に翻訳されます。

- 関数の引数のうち、各関数の説明の中で「座標」と書かれているものすべて。例：*PDF_moveto( )* の *x・y*。*PDF_circle( )* の *x・y*。*PDF_rect( )* の *x・y*（しかし *width・height* は含まず！）。*PDF_add_note( )* の *llx・lly・urx・ury*。

「相対座標」の値は、次のように、下向き座標に合うよう内部変換されます。

- テキスト（正の文字サイズを持つもの）は、ページ上端に向かって配置されます。
- マニュアルの中で、矩形や枠などについて「左下隅」と言っている場合は、ページ上でもそうなるように翻訳されます。
- 回転角が指定されている場合は、その回転の中心は依然としてユーザー座標系の原点 *(0, 0)* です。右回り回転はやはり右回りとして表示されます。

クックブック　**完全なコードサンプルがクックブックの** general/metric_topdown_coordinates **トピックにあります。**

3.2.2 ページサイズ

クックブック　**完全なコードサンプルがクックブックの** pagination/page_sizes **トピックにあります。**

規格ページサイズ　*PDF_begin/end_page_ext( )* の *width・height* オプションには、絶対値か、またはシンボリックなページサイズ名を指定できます。後者は、**< 規格 >***.width*・**< 規格 >***.height* の形をとります。ここで **< 規格 >** は標準判型のいずれかです（小文字で。例：*a4.width*）。

ページサイズの限界　PDF や PDFlib では、利用できるページサイズについてはいかなる制約も課していませんが、Acrobat の実装のほうで、ページサイズに関するプログラム的な限界が存在してしまっています。このことから留意すべきなのは、他の PDF インタプリタではもっと大きなサイズやもっと小さなサイズの文書を扱うことができたとしても、

何も怪しむに足らないということです。Acrobat のページサイズ制限を表 3.2 に示します。PDF 1.6 以上では、*PDF_begin/end_page_ext()* で *userunit* オプションを用いて、ページに対するグローバルな拡縮倍率を指定することもできます。

表 3.2 Acrobat の最小・最大ページサイズ

PDF 表示ソフト	最小ページサイズ	最大ページサイズ
Acrobat 4 以上	1/24″ = 3 pt = 0.106 cm	200″ = 14400 pt = 508 cm
Acrobat 7 以上で userunit オプションを指定	3 ユーザー単位	14 400 ユーザー単位 userunit の最大値は 75 000 なので、可能なページサイズは最大 14 400 × 75 000 ＝ 1 080 000 000 ポイント＝ 381 km

さまざまなページサイズボックス PDFlib の開発者の多くは、ページの幅と高さを指定するだけで済みますが、なかには高度なアプリケーションで（とりわけプリプレス業務では）、それ以外の PDF のボックス項目を記述したいときもあるでしょう。PDFlib では、PDF のすべてのボックス項目に対応しています。PDFlib のクライアントで指定できる項目を以下に挙げます（それぞれの定義は PDF リファレンスより）。こうした項目はある種の環境で有用です。

- MediaBox: ページの幅と高さを指定するために用いられ、通常私達がページサイズとしてとらえているものを記述します。
- CropBox: ページの内容が切り抜かれる領域。Acrobat はこのサイズを画面表示と印刷の際に利用します。
- TrimBox: 完成ページの領域を指定（裁ち切り後の）。
- ArtBox: ページ上で意味のある内容が占める領域。これがアプリケーションソフトウェアで利用されることは稀。
- BleedBox: 印刷所環境で出力されるときにページの内容が切り抜かれる領域。印刷所工程での裁ち切りの不正確さを考えに入れて少しゆとりを持たせて囲んでもよい。

PDFlib では、上記のどの値も利用することはなく、ただ出力ファイルに記録する機能を持つだけです。デフォルトでは PDFlib は、ページの幅と高さの指定から MediaBox を生成しますが、それ以外の項目は一切生成しません。以下のコード抜粋は、新しいページを開始させた後、CropBox の 4 つの値を設定するものです。

```
/* カスタムCropBoxを持つ新規ページを開始 */
p.begin_page_ext(595, 842, "cropbox={10 10 500 800}");
```

文書のページ数 PDFlib では、1 つの文書の中に生成するページの数については何も制限はありません。PDFlib は、何十万ページある文書でも Acrobat が効率的に表示していけるような PDF 構造を生成します。

3.2.3 直接パスとパスオブジェクト

パスとは、任意の数の直線・矩形・円・ベジエ曲線・楕円弧でできた輪郭です。パスは、つながっていない部分を複数含むことができます。こうした部分をサブパスといいます。パスに対して実行できる操作を以下に挙げます。

- 描線。パスに沿って線を描きます。クライアントが与えた、描画に関する引数（たとえば色や線幅）を用います。
- 塗り。パスで囲まれた領域全体を塗ります。クライアントが与えた、塗りに関する引数を用います。

- 切り抜き。以後の描画の可視領域を限定します。具体的には、カレント切り抜き領域（デフォルトでは無限定）が、カレント切り抜き領域とパスで囲まれた領域との交差部分にとって替わられます。
- ただパスを終了。見えないパスができます。それでも PDF ファイルの中には存在しています。これが有用な場合は稀です。

直接パス　関数 *PDF_moveto()*・*PDF_lineto()*・*PDF_rect()* 等を用いて、カレントページやその他の内容ストリーム（テンプレート・Type 3 グリフ記述等）へただちに書かれる直接パスを構築することもできます。パスを構築した直後に、それを *PDF_stroke()*・*PDF_fill()*・*PDF_clip()* のいずれか 1 つおよび関連する関数で処理する必要があります。これらの関数はパスを消費し削除します。パスを複数回使う唯一の方法は、*PDF_save()* と *PDF_restore()* を用いることです。

直接パスを作成しておいて上記の操作を何も適用しないとエラーになります。PDFlib のスコープ体系に従えば、クライアントはこの制約に自然に従うことになります。パスの書式属性（色・線幅等）を変えたい場合はかならず、描画操作の開始前に行う必要があります。この規則を一言で言えば「パス記述の途中で、書式を変えてはいけません」。

パスをただ作成しただけではページには何も現れません。次のように、塗りか描線をパスに適用しなければ目に見える結果は得られません。

```
p.setcolor("stroke", "rgb", 1, 0, 0, 0);
p.moveto(100, 100);
p.lineto(200, 100);
p.stroke();
```

たいていのグラフィック関数では、カレント点という概念を利用しています。これは、描画に用いているペンの位置と捉えることができます。

クックブック **完全なコードサンプルがクックブックの** graphics/starter_graphics **トピックにあります。**

パスオブジェクト　パスオブジェクトは、直接パスよりも便利で強力なものです。パスオブジェクトは、パスを構築するためのすべての描画操作をカプセル化します。パスオブジェクトは *PDF_add_path_point()* で作成することができ、あるいは、画像クリッピングパスを含んでいる画像ファイルから抽出することもできます（後述）。*PDF_add_path_point()* では、パス構築を実現するためのいくつかの便利なオプションを使うこともできます。パスオブジェクトを作成したら、それはさまざまな目的に利用することが可能です:

- パスオブジェクトを、*PDF_draw_path()* を用いてページ記述上で利用することができます。すなわち塗ったり、描線したり、クリッピングパスとして用いたりすることができます。
- パスオブジェクトを、テキストフローの回りこみ形状として用いることができます：テキストが任意の形状の内部または外部を回りこむように組まれます（222 ページ「8.2.10 テキストをパス・画像に回り込み」参照）。
- テキストをパス上に配置することもできます。すなわち、キャラクタ群がパスの直線や曲線に沿って並べられます（203 ページ「8.1.7 パス上のテキスト」参照）。
- パスオブジェクトを表セル内に配置することができます。

直接パスと異なり、パスオブジェクトは *PDF_delete_path()* で明示的に削除されるまで複数回利用することができます。パスに関する情報は *PDF_info_path()* で取得できます。下記のコードは、円を含む単純なパス形状を作成し、それをページ上の異なる 2 箇所に描線し、最後にそれを削除しています：

```
path = p.add_path_point(  -1,   0, 100, "move", "");
path = p.add_path_point(path, 200, 100, "control", "");
path = p.add_path_point(path,   0, 100, "circular", "");

p.draw_path(path,   0,   0, "stroke");
p.draw_path(path, 400, 500, "stroke");
p.delete_path(path);
```

パスオブジェクトをいちいち描画操作で作成する方法のほかに、取り込み画像からクリッピングパスを抽出する方法もあります：

```
image = p.load_image("auto", "image.tif", "clippingpathname={path 1}");

/* 画像のクリッピングパスからパスオブジェクトを作成 */
path = (int) p.info_image(image, "clippingpath", "");
if (path == -1)
        throw new Exception("エラー： クリッピングパスが見つかりません!");

p.draw_path(path, 0, 0, "stroke");
```

3.2.4 テンプレート

PDF 内のテンプレート　PDFlib では、技術用語で**フォーム** ***XObject*** と呼ばれる PDF の機能に対応しています。しかしこの用語は、対話的なフォームとまぎらわしいため、私達はこの機能を「テンプレート」と呼びます。PDFlib のテンプレートは、ページ外のバッファと捉えることができ、そこではテキスト・ベクトル・画像の操作が行えます（通常のページ上と同じように）。テンプレートができた後は、まるでラスタ画像のように使うことができ、任意の回数、任意のページに貼り付けることが可能です。画像同様、テンプレートには拡縮や斜形化などの幾何学的変形を施すことができます。1 つのテンプレートを複数のページで使った場合には（ないしは同じページで複数回）、テンプレートを構成する PDF オペレータは実際には PDF ファイル中に 1 回しか書かれていないので、PDF 出力ファイルサイズの節約になります。テンプレートは、複数のページに繰り返し現れるものに対して活用するとよいでしょう。たとえば、毎ページ同じ背景や、企業ロゴや、CAD ソフト・地図作成ソフトの吐き出す図記号などです。これ以外にもテンプレートの代表的な活用例としては、トンボ・商標・外字などがあります。

PDFlib でテンプレートを利用　テンプレートは、ページ記述の外でのみ「定義する」ことができ、そしてページ記述の中で「利用する」ことができます。ただし、テンプレートは他のテンプレートを内包することも可能です。もちろん、テンプレートをそれ自体の定義中で使うことは不可能です。定義済みのテンプレートをページ上から参照するには***PDF_fit_image()*** 関数を使えばよく、画像をページに貼り付けるのとまったく同じです（188 ページ「7.3 画像と取り込み PDF ページの配置」参照）。一般に、PDFlib でテンプレートを利用する場合には以下のようなコードになります。

```
/* テンプレートを定義 */
template = p.begin_template_ext(template_width, template_height, "");
...いろいろなテキスト・ベクトル・グラフィック関数を用いてテンプレート上に描画...
p.end_template_ext(0, 0);
...
p.begin_page(page_width, page_height);
/* テンプレートを利用 */
p.fit_image(template, 0.0, 0.0, "");
...いろいろなページ描画操作を追加...
```

```
p.end_page();
...
p.close_image(template);
```

あらゆるテキスト・グラフィック・色関数がテンプレート上では使えます。ただし、次に挙げる関数は、テンプレートを作成している間には使ってはいけません。

- *PDF_load_image()*：これは大きな制約ではありません。なぜなら画像はテンプレート定義の外で開いておけば、自由にテンプレートの中で使えるからです（開く以外は）。
- すべてのインタラクティブ関数。なぜならこれらは必ず、配置したい文書ページ上で定義しなければならず、テンプレートの一部として生成することはできません。

クックブック　**完全なコードサンプルがクックブックの** general/repeated_contents **トピックにあります。**

3.2.5 外部 PDF 文書内の参照ページ

クックブック　**完全なコードサンプルがクックブックの** pdfx/starter_pdfx5g **トピックにあります。**

PDF 文書は、外部文書内のページへの参照を含むことができます：この（拡縮や回転された）参照ページは文書の一部ではありませんが、他のページ内容と全く同じように表示・印刷されます。これを利用すると、再利用するグラフィック内容（ロゴ・表紙ページ等）を、その PDF データを含めることなく参照することができます。PDFlib は、強い参照に、すなわち、参照ページが内部メタデータを通じて同定される参照に対応しています。参照ページが得られないときや、メタデータと整合しないときは、参照ページではなく代理画像が表示されます。

参照ページ（技術用語では**参照 *XObject***）は、PDF 1.4（すなわち Acrobat 5 のファイル形式）ですでに仕様に記載されていましたが、それを正しく表示・印刷するには Acrobat 9 以上が必要です。参照ページは PDF/X-5g・PDF/X-5pg の重要な要素でもあります。生成される（新しい）文書をコンテナ文書といい、参照ページがある外部 PDF 文書をターゲットファイルといいます。

参照ページを Acrobat 9 か X で使うには、Acrobat を下記のように正しく設定することが重要です：

- **「編集」→「環境設定」→「一般 ...」→「ページ表示」→「参照 *XObject* ターゲットを表示」**：**「つねに」**に設定（設定**「*PDF-X/5* 準拠のもののみ」**は、Acrobat のバグのため動作しません）。
- **「編集」→「環境設定」→「一般　...」→「ページ表示」→「参照されるファイルの場所」**：ターゲットファイルがあるディレクトリの名前を入力。
- **「編集」→「環境設定」→「一般 ...」→「セキュリティ（拡張）」→「セキュリティ特権の場所」→「フォルダのパスを追加」**：コンテナ文書があるディレクトリの名前を追加。これは**「拡張セキュリティを有効にする」**の設定にかかわらず行う必要があります。

ターゲットページは、コンテナ PDF 内部でそのファイル名とページ番号が指定されており、下記の条件すべてを満たすときにのみ代理でなくそれが表示されます：

- コンテナ文書が Acrobat の設定により信頼されている。
- 指定されたディレクトリ内でターゲットファイルが見つかった。
- ターゲットファイルがパスワードを一切要求せず、エラーなく開くことができる。
- コンテナ文書内で指定された参照ページのページ番号がターゲットファイル内に存在している。
- PDF/X-5 のみ：ターゲット内の ID と特定の XMP メタデータ項目群が、コンテナ文書内の対応する項目と整合する必要があります。

これらの条件に 1 つでも違反しているときは、ターゲットページでなく代理が表示されます。Acrobat はいかなるエラーメッセージも出しません。

PDFlib は、別の取り込みページを代理（ターゲットを単純化したもの等）として用いることもできます。あるいは、テンプレートを代理として用いて、たとえば単純な幾何学形状を代理のプレースホルダとすることも可能です：

```
proxy = p.begin_template_ext(0, 0,
        "reference={filename=target.pdf pagenumber=1 strongref=true}");
if (proxy == -1 )
{
        /* エラー */
}
```

3.3 暗号化 PDF

3.3.1 PDF のセキュリティ機能

PDF は、文書の内容を保護する助けとなるさまざまなセキュリティ機能に対応しています。こうしたセキュリティ機能は、対称型暗号化を用いる Acrobat の標準の暗号化ハンドラに基づいています。Acrobat Reader と Acrobat 製品は下記のセキュリティ機能に対応しています：

- 権限：PDF 文書に対し、印刷やテキスト抽出といった特定の操作を制限します。
- ユーザーパスワード：ファイルを開く際に要求させることができます。
- マスターパスワード：権限・ユーザーパスワード・マスターパスワードのうちのいずれかのセキュリティ設定を変更する際に要求させることができます。ユーザーパスワードとマスターパスワードを持つファイルは、いずれかのパスワードで閲覧・印刷が可能になります。
- （PDF 1.6）保護していない文書内で、添付だけを暗号化することもできます。

ファイルが、ユーザーパスワードかマスターパスワードか何らかの権限制限を持つ場合、そのファイルは暗号化されます。

操作権限　「印刷：許可しない」のように、何らかの操作権限を設定すると、Acrobat ではその機能が無効になります。しかしこれは必ずしも、サードパーティの PDF ビューア等のソフトウェアでもこうなるとはかぎりません。操作権限に従うかどうかは PDF ツールの開発者次第なのです。実際、いくつかの PDF ツールは権限設定を全く無視することが知られていますし、商用の PDF クラッキングツールを使えば、いかなる操作制限をも無効化することが可能です。これは暗号化のクラッキングとは関係のない話で、単純に、PDF ファイルが表示可能であるかぎり、それを絶対印刷されないようにすることなどできるはずがないからです。このことは実際、Adobe 自身の PDF リファレンスにも示されています：

There is nothing inherent in PDF encryption that enforces the document permissions specified in the encryption dictionary. It is up to the implementors of PDF viewers to respect the intent of the document creator by restricting user access to an encrypted PDF file according to the permissions contained in the file.（PDF の暗号化には、暗号化辞書で指定された文書権限設定に関して、本来的な強制力は何もありません。暗号化されている PDF ファイルの、ユーザーによる利用を、そのファイルに含まれた権限設定に従って制限することによって、文書の作成者の意図を尊重するかどうかは、PDF ビューアの実装者次第です。）

Unicode パスワード　PDF 1.7 拡張レベル 3（Acrobat 9）は Unicode パスワードに対応しています。それ以前のバージョンでは WinAnsi エンコーディングのキャラクタによるパスワードにしか対応していないのに対し、PDF 1.7 拡張レベル 3 では任意の Unicode キャラクタをユーザー・マスターパスワードで用いることが可能です。

良いパスワード・悪いパスワード　PDF の暗号化の強度は、暗号化の鍵長によってのみ決まるのではなく、パスワードの長さと質によっても決まります。よく知られているのは、名前や普通の単語などをパスワードとして使うべきではないということで、なぜならそれらは簡単に推測ができ、あるいはいわゆる辞書攻撃を使って系統的に割り出されてしまうからです。さまざまな調査が明らかにしているところによれば、かなり多くのパスワードは配偶者やペットの名前、ユーザーの誕生日、子供の愛称などが選ばれており、容易に推測が可能となっています。

PDF の暗号化は、内部的には 40 ビットか 128 ビットの鍵で動作しますが、ユーザーのレベルでは、PDF 1.7 以下ではパスワードの 32 文字までが、PDF 1.7 拡張レベル 3 では

UTF-8 の 127 バイトまでが使われます。PDF 文書を暗号化するのに使われる内部鍵は、ユーザーが与えたパスワードに対して、ある複雑な計算を行うことで導き出されます。パスワードがもしも弱ければ、鍵長にかかわらず、結果として保護も弱くなります。128 ビット鍵も AES 暗号化も、短いパスワードを使えばあまり安全ではありません。

3.3.2 PDFlib による文書保護

暗号化アルゴリズムと鍵長　保護された文書を生成するとき、PDFlib は、クライアントが選んだ PDF 互換レベルで利用可能な最も強い暗号化と鍵長を選びます：

- PDF 1.3（Acrobat 4）では、40 ビット鍵の RC4 が使われます。
- PDF 1.4（Acrobat 5）では、128 ビット鍵の RC4 が使われます。
- PDF 1.5（Acrobat 6）では、128 ビット鍵の RC4 が使われます。これは PDF 1.4 と鍵長が同じですが、暗号化方式は若干異なるものが使われるので、そのために Acrobat 6 が必要です。
- PDF 1.6（Acrobat 7）以上では、128 ビット鍵の AES（Advanced Encryption Standard）が使われます。
- PDF 1.7 拡張レベル 3（Acrobat 9）では、256 ビット鍵の AES（Advanced Encryption Standard）が使われます。これ以前のアルゴリズムでは WinAnsi エンコーディングのキャラクタによるパスワードにしか対応していないのに対し、このバージョンでは Unicode パスワードも使えます。

パスワード　パスワードは、*PDF_begin_document()* の *userpassword*・*masterpassword* オプションで設定することができます。PDFlib は、クライアントが与えたパスワードと下記のように相互作用します：

- ユーザーパスワードと権限（後述）が与えられているのにマスターパスワードが与えられていない場合は、正規ユーザーがセキュリティ設定を変えることができてしまいます。この理由から PDFlib はこの場合をエラーと見なします。
- ユーザーパスワードとマスターパスワードが同じである場合は、ファイルのユーザーと所有者との区別が不可能になってしまいますので、この場合も有効な保護は望めません。PDFlib はこの場合をエラーと見なします。
- ユーザーパスワード・マスターパスワードとも、32 文字まで許されます。空のパスワードは許されません。

与えられたパスワードは、その後に生成されるすべての文書に対して用いられます。

セキュリティ上の推奨事項　最大限のセキュリティを得るために、以下のことを推奨します。

- Acrobat 7 より古いビューアに対応する必要がない限り、AES 暗号化を使うこと。
- マスターパスワードだけを持ち、ユーザーパスワードを持たない文書は、必ずクラック可能です。ですので、ユーザーパスワードをかけることを考慮するべきです（ただしもちろんそのユーザーパスワードは、文書の正当な利用者には渡らなければなりません）。
- パスワードは最低でも 8 文字とし、アルファベット以外の文字を混ぜること。辞書にある言葉や人名・地名をパスワードに使わないこと。

権限　操作制限は *PDF_begin_document()* の *permissions* オプションで設定することができます。それは操作制限の入った文字列（複数可）で構成されます。*permissions* オプションを設定する際には *masterpassword* オプションも設定しなければなりません。なぜなら

そうでなければAcrobatユーザーは簡単に権限設定を取り除くことができてしまうからです。デフォルトではすべての操作が許可されています。操作制限を指定するとAcrobatのその機能は無効になります。操作制限はユーザーパスワードなしで適用することができます。下記の例のようにスペースで区切れば、複数の制限キーワードを指定することもできます：

```
p.begin_document(filename, "masterpassword=abcd1234 permissions={noprint nocopy}");
```

表3.3に、使えるすべての操作制限キーワーを挙げます。

クックブック 完全なコードサンプルがクックブックの general/permission_settings トピックにあります。

表3.3 PDF_begin_document( ) の permissions オプションに対する操作制限キーワード一覧

キーワード	解説
noprint	Acrobatが、ファイルの印刷を拒みます。
nomodify	Acrobatが、ユーザーによるフォームフィールドの追加やその他あらゆる変更を拒みます。
nocopy	Acrobatが、テキストやグラフィックのコピーや抽出を拒み、アクセシビリティインタフェースを無効にします。
noannots	Acrobatが、コメントやフォームフィールドの追加・変更を拒みます。
noforms	（PDF 1.4。nomodify・noannotsを暗黙に前提します）Acrobatが、フォームフィールドへの記入を拒みます。
noaccessible	（PDF 1.4）Acrobatが、アクセシビリティを目的としたテキストやグラフィックの抽出（たとえばスクリーンリーダーソフト）を拒みます。
noassemble	（PDF 1.4。nomodifyを暗黙に前提します）Acrobatが、ページの挿入・削除・回転やしおり・サムネールの作成を拒みます。
nohiresprint	（PDF 1.4）Acrobatが、高解像度印刷を拒みます。noprintが指定されていない場合は、印刷は「画像として印刷」機能に制限されます。すなわちその場合、ページが低解像度に変換されたものを印刷することしかできません。
plainmetadata	（PDF 1.5）暗号化文書でも、文書のメタデータを暗号化しないままにします。これはXMPメタデータにのみ影響し、文書情報フィールドには影響しません。

注 PDFをウェブ上で提供する場合には必ず、クライアントが自分のブラウザを使ってその文書のローカルコピーをとることができてしまいます。ユーザーがローカルコピーを保存することをPDF側から防ぐ方法はありません。

添付ファイルの暗号化 PDF 1.6以上では、文書が保護されていなくても、添付ファイルだけを暗号化することもできます。これを実現するには、*PDF_begin_document()* で *attachmentpassword* オプションを与えます。

3.4 Web 最適化（線形化）PDF

PDFlib は、PDF 文書に線形化という処理を行うことができます（PDF の線形化は**「最適化」・「*Web* 表示用に最適化」**ともいいます）。線形化は PDF ファイルの中身を再配列し、補足情報を追加します。この情報が活用されることでアクセスの高速化が可能になります。

非線形化 PDF はまるごとクライアントへ転送しなければならないのに対し、線形化 PDF は Web サーバからバイトサービングという処理を用いてページごとに転送することが可能です。これにより Acrobat（ブラウザのプラグインとして動作している）は、PDF 文書内の各部分を個別に取得できます。ですので、文書の先頭ページを、その文書がサーバから全部ダウンロードされてくるまでユーザーを待たせることなく表示することができます。これはユーザー体験の向上をもたらします。

ブラウザへ PDF データをストリームするのは Web サーバであって PDFlib ではないことに留意してください。PDFlib がすることは、バイトサービング可能な PDF ファイルを生成することです。バイトサービング PDF を活用するには、下記のすべての必要条件を満たす必要があります：

- PDF 文書が線形化されている必要があります。線形化は ***PDF_begin_document()*** の ***linearize*** オプションで下記のように行うことができます：

  ```
  p.begin_document(outfilename, "linearize");
  ```

  Acrobat で文書のプロパティを見れば、そのファイルが線形化されているかどうかを調べることができます（**「*Web* 表示用に最適化」**：**「はい」**）。
- Web サーバがバイトサービングに対応している必要があります。その基礎であるバイトレンジプロトコルは HTTP 1.1 に含まれていますので、現行のすべての Web サーバに実装されています。
- ユーザーは Acrobat をブラウザのプラグインとして使用するとともに、Acrobat でページごとのダウンロードを有効にしておく必要があります（**「編集」→「環境設定」→（「一般 …」→）「インターネット」→「*Web* 表示用に最適化を許可」**）。なお、これはデフォルトでは有効になっています。

PDF ファイルが大きい（ページ数か MB で数えて）ほど、Web 上で転送した時の線形化の効用は高まります。

注 PDF 文書を線形化すると、ファイルサイズが若干大きくなります。これは、線形化情報が追加されるためです。

線形化に必要な一時領域　PDFlib では、線形化を行うには、まずその文書全体の作成が完了する必要があります。線形化処理は、文書の作成が完了した後に別個の段階として行われるのです。このため、PDFlib で最適化を行うには、追加の格納領域が必要になります。必要な一時領域はおおよそ、生成された文書（線形化前の）と同じ容量です。PDFlib は線形化データを、***PDF_begin_document()*** の ***inmemory*** オプションに従って、メモリ内か一時ディスクファイルのどちらかに格納します。

3.5 色を扱う

注 PDFlib API リファレンスに、対応している色空間の一覧と説明があります。

クックブック 色の諸側面に関するコードサンプルが PDFlib クックブックの color カテゴリにあります。色空間の使用の概要については、クックブックの color/starter_color トピックを参照してください。

3.5.1 パターンとスムーズシェーディング

単色の色のかわりに、パターンやスムーズシェーディングを特殊な色として利用することもできます。任意の物の描線や塗りに使用できます。

パターン パターンは、任意の数の塗り操作を 1 つの実体にまとめたものによって定義されます。このグループは任意の他の物の塗りや描線に用いることができ、塗りの場合は全域内に、描線の場合はパス上にグループが複製（縦横に並ぶ）されます。パターンを使った作業では次の手順を踏みます。

- まず、*PDF_begin_pattern()* と *PDF_end_pattern()* の間でパターンを定義しなければなりません。パターンの定義にはたいていの画像オペレータが利用できます。
- *PDF_begin_pattern()* によって返されたパターンハンドルは、*PDF_setcolor()* を使用して、パターンをカレント色として設定するために用いることができます。

PDF_begin_pattern() の *painttype* 引数によって、パターン定義がそれ自身の色指定を含むことができるかどうかが決まります。*painttype* が 1 の場合、パターン定義はそれ自身の色指定を含まなければならず、つねに見た目が同じになります。*painttype* が 2 の場合、パターン定義は色指定を一切含んでいてはなりません。そのパターンが塗りや描線に使われるときには、カレントの塗りや描線の色が適用されます。

注 パターンはスムーズシェーディングに基づいて定義することもできます（後述）。

クックブック 完全なコードサンプルがクックブックの graphics/fill_pattern・images/tiling_pattern トピックにあります。

スムーズシェーディング スムーズシェーディングは、カラーブレンドやグラディエントともいい、ある色から別の色へ連続的に遷移するもののことを言います。2 つの色は同じ色空間で指定する必要があります。PDFlib はスムーズシェーディングについて、次の 2 種類の方向に対応しています。

- 線形シェーディング。線に沿って定義されます。
- 放射シェーディング。2 つの円の間に定義されます。

シェーディングは 2 つの色の間の遷移として定義されます。1 番目の色にはつねにカレント塗り色が用いられます。2 番目の色は *PDF_shading()* の *c1・c2・c3・c4* 引数で与えられます。この数値は、*PDF_setcolor()* の記述に従った 1 番目の色の色空間で解釈されます。

PDF_shading() を呼び出すと、シェーディングオブジェクトのハンドルが返されます。これは次の 2 つの方式のいずれかで利用することができます。

- *PDF_shfill()* で領域を塗ります。このメソッドを使うことができるのは、塗りを適用したい対象の形がシェーディングの形と同じである場合です。この関数はその名前と違って、物の内部を塗るだけではなく、その外部に対しても効力を持ちます。この動作は *PDF_clip()* で変更できます。

- より複雑な物の塗りに用いるシェーディングパターンを定義します。具体的には、*PDF_shading_pattern()* を呼び出してシェーディングに基づいたパターンを作成し、このパターンを任意の物の塗りや描線に用います。

クックブック 完全なコードサンプルがクックブックの color/color_gradient トピックにあります。

3.5.2 Pantone・HKS・カスタムスポットカラー

PDFlib はスポットカラーに対応しています（技術的には、PDF では特色（Separation）色空間として知られているのですが、通常、separation という用語はプロセスカラーに対しても用いられます）。スポットカラーとは、プロセスカラーの混色領域外にあるカスタム色の印刷に用いられるものです。スポットカラーは名前で指定され、PDF ではつねに代替色を伴います。代替色は、そのスポットカラーに近い色が選ばれますが、まったく同じ色ではありません。Acrobat ではこの代替色を用いて、画面表示やスポットカラー非対応機（事務用プリンタなど）への印刷を行います。印刷機では、要求されたスポットカラーが適用され、文書内で用いられているその他すべてのプロセスカラーとともに印刷されます。このために PDF ファイルは、色分版という処理で後処理されることが必要になります。

PDFlib は、さまざまな組み込みスポットカラーライブラリにも対応していますし、カスタム（ユーザー定義）スポットカラーにも対応しています。スポットカラー名が *PDF_makespotcolor()* によって要求されると、PDFlib はまず、その要求されたスポットカラーが PDFlib 組み込みライブラリのうちのいずれかの中で見つかるかどうかを調べます。もし見つかれば、PDFlib は代替色に関して、組み込まれた値を用います。見つからない場合、そのスポットカラーはユーザー定義色と見なされるので、クライアント側でそれに対して適切な代替色値を与える必要があります（カレント色を通じて）。スポットカラーには濃度を指定することができます。すなわち、0 から 1 までのパーセント値とともに用いることができます。

デフォルトでは、組み込みスポットカラーはカスタム代替値で再定義することはできません。しかし、この動作は *spotcolorlookup* パラメタで変更することができます。これを使うと、古いアプリケーションが異なる色定義を用いているような場合との互換性がとれますし、また、PANTONE カラーに対する PDFlib の Lab 代替値を扱うことのできないワークフローにおいても有用です。

PDFlib は、PDF/X か PDF/A の準拠レベルが選択されていると、自動的に適切な代替色を生成します（253 ページ「10.3 PDF/X による印刷出力」参照）。カスタムスポットカラーに関しては、選択された PDF/X か PDF/A の準拠レベルと互換の代替色を与えるのはユーザー側の役割となります。

注 組み込みスポットカラーのデータと各商標については、PDFlib ソフトウェアでの使用のためのライセンスを PDFlib GmbH は各商標権者から取得しています。

クックブック 完全なコードサンプルがクックブックの color/spot_color トピックにあります。

PANTONE® カラー PANTONE カラーは世界的に有名で広く利用されています。PDFlib は、Pantone Matching System®（スウォッチ総数 26,000 色）に加えて、2008 年に導入されたコート紙 / 上質紙用の追加 2058 色を持つ Pantone® Goe® System に完全対応しています。表 3.4 に示すデジタルカラーライブラリにあるすべてのカラースウォッチ名が利用できます。商用 PDFlib のお客様は、PANTONE スポットカラーの全一覧のテキストファイルを、私達のサポートから得ることができます。

スポットカラー名は大文字・小文字を区別します。上記の例と同様に大文字を使ってください。旧形式の色名接頭辞 CV・CVV・CVU・CVC・CVP も使用することができます。これらは、対応する新しい色名に変換されます。ただし、*preserveoldpantonenames* パラメタが *true* の場合には変換されません。PANTONE という接頭辞は必ず、例示したようにスウォッチ名につける必要があります。一般に、PANTONE カラー名は次の方式に従って構成する必要があります。

```
PANTONE <id> <paperstock>
```

ここで *<id>* は色の識別子であり（たとえば 185）、*<paperstock>* は利用するペーパーストックの頭文字です（たとえば *C* は coated ＝コート紙）。スウォッチ名を構成する各要素の間にはスペースを 1 つずつ入れる必要があります。PANTONE 接頭辞ではじまる名前のスポットカラーが要求されたにもかかわらず、その名前が有効な PANTONE カラーを表していなかった場合には、関数は失敗します。次のコードは、ある PANTONE カラーを濃度値 70 パーセントで用いた例です。

```
spot = p.makespotcolor("PANTONE 281 U");
p.setcolor("fill", "spot", spot, 0.7, 0, 0);
```

注 ここで示した PANTONE® カラーは PANTONE の定義標準と一致しないことがあります。正確な色については現在の PANTONE カラー発行物を参照してください。PANTONE® およびその他の Pantone, Inc. の諸商標は Pantone, Inc. の所有物です。© Pantone, Inc., 2003.

注 PANTONE® カラーは PDF/X-1a モードでは対応していません。

表 3.4 PDFlib に内蔵の PANTONE スポットカラーライブラリ一覧

カラーライブラリ名	色名の例	注
PANTONE ソリッド／コート紙	PANTONE 185 C	
PANTONE ソリッド／上質紙	PANTONE 185 U	
PANTONE ソリッド／マットコート紙	PANTONE 185 M	
PANTONE プロセス／コート紙	PANTONE DS 35-1 C	
PANTONE プロセス／上質紙	PANTONE DS 35-1 U	
PANTONE プロセス／コート紙（EURO）	PANTONE DE 35-1 C	
PANTONE プロセス／上質紙（EURO）	PANTONE DE 35-1 U	2006 年 5 月追加
PANTONE パステル／コート紙	PANTONE 9461 C	2006 年追加分の新色を含む
PANTONE パステル／上質紙	PANTONE 9461 U	2006 年追加分の新色を含む
PANTONE メタリック／コート紙	PANTONE 871 C	2006 年追加分の新色を含む
PANTONE カラーブリッジ CMYK（PC）	PANTONE 185 PC	PANTONE ソリッドをプロセス／コート紙に置き換え
PANTONE カラーブリッジ CMYK（EURO）	PANTONE 185 EC	PANTONE ソリッドをプロセス／コート紙（EURO）に置き換え
PANTONE カラーブリッジ／上質紙	PANTONE 185 UP	2006 年 7 月追加

表 3.4 PDFlib に内蔵の PANTONE スポットカラーライブラリー覧

カラーライブラリ名	色名の例	注
PANTONE ヘキサクローム／コート紙	PANTONE H 305-1 C	非推奨。廃止予定
PANTONE ヘキサクローム／上質紙	PANTONE H 305-1 U	非推奨。廃止予定
PANTONE ソリッド イン ヘキサクローム／コート紙	PANTONE 185 HC	
PANTONE ソリッド トゥ プロセス／コート紙	PANTONE 185 PC	PANTONE カラーブリッジ CMYK（PC）で置き換え
PANTONE ソリッド トゥ プロセス／コート紙（EURO）	PANTONE 185 EC	PANTONE カラーブリッジ CMYK（EURO）で置き換え
PANTONE Goe ／コート紙	PANTONE 42-1-1 C	2008 年追加の 2058 色
PANTONE Goe ／上質紙	PANTONE 42-1-1 U	2008 年追加の 2058 色

HKS® カラー　HKS カラーシステムはドイツなどの欧州諸国で広く利用されています。PDFlib は HKS カラーに完全対応しています。以下のデジタルカラーライブラリ（*Farbfächer*）にあるカラースウォッチ名がすべて利用可能です（カッコ内にサンプルスウォッチ名を示します）。

- HKS K *(Kunstdruckpapier)*、グロスアート紙用、88 色 *(HKS 43 K)*
- HKS N *(Naturpapier)*、ナチュラル紙用、86 色 *(HKS 43 N)*
- HKS E *(Endlospapier)*、連続ステーショナリ用 / コート、88 色 *(HKS 43 E)*
- HKS Z *(Zeitungspapier)*、ニュースプリント用、50 色 *(HKS 43 Z)*

商用 PDFlib のお客様は、HKS スポットカラーの全一覧のテキストファイルを、私達のサポートから得ることができます。

スポットカラー名は大文字・小文字を区別します。上記の例と同様に大文字を使ってください。HKS という接頭辞は必ず、例示したようにスウォッチ名につける必要があります。一般に、HKS カラー名は次の方式のうちのいずれかに従って構成する必要があります。

```
HKS <id> <paperstock>
```

ここで *<id>* は色の識別子であり（たとえば 43）、*<paperstock>* は利用するペーパーストックの頭文字です（たとえば ***N*** は natural 紙）。スウォッチ名を構成する ***HKS・<id>・<paperstock>*** 各要素の間にはスペースを 1 つずつ入れる必要があります。上記 2 番目の方式はライブラリかライブラリの中の色に対してのみ用いることができます。HKS 接頭辞ではじまる名前のスポットカラーが要求されたにもかかわらず、その名前が有効な HKS カラーを表していなかった場合には、関数は失敗します。次のコードは、ある HKS カラーを濃度値 70 パーセントで用いた例です。

```
spot = p.makespotcolor("HKS 38 E");
p.setcolor("fill", "spot", spot, 0.7, 0, 0);
```

ユーザー定義スポットカラー　上述の組み込みスポットカラーのほかに、PDFlib ではカスタムスポットカラーにも対応しています。カスタムスポットカラーには任意の名前と（ただし組み込みの色とかち合う名前は使えません）代替色を与えることができます。こ

の代替色は、画面表示と低品位印刷に用いられますが、高品位の色分版には用いられません。カスタムスポットカラーに適切な代替色を与えるのはクライアント側の役割です。

新規スポットカラーに代替色を設定するための専用の PDFlib 関数というものはありません。そのかわりに、カレント塗り色が用いられます。代替色設定のために呼び出しが 1 つ多く必要であるという点を除けば、カスタムスポットカラーの定義と利用は、以下のように、組み込みスポットカラーの利用と同様に行うことができます。

```
p.setcolor("fill", "cmyk", 0.2, 1.0, 0.2, 0);      /* 代替CMYK値を定義 */
spot = p.makespotcolor("CompanyLogo");             /* そこからスポットカラーを作成 */
p.setcolor("fill", "spot", spot, 1, 0, 0);         /* スポットカラーを設定 */
```

3.5.3 カラーマネジメントと ICC プロファイル

PDFlib は、デバイス独立色・レンダリングインテント・ICC プロファイルといった、いくつかのカラーマネジメントの概念に対応しています。

クックブック **完全なコードサンプルがクックブックの** `color/iccprofile_to_image` **トピックにあります。**

デバイス独立 CIE L*a*b* カラー *PDF_setcolor()* に色空間名 *lab* を与えれば、デバイス独立な色値を CIE 1976 $L^*a^*b^*$ 色空間で指定することができます。$L^*a^*b^*$ 色空間の色は、範囲 0 ～ 100 の輝度値 1 個と、範囲 −127 ～ 128 の色値 2 個とで指定されます。*lab* 色空間に用いられる光源は D50 です（日中光 5000 K、2° 測定）。

レンダリングインテント PDFlib クライアントがデバイス独立な色値を指定できるとはいっても、ある出力デバイスがその要求された色を正確に再現できるとは限りません。そのような場合には、色域圧縮という処理による代替を行うという妥協が必要になります。色域圧縮とは、色の分布範囲を狭めて、特定のデバイスで再現できるような小さな範囲にすることです。レンダリングインテントはこの処理を制御するために用いられます。レンダリングインテントを指定するには、個々の画像に対しては、*PDF_load_image()* に *renderingintent* パラメタまたはオプションを与えます。また、レンダリングインテントはテキストやベクトルグラフィックに対しても指定することができ、そのためには *PDF_create_gstate()* に *renderingintent* オプションを与えます。

ICC プロファイル International Color Consortium（ICC）[1] は、入力デバイスや出力デバイスの色の特徴を指定するためのファイル形式を定義しました。この ICC カラープロファイルは、工業標準と捉えられており、すべての主要なカラーマネジメントシステムとアプリケーションのベンダーがこれに対応しています。PDFlib は次の領域での ICC プロファイルによるカラーマネジメントに対応しています。

- ページ上のテキストとベクトルグラフィックのための、ICC ベースの色空間を定義。
- 取り込んだ画像ファイルに埋め込まれている ICC プロファイルを処理。
- 取り込んだ画像ファイルにICCプロファイルを適用（画像にICCプロファイルが埋め込まれていた場合は上書き）。
- グレースケール・RGB・CMYK のデータを ICC ベースの色空間へ対応づけるデフォルト色空間を定義。
- 外部 ICC プロファイルを使用して PDF/X か PDF/A の出力インテントを定義。

カラーマネジメントは色指定の中の要素数を変えません（たとえば RGB から CMYK へ）。

1. *www.color.org* を参照。

注 主要な印刷環境のための ICC カラープロファイルは、www.pdflib.com でダウンロードできるほか、その他の無料 ICC プロファイルへのリンクも掲載しています。

ICC プロファイルの検索 PDFlib は、*PDF_load_iccprofile( )* に与えられた *profilename* 引数を用い、次の手順を踏んで ICC プロファイルを検索します。

- *profilename=sRGB* ならば、PDFlib はその内部 sRGB プロファイル（後述）を用い、検索は打ち切られます。
- *ICCProfile* リソースカテゴリ内に *profilename* という名前のリソースがあるかどうかを調べます。もしあれば、その値をファイル名として以下の手順で用います。そのようなリソースがない場合は、*profilename* をファイル名として直接用います。
- 前の手順で決定されたファイル名を用い、下記の組み合わせを 1 つずつ順に試してみることにより、ディスク上のファイルを検索します：

```
<ファイル名>
<ファイル名>.icc
<ファイル名>.icm
<colordir>/<ファイル名>
<colordir>/<ファイル名>.icc
<colordir>/<ファイル名>.icm
```

Windows では *colordir* は、オペレーティングシステムがデバイス依存 ICC プロファイルを格納しているディレクトリを示します（*C:\WINNT\system32\ spool\drivers\color* が典型的）。Mac OS X では *colordir* として下記のパスが試みられます：

```
/System/Library/ColorSync/Profiles
/Library/ColorSync/Profiles
/Network/Library/ColorSync/Profiles
~/Library/ColorSync/Profiles
```

その他のシステムでは *colordir* を用いる手順は省略されます。

sRGB 色空間と sRGB ICC プロファイル PDFlib は、sRGB（元 IEC 61966-2-1）と呼ばれる工業規格の RGB 色空間に対応しています。sRGB は、さまざまなソフトウェアやハードウェアのベンダーがこれに対応しており、デジタルスチルカメラのような消費者向け RGB デバイスやカラープリンタ・モニタのような事務機器における簡単なカラーマネジメントのために広く利用されています。PDFlib は sRGB 色空間に対応しており、必要な ICC プロファイルデータを内蔵しています。ですから sRGB プロファイルをクライアントが別途用意する必要はなく、あえて用意しなくてもつねに利用可能です。これを利用するには、*PDF_load_iccprofile( )* を *profilename=sRGB* で呼び出します。

画像に埋め込まれているプロファイルの利用（ICC タグ付き画像） 画像のなかには、その画像の色値の特徴を記述した ICC プロファイルが埋め込まれていることがあります。たとえば、埋め込まれた ICC プロファイルは、画像データの生成に用いられたスキャナの色特性を記述することができます。PDFlib では、PNG・JPEG・TIFF 各画像ファイル形式の中に埋め込まれた ICC プロファイルを扱うことができます。*honoriccprofile* オプションまたはパラメタが true に設定されている場合（デフォルトではそうなっています）、画像内に埋め込まれている ICC プロファイルはその画像から抽出され、PDF 出力内に埋め込まれて、Acrobat がその画像に適用できるようにされます。この処理は、画像への ICC プロファイルのタグ付けと呼ばれることもあります。PDFlib は画像のピクセル値に変更は加えません。

image:iccprofile パラメタを使うと、画像内に埋め込まれているプロファイルに対するICC プロファイルハンドルを得ることができます。これは、同じプロファイルを複数の画像に適用したい場合に有用です。

未知の ICC プロファイル内の色要素数を調べるには *icccomponents* パラメタを用います。

外部 ICC プロファイルの画像への適用（タグ付け） 画像に埋め込まれている ICC プロファイルを使うのではなく、外部プロファイルを各画像に適用することもできます。そのためには *PDF_load_image()* の *iccprofile* オプションでプロファイルハンドルを与えます。

ページ記述用の ICC ベースの色空間 テキストやベクトルグラフィックの色値は、プロファイルによって指定される ICC ベースの色空間で直接指定することができます。まずは色空間を、*setcolor:iccprofilegray*・*setcolor:iccprofilergb*・*setcolor:iccprofilecmyk* のうちのいずれかのパラメタの値としてICCプロファイルハンドルを与えることによって設定する必要があります。つづいて次のように、ICC ベースの色値を、*iccbasedgray*・*iccbasedrgb*・*iccbasedcmyk* のうちのいずれかの色空間キーワードとともに、*PDF_setcolor()* に与えることができます。

```
p.set_parameter("errorpolicy", "return");
icchandle = p.load_iccprofile(...);
if (icchandle == -1)
{
        return;
}
p.set_value("setcolor:iccprofilecmyk", icchandle);
p.setcolor("fill", "iccbasedcmyk", 0, 1, 0, 0);
```

ICC ベースのデフォルト色空間へデバイスカラーを対応づけ PDF には、ページ記述中のデバイス依存なグレー・RGB・CMYK の色をデバイス独立色に対応づける機能があります。これを利用すると、そのままではデバイス依存な色値に対して、正確な測色指定を与えることができます。この方式で色値を対応づけさせるには、DefaultGray・DefaultRGB・DefaultCMYK のうちのいずれかの色空間定義を与える必要があります。PDFlib でこれを実現するには、*PDF_begin_page_ext()* の *defaultgray*・*defaultrgb*・*defaultcmyk* オプションを設定して、ICC プロファイルハンドルをその値として与えます。下記の作成例では、sRGB 色空間を、テキスト・画像・ベクトルグラフィックのデフォルト RGB 色空間として設定しています：

```
/* sRGBはつねに利用できることが保証済 */
icchandle = p.load_iccprofile("sRGB", 0, "usage=iccbased");
p.begin_page_ext(595, 842, "defaultrgb=" + icchandle);
```

PDF/X・PDF/A のための出力インテントを定義 出力デバイス（プリンタ）のプロファイルを用いて、PDF/X のための出力条件を指定することができます。そのためには、*PDF_load_iccprofile()* への呼び出しで *usage=outputintent* を指定します。PDF/A に対しては、あらゆる種類のプロファイルを出力インテントとして指定できます。詳しくは 253 ページ「10.3 PDF/X による印刷出力」と 261 ページ「10.4 PDF/A によるアーカイビング」を参照してください。

3.6 さまざまなインタラクティブ要素

クックブック　インタラクティブ要素を作成するコードサンプルが PDFlib クックブックの interactive カテゴリにあります。

3.6.1 リンク・しおり・注釈

この項では、しおり・フォームフィールド・注釈といったさまざまなインタラクティブ要素の作成方法を説明します。この項で作成するつもりのインタラクティブ要素がすべてできあがった完成文書を図 3.1 に示します。この文書には以下のインタラクティブ要素があります。

- 右上には、テキスト *www.kraxi.com* の所に、*www.kraxi.com* への非表示の Web リンクがあります。この領域をクリックすると、指定された Web ページが表示されます。
- 灰色のフォームフィールドが、種類はテキストで、Web リンクの下に作ってあります。JavaScript を使ってここには今日の日付が自動的に記入されます。
- 赤い押しピンは添付を持った注釈です。クリックするとファイル添付が開きます。
- 左下にはフォームフィールドがあり、種類はボタンで、プリンタのアイコンを表示しています。このボタンをクリックすると Acrobat のメニュー項目**「ファイル」→「印刷」**が実行されます。
- ナビゲーションパネルにはしおり「Our Paper Planes Catalog」があります。このしおりをクリックすると、別の PDF 文書のページが表示されます。

以下、こうしたインタラクティブ要素を PDFlib で作成する方法を詳しく説明します。

Web リンク　まずは、Web サイト *www.kraxi.com* へのリンクを作りましょう。これは 3 つの段階で達成できます。まず、Web リンクをつけたいテキストを配置します。*matchbox* オプションで *name=kraxi* を指定して、テキストのはめ込み枠を後で参照できるようにしておきます。

次に、*URI* 型（Acrobat では**「*Web* ページを開く」**）のアクションを作成します。するとアクションハンドルが得られます。このアクションハンドルは後でインタラクティブ要素（複数可）に割り当てることができます。

最後に、リンクを実際に作成します。PDF ではリンクは *Link* 型の注釈です。リンクに対する *action* オプションでは、イベント名 *activate* を指定してアクションをトリガさせ、加えて上記で作成した *act* ハンドルをアクション内容として与えます。デフォルトではリ

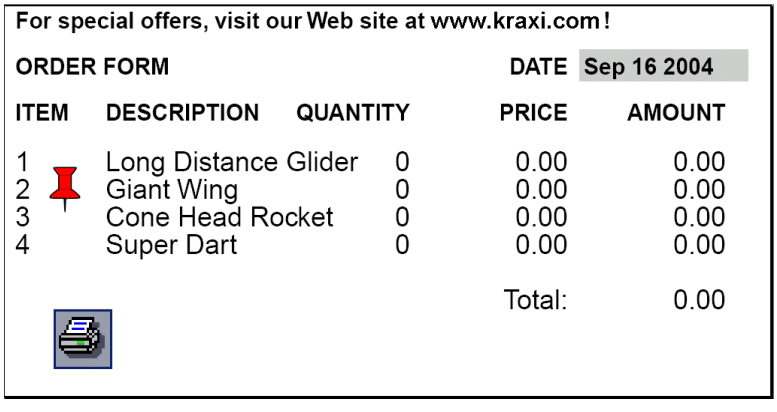


図 3.1
いろいろなハイパーテキスト要素を持つ文書

ンクは細い黒枠線がついて表示されます。最初のうちはこのほうが位置合わせを正確にできて便利ですが、ここでは *linewidth=0* で枠線を非表示にしてあります。

```
normalfont = p.load_font("Helvetica", "unicode", "");
p.begin_page_ext(pagewidth, pageheight, "topdown");

/* テキスト行「Kraxi Systems, Inc.」を配置。範囲枠を使用 */
String  optlist =
        "font=" + normalfont + " fontsize=8 position={left top} " +
        "matchbox={name=kraxi} fillcolor={rgb 0 0 1} underline";

p.fit_textline("Kraxi Systems, Inc.", 2, 20, optlist);

/* URIアクションを作成 */
optlist = "url={http://www.kraxi.com}";
int act = p.create_action("URI", optlist);

/* 範囲枠「kraxi」上にLink注釈を作成 */
optlist = "action={activate " + act + "} linewidth=0 usematchbox={kraxi}";
/* 矩形座標の0は範囲枠の座標に置き換えられる */
p.create_annotation(0, 0, 0, 0, "Link", optlist);

p.end_page_ext("");
```

画像やテキストフローの一部への Web リンクの作成例については 242 ページ「8.4 範囲枠」を参照してください。

クックブック **完全なコードサンプルがクックブックの** interactive/link_annotations **トピックにあります。**

別のファイルに移動するしおり　今度は、別のPDFファイルに移動するしおり「Our Paper Planes Catalog」を作成しましょう。ファイル名は *paper_planes_catalog.pdf* であるものとします。まず、*GoToR* 型のアクションを作成します。このアクションに対するオプションリストで、移動先文書の名前を *filename* オプションで定義します。また *destination* オプションで、拡大表示させたいページ内の一部分を指定します。具体的には、表示されるのは文書の 2 ページ目で（*page 2*）、位置・倍率指定表示で（*type fixed*）、ページの中頃が表示され（*left 50 top 200*）、表示倍率 200%（*zoom 2*）となります。

```
String  optlist =
        "filename=paper_planes_catalog.pdf " +
        "destination={page 2 type fixed left 50 top 200 zoom 2}";

goto_action = p.create_action("GoToR", optlist);
```

次の段階として、実際にしおりを作成します。このしおりに対する *action* オプションで、アクションのトリガとして *activate* イベントを指定し、起こしたいアクションとして上で作成した *goto_action* ハンドルを指定します。*fontstyle bold* オプションで太字のテキストを指定し、*textcolor {rgb 0 0 1}* でしおりを青くします。しおりのテキスト「Our Paper Planes Catalog」は関数の引数として与えます。

```
String  optlist=
        "action={activate " + goto_action + " } fontstyle=bold textcolor={rgb 0 0 1}";

catalog_bookmark = p.create_bookmark("Our Paper Planes Catalog", optlist);
```

このしおりをクリックすると、移動先文書内のページの指定部分が表示されます。

クックブック **完全なコードサンプルがクックブックの** interactive/nested_bookmarks **トピックにあります。**

ファイル添付による注釈　次に、ファイル添付を作成します。まず *FileAttachment* 型の注釈を作成します。*filename* オプションで添付の名前を指定し、*mimetype image/gif* オプションでその種類を指定します（MIME はファイル内容の分類のために広く使われている記述方式）。このしおりは押しピンとして表示され（*iconname pushpin*）、色は赤で（*annotcolor {rgb 1 0 0}*）、説明を持ちます（*contents {Get the Kraxi Paper Plane!}*）。印刷されません（*display noprint*）。

```
String  optlist =
        "filename=kraxi_logo.gif mimetype=image/gif iconname=pushpin " +
        "annotcolor={rgb 1 0 0} contents={Get the Kraxi Paper Plane!} display=noprint";

p.create_annotation(left_x, left_y, right_x, right_y, "FileAttachment", optlist);
```

なお、*iconname* で定義したアイコンの大きさは変化しません。アイコンはその標準サイズのまま、指定した矩形の左上隅に表示されます。

3.6.2 フォームフィールド・JavaScript

印刷ボタンフォームフィールド　次に、文書の印刷に使えるボタンフォームフィールドを作成します。最初のバージョンではボタンにラベルをつけておきます。その後でラベルをやめてプリンタのアイコンを使いましょう。まず *Named* 型（Acrobat では**「メニュー項目を実行」**）のアクションを作成します。また、ラベルのフォントも指定しておく必要があります。

```
print_action = p.create_action("Named", "menuname=Print");
button_font = p.load_font("Helvetica-Bold", "unicode", "");
```

このボタンフォームフィールドに対する *action* オプションで、アクション実行のトリガとして *up* イベント（Acrobat では**「マウスボタンを放す」**）を指定し、アクションそのものとして上で作成した *print_action* ハンドルを指定します。*backgroundcolor {rgb 1 1 0}* オプションで背景を黄色に指定し、*bordercolor {rgb 0 0 0}* で枠線を黒に指定します。オプション *caption Print* でボタンにテキスト *Print* をつけ、*tooltip {Print the document}* でユーザーのための追加説明を作成します。*font* オプションで、上で作成した *button_font* ハンドルを用いてフォントを指定します。デフォルトでは、ラベルのサイズはボタンの領域にちょうど収まるよう自動調整されます。そして、実際にボタンフォームフィールドを作成する際に、適当な座標と、名前 *print_button*、*pushbutton* 型、適切なオプション群を指定します。

```
String  optlist =
        "action {up " + print_action + "} backgroundcolor=yellow " +
        "bordercolor=black caption=Print tooltip={Print the document} font=" +
        button_font;

p.create_field(left_x, left_y, right_x, right_y, "print_button", "pushbutton", optlist);
```

それでは、この最初のバージョンのボタンを改良して、テキスト *Print* をやめて小さいプリンタアイコンに替えてみましょう。これを達成するには、その画像ファイル *print_*

icon.jpg をテンプレートとしてページ作成前に読み込みます。*icon* オプションを用いてテンプレートハンドル *print_icon* をボタンフィールドに割り当てつつ、上記のコードと同様にフォームフィールドを作成します。

```
print_icon = p.load_image("auto", "print_icon.jpg", "template");
if (print_icon == -1)
{
        /* エラー処理 */
        return;
}
p.begin_page_ext(pagewidth, pageheight, "");
...
String  optlist = "action={up " + print_action + "} icon=" + print_icon +
        " tooltip={Print the document} font=" + button_font;

p.create_field(left_x, left_y, right_x, right_y, "print_button", "pushbutton", optlist);
```

クックブック **完全なコードサンプルがクックブックの** `interactive/form_pushbutton` **トピックにあります。**

単純なテキストフィールド　今度は、テキストフィールドをページ右上隅付近に作成します。ユーザーは今日の日付をこのフィールドに入力することができます。フォントハンドルを取得し、*textfield* 型のフォームフィールドを作成して名前を *date*、背景を灰色とします。

```
textfield_font = p.load_font("Helvetica-Bold", "unicode", "");
String  optlist = "backgroundcolor={gray 0.8} font=" + textfield_font;
p.create_field(left_x, left_y, right_x, right_y, "date", "textfield", optlist);
```

デフォルトでは文字サイズは *auto* であり、この場合フィールドの高さがそのまま初期の文字サイズとなります。入力がフィールドの終わりまで達すると文字サイズは小さくなって、テキストがつねにフィールドに収まるよう自動調整されます。

クックブック **完全なコードサンプルがクックブックの** `interactive/form_textfield_layout`・`interactive/form_textfield_height` **トピックにあります。**

JavaScript を持つテキストフィールド　上で作成したテキストフォームフィールドを改良して、ページを開くと自動的に今日の日付が記入されるようにしましょう。第一段階として、*JavaScript* 型（Acrobat では「*JavaScript* を実行」）のアクションを作成します。このアクションのオプションリストの中の *script* オプションで JavaScript スニペットを定義します。このスニペットは、今日の日付を *date* テキストフィールドに月日年形式で表示します。

```
String  optlist =
        "script={var d = util.printd('mmm dd yyyy', new Date()); "
        "var date = this.getField('date'); date.value = d;}"

show_date = p.create_action("JavaScript", optlist);
```

第二段階として、ページを作成します。オプションリストで *action* オプションを与え、その中で、上で作成した *show_date* アクションをトリガイベント *open*（Acrobat では「ページを開く」）に対して設定します。

```
String optlist = "action={open " + show_date + "}";
p.begin_page_ext(pagewidth, pageheight, optlist);
```

最終段階として、上と同様にテキストフィールドを作成します。ページを開くたび、ここに自動的に今日の日付が記入されます。

```
textfield_font = p.load_font("Helvetica-Bold", "winansi", "");
String  optlist = "backgroundcolor={gray 0.8} font=" + textfield_font;
p.create_field(left_x, left_y, right_x, right_y, "date", "textfield", optlist);
```

クックブック **完全なコードサンプルがクックブックの** interactive/form_textfield_fill_with_js **トピックにあります。**

テキストフィールドのさまざまなフォーマットオプション Acrobat では、テキストフィールドに対してさまざまなオプションを指定して内容をフォーマットすることが可能です。たとえば通貨・日付・パーセントなどです。これは、カスタムの JavaScript コードを Acrobat が使用することによって実装されています。こういったフォーマット機能は PDF リファレンスには記載されていない物ですから、PDFlib は直接にはこれに対応をしていません。しかしながら、PDFlib ユーザーの便宜を図るため、以下、フォーマットオプションを実現するための簡単な JavaSctipt コードを ***PDF_create_field()*** の ***action*** オプションで与える方法について説明します。

テキストフィールドがフォーマットされるようにするには、JavaScript スニペットをテキストフィールドの ***keystroke・format*** のアクションとして設定します。その JavaScript コードから何らかの内部 Acrobat 関数を呼び出し、この関数の引数群でフォーマットの詳細を制御します。

以下の作成例では、***keystroke・format*** の 2 つのアクションを作成し、これらを 1 つのフォームフィールドに対して設定して、フィールド内容のフォーマットが小数点以下の桁数 2・通貨記号 EUR となるようにしています。

```
keystroke_action = p.create_action("JavaScript",
        "script={AFNumber_Keystroke(2, 0, 3, 0, \"EUR \", true); }");

format_action = p.create_action("JavaScript",
        "script={AFNumber_Format(2, 0, 0, 0, \"EUR \", true); }");

String  optlist = "font=" + font + "action={keystroke " + keystroke_action +
                " format=" + format_action + "}";
p.create_field(50, 500, 250, 600, "price", "textfield", optlist);
```

クックブック **完全なコードサンプルがクックブックの** interactive/form_textfield_input_format **トピックにあります。**

Acrobat で対応しているさまざまなフォーマットを指定するには、それぞれ適切な関数を JavaScript コード内で用いる必要があります。対応しているすべてのフォーマットについて、それぞれ実現するために ***keystroke・format*** アクションで用いるべき JavaScript 関数名を表 3.5 に挙げます。表 3.6 は 関数の引数の解説です。これらの関数を、上記の例と同様に使用してください。

表 3.5 テキストフィールドのための JavaScript フォーマット関数一覧

フォーマット	keystroke・format アクションで用いるべき JavaScript 関数
数値	AFNumber_Keystroke(nDec, sepStyle, negStyle, currStyle, strCurrency, bCurrencyPrepend) AFNumber_Format(nDec, sepStyle, negStyle, currStyle, strCurrency, bCurrencyPrepend)
パーセント	AFPercent_Keystroke(ndec, sepStyle), AFPercent_Format(ndec, sepStyle)
日付	AFDate_KeystrokeEx(cFormat), AFDate_FormatEx(cFormat)
時間	AFTime_Keystroke(tFormat), AFTime_FormatEx(cFormat)
特殊	AFSpecial_Keystroke(psf), AFSpecial_Format(psf)

表 3.6 JavaScript フォーマット関数に対するパラメター一覧

パラメタ	説明・とりうる値
nDec	小数点以下の桁数
sepStyle	区切りのスタイル。 *0* 1,234.56 *1* 1234.56 *2* 1.234,56 *3* 1234,56
negStyle	負数の表記方法。 *0* 標準 *1* 赤い字にする *2* かっこでくくる *3* 両方
strCurrency	通貨記号文字列。例：\u20AC でユーロ記号
bCurrency-Prepend	*false* 通貨記号を頭につけない *true* 通貨記号を頭につける
cFormat	日付形式文字列。以下の形式指定文字列を含むことができるほか、後述の tFormat の時刻形式をどれでも含むことができる。 *d* 日 *dd* 日の頭にゼロを適宜つけたもの *ddd* 曜日の短縮形 *m* 月を数で表す *mm* 月を数で表し頭にゼロを適宜つけたもの *mmm* 月名の短縮形 *mmmm* 月名の全体形 *yyyy* 年の 4 桁 *yy* 年の下 2 桁

表 3.6 JavaScript フォーマット関数に対するパラメター覧

パラメタ	説明・とりうる値
tFormat	時刻形式文字列。以下の形式指定文字列を含むことができる。
	h 時（0 ～ 12）
	hh 時（0 ～ 12）の頭にゼロを適宜つけたもの
	H 時（0 ～ 24）
	HH 時（0 ～ 24）の頭にゼロを適宜つけたもの
	M 分
	MM 分の頭にゼロを適宜つけたもの
	s 秒
	ss 秒の頭にゼロを適宜つけたもの
	t 午前「a」、午後「p」
	tt 午前「am」、午後「pm」
psf	特殊オプション。
	0 ZIP コード
	1 ZIP コード + 4
	2 電話番号
	3 社会保障番号

3.7 地理空間 PDF

クックブック 完全なコードサンプルがクックブックの interactive/starter_geospatial トピックにあります。

3.7.1 GeoPDF を Acrobat で使う

PDF 1.7ext3 では、地理空間参照情報（世界座標）を PDF ページ内容に追加することができるようになっています。Acrobat 9 以上では、地理空間参照付き PDF 文書（縮めて GeoPDF）でいくつかのことができます：

- マウスカーソルの下の地図上の位置の座標を表示：**「ツール」→「分析」→「地図位置ツール」**（Acrobat X では、**「ツール」**ペーンの上端のボタンを使って**「分析」**ツールバーを表示させる必要があるかもしれません）。マウスカーソルの下の地図上の位置の座標を、右クリックして**「座標をクリップボードにコピー」**を選択することでコピーできます。
- 地図上の位置を検索：**「ツール」→「分析」→「地図位置ツール」**→右クリックして**「位置を検索」**を選択→求める座標を入力。
- 地図上の位置をマーク：**「ツール」→「分析」→「地図位置ツール」**→右クリックして**「位置をマーク」**を選択。
- 地図上の距離・周辺・面積を測定：**「ツール」→「分析」→「ものさしツール」**。
- 座標表示に用いたい座標系等、地理空間測定のさまざまな設定を、**「編集」→「環境設定」→「一般 ...」→「ものさし（地図情報）」**で変更することができます。

PDFlib の地理空間機能は下記の関数とオプションで実装されています：

- 1 つのページに対して、1 つないし複数の地理参照付き領域を、*PDF_begin/end_page_ext()* の *viewports* オプションで割り当てることができます。ビューポートを使うと、ページ上の別々の領域で別々の地理空間参照（*georeference* オプションで指定）を用いることが可能になります。これはたとえば同一ページ上に複数の地図があるときなどに有用です。
- *PDF_load_image()* の *georeference* オプションを使うと、画像に地球ベースの座標を割り当てることができます。
- この *georeference* オプションを *PDF_open_pdi_page()*・*PDF_begin_template_ext()* でも提供して、取り込み PDF ページ・テンプレートにも地球ベースの座標を割り当てられるようにできるとよいのですが、残念ながらこれは Acrobat 9・Acrobat X でうまく動作しないので、対応していません。

3.7.2 地理座標系と投影座標系

地理座標系は地球を地理座標で、すなわち緯度と経度を度単位で表して記述します。投影座標系は、地理座標系の上に指定することができ、地理座標系における点から二次元（投影）座標系への変換を記述します。ここから算出される座標を Northing・Easting 値といい、投影座標系では角度はもはや不要です。地理座標系が GPS などの全球的応用分野で用いられているのに対して、投影はそれよりもある程度局地的な地図製作などの応用分野で必要となります。

歴史的・数学的理由により、世界じゅうでさまざまな座標系が用いられています。地理座標系・投影座標系とも、EPSG・WKT という 2 種類の普及した方式で記述することができます。

EPSG EPSG は何千もの座標系の集合であり、おのおのが数値コードで参照されます。EPSG は、今はなき欧州石油調査グループの略称であり、現在は国際石油・天然ガス生産者協会（OGP）によって保守されています。

EPSG 参照コードは、EPSG データベース内の座標系群のうちの 1 つを指し示します。EPSG データベース全体を、下記の場所からダウンロードすることができます：

```
www.epsg.org
```

WKT（Well-known text） WKT（Well-Known Text）系は記述的であり、座標系のあらゆる関連パラメタのテキスト表記から成っています。WKT の仕様は文書「*OpenGIS® Implementation Specification: Coordinate Transformation Services*」に示されており、この文書は Open Geospatial Consortium（OGC）によって Document 01-009 として発行されています。これは下記の場所で入手可能です：

```
www.opengeospatial.org/standards/ct
```

WKT は ISO 19125-1 で標準化もされています。WKT・EPSG とも Acrobat で使用できます（また、PDFlib でも対応しています）が、Acrobat はすべての可能な EPSG コードを実装しているわけではありません。とくに、地理座標系のための EPSG コード群には Acrobat は対応していないようです。その場合には WKT の使用を推奨します。下記の Web サイトで、特定の EPSG コードに対応する WKT を示しています：

```
www.spatialreference.org/ref/epsg
```

3.7.3 座標系関連の作成例

地理座標系関連の作成例 WGS84（世界測地系）地理座標系は、GPS をはじめとする多くの応用分野（OpenStreetMap 等）の基礎となっています。これは下記のように *georeference* オプションの *worldsystem* サブオプションで表現できます：

```
worldsystem={type=geographic wkt={
GEOGCS["WGS 84",
    DATUM["WGS_1984", SPHEROID["WGS 84", 6378137, 298.257223563]],
    PRIMEM["Greenwich", 0],
    UNIT["degree", 0.01745329251994328]]
}}
```

ETRS（欧州地球基準系）地理座標系は WGS84 とほとんど等価です。これは下記のように指定できます：

```
worldsystem={type=geographic wkt={
GEOGCS["ETRS_1989",
    DATUM["ETRS_1989", SPHEROID["GRS_1980", 6378137.0, 298.257222101]],
    PRIMEM["Greenwich", 0.0],
    UNIT["Degree", 0.0174532925199433]]
}}
```

注 WGS84・ETRS 系に対する EPSG コードはここでは示していません。なぜなら Acrobat は地理座標系に対する EPSG コードに対応しておらず、投影座標系にしか対応していないようだからです（後述）。

投影座標系関連の作成例 投影は、その背景にある地理座標系に基づいています。下記の例では、GPS 座標での利用に適した投影座標系を指定します。

中欧では、ETRS89 UTM zone 32 N という系が適用されます。これは広く利用されている UTM（国際メルカトル投影）を用いており、下記のように *georeference* オプションの *worldsystem* サブオプションで表現できます：

```
worldsystem={type=projected wkt={
    PROJCS["ETRS_1989_UTM_Zone_32N",
      GEOGCS["GCS_ETRS_1989",
        DATUM["D_ETRS_1989", SPHEROID["GRS_1980", 6378137.0, 298.257222101],
          TOWGS84[0, 0, 0, 0, 0, 0, 0]],
        PRIMEM["Greenwich", 0.0],
        UNIT["Degree", 0.0174532925199433]],
      PROJECTION["Transverse_Mercator"],
      PARAMETER["False_Easting", 500000.0],
      PARAMETER["False_Northing", 0.0],
      PARAMETER["Central_Meridian", 9.0],
      PARAMETER["Scale_Factor", 0.9996],
      PARAMETER["Latitude_Of_Origin", 0.0],
      UNIT["Meter", 1.0]]
}}
```

この座標系に対応する EPSG コードは 25832 です。WKT のかわりに、上記の系はその EPSG コードを通じて下記のように指定することもできます：

```
worldsystem={type=projected epsg=25832}
```

3.7.4 Acrobat における GeoPDF の制約

GeoPDF を Acrobat 9・Acrobat X で扱うなかで、私たちは下記の難点に遭遇しています：

- EPSG コードは地理座標系に対してはまったく動作せず、投影系に対してのみ動作するようです。
  回避策：EPSG コードでなく、対応する WKT を使うこと。
- 地理空間データをベクトルオブジェクトに紐付けても動作しません。PDF Reference によればこれは理論上動作するはずなのですが、この理由から PDFlib では、*PDF_open_pdi_page()*・*PDF_begin_template_ext()* に対する *georeference* オプションには対応していません。
  ベクトルベースの地図を作成するための回避策：地理空間データをページに紐付けることはできます。すなわち、*PDF_begin_page_ext()* の *viewports* オプションを用います。
- 重なり合った地図：同一ページ上に複数の画像ベースの地図を貼り付けることができます。複数の地図が重なり合っているとき、重なり合っている領域内の点の座標を表示させると、Acrobat は、最後に貼り付けられた地図の座標を用います（これがすなわち見えている地図でもありますのでこれは理にかなっています）。しかし、双方の画像ハンドルが同一の（すなわち *PDF_load_image()* への 1 回の呼び出しで取得された）場合には、Acrobat はもはやさまざまな画像の領域を考慮しなくなり、1 番目の画像の座標が誤って 2 番目の画像の領域へ拡張され、誤った座標表示となってしまいます。
  回避策：同一ページ上に同じ画像をベースにした地図を複数枚貼り付けたいときは、その画像を複数回開くこと。
- 面積ものさしツールは地理座標系に対しては正しく動作せず、投影系に対してのみ正しく動作します。

4 Unicode とレガシエンコーディング

この章では、Unicode やその他のエンコーディング方式に関する基礎的な情報を提供します。PDFlib におけるテキスト処理は Unicode 規格に大きく依存していますが、さまざまなレガシエンコーディングや特殊エンコーディングにも対応しています。

4.1 Unicode の重要な概念

キャラクタとグリフ　テキストを扱う際には、次の概念をはっきり区別することが大切です。

- **キャラクタ**は、言語の中で情報を伝達する最小の単位です。代表的な例はラテンアルファベットの文字、中国語の表意文字、日本語の音節文字です。キャラクタは意味を持ちます。すなわちキャラクタは意味実体です。
- グリフは、さまざまな視覚表現で、1 個ないし複数のキャラクタを表します。グリフは外見を持ちます。すなわちグリフは表現実体です。

キャラクタとグリフの間に一対一の対応は存在しません。たとえば合字は 1 つのグリフですが、2 つ以上のキャラクタで表現されます。かと思えば、1 つのグリフが場面によって別々のキャラクタを表すのに使われることもあります（キャラクタには同じ形のものがあります。図 4.1 参照）。

BMP と PUA　下記の用語が Unicode ベースの環境では頻繁に登場します：

- **基本多言語面**（*BMP*）：Unicode の範囲 U+0000 ～ U+FFFF 内のコード点から成ります。Unicode 規格はこのほかにも多くのコード点を、追加面群、すなわち範囲 U+10000 ～ U+10FFFF 内に含んでいます。
- **私用領域**（*PUA*）：私用のために予約されているいくつかの領域の 1 つです。PUA のコード点は一般的なやりとりには利用できません。なぜなら、Unicode 規格ではこの領域の中にいかなるキャラクタをも指定していないからです。基本多言語面は範囲 U+E000 ～ U+F8FF に PUA を含んでいます。第 15 面（U+F0000 ～ U+FFFFD）と第 16 面（U+100000 ～ U+10FFFD）は私用のためにすっかり予約されています。

図 4.1
グリフとキャラクタの関係

Unicode エンコーディング形式（UTF 形式） Unicode 標準は各キャラクタに数（コード点）を割り当てています。この数をコンピュータ処理で使うには、何らかの方式で表現しなければなりません。Unicode 標準ではこれをエンコーディング形式と呼びます（旧称：変換形式）。この用語はフォントのエンコーディングと混同してはいけません。Unicode は以下のエンコーディング形式を定義しています。

- *UTF-8*：これは可変幅の形式で、コード点は 1 ～ 4 バイトで表されます。範囲 U+0000 ～ U+007F の ASCII キャラクタは範囲 00 ～ 7F のシングルバイトで表されます。範囲 U+00A0 ～ U+00FF の Latin-1 キャラクタは 2 バイトで表され、その第一バイトはつねに 0xC2 か 0xC3 になります（これらの値は Latin-1 で **Â** と **Ã** を表します）。
- *UTF-16*：基本多言語面（BMP）のコード点は 1 つの 16 ビット値で表されます。補助多言語面のコード点、すなわち範囲 U+10000 ～ U+10FFFF のコード点は 16 ビット値の対で表されます。この対をサロゲートペアと呼びます。1 つのサロゲート値は、範囲 D800 ～ DBFF の高位サロゲート値 1 つと範囲 DC00 ～ DFFF の低位サロゲート値 1 つから成っています。高位と低位のサロゲート値はサロゲートペアの中にのみ現れ、他の場面で使われることはありません。
- *UTF-32*：各コード点は 1 つの 32 ビット値で表されます。

クックブック 完全なコードサンプルがクックブックの text_output/process_utf8 トピックにあります。

Unicode エンコーディング体系とバイト順序マーク（BOM） コンピュータアーキテクチャは種類によって、バイトの並べ方が違います。すなわちバイトがより大きな値（16 ビットや 32 ビット）を構成するとき、最上位バイトを最初に格納する方式（ビッグエンディアン）と、最下位バイトを最初に格納する方式（リトルエンディアン）があります。ビッグエンディアンアーキテクチャの代表例は PowerPC であり、一方 x86 アーキテクチャはリトルエンディアンです。UTF-8 と UTF-16 はシングルバイトより大きな値を用いていますので、やはりバイトの並べ方を論じる必要が出てきます。エンコーディング体系は（上述のエンコーディング形式とは違うので注意）、エンコーディング形式に加えてバイト順序を指定します。たとえば UTF-16BE は、UTF-16 でバイト順序はビッグエンディアンという意味です。バイト順序があらかじめわからないときは、それを指定する手段としてコード点 U+FEFF があります。これをバイト順序マーク（BOM）といいます。BOM は UTF-8 では不要ですが、存在していてもよく、それを利用してバイトストリームを UTF-8 と同定することもできます。さまざまなエンコーディング形式に対する BOM の表し方を表 4.1 に示します。

表 4.1 さまざまな Unicode エンコーディング形式に対するバイト順序マーク一覧

エンコーディング形式	バイト順序マーク（16 進）	WinAnsi における視覚表現[1]
UTF-8	EF BB BF	ï»¿
UTF-16 ビッグエンディアン	FE FF	þÿ
UTF-16 リトルエンディアン	FF FE	ÿþ
UTF-32 ビッグエンディアン	00 00 FE FF	▮▮þÿ
UTF-32 リトルエンディアン	FF FE 00 00	ÿþ▮▮

1. 黒四角 ▮は null バイトを意味します。

4.2 シングルバイト（8 ビット）エンコーディング

8 ビットエンコーディング（シングルバイトエンコーディングともいいます）は、バイト値 0x01 ～ 0xFF をそれぞれ、BMP（すなわち U+0000 ～ U+FFFF）内の Unicode 値を持つ 1 つのキャラクタに割り当てます。同時に使えるキャラクタは 255 種類までです。なぜならコード 0（ゼロ）は *.notdef* キャラクタ U+0000 のために予約されているからです。PDFlib の定義済みエンコーディングを表 4.2 に記します。普通のフォントでは満たせない要求をする用字系や言語もある、という認識が重要です。

注 PDFlib ディストリビューションに同梱の作成例「chartab」を使うと、任意のフォントとエンコーディングの組み合わせについて、そのキャラクター覧表を簡単に印刷できます。

表 4.2 PDFlib の定義済みエンコーディング一覧

コードページ	対応言語
winansi	cp1252（ISO8859-1 のスーパーセット）と同じ
macroman	Mac Roman エンコーディング、オリジナル Macintosh キャラクタセット
macroman_apple	macroman とほぼ同じ、ただし通貨をユーロで置き換え、数学 / ギリシャ語記号を追加で含みます
ebcdic	EBCDIC コードページ 1047
ebcdic_37	EBCDIC コードページ 037
pdfdoc	PDFDocEncoding
iso8859-1	（Latin-1）西欧諸語
iso8859-2	（Latin-2）中欧のスラブ諸語
iso8859-3	（Latin-3）エスペラント・マルタ語
iso8859-4	（Latin-4）エストニア語・バルト諸語・グリーンランド語
iso8859-5	ブルガリア語・ロシア語・セルビア語
iso8859-6	アラビア語
iso8859-7	現代ギリシャ語
iso8859-8	ヘブライ語・イディッシュ語
iso8859-9	（Latin-5）西欧諸語・トルコ語
iso8859-10	（Latin-6）北欧諸語
iso8859-13	（Latin-7）バルト諸語
iso8859-14	（Latin-8）ケルト語
iso8859-15	（Latin-9）Latin-1 に、ユーロとフランス語・フィンランド語キャラクタを加えています
iso8859-16	（Latin-10）ハンガリー語・ポーランド語・ルーマニア語・スロベニア語
cp1250	中欧諸語
cp1251	キリル諸語
cp1252	西欧諸語（winansi と同じ）
cp1253	ギリシャ語
cp1254	トルコ語
cp1255	ヘブライ語
cp1256	アラビア語
cp1257	バルト諸語
cp1258	ベトナム語

ホストエンコーディング　特殊なエンコーディング *host* は固定した意味を持たず、環境のプラットフォームによって、下記の 8 ビットエンコーディングに対応づけられます（表 4.2 参照）：

- MVS か USS を持つ IBM zSeries では ***ebcdic*** に対応づけられます。
- IBM i5/iSeries では ***ebcdic_37*** に対応づけられます。
- Windows では ***winansi*** に対応づけられます。
- それ以外のすべてのシステム（Mac OS X を含む）では ***iso8859-1*** に対応づけられます。

ホストエンコーディングが有用なのは何といっても、プラットフォーム非依存のテストプログラム（PDFlib ディストリビューションの中にあるような）や、その他の単純なプログラムを書くときです。製品版でのホストエンコーディングの使用は推奨しませんので、何らかの適切なエンコーディングに置き換えるべきです。

host エンコーディングは、Unicode 非対応の言語バインディングでは、名前文字列のデフォルトエンコーディングとして用いられています（107 ページ「4.4.3 Unicode 非対応言語バインディングの文字列」参照）。なぜならこれがファイル名などに一番適切だからです。

自動エンコーディング　PDFlib は、特定の環境に対してもっとも自然なエンコーディングを手間なく指定できるしくみに対応しています。エンコーディング名としてキーワード ***auto*** を与えると、プラットフォームや環境によって、以下のテキストフォント用 8 ビットエンコーディングを指定したことになります。

- Windows では：カレントのシステムコードページ（詳しくは後述）
- Unix・Mac OS X では：***iso8859-1***（ただし Mac 上の LWFN PostScript フォントでは ***auto*** は ***macroman*** に対応）
- IBM i5/iSeries では：カレントジョブのエンコーディング（***IBMCCSIDoooooooooooo***）
- IBM zSeries では：***ebcdic***（＝コードページ 1047）

記号フォントでは、キーワード ***auto*** は ***builtin*** エンコーディングに対応づけられます（127 ページ「5.4.2 記号フォントに対するエンコーディングを選ぶ」参照）。自動エンコーディングは多くの場面で便利ですが、その半面、この方式を用いた PDFlib クライアントプログラムは他の環境では使えなくなります。

システムコードページの利用　PDFlib は、コードページ定義をシステムから取得して、それを内部利用のために適切に変換させるようにすることができます。この機能を利用すれば、コードページの実装作業をしなくてすむのでとても便利です。組み込みエンコーディングやユーザー定義エンコーディングの名前を ***PDF_load_font()*** に与えるのではなく、ただ、システムが知っているエンコーディング名を利用すればよいのです。この機能が利用できるのはいくつかの限られたプラットフォーム上だけであり、そのエンコーディング文字列の文法は次のようにプラットフォーム依存です。

- Windows では、エンコーディング名は ***cp<*** 番号 **>** です。ここで **<** 番号 **>** は、システムにインストールされている任意のシングルバイトコードページの番号です（マルチバイトの Windows コードページについては 171 ページ「6.5.2 カスタム日中韓フォント」参照）。

  ```
  font = p.load_font("Helvetica", "cp1250", "");
  ```

  シングルバイトコードページは内部の 8 ビットエンコーディングに変換されるのに対し、マルチバイトコードページは実行時に ***unicode*** に対応づけられます。テキストは、選んだコードページと互換の形式で与える必要があります（たとえば ***cp932*** に対しては SJIS）。

- IBM i5/iSeries では、任意の ***Coded Character Set Identifier***（CCSID）が使えます。CCSID は文字列として与える必要があり、PDFlib は、与えられたコードページ番号に ***IBMCCSID*** という接頭辞をつけます。また PDFlib は、コードページ番号が 5 文字に満たないときには頭に 0 を補います。コードページ番号として 0（ゼロ）を与えると、カレントジョブのエンコーディングが用いられることになります。

```
font = p.load_font("Helvetica", "273", "");
```

- USS か MVS を持つ IBM zSeries では、任意の ***Coded Character Set Identifier***（CCSID）が使えます。CCSID は文字列として与える必要があり、PDFlib は、与えられたコードページ名に一切変更を加えずそのままシステムに渡します。

```
font = p.load_font("Helvetica", "IBM-273", "");
```

ユーザー定義 8 ビットエンコーディング　定義済みエンコーディングのほか、PDFlib はユーザー定義 8 ビットエンコーディングにも対応しています。これを使いたいのは、PDFlib の内部で得られない何かのキャラクタセットを扱いたいときです。たとえば、PDFlib 内部で対応しているのとは違う EBCDIC キャラクタセットを扱うこともできます。PDFlib は、PostScript グリフ名で定義されたエンコーディングテーブルに対応しているほか、Unicode 値で定義されたテーブルにも対応しています。

ユーザー定義エンコーディングを PDFlib プログラム内で利用できるようにするには、以下の作業をあらかじめ行う必要があります（あるいはエンコーディングは、***PDF_encoding_set_char()***）を使って実行時に構築することもできます）。

- エンコーディングの記述を単純なテキスト形式で作成する。
- そのエンコーディングを PDFlib リソースファイル内で（59 ページ「3.1.3 リソース設定とファイル検索」参照）、または ***PDF_set_parameter()*** で設定する。
- エンコーディングが使うすべてのキャラクタに対応したフォントを与える（メトリックと、あるいはアウトラインファイルも）。

エンコーディングファイルは、グリフ名と番号を 1 行ずつ単純に列挙したものです。エンコーディング定義の冒頭部分は次のようになります。

```
% PDFlib用エンコーディング定義。グリフ名を使用
% 名前          コード  Unicode（オプション）
space           32      0x0020
exclam          33      0x0021
...
```

Unicode 値が指定されていないときは、PDFlib はその内部テーブルの中で適切な Unicode 値をさがします。グリフなでなく Unicode 値を指定することもできます：

```
% PDFlib用コードページ定義。Unicode値を使用
% Unicode       コード
0x0020          32
0x0021          33
...
```

より厳密に言えば、エンコーディングかコードページのファイルの内容は、以下の規則に従う必要があります。

- 注釈はパーセントキャラクタ「%」で始まり、行末で終わります。
- 各行の先頭項目は、PostScript グリフ名か、または 16 進 Unicode 値です。Unicode 値は、接頭辞 ***ox*** と 16 進 4 桁（大文字でも小文字でも）で構成されます。その後に、空白キャ

ラクタと、16 進（0x00 ～ 0xFF）か 10 進（0 ～ 255）の文字コードが続きます。オプションとして、グリフ名によるエンコーディングファイルの場合は、対応する Unicode 値を 3 列目に持つこともできます。

- エンコーディングファイル内で述べられていない文字コードには、未定義と見なされます。あるいは未定義位置に対しては、Unicode 値 ***oxoooo*** かキャラクタ名 ***.notdef*** を与えることも可能です。
- エンコーディングかコードページのファイルの中のUnicode値はすべてU+FFFFよりも小さくなければなりません。

命名規則としては、グリフ名によるテーブルをエンコーディングファイル（****.enc***）と呼び、Unicode によるテーブルをコードページファイル（****.cpg***）と呼びますが、しかし PDFlib は両者を同様に扱います。

4.3 日本語・中国語・韓国語エンコーディング

歴史的に、非常に多くの日中韓エンコーディング方式が、無数の規格化団体や企業によって開発されてきました。幸い、普及しているエンコーディングにはすべて、Acrobat や PDF はデフォルト対応しています。日中韓テキストにおけるエンコーディングの概念は、欧文テキストの場合よりはるかに複雑なので、単なる 8 ビットエンコーディングではもはや不充分です。そのかわりとして PostScript と PDF では、キャラクタコレクションとキャラクタマップ（*CMap*）という概念を用いてフォント内のキャラクタを組織化しています。

標準日中韓エンコーディングに対する定義済み CMap　定義済みの日中韓 CMap を表 4.3 に示します。表から見て取れるように、Mac・Windows・Unix システムで使われる日中韓エンコーディングの多くに対応しているほか、ベンダー独自エンコーディングにもいくつか対応しています。たとえば日本語なら Shift-JIS・EUC・ISO 2022、中国語なら GB・Big5、韓国語なら KSC などです。Unicode にもすべてのロケールで対応しています。

表 4.3 日本語・中国語・韓国語テキスト用定義済み CMap 一覧（PDF リファレンスより）

ロケール	CMap 名	文字セットとテキスト形式
日本語	UniJIS-UCS2-H, -V	Adobe-Japan1 キャラクタコレクションの Unicode（UCS-2）エンコーディング
	UniJIS-UCS2-HW-H UniJIS-UCS2-HW-V	UniJIS-UCS2-H と同じだが、プロポーショナルの欧文文字が半角の形に置き換わっている
	UniJIS-UTF16-H UniJIS-UTF16-V	Adobe-Japan1 キャラクタコレクションの Unicode（UTF-16BE）エンコーディング。JIS X 0213:1000 キャラクタセットの全キャラクタへの割り当てを含む。
	83pv-RKSJ-H	Mac、JIS X 0208 キャラクタセット＋漢字 Talk6 拡張、Shift-JIS、Script Manager コード 1
	90ms-RKSJ-H 90ms-RKSJ-V	Microsoft コードページ 932（charset 128）、JIS X 0208 キャラクタセット＋NEC・IBM 拡張
	90msp-RKSJ-H 90msp-RKSJ-V	90ms-RKSJ-H と同じだが、半角英字をプロポーショナル形に替えたもの
	90pv-RKSJ-H	Mac、JIS X 0208 ＋漢字 Talk7 拡張、Shift-JIS、Script Manager コード 1
	Add-RKSJ-H, -V	JIS X 0208 キャラクタセット＋富士通 FMR 拡張、Shift-JIS エンコーディング
	EUC-H, -V	JIS X 0208 キャラクタセット、EUC-JP エンコーディング
	Ext-RKSJ-H, -V	JIS C 6226（JIS78）キャラクタセット＋ NEC 拡張、Shift-JIS エンコーディング
	H, V	JIS X 0208 キャラクタセット、ISO-2022-JP エンコーディング
中国語 簡体字	UniGB-UCS2-H UniGB-UCS2-V	Adobe-GB1 キャラクタコレクションの Unicode（UCS-2）エンコーディング
	UniGB-UTF16-H UniGB-UTF16-V	Adobe-GB1 キャラクタコレクションの Unicode（UTF-16BE）エンコーディング。GB18030-2000 キャラクタセットの全キャラクタへの割り当てを含む。
	GB-EUC-H GB-EUC-V	Microsoft コードページ 936（charset 134）、GB 2312-80 キャラクタセット、EUC-CN エンコーディング
	GBpc-EUC-H GBpc-EUC-V	Macintosh、GB 2312-80 キャラクタセット、EUC-CN エンコーディング、Script Manager コード 2
	GBK-EUC-H, -V	Microsoft コードページ 936（charset 134）、GBK キャラクタセット、GBK エンコーディング
	GBKp-EUC-H GBKp-EUC-V	GBK-EUC-H と同じだが、半角英字をプロポーショナル形に替え、コード 0x24 に元（ｴ）のかわりにドル（$）を割り当てたもの。
	GBK2K-H, -V	GB 18030-2000 キャラクタセット、1・2・4 バイト混在エンコーディング
中国語 繁体字	UniCNS-UCS2-H UniCNS-UCS2-V	Adobe-CNS1 キャラクタコレクションの Unicode （UCS-2）エンコーディング

表 4.3 日本語・中国語・韓国語テキスト用定義済み CMap 一覧（PDF リファレンスより）

ロケール	CMap 名	文字セットとテキスト形式
	UniCNS-UTF16-H UniCNS-UTF16-V	Adobe-CNS1 キャラクタコレクションの Unicode （UTF-16BE）エンコーディング。HKSCS-2001（2・4 バイト文字コード）の全キャラクタへの割り当てを含む。
	B5pc-H, -V	Macintosh、Big Five キャラクタセット、Big Five エンコーディング、Script Manager コード 2
	HKscs-B5-H[1] HKscs-B5-V[1]	Hong Kong SCS（Supplementary Character Set）、Big Five のキャラクタセットとエンコーディングに対する拡張
	ETen-B5-H, -V	Microsoft コードページ 950（charset 136）、Big Five キャラクタセット＋ ETen 拡張
	ETenms-B5-H ETenms-B5-V	ETen-B5-H と同じだが、半角英字をプロポーショナル形に替えたもの
	CNS-EUC-H, -V	CNS 11643-1992 キャラクタセット、EUC-TW エンコーディング
韓国語	UniKS-UCS2-H, -V	Adobe-Korea1 キャラクタコレクションの Unicode（UCS-2）エンコーディング
	UniKS-UTF16-H, -V	Adobe-Korea1 キャラクタコレクションの Unicode（UTF-16BE）エンコーディング
	KSC-EUC-H, -V	KS X 1001:1992 キャラクタセット、EUC-KR エンコーディング
	KSCms-UHC-H KSCms-UHC-V	Microsoft コードページ 949（charset 129）、KS X 1001:1992 キャラクタセットにハングル 8822 種を加えたもの、Unified Hangul Code（UHC）エンコーディング
	KSCms-UHC-HW-H KSCms-UHC-HW-V	KSCms-UHC-H と同じだが、プロポーショナル形英字を半角に替えたもの
	KSCpc-EUC-H	Mac、KS X 1001:1992 キャラクタセット＋ Mac OS KH 拡張・Script Manager コード 3

注 Unicode 対応の言語バインディングは、Unicode CMap（UCS2 か UTF16）にのみ対応しています。それ以外の CMap を使ってはいけません（107 ページ「4.4.2 Unicode 対応言語バインディングの文字列」参照）。

標準 CMap に対する日中韓テキストのエンコーディング　要求された CMap に対して、それに合うようエンコードされたテキストを与えるのはクライアント側の役割です。与えられたテキストが、要求された CMap に従っているかどうかを、PDFlib はチェックして、選ばれた CMap に従っていない誤ったテキスト入力に対しては例外を発生させます。

CMap の設定　定義済み CMap の 1 つを用いて日本語・中国語・韓国語（日中韓）テキストを生成するには、PDFlib は対応する CMap ファイルを必要とします。入ってくるテキストを処理して、日中韓エンコーディングを Unicode に対応づけるためです。CMap ファイルは別パッケージで入手できます。以下のようにインストールする必要があります。

- Windows では CMap ファイルは、PDFlib のインストレーションディレクトリ内の *resource/cmap* ディレクトリに置いておけば、自動的に見つけられます。
- それ以外のシステムでは CMap ファイルは、任意の好都合なディレクトリに置くことができ、実行時に *SearchPath* を下記のように設定することで、CMap ファイルを手動で設定する必要があります。

```
p.set_parameter("SearchPath", "/パス/パス/resource/cmap");
```

日中韓 CMap ファイルの検索先を設定する方式のかわりに、適切な *SearchPath* 定義を含む UPR 設定ファイルを指すよう *PDFLIBRESOURCEFILE* 環境変数を設定することもできます。

注 MVS では CMap ファイルは、短いファイル名の CMap を持つ別パッケージからインストールする必要があります。

カスタム日中韓フォントに対するコードページ Windows では PDFlib は、システムにインストールされているすべてのコードページに対応しています。それ以外のプラットフォームでは、表 4.4 に示すコードページが使えます。これらのコードページは内部的に、対応する CMap に対応づけられます（**例**：*cp932* を *90ms-RKSJ-H/V* に対応づけ）。この対応づけのために、適切な CMap を設定する必要があるのです（上述）。*textformat* パラメタは *auto* に設定する必要があり、そしてテキストは、選んだコードページと互換な形式で与える必要があります。

表 4.4 日中韓コードページ一覧（textformat=auto か textformat=bytes で用いる必要があります）

ロケール	コードページ	形式	キャラクタセット
日本語	cp932	Shift-JIS	JIS X 0208:1997 ＋ Microsoft 拡張
中国語簡体字	cp936	GBK	GBK
中国語繁体字	cp950	Big Five	Big Five ＋ Microsoft 拡張
韓国語	cp949	UHC	KS X 1001:1992 ＋残り 8822 種のハングル拡張
	cp1361	Johab	Johab

4.4 PDFlib の文字列処理

4.4.1 内容文字列・ハイパーテキスト文字列・名前文字列

PDF とオペレーティングシステムの要請により、PDFlib での文字列の処理方式は、その文字列の使い道によって異なります。PDFlib API は、以下の文字列型を定義して用いています。PDFlib API リファレンスでは、すべての関連するパラメタ・オプションには、内容文字列・ハイパーテキスト文字列・名前文字列と記してあります。

内容文字列・ハイパーテキスト文字列・名前文字列は、Unicode と 8 ビットエンコーディングで使えます。非 Unicode の日中韓 CMap は、Unicode 非互換の言語バインディングでのみ使えます。文字列処理の実際は言語バインディングによって異なりますので、107 ページ「4.4.2 Unicode 対応言語バインディングの文字列」と 107 ページ「4.4.3 Unicode 非対応言語バインディングの文字列」で説明します。

内容文字列　内容文字列は、ユーザーが特定のフォントで選んだエンコーディングに従ってページ本体内容（ページ記述）を作成するために用いられます。PDFlib API リファレンスの中で、ページ内容関数における *text* という名前の関数引数はすべてこの種類に属します。内容文字列は特定フォント内のグリフによって表現されますので、使用可能なキャラクタの範囲はフォント / エンコーディングの組み合わせに依存します。

例：

PDF_show()・*PDF_fit_textline()*・*PDF_add_textflow()* の *text* 引数。

ハイパーテキスト文字列　ハイパーテキスト文字列は、しおりや注釈などのインタラクティブ機能のために用いられるものであり、関数の説明では明示的に**「ハイパーテキスト文字列」**と記してあります。インタラクティブ機能のための関数の多くの引数やオプションがこの種類に属すほか、それ以外にも若干この種類に属するものがあります。表示できるキャラクタの範囲は、Acrobat で利用可能なフォントやオペレーティングシステムといった外部要因に依存します。

例：

PDF_add_table_cell() の *fieldname* オプション
PDF_define_layer() の *name* オプション
PDF_create_action() の *destname* オプション
PDF_create_bookmark() の *text* 引数

名前文字列　名前文字列は、外部ファイル名・フォント名・ブロック名などのために用いられるものであり、関数の説明では**「名前文字列」**と記してあります。ハイパーテキスト文字列とわずかに違いますが、その違いが現れるのは Unicode 非対応の言語においてのみです。

例：

PDF_begin_document()・*PDF_create_pvf()* の *filename* 引数
PDF_load_font() の *fontname* 引数
PDF_load_iccprofile() の *profilename* 引数

ファイル名は特殊なケースです：オプション *filenamehandling* は、PDFlib が API に与えられたファイル名をローカルファイルシステムで使えるものへ変換する方法を指定します。

4.4.2 Unicode 対応言語バインディングの文字列

開発環境が文字列データ型に対応していて、かつ内部的に Unicode を用いており、そして対応する PDFlib 言語ラッパがその言語の Unicode 文字列に対応している場合に、そのバインディングを Unicode 対応と呼ぶことにします。下記の PDFlib 言語バインディングは Unicode 対応です：

- C++
- COM
- .NET
- Java
- Python
- REALbasic
- RPG
- Tcl

こうした環境での文字列処理は単純です：文字列はすべて自動的に、ネイティブな UTF-16 形式の Unicode 文字列として PDFlib カーネルに与えられます。こうした言語のラッパは、クライアントから与えられる Unicode 文字列を正しく取り扱うことができ、また、いくつかの PDFlib のパラメタを自動的に設定します。このことから下記の結果が生じます：

- 必要な変換を PDFlib 言語ラッパがすべて行い、クライアントから与えられたハイパーテキスト文字列がつねに ***utf16*** 形式の ***unicode*** エンコーディングで PDFlib に供給されるよう動作します。
- つねにUTF-16を言語環境がPDFlibに供給するため、UTF-8はUnicode対応言語では使用できません。あらかじめ UTF-16 に変換する必要があります。
- ページの内容には ***unicode*** エンコーディングを用いることが、エンコーディングを Unicode 対応言語で取り扱ううえでもっとも簡単な方法です。ただし 8 ビットエンコーディングや、記号フォントのシングルバイトテキストも、使いたければ使うことができます。
- 日中韓フォントに非 Unicode CMap を用いることは（103 ページ「4.3 日本語・中国語・韓国語エンコーディング」参照）避けなければなりません。なぜなら、ラッパがつねに Unicode を PDFlib コアに与えるからです。Unicode CMap のみが利用可能です。

要するに基本的には、クライアントは生の Unicode 文字列を PDFlib 関数に与えることができ、その際に別途設定やパラメタ設定は必要ないということです。関数の説明に書いてある、ハイパーテキスト文字列と名前文字列の違いは、Unicode 対応言語バインディングでは関係がありません。

Unicode 変換関数　文字列をUnicode以外のエンコーディングで扱わなければならない場合は、それを PDFlib に渡す前には Unicode に変換する必要があります。29 ページ「2 章 PDFlib の言語バインディング」の言語ごとの節に、代表的な言語バインディングで提供されている有用な Unicode 文字列変換方法を詳しく説明してあります。

4.4.3 Unicode 非対応言語バインディングの文字列

下記の PDFlib 言語バインディングは Unicode 非対応です：

- C（ネイティブな文字列データ型なし）
- PDFlib 7 との互換のためのレガシ C++ バインディング（設定に関する情報は 36 ページ「2.5 C++ バインディング」参照）
- Cobol（ネイティブな文字列データ型なし）

- Perl
- PHP
- Ruby

ネイティブな文字列データ型に対応していない言語バインディング（すなわち C・Cobol）では、UTF-16 文字列の長さを別途 ***length*** 引数で与える必要があります。これらの言語でも Unicode テキストは使用できますが、各種の文字列の処理は以下のように若干複雑になります。

内容文字列　内容文字列とは、ページ内容の作成に使われる文字列です。この種の文字列の解釈は、***PDF_load_font()*** の ***textformat*** 引数（後述）と ***encoding*** 引数によって制御されます。***textformat=auto*** ならば（これがデフォルト）、***unicode・glyphid*** エンコーディングと UCS-2・UTF-16 CMap に対しては ***utf16*** 形式が用いられます。それ以外のすべてのエンコーディングに対しては形式は ***bytes*** になります。ネイティブな文字列データ型を持たない言語（上記一覧参照）では、UTF-16 文字列の長さを ***length*** 引数で別途与える必要があります。

ハイパーテキスト文字列　文字列の解釈は ***hypertextformat・hypertextencoding*** パラメタ（後に詳述）によって制御されます。***hypertextformat=auto*** ならば（これがデフォルト）、***hypertextencoding=unicode*** の場合には ***utf16*** 形式が用いられ、それ以外の場合には ***bytes*** が用いられます。ネイティブな文字列データ型を持たない言語（上記リスト参照）では、UTF-16 文字列の長さを ***length*** 引数で別途与える必要があります。

名前文字列　名前文字列は、ページ記述文字列とはやや異なる方式で解釈されます。デフォルトでは名前文字列は ***host*** エンコーディングで解釈されます。しかし、先頭に UTF-8 BOM があるときは UTF-8 として（先頭に EBCDIC UTF-8 BOM があるときは EBCDIC UTF-8として）解釈されます。***usehypertextencoding*** パラメタが ***true*** なら、***hypertextencoding*** で指定されたエンコーディングが名前文字列にも適用されます。これはたとえば、フォントやファイルの名前を Shift-JIS で指定するのに使えます。***hypertextencoding=unicode*** の場合は、PDFlib は UTF-16 文字列を前提しますので、2 個の null バイトで終了する必要があります。

C では、UTF-8 文字列に対しては ***length*** 引数は 0 でなければなりません。0 以外なら文字列は UTF-16 として解釈されます。それ以外のすべての Unicode 非対応の言語バインディングでは、API 関数に ***length*** 引数はないので、名前文字列はかならず UTF-8 形式で与える必要があります。この場合、Unicode の名前文字列を作成するには、***PDF_utf16_to_utf8()*** ユーティリティ関数を用いて UTF-8 を作成することが可能です（後述）。

Unicode 変換関数　Unicode 非対応の言語バインディングでは、PDFlib は変換関数 ***PDF_utf16_to_utf8()・PDF_utf8_to_utf16()・PDF_utf32_to_utf16()*** を提供しているので、それを使って UTF-8 や UTF-16 の文字列を作成して、PDFlib に渡すことができます。

29 ページ「2 章 PDFlib の言語バインディング」の言語ごとの節に、代表的な言語バインディングで提供されている有用な Unicode 文字列変換方法を詳しく説明してあります。

内容・ハイパーテキスト文字列のテキスト形式　PDFlib の Unicode 文字列は、UTF-8・UTF-16・UTF-32 のいずれかの形式で、任意のバイト順序で与えることができます。形式の選択は、***textformat*** パラメタでページ記述の全テキストに対して、***hypertextformat*** パラメタでインタラクティブ要素群に対して制御することができます。この両パラメタで対

応している値を表 4.5 に示します。*[hyper]textformat* パラメタのデフォルトは *auto* です。名前文字列群に対して同じ動作を強制するには *usehypertextencoding* パラメタを用います。*hypertextencoding* パラメタのデフォルトは *auto* です。

表 4.5 textformat・hypertextformat パラメタに対する値一覧

[hyper]textformat	説明
bytes	文字列の 1 バイトが 1 キャラクタに対応します。これは主に 8 ビットエンコーディングと記号フォントで有用です。文字列の先頭に UTF-8 BOM があるときは、評価されたのち除去されます。
utf8	文字列は UTF-8 形式であると期待されます。不正な UTF-8 列があったときは、glyphcheck=error なら例外が発生し、そうでなければ削除されます。
ebcdicutf8	文字列は、EBCDIC コードされた UTF-8 形式であると期待されます（iSeries・zSeries のみ）。
utf16	文字列は UTF-16 形式であると期待されます。文字列の先頭に Unicode バイト順序マーク（BOM）があるときは、評価されたのち除去されます。BOM がないときは、その文字列はマシンのネイティブなバイト順序であると期待されます（Intel x86 アーキテクチャではネイティブなバイト順序はリトルエンディアンで、一方 Sparc・PowerPC システムではビッグエンディアン）。
utf16be	文字列は UTF-16 形式で、バイト順序はビッグエンディアンであると期待されます。バイト順序マークについて特別な扱いはありません。
utf16le	文字列は UTF-16 形式で、バイト順序はリトルエンディアンであると期待されます。バイト順序マークについて特別な扱いはありません。
auto	内容文字列：8 ビットエンコーディングと非 Unicode CMap に対しては bytes と等価で、ワイドキャラクタ指定（unicode か glyphid、または UCS2 か UTF16 の CMap）に対しては utf16 と等価です。 ハイパーテキスト文字列：BOM つきの UTF-8・UTF-16 文字列は検知されます（C では UTF-16 文字列はダブル null で終了する必要があります）。文字列の先頭に BOM がないときは、8 ビットエンコーディングの文字列として、hypertextencoding パラメタに従って解釈されます。 この設定にしておけば、Unicode をネイティブに用いないたいていの環境で、適切なテキスト解釈ができるようになります。

textformat の設定はあらゆるエンコーディングに対して効果がありますが、とりわけ *unicode* エンコーディングに対して有用です。エンコーディングと *textformat* 設定のさまざまな組み合わせに対するテキスト文字列の解釈を表 4.6 で説明します。内容文字列内のコードまたは Unicode 値が、選ばれたフォント内の適切なグリフで表現できない場合については、オプション *glyphcheck* が PDFlib の動作を制御します（123 ページ「グリフ置換」参照）。

表 4.6 エンコーディングとテキスト形式の関係

[hypertext]encoding	textformat=bytes	textformat=utf8・utf16・utf16be・utf16le
すべての文字列種別：		
auto	100 ページ「自動エンコーディング」の項を参照	
unicode と UCS2 か UTF16 の CMap	8 ビットコードは U+0000 ～ U+00FF の Unicode 値	選ばれたテキスト形式に従ってエンコードされた任意の Unicode 値 [1]

表 4.6 エンコーディングとテキスト形式の関係

[hypertext]encoding	textformat=bytes	textformat=utf8・utf16・utf16be・utf16le
その他全 CMap（非 Unicode ベース）	選ばれた CMap に従う任意のシングル・マルチバイトコード	PDFlib は例外を発生させます
内容文字列のみ：		
8 ビットと builtin	8 ビットコード	選ばれたエンコーディングに従って Unicode 値を 8 ビットコードに変換します[1]。内容文字列でないときや、フォント内に 8 ビットエンコーディングが見つからないときは、PDFlib は例外を発生させます（8 ビットエンコーディングが得られるのは Type 1・Type 3 フォント）。
glyphid	8 ビットコードは 0 ～ 255 のグリフ ID	Unicode 値はグリフ ID として解釈されます[2]

1. その Unicode キャラクタがフォント内で得られないときは、PDFlib は glyphcheck オプションに従って、例外を発生させるか、またはそれを置き換えます。
2. そのグリフ ID がフォント内で見つからないときは、PDFlib は glyphcheck 設定に従って、例外を発生させるか、またはそれをグリフ ID 0 に置き換えます。

オプションリスト内の文字列　オプションリスト内の文字列については、特別な注意が必要です。なぜなら Unicode 非対応の言語バインディングでは、それは UTF-16 形式の Unicode 文字列として表現できず、バイト文字列としてしか表現できないからです。この理由から、Unicode のオプションに対しては UTF-8 が用いられます。PDFlib はオプションの先頭の BOM を見ることで、それをどう解釈するかを決定します。BOM を用いて文字列の形式が決定され、そして文字列の種類（上述の内容文字列・ハイパーテキスト文字列・名前文字列）を用いて適切なエンコーディングが決定されます。具体的には、文字列オプションの解釈は下記のように動作します：

- オプションの先頭に UTF-8 BOM（*oxEF oxBB oxBF*）があるなら、それは UTF-8 として解釈されます。EBCDIC ベースのシステムの場合：オプションの先頭に EBCDIC UTF-8 BOM（*ox57 ox8B oxAB*）があるなら、それは EBCDIC UTF-8 として解釈されます。BOM が見つからないときは、文字列の解釈は文字列の種類に依存します。
- 内容文字列は、適用可能な *encoding* オプションか、あるいは対応するフォントのエンコーディング（どちらか存在するほう）に従って解釈されます。
- ハイパーテキスト文字列は、*hypertextencoding* パラメタかオプションに従って解釈されます。
- 名前文字列は、*usehypertextencoding=true* なら *hypertext* の設定に従って、そうでなければ *host* エンコーディングで解釈されます。

なお、キャラクタ { } はオプションリスト内の文字列内では特別な扱いを要し、文字列オプション内で用いるときはキャラクタ \ を前につける必要があります。この要請は、Shift-JIS のようなレガシエンコーディングについても効いてきます。すなわち、バイト値 *ox7B* と *ox7D* が出現するときは必ず、前に *ox5C* をつけなければなりません。この理由から、UTF-8 をオプションで使うことを推奨します（Shift-JIS 等のレガシエンコーディングでなく）。

4.5 キャラクタを指定

環境によってはプログラマーは、ソースコードを 8 ビットエンコーディング（*winansi*・*macroman*・*ebcdic* 等）で書くことを要求されます。このような環境で Unicode キャラクタを用いたいときは、テキストの全キャラクタをマルチバイトエンコーディングに変えることは許されないため、その一部の Unicode キャラクタだけを 8 ビットエンコーディングのままのテキストの中へ入れ込む、などということができるのかと悩んでしまいます。この状況から開発者を救うため、PDFlib では、テキストを表す補助手段がいくつか使えます。

4.5.1 エスケープシーケンス

PDFlib は、**エスケープシーケンス**（これは実際には誤称です：**バックスラッシュ置換**という用語のほうがよいでしょう）というしくみを通じてテキスト文字列内に任意の値を簡単に入れ込める方式に対応しています。たとえば、テキストブロックのデフォルトテキスト内で *\t* シーケンスを使えば、直接キーボード入力では無理かもしれないタブキャラクタが入れられます。同様にエスケープシーケンスは、記号フォントにおけるコードを表すにも便利ですし、エスケープシーケンスを持たない言語バインディングにおいてはリテラル文字列をあらわすにも便利です。

エスケープシーケンスは、シーケンスを 1 個のバイト値へ置換する命令です。シーケンスは、文字列のカレントエンコーディング内のバックスラッシュキャラクタ「\」に対するコードで開始します。エスケープシーケンスの置換から得られるバイト値を表 4.7 に示します。ASCII と EBCDIC のプラットフォームでは違うものもあります。エスケープシーケンスで表せるのは、0 ～ 255 のバイト値だけです。

いくつかのプログラミング言語と異なり、PDFlib のエスケープシーケンスはその種類に応じてつねに固定長となります。ですのでシーケンスの終了キャラクタは必要ありません。

表 4.7 エスケープシーケンスとバイト値一覧

シーケンス	長さ	Mac・Windows・Unix	EBCDIC プラットフォーム	広く知られる解釈
\f	2	0C	0C	フォームフィード
\n	2	0A	15/25	ラインフィード
\r	2	0D	0D	キャリッジリターン
\t	2	09	05	水平タブ
\v	2	0B	0B	ラインタブ
\\	2	5C	E0	バックスラッシュ
\xNN	4	バイト値を表す 16 進 2 桁。例：\xFF		
\NNN	4	バイト値を表す 8 進 3 桁。例：\377		

エスケープシーケンスはデフォルトでは置換されません。エスケープシーケンスを文字列において使いたいなら、*escapesequence* パラメタかオプションを明示的に *true* に設定する必要があります：

```
p.set_parameter("escapesequence", "true");
```

クックブック 完全なコードサンプルがクックブックの fonts/escape_sequences トピックにあります。

エスケープシーケンスはすべての内容文字列・ハイパーテキスト文字列・名前文字列内で、BOM 検出の後に、しかしターゲット形式への変換の前に、評価されます。*textformat=utf16le* か *utf16be* ならエスケープシーケンスは、選ばれた形式に従って 2 バイト値として表す必要があります。エスケープシーケンス内の各キャラクタは 2 バイトで表現され、そのうち 1 バイトは値 0 になります。*textformat=utf8* なら、生成コードは UTF-8 に変換されません。

エスケープシーケンスが解決できないときには（*\x* の後の 16 進数が不正等）、例外が発生します。内容文字列についてはこの動作は、*glyphcheck*・*errorpolicy* 設定で制御されます。

エスケープシーケンスを有効にしているときは、バックスラッシュキャラクタを含む Windows のパス名に注意してください。

4.5.2 文字参照

クックブック 完全なコードサンプルがクックブックの fonts/character_references トピックにあります。

文字参照は、置換シーケンスを Unicode 値で置換する命令です。参照シーケンスは、カレントエンコーディング内のアンパサンドキャラクタ「&」のコードで開始し、セミコロンキャラクタ「;」のコードで終了します。ターゲット Unicode 値を表現するためにいくつかの方式が利用可能です：

HTML 文字参照　PDFlib は、HTML 4.0 で定義されているすべての文字実体参照に対応しています。数値文字参照は 10 進・16 進記法で与えることができます。HTML 文字参照の全一覧は下記の場所にあります：

```
www.w3.org/TR/REC-html40/charset.html#h-5.3
```

例：

```
&shy;           U+00AD ソフトハイフン
&euro;          U+00AD ユーログリフ（実体名）
&lt;            U+00AD 小なり記号
&gt;            U+00AD 大なり記号
&amp;           U+00AD アンパサンド記号
&Alpha;         U+0391 ギリシャ文字 A
```

数値文字参照　Unicode キャラクタに対する数値文字参照も HTML 4.0 で定義されています。これはハッシュキャラクタ「#」と 10 進または 16 進の数値を必要とし、16 進数値のは小文字か大文字の「X」キャラクタを頭につけます。例：

```
&#173;          U+00AD ソフトハイフン
&#xAD;          U+00AD ソフトハイフン
&#229;          U+00AD 文字aの上に小さな丸（10進）
&#xE5;          U+00E5 文字aの上に小さな丸（16進、小文字x）
&#Xe5;          U+00E5 文字aの上に小さな丸（16進、大文字X）
&#x20AC;        U+20AC ユーログリフ（16進）
&#8364;         U+20AC ユーログリフ（10進）
```

注 128 ～ 159（10 進）すなわち 0x80 ～ 0x9F（16 進）のコード点は、winansi コード点を参照しません。Unicode ではこれらは、印刷可能キャラクタでなく制御キャラクタを参照します。

PDFlib 独自の実体名　PDFlib では、下記のグループの Unicode 制御キャラクタに対して、カスタムの文字実体参照を使うことができます：

- 表 6.4 に挙げるデフォルトシェーピング動作を上書きする制御キャラクタ群。
- 表 6.5 に挙げるデフォルト双方向組版を上書きする制御キャラクタ群。
- 表 8.1 に挙げるテキストフローの改行と組版のための制御キャラクタ群。

例：

```
&linefeed;          U+000A ラインフィード制御キャラクタ
&hortab;            U+0009 水平タブ
&ZWNJ;              U+200C ゼロ幅非接合子
```

グリフ名参照　グリフ名は下記の情報源から導かれます：

- 代表的なグリフ名は内蔵リスト内で検索されます
- フォント独自のグリフ名はカレントフォント内で検索されます。この種類の文字参照は必ずフォントを必要とするので、内容文字列でのみ動作します。

グリフ名参照を同定するために、実際の名前はアンパサンドキャラクタ「&」の後にピリオドキャラクタ「.」を必要とします。例：

```
&.three;            U+0033 数字3の代表的グリフ名
&.mapleleaf;        （PUA Unicode値）Cartaフォントにおけるカスタムグリフ名
&.T.swash;          （PUA Unicode値）2番目のピリオドキャラクタはグリフ名の一部です
```

グリフ名による文字参照は下記のシナリオで有用です：

- フォント独自グリフ名による文字参照は、内容文字列内で異体字（スウォッシュキャラクタ等）や、特定の Unicode セマンティクスを持たないグリフ（記号・アイコン・装飾）を選ぶのに有用です。ただし、等幅数字をはじめとする多くの機能は、OpenType 機能でもっと簡単に実装できます（154 ページ「6.3 OpenType レイアウト機能」参照）。
- Adobe グリフリスト内の名前（***uniXXXX*・*u1XXXX*** の形のものも）、および特定の共通の「名前誤り」グリフ名は、内容文字列とハイパーテキスト文字列でつねに受け入れられます。

バイト値参照　数値を文字参照で与えることもできます。これは記号フォント内のグリフを指定するのに有用でしょう。この方式では、ハッシュキャラクタ「#」を加えた 10 進または 16 進数が必要です。ここで 16 進数は小文字か大文字の「X」キャラクタを頭に付けます。例（Wingdings フォントを前提）：

```
&.#x9F;             Wingdingsフォントのビュレット記号
&.#159;             Wingdingsフォントのビュレット記号
```

文字参照の利用　文字参照はデフォルトでは置換されませんので、内容文字列内で文字参照を使うには、***charref*** パラメタまたはオプションを明示的に ***true*** に設定する必要があります。例：

```
p.set_parameter("charref", "true");
font = p.load_font("Helvetica", "winansi", "");
if (font == -1) { ... }
p.setfont(font, 24);
p.show_xy("Price: 500&euro;", 50, 500);
```

文字参照を使ううえでのその他の注意点を挙げます：

- 文字参照は、内容文字列・ハイパーテキスト文字列・名前文字列のいずれでも使えます。例外として、フォント独自グリフ名参照は上述のように内容文字列でのみ動作します。
- 文字参照は*builtin*エンコーディングのテキスト内では置換されません。しかし、*unicode*エンコーディングを用いることによって記号フォントに対して文字参照を使うことはできます。
- 文字参照はオプションリスト内では置換されません。ただし、*Unichar* データ型のオプション内では認識されます。この場合、「*&*」・「*;*」修飾は外す必要があります。この認識はつねに有効であり、*charref* パラメタ・オプションには従いません。
- Unicode 非対応言語バインディングでは、*textformat=utf16*・*utf16be*・*utf16le* のときは文字参照は 2 バイト値で表す必要があります。*encoding=unicode* かつ *textformat=bytes* のときは文字参照は ASCII で表す必要があります（EBCDIC ベースのプラットフォームでも）。
- 文字参照が解決できないとき（*&#* の後に無効な 10 進数があるときや、*&* の後に未知の実体名があるとき等）は例外が発生します。内容文字列についてはこの動作は *glyphcheck*・*errorpolicy* 設定で制御されます。*glyphcheck=none* の場合は、参照シーケンスがそのまま生成出力に現れます。

5 フォント処理

5.1 さまざまなフォント形式

5.1.1 TrueType フォント

さまざまな TrueType フォント形式　PDFlib はベクトルベースの TrueType フォントには対応していますが、ビットマップベースの TrueType フォントには対応していません。TrueType フォントファイルは自己充足しています。すなわち 1 個のファイルの中に必要な情報がすべて入っています。PDFlib は TrueType フォントについて下記のファイル形式に対応しています：

- Windows の TrueType フォント（**.ttf*）。欧文・記号・日中韓フォントを含みます。
- TrueType コレクション（**.ttc*）。1 つのファイルの中に複数のファイルが入っています。TTC ファイルは通常、日中韓フォントをグループ化するために用いられますが、Apple は、1 つの欧文フォントの複数のメンバを 1 個のファイルにパッケージするのにもこれを用いています。
- エンドユーザー定義キャラクタ（EUDC）フォント（**.tte*）。Microsoft の *eudcedit.exe* ツールで作られます。
- Mac 上では、システムにインストールされた TrueType フォント（*.dfont* を含めて）はすべて PDFlib でも使えます。

TrueType フォントの名前　フォントファイルを扱う際には、任意の別名を用いることができます（128 ページ「フォントデータの情報源」参照）。生成された PDF では、TrueType フォントの名前が PDFlib（や Windows）で用いていた名前と異なることがあります。これは正常であり、PDF は TrueType の PostScript 名を用いているという事実によるものです。PostScript 名は本当の TrueType とは異なります（例：*TimesNewRomanPSMT* に対して *Times New Roman*）。

5.1.2 OpenType フォント

OpenType フォント形式は、PostScript と TrueType 技術を結合したものです。これは TrueType ファイル形式の拡張として実装されており、統一形式を提供します。OpenType フォントは、合字やスウォッシュキャラクタ等、テキスト出力の改善に利用できるオプショナルなテーブル（154 ページ「6.3 OpenType レイアウト機能」参照）のほか、複雑用字系シェーピングのためのテーブルを含むこともできます（161 ページ「6.4 複雑用字系出力」参照）。

OpenType フォントは、すべてのプラットフォーム上で動作する単一のコンテナ形式を提供していますが、OpenType には下記のようにさまざまな種類があり、これが混乱の元になることもありますので、それを理解しておくことも有用でしょう：

- アウトライン形式：OpenType フォントは、TrueType と PostScript のいずれをベースとしたグリフ記述も内容として持つことができます。PostScript ベースのほうは Compact Font Format（CFF）や Type 2 とも呼ばれ、たいてい **.otf* 接尾辞をつけて用いられます。Windows Explorer は OpenType フォントをつねに「O」ロゴで表示します。
- TrueType フォントと、TrueType アウトラインを持った OpenType フォントとは、ともに **.ttf* 接尾辞を用いている可能性があるため、容易には見分けがつきません。この識別

の難しさから、Windows Explorer は右記の基準で動作します：*.ttf* フォントが電子署名を含んでいるならば、それは「O」ロゴで表示され、そうでなければそれは「TT」ロゴで表示されます。しかし、電子署名は OpenType フォントにおいて必須なわけではありませんので、これはプレーンな旧来の TrueType フォントと OpenType フォントとを見分ける有用な基準としては利用できません。

- CID（キャラクタ ID）アーキテクチャが日中韓フォントに対して用いられています。現代の CID フォントは、PostScript アウトラインを持つ OpenType *.otf フォントとしてパッケージされています。実用的な観点からは、これはプレーンな OpenType フォントと見分けがつきません。Windows Explorer は OpenType CID フォントをつねに「O」ロゴで表示します。

ファイル名の接尾辞も、Windows Explorer によって表示されるロゴも、フォント内に OpenType レイアウト機能があるかどうかについては何も語らないことに留意してください。詳しくは 154 ページ「6.3 OpenType レイアウト機能」を参照してください。

5.1.3 PostScript Type 1 フォント

PostScript アウトライン・メトリックファイル形式　PostScript Type 1 フォントはかならず 2 つの部分に分かれています。すなわちアウトラインデータ本体とメトリック情報です。PDFlib は、PostScript Type 1 のあらゆるプラットフォーム上でのアウトラインデータとメトリックデータのための下記のファイル形式に対応しています：

- プラットフォーム非依存な AFM（Adobe Font Metrics）形式と、Windows 固有の PFM（Printer Font Metrics）フォント形式。
- プラットフォーム独立な PFA（Printer Font ASCII）形式と、Windows 独自の PFB（Printer Font Binary）形式。どちらも PostScript Type 1 形式のフォントアウトライン情報用の形式です（「ATM フォント」ということもあります）。
- Mac 上では、リソースを利用した PostScript Type 1 フォント、すなわち LWFN（LaserWriter Font）アウトラインフォントにも対応しています。この種のフォントにはフォントスーツケース（FOND リソース。FFIL ともいう）がついており、その中にメトリックデータが入っています（スクリーンフォントも入っていますが PDFlib はそれは無視します）。
- PostScript ホストフォントを扱う際には、LWFN はそのフォントスーツケースと同じディレクトリに置かれている必要があり、かつ 5+3+3 規則に従って命名されている必要があります。

PostScript フォントの名前　フォントファイルを扱う際には、任意の別名を利用できます（128 ページ「フォントデータの情報源」参照）。フォントの内部名を知りたいとき、それを取得できる方法はいくつかあります。

- フォントアウトラインファイル（**.pfa* か **.pfb*）を開いて */FontName* 項目を見つけてその右の文字列を見ます。最初のキャラクタ（\）を無視した残りのキャラクタをフォント名として使います。
- Windows と Mac OS X 以上では、フォントファイルをダブルクリックすると、そのフォントのサンプルが現れてフォントの PostScript 名も表示されます。
- AFM メトリックファイルを開いて *FontName* 項目を見つけてその右の文字列を見ます。

注 PostScript 名は Windows のフォントメニュー名とはかなり違うことがあります。たとえば「AvantGarde-Demi」（PostScript 名）は「AvantGarde, Bold」（Windows のフォントメニュー名）となります。

5.1.4 SING フォント（グリフレット）

SING フォント（*Smart Independent Glyphlets*）は、技術的には OpenType ファイル形式の拡張です。SING フォントは、日中韓テキストにおける外字問題、すなわち Unicode や広く用いられている日中韓レガシエンコーディングのいずれにもエンコードされていないカスタムなグリフに対する解決策として開発されたものです。SING アーキテクチャに関する詳細を知るには、*Adobe Glyphlet Development Kit (GDK) for SING Gaiji Architecture* を下記の場所からダウンロードすることができます：

```
www.adobe.com/devnet/opentype/gdk/topic.html
```

SING フォントはたいてい、1 個のグリフだけを内容として持ちます（あわせて縦書き字体も含むこともあります）。この「メイン」グリフの Unicode 値は、PDFlib で取得することができ、それにはグリフ ID を要求し、ついでこのグリフ ID に対する Unicode 値を要求します：

```
maingid = (int) p.info_font(font, "maingid", "");
uv = (int) p.info_font(font, "unicode", "gid=" + maingid);
```

SING フォントを、*PDF_load_font()* の *fallbackfonts* オプションの *forcechars* オプションの *gaiji* サブオプションを用いて予備フォントとして利用することを推奨します。詳しくは 172 ページ「6.5.3 EUDC・SING フォントで外字キャラクタを利用」を参照してください。

クックブック **完全なコードサンプルがクックブックの** `fonts/starter_fallback` **トピックにあります。**

廉価な FontLab SigMaker ツールを利用して、既存の画像や他フォント内のグリフをベースに SING フォントを生成することもできます：

```
www.fontlab.com/font-utility/sigmaker/
```

5.1.5 Type 3 フォント

他のすべてのフォント形式とは異なり、Type 3 フォントはディスクファイルから取得されるのではなく、標準 PDFlib グラフィック関数群を用いて実行時に定義される必要があります。Type 3 フォントは下記の用途で有用です：

- ビットマップフォント
- ロゴ等のカスタムグラフィックを、シンプルなテキスト操作命令で簡単に印刷可能
- いずれの定義済みのフォントやエンコーディングでも入手できない日本語の外字（ユーザー定義キャラクタ）。

Type 3 フォントの定義の中では、PDFlib のベクトルグラフィック・ラスタ画像の機能がすべて使えますし、テキスト出力の機能でさえすべて使うことができるので、Type 3 フォントのキャラクタの中身に関しては制約は何もありません。PDF 取り込みライブラリ PDI と組み合わせれば、さまざまな描画の組み合わせを 1 枚の PDF のページとして取り込んで、それを用いて Type 3 フォントのキャラクタを定義することさえ可能です。しかし、Type 3 フォントが最もしばしば利用されるのはビットマップグリフのためであり、それはこれが PDF 内でグリフにラスタ画像を使える唯一のフォント形式だからです。

Type 3 フォントはページの外で完全に定義する必要があります（より厳密にいえば、フォント定義は**文書**スコープ中で行われなければなりません）。以下の作成例は、単純な Type 3 フォントを定義しています。

```
p.begin_font("Fuzzyfont", 0.001, 0.0, 0.0, 0.001, 0.0, 0.0, "");

p.begin_glyph("circle", 1000, 0, 0, 1000, 1000);
p.arc(500, 500, 500, 0, 360);
p.fill();
p.end_glyph();

p.begin_glyph("ring", 400, 0, 0, 400, 400);
p.arc(200, 200, 200, 0, 360);
p.stroke();
p.end_glyph();

p.end_font();
```

クックブック **完全なコードサンプルがクックブックの** `fonts/starter_type3font`・`fonts/type3_bitmaptext`・`fonts/type3_rasterlogo`・`fonts/type3_vectorlogo` **トピックにあります。**

こうしたフォントは PDFlib の中に登録され、その名前は *PDF_load_font()* に与えることができます。その際、その Type 3 フォントの中のグリフの名前を含むエンコーディングも一緒に指定する必要があります。Type 3 フォントを扱う時は以下のことに留意してください。

- パターンやテンプレートでの場合と同様、グリフ定義の中で画像を開くことはできません。ただし、グリフ定義開始前に開いておいた画像をグリフ定義内で貼り付けることは可能です。あるいは、小さなビットマップの場合ならインライン画像を用いてこの制約を克服することもできます。
- PDF 読み込み機能を持つさまざまなソフトウェアが持つ制約のため、テキスト出力内で使われるキャラクタは、すべて実際にフォント中で定義されていなければなりません。具体的には、文字コード *x* を任意のテキスト出力関数で表示したい場合、エンコーディングの位置 *x* に *glyphname* があるならば、その *glyphname* は *PDF_begin_glyph()* で定義されていなければなりません。
- PDF 読み込み機能を持つソフトウェアのなかには、対応するグリフ名がフォント中で定義されていないコードが用いられる場合、*.notdef* という名前のグリフを必要とするものがあります。Acrobat 8 は、*.notdef* グリフがないとクラッシュすることさえあります。*.notdef* グリフがありさえすればよく、その中身は空のグリフ定義でかまいません。
- 普通のビットマップデータを用いてキャラクタを定義する場合、ビットマップ中の使われていないピクセルは、背景にかかわらず白く印刷されます。これを避けて、元の背景色が透けて見えるようにするには、ビットマップ画像を作成する際に *mask* パラメタを用います。
- 画像に対して *interpolate* オプションを指示すると、Type 3 ビットマップフォントの画面上や印刷時の見栄えを向上させるのに有用です。
- Type 3 フォントには、アセンダやディセンダなど、タイポグラフィ的なプロパティは一切含まれていません。しかし、*PDF_load_font()* のそれぞれ対応するオプションを使えば設定できます。

5.2 Unicode のキャラクタとグリフ

5.2.1 グリフ ID

フォントはグリフの集合であり、各グリフがその輪郭によって定義されています。PDFlib は、フォント内の各グリフに番号を割り当てます。この番号をグリフ ID または GID といいます。GID 0（ゼロ）は、すべてのフォント形式において *.notdef* グリフを指します。*.notdef* グリフの見た目はフォント形式やベンダによって異なりますが、よくある実装は空白グリフか白四角か四角バッテンです。最高の GID は、そのフォント内のグリフ数より 1 少ない数であり、これは *PDF_info_font()* の *numglyphs* キーワードで取得することができます。

グリフ ID の割り当て方はフォント形式によって異なります：

- TrueType・OpenType フォントはすでに内部 GID を含んでいますので、PDFlib はこの GID を用います。
- CID キー付き OpenType 日中韓フォントでは、CID が GID として用いられます。
- それ以外のフォント種別については、PDFlib がグリフに、フォント内のそのアウトライン記述の順番に従って付番します。

PDFlib では、Unicode などのエンコーディングではなく GID でグリフを選ぶこともできます（127 ページ「グリフ ID エンコーディング」参照）。直接 GID 指定は、グリフ数を取得して全グリフをなめることでフォントの概観表を印刷する等、特殊な応用でのみ有用です。

5.2.2 グリフに対する Unicode マッピング

Unicode マッピング　PDFlib は、各 GID に一意な Unicode 値を割り当てます。このマッピング処理はフォント形式によって異なりますので、対応しているフォント種別ごとに以下の各項で解説しています。各 GID には一意な Unicode 値が割り当てられますが、その逆は必ずしも真ではありません。すなわち、ある 1 つのグリフが複数の Unicode 値を表すこともあります。多くの TrueType・OpenType フォントでよくある例は、空グリフが U+0020 の空白と U+00A0 のノーブレークスペースの両方を表す場合や、1 つのグリフが U+2126 のオーム記号と U+03A9 のギリシャ文字 Ω を表す場合です。複数の Unicode 値がフォント内の同一のグリフを指している場合、PDFlib はそのフォント内で最初に見つかった Unicode 値を割り当てます。

マップなしグリフと私用領域（PUA）　場合によっては、ある特定のグリフに対する Unicode 値をフォントが提供していないことがあります。この場合には、PDFlib は Unicode 私用領域（97 ページ「4.1 Unicode の重要な概念」参照）内の値をそのグリフに割り振ります。このようなグリフを**マップなしグリフ**といいます。フォント内のマップなしグリフの数は、*PDF_info_font()* の *unmappedglyphs* キーワードで取得することができます。マップなしグリフは、フォントの検索性とテキスト抽出を司る ToUnicode CMap 内では Unicode 置換キャラクタ U+FFFD で表されます。結果としてマップなしグリフは、生成された PDF からテキストとして正しく抽出できなくなります。

PDFlib が PUA 値をマップなしグリフに割り振っていく際には、下記のプール内の若い値から順に用いていきます：

- 基本となるのは基本多言語面（BMP）内の Unicode PUA 領域、すなわち範囲 U+E000 ～ U+F8FF です。必要であればこれに加え、第 15 面（U+F0000 to U+FFFFD）内の PUA 値も用いられます。

- そのフォントが内部的にすでに割り振っている PUA 値は、新たな PUA 値を作成する際には用いられません。
- Adobe 領域 U+F600 ～ F8FF 内の PUA 値は用いられません。

生成される PUA 値は、1 つのフォント内で一意です。あるフォント内のグリフに対して生成される PUA 値の割り振りは、他のフォントからは独立です。

TrueType・OpenType・SING フォントに対する Unicode マッピング　PDFlib は、フォントの *cmap* テーブルで見つかった Unicode マッピングを保持します（*cmap* の選択は、*PDF_load_font()* に与えるエンコーディングに依存します）。1 つのグリフが複数の Unicode 値に対して用いられている場合、PDFlib はそのフォント内で見つかった最初の Unicode 値を用います。

cmap が何の Unicode マッピングも提供していないグリフについては、PDFlib はそのグリフ名を *post* テーブル（そのフォント内にあれば）内でチェックし、Type 1 フォントについて後述するようにそのグリフ名に基づいて Unicode マッピングを決定します。

場合によっては、*cmap* も *post* テーブルもそのフォント内のすべてのグリフに対する Unicode 値を提供していないことがあります。これは Unicode 規格外の異体字（スウォッシュキャラクタ等）・拡張合字・非テキスト記号についてあてはまります。この場合 PDFlib は、119 ページ「マップなしグリフと私用領域（PUA）」で述べたように問題のグリフに PUA 値を割り当てます。

Type 1 フォントに対する Unicode マッピング　Type 1 フォントは明示的な Unicode マッピングを内蔵しておらず、各グリフに一意な名前を割り当てています。PDFlib はこのグリフ名に基づいて Unicode 値の割り当てを試みます。そのためには内蔵の、さまざまな言語や用字系に対して広く用いられる 7,000 種を超すグリフ名に対する Unicode マッピングを内容として持つマッピングテーブルが使われます。このマッピングテーブルには Adobe Glyph List（AGL）[1] 内のおよそ 4,200 種のグリフ名が含まれています。しかし、Type 1 フォントはこの内蔵マッピングテーブルに含まれていないグリフ名を含んでいることがあり、これはとりわけ記号フォントについて顕著です。この場合 PDFlib は、119 ページ「マップなしグリフと私用領域（PUA）」で述べたように問題のグリフに PUA 値を割り当てます。

Type 1 フォントに対するメトリックが PFM ファイルから読み込まれ、PFB または PFA アウトラインファイルが得られないときは、PDFlib はそのフォントのグリフ名を知ることができません。この場合、PDFlib は Unicode 値を PFM ファイル内のエンコーディング（charset）項目に基づいて割り当てます。

Type 3 フォントに対する Unicode マッピング　Type 3 フォントもグリフ名に基づいていますので、Type 1 フォントと同様に扱われます。ただし重要な違いは、Type 3 フォントではグリフ名はユーザーの制御下にある（*PDF_begin_glyph()* の *glyphname* 引数を通じて）ということです。ですので、ユーザー定義 Type 3 フォントに対しては AGL 内の適切なグリフ名を用いることを強く推奨します。これによって、正しい Unicode 値が PDFlib によって自動的に正しく割り当てられるようになり、生成される PDF 文書内でテキストが検索可能になります。

5.2.3 Unicode 制御キャラクタ

制御キャラクタは、いかなるグリフをも表さず、何らかの組版情報の伝達に用いられる Unicode 値です。PDFlib は、下記のグループの Unicode 制御キャラクタを処理します：

1. AGL は *partners.adobe.com/public/developer/en/opentype/glyphlist.txt* にあります。

- デフォルトのシェーピング動作を上書きするための制御キャラクタ（表 6.4 に挙げる）と、デフォルトの双方向組版を上書きするための制御キャラクタ（表 6.5 に挙げる）は、テキスト行・テキストフロー内の複雑用字系のシェーピングと OpenType レイアウト機能の処理を制御します。これらの制御キャラクタは、評価された後に削除されます。
- 表 8.1 に挙げる改行とテキストフロー組版のための組版制御キャラクタ。これらの制御キャラクタは、評価された後に削除されます。
- これ以外の範囲 U+0001 ～ U+0019・U+007F ～ U+009F の Unicode 制御キャラクタは**置換**キャラクタで置換されます。

フォントが制御キャラクタに対するグリフを含んでいたとしても、そのグリフは通常は視覚化されません。なぜなら PDFlib が制御キャラクタを除去するからです（この規則の例外として *** *** と ***&SHY;*** は除去されません）。ただし、***encoding=glyphid*** の場合は制御キャラクタはテキスト内に保持され、視覚出力を生み出すことができます。

5.3 テキスト処理パイプライン

クライアントアプリケーションは、ページ出力したいテキストを PDFlib に与えます。このテキストは、アプリケーション個別の何らかのエンコーディングと形式に従ってエンコードされています。しかし、PDFlib の内部処理は Unicode 規格に基づいており、また最終テキスト出力はフォント固有のグリフ ID を必要とします。ですので PDFlib は、ページ内容のために与えられる文字列をテキスト処理パイプラインの中で3つの過程を経て処理します：

- 入力コードを Unicode 値へ正規化。この処理は選択されているエンコーディングにより制約を受けます。
- Unicode 値をフォント固有のグリフ ID へ変換。この処理はそのフォント内で利用可能なグリフにより制約を受けます。
- グリフ ID を転換。この処理は出力エンコーディングにより制約を受けます。

テキスト処理パイプラインのこれら 3 つの過程は、いくつかの下部処理を含んでおり、それはオプションで制御することができます。

5.3.1 入力文字列を Unicode へ正規化

以下のステップは、***encoding=glyphid*** と非 Unicode CMap 以外のすべてのエンコーディングに対して実行されます：

- Unicode 対応言語バインディング：シングルバイトエンコーディングが指定されているときは、UTF-16 テキストは高次バイトを捨てることでシングルバイトテキストへ変換されます。
- Windows：マルチバイトテキスト（***cp932*** 等）を Unicode へ変換します。
- エスケープシーケンス（111 ページ「4.5.1 エスケープシーケンス」参照）を、対応する数値へ置き換えます。
- 文字参照を解決し、それを対応する Unicode 値へ置き換えます（112 ページ「4.5.2 文字参照」および次項参照）。
- シングルバイトエンコーディング：シングルバイトテキストを、指定されたエンコーディングに従って Unicode へ変換します。

さまざまなフォント形式やキャラクタの種別に対する Unicode の割り当てについて詳しくは、119 ページ「5.2.2 グリフに対する Unicode マッピング」を参照してください。

グリフ名による文字参照　フォント内のグリフは、その対応する Unicode 値をあらかじめ知ることができない（PDFlib が実行時に PUA 値を割り振るので）ものがあり、そのようなグリフは直接指定することができません。そのようなグリフを指定するには、グリフ名による文字参照を使うことができます。文法の解説は 112 ページ「4.5.2 文字参照」を参照してください。この参照は、対応する Unicode 値へ置き換えられます。

文字参照が内容文字列内で用いられているときは、PDFlib はその指定されたグリフをカレントフォント内で見つけようと試み、そしてその参照をそのグリフの Unicode 値へ置き換えます。指定されたグリフがフォント内で見つからないときは、PDFlib は Unicode 値を決定するためにその内蔵グリフ名テーブルを検索します。この Unicode 値はさらに、適切なグリフがフォント内で得られるかどうかを調べるために使われます。そのようなグリフが見つからないときは、動作は ***glyphcheck*** ・ ***errorpolicy*** 設定で制御されます。文字参照は、***glyphid*** ・ ***builtin*** エンコーディングでは使うことができません。

5.3.2 Unicode 値をグリフ ID へ変換

前項で決定された Unicode 値は、いくつかの理由により変更が必要な場合があります。以下のステップは、下記のように処理される *encoding=glyphid* と非 Unicode CMap 以外のすべてのエンコーディングに対して実行されます：

- 非 Unicode CMap の場合：無効なコード列はつねに例外を発生させます。
- *encoding=glyphid* の場合：無効なグリフ ID は、*replacementchar*（*glyphcheck=replace* のとき）かグリフ ID 0（*glyphcheck=none*）へ置き換えられます。*glyphcheck=error* のときは例外が発生します。

予備フォント内の強制キャラクタ　Unicode 値を、*fallbackfonts* オプションの *forcechars* サブオプションに従って置き換えて、対応する予備フォントのグリフ ID を決定します。詳しくは 135 ページ「5.4.5 予備フォント」を参照してください。

グリフ ID へ変換　残りの Unicode 値を、119 ページ「5.2.2 グリフに対する Unicode マッピング」で決定されるマッピングに従ってグリフ ID へ変換します。Unicode 値に対応するグリフ ID がフォント内に見つからないときは、その次のステップは *glyphcheck* オプションによって異なります：

- *glyphcheck=none*：グリフ ID 0 が用いられます。これはすなわち、*.notdef* グリフがテキスト出力内で用いられることを意味します。*.notdef* グリフが視覚的形状を内容として持つ場合（たいていは白四角か四角バッテン）には、問題の起きたキャラクタが PDF ページ上で目に見えるようになります。それが望ましいかどうかは場合によるでしょう。
- *glyphcheck=replace*（これがデフォルト）：警告メッセージがログ記録され、PDFlib は、マップできない Unicode 値を後述のグリフ置換機構によって置き換えようと試みます。
- *glyphcheck=error*：PDFlib はエラーを発生させます。*errorpolicy=return* の場合には、これはすなわち関数呼び出しがテキスト出力を一切生成せずに終了されることを意味し、*PDF_add/create_textflow()* が −1（PHP では 0）を返すことを意味します。*errorpolicy=exception* の場合は例外が発生します。

グリフ置換　*glyphcheck=replace* の場合は、マップできない Unicode 値は再帰的に以下のように置き換えられます：

- マスターフォントを読み込んだ際に指定した予備フォントの中で、その Unicode 値に対するグリフが検索されます。各フォントに対して予備フォントは複数指定することもできるので、ここでは任意の数のフォントが関わる可能性があります。予備フォントのうちの 1 つでグリフが見つかったときはそれが使われます。
- タイポグラフィ的に類似のグリフを、PDFlib の内蔵置換テーブル内の Unicode 値に従って選びます。この内蔵リストからこれらの置換のいくつかを以下に抜粋します。リスト内の 1 番目のキャラクタがフォント内で見つからないとき、それは 2 番目のキャラクタへ置き換えられます：

U+00A0（ノーブレークスペース）	U+0020（空白）
U+00AD（ソフトハイフン）	U+002D（ハイフン-マイナス）
U+2010（ハイフン）	U+002D（ハイフン-マイナス）
U+03BC（ギリシャ文字 μ）	U+00C5（マイクロ記号）
U+212B（オングストローム記号）	U+00B5（欧文Aの上に丸）
U+220F（多項総乗演算子）	U+03A0（ギリシャ文字Π）
U+2126（オーム記号）	U+03A9（ギリシャ文字Ω）

内蔵テーブルに加えて、全角キャラクタ U+FF01 ～ U+FF5E は、フォント内で全角字体が得られない場合には、対応する ISO 8859-1 キャラクタ（すなわち U+0021 ～ U+007E）へ置き換えられます。

- Unicode の合字を、その構成グリフ群へ分解します（例：U+FB00 **欧文合字** ***ff*** をシーケンス U+0066 *f*・U+0066 *f* へ置き換え）。
- 同じ Unicode セマンティクスを持つグリフを、そのグリフ名に従って選びます。特に、ピリオドで区切られたグリフ名接尾辞はすべて、対応するグリフが得られないときは除去されます（例：***A.swash*** を ***A*** へ置き換え。***g.alt*** を ***g*** へ置き換え）。

これらの方式がいずれも Unicode 値に対するグリフを与えないときは、***replacementchar*** オプションで指定されたキャラクタが使われます。それに対応するグリフ自体がフォント内で得られないとき（または ***replacementchar*** オプションが指定されていないとき）は、U+00A0（ノーブレークスペース）と U+0020（空白）が試されます。これらさえも得られないときは、グリフ ID 0（グリフなし記号）が使われます。

PDF/A-1・PDF/X-4・PDF/X-5 では、グリフ ID 0 へマップされる入力コードはスキップされます。

クックブック **完全なコードサンプルがクックブックの** `fonts/glyph_replacement` **トピックにあります。**

5.3.3 グリフ ID を転換

決定されたグリフ ID はまだ最終的なものではありません。なぜなら最終出力を生成する前に、いくつかの転換を行わなければならない可能性があるからです。具体的にどのような転換が必要かは、フォントやいくつかのオプションによって異なります。以下のステップは、非 Unicode CMap で ***keepnative=true*** の場合を除き、すべてのエンコーディングに対して行われます。

縦書きグリフ　縦書きモードのフォントでは、いくつかのグリフは縦書き字体へ置き換わる可能性があります。この置換は、フォント内に ***vert*** OpenType レイアウト機能テーブルが必要です。

OpenType レイアウト機能　OpenType 機能は、合字・スウォッシュキャラクタ・スモールキャピタルをはじめとするさまざまなタイポグラフィバリエーションを、1 個ないし複数のグリフ ID を他の値へ置き換えることによって作り出すことができます。OpenType 機能については 154 ページ「6.3 OpenType レイアウト機能」で解説しています。OpenType レイアウト機能は、適切なフォントに対してのみ有効であり（157 ページ「OpenType レイアウト機能のための要件」参照）、かつ ***features*** オプションに従って適用されます。

複雑用字系のシェーピング　シェーピングは、テキストの順序を替え、また、キャラクタの位置によって適切な字体のグリフを決定します（例：アラビア文字キャラクタの頭字・中字・尾字・単独形）。これは適切なフォントに対してのみ有効であり（163 ページ「シェーピングのための要件」参照）、かつ ***shaping*** オプションに従って適用されます。

5.4 フォントを読み込む

5.4.1 テキストフォントに対するエンコーディングを選ぶ

フォントは、明示的に *PDF_load_font()* 関数で読み込むこともでき、あるいは暗黙的に、*PDF_add/create_textflow()*や*PDF_fill_textblock()*といった特定の関数に*fontname・encoding* オプションを与えることで読み込むこともできます。どのような方式を用いてフォントを読み込むのかにかかわらず、適切なエンコーディングを指定する必要があります。エンコーディングは下記を決定します：

- PDFlib が与えられるテキストをどのテキスト形式であると見なすか。
- フォント内のどのグリフが使えるか。
- ページ上のテキストとフォント内のグリフデータが PDF 出力文書内にどのように格納されるか。

PDFlib のテキスト処理は Unicode 規格[1] に基づいています。これは ISO 10646 とほぼ等価です。今どきの開発環境の多くが Unicode 規格に対応していますので、Unicode 文字列を使ってできるだけ簡単に PDF 出力を生成できるようにすることが私たちの目標です。ただし、Unicode を扱わない開発者はそのアプリケーションを Unicode へ切り替える必要はありません。レガシエンコーディングも使うことができるからです。

どのエンコーディングを選ぶかは、フォントや、得られるテキストデータや、いくつかのプログラミング的側面によって決まります。以下この項では、さまざまな分類のエンコーディングの概観を示すことで、適切なエンコーディングを選ぶための助けとします。

Unicode エンコーディング　*encoding=unicode* とすると、Unicode 文字列を PDFlib に渡すことができます。このエンコーディングはすべてのフォント形式に対して使えます。利用する言語バインディングによって、そのプログラミング言語（Java 等）が提供している Unicode 文字列データ型を利用できる場合もあれば、Unicode を UTF-8・UTF-16・UTF-32 のいずれかの形式でリトルエンディアン・ビッグエンディアンのいずれかのバイト順序で内容として持つバイト配列を用いる場合もあります（C 等）。

encoding=unicode では、フォント内のすべてのグリフが指定でき、複雑用字系のシェーピングと OpenType レイアウト機能を使うことができます。PDFlib は、要求された Unicode 値に対するグリフをフォントが含んでいるかどうかを調べます。グリフが得られないときは、代替グリフを同一フォントか別フォントから持って来ることができます（135 ページ「5.4.5 予備フォント」参照）。

Unicode 非対応の言語バインディングでは、PDFlib はデフォルトではテキストが UTF-16 エンコードされていると見なします。しかし、*textformat=bytes* を指定すればシングルバイト文字列を与えることができます。この場合、このバイト値はキャラクタ U+0001 ～ U+00FF を、すなわち基本欧文キャラクタ群を持つ先頭 Unicode ブロック（ISO 8859-1 と等価）を表します。ただし、文字参照を利用すれば、この範囲の外の Unicode 値をシングルバイトテキスト内で指定することも可能です。

PDF 内のいくつかのフォント形式（Type 1、Type 3、グリフ名に基づく OpenType フォント）は、シングルバイトテキストにのみ対応しています。しかし、PDFlib ではこうした種類のフォントでも 255 種類を超えるキャラクタを扱えるよう工夫しています。

encoding=unicode の難点は、昔ながらのシングルバイトやマルチバイトのエンコーディングのテキストを用いることができない（ISO 8859-1 を除き）ことです。

1. *www.unicode.org* を参照。

シングルバイトエンコーディング　8ビットエンコーディング（シングルバイトエンコーディングともいう）は、テキスト文字列内の各バイトを1個のキャラクタへマップしますので、同時に扱えるキャラクタは255種類までに制限されます（値0は利用できません）。この種類のエンコーディングはすべてのフォント形式に対して使えます。PDFlibは、選ばれたエンコーディングに合ったグリフをフォントが含んでいるかどうかを調べます。使えるグリフの数が最低限の数に届かないときは、PDFlib は警告メッセージをログ記録します。選ばれたエンコーディングに対して使えるグリフがフォント内でまったく得られないときは、フォント読み込みは「*font doesn't support encoding*」というメッセージを出して失敗します。PDFlibは、要求された入力値に対するグリフをフォントが含んでいるかどうかを調べます。グリフが得られないときは、代替グリフを同一フォントか別フォントから持って来ることができます（135ページ「5.4.5 予備フォント」参照）。

Unicode非対応の言語バインディングでは、PDFlibはデフォルトではテキストがシングルバイトエンコードされていると見なします。しかし、*textformat=utf8* か *utf16* を指定すればUTF-8かUTF-16文字列を与えることができます。

8ビットエンコーディングについて詳しくは99ページ「4.2 シングルバイト（8ビット）エンコーディング」で解説しています。これはさまざまな情報源から持って来ることができます：

- 表4.2に従った大量の定義済みエンコーディング。これはさまざまなシステム、さまざまなロケールで現在利用されている最も重要なエンコーディング群を網羅しています。
- ユーザー定義エンコーディング。これは、外部ファイルで与えるか、または実行時に *PDF_encoding_set_char( )* で動的に構築することができます。このエンコーディングは、グリフ名かUnicode値に基づくことができます。
- オペレーティングシステムから持ってきたエンコーディング。**システムエンコーディング**ともいいます。この機能は、Windows と IBM iSeries・zSeries で利用可能です。

シングルバイトエンコーディングの難点は、キャラクタやグリフの数に限りがあることです。この理由から、複雑用字系のシェーピングとOpenTypeレイアウト機能はシングルバイトエンコーディングに対しては使えません。

ビルトインエンコーディング　*encoding=builtin* を指定して、記号フォント内の非テキストグリフに対するシングルバイトコードを使うという方法もあります。フォントの内部エンコーディングの形式はフォントの種類によって異なります：

- TrueType：エンコーディングは、フォントのシンボリック cmap に、すなわち *cmap* テーブル内の *(3, 0)* 項目に基づいて作られます。
- OpenType フォントはエンコーディングを CFF テーブル内に含んでいることがあります。
- PostScript Type 1フォントはつねにエンコーディングを含んでいます。
- Type 3の場合、エンコーディングはフォントの先頭255グリフによって定義されます。

フォントがビルトインエンコーディングを何ら含んでいないときは、フォント読み込みは失敗します（OpenType 日中韓フォント等）。*PDF_info_font( )* で *symbolfont* キーを用いることもできます。これが *false* を返したなら、そのフォントはテキストフォントであり、広く利用されているシングルバイトエンコーディングのいずれかで読み込むこともできます。*symbolfont* キーが *true* を返したならそれはできません。そのような記号フォント内のグリフは、各グリフに対するコードがわかっている場合にのみ利用することが可能です（127ページ「5.4.2 記号フォントに対するエンコーディングを選ぶ」参照）。

Unicode 非対応の言語バインディングでは、PDFlib はデフォルトではテキストがシングルバイト形式であると見なします。これは、いくつかの記号フォントでの指定に伝統的に使われているシングルバイト値を使えるという利点があります。これは他のエンコーディングでは不可能です。しかし、*textformat=utf16* 等を指定すればテキストを Unicode 形式で与えることができます。

encoding=builtin の難点は、シングルバイトエンコードされたテキスト内では文字参照が使えないことです。

マルチバイトエンコーディング　この種類のエンコーディングは日中韓フォントに、すなわち日本語・中国語・韓国語のキャラクタ群を持つ TrueType・OpenType CID フォントに対して使えます。これらの用字系で使うためにさまざまなエンコーディング方式が開発されてきました。日本語では Shift-JIS・EUC、中国語では GB・Big5、韓国語では KSC 等です。マルチバイトエンコーディングは Adobe CMap か Windows コードページで定義されています（103 ページ「4.3 日本語・中国語・韓国語エンコーディング」参照）。

これらの伝統的エンコーディングは、*encoding=unicode* と等価である Unicode CMap を除き、Unicode 非対応言語バインディングでのみ使うことができます。

Unicode 非対応の言語バインディングでは、PDFlib はデフォルト（*textformat=bytes*）ではテキストがマルチバイトバイトエンコードされていると見なします。

マルチバイトエンコーディングでは、*keepnative* オプション *true* がならば、テキストはユーザーから与えられたものがそのまま PDF 出力へ書き込まれます。

マルチバイトエンコーディングの難点は、PDFlib は入力テキストについて文法的有効性のみを検査し、与えられたテキストに対するグリフがフォント内で得られるかどうかは検査しない点です。また、Unicode テキストを与えることもできません。なぜなら PDFlib は Unicode 値をそれに対応するマルチバイト列へ変換することができないからです。かつ、文字参照、OpenType レイアウト機能、複雑用字系のシェーピングも利用できません。

グリフ ID エンコーディング　PDFlib では *encoding=glyphid* をすべてのフォント形式に対して使えます。このエンコーディングでは、119 ページ「5.2.1 グリフ ID」で解説する付番方式を用いて、フォント内のすべてのグリフが指定できます。数値グリフ ID は 0 から始まり、理論上の最大値は 65,565 です（そのような大量のグリフを持つフォントはありませんが）。最大グリフ ID 値は *PDF_info_font()* で *maxcode* キーを用いて取得できます。

Unicode 非対応の言語バインディングでは、PDFlib はデフォルト（*textformat=utf16*）ではテキストがダブルバイトバイトエンコードされていると見なします。

PDFlib は、与えられたグリフ ID がフォント内で有効であるかどうかを調べます。複雑用字系のシェーピングと OpenType レイアウト機能が使えます。

グリフ ID は特定のフォントに固有のものであり、場合によっては PDFlib によって生成されることすらあるので、*encoding=glyphid* は一般に通常のテキスト出力には適しません。このエンコーディングの主な用途は、フォントの全グリフ一覧を印刷することです。

5.4.2 記号フォントに対するエンコーディングを選ぶ

記号フォントは、記号・ロゴ・ピクトグラムなどの非テキストグリフ群を内容として持つフォントです。これは、テキストフォントにはないいくつかの問題を提起します。背景にある問題は、Unicode 規格は一般にあえて記号グリフをエンコードしていないことです（広く利用されている ZapfDingbats フォント内のグリフ等、この規則には例外もありますが）。記号フォントを Unicode 指向ワークフローに適合して利用できるようにするため、TrueType・OpenType フォントはたいていそのグリフに私用領域（PUA）内の Unicode 値を割り当てています。Unicode マッピングテーブルがないことから、PostScript Type 1 は

これを行えず、一般にそのグリフを選ぶのに欧文キャラクタのコードを用いています。すべてのフォント形式において、記号グリフはたいていカスタムのグリフ名を持っています。

こうした状況から、記号フォント内のグリフの選択に関して下記のような結果が生じます：

- 記号 TrueType・OpenType フォントは ***encoding=unicode*** で最も良く読み込めます。グリフに割り当てられている PUA 値がわかっている場合には、記号グリフを選ぶためにテキスト内でこの値を与えることができます。そのためにはそのフォント内の PUA 割り当てがあらかじめわかっている必要があります。
- PDFlib は記号 PostScript Type 1 フォントに対して PUA 値を内部的に割り振りますので、この PUA 値は事前にはわかりません。
- 記号フォント内のグリフを 8 ビットコードを使って指定したい場合には、フォントを ***encoding=builtin*** で読み込んで、テキスト内で 8 ビットコードを与えることができます。たとえば、数字 4（コード 0x34）は ZapfDingbats フォント内でチェックマーク記号を選びます。

記号フォントを ***encoding=unicode*** で使うためには、適切な Unicode 値をテキストで用いる必要があります：

- Symbol フォント内のキャラクタはすべて正しい Unicode 値を持っています。
- ZapfDingbats フォント内のキャラクタは範囲 U+2007 ～ U+27BF の Unicode 値を持っています。
- Wingdings・Webdings 等の Microsoft の記号フォントは、範囲 U+F020 ～ U+F0FF の PUA Unicode 値を用いています（***charmap*** アプリケーションはこれをシングルバイトコードで表しますが）。
- それ以外のフォントについては、フォント内の個々のグリフに対する Unicode 値をあらかじめ知っておくか、または実行時に ***PDF_info_font()*** で（例：PostScript Type 1 フォントの場合はグリフ名を与えて）決定する必要があります。

制御キャラクタ　表 8.1 に挙げる範囲 U+0001 ～ U+001F の Unicode 制御キャラクタは、***encoding=builtin*** でもテキストフロー内で使えます。0x20 未満のコードは、記号フォントがそのコードに対するグリフを持たないならば、制御キャラクタとして解釈されます。これは大多数の記号フォントにあてはまります。

ラインフィードキャラクタに対するコードは ASCII と EBCDIC とで異なっていますので、EBCDIC システム上でリテラルなキャラクタ 0x0A を使うことは避け、オプション ***escapesequence=true*** を用いて PDFlib のエスケープシーケンス ***\n*** を使うことを推奨します。この ***\n*** は PDFlib API まで届く必要があることに留意してください。たとえば C では ***\\n*** と書く必要があります。

文字参照　文字参照は記号フォントに対して使うことができます。しかし記号フォントは一般に、文字参照を開始するアンパサンドキャラクタ U+0026「&」に対するグリフを含んでいません。コード 0x26 も、フォント内の既存のグリフへマップできないので使えません。ですので記号フォントは、文字参照を使う必要があるときは、***encoding=unicode*** で読み込む必要があります。文字参照は ***encoding=builtin*** では動作しません。

5.4.3 フォントを検索

フォントデータの情報源　先述のように、フォントは明示的に ***PDF_load_font()*** 関数で読み込むこともできますし、あるいは暗黙的に、さまざまなテキスト出力関数に ***fontname***・

encoding オプションを与えて読み込むこともできます。フォントのネイティブな名前を使うこともできますし、フォントデータの場所を決定するために用いられる任意のカスタム名を扱うこともできます。カスタムフォント名は文書内で一意である必要があります。*PDF_info_font()* で、このフォント名は *apiname* キーで取得することができます。

PDF_load_font() を続けて呼び出すと、すべてのオプションがこの関数を最初に呼び出した際に与えたものと等しければ、同じフォントハンドルが返されます（扱いが異なるオプションが若干ありますので、詳しくは PDFlib API リファレンスを参照）。そうでなければ、同じフォント名に対して新規のフォントハンドルが作成されます。PDFlib ではフォントデータの情報源として下記に対応しています：

- ディスクベースまたは仮想のフォントファイル
- Windows または Mac オペレーティングシステムから持って来たフォント（ホストフォント）
- PDF 標準フォント：これらは、よく知られた名前の少数の欧文・日中韓フォントです
- *PDF_begin_font()* および関連関数群で定義された Type 3 フォント

クックブック **完全なコードサンプルがクックブックの** `fonts/font_resources` **トピックにあります。**

フォントの検索順序　PDFlib に与えられるフォント名は名前文字列です。PDFlib は与えられたフォント名を使って、さまざまな種類のフォントを以下の順序で探します。この検索処理は、使えるフォントがステップの 1 つで見つかるとともに終わります：

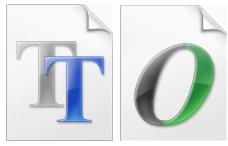
- フォント名が標準日中韓フォントの名前と一致し、指定されたエンコーディングが定義済み CMap の名前である（169 ページ「6.5.1 標準日中韓フォント」参照）。
- フォント名が、これ以前に同一文書内で *PDF_begin_font()* で作成された Type 3 フォントの名前と一致する（117 ページ「5.1.5 Type 3 フォント」参照）。
- フォント名が、フォント名を TrueType・OpenType フォントファイルの名前と紐づける *FontOutline* リソース内の名前と一致する。
- フォント名が、フォント名を PostScript Type 1 フォントのメトリックファイルの名前と紐づける *FontAFM* または *FontPFM* リソース内の名前と一致する。
- フォント名が、フォント名をシステムにインストールされているフォントの名前と紐づける *HostFont* リソース内の名前と一致する。
- フォント名が欧文コアフォントの名前と一致する（131 ページ「欧文コアフォント」参照）。
- フォント名が、システムにインストールされているホストフォントの名前と一致する（133 ページ「5.4.4 Windows・Mac OS X 上のホストフォント」参照）。
- フォント名がフォントファイルのベース名（すなわちファイル名接尾辞を除いた名前）と一致する。

フォントが見つからなかったときは、フォント読み込みは下記のエラーメッセージを出して止まります：

```
Font file (AFM, PFM, TTF, OTF etc.) or host font not found
```

さまざまなリソースカテゴリについて詳しくは 59 ページ「3.1.3 リソース設定とファイル検索」を参照してください。以下の各項では、さまざまな分類のフォントに対するフォント読み込みについてさらに詳しく解説していきます。

TrueType・OpenType フォント　フォント名は、使いたいフォントファイルの名前に対して、*FontOutline* リソースを通じて紐づける必要があります：



```
p.set_parameter("FontOutline", "Arial=/usr/fonts/Arial.ttf");
font = p.load_font("Arial", "unicode", "embedding");
```

等号の左側のフォント別名は任意に選べます：

```
p.set_parameter("FontOutline", "f1=/usr/fonts/Arial.ttf");
font = p.load_font("f1", "unicode", "embedding");
```

実行時に *PDF_set_parameter( )* で設定するのでなく、UPR ファイル内で *FontOutline* リソースを設定することもできます（59 ページ「3.1.3 リソース設定とファイル検索」参照）。絶対ファイル名を避けるため、*SearchPath* リソースカテゴリを用いることもできます（この場合も、*SearchPath* リソースカテゴリを UPR ファイル内で設定することも可能です）。例：

```
p.set_parameter("SearchPath", "/usr/fonts");
p.set_parameter("FontOutline", "f1=Arial.ttf");
font = p.load_font("f1", "unicode", "");
```

TrueType コレクション　TrueType コレクション（TTC、171 ページ「6.5.2 カスタム日中韓フォント」参照）ファイル内に含まれているフォントを選ぶには、そのフォントの名前を直接指定します：

```
p.set_parameter("FontOutline", "MS-Gothic=msgothic.ttc");
font = p.load_font("MS-Gothic", "unicode", "embedding");
```

フォント名は、TTC ファイル内のすべてのフォントの名前と照合されます。あるいは、TTC ファイル内の *n* 番目のフォントを選ぶには、フォント名の後にコロンをつけて番号 *n* を指定することができます。この場合は、等号の左側のフォント別名は任意に選べます：

```
p.set_parameter("FontOutline", "f1=msgothic.ttc");
font = p.load_font("f1:0", "unicode", "");
```

PostScript Type 1 フォント　フォント名は、使いたいフォントのメトリックファイルの名前に対して、そのメトリックファイルの種類に応じて、*FontAFM*・*FontPFM* リソースカテゴリのいずれかを通じて紐づける必要があります：

```
p.set_parameter("FontPFM", "lucidux=LuciduxSans.pfm");
font = p.load_font("lucidux", "unicode", "embedding");
```

PostScript フォントを埋め込む必要があるときは、その名前を、そのフォントアウトラインファイル（PFA または PFB）に対して、*FontOutline* リソースカテゴリを通じて紐づける必要があります：

```
p.set_parameter("FontPFM", "lucidux=LuciduxSans.pfm");
p.set_parameter("FontOutline", "lucidux=LuciduxSans.pfa");
font = p.load_font("lucidux", "unicode", "");
```

PostScript Type 1 フォントに対しては、*FontOutline* リソースだけでは充分でないことに留意してください。必ずメトリックファイルが必要ですので、フォントを読み込むためには AFM または PFM ファイルが得られる必要があります。

フォントメトリック・アウトラインファイルが検索されるディレクトリは、*SearchPath* リソースカテゴリを通じて指定することができます。

欧文コアフォント　PDF ビューアは、つねに利用可能と見なされるフォント 14 種のコアセットに対応しています。コアフォントの完全なメトリック情報はすでに PDFlib に内蔵されていますので、追加データは必要ありません（フォントを埋め込みたい場合を除き）。コアフォントは下記の名前を持ちます：

Courier・Courier-Bold・Courier-Oblique・Courier-BoldOblique・
Helvetica・Helvetica-Bold・Helvetica-Oblique・Helvetica-BoldOblique・
Times-Roman・Times-Bold・Times-Italic・Times-BoldItalic・
Symbol・ZapfDingbats

フォント名がリソースを通じていかなるファイル名へも紐づけられていないときは、PDFlib はそのフォントを欧文コアフォントのリスト内で探します。このステップは、*embedding* オプションが指定されている場合、またはそのフォント名に対して *FontOutline* リソースが得られる場合にはスキップされます。下記のコードは、コアフォントの 1 つを何の設定もなしに要求します：

```
font = p.load_font("Times-Roman", "unicode", "");
```

内部リストで見つかったコアフォントは決して埋め込まれません。これらのフォントのいずれかを埋め込むためには、フォントアウトラインファイルを設定する必要があります。

ホストフォント　フォント名がリソースを通じていかなるファイル名へも紐づけられていないときは、PDFlib はそのフォントを、Windows または Mac システムにインストールされているフォントのリスト内で探します。システムにインストールされているフォントを**ホストフォント**といいます。ホストフォントの名前は ASCII でエンコードされている必要があります。Windows では Unicode も使えます。ホストフォントについて詳しくは 133 ページ「5.4.4 Windows・Mac OS X 上のホストフォント」を参照してください。例：

```
font = p.load_font("Verdana", "unicode", "");
```

Windows では、フォント名の後にカンマをつけてフォントスタイルを追加することもできます：

```
font = p.load_font("Verdana,Bold", "unicode", "");
```

コアフォントのいずれかの名前でホストフォントを読み込むためには、そのフォント名を、ほしいホストフォントの名前に対して、*HostFont* リソースカテゴリを通じて紐づける必要があります。下記のコードは、内蔵コアフォントデータを使うのでなく、Symbol フォントのメトリック・アウトラインデータがホストシステムから持って来られるようにしています：

```
p.set_parameter("HostFont", "Symbol=Symbol");
font = p.load_font("Symbol", "unicode", "embedding");
```

等号の左側のフォント別名は任意に選べますが、ここではホストフォントの名前をそのまま使いました。

フォントファイルを拡張子に基づいて検索　Type 3 以外のすべての種類のフォントは、指定されたフォント名をフォントメトリック・アウトラインファイルのベース名（ファイ

ル接尾辞の一切ない名前）として用いて検索することができます。PDFlib は、指定された名前のフォントが見つからなかったときは、*SearchPath* リソースカテゴリ内のすべての項目をなめながら、与えられたファイル名に対して、知られているすべてのファイル名接尾辞を付加することによって、フォントメトリック・アウトラインデータを見つけようと試みます。この拡張子に基づく検索は具体的には以下のようなアルゴリズムです：

- 下記の接尾辞をフォント名に付加し、できたファイル名のフォントメトリック（および TrueType・OpenType フォントの場合はアウトライン）があるかどうかを順番に調べてみます：

  ```
  .tte .ttf .otf .gai .afm .pfm .ttc
  .TTE .TTF .OTF .GAI .AFM .PFM .TTC
  ```

- PostScript フォントの埋め込みが要求されている場合は、下記の接尾辞をフォント名に付加し、その名前のフォントアウトラインファイルがあるかどうかを順番に調べてみます：

  ```
  .pfa .pfb
  .PFA .PFB
  ```

  フォントファイルが見つからなかったときは、フォント読み込みは下記のエラーメッセージを出して止まります：

  ```
  Font cannot be embedded (PFA or PFB font file not found)
  ```

- 上述の候補ファイル名群を「ありのままに」検索し、ついで、*SearchPath* リソースカテゴリ内で設定されているすべてのディレクトリ名を前につけて検索します。

これはすなわち、手作業で一切設定をしなくても、フォントの種類に従った標準的なファイル名接尾辞をフォント名に付加した名前をフォントファイルが持っており、かつ SearchPath ディレクトリのいずれかの中に置かれているならば、PDFlib はそのフォントを発見するということを意味します。

下記の 2 つのコードは、フォントアウトラインファイルを見つけるうえで同等の効力を持ちます：

```
p.set_parameter("FontOutline", "Arial=/usr/fonts/Arial.ttf");
font = p.load_font("Arial", "unicode", "");
```

と

```
p.set_parameter("SearchPath", "/usr/fonts");
font = p.load_font("Arial", "unicode", "");
```

標準日中韓フォント　Acrobat は、日中韓テキストのためのさまざまな標準フォントに対応しています。詳細とフォント名一覧は 169 ページ「6.5.1 標準日中韓フォント」を参照してください。PDFlib は標準日中韓フォントを、下記の要件が満たされるならば、フォント検索処理のいちばん最初の段階で発見します：

- 指定されたフォント名が標準日中韓フォントの名前と一致する。
- 指定されたエンコーディングが、定義済み CMap のいずれか 1 つの名前である。
- *embedding* オプションが指定されなかった。

これらの要件のいずれかに反するときは、フォント検索は継続されます。内部リストで見つかった標準日中韓フォントは決して埋め込まれません。これらのいずれかを埋め込むためには、フォントアウトラインファイルを設定する必要があります。例：

```
font = p.load_font("KozGoPro-Medium", "90msp-RKSJ-H", "");
```

Type 3 フォント　Type 3 フォントは、実行時に、標準 PDFlib グラフィック関数群でグリフを定義することによって定義する必要があります（117 ページ「5.1.5 Type 3 フォント」参照）。*PDF_begin_font()* に与えられたフォント名が、*PDF_load_font()* で要求されたフォント名と一致するときは、そのフォントがフォント検索の最初の段階で選ばれます（そのフォント名が標準日中韓フォントの名前と一致しなかったという前提で）。例：

```
p.begin_font("PDFlibLogoFont", 0.001, 0.0, 0.0, 0.001, 0.0, 0.0, "");

...
p.end_font();
...
font = p.load_font("PDFlibLogoFont", "logoencoding", "");
```

5.4.4 Windows・Mac OS X 上のホストフォント

Mac・Windows システムでは PDFlib は、オペレーティングシステムにインストールされている TrueType・OpenType・PostScript フォントを利用することができます。こうしたフォントを**ホストフォント**といいます。手作業でフォントファイルを設定しなくても、そのフォントを単純にシステムにインストール（たいていは、適切なディレクトリへそれをドロップすることによって）すれば、PDFlib はそれをうまく利用します。

ホストフォントを扱う際には、その正確な（大文字・小文字を区別した）フォント名を用いることが重要です。フォント名は重要ですので、フォント名決定のためのいくつかのプラットフォームごとの方式を以下に述べます。フォント名についてはさらに詳しい情報が 116 ページ「5.1.3 PostScript Type 1 フォント」にあります。

Windows 上のホストフォント名を知る　インストールされているフォントの名前は、そのフォントファイルをダブルクリックして、現れるウィンドウの 1 行目に表示される完全フォント名を見れば簡単に知ることができます。フォントによっては、使っている Windows のバージョンに従ってその名前の一部がローカライズされていることもあります。たとえば、フォント名の一部として広く使われている *Bold* は、ドイツ語システム上では翻訳された単語 *Fett* として表示されることがあります。Windows システムからホストフォントデータを取得するには、変換された形のフォント名（*Arial Fett* 等）を PDFlib で用いるか、あるいはフォントスタイル名（後述）を用いる必要があります。しかし、フォントデータをファイルから直接取得するには、正規の（ローカライズされていない）形のフォント名（*Arial Bold* 等）を用いる必要があります。

注 この国際化の問題は、ローカライズされた形のフォント名を用いるのでなく、フォントスタイル名（「*,Bold*」等、後述）を付加することによって回避することができます。

TrueType フォントをもっと詳しく調べたいときは、Microsoft の無償の「*Font properties extension*」[1] を見てみましょう。フォントの TrueType テーブルのさまざまな項目が、人間に読める形で表示されます。

Windows のフォントスタイル名　Windows オペレーティングシステムからホストフォントを読み込む際には、PDFlib ユーザーは、Windows のフォント選択機構が提供する機能を利用することができます：太さと斜体についてスタイル名を与えることができます。例：

```
font = p.load_font("Verdana,Bold", "unicode", "");
```

1. *www.microsoft.com/typography/TrueTypeProperty21.mspx* を参照。

これはWindowsに対して、ベースフォントのボールド・イタリック等ある特定のバリエーションを探すよう命令します。得られるフォントによって、Windowsは、求められたフォントに最も似通っているフォントを選びます（これは新たなフォントバリエーションを作り出します）。Windowsが見つけたフォントは、求めたフォントとは異なる可能性があり、生成されるPDF内のフォント名は、求めた名前とは異なる可能性があります。PDFlibは、Windowsのフォント選択に対していかなる制御もできません。フォントスタイル名はホストフォントでのみ働き、フォントファイルを通じて設定されたフォントに対しては働きません。

下記のキーワード（フォント名とカンマで区切って）は、ベースフォント名に付加してフォントの太さを指定することができます：

```
none, thin, extralight, ultralight, light, normal, regular, medium,
semibold, demibold, bold, extrabold, ultrabold, heavy, black
```

このキーワードは大文字・小文字を区別します。上記のかわりに、あるいは上記に加えて、*italic* キーワードを指定することもできます。2つのスタイル名を用いるときは、両者をカンマで区切る必要があります。例：

```
font = p.load_font("Verdana,Bold,Italic", "unicode", "");
```

フォントの太さの数値も、フォントスタイル名の等価な代用として用いることができます：

```
0 (none), 100 (thin), 200 (extralight), 300 (light), 400 (normal), 500 (medium), 600
(semibold), 700 (bold), 800 (extrabold), 900 (black)
```

下記の例はフォントのboldバリエーションを選びます：

```
font = p.load_font("Verdana,700", "unicode", "");
```

注 Windows のフォントに対するスタイル名は、ローカライズされたフォント名を扱う必要があるときには有用でしょう。なぜならこれは、フォントバリエーションのローカライズされた名前にかかわらずそれを指定するための汎用的な方式を提供するからです。

注 Windows のスタイル名の命名方式は、*fontstyle* オプションと混同してはいけません。両者は一見似ていますが、まったく異なる動作をするものです。

Windows上のホストフォント指定で起こりうる問題 Windows上のフォントインストールに関して起こりうる問題についてユーザーに注意を喚起します。フォントを**「ファイル」→「新しいフォントのインストール...」**メニュー項目でインストールする（フォントを *Fonts* ディレクトリへドラッグするのではなく）場合、チェックボックス**「フォントフォルダにフォントをコピーする」**があります。このボックスのチェックを外していると、Windowsは元のフォントファイルへのショートカット（リンク）をフォントフォルダ内に置くだけです。この場合は元のフォントファイルは、PDFlibを使っているアプリケーションから利用可能なディレクトリ内に置いてある必要があります。特に、Windowsの *Fonts* ディレクトリ外のフォントファイルはIISからはデフォルトのセキュリティ設定では利用できません。解決策：フォントファイルを *Fonts* ディレクトリへコピーするか、あるいは元のフォントファイルをIISが読み取り権限を持つディレクトリ内に置きます。

Adobe Type Manager（ATM）でフォントをインストールする際に**「ファイルをコピーせずに追加」**オプションをチェックしていた場合も、同様の問題が起こる可能性があります。

Mac 上のホストフォント名　Mac OS X に入っている *Font Book* ユーティリティを利用すれば、インストールされているホストフォントの名前を知ることができます。プログラム的にホストフォントのリストを作成するには、Apple の無償提供している Font Tools[1] を推奨します。このコマンドラインユーティリティのスイートには ***ftxinstalledfonts*** というプログラムが含まれており、これはインストールされているすべてのフォントの正確な名前を知るために有用です。PDFlib はホストフォントの名前としていくつかの種類に対応しています：

- QuickDraw フォント名：これは Mac OS 上で長い間使われてきた、しかしもう昔のものであると考えられている旧式のフォント名です。QuickDraw フォント名を知るには、ターミナルウィンドウで下記のコマンドを実行します：

```
ftxinstalledfonts -q
```

- 「一意」フォント名：これは新しいフォント名であり（Mac OS では新しい ATS フォント関数で使えます）、東アジアフォント等に対して Unicode でエンコードすることもできます。一意フォント名を知るには、ターミナルウィンドウで下記のコマンドを実行します（「:」を含む項目が出力に含まれることがありますが、これは除去する必要があります）：

```
ftxinstalledfonts -u
```

- PostScript フォント名。PostScript フォント名を知るには、ターミナルウィンドウで下記のコマンドを実行します：

```
ftxinstalledfonts -p
```

注 PDFlib の Leopard ビルド（Mac OS X 10.5 以上用）は、上記 3 種類のホストフォント名すべてに対応しています。非 Leopard ビルドでは QuickDraw フォント名しか使えません。

Mac 上でホストフォントを利用する際に起こりうる問題　私たちのテストによれば、新規にインストールされたフォントは、ユーザーがコンソールからログアウトして、再びログインするまで、PDFlib のような UI なしアプリケーションからは利用できないことがあります。

Mac OS X 10.5（Leopard）では、ホストフォントは、リモートコンピュータからのターミナルセッション内で動作しているプログラムからは利用できません。これは PDFlib の制約ではなく、Font Tools など他のプログラムにもあてはまります。この問題は Mac OS X 10.5.6 では修正されています。

5.4.5 予備フォント

クックブック 完全なコードサンプルがクックブックの text_output/starter_fallback トピックにあります。

予備フォントは、フォントとエンコーディングの不足な点を扱う強力なしくみを提供します。必要なフォント変更が PDFlib によって自動的に行われますので、これはさまざまな場面でテキスト出力の実現に活用することができます。このしくみは、所与のフォント（ベースフォントといいます）を、他の 1 つないし複数のフォント内のグリフをこのベースフォントに連結することによって強化するものです。より正確には：フォントは実際には変更されないのですが、PDFlib が PDF ページ記述内の必要なフォント変更をすべて自動的に行います。予備フォントは下記の機能を提供します：

1. *developer.apple.com/textfonts/download* を参照。

- ベースフォント内で得られないグリフは自動的に、1 つないし複数の予備フォント内で検索されます。言い換えれば、フォントにグリフを追加することが可能です。複数の予備フォントをベースフォントに対して紐付けることが可能ですので、少なくとも 1 つのフォントが適切なグリフを含んでいる Unicode キャラクタをすべて有効に使うことができます。
- ある特定の予備フォント内のグリフを用いて、ベースフォント内のグリフを上書きすることができます。すなわち、フォント内のグリフを置き換えることが可能です。1 つないし複数の個別のグリフを置き換えることもできますし、あるいは置き換えたい Unicode キャラクタ群の範囲を 1 つないし複数指定することもできます。

予備フォントからのグリフのサイズと縦位置は、ベースフォントに合うよう調整できます。ちょっと驚くことには、ベースフォントそれ自身も予備フォントとして使うことが可能です（同一の、または異なるエンコーディングで）。これを利用すると下記のトリックが実装できます：

- ベースフォントそれ自身を予備フォントとして使うことによって、フォント内のグリフ群の一部ないし全部のサイズまたは位置を調節することができます。
- ベースフォントの実際のエンコーディング外のキャラクタを追加できます。

予備フォント機構は、*fallbackfonts* フォント読み込みオプションによって司られ、すべてのテキスト出力関数に対して効力を持ちます。あらゆるフォント読み込みオプションと同様に、*fallbackfonts* オプションは *PDF_load_font()* への明示的な呼び出しで与えることもできますし、あるいは暗黙的フォント読み込みのためのオプションリスト内で与えることもできます。1 つのベースフォントに対しては複数の予備フォントを指定することも可能なことから、*fallbackfonts* オプションは値としてオプションリストのリストをとります（すなわち、中括弧がその分必要です）。

PDF_info_font() を利用すると、予備フォント機構の結果を取得することができます（Section 5.12.2, ?Querying Codepage Coverage and Fallback Fonts? 参照）。

注意　予備フォントを扱う際には下記に留意してください：

- フォントの組み合わせは必ずしも、タイポグラフィ的に美しい結果を生み出すわけではありません。ベースフォントのグリフデザインに整合するグリフデザインを持つ予備フォントだけを使うよう注意を払う必要があります。
- 予備フォントに対するフォント読み込みオプションは、*fallbackfonts* オプションリスト内で別途指定する必要があります。たとえば、ベースフォントに対して埋め込みを指定していても、予備フォントは自動的には埋め込まれません。
- 予備フォントは、そのフォントが正しい Unicode 情報を含んでいる場合にのみ動作します。置換グリフは、被置換グリフと同じ Unicode 値を持っている必要があります。
- 用字系固有のシェーピング（オプション *shaping・script・locale*）と OpenType 機能（オプション *features・script・language*）は、同一フォント内のグリフ群に対してのみ適用され、ベースフォントと 1 つないし複数の予備フォントとにわたるグリフ群に対しては適用されません。
- 下線 / 上線 / 取り消し線機能は、予備フォントを扱う際には注意して使う必要があります。アセンダ等のタイポグラフィ値についても同様です。ベースフォント内で決定される下線の太さ・位置は、予備フォント内の値とは一致しない可能性があります。その場合、下線の位置または太さが見苦しくガタつくことになります。この問題に対する単純な回避策は、統一的な値を、*PDF_fit_textline()・PDF_add/create_textflow()* の *underlineposition・underlinewidth* オプションで指定することです。この値は、ベースフォントとすべての予備フォントにおいてうまくいくように選ぶ必要があります。

以下の各項で、予備フォントの重要な用途をいくつか解説し、それを実現するオプションリストを示します。

テキストフォントに数学キャラクタを追加　数学グリフがないときの非常に荒っぽい解決法として、*fallbackfonts* オプションに対して下記のフォント読み込みオプションを用いて、Symbol フォント内の数学グリフをテキストフォントに追加することができます：

```
fallbackfonts={{fontname=Symbol encoding=unicode}}
```

複数の用字系で使えるようフォントを合体　場合によっては、入力テキストデータの用字系が事前にわからないことがあります。たとえば、データベースが欧文・ギリシャ文字・キリル文字のテキストを含んでいるのに、得られるフォントはこれらの用字系のうちの 1 つしか同時に網羅していないという場合があるかもしれません。用字系を決定して適切なフォントを選ぶのではなく、いくつかのフォントを結びつけたフォントを構築して、実質的にすべての用字系のスーパーセットを網羅することが可能です。*fallbackfonts* オプションに対して下記のフォント読み込みオプションを用いて、ギリシャ文字フォントとキリル文字フォントを欧文フォントに追加することができます：

```
fallbackfonts={
 {fontname=Times-Greek encoding=unicode embedding forcechars={U+0391-U+03F5}}
 {fontname=Times-Cyrillic encoding=unicode embedding forcechars={U+0401-U+0490}}
}
```

8 ビットエンコーディングを拡張　入力データがレガシ8ビットエンコーディングに限られていたとしても、このエンコーディング外のキャラクタを使うことができます。予備フォントを利用して（ここではベースフォントそれ自身を予備フォントとします）、かつ、PDFlib の文字参照のしくみを用いてエンコーディング外のキャラクタを指定すればよいのです。Helvetica フォントを encoding=iso8859-1（このエンコーディングはユーログリフを含んでいません）で読み込んだとすると、*fallbackfonts* オプションに対して下記のフォント読み込みオプションを用いて、ユーログリフをフォントに追加することができます：

```
fallbackfonts={{fontname=Helvetica encoding=unicode forcechars=euro}}
```

入力エンコーディングはユーロキャラクタを含んでいませんので、それを 8 ビット値で指定することはできません。この制約を回避するには文字参照かグリフ名参照（*€* 等）を用います（112 ページ「4.5.2 文字参照」参照）。

別フォントからのユーログリフを使う　上記とほとんど同じですが、ベースフォントがユーログリフを含んでいない場合を考えます。*fallbackfonts* オプションに対して下記のフォント読み込みオプションを用いて、ユーログリフを別のフォントから持って来ることができます：

```
fallbackfonts={{fontname=Helvetica encoding=unicode forcechars=euro textrise=-5%}}}
```

textrise サブオプションを用いて、ユーログリフを若干下へ下げました。

フォント内の一部ないし全部のグリフを大きくする　予備フォントを使うと、フォント内の一部ないしすべてのグリフを、文字サイズを変えずに大きくすることができます。この場合も、ベースフォントそれ自身を予備フォントとして用います。この機能は、さまざまなフォントのデザインを、コード内で文字サイズを調整せずに見た目上協調させるのに

有用です。*fallbackfonts* オプションに対して下記のフォント読み込みオプションを用いて、指定した範囲内のすべてのグリフを 120% へ大きくすることができます：

```
fallbackfonts={
 {fontname=Times-Italic encoding=unicode forcechars={U+0020-U+00FF} fontsize=120%}
}
```

拡大したピクトグラムを追加　*fallbackfonts* オプションに対して下記のフォント読み込みオプションを用いて、ZapfDingbats フォントから記号を持って来ることができます：

```
fallbackfonts={
 {fontname=ZapfDingbats encoding=unicode forcechars=.a12 fontsize=150% textrise=-15%}
}
```

この場合も、*fontsize・textrise* サブオプションを用いて、記号のサイズと位置をベースフォントに合わせています。

日中韓フォント内のグリフを置き換え　*fallbackfonts* オプションに対して下記のフォント読み込みオプションを用いて、ASCII 範囲内の欧文キャラクタを別フォントからのものに置き換えることができます：

```
fallbackfonts={
 {fontname=Courier-Bold encoding=unicode forcechars={U+0020-U+007E}}
}
```

アラビア文字フォントに欧文キャラクタを追加　この用途は 167 ページ「6.4.5 アラビア文字テキスト組版」で解説しています。

足りないグリフを知る　無償提供されているフォント *Unicode BMP Fallback SIL* は、実際のグリフでなく各 Unicode 値の 16 進値を表示します。このフォントは、ワークフローにおけるフォント関連の問題を診断するのに非常に有用なときがあります。*fallbackfonts* オプションに対して下記のフォント読み込みオプションを用いて、任意のフォントを、足りないキャラクタが可視化されるようこの特殊な予備フォントで強化することができます：

```
fallbackfonts={{fontname={Unicode BMP Fallback SIL} encoding=unicode}}
```

外字キャラクタをフォントに追加　この用途は 172 ページ「6.5.3 EUDC・SING フォントで外字キャラクタを利用」で解説しています。

5.5 フォントの埋め込み・サブセット化

5.5.1 フォントの埋め込み

Acrobat における PDF のフォント埋め込みとフォント置換　PDF 文書は、正しいテキスト表示を確保するために、フォントデータをさまざまな形式で含むことができます。あるいは、キャラクタのメトリックといくつかの一般的なフォント情報だけを含む（グリフのアウトライン本体を含まない）フォント記述子を埋め込むこともできます。フォントが PDF 文書に埋め込まれていない場合、Acrobat はそれがターゲットシステムで得られればそれを取り（**「ローカルフォントを用いる」**）、あるいはフォント記述子に従って代替フォントを組み立てようと試みます。代替フォントが使われることによってテキストは読めるようになりますが、そのグリフは元のフォントとは異なる可能性があります。より重要なことは、Acrobat の代替フォント（***AdobeSansMM*・*AdobeSerifMM***）は欧文テキストに対してのみ働き、それ以外の用字系や記号グリフに対しては一切働かないということです。同様に、代替フォントは、複雑用字系のシェーピングか OpenType レイアウト機能が使われているときには働きません。こうした理由から、一般にはフォントの埋め込みを推奨します。ただし、文書がフォントを埋め込んでいなくてもターゲットシステム上での表示が許容範囲内になるとわかっている場合は例外です。そのような PDF ファイルは本質的に非可搬ですが、すべてのワークステーション上で必要フォントが得られるとわかっている企業ネットワークなどの制御された環境においては役に立つかもしれません。

PDFlib でフォントを埋め込み　フォントの埋め込みは、フォントを読み込む際に ***embedding*** オプションで司られます（ただし場合によっては PDFlib はフォントを強制的に埋め込みます）：

```
font = p.load_font("WarnockPro", "winansi", "embedding");
```

表 5.1 に挙げるように、使用フォントごとに PDFlib が必要とするフォント・メトリックファイルは使用フォントごとに異なります。表 5.1 に挙げる要請に加えて、（標準またはカスタムの）日中韓フォントをいずれかの標準 CMap で使うには、その CMap ファイルが（場合によってはその文字集合に対する Adobe-Japan1-UCS2 等の Unicode マッピング CMap も）得られる必要があります。

不可視テキスト（主に OCR 出力に有用）にのみ使われるフォントに対するフォント埋め込みは、フォントを読み込む際に ***optimizeinvisible*** オプションで制御することができます。

表 5.1 さまざまなフォント使用状況と必要ファイル

使用フォント	フォントのメトリックファイルが必要か	フォントのアウトラインファイルが必要か
14 コアフォントのいずれか	×	embedding=true の場合のみ
Mac か Windows システムにインストールされている TrueType・OpenType・PostScript Type 1 ホストフォント	×	×
非コア PostScript フォント	○	embedding=true の場合のみ

表 5.1 さまざまなフォント使用状況と必要ファイル

使用フォント	フォントのメトリックファイルが必要か	フォントのアウトラインファイルが必要か
TrueType フォント	n/a	○
OpenType・SING フォント	n/a	○
標準日中韓フォント[1]	×	×

1. 日中韓フォントに関して詳しくは 169 ページ「6.5 日本語・中国語・韓国語テキスト出力」を参照。

フォント埋め込みの法的側面　留意しておかなければならない重要なことは、あるフォントファイルを持っているという理由だけでは、そのフォントを PDF に埋め込んでもよいという正当化はできないということです。たとえ合法なフォントライセンスを保持していても同様です。多くのフォントベンダーが、自社のフォントの埋め込みには制限を加えています。さまざまな書体工房のなかには、PDF のフォント埋め込みを完全に禁止しているところもありますし、自社のフォントに対する特別なオンラインライセンスや埋め込みライセンスを提示しているところもありますし、また、フォントにサブセット化を施す限りにおいてフォント埋め込みを許しているところもあります。フォントを PDFlib で埋め込んでみようと試みる前に、まず、そのフォントの埋め込みが法的にはどのようなことになるのか調べてください。TrueType フォントや OpenType フォントの中では埋め込み制限を指定しておくことができるので、PDFlib はその指定に従います。TrueType フォントの中の埋め込みフラグが“埋め込み不可”に設定されている場合[1]、PDFlib はフォントベンダーの要請に従い、そのフォントを埋め込もうとするあらゆる試みを拒否します。

5.5.2 フォントのサブセット化

PDF 出力のサイズを減らすために、PDFlib は、あるフォントの中で実際にその文書の中で使われているグリフだけを埋め込むことができます。この処理をフォントのサブセット化といいます。サブセット化を行うと新しいフォントが作成され、その中ではグリフの数が元のフォントよりも少なく、PDF の表示に必要ないフォント情報も省略されています。ただし、Acrobat の TouchUp テキストツールは、サブセットフォントのテキストでは動作してくれません。フォントのサブセット化は特に日中韓フォントにおいて重要です。PDFlib は次の種類のフォントのサブセット化に対応しています。

- TrueType フォント。
- PostScript か TrueType のアウトラインを持つ OpenType フォント。
- Type 3 フォント（特別な扱いが必要、141 ページ「Type 3 フォントのサブセット化」を参照）

サブセット化を要求されているフォントが文書内で使われている場合、PDFlib は実際にテキスト出力に使われているキャラクタを調べます。サブセット化の動作を制御するにはいくつかの方法があります（***autosubsetting*** は指定していないとして）：

- デフォルトのサブセット化の動作は ***autosubsetting*** パラメタで制御されます。もしこのパラメタが ***true*** ならば、サブセット化可能なすべてのフォントに対してサブセット化が有効になります（特別な扱いが必要な Type 3 フォントの場合を除きます、後述）。デフォルト値は ***true*** です。
- ***autosubsetting=true*** の場合：***subsetlimit*** パラメタはパーセント値を持ちます。文書内で用いられている、あるフォント内のグリフの数がこの割合を超える場合には、そのフォ

1. もっと明確に言えば、そのフォントの OS/2 テーブル内の fsType フラグが値 2 を持つ場合。

ントのサブセット化は無効となり、かわりにフォント全体が埋め込まれます。これによって処理時間がある程度短縮できますが、そのかわりに出力ファイルの容量は大きくなります。

```
p.set_value("subsetlimit", 75);  /* サブセットの限界を75%に設定 */
```

subsetlimit のデフォルト値は 100 パーセントです。言い換えれば、クライアントが明示的に 100 パーセント未満の限界値を要求しない限り、*PDF_load_font()* で要求されるサブセット化オプションは効力を持ちます。

- *autosubsetting=true* の場合：*subsetminsize* パラメタを用いると、容量の小さなフォントのサブセット化を完全に無効にすることができます。元のフォントファイルの容量が *subsetminsize* の値よりも KB 単位で小さい場合、そのフォントのサブセット化は無効になります。

TrueType フォントの埋め込みとサブセット化　ある TrueType フォントが、*winansi* か *macroman* 以外のエンコーディングで使用されるフォントである場合、デフォルトではそのフォントは PDF 出力の際に CID フォントに変換されます。Adobe グリフリスト（AGL）の中にあるキャラクタだけを含むエンコーディングの場合には、*autocidfont* パラメタを *false* に設定すればこの変換は行われません。

初期フォントサブセットを指定　フォントサブセットは、文書内で使われているすべてのグリフのアウトライン記述を含んでいます。これはすなわち、生成される文書サブセットは文書ごとに変化することを意味します。なぜなら一般に、各文書内ではそれぞれ異なるキャラクタ（ひいてはグリフ）のセットが用いられているからです。フォントサブセットを埋め込んだたくさんの小さな文書を大きな文書へ連結する際には、フォントサブセットがそれぞれまちまちであることは厄介です：埋め込まれているサブセットは全部互いに異なるため、除去できないのです。

こうした場合のために、PDFlib では、*PDF_load_font()* の *initialsubset* オプションでフォントサブセットの初期内容を指定することが可能です。PDFlib がデフォルトでは空のサブセットから開始し、生成されるテキスト出力からの要請に応じてグリフを追加していくのに対して、initialsubset オプションを利用すると、空でないサブセットを指定することができます。たとえば、Latin-1 テキスト出力のみが生成されるとわかっている場合に、フォントがその他のグリフもたくさん含んでいるならば、先頭 Unicode ブロックを初期サブセットとして指定することができます：

```
initialsubset={U+0020-U+00FF}
```

これはすなわち、指定した範囲内のすべての Unicode キャラクタに対するグリフがサブセット内に入れ込まれることを意味します。この範囲を、生成される文書内のすべてのテキストを網羅するように選んでおいたならば、生成されるフォントサブセットはすべての文書内で等しくなるでしょう。そうすれば、このような文書群をあとで 1 個の PDF へ連結する際に、等しいフォントサブセット群は *PDF_begin_document()* の *optimize* オプションで除去することが可能になります。

Type 3 フォントのサブセット化　Type 3 フォントを文書で使えるようにするには、その前にまず（グリフの幅が必要なので）それを定義しなければならず、ひいては埋め込まなければなりません。ところがその一方でサブセット化は、すべてのページを作成した後ではじめて可能になるものです（正しいサブセットを決定するには、どのグリフが文書内で使われたかを知る必要があるので）。この矛盾を避けるために PDFlib は、幅オンリー Type

3 フォントに対応しています。Type 3 フォントについてサブセット化が必要なときは、以下の 2 回に分けてフォントを定義する必要があります。

- 1 回目は、フォントを使う前に、*PDF_begin_font()* で *widthsonly* オプションを指定して行う必要があります。ここではフォントとグリフのメトリック（幅）だけを定義します。*PDF_begin_font()* のフォントマトリックスと、*PDF_begin_glyph()* の *wx* とグリフ外接枠を与える必要があり、かつ実際のグリフのメトリックを正確に記述する必要があります。グリフごとに *PDF_begin_glyph()* と *PDF_end_glyph()* だけが必要であり、それ以外に実際のグリフの形を定義する呼び出しは一切不要です。グリフ定義の開始と終了の間で他の関数を呼び出した場合、それは PDF 出力に対して何ら効力を持たず、例外も一切発生しません。
- 2 回目は、このフォントのテキストをすべて作成した後に行う必要があり、実際のグリフのアウトラインかビットマップを定義します。フォントとグリフのメトリックは、1 回目ですでにわかっているので無視されます。最後のページが作成された後、PDFlib はどのグリフが文書内で使われているかも知っているので、必要なグリフ定義だけを埋め込んでフォントサブセットを構成します。

1 回目と 2 回目では、同じグリフ群を与えなければなりません。サブセット化を伴う Type 3 フォントは、*PDF_load_font()* で 1 回だけ読み込むことができます。

クックブック 完全なコードサンプルがクックブックの `fonts/type3_subsetting` **トピックにあります。**

5.6 フォント情報を取得

PDF_info_font() を利用すると、フォント・エンコーディング・Unicode・グリフに関して有用な情報を取得することができます。取得したい情報の種類によっては、有効なフォントハンドルがこの関数の引数として必要な場合もあります。以下すべての例で、表 5.2 に挙げる変数を用いることにします。

表 5.2 PDF_info_font( ) の利用例で用いる変数一覧

変数	注釈
int uv;	Unicode の数値。あるいは、グリフ名参照から「&」・「;」修飾を除いたものをオプションリスト内で用いることもできます（unicode=euro 等）。詳しくは、PDFlib API リファレンスの Unichar オプションリストデータ型の解説を参照してください。
int c;	8 ビット文字コード
int gid;	グリフ ID
int cid;	CID 値
String gn;	グリフ名
int gn_idx;	グリフ名の文字列番号。gn_idx が −1 以外ならば、対応する文字列を下記のように取得できます： gn = p.get_parameter("string", gn_idx);
String enc;	エンコーディング名
int font;	PDF_load_font( ) で取得した有効なフォントハンドル

求められたキーワードとオプション（群）の組み合わせが得られないときは、*PDF_info_font()* は −1 を返します。これは、クライアントアプリケーション側でチェックする必要があり、またこれを用いて、求めるグリフがフォント内にあるかどうかをチェックすることができます。

以下のサンプルコード行は、互いに依存していませんので、独立に抜き出して利用することができます。

5.6.1 フォント非依存のエンコーディング・Unicode・グリフ名取得

エンコーディング取得　エンコーディングの取得には、有効なフォントハンドルは必要ではありません。すなわち、*PDF_info_font()* の *font* 引数に値 −1（PHP では 0）を与えることができます。*gn* には、PDFlib が内部的に知っているグリフ名だけを与えることができ、フォント独自のグリフ名を与えることはできません。

Unicode キャラクタ、指名したグリフの 8 ビットエンコーディング内の 8 ビットコードを取得：

```
c = (int) p.info_font(-1, "code", "unicode=" + uv + " encoding=" + enc);
c = (int) p.info_font(-1, "code", "glyphname=" + gn + " encoding=" + enc);
```

8 ビットコード、指名したグリフの 8 ビットエンコーディング内の Unicode 値を取得：

```
uv = (int) p.info_font(-1, "unicode", "code=" + c + " encoding=" + enc);
uv = (int) p.info_font(-1, "unicode", "glyphname=" + gn + " encoding=" + enc);
```

8 ビットコード、Unicode 値の 8 ビットエンコーディング内の登録されたグリフ名を取得：

```
gn_idx = (int) p.info_font(-1, "glyphname", "code=" + c + " encoding=" + enc);
gn_idx = (int) p.info_font(-1, "glyphname", "unicode=" + uv + " encoding=" + enc);

/* 文字列番号を用いて実際のグリフ名を取得 */
gn = p.get_parameter("string", gn_idx);
```

Unicode・グリフ名取得　*PDF_info_font()* を利用すると、特定の 8 ビットエンコーディングに依存せず、Unicode 値と PDFlib が内部的に知っている名前との関係にかかわる情報取得を行うことも可能です。これらの情報取得はいかなるフォントにも依存しませんので、有効なフォントハンドルは必要ではありません。

内部的に知られているグリフ名の Unicode 値を取得：

```
uv = (int) p.info_font(-1, "unicode", "glyphname=" + gn + " encoding=unicode");
```

Unicode 値の内部グリフ名を取得：

```
gn_idx = (int) p.info_font(-1, "glyphname", "unicode=" + uv + " encoding=unicode");

/* 文字列番号を用いて実際のグリフ名を取得 */
gn = p.get_parameter("string", gn_idx);
```

5.6.2 フォント依存のエンコーディング・Unicode・グリフ名取得

以下の情報取得は、特定のフォントにかかわるものですので、有効なフォントハンドルでフォントを指定する必要があります。*gn* 変数を用いて、内部的に知られているグリフだけでなく、フォント独自のグリフ名を与えることもできます。以下すべての例において、戻り値 -1 は、求めたグリフをそのフォントが含んでいないことを意味します。

8 ビットエンコーディングで読み込んだフォント内の Unicode 値、グリフ ID、指名したグリフ、CID に対する 8 ビットコードを取得：

```
c = (int) p.info_font(font, "code", "unicode=" + uv);
c = (int) p.info_font(font, "code", "glyphid=" + gid);
c = (int) p.info_font(font, "code", "glyphname=" + gn);
c = (int) p.info_font(font, "code", "cid=" + cid);
```

フォント内のコード、グリフ ID、指名したグリフ、CID に対する Unicode 値を取得：

```
uv = (int) p.info_font(font, "unicode", "code=" + c);
uv = (int) p.info_font(font, "unicode", "glyphid=" + gid);
uv = (int) p.info_font(font, "unicode", "glyphname=" + gn);
uv = (int) p.info_font(font, "unicode", "cid=" + cid);
```

フォント内のコード、Unicode 値、指名したグリフ、CID に対するグリフ ID を取得：

```
gid = (int) p.info_font(font, "glyphid", "code=" + c);
gid = (int) p.info_font(font, "glyphid", "unicode=" + uv);
gid = (int) p.info_font(font, "glyphid", "glyphname=" + gn);
gid = (int) p.info_font(font, "glyphid", "cid=" + cid);
```

任意の 8 ビットエンコーディングにおけるフォント内のコード、Unicode 値、指名したグリフに対するグリフ ID を取得：

```
gid = (int) p.info_font(font, "glyphid", "code=" + c + " encoding" + enc);
gid = (int) p.info_font(font, "glyphid", "unicode=" + uv + " encoding=" + enc);
gid = (int) p.info_font(font, "glyphid", "glyphname=" + gn + " encoding=" + enc);
```

コード・Unicode 値・CID で指定したグリフのフォント独自の名前を取得：

```
gn_idx = (int) p.info_font(font, "glyphname", "code=" + c);
gn_idx = (int) p.info_font(font, "glyphname", "unicode=" + uv);
gn_idx = (int) p.info_font(font, "glyphname", "glyphid=" + gid);
gn_idx = (int) p.info_font(font, "glyphname", "cid=" + cid);

/* 文字列番号を用いて実際のグリフ名を取得 */
gn = p.get_parameter("string", gn_idx);
```

グリフ互換性を調べる　*PDF_info_font()* を利用すると、自分のアプリケーションで必要なグリフをある特定のフォントが含んでいるかどうかを調べることができます。たとえば、下記のコードはユーログリフがフォント内に含まれているかどうかを調べます：

```
/* "unicode=U+20AC"でもよい */
if (p.info_font(font, "code", "unicode=euro") == -1)
{
        /* ユーログリフに対するグリフがフォント内で得られない */
}
```

クックブック　**完全なコードサンプルがクックブックの** fonts/glyph_availability **トピックにあります。**

あるいは *PDF_info_textline()* を使って、所与のテキスト文字列におけるマップなしキャラクタの数を、すなわち文字列内の、フォント内で適当なグリフが得られないキャラクタの数を調べることもできます。下記のコードでは、ユーロキャラクタ（グリフ名参照で表現）1 個だけを内容とする文字列についての結果を取得しています。もしもマップなしキャラクタが 1 個見つかれば、これはすなわちそのフォントがユーロ記号に対するグリフを一切含んでいないことを意味します：

```
String optlist = "font=" + font + " charref";

if (p.info_textline("&euro;", "unmappedchars", optlist) == 1)
{
        /* ユーロ記号に対するグリフはフォント内で得られない */
}
```

5.6.3 コードページ網羅性と予備フォントを取得

PDF_info_font() を利用すると、ある特定の言語ないし用字系のテキスト出力を生成するのにフォントが適しているかどうかを調べることもできます。そのためには、そのテキストでどのコードページが必要かがわかっている必要があります。コードページ網羅性は、フォントの OS/2 テーブル内にエンコードされています。ただし、フォントがある特定のコードページに対応しているとは正確には何を意味するのかは、フォントデザイナーの考え方一つで決まります。フォントがある特定のコードページに対応していると言っているからといって、必ずしもそれがそのコードページ内のすべてのキャラクタに対するグリフを含んでいるとは限りません。より正確な網羅性情報が必要な場合は、144 ページ「5.6.2 フォント依存のエンコーディング・Unicode・グリフ名取得」で示したようにしてすべての必要なキャラクタの入手可能性を取得することができます。

フォントがコードページに対応しているかどうかを調べる　下記のコードでは、フォントがある特定のコードページに対応しているかどうかを調べています：

```
String cp="cp1254";
```

```
result = (int) p.info_font(font, "codepage", "name=" + cp);

if (result == -1)
        System.err.println("コードページ網羅性不明");
else if (result == 0)
        System.err.println("コードページはこのフォントでは対応していません");
else
        System.err.println("コードページはこのフォントで対応しています");
```

対応している全コードページのリストを取得　下記のコードでは、TrueType　または OpenType フォントが対応しているすべてのコードページのリストを取得しています：

```
cp_idx = (int) p.info_font(font, "codepagelist", "");

if (cp_idx == -1)
        System.err.println("コードページリスト不明");
else
{
        System.err.println("コードページリスト:");
        System.err.println(p.get_parameter("string", cp_idx));
}
```

これは、広く用いられている Arial フォントに対しては下記のリストを生成します：

```
cp1252 cp1250 cp1251 cp1253 cp1254 cp1255 cp1256 cp1257 cp1258 cp874 cp932 cp936 cp949
cp950 cp1361
```

予備フォントを取得　*PDF_info_font()* を利用すると、予備フォント機構の結果を取得することができます（予備フォントについて詳しくは 135 ページ「5.4.5 予備フォント」を参照）。下記のコードでは、指定した Unicode キャラクタを表すのに用いられているベースフォントか予備フォントの名前を調べています：

```
result = p.info_font(basefont, "fallbackfont", "unicode=U+03A3");
/* result==ベースフォントならば、ベースフォントが使われ予備フォントは必要なかった */
if (result == -1)
{
        /* キャラクタはベースフォントでも予備フォントでも表示できない */
}
else
{
        idx = p.info_font(result, "fontname", "api");
        fontname = p.get_parameter("string", idx);
}
```

6 テキスト出力

6.1 さまざまなテキスト出力方式

PDFlib は、テキスト出力にいくつかのレベルで対応しています：

- *PDF_show( )* や類似の関数群による低レベルテキスト出力。
- *PDF_fit_textline( )* による一行に組まれたテキスト出力。この関数はパス上のテキストにも対応しています。
- テキストフローによる複数行テキスト組版出力（*PDF_fit_textflow( )* および関連する関数群）。テキストフロー組版機能は、ベクトルベースの形状の内側または外側にテキストを回りこませることもできます。
- 表内のテキスト。表組版機能は、表セル内のテキスト行・テキストフロー内容に対応しています。

低レベルテキスト出力　*PDF_show( )* のような関数群を使うと、テキストをページ上のある特定の場所に、いかなる組版支援をも利用せずに配置することができます。これは、非常に基本的な出力要請を持つアプリケーション（プレーンテキストファイルを PDF へ変換する等）、あるいはすでに完全なテキスト配置情報を持っているアプリケーションの場合にのみ推奨します（別形式のページを PDF へ変換するドライバ等）。

下記のコードは低レベル関数群でテキスト出力を生成します：

```
font = p.load_font("Helvetica", "unicode", "");

p.setfont(font, 12);
p.set_text_pos(50, 700);
p.show("Hello world!");
p.continue_text("(says Java)");
```

テキスト行で組まれた一行テキスト出力　*PDF_fit_textline( )* は、一行だけのテキスト出力を生成し、さまざまな組版機能を提供します。ただし、テキスト行ごとの位置はクライアントアプリケーションが決定する必要があります。

下記のコードはテキスト行でテキスト出力を生成します。フォント・エンコーディング・文字サイズはオプションとして指定できますので、これに先立って *PDF_load_font( )* を呼び出しておく必要はありません：

```
p.fit_textline(text, x, y, "fontname=Helvetica encoding=unicode fontsize=12");
```

テキスト行について詳しくは 197 ページ「8.1 テキスト行の配置とはめ込み」を参照してください。

テキストフローによる複数行テキスト出力　*PDF_fit_textflow( )* は、任意の行数のテキスト出力を生成し、また、テキストを複数の段組みまたはページにわたらせることもできます。テキストフロー組版機能はたくさんの組版機能に対応しています。

下記のコードはテキストフローでテキスト出力を生成します：

```
tf = p.add_textflow(tf, text, optlist);
result = p.fit_textflow(tf, llx, lly, urx, ury, optlist);
p.delete_textflow(tf);
```

テキストフローについて詳しくは 205 ページ「8.2 複数行のテキストフロー」を参照してください。

表内のテキスト　テキスト行とテキストフローを使って、表セル内にテキストを配置することもできます。表機能について詳しくは Section 8.3, ?Table Formatting?, page 209 を参照してください。

6.2 フォントメトリックとテキストバリエーション

6.2.1 フォントとグリフのメトリック

テキスト位置　PDFlib はテキスト位置を、グラフィック描画のカレント点とは独立に保持します。前者は *textx/texty* パラメタで取得でき、後者は *currentx/currenty* で取得できます。

グリフのメトリック　PDFlib では、PostScript や PDF で用いられているグリフとフォントのメトリックの体系を用いています。ここで簡単に説明しておきましょう。

PDFlib のユーザーが指定する必要のある文字サイズというのは、テキストの行と行の間で文字が重なりあわないために必要な最小間隔のことです。文字サイズは一般にフォント内の各文字よりも大きくなっています。なぜならその中にはベースラインより上の部分も下の部分も含んでいるからであり、また、それに加えて行と行の間の間隔をもっと広くとっていることもあるからです。

leading（行送り）は、テキストの 1 つの行のベースラインと次の行のベースラインとの間の縦の間隔を指定します。デフォルトではこれは文字サイズと同じ値に設定されています。*capheight*（キャップハイト）は、多くの欧文フォントでは *T* や *H* のような大文字の高さです。*xheight*（x ハイト）は、多くの欧文フォントでは *x* のような小文字の高さです。*ascender*（アセンダ）は、多くの欧文フォントでは *f* や *d* のような小文字の高さです。*descender*（ディセンダ）は、多くの欧文フォントでは、ベースラインから *j* や *p* のような小文字の下端までの間隔です。ディセンダはふつう負の値です。*xheight*・*capheight*・*ascender*・*descender* の値は文字サイズに対する割合として表されているので、必要な文字サイズを掛けてから用いる必要があります。

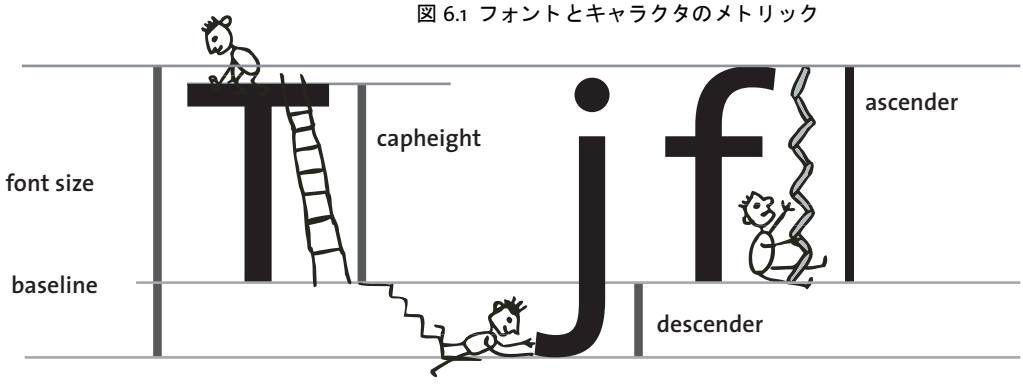
gaplen プロパティは TrueType・OpenType フォントでのみ得られます（それ以外のフォントでは算出されます）。*gaplen* 値はフォントファイルから読み出され、ベースライン間の推奨間隔とアセンダ＋ディセンダとの差を示します。

PDFlib は、これらの値のうちの 1 つないし複数を、推算で求めなければならない場合もあります。なぜならこれらの値は、フォントやメトリックファイルの中に存在しているという保証がないためです。用いられている値が本当の値なのか、それとも推測値なのかを知るには、*PDF_info_font()* を呼び出してオプション *faked* で *xheight* を取得します。あるフォントのキャラクタメトリックを PDFlib で取得するには下記のように記述します：

```
font = p.load_font("Times-Roman", "unicode", "");

capheight = p.info_font(font, "capheight", "");
```

図 6.1 フォントとキャラクタのメトリック



```
ascender = p.info_font(font, "ascender", "");
descender = p.info_font(font, "descender", "");
xheight = p.info_font(font, "xheight", "");
```

注 上付き・下付きの位置とサイズは PDFlib では取得できません。

クックブック 完全なコードサンプルがクックブックの `fonts/font_metrics_info` トピックにあります。

CPI の計算　多くのフォントは可変の字幅を持っていますが、等幅フォントという種類のフォントではすべての文字に対して等しい幅を用います。PDF のフォントメトリックと、高速印刷環境でよく用いられる characters per inch（CPI）表記との関係を理解するには、等幅の Courier フォントでの計算例が役に立つのではないでしょうか。Courier では、文字矩形の全幅がポイントあたり 1000 単位であるのに対して、すべての文字は 600 単位の幅を持っています(この値はそれぞれのAFMメトリックファイルから得ることができます)。たとえば 12 ポイントのテキストでは、すべての文字の実際の幅は次のようになります。

```
12 ポイント × (600 ÷ 1000) = 7.2 ポイント
```

最適行送りが 12 ポイントです。1 インチは 72 ポイントですので、Courier 12 ポイントの文字は 1 インチの中にちょうど 10 個収まることになります。つまり、12 ポイントの Courier は 10 cpi フォントであるということになります。10 ポイントのテキストの場合は、字幅は 6 ポイントですので、すなわち 72 ÷ 6 = 12 cpi フォントとなります。同様に、8 ポイントの Courier は 15 cpi となります。

6.2.2 カーニング

さまざまな文字の組み合わせのなかには、望ましくない見ばえになってしまうものがあります。たとえば、2 つの V が隣り合うと W のように見えてしまいますし、T と e の間の間隔は縮めないと広くあきすぎて不恰好になってしまいます。このような補正のことをカーニングといいます。多くのフォントが、問題となる文字の組み合わせそれぞれに対する間隔調整を指定した包括的なカーニング情報を持っています。PDFlib は、下記の情報源からのカーニングデータを使用します：

- TrueType・OpenType フォント：***kern*** テーブルで指定されたカーニング対。
- OpenTypeフォント：***kern***機能と***GPOS***テーブルを通じて指定された対ベースと分類ベースのカーニングデータ。
- PostScript Type 1 フォント：AFM・PFM ファイルで指定されたカーニング対。
- PDF コアフォントに対するカーニング対は PDF によって内部的に提供されます。

PDFlib には、カーニングの動作を制御する方式が次の 2 種類あります：

- デフォルトでは、フォント内のカーニング情報はそのフォントを読み込む際に読み取られます。カーニングが必要でない場合は、***PDF_load_font()*** で ***readkerning*** オプションを ***false*** に設定します。
- テキスト出力に対するカーニングは、テキスト出力関数群が対応している ***kerning*** テキスト書式オプションで有効にする必要があります。

一時的にカーニングを無効にすることがたとえばどんなときに有用かというと、数表を組みたいときに、カーニングデータが数字どうしの組み合わせを含んでいる場合というのが挙げられます。カーニングされた数字は表内できれいに並ばないからです。なお、今どきの TrueType・OpenType フォントはこの目的のための特殊な数字を含んでおり、これは**等幅数字**レイアウト機能とオプション ***features={tnum}*** で利用できます。

カーニングは、どのような字間・単語間隔・横倍率が指定されていたとしても、それに加えて適用することができます。PDFlib では、1 つのフォント内のカーニング対の数についてはまったく無制限です。

6.2.3 テキストバリエーション

擬似フォントスタイル　フォントのボールド体やイタリック体は通常、適切なフォントを選ぶことによって印字するべきものです。それ以外の方法として、PDFlib は擬似フォントスタイルにも対応しています。これは、レギュラーフォントに基づいて、Acrobat が、ベースフォントを太くしたり斜体にしたりすることによってボールド体やイタリック体やボールドイタリック体を擬似的に実現するものです。擬似フォントスタイルの美的な質は、フォントデザイナーが作り込んだ本当のボールドフォントやイタリックフォントにかなうものではありません。しかし、あるフォントスタイルを直接利用することができない場面においては、擬似スタイルがその代用として活用できます。とりわけ擬似フォントスタイルは標準日中韓フォントに対して有用です。標準日中韓フォントにはノーマルフォントしかなく、ボールド体やイタリック体が一切ないからです。

注 フォントスタイル機能を標準日中韓フォント以外のフォントに対して用いることは推奨しません。また、フォントスタイル機能は Adobe Acrobat 以外の PDF ビューアでは動作しないことにも留意してください。

Adobe Acrobat が持つ制約のため、擬似フォントスタイルは、以下のすべての条件がみたされた時にのみ動作します。

- ベースフォントが TrueType フォントか OpenType フォントであること。標準またはカスタムの日中韓フォントを含みます。ベースフォントが PDF コアフォントのうちのいずれかであってはなりません（131 ページ「欧文コアフォント」参照）。TrueType Collections（TTC）にはフォントスタイルは適用できません。
- エンコーディングが ***winansi*** であるか、***macroman*** であるか、または、表 4.3 に挙げる定義済み日中韓 CMap のうちのいずれかであること（なぜならそうでない場合 PDFlib は強制的にフォントを埋め込むので）。
- ***embedding*** オプションは ***false*** に設定している必要があります。

図 6.2 カーニング

- PDF の表示される相手先のシステムにベースフォントがインストールされている必要があります。

PDFlib は最初の 3 つの条件はチェックしますが、最後の条件をみたすことはユーザー側の役割となります。

擬似フォントスタイルを使うには、次のように、*PDF_load_font()* の *fontstyle* オプションに *normal*（ベースフォントそのまま）・*bold*・*italic*・*bolditalic* のいずれかのキーワードを用います。

```
font = p.load_font("HeiseiKakuGo-W5", "UniJIS-UCS2-H", "fontstyle bold");
```

fontstyle 機能は、Windows のフォントスタイル名と一見似た概念ですが、混同してはいけません。フォントスタイルが上記の条件のもとでのみ動作し、擬似フォントスタイルの実現は Acrobat に依存しているのに対して、Windows のスタイル名は完全に Windows のフォント選択エンジンに基づいており、存在しないスタイルをこれで実現することはできません。

クックブック **完全なコードサンプルがクックブックの `fonts/artificial_fontstyles` トピックにあります。**

擬似ボールドフォント *fontstyle* 機能がフォントに作用するのに対して、PDFlib は、個別のテキスト文字列を擬似的なボールドテキストにする代用機能にも対応しています。これは *fakebold* パラメタかオプションで制御されます。

クックブック **完全なコードサンプルがクックブックの `fonts/simulated_fontstyles` トピックにあります。**

擬似斜体フォント *fontstyle* 機能のかわりに *italicangle* パラメタまたはオプションを使って、レギュラーフォントしか利用できないときにイタリックフォントのような効果を出すこともできます。この方式は偽イタリックフォントを作り出すためにレギュラーフォントをユーザーから与えられた角度だけ傾けるものであり、フォントスタイルに関する上述の制約には縛られません。負の値でテキストは右に傾きます。もちろん、本物のイタリックや斜体フォントを使ったほうがはるかにきれいな出力が得られることを忘れてはいけません。しかしイタリックフォントが入手できないときに *italicangle* パラメタかオプションを使えば、簡単にそれに似た効果を出すことができます。この機能は特に日中韓フォントで便利です。*italicangle* パラメタかオプションの値は普通 −12 から −15 度の範囲です。

注 ***italicangle* パラメタ・オプションは縦書きでは使えません。**

注 **PDFlib はグリフ幅を、斜体化したグリフの新しい外接枠には合わせません。たとえばテキストを均等配置するとき、斜体化したグリフははめこみ枠からはみ出す可能性があります。**

影付きテキスト PDFlib は、同じテキストを場所を少しずつずらして複数個印字することによって影付き効果を生み出すことができます。影付きテキストは *PDF_fit_textline()* の *shadow* オプションで生成できます。影の色や、その主テキストに対する相対位置、グラフィック状態パラメタ群は、サブオプションで指定できます。

テキストの下線・上線・取り消し線 PDFlib では、テキストの下や上や中央に線をひくことができます。線幅やベースラインからの間隔は、フォントのメトリック情報に基づいて計算されます。また、横倍率やテキストマトリックスのカレント値も線幅の計算には加

味されます。下線・上線・取り消し線の有無を切り替えるには、それぞれ *PDF_set_parameter()* で *underline*・*overline*・*strikeout* パラメタを、またはテキスト出力関数群で同名のオプションをオンオフします。*underlineposition*・*underlinewidth* パラメタかオプションを使って微調整もできます。

線の色にはカレント描線色が用いられます。しかし、カレントの linecap・dash パラメタは無視されます。体裁上の注意：多くのフォントでは、下線は文字のベースラインより下の部分とぶつかってしまい、また、上線は文字の上の付加記号とぶつかってしまいます。

クックブック **完全なコードサンプルがクックブックの** `text_output/starter_textline` **トピックにあります。**

テキスト表現モード PDFlib は、テキストの見栄えを変更する表現モードにいくつか対応しています。これを用いると、テキストの輪郭を描いたり、テキストをクリッピングパスとして利用したりすることができます。また、テキストを不可視にすることもでき、これはたとえば、スキャン画像の上にテキストを乗せてテキスト検索や索引生成を可能にしつつ、テキスト自体は隠す、といった活用ができるでしょう。表現モードの一覧は ***PDFlib API リファレンス***に挙げてあります。表現モードは、*textrendering* パラメタかオプションで設定することができます。

テキストを描線すると、線幅や色といったグラフィック状態のパラメタ群がグリフの輪郭に適用されます。表現モードは、Type 3 フォントを用いて印字されるテキストには何の効果も持ちません。

クックブック **完全なコードサンプルがクックブックの** `text_output/text_as_clipping_path`・`text_output/invisible_text` **トピックにあります。**

テキストの色 テキストは通常、カレント塗り色で印字されます。塗り色は *PDF_setcolor()* を用いて設定できます。ただし、表現モードで 0 以外が選択されている時は、その選択されている表現モードによって、描線色と塗り色はどちらもテキストに対して効力を持つ可能性があります。

クックブック **完全なコードサンプルがクックブックの** `text_output/starter_textline` **トピックにあります。**

6.3 OpenType レイアウト機能

クックブック 完全なコードサンプルがクックブックの text_output/starter_opentype トピックにあります。

6.3.1 対応している OpenType レイアウト機能

PDFlib は、いくつかのフォント内の追加情報に従った高度なテキスト出力に対応しています。これらのフォント拡張を OpenType レイアウト機能といいます。たとえば、フォントが *liga* 機能を含んでいるかもしれません。この機能は、*f・f・i* グリフは結合して合字を形作れるという情報を含んでいます。他によく利用される例としては、*smcp* 機能によるスモールキャピタル、すなわち通常の大文字キャラクタよりも小さな大文字キャラクタや、*onum* 機能によるオールドスタイル数字、すなわちアセンダとディセンダを持つ数字（ベースライン上にすべて配置される横並びの数字とは異なる）などが挙げられます。合字は非常に広く利用される OpenType 機能ではありますが、可能な何ダースもの機能のうちの 1 つにすぎません。OpenType 形式と OpenType 機能テーブルに関するあらましは下記にあります：

```
www.microsoft.com/typography/developers/opentype/default.htm
```

PDFlib は下記のグループの OpenType 機能に対応しています：

- 表 6.1 に挙げる欧文タイポグラフィのための OpenType 機能群。これらは *features* オプションで司られます。
- 表 6.7 に挙げる日本語・中国語・韓国語テキストのための OpenType 機能群。これらも *features* オプションで司られますが、詳しくは 173 ページ「6.5.4 OpenType レイアウト機能と高度な日中韓テキスト出力」で解説します。
- 複雑用字系のシェーピングと縦書きテキスト出力のための OpenType 機能群。これらは *shaping*・*script* オプションに従って自動的に評価されます（161 ページ「6.4 複雑用字系出力」を参照）。この *vert* 機能は *vertical* フォントオプションで司られます。
- カーニングのための OpenType 機能テーブル群。ただし、PDFlib は *kerning* を OpenType 機能としては扱いません。なぜならカーニングデータは OpenType 機能テーブル以外の手段でも表現することができるからです。カーニングを制御するのではなく、*readkerning* フォントオプションと *kerning* テキストオプションを用いてください（150 ページ「6.2.2 カーニング」を参照）。

OpenType レイアウト機能について詳しい解説は下記にあります：

```
www.microsoft.com/typography/otspec/featuretags.htm
```

OpenType 機能を見つける　OpenType 機能テーブルを見つけるには下記のツールが使えます：

- FontLab フォントエディタはフォントを作成・編集するためのアプリケーションです。その無償デモ版（*www.fontlab.com*）は OpenType 機能を表示・プレビューします。
- DTL OTMaster Light（*www.fonttools.org*）はフォントを表示・分析するための無償アプリケーションであり、OpenType 機能テーブルも表示・分析できます。
- Microsoft の無償の「font properties extension」[1]は、フォント内で得られる OpenType 機能のリストを表示します（図 6.3 参照）。

1. *www.microsoft.com/typography/TrueTypeProperty21.mspx* を参照。

- PDFlib の ***PDF_info_font()*** インタフェースを使って、対応している OpenType 機能を取得することもできます（160 ページ「OpenType 機能をプログラム的に取得」を参照）。

表 6.1 欧文タイポグラフィのための対応 OpenType 機能一覧（日中韓テキストのための OpenType 機能は表 6.7 に挙げます）

キーワード	名前	説明
_none	全機能無効	表 6.1・表 6.7 に挙げるすべての OpenType 機能を無効に。
afrc	別形式分数	スラッシュで区切られた数字群を別形式へ置き換え。
c2pc	大文字からのプティットキャピタル	大文字キャラクタをプティットキャピタルに変えます。
c2sc	大文字からのスモールキャピタル	大文字キャラクタをスモールキャピタルに変えます。
case	ケースセンシティブ字体	さまざまな句読点を上へずらして、全大文字シーケンスや横並び数字のセットとの調和を高めます。また、オールドスタイル数字を横並び数字に変えます。
dlig	随意合字	複数グリフのシーケンスを、タイポグラフィ的観点からより好ましい 1 個のグリフへ置き換え
dnom	分母	スラッシュの直後の数字を分母数字へ置き換え
frac	分数	スラッシュで区切られた数字群を、「通用の」（斜め線による）分数へ置き換え
hist	歴史的字体	デフォルトの（現行の）字体を歴史的字体へ置き換え。いくつかの字の字体は過去には広く用いられていましたが、現在では時代遅れに見えます。
hlig	歴史的合字	この機能はデフォルトの（現行の）合字を歴史的合字へ置き換えます。
liga	標準合字	複数グリフのシーケンスを、タイポグラフィ的観点からより好ましい 1 個のグリフへ置き換え
lnum	横並び数字	数字をオールドスタイルからデフォルトの横並び字体へ変えます。
locl	ローカライズ字体	グリフのデフォルト字体からローカライズ字体への置換を有効にします。この機能は script・language オプションを必要とします。
mgrk	数学ギリシャ文字	ギリシャ文字グリフの標準タイポグラフィ字体を、広く用いられている数学的表記へ置き換え。
numr	分子	スウォッシュの直前の数字を分子数字へ置き換え、タイポグラフィスラッシュを分数スラッシュへ置き換え。
onum	オールドスタイル数字	数字を、デフォルトの横並び形式からオールドスタイル字体に変えます。
ordn	序数	デフォルトのアルファベットグリフを、数字の後に用いるための、対応する序数字体へ置き換え。概して Numero（U+2116）キャラクタも生成します。
ornm	飾り文字	ビュレットキャラクタと ASCII キャラクタ群を飾り文字へ置き換え。
pcap	プティットキャピタル	小文字キャラクタをプティットキャピタルに、すなわち通常のスモールキャップより背の低い大文字に変えます。
pnum	プロポーショナル数字	等幅（表形式）数字を、プロポーショナル幅を持つ数字へ置き換え。
salt	スタイリスティック字体	デフォルト字体をスタイリスティック字体へ置き換え。これらの異体字は、スウォッシュや歴史的といった決まった分類には必ずしもあてはまりません。
sinf	科学的下付き	横並びまたはオールドスタイル数字を、主に化学や数学表記用の下付き数字（より小さなグリフ）へ置き換え。

表 6.1 欧文タイポグラフィのための対応 OpenType 機能一覧（日中韓テキストのための OpenType 機能は表 6.7 に挙げます）

キーワード	名前	説明
smcp	スモールキャピタル	小文字キャラクタをスモールキャピタルに変えます。
ss01 ... ss20	スタイリスティックセット 1 ～ 20	個々のグリフのスタイリスティック字体（salt 機能を参照）に加えて、あるいはそれに替えて、フォントによっては、その文字セットの一部（複数可）に対応するスタイリスティックバリアントグリフ群のセットを含んでいるものがあります。例：欧文フォント内の小文字に対して複数のバリアント。
subs	下付き	デフォルトグリフを下付きグリフへ置き換え。
sups	上付き	横並びまたはオールドスタイル数字を上付き数字へ置き換え（主に脚注表示用）、小文字を上付き文字へ置き換え（主にフランス語敬称の省略形用）。
swsh	スウォッシュ	デフォルトグリフを、対応するスウォッシュグリフへ置き換え。
titl	見出し化	デフォルトグリフを、見出し用にデザインされた、対応する字体へ置き換え。
tnum	等幅数字	プロポーショナル数字を等幅（表形式）数字へ置き換え。
unic	ユニケース	大文字と小文字を、小文字とスモールキャピタル字体の混在するセットへマップして、高さ一定のアルファベットにします。
zero	スラッシュ付きゼロ	数字ゼロに対するグリフを、中空部に斜め線を引いた字体へ置き換え。

6.3.2 テキスト行・テキストフローで OpenType レイアウト機能

PDFlib は、テキスト行・テキストフロー関数群では OpenType レイアウト機能に対応していますが、低レベルテキスト出力関数群（*PDF_show( )* 等）では対応していません。

OpenType レイアウト機能のための要件　OpenType レイアウト機能とともに使うフォントは、下記の要件を満たす必要があります：

- フォントは、TrueType（**.ttf*）・OpenType（**.otf*）・TrueType コレクション（**.ttc*）フォントのいずれかである必要があります。標準日中韓フォントについては、そのフォントファイルが得られる必要があります。
- フォントファイルは、テキスト内で使いたい OpenType 機能に対する対応ルックアップを持つ GSUB テーブルを含んでいる必要があります（後述）。
- フォントは、*encoding=unicode* か *glyphid* で、または Unicode CMap を指定して読み込む必要があります。
- *PDF_load_font( )* の *readfeatures* オプションは *false* に設定する必要があります。
- *PDF_load_font( )* の *fallbackfonts* オプションを用いている場合は、1 つのテキスト区間内のテキストは、ベースフォントからのグリフと予備フォントからのグリフを（または複数の予備フォントからのグリフを）同時に含んではいけません。

注 PDFlib は、GSUB ルックアップ種別 1（一対一置換）と 4（多対一置換）を持つ OpenType 機能に対応しています。カーニングを除き、PDFlib は、GPOS テーブルに基づく OpenType レイアウト機能には対応していません。

注意　OpenType 機能を扱う際には下記に留意してください：

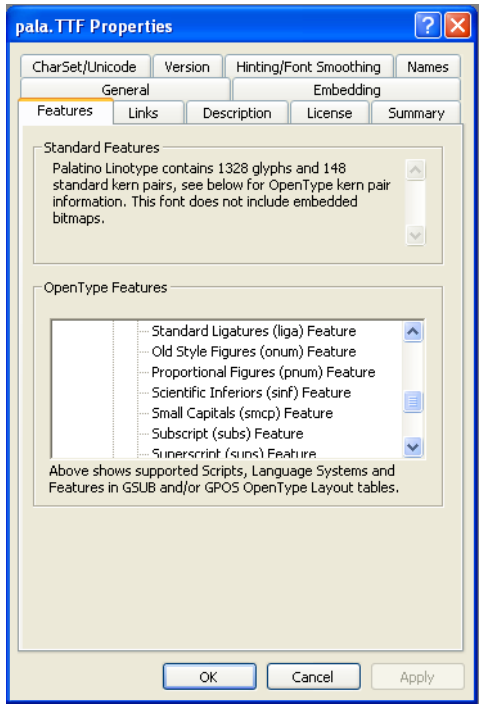


図 6.3
Microsoft の font property extension はフォント内の OpenType 機能一覧を表示します

- OpenType 機能（オプション *features*・*script*・*language*）は、同一フォント内のグリフ群に対してのみ適用されますので、予備フォントを指定している場合は、ベースフォントと 1 個ないし複数の予備フォントとにわたるグリフ群には適用されません。
- 必ず、必要な場面でそのつど各機能を有効・無効を切り替えてください。OpenType 機能をうっかり全テキストに対して有効なままにしておくと、予期しない結果が生じることがあります。

OpenType 機能の有効・無効を切り替える　テキストの部分部分に対して、必要に応じて OpenType 機能の有効と無効を切り替えることができます。機能を有効にするには、*features* テキストオプションを用いてその名前を与えます。機能名の頭に *no* をつけるとそれを無効にできます。たとえば、テキストフローに対するインラインオプションリストでは機能制御は下記のようになります：

```
<features={liga}>ffi<features={noliga}
```

テキスト行に対しては OpenType 機能を下記のように有効にできます：

```
p.fit_textline("ffi", x, y, "features={liga}");
```

OpenType 機能は、PDFlib Personalization Server（PPS）で利用するためのブロックプロパティとして有効にすることも可能です。

複数の機能を同一テキストに対して適用することも可能ですが、フォント内の機能テーブル群がこの状況に対して整備されている必要があり、対応する機能ルックアップを正しい順序で含んでいる必要があります。たとえば、単語 *office* に対して合字機能（*liga*）とスモールキャピタル機能（*smcp*）を適用する場合を考えます。両方の機能を有効にする場合（対応する機能エントリ群がフォントに含まれている前提で）、スモールキャピタ

ル機能が適用され、合字機能は適用されないことを期待すると思います。これがフォントのテーブル内に正しく実装されている場合は、PDFlib は期待どおりの出力を生成します。すなわち合字なしでスモールキャピタルを適用します。

制御キャラクタで合字を無効に　言語によっては、ある特定の状況では合字を使ってはいけないものがあります。ドイツ語などの言語におけるタイポグラフィ規則では、複合語の境界をまたぐ合字の使用を禁じています。たとえば *f+i* の組み合わせは、単語 *Schilfinsel* 内では合字へ置き換えてはいけません。合成された 2 つの単語の境界をまたいでいるからです。

先述のように、合字などの OpenType 機能の処理は、*features* オプションで有効・無効を切り替えることができます。上記のような例外的場合のたびに合字をオプションで無効にしていくのは面倒です。シンプルな合字制御を提供するために、テキスト内の制御キャラクタで合字を無効にする機能がありますので、これを利用すれば、機能の有効・無効をたくさんのオプションで切り替える必要はなくなります。連続するキャラクタの間に**ゼロ幅非接合子**（U+200C、*&zwnj*。表 6.4 も参照）キャラクタを挿入すれば、合字が *features* オプションで有効にしてあっても、その箇所は合字へ置き換わらなくなります。たとえば、下記のコードは *f+i* 合字を生成しません：

```
<features={liga charref=true}>Schilf&zwnj;insel
```

用字系・言語固有の OpenType レイアウト機能　OpenType 機能は、あらゆる場面で適用されるものもありますし、ある特定の用字系に対して実装されているものもあります。ある特定の用字系と言語との組み合わせに対して実装されているものもあります。このため、*features* オプションに加えて *script*・*language* テキストオプションを与えることが可能です。これらが実効性を持つのは、その機能がフォント内で用字系または言語に固有な形で実装されている場合のみです。

たとえば、*f*・*i* グリフに対する合字は、いくつかのフォントにおいては、トルコ語が選ばれているときには得られません（なぜならトルコ語では点のない *i* が頻出するため、*i* の合字形はそれと混同しやすいからです）。そのようなフォントを用いている場合に、下記のテキストフローオプションは、用字系 / 言語を一切指定していないので合字を生成します：

```
<features={liga}>fi
```

しかし、下記のテキストフローオプションリストは、トルコ語オプションがあるので合字を生成しません：

```
<script=latn language=TRK features={liga}>fi
```

locl 機能は、明示的に言語固有のキャラクタ字体を選択します。*liga* 機能は、言語固有の合字を含んでいます。言語固有機能のいくつかの例：

セルビア語用キャラクタ字体:

```
<features={locl} script=cyrl language=SRB charref>&#x0431;
```

ウルドゥー語用数字字体:

```
<features={locl} script=arab language=URD charref>&#x0662;&#x0663;&#x0664;&#x0665;
```

使える言語・用字系キーワードについては 163 ページ「6.4.2 用字系と言語」を参照してください。

OpenType 機能とシェーピングを組み合わせ　複雑用字系のシェーピング（161 ページ「6.4 複雑用字系出力」を参照）は、自動的に選ばれる OpenType フォント機能に大きく依存しています。しかし、フォントによっては、シェーピングのために自動的に選ばれる OpenType 機能を、クライアントアプリケーションが選んだ OpenType 機能と組み合わせることが意味を持つ場合があります。PDFlib は、まずユーザーが選んだ OpenType 機能を適用し（オプション *features*）、ついでシェーピング関連の機能を自動的に適用します（オプション *shaping*・*script*・*language*）。

OpenType 機能をプログラム的に取得　フォント内の OpenType 機能を *PDF_info_font()* で取得することができます。下記のコードでは、フォント内で得られる、かつ PDFlib が対応しているすべての OpenType 機能をカンマで区切ったリストを取得しています：

```
result = (int) p.info_font(font, "featurelist", "");
if (result != -1)
{
        /* スペース区切りされた機能リストを内容として持つ文字列を取得 */
        featurelist = p.get_parameter("string", result);
}
else
{
        /* 対応機能は全く見つからなかった */
}
```

ある特定の機能に PDFlib と対象フォントが対応しているかどうかを調べるには下記のコードを用います。この場合は合字（*liga*）：

```
result = (int) p.info_font(font, "feature", "name=liga");
if (result == 1)
{
        /* 機能にフォントとPDFlibが対応している */
}
```

6.4 複雑用字系出力

クックブック 完全なコードサンプルがクックブックの scripts/starter_shaping トピックにあります。

6.4.1 複雑用字系

欧文用字系では基本的に左から右へ、1 つのキャラクタの後に次のキャラクタを置いていきます。それ以外の書記系では、正しいテキスト出力のためには追加の要請があります。このような書記系を複雑用字系と呼ぶことにします。PDFlib は、表 6.2 に挙げるものを含むさまざまな用字系に対して、複雑用字系のためのテキスト処理を実行します。

この項では、複雑用字系におけるシェーピングについて詳しく解説します。西洋の諸言語の多くは単純に左から右へ1つのキャラクタの後に次のキャラクタを置いていけば書けるのに対し、書記系（用字系）によっては追加の処理を必要とするものがあります：

- アラビア文字とヘブライ文字の用字系では、テキストを右から左へ置いていきます。混合テキスト（アラビア文の中に欧文がはさまった）は、右書きの部分と左書きの部分の両方を含みます。これらの部分は並べ替えの必要があり、これを双方向（Bidi）問題といいます。
- いくつかの用字系、とりわけアラビア文字では、キャラクタの位置（単独、語頭 / 語中 / 語尾）によって異なるキャラクタ形状を用います。
- キャラクタ列を必須合字へ置き換えます。
- グリフの位置を縦・横へ調整する必要があります。
- インド系用字系では、いくつかのキャラクタの並べ替えが必要です。すなわち、キャラクタはテキスト内での位置が変わることがあります。
- いくつかの用字系には、特殊な単語境界・均等配置規則が適用されます。

これらの処理ステップを 1 つないし複数必要とする用字系を複雑用字系と呼びます。入力された論理テキストから正しい表記を作り上げる処理をシェーピングといいます（この用語は並べ替えと双方向処理をも含んでいます）。ユーザーはつねに、シェーピングされていない形で論理的順序のテキストを与え、それに対して PDFlib が、必要なシェーピングを実行してから PDF 出力を生成します。

複雑用字系のシェーピングは、*shaping* テキストオプションで有効にできます。このオプションは *script* オプションを必要とし、また、*language* オプションをあわせて指定することもできます。下記のオプションリストはアラビア文字のシェーピング（と双方向処理）を可能にします：

```
shaping script=arab
```

注意　複雑用字系のシェーピングを扱う際には下記に留意してください：

- PDFlib は *shaping*・*script* オプションを自動的には設定せず、ユーザーがそれらを与えると前提しています。
- 用字系固有のシェーピング（オプション *shaping*・*script*・*language*）は、同一フォントからのグリフ群に対してのみ適用され、複数のフォントにわたるグリフ群には適用されません。予備フォントを使っている場合は、シェーピングは同一（ベースまたは予備）フォントのテキスト区間内でのみ適用されます。
- シェーピングはテキスト内のキャラクタの順番を変えることがありますので、単語内の属性変更については注意を払う必要があります。たとえば、テキストフロー内でインラインオプションを用いて単語内の 2 番目のキャラクタに色をつけようとしている場合に、シェーピングが 1 番目と 2 番目のキャラクタを入れ替えたらどうなるのでしょ

表 6.2 複雑用字系と script オプションのキーワード一覧

書記系	用字系の名前	言語 / 地域（不完全なリスト）	スクリプトキーワード
用字系指定なし	–		_none
用字系自動検出	–	このキーワードは、テキスト内のキャラクタの多数が属する用字系を選択します。その際、_latn・_none は無視されます。	_auto
欧州アルファベット	欧文	多くの欧州などの諸語	latn
	ギリシャ文字	ギリシャ語	grek
	キリル文字	ロシア語など多くのスラブ諸語	cyrl
中東	アラビア文字	アラビア語・ペルシャ語（ファールシー）・ウルドゥー語・パシュトー語など	arab
	ヘブライ文字	ヘブライ語・イディッシュ語など	hebr
	シリア文字	シリア正教会・マロン派・アッシリア教会	syrc
	ターナ文字	ディベヒ語 / モルディブ	thaa
南アジア（インド）	デーヴァナーガリー	ヒンディー・古典サンスクリット	deva
	ベンガル文字	ベンガル語・アッサム語	beng
	グルムキー文字	パンジャーブ語	guru
	グジャラート文字	グジャラート語	gujr
	オリヤー文字	オリヤー語 / オリッサ	orya
	タミル文字	タミル語 / タミルナードゥ・スリランカ	taml
	テルグ文字	テルグ語 / アーンドラプラデーシュ	telu
	カンナダ文字	カンナダ語 / カルナータカ	knda
	マラヤーラム文字	マラヤーラム語 / ケーララ	mlym
東南アジア	タイ文字	タイ語	thai
	ラーオ文字	ラーオ語	「lao 」[1]
	クメール文字	クメール語（カンボジア語）	khmr
東アジア	漢字	中国語・日本語・韓国語	hani
	ひらがな	日本語	hira
	カタカナ	日本語	kana
	ハングル	韓国語	hang
その他	OpenType 仕様に従ったその他の 4 文字コードも働きますが、対応していません。全一覧は下記にあります。 www.microsoft.com/typography/developers/OpenType/scripttags.aspx		

1．末尾の空白キャラクタに注意。

うか。この理由から、書式の変更は単語の途中では行わずに、単語の境界でのみ行う必要があります。

シェーピングのための要件　複雑用字系のシェーピングとともに用いるためのフォントは、対象用字系のグリフ群を含んでいるということに加えて、下記の要件を満たしている必要があります：

- GDEF・GSUB・GPOS 機能テーブルを持ち、かつ、対象用字系に適合した正しい Unicode マッピングを持つ TrueType または OpenType フォントである必要があります。あるいはこうした OpenType テーブルを持たずに、アラビア文字・ヘブライ文字の場合は、フォントが Unicode の表示形を含んでいることもできます（Arabic Apple フォント等はこの方式で構築されています）。この場合は内部テーブルがシェーピング処理に使われます。タイ文字テキストについては、Microsoft・Apple・Monotype Worldtype（いくつかの IBM 製品等で用いられています）のタイ文字用規則に従ったコンテキスト依存字体をフォントが含んでいる必要があります。
- 標準日中韓フォントを使う場合は、そのフォントファイルが得られる必要があります。
- フォントを *encoding=unicode* か *glyphid* で読み込む必要があります。
- *PDF_load_font()* の *monospace・vertical* オプションは用いてはいけません。また、*readshaping* オプションを false に設定してはいけません。
- *PDF_load_font()* の *fallbackfonts* オプションが用いられた場合は、1 つのテキスト区間内のテキストが予備フォントからのグリフを含んではいけません。

6.4.2 用字系と言語

用字系と言語は、以下に挙げる機能面での役割を果たします。これらは以下のオプションで制御できます：

- *script* テキストオプションは対象用字系（書記系）を指定します。表 6.2 に挙げた 4 文字のキーワードを使えます。例：

```
script=latn
script=cyrl
script=arab
script=hebr
script=deva
script={lao }
```

script=_auto にすると、PDFlib は、テキスト内のキャラクタの多数が属する用字系を自動的に割り当てます。欧文テキストはシェーピングを必要としませんので、この用字系を自動的に決定する際には考慮されません。
PDF_info_textline() の *scriptlist* キーワードを使うと、テキストに対して用いられている用字系を取得することができます。

- *language* オプションは、テキストが書かれている自然言語を指定します。表 6.2 に挙げる 3 キャラクタのキーワードが使えます。例：

```
language=ARA
language=URD
language=ZHS
language=HIN
```

複雑用字系処理　複雑用字系処理（オプション *shaping*）には *script* オプションが必要です。*language* オプションを追加で与えることもできます。これはシェーピングの言語固有の側面を制御します。たとえばアラビア語とウルドゥー語で数字が別になります。しかし、言語固有用字系シェーピングテーブルを含むフォントはわずかですので、多くの場合

は *script* オプションを指定すれば充分であり、*language* オプションを指定してもシェーピングは改善できません。

OpenType レイアウト機能　フォントは、OpenType レイアウト機能を言語固有なやり方で実装することができます（159 ページ「用字系・言語固有の OpenType レイアウト機能」を参照）。若干の機能は、*script*・*language* オプションに従って動作が変わることがありますが、これらのオプションなしでも使えるのに対し（*liga* 等）、locl 機能は *script*・*language* オプションと組み合わせてのみ意味を持ちます。

注 テキストフローにおける高度な改行（221 ページ「8.2.9 高度な用字系固有の改行」を参照）でも、言語固有の処理が行われますが、これは *language* オプションによって司られるのではなく、*locale* オプションによって司られます。*locale* オプションは言語だけでなく、国と地域をも特定するものです。

6.4.3 複雑用字系のシェーピング

シェーピング処理は、キャラクタが単語の先頭・途中・末尾または単独位置のいずれにあるかによって適切なグリフ字体を選択します。シェーピングは、アラビア文字・ヒンディー文字テキスト処理の不可欠な要素です。シェーピングは、2 個以上のキャラクタ列を適切な合字へ置き換えることもあります。シェーピング処理は適切なキャラクタ字体を自動的に決定しますので、明示的な合字と Unicode の表示形（U+FB50 のアラビア文字表示形 A 等）を入力キャラクタとして用いてはいけません。

複雑用字系は、1 つのキャラクタに対して複数の異なるグリフ字体を必要とし、かつこれらのグリフを選択し配置するための追加の規則を要することから、複雑用字系のシェーピングはあらゆる種類のフォントで働くわけではなく、必要な情報を含んだ適当なフォントが必要です。シェーピングは、必要な機能テーブルを含む TrueType・OpenType フォントで働きます（要件の詳細は 163 ページ「シェーピングのための要件」を参照）。

シェーピングは、同一フォント内のキャラクタ群に対してのみ行うことができます。なぜならシェーピング情報は特定のフォントに固有のものだからです。たとえば、複数の異なるフォントにわたる合字を形成することは意味がありませんので、複雑用字系のシェーピングは、複数の異なるフォントからのキャラクタを含む単語には適用することができません。

シェーピング動作を上書き　場合によっては、ユーザーがデフォルトのシェーピング動作を上書きしたいこともあります。PDFlib はこの目的のためにいくつかの Unicode 組版キャラクタに対応しています。利用の便宜のため、これらの組版キャラクタは実体で指定することも可能です（表 6.4 参照）。

6.4.4 双方向組版

クックブック 完全なコードサンプルがクックブックの`complex_scripts/bidi_formatting`トピックにあります。

右書きのテキスト（アラビア文字・ヘブライ文字をはじめとするさまざまな用字系）においては、アドレスや別言語による引用等で、左書きの欧文テキスト列が入れ子になることが非常に頻繁にあります。このような混在テキスト列では双方向（Bidi）組版が必要になります。数字はつねに左書きされますので、双方向問題は、全くアラビア文字・ヘブライ文字だけで書かれたテキストにも生じます。PDFlib は双方向テキスト並べ替えを、Unicode 規格付録 #9[1] に示された Unicode 双方向アルゴリズムに従って実装しています。双方向処

表 6.3 language オプションのキーワード一覧

キーワード	言語	キーワード	言語	キーワード	言語
_none	言語指定なし	FIN	フィンランド語	NEP	ネパール語
AFK	アフリカーンス	FRA	フランス語	ORI	オリヤー語
SQI	アルバニア語	GAE	ゲール語	PAS	パシュトー語
ARA	アラビア語	DEU	ドイツ語	PLK	ポーランド語
HYE	アルメニア語	ELL	ギリシャ語	PTG	ポルトガル語
ASM	アッサム語	GUJ	グジャラート語	ROM	ルーマニア語
EUQ	バスク語	HAU	ハウサ語	RUS	ロシア語
BEL	ベラルーシ語	IWR	ヘブライ語	SAN	サンスクリット
BEN	ベンガル語	HIN	ヒンディー	SRB	セルビア語
BGR	ブルガリア語	HUN	ハンガリー語	SND	シンド語
CAT	カタルーニャ語	IND	インドネシア語	SNH	シンハラ語
CHE	チェチェン語	ITA	イタリア語	SKY	スロバキア語
ZHP	中国語注音符号	JAN	日本語	SLV	スロベニア語
ZHS	中国語簡体字	KAN	カンナダ語	ESP	スペイン語
ZHT	中国語繁体字	KSH	カシミール語	SVE	スウェーデン語
COP	コプト語	KHM	クメール語	SYR	シリア語
HRV	クロアチア語	KOK	コンカニ語	TAM	タミル語
CSY	チェコ語	KOR	韓国語	TEL	テルグ語
DAN	デンマーク語	MLR	改正マラヤーラム語	THA	タイ語
NLD	オランダ語	MAL	伝統マラヤーラム語	TIB	チベット語
DZN	ゾンカ	MTS	マルタ語	TRK	トルコ語[1]
ENG	英語	MNI	マニプリ語	URD	ウルドゥー語
ETI	エストニア語	MAR	マラーティー語	WEL	ウェールズ語
FAR	ペルシア語	MNG	モンゴル語	JII	イディッシュ語

1. いくつかのフォントはトルコ語に対して誤って TUR を用いていますので、PDFlib はこのタグを TRK と等価として扱います。

表 6.4 デフォルトシェーピング動作を上書きするための Unicode 制御キャラクター一覧

組版キャラクタ	実体名	Unicode 名	機能
U+200C	ZWNJ	ゼロ幅非接合子	隣り合う 2 個のキャラクタが続け字にならないようにします
U+200D	ZWJ	ゼロ幅接合子	隣り合う 2 個のキャラクタが続け字になるようにします

1. *www.unicode.org/unicode/reports/tr9/* を参照。

理は、オプションで有効にする必要はなく、右書きのテキストが適切な *script* オプションとともに現れた際には、シェーピング処理の一環として自動的に適用されます。

注 双方向処理は現在のところ、複数行テキストフローでは対応しておらず、テキスト行（すなわち一行テキスト出力）でのみ対応しています。

双方向アルゴリズムを上書き 自動双方向処理は多くの場合において適切な結果を与えますが、場合によっては明示的なユーザー制御が必要なこともあります。PDFlib ではこの目的のためにいくつかの方向組版コードに対応しています。利用の便宜のため、これらの組版キャラクタは実体で指定することも可能です（表 6.5 参照）。これらの双方向組版コードは、下記のような場合にデフォルトの双方向アルゴリズムを上書きするのに有用です：

- 右書きの段落が左書きのキャラクタ群で始まる場合。
- 混在テキスト列が入れ子になっている場合。
- 左書きテキストと右書きテキストとの間の境界に、句読点等の弱いキャラクタ群がある場合。
- 混在テキストを含む製品番号等の場合。

表 6.5 双方向アルゴリズムを上書きするための方向組版コード一覧

組版コード	実体名	Unicode 名	機能
U+202A	LRE	左書き埋め込み（LRE）	埋め込み左書き列を開始します
U+202B	RLE	右書き埋め込み（RLE）	埋め込み右書き列を開始します
U+200E	LRM	左書きマーク（LRM）	左書きのゼロ幅キャラクタ
U+200F	RLM	右書きマーク（RLM）	右書きのゼロ幅キャラクタ
U+202D	LRO	左書き上書き（LRO）	キャラクタ群を強い左書きキャラクタ群として扱うよう強制
U+202E	RLO	右書き上書き（RLO）	キャラクタ群を強い右書きキャラクタ群として扱うよう強制
U+202C	PDF	方向組版ポップ（PDF）	直前の LRE・RLE・RLO・LRO の前の双方向状態へ復帰

右書き文書処理の向上のためのオプション さまざまな組版オプションや Acrobat の動作のデフォルト設定は、左書きテキスト出力に合わせて設定されています。右書きのテキスト組版と文書表示のためには、下記のオプションを用います：

- テキスト行を、下記のはめ込みオプションで右揃えで配置します：

```
position={right center}
```

- リーダを、テキストと左枠の間に生成します：

```
leader={alignment=left text=.}
```

- *PDF_begin/end_document()* の下記のオプションを用いて、Acrobat での右書き文書・ページ表示を改善します：

```
viewerpreferences={direction=r2l}
```

コード内で双方向テキストを扱う 双方向テキストを扱う際には下記も有用でしょう：

- *PDF_info_textline()* の *startx/starty・endx/endy* キーワードを用いると、それぞれ論理的開始・終了キャラクタの座標を知ることができます。

- *PDF_info_textline()* の *writingdirx* キーワードを用いると、テキストの主流な書記方向を知ることができます。この方向は、テキストの先頭キャラクタ群から、または表 6.5 に従った方向組版コード（テキスト内にあれば）から推定されます。
- *PDF_info_textline()* の *position* オプションで *auto* キーワードを用いると、自動的にアラビア文字またはヘブライ文字のテキストは右枠へ、欧文テキストは左枠へ寄せられます。たとえば下記のテキスト行オプションは、テキストをベースライン上に右寄せまたは左寄せします：

```
boxsize={width 0} position={auto bottom}
```

6.4.5 アラビア文字テキスト組版

クックブック **完全なコードサンプルがクックブックの**complex_scripts/arabic_formatting**トピックにあります。**

上述の双方向組版とテキストのシェーピングに加え、アラビア用字系のテキスト出力の生成に関しては、ほかにも組版上の側面がいくつかあります。

アラビア合字　アラビア用字系は合字を多用します。多くのアラビア文字フォントは 2 種類の合字を含んでおり、それぞれ PDFlib では異なる扱いを受けます：

- 必須合字（*rlig* 機能）はつねに適用する必要があります。ラーム－アリフおよびその派生形等がこれにあたります。必須合字は、*script=arab* で *shaping* オプションを有効にしている場合に用いられます。
- 任意アラビア合字（*liga*・*dlig*　機能）は自動的には用いられず、他のユーザー制御の OpenType 機能と同様に、*features={liga}* で有効にすることができます。任意アラビア合字は、複雑用字系処理とシェーピングの後に適用されます。

アラビア文字テキスト内の欧文合字　テキスト行では、OpenType 機能の用字系固有処理は予期しない結果を生み出すことがあります。たとえば、欧文合字は同一テキスト行内でアラビア文字テキストと混在すると働きません。その原因は、テキスト行の内容に対しては *script* オプションを一度しか与えることができず、それが *shaping*・*feature* オプションの両方に効力を持つからです：

```
shaping script=arab features={liga}          誤り。多くのフォントでは働きません！
```

しかし、アラビア文字フォントは通常、欧文合字を、アラビア用字系指定に対しては含んでおらず、デフォルトまたは欧文用字系に対してのみ含んでおり、しかし 1 つのテキスト行の中で *script* オプションは変えることができません。このため、PDFlib は欧文合字を一切見つけることができずに、プレーンなキャラクタを出力します。

合字をさせない　ある種の略称等、場合によっては、隣り合うキャラクタを合字にさせたくないことがあります。この場合には、表 6.4 に挙げた組版キャラクタを用いて、合字を強制したり禁止したりすることができます。たとえば、下記の例のゼロ幅非結合子は、キャラクタが合字を形成することを禁止して、正しい略称表記を生成しています：

```
&#x0623;&#x064A;&ZWNJ;&#x0628;&#x064A;&ZWNJ;&#x0625;&#x0645;
```

アラビア文字テキストにおけるタトウィール　タトウィールキャラクタ U+0640（カシーダともいう）を 1 個ないし複数挿入することによって、アラビア単語を引き伸ばすことができます。PDFlib は自動的にタトウィールキャラクタを挿入することによるテキストの均

等揃えは行いませんが、このキャラクタを自分で入力テキスト内に挿入して単語を引き伸ばすことはできます。

アラビア文字フォントに欧文キャラクタを追加　いくつかのアラビア文字フォントは、欧文キャラクタに対するグリフを一切含んでいません。Apple Mac OS X に同梱しているアラビア文字フォント等がこれにあたります。この場合は *fallbackfonts* オプションを使って、欧文キャラクタをアラビア文字フォントに連結することができます。PDFlib は、欧文またはアラビア文字のテキスト入力に従って自動的に両フォントを切り替えます。すなわち、アプリケーション側でフォントを切り替える必要はなく、欧文とアラビア文字が混在するテキストを 1 個のフォント指定で与えることができます。

fallbackfonts オプションに対して下記のフォント読み込みオプションリストを用いると、読み込んだアラビア文字フォントへ Helvetica フォントから欧文キャラクタを追加することができます：

```
fallbackfonts={
 {fontname=Helvetica encoding=unicode forcechars={U+0021-U+00FF}}
}
```

6.5 日本語・中国語・韓国語テキスト出力

6.5.1 標準日中韓フォント

Acrobat は、日中韓フォント用のさまざまな標準フォントに対応しています。こうしたフォントは、Acrobat のインストールとともに（または Asian FontPack で）提供されますので、PDF ファイルに埋め込む必要がありません。このようなフォントは、よく利用されるエンコーディングに必要なキャラクタをすべて含んでおり、また、縦書きにも横書きにも対応しています。標準フォントの一覧を、適用可能な CMap とともに表 6.6 に示します（日中韓CMapについての詳細は103ページ「4.3 日本語・中国語・韓国語エンコーディング」参照）。

注 カスタム日中韓フォントに対しては、Unicode　CMap（UCS2 または UTF16）ではなく、*encoding=unicode* の使用を推奨します。

注 Acrobat の標準日中韓フォントはボールド体とイタリック体に対応していません。しかし、擬似フォントスタイル機能で擬似的に印字させることはできます（151 ページ「6.2.3 テキストバリエーション」参照）。

表 6.6 日本語・中国語・韓国語テキスト用の Acrobat 標準フォント・CMap（エンコーディング）一覧

ロケール	フォント名	印字例	対応 CMap（エンコーディング）
日本語	KozMinPro-Regular-Acro[1] KozGoPro-Medium[2] KozMinProVI-Regular[2]	日本語	83pv-RKSJ-H, 90ms-RKSJ-H, 90ms-RKSJ-V, 90msp-RKSJ-H, 90msp-RKSJ-V, 90pv-RKSJ-H, Add-RKSJ-H, Add-RKSJ-V, EUC-H, EUC-V, Ext-RKSJ-H, Ext-RKSJ-V, H, V, UniJIS-UCS2-H, UniJIS-UCS2-V, UniJIS-UCS2-HW-H[1], UniJIS-UCS2-HW-V[1], UniJIS-UTF16-H[3], UniJIS-UTF16-V[3]
中国語 簡体字	AdobeSongStd-Light[2]	国际	GB-EUC-H, GB-EUC-V, GBpc-EUC-H, GBpc-EUC-V, GBK-EUC-H, GBK-EUC-V, GBKp-EUC-H, GBKp-EUC-V, GBK2K-H, GBK2K-V, UniGB-UCS2-H, UniGB-UCS2-V, UniGB-UTF16-H[3], UniGB-UTF16-V[3]
中国語 繁体字	AdobeMingStd-Light[2]	中文	B5pc-H, B5pc-V, HKscs-B5-H, HKscs-B5-V, ETen-B5-H, ETen-B5-V, ETenms-B5-H, ETenms-B5-V, CNS-EUC-H, CNS-EUC-V, UniCNS-UCS2-H, UniCNS-UCS2-V, UniCNS-UTF16-H[3], UniCNS-UTF16-V[3]
韓国語	AdobeMyungjoStd-Medium[2]	한국	KSC-EUC-H, KSC-EUC-V, KSCms-UHC-H, KSCms-UHC-V, KSCms-UHC-HW-H, KSCms-UHC-HW-V, KSCpc-EUC-H, UniKS-UCS2-H, UniKS-UCS2-V, UniKS-UTF16-H[3], UniKS-UTF16-V[3]

1. HW CMap 群は、KozMinPro-Regular-Acro・KozGoPro-Medium-Acro フォントに対しては用いることはできません。なぜならこれらのフォントにはプロポーショナル ASCII キャラクタしかなく、半角文字がないためです。
2. PDF 1.6 以上を生成するときのみ利用可能
3. PDF 1.5 以上を生成するときのみ利用可能

ネイティブな日中韓レガシコードを保持　*keepnative=true* の場合には、選択された CMap に従ったネイティブなレガシキャラクタコード（Shift-JIS 等）が PDF 出力へ書き込まれます。そうでなければテキストは Unicode へ変換されます。*keepnative=true* の利点は、そのようなフォントは埋め込みなしでフォームフィールドで使えることです（*keepnative* フォ

ント読み込みオプションの説明は PDFlib API リファレンスを参照してください）。*keepnative=false* の場合には、レガシコードは CID 値へ変換されたうえで PDF 出力へ書き込まれます。その利点は、OpenType 機能が利用できることと、テキストフロー組版機能が利用できることです。見た目はどちらの場合も同等です。

横書きと縦書き　PDFlib は、横書きにも縦書きにも対応しています。縦書きはさまざまな方法で要求できます（ただし Type 1 フォントでは縦書きには対応していません）：

- 標準日中韓フォント・CMap については、横書きか縦書きかは、適切な CMap 名を選ぶことによってエンコーディングとともに選択されます。CMap 名の末尾が *-H* なら横書きが選択され、*-V* なら縦書きが選択されます。
- CMap以外のエンコーディングを持つフォントの場合は、*vertical*オプションを与えれば縦書きで利用できます。
- 「@」キャラクタで始まるフォント名はつねに縦書きで処理されます。

注 字間は縦書きでは負の値でなければなりません。これは、文字を１個ずつ印字しては戻るためです。

標準日中韓フォントの利用例　標準日中韓フォントを選択するには、*PDF_load_font()* インタフェースを用いて、CMap 名を *encoding* パラメタとして与えます。しかし、ある 1 つの日中韓フォントはある特定の CMap のセットにしか対応していませんし（表 6.6 参照）、Unicode 利用可能な言語バインディングは UCS2 互換の CMap にしか対応していないことを考慮する必要があります。表 6.6 にある *KozMinPro-Regular-Acro* の印字例は下記のコードで生成することができます：

```
font = p.load_font("KozMinPro-Regular-Acro", "UniJIS-UCS2-H", "");
if (font == -1) { ... }
p.setfont(font, 24);
p.set_text_pos(50, 500);
p.show("\u65E5\u672C\u8A9E");
```

上記の命令群は日本語の標準フォントの 1 つを特定し、Shift-JIS 互換の CMap（*Ext-RKSJ*）と横書き（*H*）を選びます。*fontname* 引数はフォントの正確な名前でなければならず、エンコーディングや縦書き横書きを表す接尾辞は一切つけてはいけません。*encoding* 引数は対応 CMap のうちのいずれかの名前であり（どれを選ぶかはフォントによります）、縦書きか横書きかもその中で表されます（上述）。PDFlib は、Acrobat のすべてのデフォルト CMap に対応しており、要求されたフォントと CMap とが合わないことを発見したときにはそのことを知らせてきます。たとえば、韓国語フォントを日本語エンコーディングで使用するよう要求されても PDFlib は拒否します。

等幅フォントを強制　アプリケーションのなかには、プロポーショナルな日中韓フォントを取り扱う仕組みを持たず、テキスト幅の計算を固定グリフ幅×グリフ数で済ませてしまうものもあります。PDFlib では、グリフの幅が通常一定でないフォントに対しても、等幅のグリフを強制させることが可能です。*PDF_load_font()* で *monospace* オプションを用いて、すべてのグリフに対してとらせたい幅を指定します。標準日中韓フォントに対しては、下記のように値 1000 を指定すれば満足できる結果が得られます：

```
font = p.load_font("KozMinPro-Regular-Acro", "UniJIS-UCS2-H", "monospace=1000");
```

この *monospace* オプションは、標準日中韓フォントに対してのみ推奨されます。

6.5.2 カスタム日中韓フォント

注 PDFlib GmbH では、MS ゴシック・MS 明朝フォントを www.pdflib.com で無償ダウンロード提供しています。PDFlib のライセンスをお持ちの方はこれらのフォントを、別途フォントライセンスを得る必要なく利用する権利を有します。

Acrobat の標準日中韓フォントのほかに、PDFlib は、TrueType 形式（TrueType Collections = TTC を含む）や OpenType 形式のカスタム日中韓フォント（表 6.6 の一覧にないフォント）にも対応しています。カスタム日中韓フォントは次のように処理されます。

- *embedding* オプションが *true* ならば、フォントは CID フォントに変換されて、PDF 出力に埋め込まれます。
- Windows 上の日中韓ホストフォント名は、先頭に BOM のついた UTF-8 形式か UTF-16 形式で *PDF_load_font()* に与えることができます。しかし、Mac では非欧文ホストフォント名に対応していません。
- *keepnative* オプションはデフォルトで *false* です。Acrobat での面倒な問題を避けるため、フォント埋め込みをしなくてよいなら *keepnative=false* に設定することを、また、*keepnative=true* にしたいなら *embedding=true* に設定することを推奨します。

日本語 Shift-JIS テキストでのカスタム日中韓フォントの利用例 下記の例は、MS 明朝フォントを用いて、Windows コードページ 932 に従って Shift-JIS 形式で与えられた日本語テキストを表示するものです：

```
font = PDF_load_font(p, "MS Mincho", 0, "cp932", "");
if (font == -1) { ... }

PDF_setfont(p, font, 24);
PDF_set_text_pos(p, 50, 500);

PDF_show2(p, "\x82\xA9\x82\xC8\x8A\xBF\x8E\x9A", 8);
```

中国語 Unicode テキストでのカスタム日中韓フォントの利用例 下記の例では、中国語テキストを ArialUnicodeMS フォントで印字しています。フォントは、システムにインストールされているか、あるいは 128 ページ「5.4.3 フォントを検索」に従って設定されている必要があります：

```
font = p.load_font("Arial Unicode MS", "unicode", "");

p.setfont(font, 24);
p.set_text_pos(50, 500);

p.show("\u4e00\u500b\u4eba");
```

TrueType Collection（TTC）内の個々のフォントを利用 TTCファイルは、複数の別々のフォントを持っています。各フォントを利用するにはその適切な名前を与えます。ただし、TTC ファイル内がどのフォントを持っているかを知らない場合には、各フォントを番号で指定することも可能です。具体的には、コロンとフォント番号（0 から始まる）を追加することにより指定します。番号が 0 の場合には省略可能です。たとえば、TTC ファイル *msgothic.ttc* は複数のフォントを持っており、*PDF_load_font()* で下記のように指定することができます（各行内のフォント名は等価）：

```
msgothic:0      MS Gothic       msgothic:
msgothic:1      MS PGothic
msgothic:2      MS UI Gothic
```

ただし、*msgothic*（接尾辞をつけない）はフォント名としては扱われません。なぜならそれではフォントを一意に特定できないからです。フォント名の別名（128 ページ「フォントデータの情報源」参照）を TTC 番号と組み合わせて用いることも可能です。指定された番号のフォントが見つからないときには、関数呼び出しは失敗します。

TTC フォントファイルは 1 回のみ設定しなければなりません。TTC ファイル内のすべての番号づけられたフォントは自動的に発見されます。以下に、*msgothic.ttc* 内のすべての番号づけられたフォントを設定するために充分なコードを示します（128 ページ「5.4.3 フォントを検索」参照）：

```
p.set_parameter("FontOutline", "msgothic=msgothic.ttc");
```

6.5.3 EUDC・SING フォントで外字キャラクタを利用

PDFlib は、日中韓テキストのためのカスタム外字キャラクタを利用できる Windows EUDC（エンドユーザー定義キャラクタ、**.tte*）・SING フォント（**.gai*）に対応しています。最も便利なのは、カスタムキャラクタを持ったフォントを予備フォント機構で他のフォントへ統合することでしょう。外字キャラクタは多くの場合、EUDC または SING フォントで提供されます。あるいは、外字キャラクタを Type 3 フォントとして与えることもできますが、これにはプログラミング工数がより多く必要になります。

予備フォントで外字キャラクタを利用　通常、外字キャラクタは Windows EUDC または SING グリフレットから持って来ますが、*fallbackfonts* オプションはあらゆる種類のフォントを受け付けます。ですのでこの方法は外字キャラクタに限らず、あらゆる種類の記号に対して利用することができます（例：企業ロゴが別フォント内にある場合）。*fallbackfonts* オプションに対して下記のフォント読み込みオプションを用いれば、読み込んだフォントに対して、EUDC フォントからのユーザー定義（外字）キャラクタを追加することができます：

```
fallbackfonts={
 {fontname=EUDC encoding=unicode forcechars=U+E000 fontsize=140% textrise=-20%}
}
```

ベースフォントをひとたびこの予備フォント設定で読み込めば、テキスト内の EUDC キャラクタは、フォントを変える必要なく使うことができます。

SING フォントの場合、Unicode 値は PDFlib によって自動的に決定されますので、与える必要はありません：

```
fallbackfonts={
 {fontname=PDFlibWing encoding=unicode forcechars=gaiji}
}
```

EUDC フォントを用意　Windows で得られる EUDC エディタを利用すると、PDFlib で使うカスタムキャラクタを作成することができます。以下のように操作します：

- *eudcedit.exe* を使って、1 個ないし複数のカスタムキャラクタを、望む Unicode 位置に作成します。
- ディレクトリ *\Windows\fonts* 内で *EUDC.TTE* ファイルを見つけ、それをどこか別のディレクトリへ複製します。このファイルは Windows Explorer では不可視ですので、DOS

ボックス内で *dir・copy* コマンドを用いてファイルを見つけます。そしてこのフォントを PDFlib で利用できるよう、128 ページ「5.4.3 フォントを検索」で示した方式のいずれかで設定します：

```
p.set_parameter("FontOutline", "EUDC=EUDC.TTE");
p.set_parameter("SearchPath", "...ディレクトリ名...");
```

または *EUDC.TTE* をカレントディレクトリ内に置きます。
あるいは、このように明示的にフォントファイル設定をするのではなく、下記の関数呼び出しを用いて、Windows ディレクトリから直接フォントファイルを設定することもできます。こうすると、Windows 内で用いられているカレント EUDC フォントをつねに利用することになります：

```
p.set_parameter("FontOutline", "EUDC=C:\Windows\fonts\EUDC.TTE");
```

- EUDCフォントを、任意のベースフォントに対して、先述のように*fallbackfonts*オプションを用いて統合します。フォントを直接利用したい場合は、下記の呼び出しを用いてフォントを通常どおりに PDFlib へ読み込みます：

```
font = p.load_font("EUDC", "unicode", "");
```

そして最初のステップで選んだ Unicode 値を与えてキャラクタを出力します。

6.5.4 OpenType レイアウト機能と高度な日中韓テキスト出力

154 ページ「6.3 OpenType レイアウト機能」で解説したように、PDFlib は OpenType・TrueType フォント内の高度なタイポグラフィレイアウトテーブルに対応しています。たとえば、OpenType 機能を用いて、欧文グリフのプロポーショナル幅か半角かいずれかの字体を選択したり、異体字を選択したりすることが可能です。表 6.7 に、日中韓テキスト出力のための OpenType 機能を挙げます。

vert 機能（縦書き）は、縦書きのフォント（すなわち、*PDF_load_font()* に *vertical* オプションを与えていた場合）に対しては自動的に有効になり、横書きのフォントに対しては無効になります。

表 6.7 日本語・中国語・韓国語テキスト用対応 OpenType レイアウト機能一覧（これに加えて表 6.1 に、一般的利用のための OpenType レイアウト機能を挙げています）

キーワード	名前	説明
expt	エキスパート字形	JIS78 字と同様、この機能は、標準の和文字形を、対応する、タイポグラファーが望む字形へ置き換えます。
fwid	全角	他の幅に設定されたグリフを、全角（たいていは em）に設定されたグリフへ置き換え。これは欧文キャラクタやさまざまな記号を含む可能性があります。
hkna	横組み用仮名	標準の仮名を、横書き専用に特にデザインされた字形へ置き換え。
hngl	ハングル	韓文漢字キャラクタを、対応するハングル（音素）キャラクタへ置き換え。
hojo	補助漢字字形（JIS X 0212-1990）	JIS X 0213:2004 字形がデフォルトとしてエンコードされている場合に、JIS X 0212-1990 字形（「補助漢字」ともいいます）を利用。
hwid	半角	プロポーショナル幅の、または em の半分以外の等幅のグリフを、em の半分（en）の幅のグリフへ置き換え。
ital	イタリック	ローマン体のグリフを、対応するイタリック体のグリフへ置き換え。

表 6.7 日本語・中国語・韓国語テキスト用対応 OpenType レイアウト機能一覧（これに加えて表 6.1 に、一般的利用のための OpenType レイアウト機能を挙げています）

キーワード	名前	説明
jp04	JIS2004 字形	（nlck 機能のサブセット）JIS X 0213:2004 グリフを利用。
jp78	JIS78 字形	デフォルト（JIS90）の和文グリフを、対応する JIS C 6226-1978（JIS78）の字形へ置き換え。
jp83	JIS83 字形	デフォルト（JIS90）の和文グリフを、対応する JIS X 0208-1983（JIS83）の字形へ置き換え。
jp90	JIS90 字形	JIS78 または JIS83 の和文グリフを、対応する JIS X 0208-1990（JIS90）の字形へ置き換え。
locl	ローカライズ字形	グリフのローカライズされた字形で、デフォルト字形を置き換えることを可能にします。この機能は script・language オプションを必要とします。
nalt	修飾字形	デフォルトのグリフを、さまざまな表記字形へ置き換え（白丸囲み・黒丸囲み・四角囲み・括弧付き・菱形囲み・角丸四角囲み等）。
nlck	国語審議会漢字字形	日本の国語審議会（NLC）が多くの JIS キャラクタに対して 2000 年に制定した新たなグリフ形状を利用。
pkna	プロポーショナル仮名	等幅（半角または全角）に設定された仮名および仮名関連のグリフを、プロポーショナルなグリフへ置き換え。
pwid	プロポーショナルグリフ	等幅（通常、全角または em の半分）に設定されたグリフを、プロポーショナルな字送りのグリフへ置き換え。
qwid	4 分の 1 幅	他の幅のグリフを、em の 4 分の 1（en の半分）の幅に設定されたグリフへ置き換え。
ruby	ルビ表記字形	デフォルトの仮名グリフを、（通常、上付きにされた）ルビ用にデザインされた、より小さいグリフへ置き換え。
smpl	簡体字	和文漢字または中文繁体字を、対応する簡体字へ置き換え。
tnam	名前旧字体	和文新字体を、対応する旧字体へ置き換え。これは trad 機能と等価ですが、ただし個人の名前における使用が適切であると認められる旧字体に限られます。
trad	旧字体	和文漢字または中文簡体字を、対応する旧字体 / 繁体字へ置き換え。
twid	3 分の 1 幅	他の幅のグリフを、em の 3 分の 1 の幅に設定されたグリフへ置き換え。
vert	縦組み	デフォルトの字形を、縦書き用に調整された異体字へ置き換え。
vkna	縦組み用仮名	標準の仮名を、縦書き専用に特にデザインされた字形へ置き換え。
vrt2	縦組み異体字・回転	（vrt2 機能のサブセットである vert 機能を上書きします）等幅（半角・全角・4 分の 1 幅のいずれか）またはプロポーショナル幅（多くは欧文またはカタカナ）のグリフを、縦書きに適した字形へ置き換え（すなわち、90° 時計回りに回転）。

7 画像・PDF ページの取り込み

PDFlib は、ラスタ画像や既存 PDF 文書内ページを取り込んでページ上に配置するためのさまざまな機能を提供します。この章では、ラスタ画像の取り扱いや既存 PDF 文書内ページの取り込みに関する詳細を解説します。また、画像や PDF ページを出力ページ上に配置する方法については 188 ページ「7.3 画像と取り込み PDF ページの配置」を参照してください。

クックブック **画像の諸側面に関するコードサンプルが PDFlib クックブックの** images **カテゴリにあります。**

7.1 ラスタ画像の取り込み

7.1.1 基本的な画像処理

PDFlib でラスタ画像を貼り付けるのは簡単です。まず、画像パラメタ群の簡単な解析を行う PDFlib 関数で画像を開く必要があります。その *PDF_load_image()* 関数は、画像記述子の役割をするハンドルを返します。このハンドルを使って *PDF_fit_image()* を呼び出すことができるので、その際に位置・縮尺パラメタも与えます。具体的には以下のようになります。

```
image = p.load_image("auto", "image.jpg", "");
if (image == -1)
        throw new Exception("エラー : " + p.get_errmsg());

p.fit_image(image, 0.0, 0.0, "");
p.close_image(image);
```

PDF_fit_image() の最後の引数は、画像の位置・縮尺・回転を指定できるさまざまなオプションを持たせることのできるオプションリストです。このオプション群については 188 ページ「7.3 画像と取り込み PDF ページの配置」で詳しく説明します。

クックブック **完全なコードサンプルがクックブックの** images/starter_image **トピックにあります。**

画像データの再利用　ラスタ画像の反復利用のための重要な PDF 最適化技法に PDFlib は対応しています。複数のページに同じロゴや背景を使うレイアウトを考えてみましょう。そのような場合には画像データ本体は一度しか PDF 内に取り込まずに、その画像を使う各ページからそこへの参照だけを生成することが可能です。画像ファイルを一度読み込んでおいて、各ページにロゴや背景を配置するたびに *PDF_fit_image()* を呼び出せばよいのです。複数のページにその画像を配置したり、同じ画像でも貼り付けるたびに縮尺を変えたりすることもできます（画像を閉じていない限り）。画像の容量や使用回数によってはこの技法は顕著な容量節減をもたらすでしょう。

拡縮と dpi 計算　取り込んだ画像のピクセル数を PDFlib が変えることはありません。拡縮すると画像のピクセルは膨らんだり縮んだりしますが、ダウンサンプリングが行われることはありません（画像のピクセル数はいつも同じままです）。縮尺が 1 ならば 1 ピクセルの大きさはユーザー座標の 1 単位と同じになります。いいかえれば、ユーザー座標系が拡縮されていなければ画像はその元の解像度で（その画像が解像度情報を持っていなければ 72 dpi で）取り込まれます（デフォルトでは 1 インチが 72 単位なので）。

クックブック **完全なコードサンプルがクックブックの images/image_dimensions トピックにあります。そこに、画像の寸法を得る方法や、それをさまざまな大きさで貼る方法を示してあります。**

取り込み画像の色空間 *PDF_load_image()* で与えられたオプションに従って ICC プロファイルの追加・削除をしたりスポットカラーを適用したりする場合を除いて、PDFlib は一般に、取り込んだ画像の元の色空間を保持しようとします。しかし、まれにこれが不可能な組み合わせがあります。その一例として、TIFFにおけるYCbCrはRGBに変換されます。

PDFlib は、RGB と CMYK との間の変換は一切行いません。そのような変換が必要なときは、画像を PDFlib に取り込む前にその画像データに適用しておく必要があります。

複数ページ画像 PDFlib は、複数の画像を持つ GIF・TIFF・JBIG2 画像に対応しています。これを複数ページ画像ファイルともいいます。複数ページ画像を利用するには、*PDF_load_image()* で *page* オプションを用います：

```
image = p.load_image("tiff", filename, "page=2");
```

この *page* オプションは、複数画像ファイルが利用されることを示し、利用したい画像の番号を指定します。先頭画像の番号は 1 です。このオプションは、*PDF_load_image()* がファイル内でもう画像が得られないことを示す -1 を返すまで値を増やしていくことができます。

クックブック **複数画像TIFFファイル内のすべての画像を複数ページPDFファイルへ変換する完全なコードサンプルがクックブックの images/multi_page_tiff トピックにあります。**

インライン画像 再利用可能画像は Image XObject として PDF 出力へ書き出されますが、これに対してインライン画像は各コンテントストリーム（ページかパターンかテンプレートかグリフ定義）の中に直接書き込まれます。これにより若干の容量が節約できますが、PDF リファレンス中の推奨に従い、容量の小さな画像データ（4 KB まで）での利用に留めるべきです。インライン画像の主用途は Type 3 フォントのビットマップグリフ定義です。

インライン画像を生成するには *PDF_load_image()* インタフェースで *inline* オプションを与えます。インライン画像の再利用はできません。すなわちそのハンドルを画像ハンドルとして呼び出しに与えてはいけません。そのため、*inline* オプションが与えられると *PDF_load_image()* の内部動作は以下と等価になります。

```
p.fit_image(image, 0, 0, "");
p.close_image(image);
```

インライン画像は、*imagetype=ccitt・jpeg・raw* でのみ対応しています。他の種類の画像では、*inline* オプションは静かに無視されます。

OPI 対応 画像を読み込む際には、OPI（Open Prepress Interface）バージョン 1.3 または 2.0 に従った追加の情報を、*PDF_load_image()* への呼び出しで与えることができます。PDFlib は、すべての標準 OPI 1.3 または 2.0 PostScript コメント（対応する PDF キーワードではありません！）をオプションとして受け付け、その与えられた OPI 情報を、一切変更を加えずに生成 PDF 出力へパススルーします。下記の例では、OPI 情報を画像に添付しています：

```
String  optlist13 =
        "OPI-1.3 { ALDImageFilename bigfile.tif " +
```

```
        "ALDImageDimensions {400 561} " +
        "ALDImageCropRect {10 10 390 550} " +
        "ALDImagePosition {10 10  10 540  390 540  390 10} }";

image = p.load_image("tiff", filename, optlist13);
```

注 Helios EtherShare に内蔵されているものなどいくつかの OPI サーバは、PDF の画像 XObject に対する OPI 処理を正しく実装していません。PDFlib はデフォルトでは画像 XObject を生成しますが、このような場合には、*PDF_load_image()* に *template* オプションを与えることでフォーム XObject の生成を強制することができます。

画像内の XMP メタデータ　画像ファイルは XMP メタデータを含んでいることがあります。PDFlib はデフォルトで、TIFF・JPEG・JPEG 2000 形式の画像に対する画像メタデータに対応します。この XMP メタデータは、出力 PDF 文書内の生成画像に添付あれます。

画像のメタデータは通常、保持しておくことが推奨されますが、出力ファイルサイズを抑えるために無効化することも可能です。*PDF_load_image()* の下記のオプションを用いると、取り込み画像内に XMP メタデータがあったときにはそれを破棄します：

```
metadata={keepxmp=false}
```

無効な XMP があると画像は読み込まれませんので、上記のオプションリストは、無効な XMP を回避するために利用することもできます。

7.1.2 対応画像ファイル形式

以下に述べる画像ファイル形式を PDFlib は取り扱います。デフォルトでは PDFlib は、圧縮された画像データについては可能な限り無変更のまま PDF 出力に送ります。なぜなら、よく利用される画像ファイル形式で用いられる圧縮方式の多くに PDF は標準対応しているからです。この技法（以下の解説では**「パススルーモード」**と呼びます）では画像データを展開してまた再圧縮する必要がないため、画像の取り込みが非常に速くなります。しかしこのモードでは圧縮画像データの整合性を PDFlib が検査することはできません。画像データが不完全だったり壊れていたりすると、PDF 文書を Acrobat で利用する時にエラーや警告のメッセージが出ます（たとえば「*Read less image data than expected*」）。パススルーモードは、*PDF_load_image()* の *passthrough* オプションで制御することができます。

画像ファイルの取り込みが不成功の時は *PDF_load_image()* はエラーコードを返します。その画像の失敗についてもっと詳しく知る必要がある場合には、*get_errmsg()* を呼び出して詳しいエラーメッセージを取得します。

PNG 画像　あらゆる種類の PNG 画像（ISO 15948）に PDFlib は対応しています。PNG 画像は多くの場合パススルーモードで処理されます。PNG 画像が透過情報を持っている場合、その透過は生成 PDF 内で保たれます（180 ページ「7.1.4 画像マスクと透過」参照）。

JPEG 画像　JPEG 画像（ISO 10918-1）は決して圧縮されません。ただしその種類によっては、Acrobat における適切な表示のために変換が必要な場合があります。PDFlib は、ある種の JPEG 画像に対して自動的に変換を適用します。しかし変換の制御は、*PDF_load_image()* の *passthrough* オプションで行うこともできます。変換は、ピクセルの数または画像の色を変えず、かつ、圧縮 / 解凍されたという不自然さを見た目に一切もたらしません。以下の種類の JPEG 画像圧縮に PDFlib は対応しています。

- グレースケール・RGB（通常は YCbCr エンコードされている）・CMYK カラー。

- ベースライン JPEG 圧縮。JPEG 画像の圧倒的大多数はこの種類です。
- プログレッシブ JPEG 圧縮。

JPEG 画像は何種類かのファイル形式に納めることができます。よく利用されるあらゆる JPEG ファイル形式に PDFlib は対応しており、また、以下の種類の中から解像度情報を読み取ります。

- JFIF。さまざまな画像処理アプリケーションによって生成されます。
- Adobe Photoshop などの Adobe アプリケーションによって出力される JPEG ファイル。Photoshop 由来の CMYK の JPEG ファイルについては、正しく処理するために必要な特別の動作を PDFlib は行います。また PDFlib は、Adobe Photoshop で作成された JPEG 画像からクリッピングパスを読み取ります。

多くのデジタルカメラが生成する EXIF 形式の JPEG 画像は、sRGB 画像として扱われますので、画像には sRGB ICC プロファイルが添付されます。ただし、***honoriccprofile*** オプションを ***false*** にしているか、あるいは画像に ***iccprofile*** オプションで他の ICC プロファイルを割り当てている場合はこの限りではありません。

JPEG 2000 画像　JPEG 2000 画像（ISO 15444-2）には PDF 1.5 以上が必要で、つねにパススルーモードで処理されます。PDFlib による JPEG 2000 画像への対応は以下のとおりです。

- JP2・JPX ベースライン画像（通常 ****.jp2*** または ****.jpf***）に対応しています。ただし下記の色空間条件に従う必要があります。すべての有効な色深度値に対応しています。
  次の色空間に対応しています：sRGB・sRGB グレー・ROMM-RGB・sYCC・e-sRGB・e-sYCC・CIELab・ICC ベースの色空間群（制限された、または完全な ICC プロファイル）・CMYK。PDFlib は、JPEG 2000 画像ファイル内の元の色空間に変更を加えません。
- ソフトマスクを持つ画像は、***mask*** オプションを用いて、他の画像にそのマスクを適用させることができます。
- 外部 ICC プロファイルは JPEG 2000 画像には適用できません。すなわち、***iccprofile*** オプションは用いることができません。JPEG 2000 画像が持つ ICC プロファイルはつねに保持されます。すなわち、***honoriccprofile*** オプションはつねに ***true*** です。

JBIG2 画像　PDFlib は、単ページ・複数ページの JBIG2 画像（ISO 14492）に対応しています。JBIG2 画像はつねに単色ピクセルデータを内容として持ち、PDF 1.4 以上を必要とします。

JBIG2 圧縮の性質から、複数ページ JBIG2 ストリーム内のいくつかのページが同一のグローバルセグメントを参照している場合があります。複数ページ JBIG2 ストリームの 1 つないし複数のページが変換される際、グローバルセグメントは、生成 PDF 画像間で共用できます。***PDF_load_image()*** への呼び出しは互いに独立ですので、あらかじめ PDFlib に、同一 JBIG2 ストリームから複数ページを変換すると知らせておく必要があります。これは下記のように行います：

- 先頭ページを読み込む際に、すべてのグローバルセグメントを PDF へ複製します。***PDF_load_image()*** に対して下記のオプションリストを用います：

```
page=1 copyglobals=all
```

- 同一 JBIG2 ストリームから、以降のページを読み込む際に、ページ 1 に対する画像ハンドル ***<N>*** を与えて、ページ 1 とともに複製されたグローバルセグメントへの参照を PDFlib が生成できるようにする必要があります。***PDF_load_image()*** に対して下記のオプションリストを用います：

```
page=2 imagehandle=<N>
```

クライアントアプリケーション側では必ず、同一 JBIG2 画像ストリームから抽出された複数ページに対してのみこの *copyglobals/imagehandle* 機構が適用されるようにする必要があります。*copyglobals* オプションがない場合は、PDFlib は自動的にカレントページに対する必要データをすべて複製します。

GIF 画像　PDFlib は、ピクセルデータがインタレースされているものとされていないものの両方の、あらゆるパレットサイズの、あらゆる種類の GIF（具体的には GIF 87a・89a）に対応しています。GIF 画像はつねに Flate 圧縮で再圧縮されます。

TIFF 画像　PDFlib は、ほとんどすべての種類の TIFF 画像を取り込みます：

- 圧縮方式：非圧縮・CCITT（グループ 3・グループ 4・RLE）・ZIP（＝ Flate）・PackBits（＝ RunLength）・LZW・新旧方式 JPEG に加え、いくつかのまれな圧縮方式。
- 色空間：単色・グレースケール・RGB・CMYK・CIELab・YCbCr 画像。
- 色深度は、色要素あたり 1・2・4・8・16 ビットのいずれかでなければなりません。16 ビット画像は PDF 1.5 を必要とします。

画像を取り込む際、下記の TIFF 機能は処理されます：

- 複数の画像を含んだ TIFF ファイル（176 ページ「 複数ページ画像」参照）。TIFF ファイル内の画像を選ぶには *page* オプションを用います。
- アルファチャンネルまたはマスク（180 ページ「7.1.4　画像マスクと透過」参照）に、*ignoremask* オプションが設定されていない限り従います。*alphachannelname* オプションで明示的にアルファチャンネルを選ぶこともできます。
- PDFlib は、Adobe Photoshop や互換プログラムで作成された TIFF 画像内のクリッピングパスに、*ignoreclippingpath* オプションが設定されていない限り従います。
- PDFlib は、TIFF 画像内の ICC プロファイルに、*honoriccprofile* オプションが *false* に設定されていない限り従います。
- 画像の望ましい向きを指定した Orientation タグに従います。これを無視する（多くのアプリケーション同様）には *ignoreorientation* オプションを用います。

いくつかの TIFF 機能（スポットカラーなど）や複数機能の組み合わせには対応していません。

BMP 画像　BMP 画像はパススルーモードでは処理できません。以下の種類の BMP 画像に PDFlib は対応しています。

- BMP バージョン 2・3。
- 色深度はコンポーネントあたり 1･4･8 ビット。3 × 8 ＝ 24 ビットの TrueColor を含みます。16 ビット画像は 5+5+5 プラス未使用 1 ビットとして扱われます。32 ビット画像は 3 × 8 ビット画像として扱われます（残りの 8 ビットは無視されます）。
- 単色か RGB カラー（インデックス・直接）。
- 非圧縮と 4 ビット・8 ビット RLE 圧縮。
- ピクセルがボトムアップ順で格納されている場合 PDFlib は画像を反転させません（この BMP の機能はめったに使われず、アプリケーションによって異なった解釈をされます）。

CCITT 画像　グループ 3・グループ 4 の FAX 圧縮された画像データはつねにパススルーモードで処理されます。この形式は実は生の CCITT 圧縮された画像データという意味で

あり、CCITT 圧縮を用いた TIFF ファイルという意味ではないことに注意してください。生の CCITT 圧縮画像ファイルはエンドユーザーアプリケーションではふつう対応しておらず、これを生成できるのは FAX 関係のソフトウェアだけです。PDFlib は CCITT 画像を分析する能力を持たないので、すべての画像関連パラメタはクライアントから *PDF_load_image()* に渡す必要があります。

RAW データ　非圧縮の（RAW）画像データはいくつかの特殊な応用では有用でしょう。画像の種類は色要素の数から推測できます。1 要素ならグレースケール画像、3 要素なら RGB 画像、4 要素なら CMYK 画像であることをそれぞれ示唆しています。

7.1.3 クリッピングパス

PDFlib は、Adobe Photoshop で作成された TIFF・JPEG 画像の中のクリッピングパスに対応しています。1 つの画像ファイルには、複数の名前付きパスを含む場合があります。*PDF_load_image()* の *clippingpathname* オプションを使えば、名前付きパスのうちの 1 つを選ぶことができ、それがクリッピングパスとして使われます。すると画像は、クリッピングパスの内部だけが可視となり、それ以外の部分は不可視になります。これは背景と前景を分離したり、画像の不要部分を除去したりするのに有用です。

あるいは、画像ファイルはデフォルトクリッピングパスを含む場合があります。PDFlib は、画像ファイル内にクリッピングパスを見つけた場合、それを自動的に画像に適用します（図 7.1 参照）。デフォルトクリッピングパスが適用されないようにするには、*PDF_load_image()* で *honorclippingpath* オプションを *false* に設定します。同じ画像のインスタンスが複数あって、しかもそのうち一部のインスタンスにしかクリッピングパスを適用したくないときは、*PDF_fit_image()* に *ignoreclippingpath* オプションを与えてクリッピングパスを無効にすることができます。クリッピングパスが適用されると、画像の配置やはめ込みに関するすべての計算は、切り抜かれた画像の外接枠をもとに行われます。

クックブック　**完全なコードサンプルがクックブックの `images/integrated_clipping_path` トピックにあります。**

7.1.4 画像マスクと透過

3 種類の画像内透過情報に PDFlib は対応しています：アルファチャンネルによる内在透過、外在透過、画像マスクです。

図 7.1
クリッピングパスを利用して前景と背景を分離

アルファチャンネルによる内在透過　ラスタ画像は、部分的に透明にすることもできます。すなわち、画像を透かして背景が見えるようにすることが可能です。これはたとえば、画像の背景を無視して、前景の人物や物体だけを見せたいときに有用です。透過情報は、別途のアルファチャンネル内に、あるいは（パレットベースの画像の場合）透過パレット項目として格納できます。透過画像は、PDF/A-1・PDF/X-1・PDF/X-3 では許されていません。PDFlib は、下記の画像形式の透過情報を処理します：

- GIF 画像ファイルは、1 個の透過色値（パレット項目）を持つことができ、PDFlib はそれに従います。
- TIFF 画像は、1 個の関連づけられたアルファチャンネルを含むことができ、PDFlib はそれに従います。あるいは TIFF 画像は、関連づけられていない、名前で指定されるチャンネルを任意の数含むこともできます。これらのチャンネルは、透過などの情報を伝達するために利用することができます。関連づけられていないチャンネルが TIFF 画像内に見つかったときは、PDFlib はデフォルトでは先頭チャンネルをアルファチャンネルとして用います。しかし、関連づけられていないアルファチャンネルを、その名前を与えることで明示的に選択することも可能です：

```
image = p.load_image("tiff", filename, "alphachannelname={apple}");
```

- PNG 画像は、1 個の関連づけられたアルファチャンネルを含むことができ、PDFlib はそれを自動的に用います。
- PNG 画像は、完全なアルファチャンネルではなく、1 個の透過色値を持つこともでき、PDFlib はそれに従います。複数の色値がアルファ値つきで与えられている場合、50 パーセント未満のアルファ値を持つもののうちの初めの 1 個が用いられます。

注 Photoshop では、完全なアルファチャンネルに加えて、固有形式で透過背景を作成することができます。しかし PDFlib はこの形式を理解しません。このような透過画像を PDFlib で利用するには、Photoshop でこれを TIFF ファイル形式で保存し、その際に「TIFF オプション」ダイアログボックスで「透明部分を保持」を選択します。

場合によっては、暗黙的な透過が画像ファイルに含まれていても、それをすべて無視したいこともあります。PDFlib の透過対応は、画像を読み込む際に *ignoremask* オプションで無効化することが可能です：

```
image = p.load_image("tiff", filename, "ignoremask");
```

外在透過　外在の場合には、2 つの段階が必要で、どちらも画像操作を伴います。第一に、マスクとして後で使うためのグレースケール画像を 1 つ用意する必要があります。そのためにはマスク画像を読み込みます。マスクを構築するには、下記の種類の画像を使うことができます：

- PNG 画像
- TIFF 画像：マルチストリップ画像を避けるため、*PDF_load_image()* で *nopassthrough* オプションを推奨します。
- RAW 画像データ

マスク内でピクセル値が 0（ゼロ）の部分の画像は塗られ、ピクセル値が 0 でない部分は背景が透けて見えます。ピクセルあたりのビット数が 1 より大きい場合、中間値は前景画像を背景にブレンドさせて透過効果を生みます。

第二段階として、そのマスクを別の画像に *masked* オプションで適用します：

```
mask = p.load_image("png", maskfilename, "");
if (mask == -1)
        throw new Exception("エラー：" + p.get_errmsg());
```

```
String optlist = "masked=" + mask;
image = p.load_image(type, filename, optlist)
if (image == -1)
        throw new Exception("エラー : " + p.get_errmsg());

p.fit_image(image, x, y, "");
```

画像とマスクは縦横のピクセル数が違っている可能性もあります。マスクは画像サイズに合わせて自動的に拡縮されます。

注 場合によってはPDFlibは、マルチストリップのTIFF画像をPDF上の複数の画像に変換すりことがあり、その場合、マスクはその各画像に対して独立にかかることになります。通常このようなことは意図されていないので、この種の画像はマスクとしてもマスク対象画像としても拒否されます。また、内在の場合と外在の場合とを互いに混在させないことが重要です。つまり、透過色値を持つ画像をマスクとして使ってはいけません。

注 マスクは、背景画像と同じ向きを持つ必要があります。そうでないとそれは拒否されます。向きは画像ファイル形式やその他の要因に依存しますので、検出が困難です。このため、マスクと画像の両方について、同じファイル形式と作成ソフトウェアを用いることを推奨します。

クックブック 完全なコードサンプルがクックブックの images/image_mask トピックにあります。

画像マスクとソフトマスク 画像マスクとはビット深度が 1 の画像（ビットマップ）であり、ゼロのビットが透過として扱われます。ページの既存内容が画像内の透過部分を通して見えます。1 のビットのピクセルはカレント塗り色で着色されます。

ソフトマスクは、画像マスクの概念を、複数ビットのマスクへ一般化したものです。これは画像を、何らかの既存の背景に溶け込ませます。PDFlib は、あらゆる種類のシングルチャンネル（グレースケール）画像をソフトマスクとして受け付けます。なお、マスクとして使えるのは真のグレースケールの画像だけであり、インデックス付きの（パレットベースの）色は使えません。これは画像マスクと同じように利用できます。画像マスクとして使えるのは以下の種類の画像です：

- PNG 画像
- TIFF 画像（シングルストリップでもマルチストリップでも）
- JPEG 画像（ソフトマスクとしてのみ。後述）
- BMP。BMP 画像は他の種類の画像と向きが違うことに注意してください。そのため BMP 画像はマスクとして使うにはまず ***x*** 軸を軸として反転させる必要があります。
- RAW 画像データ

画像マスクはただ ***mask*** オプションをつければ開くことができ、希望の塗り色を設定した後にページ上に配置することができます。

```
mask = p.load_image("tiff", maskfilename, "mask");
p.setcolor("fill", "rgb", 1.0, 0.0, 0.0, 0.0);
if (mask != -1)
{
        p.fit_image(mask, x, y, "");
}
```

ゼロのビットのピクセルを透過にせずに画像を着色したい場合は ***colorize*** オプションを使う必要があります（183 ページ「7.1.5 画像の着色」参照）。

7.1.5 画像の着色

画像マスクでは画像の不透明部分が着色されますが、類似機能としてスポットカラーによる画像の着色に PDFlib は対応しています。この機能は、単色かグレースケールの画像で動作します。

RGB パレットを持つ画像に関しては、着色が意味を持つのは、そのパレットがグレー値だけを持ち、パレットインデックスがグレー値と等価な場合だけです。しかし PDFlib はこの条件に関する検査は行いません。

画像をスポットカラーで着色するには、画像を読み込む時に *colorize* オプションを与える必要があり、それとともにそのスポットカラーハンドルを与える必要があります。スポットカラーハンドルはあらかじめ*PDF_makespotcolor( )*で取得しておく必要があります。

```
p.setcolor("fillstroke", "cmyk", 1, .79, 0, 0);
spot = p.makespotcolor("PANTONE Reflex Blue CV");

String optlist = "colorize=" + spot;
image = p.load_image("tiff", "image.tif", optlist);
if (image != -1)
{
        p.fit_image(image, x, y, "");
}
```

7.2 PDI で PDF ページを取り込み

注 この節で解説するすべての関数は PDFlib+PDI を必要とします。PDF 取り込みライブラリ（PDI）は PDFlib 基本製品には含まれていません。PDI は PDFlib のすべてのコンパイル済み版に内蔵されていますが、それを利用するには PDI（または PPS。PPS は PDI を含んでいます）のためのライセンスキーが必要です。

7.2.1 PDI の機能と用途

PDFlib+PDI ライブラリを利用することで、既存 PDF 文書内のページを取り込むことができます。PDI は PDF ファイル形式に対するパーサを持っていて、既存 PDF 文書内のページを PDFlib で簡単に利用できるようにします。概念的に、取り込まれた PDF ページは取り込まれた TIFF や PNG のようなラスタ画像と同様に扱われます。すなわち、PDF 文書を開き、取り込むページを選び、それを出力ページ上に配置し、取り込んだページに PDFlib の変形関数を適用して並行移動・拡縮・回転・斜形化させます。取り込んだページは簡単に新しい内容と組み合わせることができます。そのためには、取り込んだ PDF ページを出力ページ上に配置した後に PDFlib の任意のテキスト・グラフィック関数を使えばよいのです（取り込んだページは新しい内容の背景となると捉えられます）。PDFlib と PDI を活用すれば以下のような課題が簡単に実現できます。

- 複数の PDF 文書内の複数のページを重ね合わせ（たとえば、既存文書に便箋を追加して印刷済み用紙のようにする）。
- 既存文書内に PDF 広告を配置。
- PDF のページの表示領域を切り抜いて、見せたくない要素（トンボなど）を取り除く。または、ページの拡縮。
- 複数ページを 1 枚の紙に印刷。
- 複数の PDF/X か PDF/A の文書を処理して、新しい PDF/X か PDF/A のファイルを作る。
- ファイルの PDF/X か PDF/A の出力インテントをコピー。
- 既存 PDF のページにテキスト（ヘッダ・フッタ・スタンプ・ページ番号など）や画像（企業ロゴなど）を追加。
- 入力文書内の全ページを出力文書にコピーして、各ページにバーコードを配置。
- pCOS インタフェースを使って、PDF 文書の任意のプロパティを取得 (247 ページ「9 章 pCOS インタフェース」参照)。

PDF の背景ページを配置してそこに動的なデータを入れ込みたい場合には（たとえばメールのマージや、Web 上のパーソナライズされた PDF 文書や、フォーム記入など）、PDI を PDFlib ブロックとあわせて利用されることをおすすめします （279 ページ「11 章 PPS と PDFlib Block Plugin」参照）。

7.2.2 PDFlib で PDI 関数を利用

クックブック PDF 取り込みの諸側面に関するコードサンプルが PDFlib クックブックの pdf_import カテゴリにあります。

一般的考察 重要な注意点として、PDI は実際のページ内容だけを取り込みますので、取り込む PDF 文書内に存在しているかもしれないインタラクティブ機能（たとえばサウンド・ムービー・ファイル添付・ハイパーテキストリンク・フォームフィールド・JavaScript・しおり・サムネール・ノート）は一切取り込みません。こうしたインタラクティブ機能は各 PDFlib 関数で生成することができます。PDFlib ブロックもページ取り込み時には無視されます。タグ付き PDF 内の文書構造も、文書を取り込む際には失われます。

取り込んだページ内の個々の要素を他の PDFlib 関数で再利用することはできません。たとえば、取り込んだ文書内のフォントを他の何らかの内容のために再利用することは不可能です。必要なフォントはすべて PDFlib 内で設定する必要があります。取り込んだ複数の文書が同じフォントの埋め込みフォントデータをそれぞれ持っていたとしても、フォントデータの重複を PDI は解消させません。他方、取り込んだ PDF 内で欠けているフォントがあれば、生成される PDF 出力ファイル内でもそのフォントは欠けたままです。最適な方法としては、取り込む文書はなるべく開いたままにしておいたほうが、同じフォントが何度も出力文書内に埋め込まれずにすみます。

取り込んだ PDF 文書内の色に対して PDFlib+PDI はまったく変更を加えません。たとえば、PDF が ICC カラープロファイルを持っていればそれは出力文書内でも保持されます。

取り込んだ PDF のページを出力ページ上に配置するために PDFlib+PDI はテンプレート機能（フォーム XObject）を利用します。他の PDF 文書から取り込まれたページを含む文書をさらに PDFlib+PDI で処理することもできます。

PDF ページ取り込みのためのコード　既存 PDF 文書内のページの取り込みは非常に単純なコード構造で実現可能です。次のコードは、既存文書のページを開き、そのページ内容を出力 PDF 文書内にコピーします（出力 PDF 文書はあらかじめ開いている必要があります）。

```
int     doc, page, pageno = 1;
String  filename = "input.pdf";

if (p.begin_document(outfilename, "") == -1) {...}
...

doc = p.open_pdi_document(infilename, "");
if (doc == -1)
        throw new Exception("エラー : " + p.get_errmsg());

page = p.open_pdi_page(doc, pageno, "");
if (page == -1)
        throw new Exception("エラー : " + p.get_errmsg());

/* ダミーのページサイズ。この後adjustpageオプションによって変更される */
p.begin_page_ext(20, 20, "");
p.fit_pdi_page(page, 0, 0, "adjustpage");
p.close_pdi_page(page);
...ページに内容を追加するPDFlib関数群...
p.end_page_ext("");
p.close_pdi_document(doc);
```

PDF_fit_pdi_page() の最後の引数は、取り込むページの位置指定・拡縮・回転を指示するさまざまなオプションを持ちうるオプションリストです。このオプションについては詳しくは 188 ページ「7.3 画像と取り込み PDF ページの配置」で解説しています。

取り込んだ PDF ページの寸法　取り込んだ PDF ページは取り込んだラスタ画像と同様に扱われ、*PDF_fit_pdi_page()* を用いて出力ページ上に配置することができます。デフォルトでは、Acrobat での表示とまったく同じ形で PDI はページを取り込みます。とりわけ次のような動作をします：

- クロッピングは保持されます（技術的にいえば、CropBox が存在する場合には、PDI は MediaBox よりも CropBox を優先採用します。69 ページ「3.2.2 ページサイズ」参照）。
- ページに適用されている回転は保持されます。

cloneboxes オプションは PDFlib+PDI に対して、取り込みページのすべてのページ枠を生成出力ページへ複製するよう指示し、その結果、すべてのページサイズ情報を複製されます。

あるいは、*pdiusebox* オプションを用いて明示的に、ページの MediaBox・CropBox・TrimBox・ArtBox 項目のうちのいずれかを（もしあれば）用いて取り込みページのサイズを決めるよう PDI に指示することもできます。

レイヤーを持つ PDF ページの取り込み Acrobat 6（PDF 1.5）ではレイヤー機能が導入されました（技術的には「オプショナルコンテント」といいます）。ファイルに存在しているかもしれないレイヤー情報を PDI は一切無視します。取り込んだページ内のすべてのレイヤーは、不可視レイヤーを含めて、生成出力内では可視になります。

GeoPDF を PDI で取り込み GeoPDF を PDI で取り込む際には、地理空間情報は、それが下記のいずれかの方式で作成されていれば保持されます（画像ベースの地理空間参照）：

- PDFlib で *PDF_load_image()* の *georeference* オプションで
- Acrobat で地理空間情報を持つ画像を取り込んで。

地理空間情報はページを取り込んだ後、それが下記のいずれかの方式で作成されていた場合には失われます（ページベースの地理空間参照）：

- PDFlib で *PDF_begin/end_page_ext()* の *viewports* オプションで
- Acrobat で手作業で PDF ページを地理登録して。

OPI 情報を持つ PDF ページの取り込み 入力 PDF 内の存在する OPI 情報は出力内で変更されないまま保持されます。

複数の取り込み文書間での最適化 PDFlib 自体は、高度に最適化した PDF 出力を生成しますが、取り込んだ PDF にはもしかすると冗長なデータ構造があって、最適化の余地があるかもしれません。さらに、取り込む PDF がもし複数であれば、その複数のファイルが等価なリソース（ファイル等）を含む場合には、出力するファイルのサイズはふくれあがる可能性があります。このような場面では、*PDF_begin_document()* の *optimize* オプションを使うことができます。これは取り込んだファイル内の冗長なオブジェクトを検知して、生成する出力の体裁や品質をそこなうことなくそうしたオブジェクトを削除します。

7.2.3 受け入れ可能な PDF 文書

一般に、Acrobat で開くことのできるあらゆる種類の PDF 文書を PDI は適切に処理できます。PDF のバージョン番号や、ファイル内で使用されている機能には左右されません。暗号化文書（すなわち権限設定やパスワードを持つファイル）内のページを取り込むにはそのマスターパスワードを与える必要があります。

PDI は破損 PDF のための修復モードを実装しており、ある種の破損文書も開くことができます。しかし、まれに PDF 文書や文書内の特定ページが PDI によって拒否されることがあります。

PDF 文書やページがうまく取り込めなかった場合 *PDF_open_pdi_document()* と *PDF_open_pdi_page()* はエラーコードを返します。失敗についてもっと詳しく知りたい場合には、*PDF_get_errmsg()* で原因を取得することができます。または、*errorpolicy* オプションかパラメタを *true* に設定しておけば、文書が開けなかったときに例外が発生するようになります。

下記の種類の PDF 文書は、デフォルトでは拒絶されますが、*infomode* オプションを *true* に設定すれば、pCOS で情報を取得するために開くことはできます。

- カレントで生成中のPDF出力文書よりも高いPDFバージョン番号を用いているPDF文書は、PDI で取り込めません。理由は、高いバージョン番号の PDF を取り込んだ後では、求められている PDF バージョンに出力が本当に準拠するかどうか、もはや PDFlib は確信を持てないからです。解決策 : 出力 PDF のバージョンを *PDF_begin_document()* の *compatibility* オプションを使って必要な水準に設定します。
  PDF 1.7ext 3（Acrobat 9）・*PDF 1.7ext8*（Acrobat X）文書は、PDI に関する限り PDF 1.7 と互換です（ただし Acrobat X の暗号化にはまだ対応していません）。
  PDF/A モードでは、PDF バージョンヘッダは PDF/A 内では無視されなければならないので、入力 PDF バージョン番号は無視されます。
- 暗号化された PDF 文書でそのパスワードがないもの（*infomode* 規則に対する例外 : Distiller の設定「オブジェクトレベルの圧縮 : 最高」を用いて作成された PDF 1.6 文書。これは情報モードでも開くことができません）。
- タグ付き PDF で *PDF_begin_document()* の *tagged* オプションが *true* の場合。
- カレント出力文書のPDF/AかPDF/Xのレベルと非互換なPDF/AかPDF/Xの文書（例: PDF/A-1a を PDF/A-1b 文書へ取り込もうとした）。詳しくは 258 ページ「10.3.4 PDI による PDF/X 文書の取り込み」・265 ページ「10.4.3 PDF/A 文書を PDI で取り込み」を参照してください。

7.3 画像と取り込み PDF ページの配置

ラスタ画像とテンプレートを配置するための関数 *PDF_fit_image()* と、取り込み PDF ページを配置するための *PDF_fit_pdi_page()* は、ページ上への配置を制御するためのさまざまなオプションを提供しています。この節では、いくつかの代表的な応用作業を見てみることによって、もっとも重要ないくつかのオプションの動作を示します。すべてのオプションの完全な一覧と説明については、***PDFlib API リファレンス***を参照してください。

ラスタ画像の埋め込みは、PDFlib なら簡単に実現できます。画像ファイルはまず、*PDF_load_image()* で読み込む必要があります。この関数は画像ハンドルを返すので、それを *PDF_fit_image()* の位置合わせや拡大縮小のオプションとともに使うことができます。

取り込み PDF ページの埋め込みについても、これと同様に動作します。PDF ページを *PDF_open_pdi_page()* で開いてページハンドルを取得し、それを *PDF_fit_pdi_page()* で使う必要があります。位置合わせや拡大縮小のオプションは、ラスタ画像のものと同じものが使えます。

この節に載せる作成例はすべて、ラスタ画像でもテンプレートでも取り込み PDF ページでも、同じく動作します。コードの作成例はラスタ画像のためのものしか載せませんが、オブジェクト一般の配置方法をそこでは語っているのです。あらゆる *fit* 関数はすべて、それを呼び出す前にはかならず、*PDF_load_image()* か *PDF_open_pdi_document()* と *PDF_open_pdi_page()* への呼び出しを行う必要があります。簡潔のため、これらの呼び出しはここではあらためて示しません。

クックブック **画像と取り込み PDF ページに関するコードサンプルが PDFlib クックブックの** `images・pdf_import` **カテゴリにあります。**

7.3.1 単純にオブジェクトを配置

画像を参照点に置く　デフォルトでは、オブジェクトはその元のサイズで、左下隅を参照点に配置されます。この例では、画像の下端中央を参照点に配置してみましょう。下記のコードは、画像の下端中央を参照点 *(o, o)* に配置します。

```
p.fit_image(image, 0, 0, "position={center bottom}");
```

同様に、*position* オプションをキーワード *left・right・center・top・bottom* から別の組み合わせで使うことによって、オブジェクトそれぞれ左端・右端・中央・上端・下端を参照点に配置することができます。

画像を拡大縮小して配置　コードを下記のように変えると、画像を配置する際にその大きさも変更できます。

```
p.fit_image(image, 0, 0, "scale=0.5");
```

このコードは、オブジェクトの左下隅をユーザー座標系の点 *(o, o)* に配置します。と同時に、オブジェクトは *x・y* 方向に倍率 0.5 で拡大縮小されるので、元の 50 パーセントの大きさになります。

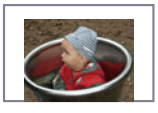
クックブック **完全なコードサンプルがクックブックの** `images/starter_image` **トピックにあります。**

7.3.2 オブジェクトを枠内に置く

オブジェクトを置くためには、あらかじめ定義した幅と高さの枠をあわせて使うこともできます。図 7.2 に、以下のいくつかの例の出力を示します。なお、青い枠と線は、枠の大きさが見られるように描き足してあるだけで、実際の出力にはありません。

画像を枠内に置く　枠を定義して、画像をその枠の右上に配置しましょう。枠は幅 70 単位、高さ 45 単位で、参照点 *(0, 0)* に配置します。画像はこの枠の右上に配置します（図 7.2a 参照）。同様に、画像を下端中央に配置することもできます。これを図示したのが図 7.2b です。なお、画像が枠より大きい場合は枠からはみ出します。

図 7.2 さまざまな位置合わせオプションに従って画像を枠内に配置

生成される出力	PDF_fit_image に与えるオプションリスト
a) 	`boxsize={70 45} position={right top}`
b) 	`boxsize={70 45} position={center bottom}`

適切なはめ込み方式を用いる　次に、さまざまなはめ込み方式を使って、オブジェクトを枠にはめ込みましょう。まずはデフォルト、すなわちはめ込み方式を使わず、切り抜きや拡大縮小も行わない場合から始めましょう。画像は幅 70・高さ 45 単位の枠の中央に配置します。その枠は参照点 *(0, 0)* に配置します。図 7.3a にその単純なケースを図示します。

枠の幅を 70 から 35 単位に縮めても、出力には何の影響もありません。画像は元の大きさを保ち、枠からははみ出します（図 7.3b 参照）。

画像を枠の中央にはめ込み　あらかじめ定義した矩形の中央に画像を置きたいときは、自分で計算をする必要は全くなく、適切なオプションを使えば実現できます。*position=center* を使って、幅 70・高さ 45 単位の枠（*boxsize={70 45}*）の中央に画像を配置しましょう。*fitmethod=meet* を使うと、画像は縦横比を保ちつつ、その上下が枠に収まりきるまで拡大縮小されます（図 7.3c 参照）。

枠の幅を 70 から 35 単位に縮めると、画像はその左右が枠に収まりきるまで縮小されます（図 7.3d 参照）。

これは画像配置でもっともよく使われる方式です。*fitmethod=meet* では、画像がつぶされないことが保証されるとともに、必ず枠内に、できるだけ大きく配置されます。

画像を枠全体にはめ込み　画像をもっと枠に合わせて、枠全体を画像が埋めるようにすることもできます。これは *fitmethod=entire* で実現できます。しかし、この組み合わせは画像をつぶすことが多いので、有用な場合はまれでしょう（図 7.3e 参照）。

画像を枠にはめ込む際に切り抜き　別のはめ込み方式（*fitmethod=clip*）を使うと、オブジェクトがはめ込んだ枠からはみ出したときに、そのオブジェクトを切り抜くことができます。枠の大きさを縦横とも 30 単位に縮めて、画像をその枠の中央に元の大きさのまま置いてみましょう（図 7.3f 参照）。

画像を枠の中央に置くことによって、画像はすべての端が均等に切り落とされます。同様に、画像の右上部分をすべて見せたいならば、*position={right top}* で置けばよいでしょう（図 7.3g 参照）。

図 7.3 さまざまなはめ込み方式に従って画像を枠にはめ込み

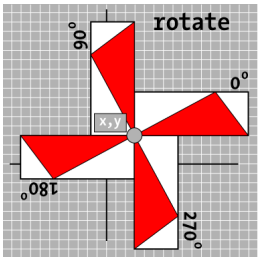
生成される出力		PDF_fit_image に与えるオプションリスト
a)		`boxsize={70 45} position=center`
b)		`boxsize={35 45} position=center`
c)		`boxsize={70 45} position=center fitmethod=meet`
d)		`boxsize={35 45} position=center fitmethod=meet`
e)		`boxsize={70 45} position=center fitmethod=entire`
f)		`boxsize={30 30} position=center fitmethod=clip`
g)		`boxsize={30 30} position={right top} fitmethod=clip`

オブジェクトをページに合わせる　与えられたページサイズにオブジェクトを合わせたいときは、オブジェクトを配置するはめ込み枠としてそのページを選べば簡単に実現できます。下記の命令は、縦横 595 × 842 の A4 サイズのページを使っています。

```
p.fit_image(image, 0, 0, "boxsize={595 842} position={left bottom} fitmethod=slice");
```

このコードでは、ページの左下隅に 1 つの枠を配置しています。その枠の大きさは、A4 ページの大きさと同じです。オブジェクトはこの枠の左下隅に配置され、縦横比を保つつ、枠全体を覆うまで拡大縮小されますので、ひいてはページ全体を覆うことになります。オブジェクトが枠からはみ出す場合は切り落とされます。*fitmethod=slice* はオブジェクトの拡大縮小を伴うのです（*fitmethod=clip* がオブジェクトを拡大縮小しないのと異なり）。もちろんこの例でも、*position* や *fitmethod* オプションは変えてもかまいません。

図 7.4
rotate オプション



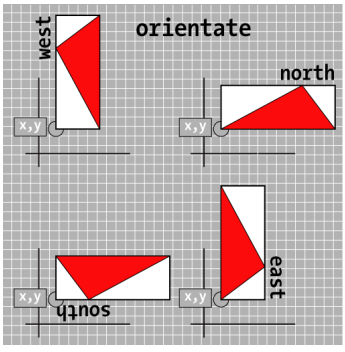


図 7.5
orientate オプション

7.3.3 オブジェクトの向きを変える

画像の向きを変えて配置　今度の例としては、画像の向きを西向きに変えてみましょう（*orientate=west*）。これはすなわち、画像が 90° 反時計回りに回転され、その回転後のオブジェクトの左下隅が参照点 *(o, o)* へ並行移動されることを意味します。画像はその場で回転します（図 7.6a 参照）。はめ込み方式を指定していないので、画像は元の大きさのまま出力されて、枠からはみ出します。

画像の向きを変えて縦横比を保ちつつ枠ちょうどにはめ込み　今度は、画像を西に向けたうえで、あらかじめ定義した大きさにすることに挑戦してみましょう。求める大きさの枠を定義して、画像の縦横比を変えずにその枠にはめ込みます（*fitmethod=meet*）。向きは *orientate=west* と指定します。デフォルトでは、画像は枠の左下隅に配置されます（図 7.6b 参照）。東に向けた画像を図 7.6c に、南向きを図 7.6d に示します。

orientate オプションは、図 7.5 に示すとおり、向きのキーワードとして *north*・*east*・*west*・*south* に対応しています。

なお、*orientate* オプションは、座標系全体にはいっさい影響せずに、配置するオブジェクトにだけ影響を及ぼします。

図 7.6 画像の向きを変える

生成される出力	PDF_fit_image に与えるオプションリスト
a) 	`boxsize={70 45} orientate=west`
b) 	`boxsize={70 45} orientate=west fitmethod=meet`
c) 	`boxsize={70 45} orientate=east fitmethod=meet`

生成される出力	PDF_fit_image に与えるオプションリスト
d) 	`boxsize={70 45} orientate=south fitmethod=meet`
e) 	`boxsize={70 45} position={center bottom} orientate=east fitmethod=clip`

向きを変えた画像を枠にはめ込んで切り抜き　画像を東に向けて（*orientate=east*）、枠の下端中央に位置を合わせましょう（*position={center bottom}*）。さらに、画像を元の大きさのまま配置し、もし画像が枠からはみ出したら切り落とします（*fitmethod=clip*）（図 7.6e 参照）。

7.3.4 オブジェクトを回転

オブジェクトの回転は、向きの変更と同じように動作します。しかし、配置するオブジェクトだけでなく、座標系全体に影響を与えます。

画像を回転させて配置　まずはじめに、画像を 90° 反時計回りに回転させることに挑戦してみましょう。オブジェクトを配置する前に、座標系は参照点 *(50, 0)* で 90° 反時計回りに回転されます。回転後のオブジェクトの右下隅（回転前のオブジェクトの左下隅だった所）が、最終的に参照点の位置になります。この場合を示したのが図 7.7a です。

回転は座標系全体に影響するので、枠の回転されます。同様に、画像を 30° 反時計回りに回転することができます（図 7.7b 参照）。図 7.4 に、*rotate* オプションの一般的な動作を図示します。

画像を回転してはめ込み　今度は、画像を 90° 反時計回りに回転させて、縦横比を保ちつつ枠にはめ込むことに挑戦してみましょう。これは *fitmethod=meet* を使えば実現できます（図 7.7c 参照）。同様に、画像を 30° 反時計回りに回転させて、その画像を縦横比を保ちつつ枠にはめ込むことができます（図 7.7d 参照）。

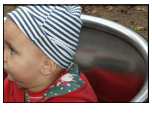
図 7.7 画像を回転

生成される出力	PDF_fit_image に与えるオプションリスト
a) 	`boxsize={70 45} rotate=90`
b) 	`boxsize={70 45} rotate=30`

生成される出力	PDF_fit_image に与えるオプションリスト
c) 	boxsize={70 45} rotate=90 fitmethod=meet
d) 	boxsize={70 45} rotate=30 fitmethod=meet

図 7.8
ページサイズの調整。左から、ちょうど・大きめ・小さめ





7.3.5 ページサイズの調整

ページサイズを画像に合わせる　今度の例としては、ページの大きさをオブジェクトの大きさに自動的に合わせましょう。これはたとえば、さまざまな画像を PDF 形式でアーカイブしておきたいときなどに有用です。参照点 *(x, y)* を使えば、ページをオブジェクトのサイズとちょうど同じにするか、それとも多少大きめや小さめにするかを、指定することができます。ページサイズを大きめにすると（図 7.8 参照）、画像のまわりにふちが残ります。ページサイズを画像より小さくすると、画像の一部は切り落とされます。まずは、ページサイズをオブジェクトの大きさとちょうど同じにしましょう。

```
p.fit_image(image, 0, 0, "adjustpage");
```

次のコードは、ページサイズを *x・y* 方向に 40 単位ずつ増やしていますので、オブジェクトのまわりに白ふちができます。

```
p.fit_image(image, 40, 40, "adjustpage");
```

次のコードは、ページサイズを *x・y* 方向に 40 単位ずつ減らしています。オブジェクトはページの端で切り落とされますので、オブジェクトの一部（幅 40 単位の）は見えなくなります。

```
p.fit_image(image, -40, -40, "adjustpage");
```

x・y 座標を手段として配置する方法（ページの端、または一般には座標軸からの、オブジェクトまでの間隔を指定する方法）のほかに、はめ込み枠を指定する方法もあります。これは、さまざまな組版規則に従ってオブジェクトが配置される矩形の領域です。この組版規則は、*boxsize・fitmethod・position* オプションで制御することができます。

取り込み PDF ページのページ枠を複製　取り込み PDF ページの、関連するページ枠（MediaBox・CropBox 等）すべてを、カレント出力ページへ複製することができます。*cloneboxes*オプションを、すべての関連する枠の値を読み取るために*PDF_open_pdi_page()*に与える必要があり、そしてその枠の値をカレントページに適用するために *PDF_fit_pdi_page()* にも与える必要があります：

```
/* ページを開き、ページ枠項目群を複製 */
inpage = p.open_pdi_page(indoc, 1, "cloneboxes");
...
/* 出力ページをダミーページサイズで開く */
p.begin_page_ext(10, 10, "");
...
/*
* 取り込みページを出力ページ上に配置し、入力ページ内にある
* すべてのページ枠を複製。これは、begin_page_ext()で用いた
* ダミーサイズを上書きします。
*/
p.fit_pdi_page(inpage, 0, 0, "cloneboxes");
```

この技法を活用すると、生成 PDF のページサイズや裁ち落とし等が必ず入力文書のページと同じになるようにすることができます。これは特にプリプレス分野において重要です。

7.3.6 配置画像・PDF ページに関する情報を取得

配置画像に関する情報　*PDF_info_image()* 関数を使って画像情報を取得できます。この関数で使えるキーワードは、一般的な画像情報（例：幅・高さをピクセル単位で）のみならず、その画像の出力ページ上への配置に関する情報も網羅しています（例：はめ込み計算実行後の幅・高さを絶対値で）。

下記のコードは、ピクセルサイズと、ある特定のはめ込みオプションにより画像を配置した後の絶対サイズの両方を取得しています：

```
String optlist = "boxsize={300 400} fitmethod=meet orientate=west";
p.fit_image(image, 0.0, 0.0, optlist);

imagewidth  = (int) p.info_image(image, "imagewidth", optlist);
imageheight = (int) p.info_image(image, "imageheight", optlist);
System.err.println("画像サイズ（ピクセル単位）: " + imagewidth + " x " + imageheight);

width  = p.info_image(image, "width", optlist);
height = p.info_image(image, "height", optlist);
System.err.println("画像サイズ（ポイント単位）: " + width + " x " + height);
```

配置 PDF ページに関する情報　*PDF_info_pdi_page()* 関数を使って、配置 PDF ページに関する情報を取得できます。この関数で使えるキーワードは、元のページに関する情報（例：その幅・高さ）のみならず、その取り込み PDF の出力ページ上への配置に関する情報も網羅しています（例：はめ込み計算実行後の幅・高さ）。

下記のコードは、取り込みページの元のサイズと、ある特定のはめ込みオプションによりそのページを配置した後のサイズの両方を取得しています：

```
String optlist = "boxsize={400 500} fitmethod=meet";
p.fit_pdi_page(page, 0, 0, optlist);
```

```
pagewidth  = p.info_pdi_page(page, "pagewidth", optlist);
pageheight = p.info_pdi_page(page, "pageheight", optlist);
System.err.println("元のページサイズ: " + pagewidth + " x " + pageheight);

width  = p.info_pdi_page(page, "width", optlist);
height = p.info_pdi_page(page, "height", optlist);
System.err.println("配置ページサイズ: " + width + " x " + height);
```

8 テキスト・表組版

8.1 テキスト行の配置とはめ込み

一行のテキストをページ上に配置するための関数*PDF_fit_textline()*にはさまざまな組版オプションがあります。この節ではもっとも重要なオプションをいくつかよく使われる応用例を用いて解説します。こうしたオプションの完全な説明は **PDFlib API リファレンス**にあります。*PDF_fit_textline()* のオプションの多くは *PDF_fit_image()* のオプションと同じです。ですのでここではテキスト関連の利用例のみを示します。画像の組版については、188 ページ「7.3 画像と取り込み PDF ページの配置」にある作成例を参照するよう推奨します。

以下の利用例では *PDF_fit_textline()* への呼び出しのみを示します。必要なフォントはすでに読み込まれて希望の文字サイズに設定されているものとします。

PDF_fit_textline() は、仮想的なテキスト枠を用いてテキストの位置合わせを決めています。テキスト枠の幅はテキストの幅と同じであり、枠の高さは フォントの大文字の高さと同じです。テキスト枠は、テキスト枠を定義する *matchbox* オプションで変更することができます。

以下の作成例では、参照点の座標は *PDF_fit_textline()* の *x・y* 引数として与えられます。テキスト行に対するはめ込み枠は、テキストが配置される領域です。それは *PDF_fit_textline()* の *x・y* 引数と適切なオプション（*boxsize・position・rotate*）で指定される矩形領域として定義されます。はめこみ枠は、*margin* オプションを用いて、左 / 右または上 / 下へ縮めることもできます。

クックブック **テキスト出力の諸側面に関するコードサンプルが PDFlib クックブックの** `text_output` **カテゴリにあります。**

8.1.1 単純なテキスト行配置

テキストを参照点に置く　デフォルトでは、テキストは左下隅を参照点に合わせて配置されます。しかしこの例では、テキストの下端中央を参照点に合わせて配置したいのです。下記のコードは、テキスト枠の下端中央を参照点 *(30, 20)* に合わせて配置します。

```
p.fit_textline(text, 30, 20, "position={center bottom}");
```

図 8.1 に、中央揃えのテキスト配置の様子を図示します。同様に、キーワード *left・right・center・top・bottom* の組み合わせを変えた *position* オプションを使って、参照点にテキストを配置することができます。

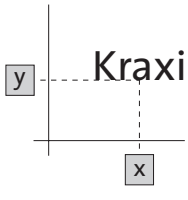


図 8.1
テキストの中央揃え

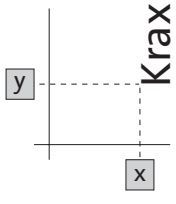


図 8.2
西向きの単純なテキスト

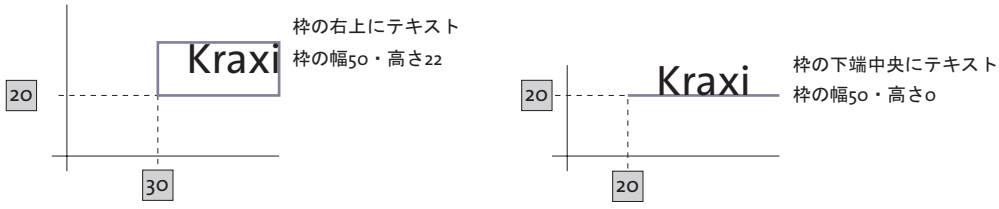


図 8.3 テキストを枠内に置く

テキストの向きを変えて配置　次は、テキストを回転させて、その左下隅（回転後の）を参照点に合わせて配置してみましょう。下記のコードは、テキストを西（90° 反時計回り）に向けた後、その回転したテキストの左下隅を参照点 *(0, 0)* へ並行移動させます。

```
p.fit_textline(text, 0, 0, "orientate=west");
```

図 8.2 に、単純なテキストの向きを変えて配置した様子を図示します。

8.1.2 テキストを枠内に置く

テキストを置くには、幅と高さをあらかじめ定義した枠をあわせて使うこともでき、テキストはその場合枠に対して相対的な位置に置くことが可能です。図 8.3 に、その一般的なはたらきを図示します。

テキストを枠内に置く　矩形の枠を定義して、この枠内の右上にテキストを配置しましょう。コード上で、幅 50 単位・高さ 22 単位の枠を、参照点 *(30, 20)* に定義します。図 8.4a のように、テキストは枠の右上に配置されます。

同様に、下端中央にテキストを配置することもできます。この場合を図示したのが図 8.4b です。

枠とテキストの間に間隔をあけるには、*margin* オプションを付け加えます（図 8.4c 参照）。

なお、図中の青い矩形や線は、枠のサイズを見せるために描いたもので、実際の出力には現れません。

図 8.4 さまざまな位置合わせオプションに従ってテキストを枠内に配置

生成される出力	PDF_fit_textline( ) に与えるオプションリスト
a) Kraxi	`boxsize={50 22} position={right top}`
b) Kraxi	`boxsize={50 22} position={center bottom}`
c) Kraxi	`boxsize={50 22} position={center bottom} margin={0 3}`
d) Kraxi	`boxsize={50 0} position={center bottom}`

生成される出力	PDF_fit_textline( ) に与えるオプションリスト
e) 	boxsize={0 35} position={left center} orientate=west

テキストを横線や縦線で整列させる　テキストの位置合わせを、横線や縦線（すなわち高さや幅がゼロの枠）に沿って行うというのは、若干極端なケースではありますが、有用な場合もあります。図 8.4d は、テキストを枠に対して下端中央揃えで配置したものです。幅を 50、高さを 0 としたことで、枠はまるで横線のようになっています。

テキストを縦線に沿って中央揃えするためには、西向きにして、枠に対して左端中央に置きましょう。この場合を図 8.4e に示します。

8.1.3 テキストを枠にはめ込み

この項では、さまざまなはめ込み方式を用いて、テキストを枠にはめ込みましょう。カレントのフォントと文字サイズはどの例でも同じとしておき、さまざまなはめ込み方式にともなって文字サイズなどのプロパティが変わる様子がわかるようにします。

デフォルトの場合から始めましょう。すなわち、何のはめ込み方式も用いないで、切り落としも拡縮も一切されないようにする場合です。テキストは、幅 100 単位・高さ 35 単位の枠の中央に配置されます（図 8.5a 参照）。

枠の幅を 100 から 50 単位に縮めても、出力には全く影響を与えません。テキストは元の文字サイズを保ち、枠からはみ出します（図 8.5b 参照）。

テキストを小さな枠につめ込み　それでは、テキスト全体を枠の中に、体裁を保ちながらつめ込んでみましょう。これは *fitmethod=auto* オプションで実現できます。図 8.5c では、枠の幅が充分広いので、テキストはまったく元の大きさのまま、変わらずに枠に収まっています。

枠の幅を 100 から 58 に縮小すると、テキストは長すぎて収まりきらなくなります。はめ込み方式 *auto* はテキストに長体を、*shrinklimit* オプション（デフォルト：0.75）を限度としてかけようとします。図 8.5d は、テキストが元の長さの 75 パーセントまで圧縮された様子を示します。

枠の幅をさらに 30 単位まで縮めると、テキストは長体をかけても収まりきらなくなります。すると、*meet* 方式が適用されます。*meet* 方式は、テキスト全体が枠に収まるまで文字サイズを下げます。この場合を示したのが図 8.5e です。

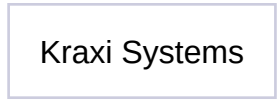
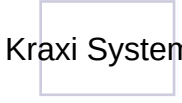
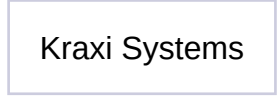
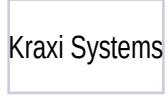
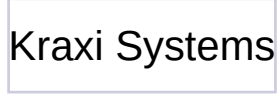
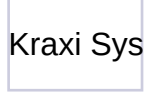
テキストの文字サイズを上げて枠にはめ込み　テキストを枠の幅（または高さ）いっぱいに拡げて、ただし縦横比は保ったまま、はめ込みたい時があるかもしれません。テキストより大きい枠に対して *fitmethod=meet* を用いると、テキストの幅が枠の幅と一致するまでテキストが大きくなります。この場合を図示したのが図 8.5f です。

テキストを枠いっぱいにはめ込み　さらに、テキストが枠の中を埋めつくすようにはめ込むことも可能です。この場合は、*fitmethod=entire* を使います。しかしこの設定は、テキストをほぼ確実にゆがめてしまうので、使うことはめったにないでしょう（図 8.5g 参照）。

テキストを枠で切り落としてはめ込み　これもレアなケースですが、テキストを元のサイズのままはめ込んで、もしも枠からはみ出た部分は切り落としたい時もあるかもしれま

せん。その場合は *fitmethod=clip* が使えます。図 8.5h では、テキストを枠の左端に配置していますが、枠の幅が足りません。テキストは右側が切り落とされます。

図 8.5 さまざまなオプションに従ってテキストをページ上の枠にはめ込み

生成される出力	PDF_fit_textline( ) に与えるオプションリスト
a) 	`boxsize={100 35} position=center fontsize=12`
b) 	`boxsize={50 35} position=center fontsize=12`
c) 	`boxsize={100 35} position=center fontsize=12 fitmethod=auto`
d) 	`boxsize={58 35} position=center fontsize=12 fitmethod=auto`
e) 	`boxsize={30 35} position=center fontsize=12 fitmethod=auto`
f) 	`boxsize={100 35} position=center fontsize=12 fitmethod=meet`
g) 	`boxsize={100 35} position=center fontsize=12 fitmethod=entire`
h) 	`boxsize={50 35} position={left center} fontsize=12 fitmethod=clip`

テキストの縦中央揃え　*PDF_fit_textline( )* におけるテキストの高さは、デフォルトではキャップハイト、すなわち大文字の *H* の高さです。テキストを枠の中央に置くと、縦方向にはそのキャップハイトにもとづいて中央揃えされます（図 8.6a 参照）。

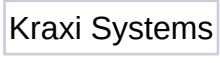
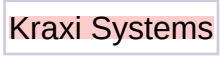
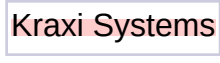
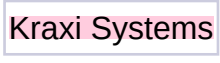
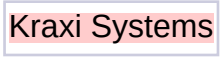
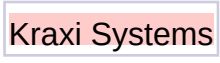
それ以外の高さをテキスト枠に対して指定するには、範囲枠機能を使います（242 ページ「8.4 範囲枠」も参照）。*PDF_fit_textline( )* の *matchbox* オプションは、テキスト行の高さを定義しており、与えられた文字サイズのキャップハイトがそのデフォルトになっています。この範囲枠の高さは、その *boxheight* サブオプションにもとづいて算出されます。*boxheight* サブオプションは、ベースラインからテキスト上端・下端までの距離を決定します。デフォルトの設定は *matchbox={boxheight={capheight none}}*、すなわち範囲枠の上

端はベースラインより上にあってキャップハイトに一致し、範囲枠の下端はベースラインを下へ越えません。

範囲枠の大きさを図示するため、ここでは赤く色を塗りましょう（図 8.6b 参照）。図 8.6c では、テキストが *xheight* にもとづいて縦中央揃えされるように、その高さの範囲枠を定義しています。

図 8.6d ～ f に、有用なさまざまな *boxheight* 設定の範囲枠（赤）と、それの決める高さにもとづいて枠（青）の中で中央揃えされたテキストを示します。

図 8.6 さまざまな範囲枠高さに従ってテキストを枠にはめ込み

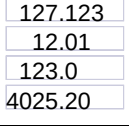
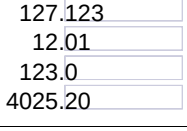
生成される出力	PDF_fit_textline( ) に与えるオプションリスト
a) 	`boxsize={80 20} position=center fitmethod=auto`
b) 	`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={capheight none} fillcolor={rgb 1 0.8 0.8}}`
c) 	`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={xheight none} fillcolor={rgb 1 0.8 0.8}}`
d) 	`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={ascender none} fillcolor={rgb 1 0.8 0.8}}`
e) 	`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={ascender descender} fillcolor={rgb 1 0.8 0.8}}`
f) 	`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={fontsize none} fillcolor={rgb 1 0.8 0.8}}`

8.1.4 テキストを文字で整列させる

テキストを文字で整列させる　テキストを、ある特定の文字で整列させたい場合があるかもしれません。たとえば数値の小数点揃えなどです。図 8.7a に示すように、テキストははめ込み枠の中央に置かれています。*PDF_fit_textline( )* で *alignchar=.* オプションを用いることで、数値が点で揃います。

点を枠の中央に配置している *position* オプションを省くこともできます。そうすれば、デフォルトの *position={left bottom}* が用いられるので、点は参照点に配置されます（図 8.7b 参照）。一般に、整列文字はその右下隅が参照点に配置されます。

図 8.7 テキスト行を点で整列させる

生成される出力	PDF_fit_textline( ) に与えるオプションリスト
a) 	`boxsize={70 8} position={center bottom} alignchar=.`
b) 	`boxsize={70 8} position={left bottom} alignchar=.`

8.1.5 スタンプを配置

クックブック 完全なコードサンプルがクックブックの text_output/simple_stamp トピックにあります。

テキストの回転を指定しなくても、スタンプという便利な機能を使えば、テキストを枠内に対角線に沿って配置することができます。スタンプ機能は洗練された自動計算を行い、テキストが枠内いっぱいに拡がるよう文字サイズと回転角を決定します。対角線スタンプを、たとえばページの背景などに配置するには、*PDF_fit_textline()* で *stamp* オプションを指定します。*stamp=ll2ur* を指定すると、テキストははめ込み枠の左下隅から右上隅へ配置されます。これに対し、*stamp=ul2lr* を指定すると、テキストははめ込み枠の左上隅から右下隅へ配置されます。図 8.8 では、*showborder=true* を用いてはめ込み枠とスタンプの外接枠を図示しています。

図 8.8 左下から右上へのスタンプのようにテキスト行をはめ込み

生成される出力	PDF_fit_textline( ) に与えるオプションリスト
	fontsize=8 boxsize={160 50} stamp=ll2ur showborder=true

8.1.6 リーダを利用

リーダを使うと、はめ込み枠の端とテキストとの間の余白を埋めることができます。たとえば点リーダは、目次の各項目とそのページ番号をつないで見やすいようによく利用されます。

目次のリーダ　*PDF_fit_textline()* で *leader* オプションと *alignment={none right}* サブオプションを用いると、テキスト行の右にリーダが付加されて、テキスト枠の右端まで繰り返されます。リーダ右端と右枠との間隔は一定ですが、テキストとリーダ左端との間隔は変動の可能性があります（図 8.9a 参照）。

クックブック テキスト行の中での点リーダの使用例を示す完全なコードサンプルがクックブックの text_output/leaders_in_textline トピックにあります。

クックブック テキストフローの中での点リーダの使用例を示す完全なコードサンプルがクックブックの text_output/dot_leaders_with_tabs トピックにあります。

ニュース電光掲示板のリーダ　別の用例としては、ニュース電光掲示板ふうにしたい場合もあるかもしれません。ここではプラスとスペース「+ 」をリーダに使いましょう。テキスト行は中央に配置し、リーダはそのテキスト行の前と後に印字させます（*alignment={left right}*）。リーダの左右端は左右の枠に揃い、テキストとの間隔は変動の可能性があります（図 8.96b 参照）。

図 8.9 リーダを使ったテキスト行をはめ込み

生成される出力		PDF_fit_textline( ) に与えるオプションリスト
a)	Features of Giant Wing ..... Description of Long Distance Glider..... Benefits of Cone Head Rocket.....	boxsize={200 10} leader={alignment={none right}}
b)	+ + + + + + + + + Giant Wing in purple!+ + + + + + + + + + + + Long Distance Glider with sensational range! + + + + + + + Cone Head Rocket incredibly fast! + + + + +	boxsize={200 10} position={center bottom} leader={alignment={left right} text={+ }}

8.1.7 パス上のテキスト

テキストを直線上に配置するのではなく、任意のパス上に配置することもできます。PDFlib は個々のキャラクタをパス上に、テキストがそのパスの曲線に沿うように配置します。パス上のテキストを作成するには *PDF_fit_textline( )* の *textpath* オプションを用います。パスはそれ以前に作成済みである必要があり、パスハンドルで表します。パスハンドルは、*PDF_add_path_point( )* および関連するパスオブジェクト関数群で明示的にパスを構築するか、あるいは既存のラスタ画像内のクリッピングパスに対するハンドルを取得することによって作成できます。下記のコードは、1 個のパスを作成し、テキストをそのパス上に配置します（図 8.10 参照）：

```
/* パスを原点に定義 */
path = p.add_path_point(  -1,   0,   0, "move", "");
path = p.add_path_point(path, 100, 100, "control", "");
path = p.add_path_point(path, 200,   0, "circular", "");

/* テキストをパス上に配置 */
p.fit_textline("Long Distance Glider with sensational range!", x, y,
    "textpath={path=" + path + "} position={center bottom}");

/* 例ですのでパスも描いてみせましょう */
p.draw_path(path, x, y, "stroke");
```

クックブック 完全なコードサンプルがクックブックの text_output/text_on_a_path トピックにあります。

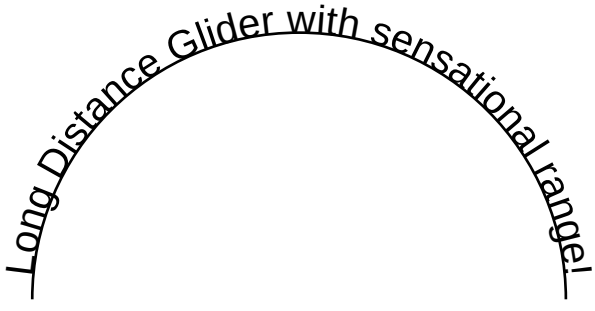


図 8.10
パス上のテキスト

画像のクリッピングパスを用いてテキストを配置　パス関数群でパスオブジェクトを手作業で構築するのではなく、画像からクリッピングパスを抽出して、そのパス上にテキストを配置することもできます。この画像は *honorclippingpath* オプションとともに読み込んである必要があり、かつ、求めるパスがその画像のデフォルトのクリッピングパスでない場合には *clippingpathname* も *PDF_load_image()* に与える必要があります：

```
image = p.load_image("auto", "image.tif", "clippingpathname={path 1}");

/* 画像のクリッピングパスからパスオブジェクトを作成 */
path = (int) p.info_image(image, "clippingpath", "");
if (path == -1)
        throw new Exception("エラー： クリッピングパスが見つかりません!");

/* テキストをパス上に配置 */
p.fit_textline("Long Distance Glider with sensational range!", x, y,
    "textpath={path=" + path + "} position={center bottom}");
```

パスとテキストの間をあける　デフォルトでは、PDFlib は個々のキャラクタをパス上に配置しますので、グリフとパスの間にはあきはありません。パスとテキストの間をあけたいときは、単純にキャラクタ枠を延長することができます。これは、*matchbox* オプションの *boxheight* サブオプションでキャラクタ枠の縦延長を指定することで簡単に実現できます。下記のオプションリストはディセンダを考慮に入れています（図 8.11 参照）：

```
p.fit_textline("Long Distance Glider with sensational range!", x, y,
    "textpath={path=" + path + "} position={center bottom} " +
    "matchbox={boxheight={capheight descender}}");
```

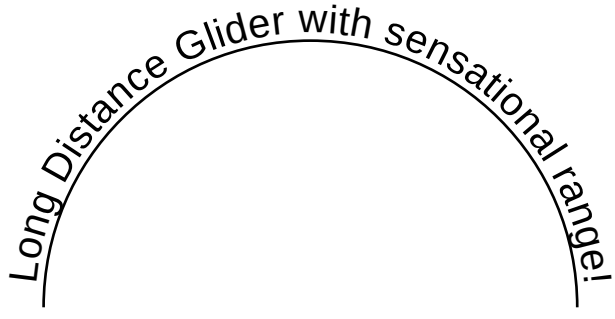


図 8.11
パス上のテキストで、パスとテキストの間をあけた

8.2 複数行のテキストフロー

1 行のテキストをページ上に配置する機能だけでなく、PDFlib は、任意の長さのテキストを配置できるテキストフローという機能にも対応しています。テキストは何行にも、何段にも、あるいは何ページにもわたることができ、またその見ばえはさまざまなオプションで制御することができます。フォント・サイズ・色といった文字属性を、テキストのどの部分にでも適用することができます。テキストの両端揃え・片端揃えや、段落のインデントや、タブ位置といったテキストフロー属性を指定できます。テキストの中にソフトハイフンで示した改行機会が考慮されます。図 8.12・図 8.13 は、請求書のさまざまな部分がページ上にテキストフロー機能を用いて配置できているさまを例示したものです。こうした出力を制御するためのオプションについて、以下の項で詳しく説明します。

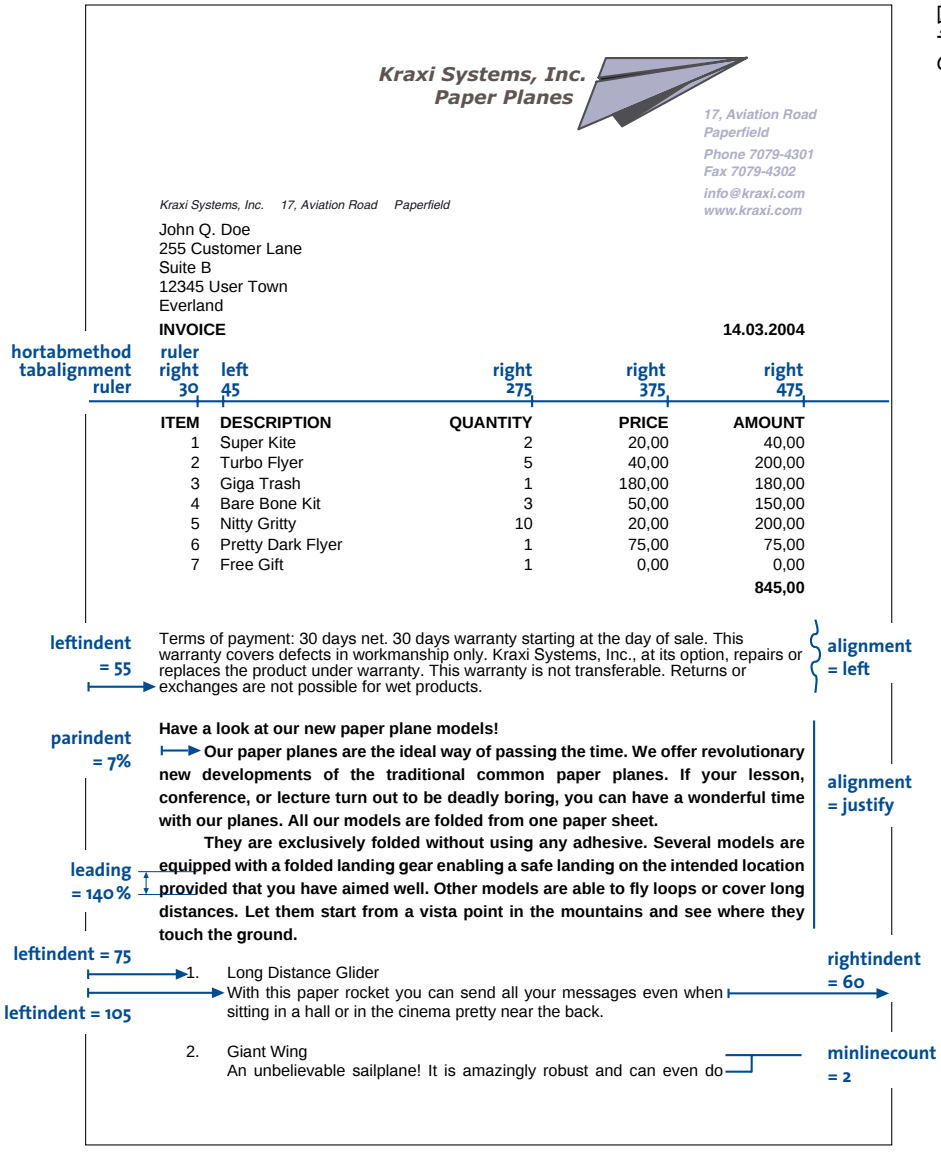


図 8.12
テキストフローの組版

複数行のテキストフローは、1 つの矩形（はめ込み枠という）内にも複数の矩形内にも配置することができます。このはめ込み枠は 1 ページ上にあっても複数ページ上にあってもかまいません。テキストフローをページ上に配置するには以下の手順を踏む必要があります。

- 関数 *PDF_add_textflow()* が、テキストを一部分ずつ、その組版オプションとともに受け入れて、そして 1 つのテキストフローオブジェクトを作成してハンドルを返します。または関数 *PDF_create_textflow()* が、組版制御のためのインラインオプションを含むことのできるテキストの全体を、一度の呼び出しで分析します。これらの関数ではページ上にテキストは配置されません。
- 関数 *PDF_fit_textflow()* が、このテキストフローの全部ないし一部を、与えられたはめ込み枠内に配置します。すべてのテキストを配置するには、この段階を数回繰り返す必要があるかもしれません。その場合はこの関数を呼び出すたびに新しいはめ込み枠を与える必要があるでしょう。このはめ込み枠は同じページにあっても別のページにあってもかまいません。
- 関数 *PDF_delete_textflow()* が、テキストフローを文書内に配置した後に、このテキストフローオブジェクトを削除します。

テキストフローを作成するための関数 *PDF_add/create_textflow()* は、組版処理を制御するためのさまざまなオプションに対応しています。このオプションは関数のオプションリストで与えることもできますし、あるいは *PDF_create_textflow()* を使うときはテキスト内に**「インライン」**オプションとして入れ込むこともできます。*PDF_info_textflow()* を使うと、組版の結果や、その他テキストフローに関するたくさんの詳細を取得することができます。以下、テキストフローの配置について、いくつかのよくある応用を例にして説明します。テキストフローオプションの完全な一覧は ***PDFlib API* リファレンス**にあります。

PDF_add/create_textflow() で対応しているオプションの中には *PDF_fit_textline()* と同様のものがたくさんあります。ですから 197 ページ「8.1 テキスト行の配置とはめ込み」の例をあわせてよく把握しておくとよいでしょう。以下の項では、複数行テキストに関係のあるオプションだけを解説します。

クックブック **テキスト出力の諸側面に関するコードサンプルが PDFlib クックブックの** `text_output` **カテゴリにあります。**

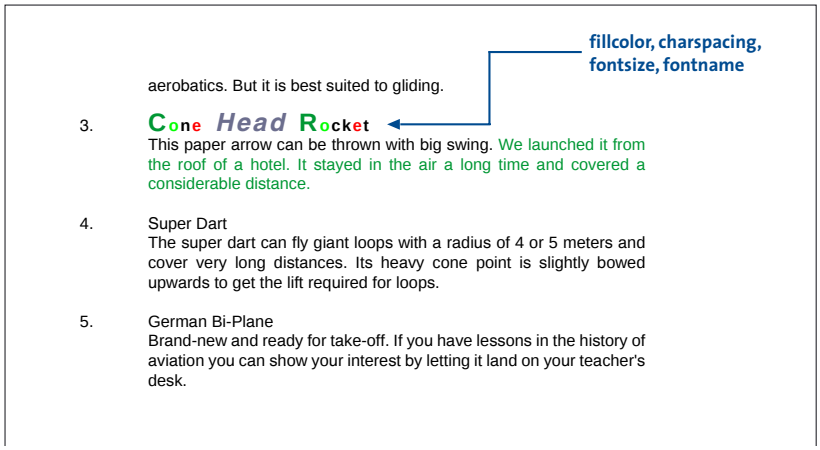


図 8.13
テキストフローの組版

8.2.1 テキストフローをはめ込み枠に配置

テキスト枠に対するはめ込み枠は、テキストが配置される領域です。それは *PDF_fit_textflow()* の *llx・lly・urx・ury* 引数で指定される矩形領域として定義されます。

テキストを１つのはめ込み枠に配置　簡単な例から始めましょう。下記のコードは、*PDF_add_textflow()* への呼び出しを 2 回使って、ボールドテキストの部分と標準テキストの部分をくっつけています。フォント・文字サイズ・エンコーディングを明示的に指定しています。1 回目の *PDF_add_textflow()* への呼び出しでは、-1 を与えれば、テキストフローハンドルが返ってきますので、その後にもしまた *PDF_add_textflow()* を呼び出すときがあるならそれが使えます。*text1・text2* には、印字したい実際のテキストが入っているものとします。

PDF_fit_textflow() を使って、できあがったテキストフローをページ上のはめ込み枠に、デフォルトの組版オプションを用いて配置します。

```
/* テキストをボールドフォントで追加 */
tf = p.add_textflow(-1, text1, "fontname=Helvetica-Bold fontsize=9 encoding=unicode");
if (tf == -1)
        throw new Exception("エラー：" + p.get_errmsg());

/* テキストを標準フォントで追加 */
tf = p.add_textflow(tf, text2, "fontname=Helvetica fontsize=9 encoding=unicode");
if (tf == -1)
        throw new Exception("エラー：" + p.get_errmsg());

/* Place all text */
result = p.fit_textflow(tf, left_x, left_y, right_x, right_y, "");
if (!result.equals("_stop"))
        { /* ... */}

p.delete_textflow(tf);
```

テキストを複数ページ上の２つのはめ込み枠に配置　*PDF_fit_textflow()* で配置したテキストがはめ込み枠内に収まりきらなかったときは、出力は中断されて関数は文字列 *_boxfull* を返します。PDFlib はすでに配置したテキストの量を記憶していて、この関数が再び呼ばれた時には残りのテキストを引き続き配置します。この他に、新規ページを作成する必要もあるかもしれません。次のコードは、1 つないし複数のページ上の、ページあたり 2 つのはめ込み枠に、1 つのテキストフローを、テキストをすべて配置しおわるまで配置する方法を例示したものです（図 8.14 参照）。

クックブック　完全なコードサンプルがクックブックの text_output/starter_textflow トピックにあります。

```
/* テキスト全部が配置されるまで回る。配置するべきテキストがまだあるなら新規ページを作成。
 * 全ページに2段組を作成。
*/
do
{
        String optlist = "verticalalign=justify linespreadlimit=120%";

        p.begin_page_ext(0, 0, "width=a4.width height=a4.height");

        /* 1段目に流し込み */
```

```
        result = p.fit_textflow(tf, llx1, lly1, urx1, ury1, optlist);

        /* まだテキストがあるなら2段目に流し込み */
        if (!result.equals("_stop"))
            result = p.fit_textflow(tf, llx2, lly2, urx2, ury2, optlist);

        p.end_page_ext("");

        /* 「_boxfull」ならまだテキストがあるので続ける必要がある。
         * 「_nextpage」は「新規段組を開始」と解釈
         */
} while (result.equals("_boxfull") || result.equals("_nextpage"));

/* エラーをチェック */
if (!result.equals("_stop"))
{
        /* 枠がとても小さくてテキストが全然入らないときは「_boxempty」が起こる。
        */
        if (result.equals( "_boxempty"))
            throw new Exception("エラー：" + p.get_errmsg());
        else
        {
            /* それ以外の戻り値は「return」オプションによるユーザー終了。
             * これを扱うにはそのためのコードが必要。
            */
        }
}
p.delete_textflow(tf);
```

8.2.2 段落の組版のオプション

上の例では段落に、デフォルトの設定を用いました。デフォルトは左揃え、行送り 100%（文字サイズそのまま）、などとなっています。

段落の組版を調整したいなら、*PDF_add_textflow()* にオプションを追加できます。たとえばテキストを左余白から 15 単位、右余白から 10 単位、インデントしたいとします。各段落の先頭行はさらに 20 単位インデントしましょう。テキストを両端揃えとし、行送

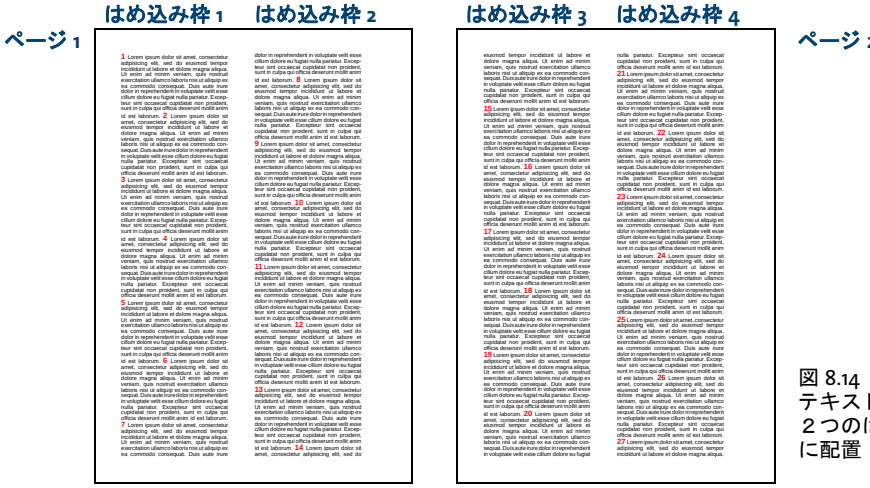


図 8.14
テキストフローを２つのはめ込み枠に配置

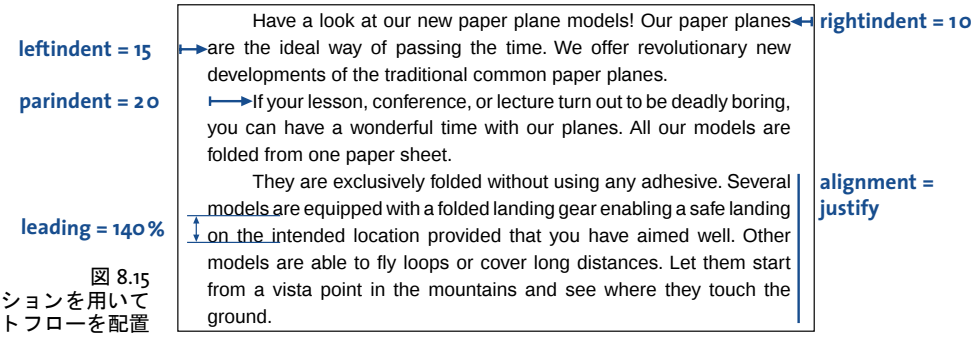


図 8.15
オプションを用いて
テキストフローを配置

りは 140% に拡げましょう。そして文字サイズを 8 ポイントに下げます。これを実現するには、*PDF_add_textflow()* のオプションリストを下記のとおり拡張します（図 8.15 参照）。

```
String  optlist =
        "leftindent=15 rightindent=10 parindent=20 alignment=justify " +
        "leading=140% fontname=Helvetica fontsize=8 encoding=unicode";
```

8.2.3 インラインオプションリストとマクロ

図 8.15 のテキストはまだ完璧ではありません。見出し「Have a look at our new paper plane models!」を独立した行にして、フォントももっと大きくして、そして中央揃えにしましょう。これを実現するにはいくつかの方法があります。

***PDF_create_textflow()* にインラインオプションリストを与える**　ここまでは組版オプションは、関数に直接与えるオプションリスト内で指定してきました。今回もこれと同じやり方を続けるならば、テキストを分割して二度の呼び出しに分ける必要があります。一度目で見出しを、二度目で残りのテキストを配置するわけです。ですが、場合によっては、たとえば書式変更箇所が多いときなどは、この方法は少々面倒かもしれません。

それならば、*PDF_add_texflow()* のかわりに *PDF_create_textflow()* を使うことができます。*PDF_create_textflow()* はテキストと、テキストの中に直接入れ込まれたインラインオプションというものを解釈します。これは単純にオプションをテキスト内に入れ込むということです。インラインオプションリストはテキスト本体の一部として与えられます。デフォルトでは、インラインオプションはキャラクタ「<」と「>」ではさみます。それでは次のように、見出しと残りの段落に対するオプションをテキスト本体の中に入れ込んでみましょう。

注 以下作成例ではすべて、インラインオプションリストに色をつけて示します。改段落キャラクタは矢印で図示します。

```
<leftindent=15 rightindent=10 alignment=center fontname=Helvetica fontsize=12
encoding=winansi>Have a look at our new paper plane models! ↵
<alignment=justify fontname=Helvetica leading=140% fontsize=8 encoding=winansi>
Our paper planes are the ideal way of passing the time. We offer
revolutionary new developments of the traditional common paper planes. ↵
<parindent=20>If your lesson, conference, or lecture
turn out to be deadly boring, you can have a wonderful time
with our planes. All our models are folded from one paper sheet. ↵
They are exclusively folded without using any adhesive. Several
models are equipped with a folded landing gear enabling a safe
```

```
landing on the intended location provided that you have aimed well.
Other models are able to fly loops or cover long distances. Let them
start from a vista point in the mountains and see
where they touch the ground.
```

オプションリストをはさむキャラクタは *begoptlistchar*・*endoptlistchar* オプションで再定義することができます。*begoptlistchar* オプションにキーワード *none* を与えるとオプションリストの検出は完全に無効になります。これは、テキストがインラインオプションリストをまったく含まない場合に、「<」・「>」を確実に通常のキャラクタとして処理させたいときに有用です。

記号キャラクタとインラインオプションリスト　記号キャラクタはテキストフローにおいて、インラインオプションリストと組み合わせも使うことができます。インラインオプションリストを開始するキャラクタに対するコード（デフォルトでは「<」U+003C）は、*encoding=builtin* によるフォントに対するテキスト内では記号コードとして解釈されません。これと同じコードの記号グリフを選択するには、テキストフォントに対して利用可能な回避策がそのまま使えます。すなわち、開始キャラクタを *begoptlistchar* オプションで再定義するか、あるいは *textlen* オプションを用いて記号グリフの数を指定します。なお、先述の理由により、文字参照（*<* 等）は回避策としては使えません。

マクロ　上記のテキストはいくつかの種類の段落でできています。すなわち見出しと本文であり、本文にはさらにインデントのあるものとないものとがあります。こうした各種の段落をそれぞれの形に組版していくわけですが、しかしテキストフローがもっと長ければ同じ指定を何度もしなければなりません。段落替えのたびに同じインラインオプション群を何度も書かなくてもすむようにするには、インラインオプション群をマクロにまとめれば、テキスト内からそのマクロを名前で参照することができます。図 8.16 に示すような 3 つのマクロを定義しましょう。*H1* は見出し用、*Body* は本文段落用、*Body_indented* はインデントする段落用です。マクロを利用するには、キャラクタ *&* をマクロ名の前につけたものをオプションリストに書きます。次のコードでは、上で用いたインラインオプション群に従って 3 つのマクロを定義し、それをテキスト内で利用しています。

```
<macro {
H1 {leftindent=15 rightindent=10 alignment=center
fontname=Helvetica fontsize=12 encoding=winansi}

Body {leftindent=15 rightindent=10 alignment=justify leading=140%
```

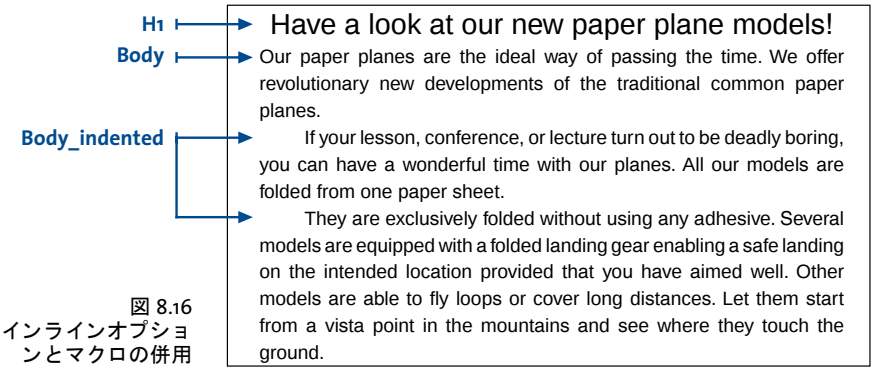


図 8.16
インラインオプションとマクロの併用

```
fontname=Helvetica fontsize=8 encoding=winansi}

Body_indented {parindent=20 leftindent=15 rightindent=10 alignment=justify
leading=140% fontname=Helvetica fontsize=8 encoding=winansi}
}>
<&H1>Have a look at our new paper plane models! ↵
<&Body>Our paper planes are the ideal way of passing the time. We offer
revolutionary new developments of the traditional common paper planes. ↵
<&Body_indented>If your lesson, conference, or lecture
turn out to be deadly boring, you can have a wonderful time
with our planes. All our models are folded from one paper sheet. ↵
They are exclusively folded without using any adhesive. Several
models are equipped with a folded landing gear enabling a safe
landing on the intended location provided that you have aimed well.
Other models are able to fly loops or cover long distances. Let them
start from a vista point in the mountains and see
where they touch the ground.
```

オプションの明示的設定　注意しなければならないのは、マクロの中で設定しなかったオプションではすべて前の値が保持されるということです。オプションが勝手に「継承」されておかしな副作用を起こさないようにしたければ、そのマクロが必要とする設定をすべて明示的に指定する必要があります。そうすればそのマクロが他のオプションリストとの順序や組み合わせにかかわらずいつも確実に同じように動作するようにすることができます。

あるいは、この動作を逆に利用して、ある特定の設定群についてはあえて明示的に与えずに、それぞれの利用箇所における設定が保持されるようにするという方式もありえます。たとえば、フォント名は指定するけれども *fontsize* オプションは与えないというマクロを作ってもよいのです。すると、文字サイズをつねに前のテキストと同じにすることができます。

インラインオプションか、関数の引数としてオプションを渡すか　テキストフローを利用する際には、テキストがプログラムコード内にリテラルに書かれているか、それともどこか外部の情報源から来るかというのは重大な違いです。また、組版指示がテキストとは別にあるか、それともテキスト内にあるかというのも重大です。たいていのアプリケーションではテキストはデータベースのような何らかの外部情報源から来るでしょう。現実的には次の 2 通りの状況が考えられます。

- テキスト内容が外部情報源から来て、組版オプションはプログラム内にある場合：実行時に、外部情報源から来る短いテキスト群をプログラム内で合成し、それを組版オプション群と（関数呼び出しの所で）組み合わせます。
- テキスト内容も組版オプションも外部情報源から来る場合：大量のテキストが組版オプション群を含んだ状態で外部情報源から来ます。組版はテキスト内のインラインオプション群によって与えられます。インラインオプションは単純なオプションとして書かれていることもあれば、マクロとして書かれていることもありえます。マクロに関しては、マクロ定義とマクロ呼び出しとを区別して考える必要があります。そうすると、面白い中間的形態が作り出せます。すなわち、テキスト内容は外部情報源から来て組版のためのマクロ呼び出しもその中に含んでいるのですが、マクロ定義は別に用意しておいて実行時にはじめて与えるようにするのです。この方式の利点は、外部のテキストに手を加えずに組版を簡単に変更できることです。たとえばグリーティングカードを作る際など、さまざまなスタイルをマクロで定義しておけば、カードの雰囲気をロマンチックにしたり、テクニカルにしたり、その他さまざまに変えることができるでしょう。

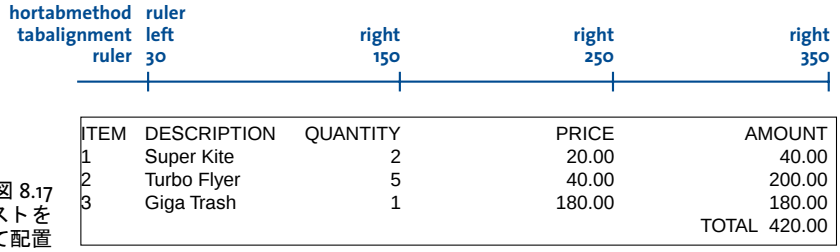


図 8.17 テキストを表として配置

8.2.4 タブ位置

次の例ではタブキャラクタを用いて左寄せ・右寄せの列のある簡単な表を配置します。表は次のような複数行のテキストを持ち、各項目が互いにタブキャラクタ（矢印で示す）で区切ってあります。

```
ITEM → DESCRIPTION → QUANTITY → PRICE → AMOUNT ↵
1 → Super Kite → 2 → 20.00 → 40.00 ↵
2 → Turbo Flyer → 5 → 40.00 → 200.00 ↵
3 → Giga Trash → 1 → 180.00 → 180.00 ↵
→ → → → TOTAL 420.00
```

この簡単な表を配置するには、*PDF__add/create_textflow()* で以下のオプションリストを用います。*ruler* オプションがタブ位置を定義しており、*tabalignment* がタブ位置の整列方向を指定しており、*hortabmethod* オプションがタブ位置の処理方式を指定しています（出力を図 8.17 に示します）。

```
String  optlist =
        "ruler={30     150   250   350} " +
        "tabalignment={left right right right} " +
        "hortabmethod=ruler leading=120% fontname=Helvetica fontsize=9 encoding=winansi";
```

クックブック 完全なコードサンプルがクックブックの text_output/tabstops_in_textflow トピックにあります。

注 複雑な表を作るには PDFlib の表機能を推奨します（226 ページ「8.3 表の組版」参照）。

8.2.5 番号リストと段落間隔

インラインオプション *leftindent* を用いて番号リストを組む方法を下記に示します（図8.19参照）。

```
1.<leftindent 10>Long Distance Glider: With this paper rocket you can send all
your messages even when sitting in a hall or in the cinema pretty near the back. ↵
<leftindent 0>2.<leftindent 10>Giant Wing: An unbelievable sailplane! It is amazingly
robust and can even do aerobatics. But it is best suited to gliding. ↵
<leftindent 0>3.<leftindent 10>Cone Head Rocket: This paper arrow can be thrown with big
swing. We launched it from the roof of a hotel. It stayed in the air a long time and
covered a considerable distance.
```

クックブック 箇条書きリストと番号リストの完全なコードサンプルがクックブックの text_output/bulleted_list・text_output/numbered_list トピックにあります。

インデント値を設定したり取り消したりしなければならないのが面倒です。しかも各段落ごとに行わなければならないのですからなおさらです。もっとエレガントな解決法としては *list* というマクロを定義します。あわせて便宜のためマクロ *indent* を定義し、定数として使用します。下記のようなマクロの定義になります。

```
<macro {
indent {25}

list {parindent=-&indent leftindent=&indent hortabsize=&indent
hortabmethod=ruler ruler={&indent}}
}>
<&list>1. →  Long Distance Glider: With this paper rocket you can send all your messages
even when sitting in a hall or in the cinema pretty near the back. ↵
2. →  Giant Wing: An unbelievable sailplane! It is amazingly robust and can even do
aerobatics. But it is best suited to gliding. ↵
3. →  Cone Head Rocket: This paper arrow can be thrown with big swing. We launched
it from the roof of a hotel. It stayed in the air a long time and covered a
considerable distance.
```

leftindent オプションで左余白からの間隔を指定しています。*parindent* オプションには、*leftindent* を負値にしたものを設定し、各段落の先頭行のインデントを打ち消しています。オプション *hortabsize*・*hortabmethod*・*ruler* では、*leftindent* に合わせたタブ位置を指定

図 8.18
テキストフローを２つのはめ込み枠に配置

1. Long Distance Glider: With this paper rocket you can send all your messages even when sitting in a hall or in the cinema pretty near the back.
2. Giant Wing: An unbelievable sailplane! It is amazingly robust and can even do aerobatics. But it is best suited to gliding.
3. Cone Head Rocket: This paper arrow can be thrown with big swing. We launched it from the roof of a hotel. It stayed in the air a long time and covered a considerable distance.

図 8.19
番号リスト

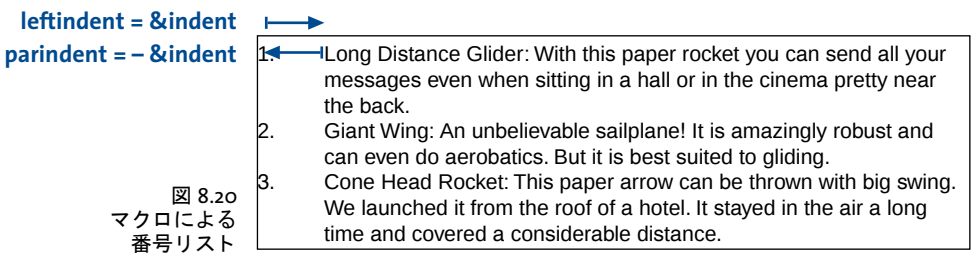


図 8.20
マクロによる
番号リスト

しています。これによって、番号の後のテキストは、*leftindent* で指定した分だけインデントされるようになります。図 8.20 に *parindent*・*leftindent* オプションの作用を示します。

二つの段落の間隔を設定　多くの場合、となりあった段落の間隔は、段落の中の行間よりも、広くとりたいものです。これを実現するには、空の行を挿入して（*nextline* オプションで作成できます）、その空行に適切な行送り値を設定します。この値は、直前の段落の最終行のベースラインと、空行のベースラインとの間隔です。下記の作成例は、2 つの段落の間に 80% のアキをとります（ここで 100% は、もっとも最近に設定された文字サイズの値に等しい）。

```
1. → Long Distance Glider: With this paper rocket you can send all your messages
even when sitting in a hall or in the cinema pretty near the back.
<nextline leading=80%><nextparagraph leading=100%>2. → Giant Wing: An unbelievable
sailplane! It is amazingly robust and can even do aerobatics. But it is best suited to
gliding.
```

クックブック　完全なコードサンプルがクックブックの text_output/distance_between_paragraphs トピックにあります。

8.2.6 制御キャラクタとキャラクタマッピング

テキストフロー内で制御キャラクタ　テキストフローの中ではさまざまなキャラクタが特別な扱いを受けます。PDFlib はシンボリックキャラクタ名に対応しており、これはその文字コードのかわりとして *charmapping* オプション（テキストを処理する前にその中のキャラクタを置換できるオプション。後述）内で用いることができます。表 8.1 に、テキストフロー関数群が評価するすべての制御キャラクタを、それぞれのシンボリック名と意味説明とともに挙げてあります。1 つのオプションリストの中で同じオプションを複数回使うことはできませんが、複数のオプションリストを連続して与えることは可能です。たとえば次の並びは空行を作成します。

```
<nextline><nextline>
```

キャラクタ・連続キャラクタのマッピング / 除去　*charmapping* オプションを使うと、テキスト内のキャラクタを別のキャラクタにマップしたり除去したりすることができます。まずは簡単な場合から始めることにして、テキスト内のすべてのタブをスペースにマップしてみましょう。これを行うには *charmapping* オプションを次のようにします。

```
charmapping={hortab space}
```

このコマンドではシンボリックキャラクタ名 *hortab*・*space* を用いています。すべてのキャラクタ名の一覧を *PDFlib API リファレンス* に挙げてあります。複数のマッピングを一

表 8.1 テキストフロー内の制御キャラクタとその意味

Unicode キャラクタ	実体名	等価なテキストフローオプション	テキストフロー内における意味
U+0020	SP, space	space	単語揃え・改行
U+00A0	NBSP, nbsp	（なし）	（no-break space）改行しない空白文字
U+0009	HT, hortab	（なし）	水平タブ。ruler・tabalignchar・tabalignment オプションに従って処理されます
U+002D	HY, hyphen	（なし）	単語のハイフネーションのための区切り文字
U+00AD	SHY, shy	（なし）	（soft hyphen）ハイフネーション機会。改行箇所でのみ表示される
U+000B U+2028	VT, verttab LS, linesep	nextline	（next line）次の行へ移る
U+000A U+000D U+000D・ U+000A U+0085 U+2029	LF, linefeed CR, return CRLF NEL, newline PS, parasep	nextparagraph	（next paragraph）nextline と同効果。それに加え、parindent オプションが次の行に影響します
U+000C	FF, formfeed	return	PDF_fit_textflow( ) 関数が中断して文字列 _nextpage を返します。

度に行うには、次のコマンドのようにすれば、すべてのタブと改行キャラクタ列をスペースに置換することができます。

```
charmapping={hortab space    CRLF space    LF space    CR space}
```

次のコマンドはすべてのソフトハイフンを削除します。

```
charmapping={shy {shy 0}}
```

次のコマンドはタブ 1 つにつきスペース 4 つに置換します。

```
charmapping={hortab {space 4}}
```

次のコマンドは任意長の連続linefeedキャラクタ列を1個のlinefeedキャラクタに縮めます。

```
charmapping={linefeed {linefeed -1}}
```

次のコマンドは CRLF キャラクタ列をそれぞれ 1 個のスペースに置換します。

```
charmapping={CRLF {space -1}}
```

この最後の例についてもう少し詳しく見てみましょう。たとえばどこかからテキストを受け取った時、何か他のソフトウェアのせいで行の途中に改行がブツブツ入っていたとしたら、そのままでは適切に組版ができません。適切にはめ込み枠内での組版が行えるようにするには、これらの改行をスペースに置換したいところです。これを行うため、任意長の連続改行を 1 個のスペースに置換しましょう。はじめのテキストは次のようなものだとします。

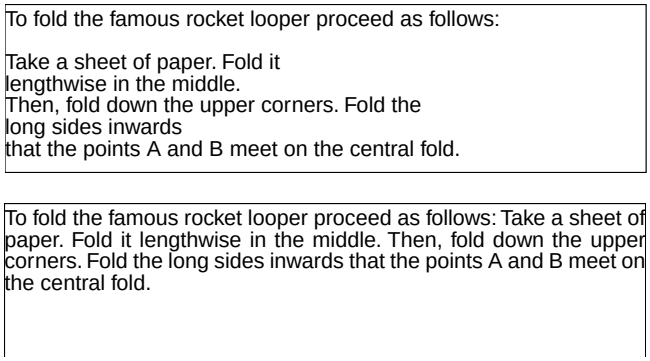


図 8.21
上：無駄な改行のあるテキスト

下：charmapping オプションで改行を置換したもの

```
To fold the famous rocket looper proceed as follows: ↵ ↵
Take a sheet of paper. Fold it ↵
lengthwise in the middle. ↵
Then, fold down the upper corners. Fold the ↵
long sides inwards ↵
that the points A and B meet on the central fold.
```

次のコードは、無駄な改行キャラクタを置換したうえでできたテキストを組版する方法を例示するものです。

```
/* オプションリストを作成 */
String  optlist = "fontname=Helvetica fontsize=9 encoding=winansi alignment=justify "
                  "charmapping {CRLF {space -1}}"
/* テキストフローをはめ込み枠に配置 */
textflow = p.add_textflow(-1, text, optlist);
if (textflow == -1)
        throw new Exception("エラー：" + p.get_errmsg());

result = p.fit_textflow(textflow, left_x, left_y, right_x, right_y, "");
if (!result.equals("_stop"))
        { /* ... */ }

p.delete_textflow(textflow);
```

図 8.21 に、処置前のテキストのテキストフロー出力と、*charmapping* オプションによる改善後の出力とを示します。

8.2.7 ハイフネーション

PDFlib は自動的にテキストのハイフネーションを行う能力は持ちませんが、テキスト内でソフトハイフンキャラクタによって明示的にハイフネーション機会が示されている場合にはそこで単語を分割することができます。ソフトハイフンキャラクタは Unicode では位置 *U+00AD* にありますが、非 Unicode 環境でソフトハイフンを指定したい場合にも以下のようないくつかの方式が利用可能です。

- *cp1250* ～ *cp1258*（*winansi* を含む）と *iso8859-1* ～ *iso8859-16* のすべてのエンコーディングではソフトハイフンは 10 進で 173、8 進で 255、16 進で 0xAD にあります。
- *ebcdic* エンコーディングではソフトハイフンは10進で202、8進で312、16進で0xCAにあります。

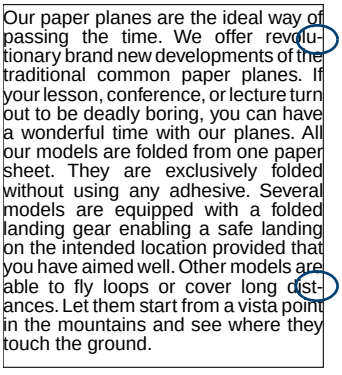


図 8.22
テキスト両端揃えでソフトハイフンあり。
デフォルト設定と広いはめ込み枠を使用。

図 8.23
テキスト両端揃えでソフトハイフンなし。
デフォルト設定と広いはめ込み枠を使用。

- エンコーディングがソフトハイフンキャラクタを含んでいない場合は（たとえば *macroman*）、** を用いることができます。

U+002D がハイフンキャラクタとして使われます。ソフトハイフンで示されたハイフネーション機会でなくても、単語は強制的にハイフネーションされることがあります。これは単語間隔伸縮や長体など、他の調整手段がうまくいかなかった極端な場合に起こります。

テキスト両端揃えでハイフンキャラクタがある場合とない場合　次の例では、次のテキストを両端揃えで印字させます。テキストにはソフトハイフンキャラクタを適宜入れてあります（ダッシュで示します）。

```
Our paper planes are the ideal way of pas – sing the time. We offer revolu – tionary
brand new dev – elop – ments of the tradi – tional common paper planes. If your lesson,
confe – rence, or lecture turn out to be deadly boring, you can have a wonder – ful time
with our planes. All our models are folded from one paper sheet. They are exclu – sively
folded without using any adhe – sive. Several models are equip – ped with a folded
landing gear enab – ling a safe landing on the intended loca – tion provided that you
have aimed well. Other models are able to fly loops or cover long dist – ances. Let them
start from a vista point in the mount – ains and see where they touch the ground.
```

図 8.22 に、テキスト両端揃えのデフォルト設定で生成されたテキスト出力を示します。完璧な見ばえとなっています。なぜなら条件が最適だからです。すなわちはめ込み枠枠が充分広くて、しかも、明示的なハイフネーション機会をソフトハイフンキャラクタで指定してあるからです。図 8.23 を見ると、明示的ソフトハイフンがない場合でも出力はおおむね良好です。オプションリストはどちらの場合も次のようになります。

```
fontname=Helvetica fontsize=9 encoding=winansi alignment=justify
```

8.2.8 標準改行アルゴリズムの制御

PDFlib は洗練された改行アルゴリズムを実装しています。改行アルゴリズムを制御するテキストフローオプションを表 8.2 に挙げます

改行規則　1 つの単語、ないし両端をスペースキャラクタで挟まれた一連のテキストが、1 つの行に収まりきらない場合、次の行へ送る必要が出てきます。このような状況で改行アルゴリズムは、どのキャラクタの後で改行が可能かを決定します。

たとえば、*-12+235/8*45* のような数式は絶対に途中で改行されませんが、一方、文字列 *PDF-345+LIBRARY* はマイナスの所で次行に送られる可能性があります。テキストがソフトハイフンキャラクタを含んでいればそのうちのいずれかの後で改行される可能性もあります。

括弧と引用符については、開く所と閉じる所とで規則が異なります。括弧や引用符が開く所では改行機会は一切与えられません。引用符については、どれが開いてどれが閉じているのかを判定する手段として、引用符を 2 つずつ数えていく仕組みになっています。

表 8.2 改行アルゴリズム制御用オプション一覧

オプション	説明
adjust-method	（キーワード）minspacing・maxspacing オプションで指定された制限内で単語間を詰めたり拡げたりしてもテキストが 1 行に収まりきらない時に行の調整に用いる方式。デフォルト：auto。 ***auto*** shrink・spread・nofit・split 方式を順に適用。 ***clip*** nofit（後述）と同じ。ただし、はめ込み枠の右端（rightindent オプションを考慮）からはみ出した部分を切り落とし。 ***nofit*** 最後の単語を次行へ送る。ただし、残される（短い）行が、nofitlimit オプションで指定されたパーセント値よりも短くならない場合に限る。均等配置の段落でも若干がたついて見える場合あり。 ***shrink*** 単語が行内に収まらないとき、収まるまでテキストを圧縮。ただし shrinklimit の制限に従い、それで収まりきらなければ nofit 方式を適用。 ***split*** 最後の単語を次行へ送らず、強制的にハイフネーションする。テキストフォントの場合はハイフンキャラクタを挿入し、記号フォントの場合はしない。 ***spread*** 最後の単語を次行へ送り、残された（短い）行を均等配置するよう単語内の字間を拡げる。ただし spreadlimit の制限に従い、それで均等配置できなければ nofit 方式を適用。
advanced-linebreak	（論理値）複雑用字系で必要な高度な改行アルゴリズムを有効にします。これはタイ文字等、単語境界を示すのに空白キャラクタを使わない用字系で改行を行うために必要です。オプション locale・script に従います。デフォルト：false
avoidbreak	（論理値）true なら、avoidbreak が false に再設定されるまで一切改行しない。デフォルト：false。
charclass	（対のリスト。ここで対の 1 番目の要素はキーワードであり、2 番目の要素は Unichar または Unichar のリスト）指定された Unichar が、指定されたキーワードに分類されます。キーワードは、そのキャラクタの改行時動作を下記のように決定します。 ***letter*** 文字（a B 等）と同様に動作 ***punct*** 句読点文字（+ / ; : 等）と同様に動作 ***open*** 開きかっこ（[等）と同様に動作 ***close*** 閉じかっこ（] 等）と同様に動作 ***default*** すべてのキャラクタ分類を PDFlib 内蔵のデフォルトにリセット 例：charclass={ close » open « letter {/ : =} punct & }
hyphenchar	（Unichar またはキーワード）改行箇所でソフトハイフンに置き換わるべきキャラクタの Unicode 値。デフォルト：U+00AD（SOFT HYPHEN）、ただしそれがフォント内になければ U+002D（HYPHEN-MINUS）。

表 8.2 改行アルゴリズム制御用オプション一覧

オプション	説明
locale	（キーワード）advancedlinebreak= true の場合に、ローカライズされた改行方式のために用いられるロケール。キーワードは 1 個ないし複数の構成要素から成っており、その中でオプショナルなコンポーネントは下線キャラクタ「_」で区切られます（その文法は NLS/POSIX ロケール ID とは若干異なっています。）： ▸ 必須の、ISO 639-2 に従った 2 文字か 3 文字の小文字の言語コード（www.loc.gov/standards/iso639-2 を参照）。例：en（英語）・de（ドイツ語）・ja（日本語）。これは language オプションとは異なります。 ▸ オプショナルな、ISO 15924 に従った 4 文字の用字系コード（www.unicode.org/iso15924/iso15924-codes.html を参照）。例：Hira（ひらがな）・Hebr（ヘブライ文字）・Thai（タイ文字）。 ▸ オプショナルな、ISO 3166 に従った 2 文字の大文字の国コード（www.iso.org/iso/country_codes/iso_3166_code_lists を参照）。例：DE（ドイツ）・CH（スイス）・GB（イギリス） キーワード _none は、ロケール固有の処理が一切行われないことを指定します。 ロケールを指定することは、タイ文字等いくつかの用字系では高度な改行のために必須です。デフォルト：_none 例：Thai・de_DE・en_US・en_GB
maxspacing ***minspacing***	（float またはパーセント値）単語間の最大間隔・最小間隔（ユーザー座標で表すか、スペースキャラクタの幅に対するパーセント値で指定）。単語間隔の算出時はこの値を限度とする（ただし wordspacing オプションが加算される）。デフォルト：minspacing=50%、maxspacing=500%。
nofitlimit	（float またはパーセント値）nofit 方式における行の長さの下限（ユーザー座標で表すか、収めるはめ込み枠の幅に対するパーセント値で指定）。デフォルト：75%。
shrinklimit	（パーセント値）shrink 方式におけるテキスト圧縮の下限。縮小率の算出時はこの値を限度とする。ただし、horizscaling オプションの値を掛け算される。デフォルト：85%。
spreadlimit	（float またはパーセント値）spread 方式における 2 文字間の間隔の上限（ユーザー座標で表すか、文字サイズに対するパーセント値で指定）。計算される文字間は charspacing オプションの値に加算される。デフォルト：0。

インラインオプションリストは通常は改行機会を生じません。それは単語内でのオプション変更を可能にするためです。ただし、オプションリストがスペースキャラクタで挟まれている場合はオプションリストの開始位置に改行機会があります。もしそのオプションリストで改行が起きてしかも *alignment=justify* の場合、オプションリストの前にあるスペース群は破棄されます。オプションリストの後のスペース群は保持され、次行先頭に表れます。

改行を防ぐ　*charclass* オプションを使うと、テキストフローが特定のキャラクタの後で改行されるのを防ぐことができます。たとえば、下記のオプションはキャラクタ / の直後での改行を防ぎます。

```
charclass={letter /}
```

ひとつながりのテキストが複数行に泣き別れてしまうのを防ぐには、それを *avoidbreak* ～ *noavoidbreak* でくくるという方法があります。

クックブック　**完全なコードサンプルがクックブックの** text_output/avoid_linebreaking **トピックにあります。**

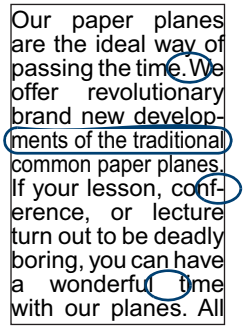


図 8.24 狭いはめ込み枠でテキスト両端揃え。デフォルト設定を用いています

日中韓テキストの組版　テキストフローエンジンは、日中韓テキストを扱えるように作られているので、Unicode 標準の通り、日中韓キャラクタを表意文字として適切に取り扱います。その結果、日中韓テキストはけっしてハイフネーションされません。組版の品質を高めるため、日中韓テキストでテキストフローを使うときは、下記の組版オプションを推奨します。こうすると、欧文テキストが混在していてもそこでハイフネーションが行われなくなり、また、均等に間隔をあけたテキスト出力が生成されます。

```
hyphenchar=none
alignment=justify
shrinklimit=100%
spreadlimit=100%
```

縦書きはテキストフローでは対応していません。

狭いはめ込み枠でテキスト両端揃え　はめ込み枠が狭ければ狭いほど、両端揃えのテキストを制御するためのオプションが重要になっていきます。図 8.24 は、狭いはめ込み枠でテキストがさまざまな方式で両端揃えされた出力結果を例示しています。図 8.24 のオプション設定は基本的におおむね良好ですが、ただ、*maxspacing* がやや広すぎる単語間隔を与えているのが気になります。とはいえ、狭いはめ込み枠に対してはこれはこのままにしておくことを推奨します。でないと、*split* 方式による見苦しい強制ハイフネーションの発生頻度が高まるでしょう。

　はめ込み枠が狭すぎるために不適切な箇所が強制ハイフネーションされてしまう場合は、対処を考慮して、ソフトハイフンを入れるなり、テキスト両端揃えを制御するオプションを変えるなりする必要があるでしょう。

テキスト両端揃えで shrinklimit オプション　見た目にもっとも受け入れやすい解決策は *shrinklimit* オプションを小さくすることでしょう。このオプションは、*shrink* 方式でかかる長体の割合の下限を指定するものです。図 8.25a は、テキストに *shrinklimit=50%* まで長体をかけることで強制ハイフネーションを防いでいる様子を示しています。

テキスト両端揃えで spreadlimit オプション　字間を拡げることで改行を制御するのも一つの方法です。これは *spread* 方式で行われ、*spreadlimit* オプションで制御されます。しかしこの方式は美しくないのでめったに使われないでしょう。図 8.25b は、*spreadlimit=5* を使って、字間の最大を非常に広く 5 単位とした例です。

図 8.25 狭いはめ込み枠内の両端揃えテキストのためのオプション

生成される出力		PDF_fit_textline( ) に与えるオプションリスト
a)	passing the time. We offer revolutionary brand new developments of the traditional common paper planes. If your lesson, conference, or lecture turn out to	alignment=justify shrinklimit=50%
b)	Our paper planes are the ideal way of passing the time. We offer revolutionary b r a n d n e w developments of the	alignment=justify spreadlimit=5
c)	ments of the traditional common paper planes. If your lesson, conference, or lecture turn out to be deadly boring, you can have	alignment=justify nofitlimit=50

テキスト両端揃えで nofitlimit オプション　*nofitlimit* オプションは、*nofit* 方式が適用されたときの行の最小の幅を制御するものです。はめ込み枠が非常に狭い場合は、これをデフォルト値 75% から下げたほうが強制ハイフネーションよりはましです。図 8.25c は、行の最小幅 50% を指定した場合の出力結果を示しています。

8.2.9 高度な用字系固有の改行

PDFlib は、標準の改行アルゴリズムに加えて、追加の改行アルゴリズムを実装しています。この高度な改行アルゴリズムは、いくつかの用字系では必須であり、また、必須でないその他の用字系 / ロケールの組み合わせのなかにも、これにより改行動作が改善されるものがあります。これは *advancedlinebreak* オプションで有効にすることができます。改行はテキストの言語に依存しますので、高度な改行アルゴリズムは *script* オプション（表 6.2 参照）と *locale* オプション（PDFlib API リファレンス参照）に従います。高度な改行は、下記の状況において正しい改行機会を決定します：

- タイ文字等、テキスト内の空白キャラクタの存在に改行が依拠しない用字系に対して。下記のテキストフローオプションは、タイ文字に対して高度な改行を有効にします：

  ```
  <advancedlinebreak script=thai locale=THA>
  ```

- 仏文テキスト内で引用符として用いられる «» ギユメキャラクタ等、ある特定の句読点キャラクタの特別な扱いを必要とする用字系 / ロケールの組み合わせにおいて。下記のテキストフローオプションは、仏文テキストに対して高度な改行を有効にします。その結果、単語を囲うギユメキャラクタが行末で単語と泣き別れしなくなります：

  ```
  <advancedlinebreak script=latn locale=fr>
  ```

locale テキストフローオプションは *language* テキストオプション（表 6.3 参照）と違うことに留意してください：*locale* オプションは同じ 3 文字の言語識別子で始まることができますが、1 個ないし 2 個の追加部分を任意に含むこともできます。ただし、これらが PDFlib で必要になることはまれです。

8.2.10 テキストをパス・画像に回り込み

回り込み機能を利用すると、任意の形状にテキストを入れたり、テキストをパスに回り込ませることができます。範囲枠、明示的な矩形 / 多角形 / 円 / 曲線、パスオブジェクトのいずれかを用いて、テキストフローに対する回り込み領域を指定することができます。画像がクリッピングパスを内蔵している場合には、テキストをその画像のクリッピングパスに自動的に回り込ませることも可能です。

範囲枠を使ったテキストの画像回り込み　最初の作成例として、テキストフローの中に画像を配置して、テキストをその画像全体のまわりに回り込ませてみましょう。まず画像を読み込み、枠内の希望の位置に配置します。この画像を後で名前で参照するために、下記のようにオプションリスト *matchbox={name=img margin=-5}* で、画像をはめ込む際に *img* という範囲枠を定義し、5 単位の余白を指定しましょう：

```
result = p.fit_image(image, 50, 35,
        "boxsize={80 46} fitmethod=meet position=center matchbox={name=img margin=-5}");
```

テキストフローを追加します。そしてそれを、次のように、画像のはめ込み枠 *img* を回り込むべき領域として *wrap* オプションを使って配置しましょう（図 8.26 参照）。

```
result = p.fit_textflow(textflow, left_x, left_y, right_x, right_y,
        "wrap={usematchboxes={{img}}}");
```

テキストを配置する前に、同じ範囲枠名を使ってさらにほかの画像をはめ込んでいくこともできます。その場合、テキストはすべての画像を回り込みます。

クックブック　**完全なコードサンプルがクックブックの** `text_output/wrap_text_around_images` **トピックにあります。**

テキストを任意のパスに回り込み　パスオブジェクトを作成して（70 ページ「3.2.3 直接パスとパスオブジェクト」参照）、それを回り込み形状として用いることもできます。下記のコードは、単純な形状（1 個の円）のパスオブジェクトを構築し、それを *PDF_fit_textflow()* の *wrap* オプションに与えています。パスを配置する参照点は、はめ込み枠の幅と高さに対するパーセント値として表現されています：

```
path = p.add_path_point(  -1,   0, 100, "move", "");
path = p.add_path_point(path, 200, 100, "control", "");
path = p.add_path_point(path,   0, 100, "circular", "");

/* パスを目に見えるようにしたければそうする */
p.draw_path(path, x, y, "stroke");
```

図 8.26
範囲枠を使ったテキストの画像回り込み

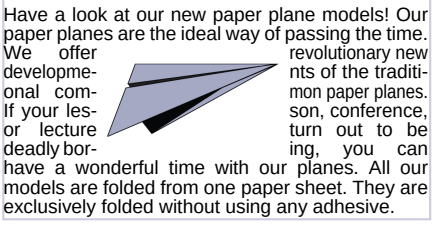
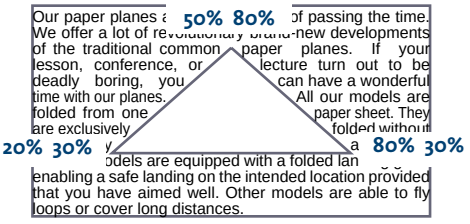


図 8.27
テキストを三角形のまわりに回り込み



```
result = p.fit_textflow(tf, llx1, lly1, urx1, ury1,
        "wrap={paths={" +
            "{path=" + path + " refpoint={100% 50%} }" +
        "}}");

p.delete_path(path);
```

inversefill オプションを用いると、テキストをパスの外側に回りこませるのではなく、パスの内側に回りこませることが可能です（すなわち、パスはテキストフロー内に穴をあけるのではなく、テキストの入れ物として機能します）：

```
result = p.fit_textflow(tf, llx1, lly1, urx1, ury1,
        "wrap={inversefill paths={" +
            "{path=" + path + " refpoint={100% 50%} }" +
        "}}");
```

テキストを画像のクリッピングパスに回り込ませる　TIFF・JPEG 画像は、クリッピングパスを内蔵することができます。このパスは、画像処理アプリケーションで作成されている必要があり、PDFlib はこれを評価します。デフォルトクリッピングパスが画像内で見つかればそれが使われますが、画像内の他の任意のクリッピングパスを *PDF_load_image()* の *clippingpathname* オプションで指定することもできます。画像がクリッピングパスとともに読み込まれていれば、そのパスを抽出して先述のように *PDF_fit_textflow()* の *wrap* オプションに与えることができます。取り込んだ画像のクリッピングパスを拡大するために *scale* オプションも与えます：

```
image = p.load_image("auto", "image.tif", "clippingpathname={path 1}");

/* 画像のクリッピングパスからパスオブジェクトを作成 */
path = (int) p.info_image(image, "clippingpath", "");
if (path == -1)
        throw new Exception("エラー： クリッピングパスが見つかりません!");

result = p.fit_textflow(tf, llx1, lly1, urx1, ury1,
        "wrap={paths={{path=" + path + " refpoint={50% 50%} scale=2}}}");

p.delete_path(path);
```

画像を配置してテキストをそれに回り込ませる　前の項では画像のクリッピングパスだけを使いました（画像自体ではなく）が、今度は画像をはめ込み枠の中に配置して、テキストをそれに回り込ませてみましょう。これを実現するには、今度も画像を *clippingpathname* オプションをつけて読み込んで、それを *PDF_fit_image()* でページ上に配置する必要があります。テキストを回り込ませるのに適切なパスオブジェクトを作成するために、*PDF_fit_image()* と同じオプションリストをつけて *PDF_info_image()* を呼び出しましょう。最後に、参照点（*PDF_fit_image()* の *x/y* 引数）を *wrap* オプションの *paths* サブオプションの *refpoint* サブオプションに与える必要があります：

```
image = p.load_image("auto", "image.tif", "clippingpathname={path 1}");

/* 画像を何らかのはめ込みオプション群でページ上に配置 */
String imageoptlist = "scale=2";
p.fit_image(image, x, y, imageoptlist);

/* 同じオプションリストを用いて、画像からパスオブジェクトを作成 */
path = (int) p.info_image(image, "clippingpath", imageoptlist);
```

```
if (path == -1)
        throw new Exception("エラー： クリッピングパスが見つかりません!");

result = p.fit_textflow(tf, llx1, lly1, urx1, ury1,
        "wrap={paths={{path=" + path + " refpoint={" + x + " " + y + "} }}}");

p.delete_path(path);
```

PDF_fit_textflow() を複数回呼び出して同一の *wrap* オプションを与えることもできます。これは多段組等、配置された画像が複数のテキストフローはめ込み枠に重なっているときに有用です。

テキストを非矩形形状に回り込ませる　テキストは、範囲枠で指定される矩形を回り込ませるだけでなく、任意のグラフィック要素を回り込み輪郭として定義することもできます。たとえば下記のオプションリストは、三角形のまわりにテキストを回り込ませます（図 8.27 参照）：

```
wrap={ polygons={ {50% 80%   20% 30%   80% 30%   50% 80%} } }
```

なお、*showborder=true* オプションを使って輪郭を図示してあります。*wrap* オプションは複数の輪郭を持つこともできます。下記のオプションリストは、2 つの三角形のまわりにテキストを回り込ませます。

```
wrap={ polygons={ {50% 80%   20% 30%   80% 30%   50% 80%}
                  {20% 90%   10% 70%   30% 70%   20% 90%} } }
```

パーセント値（はめ込み枠内における相対座標）のかわりに、ページ上の絶対座標を使うこともできます。

注　水平でも垂直でもない向きの線分を持つ輪郭を使うときは、fixedleading=true に設定することを推奨します。

クックブック　完全なコードサンプルがクックブックの text_output/wrap_text_around_polygons トピックにあります。

非矩形の輪郭へ流し込み　回り込み機能を使えば、任意の輪郭の領域の中にテキストフローを配置することもできます。これを実現するには、*wrap* オプションで *addfitbox* または *inversefill* サブオプションを使います。テキストは指定された輪郭のまわりに回り込むのではなく、輪郭（複数可）の中にテキストが配置されます。下記のオプションリストを使えば、ひし形の中へテキストを流し込む¥ことができます。ここで座標は、はめ込み枠の矩形に対するパーセント値として与えています（図 8.27 参照）。

```
wrap={ addfitbox polygons={ {50% 100%   10% 50%   50% 0%   90% 50%   50% 100%} } }
```

なお、ここでも *showborder=true* オプションを使って輪郭を図示してあります。もし *addfitbox* オプションをつけなければ、ひし形は空のまま、そのまわりにテキストが回り込むことになります。

重なり合う輪郭へ流し込み　次の例として、重なり合う 2 つの多角形から成る輪郭へ流し込みを行なってみましょう。たとえば六角形の中に四角形が入った輪郭です。*addfitbox* オプションを使えば、はめ込み枠自体は流し込みの範囲から除外され、その後のリストの中の多角形は、重なり合っている領域を除いて流し込みが行われます（図 8.28 参照）。

図 8.27
ひし形へテキストを流し込み

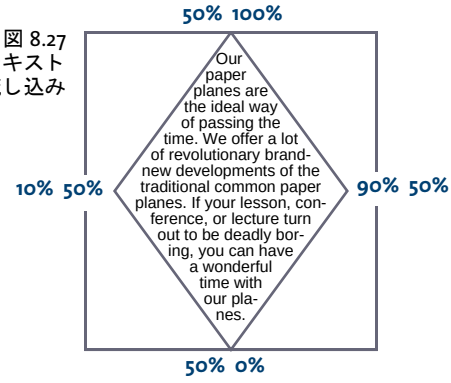
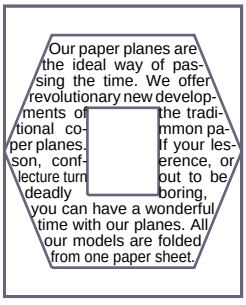


図 8.28
重なり合う輪郭へ流し込み




```
wrap={ addfitbox polygons=
       { {20% 10%   80% 10%   100% 50%   80% 90%   20% 90%   0% 50%   20% 10%}
         {35% 35%   65% 35%   65% 65%   35% 65%   35% 35%} } }
```

もし ***addfitbox*** オプションをつけなければ、これと反対の効果を得ます。すなわち、さっき流し込まれた領域は空のままとなり、さっき空だった領域へテキストが流し込まれることになります。

クックブック 完全なコードサンプルがクックブックの`text_output/fill_polygons_with_text`トピックにあります。

8.3 表の組版

表組版機能を使うと、複雑な表を自動組版することができます。表のセルには、一行ないし複数行のテキストや、画像、PDF グラフィックを入れることができます。表は 1 個のはめ込み枠に収まらなくてもよく、複数のページにわたることが可能です。

クックブック　表の諸側面に関するコードサンプルが PDFlib クックブックの tables カテゴリにあります。

表のあらまし　表整形機能の説明は、以下の概念と用語にもとづきます（図 8.29 参照）。

- **表**は、矩形の輪郭を持つ仮想のオブジェクトです。横方向の表**行**と縦方向の**列**でできています。
- **単純セル**は、表内の矩形の領域であり、表行と列の交差として定義されます。**連結セル**は、複数の列か複数の表行、ないし両方にわたっています。**セル**という用語を用いるときは、単純セルと連結セルの両方を指すものとします。
- 表は、1 つのはめ込み枠に収まりきることもありますし、複数のはめ込み枠が必要になることもあります。1 つのはめ込み枠に配置された表行群は、**表インスタンス**を構成します。*PDF_fit_table()* は、1 回呼び出されるごとに、1 つのはめ込み枠に 1 つの表インスタンスを配置します（236 ページ「8.3.5 表インスタンス」参照）。
- **ヘッダ**と**フッタ**は、表の最初か最後にある表行（複数可）のかたまりであり、すべての表インスタンスの上端か下端に繰り返し現れます。ヘッダにもフッタにも属さない表行は**本体表行**と呼びます。

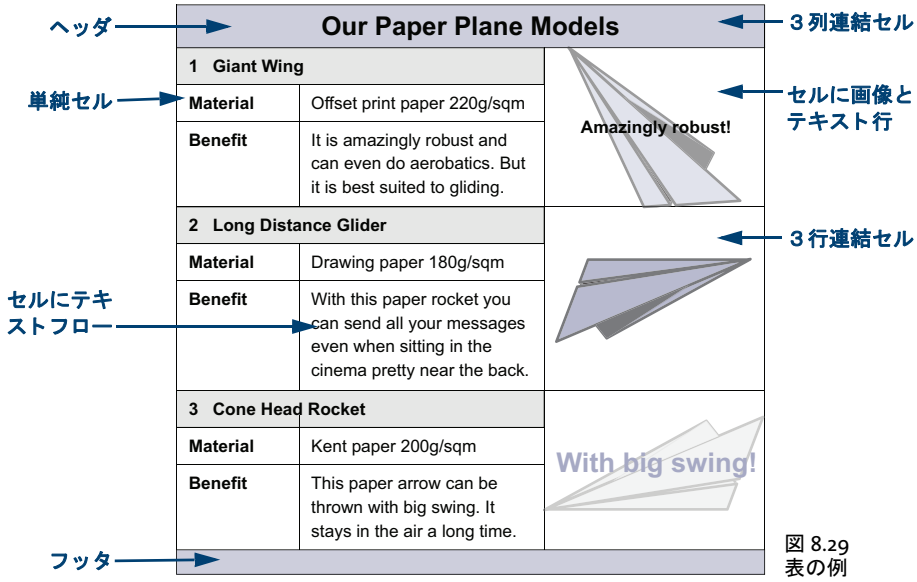


図 8.29
表の例

例として、図 8.29 の表のすべての要素の作り方を説明していきます。表組版オプションの完全な説明については *PDFlib API リファレンス*を参照してください。表の作成はまず、各表セルの内容と視覚的プロパティを *PDF_add_table_cell()* で定義していくことから始まります。それから、*PDF_fit_table()* を 1 回ないし複数回呼び出して、その表を配置します。

表を配置する際には、そのはめ込み枠の大きさと、その表行や列の罫線と塗り分けを指定することもできます。セルごとの塗り分けなどの細かい指定には、範囲枠機能を使ってください（詳しくは 242 ページ「8.4 範囲枠」参照）。

この節では、表セルの定義と表のはめ込みに必要な、もっとも重要なオプションのみを解説します。いずれの例においても、そのための *PDF_add_table_cell()* と *PDF_fit_table()* への呼び出しだけを示しますが、必要なフォントはすでに読み込まれているものとします。

注 表の処理は、カレントグラフィック状態からは独立です。表セルの定義は文書スコープでもできますが、実際に表を配置するのはページスコープで行う必要があります。

クックブック 完全なコードサンプルがクックブックの tables/starter_table **トピックにあります。**

8.3.1 単純な表を配置

表の概念をさらに詳しく説明する前に、単純な表作成の例を示します。表には 6 つのセルがあり、3 表行・2 列に配されています。4 つのセルにはテキスト行があり、1 つのセルには複数行のテキストフローがあります。セルの内容はすべて、余白を 1 単位とって横方向に左寄せ、縦方向に中央揃えになっています。

この表を作成するために、まずはテキスト行セルに対するオプションリストを作るために、その *fittextline* サブオプションリストで、必要なオプション *font*・*fontsize* と位置 *{left center}* を定義しましょう。さらに、1 単位のセル余白を定義しましょう。そして、テキスト行セルを 1 つずつそれぞれの列・表行に追加し、その際に中身のテキストも、*PDF_add_table_cell()* への呼び出しに直接与えます。

次に、テキストフローを作成し、そのテキストフローのハンドルを使ってテキストフロー表セルに対するオプションリストを構築した後、そのセルを表に追加しましょう。

最後に、*PDF_fit_table()* を使って表を配置し、その際に、表の外枠とセルの各辺に黒い罫線をつけましょう。列の幅は一切与えませんでしたので、与えたテキスト行と余白から自動的に計算されます。

クックブック 完全なコードサンプルがクックブックの tables/vertical_text_alignment **トピックにあります。**

下記のコードは、この単純な表の作り方を示します。結果を図 8.30a に示します。

```
/* 複数行のテキストフローで表セルに入れるテキスト */
String tf_text = "It is amazingly robust and can even do aerobatics. " +
                 "But it is best suited to gliding.";

/* 1列目と2列目の列幅を定義 */
int c1 = 80, c2 = 120;

/* 表インスタンスの左下隅と右上隅を定義（はめ込み枠）*/
double llx=100, lly=500, urx=300, ury=600;

/* フォントを読み込む */
font = p.load_font("Helvetica", "unicode", "");
if (font == -1)
        throw new Exception("エラー：" + p.get_errmsg());

/* 1列目に配置するテキスト行セルに用いるオプションリストを定義 */
optlist = "fittextline={position={left center} font=" + font + " fontsize=8} margin==4" +
          colwidth=" + c1;

/* 列1表行1にテキスト行セルを追加 */
tbl = p.add_table_cell(tbl, 1, 1, "Our Paper Planes", optlist);
if (tbl == -1)
```

```
        throw new Exception("エラー : " + p.get_errmsg());

/* 列1表行2にテキスト行セルを追加 */
tbl = p.add_table_cell(tbl, 1, 2, "Material", optlist);
if (tbl == -1)
        throw new Exception("エラー : " + p.get_errmsg());

/* 列1表行3にテキスト行セルを追加 */
tbl = p.add_table_cell(tbl, 1, 3, "Benefit", optlist);
if (tbl == -1)
        throw new Exception("エラー : " + p.get_errmsg());

/* 2列目に配置するテキスト行に用いるオプションリストを定義 */
optlist = "fittextline={position={left center} font=" + font + " fontsize=8} " +
          "colwidth=" + c2 + " margin=4";

/* 列2表行2にテキスト行セルを追加 */
tbl = p.add_table_cell(tbl, 2, 2, "Offset print paper 220g/sqm", optlist);
if (tbl == -1)
        throw new Exception("エラー : " + p.get_errmsg());

/* テキストフローを追加 */
optlist = "font=" + font + " fontsize=8 leading=110%";
tf = p.add_textflow(-1, tf_text, optlist);

/* 上で取得したハンドルを使ってテキストフローセルに用いるオプションリストを定義 */
optlist = "textflow=" + tf + " margin=4 colwidth=" + c2;

/* 列2表行3にテキストフロー表セルを追加 */
tbl = p.add_table_cell(tbl, 2, 3, "", optlist);
if (tbl == -1)
        throw new Exception("エラー : " + p.get_errmsg());

p.begin_page_ext(0, 0, "width=200 height=100");

/* 表をはめ込むためのオプションリストを表枠とセル罫線つきで定義 */
optlist = "stroke={{line=frame linewidth=0.8} {line=other linewidth=0.3}}";

/* 表インスタンスを配置 */
result = p.fit_table(tbl, llx, lly, urx, ury, optlist);

/* 結果を調べる。「_stop」ならすべてOKを意味する。*/
if (!result.equals("_stop")) {
        if (result.equals( "_error"))
            throw new Exception("エラー : " + p.get_errmsg());
        else {
            /* それ以外の戻り値はすべて専用のコードで扱う必要がある */
        }
}
p.end_page_ext("");

/* 表内で使われたテキストフローハンドルも一緒に削除 */
p.delete_table(tbl, "");
```

セルの内容の縦位置を調整する　表セルの縦方向中央にさまざまな種類の内容を配置すると、その辺からの距離はまちまちになります。図 8.30a において、4 つのテキスト行セルは次のオプションリストで配置されています。

```
optlist = "fittextline={position={left center} font=" + font +
          " fontsize=8} colwidth=80 margin=4";
```

テキストフローセルは特殊なオプションを一切使わずに追加されています。テキスト行を縦方向中央に配置したために、*Benefit* の行がテキストフローの分だけ下へずれます。

図 8.30 テキスト行とテキストフローを表セル内で整列させる

生成される出力

a)

Our Paper Planes	
Material	Offset print paper 220g/sqm
Benefit	It is amazingly robust and can even do aerobatics. But it is best suited to gliding.

b)

Our Paper Planes	
Material	Offset print paper 220g/sqm
Benefit	It is amazingly robust and can even do aerobatics. But it is best suited to gliding.

図 8.30b に示したように、セルの辺からセルの中身までの縦間隔は、それがテキストフローであるかテキスト行であるかにかかわらず、すべて同じにしたいものです。これを実現するために、まずテキスト行のためのオプションリストを用意しましょう。表行の高さを固定値 14 ポイント、テキスト行の位置を左上で余白 4 ポイントとして定義しましょう。

さきに与えたオプション *fontsize=8* は、文字の高さを正確には表しておらず、上下にいくらかあきができています。でも、大文字の高さはフォントの *capheight* 値で正確に表されます。ですので、*fontsize={capheight=6}* を用いれば、文字サイズが結果的にほぼ 8 ポイントになり、また（*margin=4* とあわせて）高さの合計が 14 ポイントとなって *rowheight* オプションと合います。ですので全体としては、テキスト行セルに対する *PDF_add_table_cell()* のオプションリストは次のようにしましょう。

```
/* テキスト行セルに用いるオプションリスト */
optlist = "fittextline={position={left top} font=" + font +
          " fontsize={capheight=6}} rowheight=14 colwidth=80 margin=4";
```

テキストフローの追加にあたっては、上記のテキスト行同様、*fontsize={capheight=6}* を用いれば、文字サイズが結果的にほぼ 8 ポイントになり、また（*margin=4* とあわせて）高さの合計が 14 ポイントとなります。

```
/* テキストフローの追加に用いるオプションリスト */
optlist = "font=" + font + " fontsize={capheight=6} leading=110%";
```

さらに、テキスト *Benefit* のベースラインは、テキストフローの 1 行目に整列させたいものです。と同時に、テキスト *Benefit* のセル上端からの間隔は、テキスト *Material* と同じになるべきです。上端に余白を生じさせないために、テキストフローセルの追加にあたっては *fittextflow={firstlinedist=capheight}* を用いましょう。そしてテキスト行と同じく、余白 4 ポイントを追加しましょう。

```
/* テキストフローセルの追加に用いるオプションリスト */
optlist = "textflow=" + tf + " fittextflow={firstlinedist=capheight} "
          "colwidth=120 margin=4";
```

クックブック　完全なコードサンプルがクックブックの `tables/vertical_text_alignment` トピックにあります。

8.3.2 表セルのさまざまな内容

PDF_add_table_cell( ) で表にセルを追加するときは、さまざまな種類のセル内容を指定することができます。表セルは、同時に複数の種類の内容を含むこともできます。罫線・塗りの追加も可能なほか、範囲枠を使って表セル内に追加の内容を配置することもできます。

たとえば紙飛行機の表には、図 8.31 に示す要素があります。

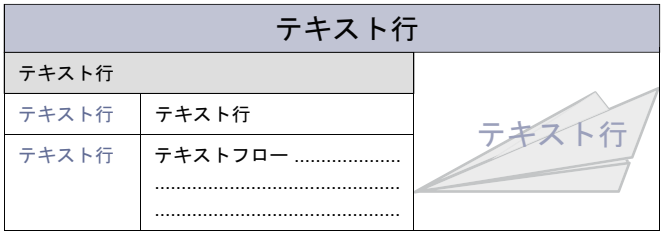


図 8.31
表セルのさまざまな内容

テキスト行による一行テキスト　テキストは、*PDF_add_table_cell( )* の *text* 引数で与えます。*fittextline* オプションで、*PDF_fit_textline( )* の組版オプションのすべてを与えることができます。デフォルトのはめ込み方式は *fitmethod=nofit* です。テキストがセルに収まりきらないときは、セルが大きくなります。これを避けるには、*fitmethod=auto* を使えば、*shrinklimit* オプションの範囲内でテキストが縮まります。表行の高さが指定されなかった場合は、組版機能はテキストサイズの 2 倍を表セルの高さとします（より正確にいうと *boxheight* の 2 倍。これは、別途指定されない限りデフォルト値 *{capheight none}* を持ちます）。テキストが回転させてあるときの表行の幅についても同じです。

テキストフローによる複数行テキスト　テキストフローは、表関数の外で用意しておいて、*PDF_add_table_cell( )* を呼び出す前に *PDF_create_textflow( )* か *PDF_add_textflow( )* で作成しておく必要があります。そのテキストフローのハンドルは、*textflow* オプションで与えます。*fittextflow* オプションで、*PDF_fit_textflow( )* の組版オプションのすべてを与えることができます。

デフォルトのはめ込み方式は *fitmethod=clip* です。すなわち次のように動作します。まず、テキストがセルに収まりきるかどうかが試されます。セルの大きさが充分でないときは、その高さが増やされます。それでもテキストが収まりきらない場合は、末尾が切り落とされます。これを避けるには、*fitmethod=auto* を使えば、*minfontsize* オプションの範囲内でテキストが縮まります。

セルが狭すぎるときは、1 つの単語を好ましくない箇所で分割させるよう、テキストフローに強制することもできます。*checkwordsplitting* オプションが *true* の場合は、単語が分割されなくなるまでセル幅が拡がります。

画像とテンプレート　画像は、*PDF_add_table_cell( )* を呼び出す前に *PDF_load_image( )* で読み込んでおく必要があります。テンプレートは、*PDF_begin_template_ext( )* で作成する必要があります。その画像やテンプレートのハンドルは、*image* オプションで与えます。*fitimage* オプションで、*PDF_fit_image( )* の組版オプションのすべてを与えることができます。デフォルトのはめ込み方式は *fitmethod=meet* です。すなわち画像 / テンプレートは、縦横比を変えないまま、セル内に収まりきるよう配置されます。セルの大きさが、画像 / テンプレートの大きさにしたがって変わることはありません。

取り込み PDF 文書のページ　PDI ページは、*PDF_add_table_cell( )* を呼び出す前に *PDF_open_pdi_page( )* で開いておく必要があります。その PDI ページのハンドルは、*pdipage* オプションで与えます。*fitpdipage* オプションで、*PDF_fit_pdi_page( )* の組版オプションのすべてを与えることができます。デフォルトのはめ込み方式は *fitmethod=meet* です。すなわち PDI ページは、縦横比を変えないまま、セル内に収まりきるよう配置されます。セルの大きさが、PDI ページの大きさにしたがって変わることはありません。

パスオブジェクト　パスオブジェクトは、*PDF_add_table_cell( )* を呼び出す前に *PDF_add_path_point( )* で作成されている必要があります。そのパスハンドルは *path* オプションで与えられます。*fitpath* オプションでは、*PDF_draw_path( )* のすべての組版オプションを指定可能です。パスの外接枠が表セル内に配置されます。セル内枠の左下隅が、パスを配置するための参照点として用いられます。

注釈　表セル内の注釈は、*PDF_create_annotation( )* の *type* 引数（ただしこの関数を呼び出す必要はありません）に対応する *PDF_add_table_cell( )* の *annotationtype* オプションで作成することができます。*fitannotation* オプションでは、*PDF_create_annotation( )* のすべてのオプションを指定可能です。セル枠が注釈矩形として用いられます。

フォームフィールド　表セル内のフォームフィールドは、*PDF_create_field( )* の *name*・*type* 引数（ただしこの関数を呼び出す必要はありません）に対応する *PDF_add_table_cell( )* の *fieldname*・*fieldtype* オプションで作成することができます。*fitfield* オプションでは、*PDF_create_field( )* のすべてのオプションを指定可能です。セル枠がフィールド矩形として用いられます。

セル内枠内でのセル内容の位置合わせ　デフォルトでは、セル内容の位置は、セル枠に合わせて決まります。*PDF_add_table_cell( )* で *margin* オプションを使えば、セルの端との間に間隔を指定することができます。その結果できる矩形を、**セル内枠**と呼びます。余白が 1 つでも定義されていれば、セル内容はセル内枠に合わせて配置されます（図 8.32 参照）。余白が 1 つも定義されていないときは、セル内枠はセル枠と同じです。

これとあわせて、セルの内容は、内容依存のはめ込みオプションで与えたオプションにも従うことがあります。233 ページ「8.3.4 さまざまな種類の内容を持った表」で説明します。

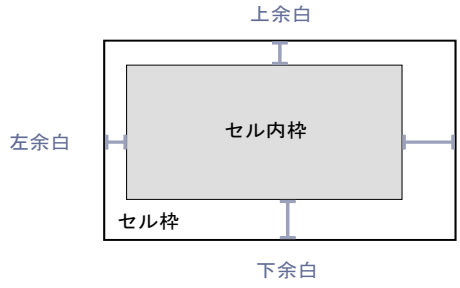


図 8.32
内容をセル内枠にはめ込み

8.3.3 表と列の幅

セルを表に追加する際には、そのセルがまたがる列か表行、または両方の数を、*colspan*・*rowspan* オプションで定義します。デフォルトではセルの列は 1 つ、表行も 1 つです。表の列と表行の総数は、セルを追加するごとに、それぞれの値だけ自動的に加算されます。図 8.33 に、3 列・4 表行の表の例を示します。

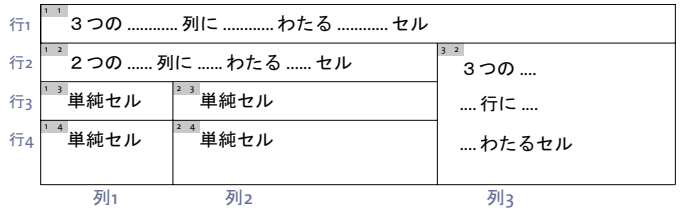


図 8.33
単純セルと、複数の表行や列を連結したセル

さらに、*colwidth* オプションを使って、セルがまたがる最初の列の幅を明示的に与えることもできます。各セルごとに、その最初の列の幅を決めて与えると、それらの幅の値はすべて、表全体の幅に自動的に加算されていきます。図 8.34 に例を示します。

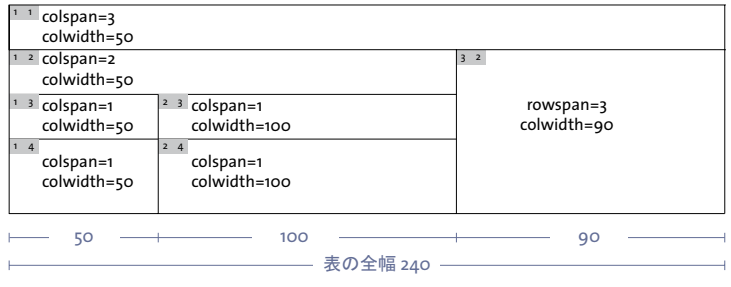


図 8.34
列幅を足し合わせると表全体の幅に

または適当であれば、列幅をパーセント値で指定することもできます。その場合この値は、表のはめ込み枠の幅に対する割合になります。パーセント値による指定を行う場合は、すべての列に対して行う必要があります。でなければ一切してはいけません。

PDF_add_table_cell() の *colscalegroup* オプションを使って、いくつかの列を列伸縮グループとしてまとめてある場合、それらの幅は、グループ内でもっとも幅の広い列と同じになります（図 8.35 参照）。

図 8.35
1 番目の表行の右 4 セルは、同じ列伸縮グループに属しているので、同じ幅になります。

絶対座標が使われている場合（パーセント値でなく）、列幅を定義されていないセルがあるときは、その未決定の幅は以下のようにして算定されます。まず、テキスト行を含む各セルについて、列幅かテキストの幅（回転されているテキストの場合はテキストの高さ）にもとづいて、実際の幅が算出されます。それから、残りの表幅が、まだ決定していない列幅に均等に分配されます。

8.3.4 さまざまな種類の内容を持った表

以下のいくつかの項では、図 8.36 に示すような、さまざまな種類の内容を持った表の例を、一歩ずつ作成していきましょう。

クックブック **完全なコードサンプルがクックブックの** `tables/mixed_table_contents` **トピックにあります。**

前準備として、2 つのフォントを読み込む必要があります。表のはめ込み枠の大きさを、その左下隅と右上隅の座標によって定義し、3 つの表列の幅を指定しましょう。そして、新規ページを A4 サイズで開始しましょう。

```
double llx = 100, lly = 500, urx = 360, ury = 600; // 表の座標

int c1 = 50, c2 = 120, c3 = 90; // 3つの表列の幅

boldfont = p.load_font("Helvetica-Bold", "unicode", "");
normalfont = p.load_font("Helvetica", "unicode", "");

p.begin_page_ext(0, 0, "width=a4.width height=a4.height");
```

手順 1 ：最初のセルを追加　まずは、表の最初のセルから始めましょう。このセルは 1 行目の 1 列目に配置し、3 つの列にわたらせます。1 列目の幅は 50 ポイントです。テキスト行は縦横の中央に置き、すべての端で余白を 4 ポイントとります。下記のコードに、最初のセルを追加する方法を示します。

```
optlist = "fittextline={font=" + boldfont + " fontsize=12 position=center} " +
          "margin=4 colspan=3 colwidth=" + c1;

tbl = p.add_table_cell(tbl, 1, 1, "Our Paper Plane Models", optlist);
if (tbl == -1)
        throw new Exception("エラー : " + p.get_errmsg());
```

手順２：２列にまたがるセルを追加　次の手順として、テキスト行 *1 Giant Wing* を持つセルを追加しましょう。これは 2 行目の 1 列目に配置し、2 つの列にわたらせます。1 列目の幅は 50 ポイントです。行の高さは 14 ポイントです。テキスト行は左上に置き、すべての端で余白を 4 ポイントとります。229 ページ「セルの内容の縦位置を調整する」で述べたのと同様に、*fontsize={capheight=6}* を用いて、テキストの縦揃えを統一しましょう。

この見出し *Giant Wing* のセルは、行全体でなく 3 列中 2 列しか連結しないので、行ベースの塗り分けオプションでは色がつけられません。かわりに範囲枠機能を使って、セルが覆う矩形を灰色の背景色で塗りましょう（範囲枠機能については 242 ページ「8.4 範囲枠」を参照）。下記のコードに、見出し *Giant Wing* のセルを追加する方法を示します。

```
optlist = "fittextline={position={left top} font=" + boldfont +
          " fontsize={capheight=6}} rowheight=14 colwidth=" + c1 +
          " margin=4 colspan=2 matchbox={fillcolor={gray .92}}";

tbl = p.add_table_cell(tbl, 1, 2, "1  Giant Wing", optlist);
if (tbl == -1)
        throw new Exception("エラー：" + p.get_errmsg());
```

図 8.36 さまざまな内容の表セルを一歩ずつ追加

生成される表

Our Paper Plane Models

1 Giant Wing		
Material	Offset print paper 220g/sqm	
Benefit	It is amazingly robust and can even do aerobatics. But it is best suited to gliding.	

生成手順

手順 1：3 列にまたがるセルを追加
手順 2：2 列にまたがるセルを追加
手順 3：さらにテキスト行セル 3 つ追加
手順 4：テキストフローセル追加
手順 5：テキスト行つき画像セル追加
手順 6：表をはめ込む

手順３：さらに３つのテキスト行セルを追加　下記のコードは、*Material*・*Benefit*・*Offset print paper*…のセルを追加します。*Offset print paper*…のセルは 2 列目で始まるので、同時に 120 ポイントの列幅を定義します。セルの内容は左上に置き、すべての端で余白を 4 ポイントとります。

```
optlist = "fittextline={position={left top} font=" + normalfont +
          " fontsize={capheight=6}} rowheight=14 colwidth=" + c1 + " margin=4";

tbl = p.add_table_cell(tbl, 1, 3, "Material", optlist);
if (tbl == -1)
        throw new Exception("エラー：" + p.get_errmsg());

tbl = p.add_table_cell(tbl, 1, 4, "Benefit", optlist);
if (tbl == -1)
        throw new Exception("エラー：" + p.get_errmsg());

optlist = "fittextline={position={left top} font=" + normalfont +
          " fontsize={capheight=6}} rowheight=14 colwidth=" + c2 + " margin=4";

tbl = p.add_table_cell(tbl, 2, 3, "Offset print paper 220g/sqm", optlist);
if (tbl == -1)
        throw new Exception("エラー：" + p.get_errmsg());
```

手順４：テキストフローセルを追加　下記のコードは、*It is amazingly…*のテキストフローセルを追加します。テキストフローの入った表セルを追加するには、まずテキストフローを追加しましょう。上記のテキスト行同様、*fontsize={capheight=6}* を用いれば、文字サイズが結果的にほぼ 8 ポイントになり、また（*margin=4* とあわせて）高さの合計が 14 ポイントとなります。

```
tftext = "It is amazingly robust and can even do aerobatics. " +
          "But it is best suited to gliding.";

optlist = "font=" + normalfont + " fontsize={capheight=6} leading=110%";

tf = p.add_textflow(-1, tftext, optlist);
if (tf == -1)
        throw new Exception("Error: " + p.get_errmsg());
```

取得したテキストフローハンドルは、表セルを追加する時に使います。テキストフローの 1 行目は、テキスト行 *Benefit* のベースラインと揃っているべきです。と同時に、テキスト *Benefit* は、そのセル上端からの間隔がテキスト *Material* と同じになるべきです。テキストフローを追加する際は、上に余白が生じないよう、*fittextflow={firstlinedist=capheight}* を用います。そしてテキスト行と同じく、余白を 4 ポイント加えます。

```
optlist = "textflow=" + tf + " fittextflow={firstlinedist=capheight} " +
          "colwidth=" + c2 + " margin=4";

tbl = p.add_table_cell(tbl, 2, 4, "", optlist);
if (tbl == -1)
        throw new Exception("エラー：" + p.get_errmsg());
```

手順５：テキスト行の入った画像セルを追加　5 番目の手順として、Giant Wing 紙飛行機の画像とテキスト行 *Amazingly robust!* の入ったセルを追加しましょう。このセルは 2 行目の 3 列目で始まり、3 つの行にまたがります。列幅は 90 ポイントです。セルの余白は 4 ポイントに設定します。1 つ目の例としては TIFF 画像をセル内に配置してみましょう。

```
image = p.load_image("auto", "kraxi_logo.tif", "");
if (image == -1)
        throw new Exception("エラー：" + p.get_errmsg());

optlist = "fittextline={font=" + boldfont + " fontsize=8} image=" + image +
          " colwidth=" + c3 + " rowspan=3 margin=4";

tbl = p.add_table_cell(tbl, 3, 2, "Amazingly robust!", optlist);
if (tbl == -1)
        throw new Exception("エラー：" + p.get_errmsg());
```

あるいは、画像は PDF ページとして取り込むこともできます。PDI ページを閉じるのは必ず、*PDF_fit_table( )* を呼び出した後にしてください。

```
int doc = p.open_pdi("kraxi_logo.pdf", "", 0);
if (tbl == -1)
        throw new Exception("エラー：" + p.get_errmsg());

page = p.open_pdi_page(doc, pageno, "");
if (tbl == -1)
        throw new Exception("エラー：" + p.get_errmsg());
```

```
optlist = "fittextline={font=" + boldfont + " fontsize=9} pdipage=" + page +
          " colwidth=" + c3 + " rrowspan=3 margin=4";

tbl = p.add_table_cell(tbl, 3, 2, "Amazingly robust!", optlist);
if (tbl == -1)
        throw new Exception("エラー: " + p.get_errmsg());
```

手順６：表をはめ込む　最後の手順として、表を *PDF_fit_table()* で配置しましょう。*header=1* を用いると、1 行目が表のヘッダになります。*fill* オプションと *area=header・fillcolor={rgb 0.8 0.8 0.87}* サブオプションは、与えた色でヘッダ行を塗るよう指定しています。*stroke* オプションと *line=frame linewidth=0.8* サブオプションを用いて、表の外枠の線幅を 0.8 として定義しましょう。*line=other linewidth=0.3* を用いると、すべてのセルの罫が線幅 0.3 として指定されます。

```
optlist = "header=1 fill={{area=header fillcolor={rgb 0.8 0.8 0.87}}} " +
          "stroke={{line=frame linewidth=0.8} {line=other linewidth=0.3}}";

result = p.fit_table(tbl, llx, lly, urx, ury, optlist);

if (result.equals("_error"))
        throw new Exception("エラー: " + p.get_errmsg());

p.end_page_ext("");
```

8.3.5 表インスタンス

1 つのはめ込み枠に配置された表行群は、表インスタンスを構成します。表全体を表現するには、複数の表インスタンスが必要なこともあります。*PDF_fit_table()* は、1 回呼び出されるごとに、1 つのはめ込み枠に 1 つの表インスタンスを配置します。これらのはめ込み枠は、同じページに多段組レイアウト等で配置しておくことも、または複数のページに配置しておくこともできます。

図 8.37 の表は、3 つのページにわたっています。各ページに 1 つずつあるはめ込み枠に、各表インスタンスが 1 つずつ配置されます。*PDF_fit_table()* を呼び出すたびに、最初の行はヘッダとして定義され、最後の行はフッタとして定義されます。

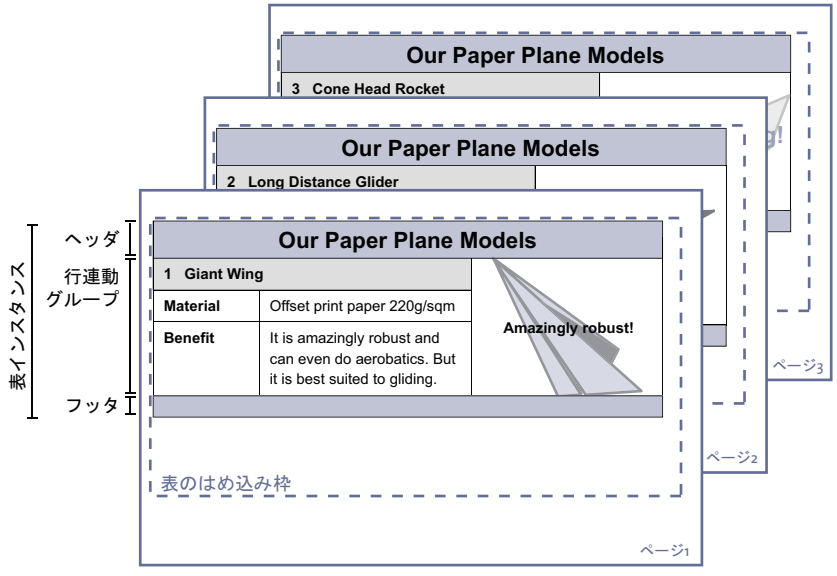


図 8.37
表は複数の表インスタンスに分解され、各はめ込み枠に 1 つずつ配置されます。

下記のコードは、表を配置しきるまで表インスタンスをはめ込みつづけるための、一般的なループを示します。配置するべき表インスタンスがある限り、そのつど新規ページを作成します。

```
do {
        /* 新規ページを作成 */
        p.begin_page_ext(0, 0, "width=a4.width height=a4.height");

        /* 最初の行をヘッダとして使い、すべての表セルに線をひく */
        optlist = "header=1 stroke={{line=other}}";

        /* 表インスタンスを配置 */
        result = p.fit_table(tbl, llx, lly, urx, ury, optlist);
        if (result.equals("_error"))
            throw new Exception("エラー : " + p.get_errmsg());

        p.end_page_ext("");

} while (result.equals("_boxfull"));

/* 結果を調べる。「_stop」ならすべてOKを意味する。*/
if (!result.equals("_stop")) {
        if (result.equals( "_error"))
            throw new Exception("エラー : " + p.get_errmsg());
        else {
            /* これ以外の戻り値はすべて「return」オプションによるユーザー終了。
             * これを扱うには専用のコードが必要。*/
            throw new Exception ("テキストフロー内でユーザーリターンを検出しました");
        }
}
```

```
/* 表内で使ったテキストフローハンドルも削除される */
p.delete_table(tbl, "");
```

ヘッダ・フッタ　*PDF_fit_table()* で *header*・*footer* オプションを使えば、表の最初か最後の行の数を定義して、それが各表インスタンスの上端か下端に配置されるようにすることができます。*fill* オプションで *area=header* か *area=footer* を使うと、ヘッダ・フッタを別の色で塗ることができます。ヘッダ行群は表定義の最初の *n* 行から成り、フッタ行群は最後の *m* 行から成っています。

ヘッダとフッタは、*PDF_fit_table()* で表インスタンスごとに指定します。結果として、表インスタンスごとに異なるものにもなりえます。すなわち、ヘッダ / フッタをつけた表インスタンスと省いた表インスタンスを混在させる、といったことも可能です。それによりたとえば、最後の表インスタンスで特別な行を指定する、といったことも可能になります。

表行の連動　いくつかの表行を必ず同じ表インスタンスに入れさせたいときは、*rowjoingroup* オプションを使って、それらを同じ表行連動グループに割り当てることができます。表行連動グループは、連続する複数の表行を持ちます。このグループの表行はすべて、複数の表インスタンスに別れさせられることがなくなります。

セルで複数表行を連結しても、それらの表行は自動的に連動グループにはなりません。

はめ込み枠が低すぎる　はめ込み枠が低すぎて、必要なヘッダ・フッタ行と、最低 1 つの本体表行ないし行連動グループが入らないときは、表がはめ込み枠に収まるまで、行の高さが一律に縮められます。ただし必要な縮小率が、*vertshrinklimit* で設定した限界よりも小さいときは、縮小は行われずに、*PDF_fit_table()* は文字列 *_error*、またはそれぞれのエラーメッセージを返します。縮小を一切させたくないときは、*vertshrinklimit=100%* を使います。

はめ込み枠が狭すぎる　表のはめ込み枠の座標は、*PDF_fit_table()* を呼び出す際に明示的に与えます。与えた列幅を合計した実際の表幅が、その表のはめ込み枠を超えたときは、表がはめ込み枠に収まるまですべての列が縮められます。ただし必要な縮小率が、*horshrinklimit* で設定した限界よりも小さいときは、縮小は行われずに、*PDF_fit_table()* は文字列 *_error*、またはそれぞれのエラーメッセージを返します。縮小を一切させたくないときは、*horshrinklimit=100%* を使います。

セルの分割　セルが行を連結している場合、後のほうの行がはめ込み枠に収まらないときは、そのセルは分割されます。画像・PDI ページ・テキスト行セルの場合は、セル内容は次の表インスタンスでも繰り返されます。テキストフローセルの場合は、セル内容は後の表行のセルに続きます。

図 8.38 に、テキストフローセルが分割されてテキストフローが次の表行に続いている様子を示します。図 8.39 に、画像セルが次の表インスタンスの最初の行で繰り返される様子を示します。

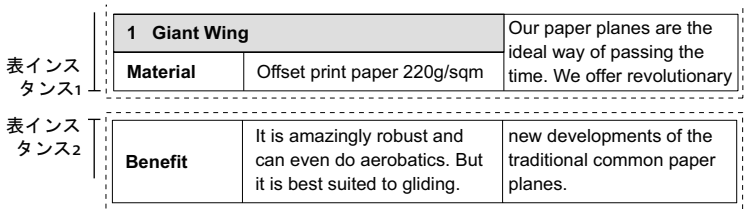


図 8.38
セルの分割

表行の分割　表のはめ込み枠に最後の本体行が収まりきらないとき、それは普通は分割されません。この動作は *PDF_fit_table()* の *minrowheight* オプションで制御され、デフォルト値は 100% です。このデフォルト設定では、表行は分割されず、まるごと次の表インスタンスへ配置されます。

minrowheight 値を減らせば、最後の本体行を分割させて、表行の内容のうち指定した割合を 1 つ目の部分に、残りを次の部分に配置することができます。

図 8.39 に、テキストフロー *It's amazingly robust…* が分割され、次の表インスタンスの最初の本体行へテキストフローが続く様子を示します。複数行にわたる画像セルが分割され、画像は繰り返されます。テキスト行 *Benefit* も繰り返されます。

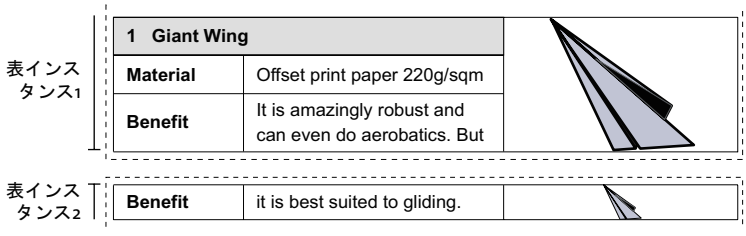


図 8.39
表行の分割

8.3.6 表組版のアルゴリズム

この項では、表組版機能が表を配置する際に行うステップを詳説します。以下、横書きテキストの場合について述べます。しかし、「表行高さ」と「列幅」という言葉を互いに入れ替えれば、縦書きや回転テキストにもあてはまります。

PDF_fit_table() への最初の呼び出しの際には、オプション *colwidth*・*rowheight*・*fittextline*・*fittextflow* がすべてのセルについて吟味され、表全体の幅と高さが、列幅・表行高さ・テキスト内容に基づいて算出されます。

テキスト行を持つ表セルの高さと幅を算出　表組版機能はまず、テキスト行を持つ表セルのうち、*colwidth* または *rowheight* のない表列または表行にわたるものすべてのサイズを決定します。これを実現するために、*fittextline* オプションに従ってテキスト行の、ひいては表セルの幅を算出します。テキストサイズの 2 倍を表セルの高さと見なします（より正確にいうと *boxheight* の 2 倍。これは、別途指定されない限りデフォルト値 *{capheight none}* を持ちます）。縦書きテキストについては、もっとも幅の広いキャラクタの幅がセル幅として用いられます。西向きまたは東向きのテキストについては、テキスト高さの 2 倍がセル幅として用いられます。

この求められた表セルの幅と高さが、そのあと、その表セルがわたる表列または表行のうち、*colwidth* または *rowheight* が指定されていないものすべてに均等に按分されます。

仮の表サイズを算出　次のステップとして組版機能は、表の仮の幅と高さを、それぞれ列幅・表行高さすべての合計として算出します。パーセント値で指定されている列幅と表行高さは、先頭はめ込み枠の幅と高さに基づいて絶対値へ変換されます。*colwidth* または *rowheight* を持たない列または表行がまだある場合には、仮の表サイズが先頭はめ込み枠に等しくなるまで、残りの余白が均等に按分されます。

ですので各表セルに対して少なくとも最小 *rowheight* は指定しておいたほうがよいでしょう。でないと表は自動的にはめ込み枠の高さに合わせて調整されるからです。

小さすぎるセルを拡大　ここで組版機能はすべてのセル内枠を決定します(図8.32参照)。余白の合計がセルの幅または高さより小さいときは、そのセル枠は、そのセルに属するすべての列と表行を均等に拡大することによって適切に拡大されます。

テキスト行の横方向をはめ込む　組版機能は、テキスト行を持つすべてのセルの幅を拡げて、テキスト行が文字サイズを下げなくてもセルにはめ込めるようにします。これが可能でないときは、テキスト行は自動的に *fitmethod=auto* で配置されます。これによって、テキスト行がセル内枠からはみ出さないことが保証されます。*fittextline* オプションで *fitmethod=auto* に設定すると、セル幅が拡がるのを防ぐことができます。

colscalegroup オプションを用いると、同一の列伸縮グループに属するすべての列が必ず等しい幅に伸縮されるように、すなわちこれらの幅が統一されて、グループ内で最も広い幅に合わせられるようにすることができます（図 8.35 参照）。

強制ハイフネーションを避ける　算出された表幅がはめ込み枠より小さいときは、組版機能はテキストフローセルの幅を拡げて、テキストが強制ハイフネーションなしにはめ込めるようにします。これはオプション *checkwordsplitting=false* で回避することもできます。このようなセルの幅は、表幅がはめ込み枠の幅に等しくなるまで拡げられます。

PDF_info_table( ) の *horboxgap* キーを用いて、表幅とはめ込み枠幅の差を取得できます。

テキストの縦方向をはめ込む　組版機能は、すべてのテキスト行・テキストフローセルの高さを拡げて、テキスト行またはテキストフローが文字サイズを下げずにセル内枠にはめ込めるよう試みます。ただし、テキスト行またはテキストフローに対してサブオプション *fitmethod=auto* が設定されている場合、またはテキストフローが *continuetextflow* オプションで他のセルに続いている場合には、セル高さは拡がりません。

このセル高さを拡げる処理は、テキスト行またはテキストフローを内容として持つセルに対してのみ適用され、それ以外の種類のセル内容、すなわち画像・PDI ページ・パスオブジェクト・注釈・フィールドに対しては適用されません。

rowscalegroup オプションを用いると、同一の表行伸縮グループに属するすべての表行が必ず等しい高さに伸縮されるようにすることができます。

表を次のはめ込み枠に続ける　算出された表全体の高さがはめ込み枠よりも大きい（すなわち、すべての表セルをはめ込み枠に収めることができない）ときは、組版機能は、そのはめ込み枠に収まらない初めての表行に出会う前に、そのはめ込み枠内に表行を配置することを止めます。

1 つのセルが複数行にわたり、そのすべての行がはめ込み枠に収まらないときは、このセルは分割されます。セルが画像・PDI ページ・パスオブジェクト・注釈・フォームフィールドを内容として持つときは、*repeatcontent=false* が指定されていない限り、そのセル内

容は次のはめ込み枠内で繰り返されます。しかしテキストフローは、セルがわたる後続の表行に続きます（図 8.38 参照）。

rowjoingroup オプションを用いると、1 つの表行連動グループに属するすべての表行が必ず 1 つのはめ込み枠内に現れるようにすることができます。ヘッダまたはフッタに属するすべての表行と 1 つの本体行は、自動的に表行連動グループを形成します。ですので組版機能は、はめ込み枠に収まらない初めての行に出会う前に表行を配置することを止めます（図 8.37 参照）。

return オプションを用いると、対象行を配置した後にその表インスタンス内に絶対もう表行が配置されないようにすることができます。

表行を分割　表行は、非常に高いとき、または 1 個の本体行しかないときには、分割されることがあります。末尾の本体行が表のはめ込み枠に完全に収まらないときは、それはまるごと次のはめ込み枠へ移動されます。この動作は *PDF_fit_table()* の *minrowheight* オプションで制御することができます。このオプションのデフォルト値は 100% です。この *minrowheight* 値を小さくすると、末尾本体行の内容のうち指定した割合がカレントはめ込み枠に配置され、その行の残りは次のはめ込み枠に配置されます（図 8.39 参照）。

PDF_info_table() の *rowsplit* キーを用いると、表行が分割されているかどうかを調べることができます。

算出された表幅を調節　算出される表幅は、テキスト行の横方向をはめ込んだ後など、決定ステップのいずれかの後に、はめ込み枠よりも大きくなることがあります。この場合には、表幅がはめ込み枠の幅に等しくなるまで、すべての列幅が均等に縮められます。この縮小処理は *horshrinklimit* オプションによって制限されます。

PDF_info_table() の *horshrinking* キーを用いると、横縮小倍率を取得することができます。

horshrinklimit の閾値を超えると、下記のエラーメッセージが現れます：

```
Calculated table width $1 is too large (> $2, shrinking $3)")
```

ここで $1 は算出された表幅を示し、$2 は可能な最大の幅、$3 は *horshrinklimit* 値です。

表のサイズを小さいはめ込み枠に合わせて調整　直前のはめ込み枠に対して算出された表幅が、カレントはめ込み枠に対して大きすぎるときは、組版機能は、表幅がカレントはめ込み枠の幅に等しくなるまですべての列を縮めます。ただしセル内容は調整されません。表幅を改めて計算しなおすには、*PDF_fit_table()* を *rewind=1* をつけて呼び出します。

8.4 範囲枠

範囲枠を使うと、PDFlib がページ上に何らかの内容を配置したときに算出した座標を利用することができます。範囲枠を定義するには、そのための専用の関数があるわけではなく、実内容を配置する *PDF_fit_textline( )* や *PDF_fit_image( )* などの関数で *matchbox* オプションを指定します。範囲枠はさまざまな目的に使えます。

- 範囲枠は装飾できます。例：色を塗る、枠線で囲む。
- 範囲枠を使って、注釈（複数可）を *PDF_create_annotation( )* で自動的に作成できます。
- 範囲枠はテキスト行の高さを定義して、それが *PDF_fit_textline( )* で枠にはめ込まれるようにしたり、テキストフロー内のテキスト範囲の高さを定義して、それが装飾されるようにしたりします（*boxheight* オプション）。
- 範囲枠は画像の切り抜きを定義します。
- 範囲枠の座標やその他のプロパティは、*PDF_info_matchbox( )* で取得して、他の作業に利用することができます。例：画像を挿入。

PDFlib はそれぞれの要素について、ページ上におけるその要素の位置（関連するすべてのオプションで指定された）を記述する外接枠に対応する矩形として、範囲枠を算出します。テキストフローと表セルの場合は、改行や表行分割によって、1 つの範囲枠が複数の矩形から成ることもあります。

範囲枠の矩形（群）は、配置する要素を描く前に描かれます。そのため、範囲枠の罫や塗りの効力は実内容では打ち消されることがありますが、その逆はありません。特に、範囲枠の中で、画像で覆われた領域と重なる部分は、画像に隠れます。画像が *fitmethod=slice* または *fitmethod=clip* で配置されているときは、画像のはめ込み枠の外の範囲枠罫も切り落とされます。この現象を避けるには、範囲枠の矩形を、*PDF_fit_image( )* を呼び出した後に、*PDF_rect( )* 等の基本的な描画関数を使って描くという方法もあります。範囲枠の矩形の座標は、*PDF_info_matchbox( )* を使って取得できますが、ただしこれはその範囲枠が *PDF_fit_image( )* への呼び出しで名前を与えられていた場合に限ります。

以下のいくつかの項で、範囲枠の利用例をいくつか示します。範囲枠のオプションリストに対応している関数については、くわしくは ***PDFlib API リファレンス***を参照してください。

8.4.1 テキスト行を装飾

テキスト行の範囲枠から話を始めましょう。*PDF_fit_textline( )* においては範囲枠は、与えられたテキストのテキスト枠と同じになります。デフォルトでは、テキスト枠の幅はテキストの幅に等しく、高さは、与えられた文字サイズのキャップハイトに等しくなります。範囲枠の大きさを図示するために、下記のコードでは範囲枠を青の背景色で塗っています（図 8.40a 参照）。

```
String  optlist =
        "font=" + normalfont + " fontsize=8 position={left top} " +
        "matchbox={fillcolor={rgb 0.8 0.8 0.87} boxheight={capheight none}}";

p.fit_textline("Giant Wing Paper Plane", 2, 20, optlist);
```

boxheight オプションは、*boxheight={capheight none}* がデフォルト設定なので、省略してもかまいません。*boxheight* オプションで枠の高さをもっと増やしてディセンダまで覆うようにすれば、より美しくなります（図 8.40b 参照）。

枠の高さを増やして文字サイズに一致させたいときは、*boxheight={fontsize descender}* が使えます（図 8.40c 参照）。

次のステップとしては、範囲枠の左・右・下へ変位を加えて拡げ、枠のすべての端をテキストと均等な間隔にしてみましょう。さらに枠線幅を指定して、範囲枠のまわりに矩形を描きましょう（図 8.40d 参照）。

クックブック 完全なコードサンプルがクックブックの text_output/text_on_color トピックにあります。

図 8.40 さまざまなサブオプションの範囲枠を使ってテキスト行を装飾

生成される出力	PDF_fit_textline( ) の matchbox オプションのサブオプション
a) Giant Wing Paper Plane	boxheight={capheight none}
b) Giant Wing Paper Plane	boxheight={ascender descender}
c) Giant Wing Paper Plane	boxheight={fontsize descender}
d) Giant Wing Paper Plane	boxheight={fontsize descender} borderwidth=0.3 offsetleft=-2 offsetright=2 offsetbottom=-2

8.4.2 テキストフローで範囲枠を利用

テキストフローの一部分を装飾　この項では、テキストフロー内の一部のテキストを装飾しましょう。*very dangerous* という言葉をマーカーペンのように目立たせましょう。これを実現するには、言葉を *matchbox* インラインオプションと *matchbox=end* インラインオプションで挟みます（図 8.41 参照）。

図 8.41 matchbox インラインオプションを含んだテキストフロー

生成される出力	PDF_create_textflow( ) に対するテキストとインラインオプション
It is very dangerous to fly the Giant Wing in a thunderstorm.	It is <matchbox={fillcolor={rgb 1 0 0} boxheight={ascender descender}}>very dangerous <matchbox=end> to fly the Giant Wing in a thunderstorm.

テキストフローの範囲枠に Web リンクを追加　今度は、テキストフローの一部分に Web リンクを追加しましょう。1 番目の手順として、リンクをつけたい部分のテキストを示す *kraxi* という範囲枠を含んだテキストフローを作成しましょう。2 番目に、URL を開くアクションを作成しましょう。3 番目に、枠が不可視の *Link* 型の注釈を作成しましょう。そのオプションリストの中で、範囲枠 *kraxi* を参照してリンクの矩形として使いましょう（*PDF_create_annotation( )* の矩形の座標は無視されます）。

クックブック 完全なコードサンプルがクックブックの text_output/weblink_in_textflow トピックにあります。

```
/* 範囲枠「kraxi」を含んだテキストフローを作成してはめ込み */
String  tftext =
        "For more information about the Giant Wing Paper Plane see the Web site of " +
        "<underline=true matchbox={name=kraxi boxheight={fontsize descender}}>" +
        "Kraxi Systems, Inc.<matchbox=end underline=false>";
```

```
String  optlist = "font=" + normalfont + " fontsize=8 leading=110%";
tflow = p.create_textflow(tftext, optlist);
if (tflow == -1)
        throw new Exception("エラー : " + p.get_errmsg());

result = p.fit_textflow(tflow, 0, 0, 50, 70, "fitmethod=auto");
if (!result.equals("_stop"))
        { /* ... */ }

/* URIアクションを作成 */
optlist = "url={http://www.kraxi.com}";
act = p.create_action("URI", optlist);

/* 範囲枠「kraxi」上にLink注釈を作成 */
optlist = "action={activate " + act + "} linewidth=0 usematchbox={kraxi}";
p.create_annotation(0, 0, 0, 0, "Link", optlist);
```

テキストが複数の行にわたる場合でも、1 回 *PDF_create_annotation()* を呼び出すだけで、適切な数のリンク注釈が自動的に作成されます。結果を図 8.42 に示します。

図 8.42
テキストフローの一部分に Web リンクを追加

8.4.3 範囲枠と画像

画像にリンクを追加　画像で覆われた領域に Web リンクを追加するには、画像の範囲枠が使えます。コードは先述の 243 ページ「テキストフローの範囲枠に Web リンクを追加」と同じです。ただし、テキストフローを配置するのでなく、画像を次のオプションリストを使ってはめ込みます。

```
String  optlist = "boxsize={130 130} fitmethod=meet matchbox={name=kraxi}";
p.fit_image(image, 10, 10, optlist);
```

クックブック　完全なコードサンプルがクックブックの interactive/link_annotations トピックにあります。

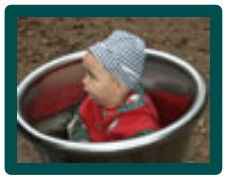
画像にふちをつける　この例では、画像の範囲枠を使って、画像のまわりにふちをつけましょう。画像に *fitmethod=meet* を使って、縦横比を保ちつつまるごと、与えられた枠に収めましょう。*borderwidth* サブオプションを使った *matchbox* オプションを使って、画像のまわりに太い枠を描きましょう。*strokecolor* サブオプションで枠の色を決め、*linecap*・*linejoin* サブオプションを使って角を丸めます。

クックブック　完全なコードサンプルがクックブックの images/frame_around_image トピックにあります。

範囲枠はつねに画像より前の時点で描かれるので、一部が画像で隠されてしまいます。これを避けるために、枠の幅の 50 パーセントの *offset* サブオプション群を使って、画像が覆う領域よりもふちを大きくしましょう。あるいは、枠をその分太くするという方法もあ

ります。図 8.43 に、ふちをつけるために *PDF_fit_image( )* で使うオプションリストを示します。

図 8.43 画像の範囲枠を使って画像にふちをつける

生成される出力	PDF_fit_image( ) に対するオプションリスト
	boxsize={60 60} position={center} fitmethod=meet matchbox={name=kraxi borderwidth=4 offsetleft=-2 offsetright=2 offsetbottom=-2 offsettop=2 linecap=round linejoin=round strokecolor {rgb 0.0 0.3 0.3}}

テキストを画像に整列させる 以下のコードは、縦方向のテキストを、画像の右余白に合わせる方法を示しています。画像ははめ込み方式 *meet* を用いて、与えられた枠に縦横比を保ってはめ込まれています。はめ込み枠の具体的な座標は *PDF_info_matchbox( )* で取得され、縦方向のテキストははめ込み枠の右下隅 *(x2, y2)* に合わせて配置されています。範囲枠の辺は描線されています（図 8.44 参照）。

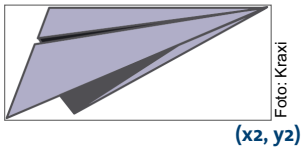
クックブック 完全なコードサンプルがクックブックの images/align_text_at_image トピックにあります。

```
/* このオプションリストを使って画像を読み込んではめ込む */
String  optlist = "boxsize={300 200} position={center} fitmethod=meet " +
        "matchbox={name=giantwing borderwidth=3 strokecolor={rgb 0.85 0.83 0.85}}";

/* 画像を読み込んではめ込む */

/* 範囲枠の右下（第二）隅の座標を取得 */
if ((int) p.info_matchbox("giantwing", 1, "exists") == 1)
{
        x1 = p.info_matchbox("giantwing", 1, "x2");
        y1 = p.info_matchbox("giantwing", 1, "y2");
}
/* その隅から2だけ間隔をあけてテキスト行を開始 */
p.fit_textline("Foto: Kraxi", x2+2, y2+2, "font=" + font + " fontsize=8 orientate=west");
```

図 8.44 画像の範囲枠の座標を使ってテキスト行をはめ込む

生成される出力	生成手順
	手順 1：画像を範囲枠とともにはめ込み 手順 2：範囲枠情報を得て座標 (x2, y2) を取得 手順 3：取得した座標 (x2, y2) から orientate=west オプションでテキスト行を開始

9 pCOS インタフェース

pCOS（*PDFlib Comprehensive Object Syntax*）インタフェースは、PDF 文書のページ内容記述以外のすべてのセクション、すなわちページサイズ・メタデータ・インタラクティブ要素などから任意の情報を取得できる簡単でエレガントな機能を提供します。

pCOS インタフェースの利用例と、pCOS パス文法の説明は、別文書として入手可能な pCOS パスリファレンスにあります。さらなる作成例が下記の pCOS クックブックにあります：

www.pdflib.com/pcos-cookbook/

10 PDF のバージョンと規格

10.1 Acrobat・PDF のバージョン

ユーザー側での選択に従い、PDFlib は下記の PDF バージョンに従った出力を生成します:

- PDF 1.3（Acrobat 4）
- PDF 1.4（Acrobat 5）
- PDF 1.5（Acrobat 6）
- PDF 1.6（Acrobat 7）
- PDF 1.7（Acrobat 8）。技術的には ISO 32 000-1 と同等
- PDF 1.7 Adobe 拡張レベル 3（Acrobat 9）
- PDF 1.7 Adobe 拡張レベル 8（Acrobat X）

PDF 出力のバージョンは、*PDF_begin_document( )* の *compatibility* オプションで制御することができます。それぞれの PDF 互換モードにおいては、それよりも高いレベルのための PDFlib 機能は利用できません（表 10.1 参照）。そのような機能を利用しようとすると例外が発生します。

PDI で取り込む文書の PDF バージョン　どの互換モードにおいても、PDI で取り込めるのはそれ以下の PDF バージョンの PDF 文書だけです。それより新しい PDF バージョンの PDF を取り込む必要がある場合は、それに合った *compatibility* オプションを設定する必要があります（186 ページ「7.2.3 受け入れ可能な PDF 文書」参照）。ただしこの上位 PDF バージョン取り込み不可ルールの例外として、PDF 1.7 拡張レベル 3（Acrobat 9）・PDF 1.7 拡張レベル 8（Acrobat X）に従った文書は PDF 1.7 文書へも取り込むことが可能です。

文書の PDF バージョンを変更　ある特定の PDF バージョンに従って出力を生成する必要があるにもかかわらず、それよりも高い PDF バージョンを用いた PDF を取り込む必要がある場合には、その文書を PDI で取り込む前にまず、出力したい PDF バージョンに下げる変換を行う必要があります。Acrobat 7/8/9 Professional でメニュー項目**「アドバンスト」→「*PDF* の最適化」→「互換性」**を、あるいは Acrobat X で**「ファイル」→「名前を付けて保存 ...」→「最適化された *PDF*...」**を用いれば、PDF バージョンを下記のように変えることができます:

- Acrobat 7：PDF 1.3 ～ PDF 1.6
- Acrobat 8：PDF 1.3 ～ PDF 1.7
- Acrobat 9：PDF 1.3 ～ PDF 1.7 拡張レベル 3
- Acrobat X：PDF 1.3 ～ PDF 1.7 拡張レベル 8

表 10.1　特定の PDF 互換モードを要する PDFlib 機能一覧

機能	PDFlib API 関数・オプション
PDF 1.7 拡張レベル 8（Acrobat X）を要する機能	
PDF/X-4:2010 でのレイヤーの直接使用（レイヤーバリアントなし）	PDF_set_layer_dependency( )：**オプション** createorderlist

表 10.1 特定の PDF 互換モードを要する PDFlib 機能一覧

機能	PDFlib API 関数・オプション
PDF 1.7 拡張レベル 3（Acrobat 9）を要する機能	
地理空間 PDF	PDF_begin_document( ) : オプション viewports PDF_load_image( ) : オプション georeference
フォルダのある PDF ポートフォリオ	PDF_add_portfolio_folder( )
256 ビットキーによる AES 暗号化	PDF_begin_document( ) : 256 ビットによる AES 暗号化は、compatibility=1.7ext3 の場合、masterpassword・userpassword・attachmentpassword・permissions オプションのいずれかが与えられているときは自動的に用いられます。
レイヤーバリアント	PDF_set_layer_dependency( ) : 依存種別 Variant （この機能は PDF 1.7 ext 3 を要しませんが、Acrobat 9 でしか動作しません）
参照 PDF	PDF_open_pdi_page( )・PDF_begin_template_ext( ) の reference オプション （この機能は PDF 1.7 ext 3 を要しませんが、Acrobat 9 でしか動作しません）
PRC 形式の 3D モデルの埋め込み	PDF_load_3ddata( ) : オプション type=PRC
バーコードフィールド	PDF_create_field( )・PDF_create_fieldgroup( ) : オプション barcode
PDF 1.7（Acrobat 8）を要する機能	
PDF ポートフォリオ	PDF_begin_document( ) : オプション portfolio PDF_add_portfolio_file( )
添付に Unicode ファイル名	PDF_begin/end_document( ) : オプション attachments、サブオプション filename
PDF 1.6（Acrobat 7）を要する機能	
ユーザー単位	PDF_begin/end_document( ) : オプション userunit
印刷の拡縮	PDF_begin/end_document( ) : viewerpreferences オプションに対するサブオプション printscaling
文書を開くモード	PDF_begin/end_document( ) : オプション openmode=attachments
128 ビットキーによる AES 暗号化	PDF_begin_document( ) : AES 暗号化は、compatibility=1.6 または 1.7 の場合、masterpassword・userpassword・attachmentpassword・permissions オプションのいずれかが与えられているときは自動的に用いられます。
ファイル添付のみを暗号化	PDF_begin/end_document( ) : オプション attachmentpassword
添付の説明	PDF_begin/end_document( ) : オプション attachments に対するサブオプション description
U3D 形式の 3D モデルの埋め込み	PDF_load_3ddata( )・PDF_create_3dview( )。PDF_create_annotation( ) : type=3D
PDF 1.5（Acrobat 6）を要する機能	
さまざまなフィールドオプション	PDF_create_field( )・PDF_create_fieldgroup( )
ページレイアウト	PDF_begin/end_document( ) : オプション pagelayout=twopageleft/right
さまざまな注釈オプション	PDF_create_annotation( )
拡張権限設定	PDF_begin_document( ) で permissions=plainmetadata、表 3.3 参照

表 10.1 特定の PDF 互換モードを要する PDFlib 機能一覧

機能	PDFlib API 関数・オプション
日中韓フォントに対するさまざまな CMap	PDF_load_font( )、表 4.3 参照
タグ付き PDF	PDF_begin_item( ) に対するさまざまなオプション。 PDF_begin/end_page_ext( ) : オプション taborder
レイヤー	PDF_define_layer( )・PDF_begin_layer( )・PDF_end_layer( )・PDF_layer_dependency( )
JPEG 2000 画像	PDF_load_image( ) で imagetype=jpeg2000
圧縮オブジェクトストリーム	圧縮オブジェクトストリームは、compatibility=1.5 以上の場合、PDF_begin_document( ) で objectstreams=none が設定されていなければ自動的に生成されます。
PDF 1.4（Acrobat 5）を要する機能	
スムーズシェーディング（カラーブレンド）	PDF_shading_pattern( )・PDF_shfill( )・PDF_shading( )
ソフトマスク	1 ビット以上のピクセル深度を持つ画像に対して masked オプションをつけて PDF_load_image( )
JBIG2 画像	PDF_load_image( ) で imagetype=jbig2
128 ビット暗号化	userpassword・masterpassword・permissions オプションをつけて PDF_begin_document( )
拡張権限設定	permissions オプションをつけて PDF_begin_document( )、表 3.3 参照
日中韓フォントに対するさまざまな CMap	PDF_load_font( )、表 4.3 参照
透過などのグラフィック状態オプション	オプション alphaisshape・blendmode・opacityfill・opacitystroke・textknockout をつけて PDF_create_gstate( )
アクションに対するさまざまなオプション	PDF_create_action( )
注釈に対するさまざまなオプション	PDF_create_annotation( )
さまざまなフィールドオプション	PDF_create_field( )・PDF_create_fieldgroup( )
タグ付き PDF	PDF_begin_document( ) で tagged オプション
参照 PDF	PDF_open_pdi_page( )・PDF_begin_template_ext( ) で reference オプション（ただし、この機能は正しく表示 / 印刷されるには Acrobat 9 を要します）

10.2 ISO 32000

PDF 1.7 は、ISO 32000-1 として標準化されています。この国際標準の技術的内容は、Acrobat 8 にファイル形式である Adobe の PDF 1.7 と等価です。PDFlib は Adobe の PDF リファレンスに忠実に準拠していますので、ひいては ISO 32000-1 にも忠実に準拠しています。ただし、現在のところ ISO 32000-1 準拠をチェックする検証ソフトウェアは得られません。

執筆時点で、この ISO 規格の次のバージョンが、ISO 32000-2 の名のもとに用意されつつあります。この規格は、GeoPDF、ヒエラルキー構造のポートフォリオ、AES-25 暗号化といった Acrobat 9 の機能（詳しい一覧は表 10.1 を参照）を取り込んだもので、PDFlib は *compatibility=pdf1.7ext3* 設定でこれに現時点で対応しています。

10.3 PDF/X による印刷出力

10.3.1 PDF/X 規格ファミリ

PDF/X 形式群は、ISO 15930 規格ファミリで記述され、商業印刷に適したデータの受け渡しに利用できる一貫した堅牢な PDF のサブセットを提供するために努力しています。PDFlib は、以下に説明する種類の PDF/X に準拠した出力を生成し入力を処理することができます。

PDF/X-1a:2001（ISO 15930-1 で定義） この「ブラインド交換」（事前の技術的すり合わせを一切必要としない印刷データのやり取り）用規格は PDF 1.3 に基づいており、CMYK・スポットカラーデータに対応しています。RGB・デバイス独立（ICC ベース・Lab）色は明示的に禁止されています。PDF/X-1a:2001 は、出版広告のやり取りやその他の応用に広く利用されています（特に北米で）。

PDF/X-1a:2003（ISO 15930-4 で定義） この規格は PDF/X-1a:2001 の後継です。PDF 1.4 に基づいており、いくつかの機能（透過など）が禁止されています。PDF/X-1a:2003 は PDF/X-3:2003 の厳密なサブセットであり、CMYK・スポットカラー・CMYK 出力デバイスに対応しています。

注 PANTONE® カラーは PDF/X-1a モードでは使えません。

PDF/X-3:2002（ISO 15930-3 で定義） この規格は PDF 1.3 に基づいており、グレースケール・CMYK・スポットカラーだけでなくデバイス独立色に基づく現在のワークフローに対応しています。特にヨーロッパの国々で広く利用されています。出力デバイスとしては単色・RGB・CMYK のいずれかを使うことができます。

PDF/X-3:2003（ISO 15930-6 で定義） この規格は PDF/X-3:2002 の後継です。PDF 1.4 に基づいており、いくつかの機能（透過など）が禁止されています。

PDF/X-4（ISO 15930-7 で定義） この規格は、PDF/X-1a と PDF/X-3 の後継ととらえることができます。PDF 1.6 に基づいており、下記の種類から成ります：

- PDF/X-4 では、透過とレイヤーは許されますが（いくつかの制限のもとに）、それ以外のいくつかの PDF 1.6 の機能は依然禁止されています。
- PDF/X-4p では、出力インテント ICC プロファイルを、容量を抑えるために PDF 文書の外に置くことが許されます。

PDFlib は、PDF/X-4 規格の 15930-7:2010 バージョンを実装しています。この 2010 バージョンでは、レイヤーの取り扱いに関していくつかの変更が加えられています。この 2010 の機能は、*PDF_set_layer_dependency()* の *createorderlist* オプションで指定することができます。このオプションは PDF/X-4:2008 に対しては用いるべきではありません。

PDF/X-5（ISO 15930-8 で定義） この規格は「部分的交換」のためのものです。部分的交換を行うには、ファイルの作り手と受け手の間で事前の協議が必要です。PDF/X-4 と PDF/X-4p の拡張ととらえることができ（すなわち PDF 1.6 に基づいており）、下記の種類から成ります：

- PDF/X-5g では、グラフィック内容を PDF 文書の外に許しています。これは、文書の送り手と受け手との間で何らかのコミュニケーションが必要です。

- PDF/X-5pgでは、外部グラフィック内容と外部出力インテントICCプロファイルを許します。
- PDF/X-5n では、n-顔料の印刷特性に対する外部出力インテント ICC プロファイルに対応しています。この種類は PDFlib では対応していません。

PDF/X-5 特有の機能を何も要しない場合は、文書は PDF/X-4 か PDF/X-4p に従って作成するべきです。なぜならこちらのほうが一般的な規格だからです。

ISO 15930-8:2008 規格は、外部参照グラフィックに対する XMP 指定項目に関して、いくつか誤りを含んでいます。この規格の修正バージョン ISO 15930-8:2010 がこの 2008 バージョンを置き換えます。PDFlib は、PDF/X-5 のこの 2010 バージョンを実装しています。

注 PDF/X-5g の検証は、Acrobat 9 のプリフライトでは、外部ページを参照している場合には失敗します。この問題は Acrobat X では修正されています。

10.3.2 PDF/X 準拠出力の生成

クックブック PDF/X を生成するためのコードサンプルが PDFlib クックブックの pdfx カテゴリにあります。

PDF/X 準拠出力を PDFlib で作成するには次のような方法を用います。

- PDFlib では、PDF/X のいくつかの形式上の設定は自動的に行われます。PDF バージョン番号や PDF/X 準拠キーの設定などです。
- PDFlib クライアントでは、ある種の関数呼び出しやオプションは明示的に行う必要があります。これにあてはまるものを表 10.2 に詳述します。
- PDFlib クライアントでは、ある種の関数呼び出しやオプションは用いないようにする必要があります。これにあてはまるものを表 10.3 に詳述します。
- 既存の PDF/X 準拠文書からページを取り込む際には、上記以外にも適用される規則があります（258 ページ「10.3.4 PDI による PDF/X 文書の取り込み」参照）。

必要な操作 表 10.2 に、PDF/X 準拠出力の生成に必要な操作をすべて挙げます。どの項目も、特記なき限り、すべての PDF/X 準拠レベルにあてはまります。PDF/X モードの時に、必要な関数のうちのいずれか 1 つでも呼ばなかった場合には、例外が発生します。

表 10.2 PDF/X 互換のために必要な操作一覧

項目	PDF/X 互換のために必要な PDFlib 関数・オプション
準拠レベル	PDF_begin_document( ) の pdfx オプションを、望みの PDF/X 準拠レベルに設定する必要があります。
出力条件（出力インテント）	PDF_begin_document( ) の直後に、PDF_load_iccprofile( ) で usage=outputintent を指定するか、または PDF_process_pdi( ) で action=copyoutputintent を指定して（ただし両方式の併用は不可）呼び出す必要があります。HKS スポットカラー・Pantone スポットカラー・ICC ベースのカラー・Lab カラーのうちのいずれかが用いられている場合は、出力デバイスの ICC プロファイルを埋め込む必要があります。この場合、標準出力条件は許されません。 PDF/X-1a の場合：出力デバイスは、単色か CMYK のデバイスである必要があります。 PDF/X-3/4/5 の場合：出力デバイスは、単色・RGB・CMYK いずれかのデバイスである必要があります。
フォント埋め込み	PDF_load_font( ) の embedding オプションを true に設定して（このオプションを受け取る他の関数でも同様）フォント埋め込みを可能にします。PDF コアフォントについても埋め込みが必要なことに注意してください。

表 10.2 PDF/X 互換のために必要な操作一覧

項目	PDF/X 互換のために必要な PDFlib 関数・オプション
ページ枠	ページボックスは、cropbox・bleedbox・trimbox・artbox オプションで設定可能ですが、以下のすべての必要条件を満たす必要があります。 ► TrimBox か ArtBox を設定しなければなりません。ただし、両方のボックス項目を設定してはなりません。TrimBox も ArtBox もない場合は PDFlib は CropBox を（もしあれば）TrimBox として採用し、CropBox もなければ MediaBox を採用します。 ► BleedBox が存在する場合、それは ArtBox と TrimBox の中に収まっていなければなりません。 ► CropBox が存在する場合、それは ArtBox と TrimBox の中に収まっていなければなりません。
グレースケールカラー	PDF/X-3/4/5：グレースケール画像と、PDF_setcolor( ) でのグレー色空間指定は、出力条件がグレースケールまたは CMYK デバイスの場合か、あるいは PDF_begin_page_ext( ) で defaultgray オプションを設定していた場合にのみ用いることができます。
***RGB*カラー**	PDF/X-3/4/5：RGB 画像と、PDF_setcolor( ) での RGB 色空間指定は、出力条件が RGB デバイスの場合か、あるいは PDF_begin_page_ext( ) で defaultrgb オプションを設定していた場合にのみ用いることができます。
***CMYK*カラー**	PDF/X-3/4/5：CMYK 画像と、PDF_setcolor( ) での CMYK 色空間指定は、出力条件が CMYK デバイスの場合か、あるいは PDF_begin_page_ext( ) で defaultcmyk オプションを設定していた場合にのみ用いることができます。
文書情報キー	Creator 情報キーと Title 情報キーを、PDF_set_info( ) で、または（PDF/X-4・PDF/X-5 で）PDF_begin/end_document( ) の metadata オプションで CreatorTool・dc:title XMP プロパティで、空でない値に設定する必要があります。

禁じられた操作　表 10.3 に、PDF/X 準拠出力の生成時には禁じられている操作をすべて挙げます。どの項目も、特記なき限り、すべての PDF/X 準拠レベルにあてはまります。PDF/X モードの時に、禁じられた関数のうちのいずれか 1 つでも呼んだ場合には、例外が発生します。同様に、取り込まれた PDF ページがカレントの PDF/X 準拠レベルに合わない場合、その PDI 呼び出しは失敗します。

表 10.3 PDF/X 互換のために禁止または制約される操作一覧

項目	PDF/X 互換のために禁止または制約される PDFlib 関数・オプション
グレースケールカラー	PDF/X-1a：PDF_begin_page_ext( ) の defaultgray オプションは避ける必要があります。
***RGB*カラー**	PDF/X-1a：RGB 画像と PDF_begin_page_ext( ) の defaultrgb オプションは避ける必要があります。
***CMYK*カラー**	PDF/X-1a：PDF_begin_page_ext( ) の defaultcmyk オプションは避ける必要があります。
***ICC*ベースカラー**	PDF/X-1a：PDF_setcolor( ) での iccbasedgray/rgb/cmyk 色空間、および、setcolor:iccprofilegray/rgb/cmyk パラメタは避ける必要があります。
***Lab*カラー**	PDF/X-1a：PDF_setcolor( ) での Lab 色空間は避ける必要があります。
注釈・フォームフィールド	BleedBox（BleedBox が存在しない場合は TrimBox か ArtBox ）内での注釈は避ける必要があります：PDF_create_annotation( )・PDF_create_field( )。
アクション・*JavaScript*	JavaScript を含むすべてのアクションは避ける必要があります：PDF_create_action( )。

表 10.3 PDF/X 互換のために禁止または制約される操作一覧

項目	PDF/X 互換のために禁止または制約される PDFlib 関数・オプション
画像	PDF/X-1a：RGB・ICC-based・YCbCr・Lab カラーの画像は避ける必要があります。着色された画像で用いられるスポットカラーの代替色もこれと同じ条件を満たす必要があります。 PDF/X-1・PDF/X-3：JBIG2 画像は避ける必要があります。 PDF_load_image( ) の OPI-1.3・OPI-2.0 オプションは避ける必要があります。
透過画像・グラフィック	PDF/X-1・PDF/X-3：画像に対してソフトマスクは避ける必要があります。PDF_load_image( ) の masked オプションは、そのマスクが 1 ビット画像を参照する場合を除き、避ける必要があります。暗黙的透過（アルファチャンネル）を持つ画像は許されませんので、それらは PDF_load_image( ) の ignoremask オプションで読み込む必要があります。PDF_create_gstate( ) の opacityfill・opacitystroke オプションは、それらが値 1 を持つ場合を除き、避ける必要があります。 透過画像・グラフィックは PDF/X-4・PDF/X-5 では許されています。
透過グループ	PDF_begin/end_page_ext( )・PDF_begin_template_ext( )・PDF_open_pdi_page( ) の transparencygroup オプションは、PDF/X-1・PDF/X-3 では許されず、PDF/X-4・PDF/X-5 でのみ許されます。 transparencygroup を用いる場合、colorspace サブオプションの値は下記の要件に従う必要があります： ► DeviceGray：PDF/X 出力条件はグレースケールまたは CMYK デバイスである必要があります。生成ページについては（テンプレートと取り込みページでは不可）、かわりに PDF_begin_page_ext( ) で defaultgray オプションを設定することもできます。 ► DeviceRGB：PDF/X 出力条件は RGB デバイスである必要があります。生成ページについては（テンプレートと取り込みページでは不可）、かわりに PDF_begin_page_ext( ) で defaultrgb オプションを設定することもできます。 ► DeviceCMYK：PDF/X 出力条件は CMYK デバイスである必要があります。生成ページについては（テンプレートと取り込みページでは不可）、かわりに PDF_begin_page_ext( ) で defaultcmyk オプションを設定することもできます。
表示設定、表示・印刷領域	PDF_begin/end_document( ) で viewerpreferences オプションに対して viewarea・viewclip・printarea・printclip サブオプションを用いる際には、media・bleed 以外の値は避ける必要があります。
文書情報キー	PDF_set_info( ) で、Trapped 情報キーに対して、または対応する XMP プロパティ pdf:Trapped に対して、True か False 以外の値は避ける必要があります。
セキュリティ	PDF_begin_document( ) で userpassword・masterpassword・permissions オプションは避ける必要があります。
***PDF*バージョン・互換性**	PDF/X-1a:2001・PDF/X-3:2002 は PDF 1.3 に基づいています。PDF 1.4 以上を必要とする操作（透過設定やソフトマスクなど）は避ける必要があります。 PDF/X-1a:2003・PDF/X-3:2003 は PDF 1.4 に基づいています。PDF 1.5 以上を必要とする操作は避ける必要があります。 PDF/X-4・PDF/X-5 は PDF 1.6 に基づいています。PDF 1.7 以上を必要とする操作は避ける必要があります。
***PDF*取り込み（*PDI*）**	取り込まれる文書は、表 10.5 に従った互換 PDF/X レベルに準拠している必要があり、同じ出力インテントに従って作られている必要があります。

表 10.3 PDF/X 互換のために禁止または制約される操作一覧

項目	PDF/X 互換のために禁止または制約される PDFlib 関数・オプション
外部グラフィック内容（参照）	PDF/X-1/3/4：PDF_begin_template_ext( )・PDF_open_pdi_page( ) で reference オプションは避ける必要があります。 PDF/X-5g・PDF/X-5pg：PDF_begin_template_ext( )・PDF_open_pdi_page( ) で reference オプションで与えるターゲットは、右記の規格のいずれかに準拠している必要があります：PDF/X-1a:2003・PDF/X-3:2003・PDF/X-4・PDF/X-4p・PDF/X-5g・PDF/X-5pg。かつ、同じ出力インテントに対して作成されている必要があります。特定の XMP メタデータ項目がターゲット内に必要ですので、あらゆる PDF/X 文書がターゲットとして受け入れ可能なわけではありません。PDFlib 8 で生成された PDF/X 文書はターゲットとして利用可能です。 reference オプションと、必要な Acrobat 設定について詳しくは、73 ページ「3.2.5 外部 PDF 文書内の参照ページ」を参照してください。
レイヤー	PDF/X-1・PDF/X-3：レイヤーは PDF 1.5 を必要としますので、使用できません。 PDF/X-4・PDF/X-5：レイヤーは使用できますが、特定の PDF/X の規則に従う必要があります： ► レイヤーの可視性は、個々のレイヤーに対してビューア制御を与えるのではなく、文書バリアントで制御する必要があります。バリアントは、PDF_set_layer_dependency( )、引数 type=variant、およびバリアントに対するさまざまなオプションで作成できます。残念ながら、レイヤーバリアントの一覧は Acrobat 9 でのみ表示され、Acrobat X では表示されません。 ► PDF_define_layer( )・PDF_set_layer_dependency( ) のさまざまなオプションは避ける必要があります。 ► PDF/X-4:2010：PDF_set_layer_dependency( ) の createorderlist オプションは許されます。これは、Acrobat X でレイヤーの一覧を表示させるために必要です。
ファイルサイズ	PDF/X-4・PDF/X-5：生成 PDF 文書のファイルサイズは 2 GB を超えてはならず、PDF オブジェクトの数は 8,388,607 個未満でなければなりません。これらの制限について詳しくは 64 ページ「3.1.5 大容量 PDF 文書」を参照してください。

10.3.3 出力インテントと標準出力条件

出力インテント（出力条件ともいいます）は、意図される対象デバイスを定義します。これは主に、校正の信頼性を得るために有用です。出力インテントは、名前として指定することもできますし（標準出力インテントといいます）、ICC カラープロファイルで指定することもできます。詳細は PDF/X の種類によって異なります：

- PDF/X-1/3/4・PDF/X-5g：出力インテントに対する ICC プロファイルを埋め込みことによって。
- PDF/X-1・PDF/X-3：標準出力インテントの名前を与えることによって。標準出力インテントは、PDFlib に内部的に知られています。標準出力インテント名と、それに対応する出力条件の説明の全一覧は、PDFlib API リファレンスを参照してください。これらの出力インテントに対する ICC プロファイルは、ローカルに得られる必要はありません。追加の標準出力インテントを、*StandardOutputIntent* リソースカテゴリを用いて定義することもできます（59 ページ「3.1.3 リソース設定とファイル検索」参照）。PDF/X 処理ソフトウェアによって認識されるであろう標準出力インテントの名前だけを与えることはユーザー側の役割です。標準出力インテントは下記のように参照できます：

```
if (p.load_iccprofile("CGATS TR 001", "usage=outputintent") == -1)
{
    /* エラー */
}
```

PDF/X-3 出力を生成する場合、HKS・PANTONE・ICC ベース・Lab カラーのいずれかを使用する際には、標準出力インテントの名前を参照するだけでは充分ではなく、その出力デバイスの ICC プロファイルを埋め込む必要があります。

- PDF/X-4p・PDF/X-5pg：出力インテントに対する外部 ICC プロファイルを参照することによって（規格の名前の中の ***p*** は、外部プロファイルが参照されることを意味しています）。標準出力インテントと異なり、出力インテント ICC プロファイルは名前によって参照されるだけではなく、文書が生成される際に ICC プロファイルがローカルに得られる必要がある強い参照が作成されます。この ICC プロファイルは PDF 出力に埋め込まれませんが、それでも強い参照を作成するためにこれは PDF 生成時に得られる必要があります。***urls*** オプションで、ICC プロファイルが見つかる 1 個ないし複数の URL を与える必要があります：

```
if (p.load_iccprofile("CGATS TR 001",
        "usage=outputintent urls={http://www.color.org}") == -1)
{
    /* エラー */
}
```

適切な PDF/X 出力インテントを選ぶ　PDF/X 出力インテントの選択は通常、印刷出力をとりしきる印刷業者との話し合いで決まります。出力インテントの選択に関する情報が印刷所側から出てこないときは、表 10.4 に挙げる標準出力インテントをたたき台として使用することもできます（PDF/X FAQ より引用）。

表 10.4 代表的な出力条件のための適切な PDF/X 出力インテント

	欧州	北米
雑誌広告	FOGRA28	CGATS TR 00
新聞広告	IFRA26	IFRA30
枚葉オフセット	用紙に依存： タイプ 1・2（コート）：FOGRA27 タイプ 3（LWC）：FOGRA28 タイプ 4（非コート）：FOGRA29	用紙に依存： グレード 1・2（プレミアムコート）：FOGRA27 グレード 5：CGATS TR 001 非コート：FOGRA29
輪転オフセット	用紙に依存： タイプ 1・2（コート）：FOGRA28 タイプ 4（非コート・白）：FOGRA29 タイプ 5（非コート・黄味）：FOGRA30	用紙に依存： グレード 5：CGATS TR 001 非コート（白）：FOGRA29 非コート（黄味）：FOGRA30

10.3.4 PDI による PDF/X 文書の取り込み

既存の PDF 文書のページを PDF-X 準拠の出力文書へ取り込む場合には、特別な規則が適用されます（詳しくは 184 ページ「7.2 PDI で PDF ページを取り込み」参照）。取り込まれる文書はすべて、表 10.5 に従った互換 PDF/X 準拠レベルに準拠している必要があります。一般則としては、入力文書が生成出力文書と同じ PDF/X 準拠レベルに準拠している場合、ないし同じレベルの古いバージョンに準拠している場合には利用可能です。これに加え、ある種の組み合わせも利用可能です。ある特定の PDF/X 準拠レベルが PDFlib で設定されていて、かつ、取り込まれた文書もその互換レベルのいずれかを遵守しているならば、生成出力はその選択された PDF/X 準拠レベルに準拠していることが保証されます。取り込まれた文書で、選ばれた PDF/X レベルを遵守していないものは拒否されます。

表 10.5 さまざまな PDF/X 出力レベルに対する互換 PDF/X 入力レベル一覧

	取り込む文書の PDF/X レベル							
PDF/X 出力レベル	*PDF/X-1a:2001*	*PDF/X-1a:2003*	*PDF/X-3:2002*	*PDF/X-3:2003*	*PDF/X-4*	*PDF/X-4p*	*PDF/X-5g*	*PDF/X-5pg*
PDF/X-1a:2001	可							
PDF/X-1a:2003	可	可						
PDF/X-3:2002	可		可					
PDF/X-3:2003	可	可	可	可				
PDF/X-4	可	可	可	可	可	可		
PDF/X-4p	可	可	可	可	可	可[1]		
PDF/X-5g	可	可	可	可	可	可	可[2]	可[2]
PDF/X-5pg	可	可	可	可	可	可[1]	可[2]	可[1,2]

1. PDF_process_pdi( ) で action=copyoutputintent を指定すると、参照が外部出力インテント ICC プロファイルへ複製されます。
2. 取り込んだページが参照 XObject を含んでいるときは、PDF_open_pdi_page( ) は代理と参照の両方をターゲットへコピーします。

複数の PDF/X 文書を取り込む場合は、それらはすべて同一の出力条件に対して作成されている必要があります。たとえば、CMYK 出力インテントを持つ文書だけが、同じ CMYK 出力インテントを用いている文書へ取り込むことができます。

PDFlib は、いくつか特定の項目を修正することはできますが、完全な PDF/X 検証動作を行ったり、取り込み文書に完全な PDF/X 互換を強制したりするようにはもともと作られていません。たとえば PDFlib は、取り込んだ PDF ページに欠けているフォントを埋め込んだりはしませんし、取り込んだページに対して色補正なども一切行いません。

ページを取り込んだ結果としてできる PDF 出力文書が、入力文書（1 つまたは複数）と同じ PDF/X 準拠レベルと出力条件に準拠するようにしたい場合は、次のように、取り込まれた PDF の PDF/X 状態を取得することができます。

```
pdfxlevel = p.pcos_get_string(doc, "pdfx");
```

このステートメントは、取り込まれた文書が ISO PDF/X レベルに準拠している場合には、PDF/X 準拠レベルを表す文字列を得ます。そうでない場合には *none* を得ます。この返ってきた文字列を用いれば、*PDF_begin_document()* の *pdfx* オプションを用いて出力文書の PDF/X 準拠レベルを適切に設定することができます。

取り込み文書から PDF/X 出力インテントをコピー　PDF/X 準拠レベルの取得の他にも、次のように、取り込まれた文書から PDF/X 出力インテントをコピーすることもできます。

```
ret = p.process_pdi(doc, -1, "action=copyoutputintent");
```

この方法は、*PDF_load_iccprofile()* による出力インテント設定のかわりに利用することができ、取り込み文書の出力インテントを、生成する文書にコピーします。この出力インテントは、標準名と ICC プロファイルのどちらで定義されたものでもかまいません。出力インテントをコピーする方法は、取り込まれた PDF/A と PDF/X の文書に対して通用します。

生成する文書の出力インテントは、取り込み文書の出力インテントをコピーするか、あるいは明示的に *PDF_load_iccprofile( )* で *usage=outputintent* にして設定するか、そのどちらかの方法で必ず一度だけ設定する必要があります。

10.4 PDF/A によるアーカイビング

10.4.1 各種の PDF/A 規格

ISO 19005 標準で定められた各種の PDF/A 形式は、長期間にわたって安全にアーカイブできる、あるいは企業や政府の環境において信頼性を持ったデータ交換に利用できる、首尾一貫、かつ堅牢な PDF のサブセットを提供することを目的としています。

PDF/A 技術センター　PDFlib GmbH は PDF/A 技術センター（PDF/A Competence Center）の創立メンバーです。この組織の目的は、ISO 19005 に従った長期アーカイビングの分野における情報と経験の交換の促進です。PDF/A 技術センターのメンバーは、PDF/A 標準とその実装に関連する情報を積極的に交換しており、このテーマに関するセミナーやカンファレンスを開いています。詳しくは PDF/A 技術センターのウェブサイト *www.pdfa.org* を参照してください。

ISO 19005-1 で定義された PDF/A-1a:2005・PDF/A-1b:2005　PDF/A の目的は、電子文書の確実な長期保存です。その規格は PDF 1.4 をベースに、色・フォント・注釈などの要素の使用にいくつかの制約を課しています。下記の 2 種類の PDF/A-1 があり、どちらも PDFlib で作成・処理が可能です。

- ISO 19005-1 レベル B 準拠（PDF/A-1b）は、文書の体裁が長期にわたって保持されることを保証します。簡単にいえば PDF/A-1b は、文書を将来いつの日か処理するときにそれが今と同じに見えることを保証するものです。
- ISO 19005-1 レベル A 準拠（PDF/A-1a）は、レベル B をベースに、「タグ付き PDF」で知られた性質を追加します。すなわち、文書の論理構造と自然な読み上げ順序を保持するために、構造情報と、信頼できるテキストの意味を追加します。簡単にいえば PDF/A-1a は、文書を将来いつの日か処理するときにそれが今と同じに見えることを保証するのみならず、その内容（意味）を信頼性を持って解釈でき、身体障碍者にも利用できることを保証するものです。PDFlib の PDF/A-1a 対応は、タグ付き PDF を制作するための機能にもとづいています（270 ページ「10.5 タグ付き PDF」参照）。

以下、PDF/A-1 と言うとき（準拠レベルをつけずに）は、両方の準拠レベルを指すものとします。

実装根拠　下記の規格・文書が、PDFlib の PDF/A-1 の実装根拠を構成しています：

- PDF/A 規格（ISO 19005-1:2005）
- 技術正誤表 1（ISO 19005-1:2005/Cor 1:2007）。
- 技術正誤表 2（ISO 19005-1:2005/Cor.2:2011）。
- PDF/A 技術センターが発行したすべての関連 TechNote。

10.4.2 PDF/A 準拠の出力を生成

クックブック　PDF/A を生成するためのコードサンプルが PDFlib クックブックの pdfa カテゴリにあります。

PDFlib による PDF/A 準拠出力の作成は、以下の手段で実現されます。

- PDF のバージョン番号や、PDF/A 準拠キーなど、PDF/A のための形式設定のいくつかは、PDFlib が自動的に行います。
- PDFlib のクライアントプログラムにおいて、表 10.6 に示す特定の関数呼び出しやオプションを明示的に使用する必要があります。
- PDFlib のクライアントプログラムにおいて、表 10.7 に示す特定の関数呼び出しやオプション設定を避ける必要があります。
- 既存の PDF/A 準拠の文書からページを取り込むときは、ほかにも従うべき規則があります（265 ページ「10.4.3 PDF/A 文書を PDI で取り込み」参照）。

PDFlib のクライアントプログラムがこれらの規則に従うなら、有効な PDF/A 出力が保証されます。PDF/A 作成の規則違反を検出したときは、PDFlib は例外を発生させるので、アプリケーション側でそれを処理する必要があります。エラーが起きると PDF 出力は作成されません。

PDF/A-1b に必要な操作　PDF/A 準拠の出力を生成するために必要なすべての操作を表 10.6 に示します。各項目は、別記ない限り両方の PDF/A 準拠レベルにあてはまります。PDF/A モードにいるときに、必要な関数をもし 1 つでも呼び出さなかった場合は、例外が発生します。

表 10.6 PDF/A-1 レベル A・B 準拠のために行うべき操作一覧

項目	PDF/A 互換のために必要な PDFlib 関数・オプション
準拠レベル	PDF_begin_document( ) の pdfa オプションに、求める PDF/A 準拠レベル、すなわち PDF/A-1a:2005 か PDF/A-1b:2005 を設定する必要があります。
出力条件（出力インテント）	デバイス依存色空間のグレー・RGB・CMYK のいずれかが文書で使われているときは、PDF_begin_document( ) の直後に、PDF_load_iccprofile( ) を usage=outputintent にして呼び出すか、または PDF_process_pdi( ) を action=copyoutputintent にして呼び出す必要があります（ただし両方式の併用は不可）。出力インテントを使うときは、ICC プロファイルを埋め込む必要があります（PDF/X と違って PDF/A では、標準出力条件では不十分です）。標準出力条件のためのプロファイルを埋め込むには、PDF_load_iccprofile( ) で embedprofile オプションを使います。
フォント	PDF_load_font( ) で embedding オプションを true にする必要があります（このオプションを受け取る他の関数でも同様）。PDF コアフォントも埋め込む必要があるので注意してください。ただし不可視テキスト（主に OCR の結果のために有用です）でのみ使われているフォントは例外で、埋め込みの必要はありません。これは optimizeinvisible オプションで制御できます。
グレースケールカラー	グレースケール画像と、グレー色空間による PDF_setcolor( ) は、出力条件がグレースケール・RGB・CMYK デバイスのいずれかである場合か、あるいは PDF_begin_page_ext( ) で defaultgray オプションを設定している場合にのみ用いることができます。
***RGB*カラー**	RGB 画像と、RGB 色空間による PDF_setcolor( ) は、出力条件が RGB デバイスである場合か、あるいは PDF_begin_page_ext( ) で defaultrgb オプションを設定している場合にのみ用いることができます。
***CMYK*カラー**	CMYK 画像と、CMYK 色空間による PDF_setcolor( ) は、出力条件が CMYK デバイスである場合か、あるいは PDF_begin_page_ext( ) で defaultcmyk オプションを設定している場合にのみ用いることができます。

禁止・制限される操作　PDF/A 準拠の出力を生成する際に禁止されるすべての操作を表 10.7 に示します。各項目は、別記ない限り両方の PDF/A 準拠レベルにあてはまります。

PDF/A モードにいるときに、禁止された関数をもし 1 つでも呼び出した場合は、例外が発生します。同様に、取り込んだ PDF 文書がカレントの PDF/A 出力レベルに準拠していなかったときも、その PDI 呼び出しは失敗します。

表 10.7 PDF/A 互換のために禁止・制限される操作一覧

項目	PDF/A 互換のために禁止される PDFlib 関数・オプション
注釈	PDF_create_annotation( ) : type=FileAttachment の注釈は不可。テキスト注釈の場合には、zoom・rotate オプションを true に設定してはいけません。annotcolor・interiorcolor オプションは、RGB の出力インテントが指定されているときにのみ使う必要があります。fillcolor オプションは、RGB か CMYK の出力インテントが指定されているときにのみ使う必要があり、それぞれ rgb か cmyk 色空間を使う必要があります。
フォームフィールド	PDF_create_field( )・PDF_create_fieldgroup( ) によるフォームフィールドの作成は不可。
アクション・*JavaScript*	PDF_create_action( ) : type=Hide・Launch・ResetForm・ImportData・JavaScript のアクションは不可。type=name については NextPage・PrevPage・FirstPage・LastPage のみ可。
画像	PDF_load_image( ) で OPI-1.3・OPI-2.0 オプションと interpolate=true オプションは不可。
***ICC*プロファイル**	PDF_load_iccprofile( ) で明示的に、あるいは PDF_load_image( ) と ICC タグ付き画像で暗黙的に読み込まれる ICC プロファイルは、ICC 仕様 ICC.1:1998-09 と、その補遺 ICC.1A:1999-04（内部プロファイルバージョン 2.x）に準拠している必要があります。
ページサイズ	PDF/A には、厳格なページサイズ制限はありません。しかし、Acrobat での問題を避けるため、ページサイズ（幅・高さ、すべての枠項目）は 3 ～ 14400 ポイント（508 cm）の範囲内に収めることを推奨します。
テンプレート	PDF_begin_template_ext( ) で OPI-1.3・OPI-2.0 オプションは不可。
透過	画像にソフトマスクは不可。PDF_load_image( ) で masked オプションは、マスクが 1 ビット画像を参照する場合以外は不可。暗黙的な透過（アルファチャンネル）を持つ画像は不可ですので、それらは PDF_load_image( ) の ignoremask オプションで読み込む必要があります。 PDF_create_gstate( ) で opacityfill・opacitystroke オプションは、値 1 以外は不可。blendmode を使う場合は Normal にすること。 PDF_create_annotation( ) で opacity オプションは不可。
透過グループ	PDF_begin/end_page_ext( )・PDF_begin_template_ext( )・PDF_open_pdi_page( ) の transparencygroup オプションは不可。
セキュリティ	PDF_begin_document( ) で userpassword・masterpassword・permissions オプションは不可。
***PDF*バージョン/互換性**	PDF/A は PDF 1.4 をベースにしています。PDF 1.5 以上を要する操作（レイヤー等）は不可。
***PDF*取り込み（*PDI*）**	取り込む文書は、出力文書と互換な PDF/A レベルに準拠している必要があり、かつ、互換な出力インテントに従って作成されている必要があります（表 10.10 参照）。
メタデータ	定義済みの XMP スキーマ（PDFlib API リファレンス参照）はすべて使えます。それ以外のスキーマ（拡張スキーマ）を使いたいときは、そのスキーマ自体が、PDF/A 拡張スキーマコンテナスキーマを使って包含されている必要があります。
外部内容	PDF_begin_template_ext( )・PDF_open_pdi_page( ) で reference オプションは不可。
ファイルサイズ	生成 PDF 文書のファイルサイズは 2 GB を超えてはならず、PDF オブジェクトの数は 8,388,607 個未満でなければなりません。これらの制限について詳しくは 64 ページ「3.1.5 大容量 PDF 文書」を参照してください。

PDF/A-1a で加わる要請と制限　PDF/A-1a を作成するときは、270 ページ「10.5 タグ付き PDF」で述べる、タグ付き PDF の出力作成のための要請をすべて守る必要があります。その他にもいくつかの操作が禁止または制約されますので、それを表 10.9 に示します。

構造情報の適切な作成はユーザー側の役割であり、PDFlib は意味上の制限を検証も強制もしません。文書のテキスト全体を 1 個の構造要素に入れたら、技術的には正しい PDF/A-1a ですが、意味の忠実再生という目標に反していますので、ひいては PDF/A-1a の精神にも反します。

表 10.8 PDF/A-1a 準拠のための追加要請一覧

項目	PDF/A-1a 準拠のために必要な PDFlib 関数・オプション
タグ付き*PDF*	タグ付き PDF に対する要請はすべて守る必要があります（270 ページ「10.5 タグ付き PDF」参照）。 下記を強く推奨します。 ▸ デフォルト文書言語を指定するために PDF_begin/end_document( ) で Lang オプションを与えるべきです。 ▸ 文書のデフォルトの言語と異なる内容項目に対してはすべて、その PDF_begin_item( ) で Lang オプションを適切に設定する必要があります。 ▸ 画像などの非テキストの内容項目は、PDF_begin_item( ) で Alt オプションを使って代替テキスト記述を与える必要があります。 ▸ ロゴや記号などの非 Unicode のテキストは、それを内容項目として包含する PDF_begin_item( ) で ActualText オプションに適切な代替テキストを与える必要があります。 ▸ 略語や頭字語は、それを内容項目として包含する PDF_begin_item( ) で E オプションに適切な展開テキストを与える必要があります。
注釈	PDF_create_annotation( ) : contents オプションには、空でない文字列を与える必要があります。

表 10.9 PDF/A-1a 準拠のために追加で禁止または制約される操作一覧

項目	PDF/A-1a 準拠のために禁止または制約される PDFlib 関数・オプション
フォント	PDF_load_font( ) で monospace オプションと unicodemap=false・autocidfont=false は不可（これらのオプションを受け取る他の関数についても同様）。
***PDF*取り込み（*PDI*）**	取り込む文書は、出力文書と互換な PDF/A レベルに準拠している必要があり（表 10.10 参照）、かつ、同じ出力インテントに従って作成されている必要があります。

出力インテント　出力条件は、意図する対象デバイスを定義します。これは一貫したカラーレンダリングのために重要です。PDF/X の場合、出力インテントは厳格に必要ですが、それと違って PDF/A では、出力インテントは指定することもできますが、必須ではありません。出力インテントが必要なのは、デバイス依存色が文書で使われている場合だけです。出力インテントは、ICC プロファイルを使って指定することができます。出力インテントは下記のように指定できます。

```
icc = p.load_iccprofile("sRGB", "usage=outputintent");
```

ICC プロファイルを読み込む方法のほかに、出力インテントは、取り込んだ PDF/A 文書からコピーすることもできます。それには *PDF_process_pdi( )* で *action=copyoutputintent* オプションを指定します。

PDF/A と PDF/X を同時に作成　PDF/A-1 文書は、同時に PDF/X-1a:2003・PDF/X-3:2003・PDF/X-4 のいずれかにも準拠させることが可能です（ただし PDF/X-4p・PDF/X-5 は不可）。そのようなコンボファイルを作成するには、*PDF_begin_document()* で *pdfa* オプションと *pdfx* オプションに適切な値を与えます。例：

```
ret = p.begin_document("combo.pdf", "pdfx=PDF/X-4 pdfa=PDF/A-1b:2005");
```

出力インテントは、PDF/A と PDF/X に対して同じにする必要があり、かつ、出力デバイスの ICC プロファイルとして指定する必要があります。PDF/X の標準出力条件が使えるのは、*embedprofile* オプションと組み合わせた場合だけになります。

10.4.3 PDF/A 文書を PDI で取り込み

PDF/A 準拠の出力文書に既存の PDF 文書を取り込むときは、特別な規則が適用されます（PDI について詳しくは 184 ページ「7.2 PDI で PDF ページを取り込み」を参照）。あらゆる取り込み文書は、表 10.10 に従ったカレントの PDF/A モードと互換な PDF/A 準拠レベルに準拠している必要があります。

注 PDFlib は、入力 PDF 文書の PDF/A 準拠に関する検証は行わず、また、任意の入力 PDF から有効な PDF/A を生成することもできません。

ある特定の PDF/A 準拠レベルが PDFlib で設定されていて、かつ、取り込んだ文書がそれと互換なレベルを厳守しているならば、生成される出力は、選ばれた PDF/A 準拠レベルに準拠することが保証されます。カレントの PDF/A レベルと非互換の文書は *PDF_open_pdi_document()* で拒絶されます。

表 10.10 各 PDF/A 出力レベルに互換な PDF/A 入力レベル

	取り込む文書の PDF/A レベル	
PDF/A 出力レベル	**PDF/A-1a:2005**	**PDF/A-1b:2005**
PDF/A-1a:2005	—	可
PDF/A-1b:2005	—	可

クックブック 完全なコードサンプルがクックブックの `pdfa/import_pdfa` トピックにあります。

1 個ないし複数の PDF/A 文書を取り込むときは、そのすべてが表 10.11 に従った互換な出力条件で作成されている必要があります。すべての取り込み文書の出力インテントは同一か互換である必要があり、この条件を満たすよう手配するのはユーザー側の役割です。

表 10.11 PDF/A 文書を取り込む際の出力インテントの互換性

	取り込む文書の出力インテント			
生成する文書の出力インテント	**なし**	**グレースケール**	**RGB**	**CMYK**
なし	有	—	—	—
グレースケールの ICC プロファイル	有	有[1]	—	—
RGB の ICC プロファイル	有	—	有[1]	—
CMYK の ICC プロファイル	有	—	—	有[1]

1. 取り込む文書の出力インテントと、生成する文書の出力インテントが、同一である必要があります。

PDFlib は、いくつか特定の項目を修正することはできますが、PDF/A の完全検証に利用されるようにも、また、取り込んだ文書に完全な PDF/A 準拠を強制するようにも作られていません。たとえば取り込んだ PDF のページに足りないフォントがあっても、PDFlib はそれの埋め込みはしません。

取り込んだページを連結する際に、できあがる PDF 出力文書が入力文書（群）と同じ PDF/A 準拠レベル・出力条件に準拠するようにしたければ、取り込んだ PDF の PDF/A 状況を下記のように取得することができます。

```
pdfalevel = p.pcos_get_string(doc, "pdfa");
```

この命令は、取り込んだ文書が PDF/A レベルに準拠していればその PDF/A 準拠レベルを示す文字列を取得し、そうでなければ *none* を返します。この返された文字列を使えば、*PDF_begin_document( )* で *pdfa* オプションを使って、出力文書の PDF/A 準拠レベルを適切に設定することができます。

取り込んだ文書から PDF/A 出力インテントをコピー　PDF/A準拠レベルを取得する以外の方法として、PDF/A 出力インテントは、取り込んだ文書からコピーすることもできます。PDF/A 文書には必ずしも出力インテントがあるとはかぎらないので（出力インテントが必須な PDF/X とは違います）、それをコピーしようとする前に、まず pCOS を使って出力インテントの存在を確かめる必要があります。

クックブック **完全なコードサンプルがクックブックの pdfa/import_pdfa トピックにあります。**

これは、*PDF_load_iccprofile( )* を使って出力インテントを設定する方法のかわりに使うことができ、取り込んだ文書の出力インテントを、生成する出力文書へコピーします。出力インテントのコピーは、取り込んだ PDF/A と PDF/X の文書で動作します。

生成する出力文書の出力インテントは、取り込んだ文書の出力インテントをコピーするか、あるいは *PDF_load_iccprofile( )* の *usage* オプションに *outputintent* を設定して明示的に設定するか、そのいずれかの方法で 1 回だけ設定する必要があります。出力インテントの設定は、*PDF_begin_document( )* の直後に行う必要があります。

10.4.4 PDF/A 作成のためのカラー戦略

PDF/A の色処理に関する要請は、わかりにくいことがあるかもしれません。PDF/A の利用設計にあたっては、表 10.12 にまとめたカラー戦略が役立つでしょう。多くの場面でうまくいく最も簡単なアプローチは、sRGB 出力インテントプロファイルを使うことであり、なぜならそれは CMYK 以外の代表的な多くの色空間に対応しているからです。それに加えて、sRGB は PDFlib で内部的に既知なので、外部プロファイルデータや設定が一切必要ありません。色空間の供給元としてありえるのは以下のとおりです。

- *PDF_load_image( )* で取り込んだ画像
- *PDF_setcolor( )* を使った明示的な色指定
- テキストフロー内などでのオプションリストによる色指定
- インタラクティブ要素では枠色の指定が可能

表 10.12 PDF/A のカラー戦略

出力インテント	文書で使える色空間				
	CIELab[1]	ICC ベース	グレースケール[2]	RGB[2]	CMYK[2]
なし	可	可	—	—	—

表 10.12 PDF/A のカラー戦略

出力インテント	文書で使える色空間				
	CIELab[1]	ICC ベース	グレースケール[2]	RGB[2]	CMYK[2]
グレースケールの*ICC*プロファイル	可	可	可	―	―
*RGB*の*ICC*プロファイル。例：*sRGB*	可	可	可	可	―
*CMYK*の*ICC*プロファイル	可	可	可	―	可

1. CIELab カラーの LZW 圧縮された TIFF 画像は RGB に変換されます。
2. ICC プロファイルのないデバイス色空間

黒のテキスト出力を、出力インテントプロファイルを一切必要とせずに作成したいときは、CIELab 色空間が使えます。Lab カラー値 *(0, 0, 0)* はデバイス独立な様式で純黒を指定し、出力インテントプロファイル一切なしで PDF/A 準拠です（出力インテントプロファイルを必要とする DeviceGray とは異なり）。PDFlib は各ページの冒頭で自動的に、カレント色を黒に初期化します。ICC 出力インテントが指定されているかいないかによって、黒を選択するために DeviceGray と Lab のどちらの色空間が使われるかが決まります。手動で Lab の黒色に設定するには、下記の呼び出しを使います。

```
p.setcolor("fillstroke", "lab", 0, 0, 0, 0);
```

表 10.12 に挙げた色空間以外にも、スポットカラーが、その対応する代替色空間に従って使えます。PDFlib は内蔵の HKS と PANTONE のスポットカラーについては CIELab を代替色に使っていますので、これらはかならず PDF/A で利用可能です。カスタムのスポットカラーについては、PDF/A の出力インテントと互換になるように代替色空間を選ぶ必要があります。

注 PDF/A と色空間に関してさらに詳しい情報は、www.pdfa.org の PDF/A 技術センターの Technical Note 0002 にあります。

10.4.5 PDF/A のための XMP 文書メタデータ

PDF/A-1 は、PDF 文書にメタデータを埋め込むために、XMP 形式に強く依存しています。ISO 19005-1 は XMP 2004 仕様を参照しており[1]、それより前や後の XMP 仕様には対応していません。PDF/A-1 では、2 つの種類の文書レベルメタデータに対応しています。1 つは、よく知られたメタデータスキーマを集めたもので、定義済みスキーマと呼ばれており、もう 1 つはカスタム拡張スキーマです。PDFlib は、XMP の中の必須の PDF/A 準拠項目群を自動的に作成するほか、いくつかのよく使われる項目も自動的に作成します（*CreationDate* 等）。

ユーザーが作成した文書メタデータは、*PDF_begin/end_document()* の *metadata* オプションで与えることができます。PDF/A モードでは、PDFlib は、ユーザーが与えた XMP 文書メタデータが PDF/A の要請に準拠しているかどうかを検証します。コンポーネントレベル（ページ・画像等）のメタデータについては、PDF/A の要請はありません。

取り込み PDF 文書の中の XMP メタデータは、pCOS パス */Root/Metadata* を用いて入力 PDF から抽出することができます。

1. *www.aiim.org/documents/standards/xmpspecification.pdf* 参照

クックブック 完全なコードサンプルがクックブックのinterchange/import_xmp_from_pdfトピックにあります。

定義済み XMP スキーマ PDF/A-1 では、XMP 2004 の中のすべてのスキーマに対応しています。これらは定義済みスキーマと呼ばれます。これを表 10.13 に列挙し、あわせてその名前空間 URI と、望ましい名前空間接頭辞を示します。XMP 2004 の中で列挙されている定義済みスキーマのプロパティだけを使う必要があります。PDF/A-1 のための定義済み XMP スキーマの中のすべてのプロパティの完全な一覧は、PDF/A 技術センターから得ることができます。

表 10.13 PDF/A-1 のための定義済み XMP スキーマ

スキーマの名称と説明 （詳しくは XMP 2004 を参照）	名前空間 URI	望ましい 名前空間接頭辞
Adobe PDF スキーマ	http://ns.adobe.com/pdf/1.3/	pdf
Dublin Core スキーマ	http://purl.org/dc/elements/1.1/	dc
EXIF 独自プロパティ用 EXIF スキーマ	http://ns.adobe.com/exif/1.0/	exif
TIFF プロパティ用 EXIF スキーマ	http://ns.adobe.com/tiff/1.0/	tiff
Photoshop スキーマ	http://ns.adobe.com/photoshop/1.0/	photoshop
XMP 基本ジョブチケットスキーマ	http://ns.adobe.com/xap/1.0/bj	xmpBJ
XMP 基本スキーマ	http://ns.adobe.com/xap/1.0/	xmp
XMP メディア管理スキーマ	http://ns.adobe.com/xap/1.0/mm/	xmpMM
XMP ページドテキストスキーマ	http://ns.adobe.com/xap/1.0/t/pg/	xmpTPg
XMP 権利管理スキーマ	http://ns.adobe.com/xap/1.0/rights/	xmpRights

XMP 拡張スキーマ 自分が必要とするメタデータが、定義済みスキーマに含まれていないときは、XMP 拡張スキーマを定義することもできます。PDF/A-1 では、カスタムスキーマを PDF/A 文書に埋め込む際に用いなければならない拡張方式を記述しています。表 10.14 に、1 個ないし複数の拡張スキーマを記述するために用いなければならないスキーマをまとめ、あわせてその名前空間 URI と、必要な名前空間接頭辞を示します。名前空間接頭辞の、必要な、という点に注意してください（定義済みスキーマで示した名前空間接頭辞とは異なり、単にこうつけるのが望ましいというだけではなく、それぞれこの通りにつける必要があります）。

PDF/A-1 で使う XMP 拡張スキーマをどう構築するかという詳細は、この説明書の対象範囲を超えています。詳しい方法説明は、PDF/A 技術センターから得られます。

XMP 文書メタデータパッケージは、*PDF_begin_document()* か *PDF_end_document()* またはその両方の *metadata* オプションに与えることができます。

クックブック 完全なコードサンプルと XMP 作成例がクックブックの pdfa/pdfa_extension_schema・pdfa/pdfa_extension_schema_with_type トピックにあります。

表 10.14 PDF/A-1 拡張スキーマコンテナスキーマと補助スキーマ

スキーマの名称と説明	名前空間 URI[1]	必要な名前空間接頭辞
PDF/A 拡張スキーマコンテナスキーマ：あらゆる拡張スキーマ記述を埋め込むためのコンテナ	http://www.aiim.org/pdfa/ns/extension/	pdfaExtension
PDF/A スキーマ値種別：1 個の拡張スキーマが任意個数のプロパティを持つのを記述	http://www.aiim.org/pdfa/ns/schema#	pdfaSchema
PDF/A プロパティ値種別：1 個のプロパティを記述	http://www.aiim.org/pdfa/ns/property#	pdfaProperty
PDF/A ValueType 値種別：拡張スキーマプロパティで用いるカスタム値種別を記述。XMP 2004 の種別一覧にない種別を使いたいときにのみ必要となります。	http://www.aiim.org/pdfa/ns/type#	pdfaType
PDF/A フィールド種別スキーマ：種別が構造化されている場合に、その中のフィールドを記述	http://www.aiim.org/pdfa/ns/field#	pdfaField

1. 名前空間 URI は、ISO 19005-1 では誤って列挙されており、技術正誤表 1 で修正されました。

10.5 タグ付き PDF

タグ付き PDF とはある種の改良 PDF であり、PDF ビューアで追加の機能を利用することができます。たとえばアクセシビリティ対応や、テキストの折り返し表示、正しいテキスト抽出、他の文書形式への変換などです。他の文書形式としては RTF・XML などが選べます。

PDFlib はタグ付き PDF の生成に対応しています。ただし、タグ付き PDF を作成するには、クライアントがその文書の内部構造に関して情報を提供し、PDF 出力生成の際にある種の規則に従う必要があります。PDFlib は、標準タグ名（標準タグの一覧は PDFlib API リファレンスにあります）だけでなく、カスタムタグにも対応しています。カスタムタグは、各カスタムタグを標準タグ名のいずれか 1 つへマップするロールマップを必要とします。

クックブック タグ付き PDF の諸側面に関するコードサンプルが PDFlib クックブックの document_interchange カテゴリにあります。

10.5.1 PDFlib でタグ付き PDF を生成

クックブック 完全なコードサンプルがクックブックの document_interchange/starter_tagged トピックにあります。

必要な操作 タグ付き PDF 出力の生成に必要なすべての操作を表 10.15 に挙げます。タグ付き PDF モードにおいて、必要な関数を 1 つでも呼び出さなかった場合は例外が発生します。

表 10.15 タグ付き PDF 生成のために行うべき操作一覧

項目	タグ付き PDF 互換のために必要な PDFlib 関数・オプション
タグ付き*PDF*出力	PDF_begin_document( ) の tagged オプションを true に設定する必要があります。
文書言語	PDF_begin_document( ) の lang オプションを指定して文書の自然言語を指定するべきです。はじめは文書全体に対して設定するべきですが、後から任意の構造レベルの項目についてそれぞれ上書き設定することも可能です。
構造情報	構造情報とアーティファクトはそのように識別指定する必要があります。内容生成のための API 関数はいずれも PDF_begin_item( ) と PDF_end_item( ) とで挟んでいくことになります。

Unicode マッピング タグ付き PDF 内のすべてのテキスト内容は、その文書が必ずアクセシブル（たとえばソフトウェアで読み上げ可能）になるように、正しい Unicode マッピングを持つ必要があります。PDFlib はほとんどすべてのフォント / エンコーディングの組み合わせに対して内部的に Unicode マッピングを生成しますので、PDF 出力は技術的に Unicode マッピングを持ちます。しかし、いくつかの記号に対して生成される PUA 値は、再生可能なテキストにはなりません。テキストの検索性を改善するために、内容の代替テキストを、*PDF_begin_item( )* で *ActualText* または *Alt* オプションで与えることを推奨します。タグなし PDF モードでは、これは *PDF_begin_mc( )* の *ActualText* オプションで実現できます。

ページ内容の順序 ページ内容を定義するテキスト・グラフィック・画像オペレータ群の順序のことを内容ストリーム順序といいます。これに対し、論理構造ツリーによって定義される内容順序のことを論理順序といいます。タグ付き PDF の生成の際には、クライアントは内容順序に関してある種の規則に従うことが必要になります。

自然でかつ推奨する方式は、1 つの構造要素の中の各部分を順番に連続的にすべて生成して、その後に次の要素へと次々進んでいくというやり方です。技術用語でいえば、構造ツリーを 1 回の深さ優先巡回だけで作成しようということです。

もう 1 つの方式は、避けるべき方式ではありますが、最初の要素を一部分だけ出力し、次の要素へ移ってまた一部分だけ出力した後、最初の要素へ戻り、等というやり方です。この方式では構造ツリーは複数の巡回で作成され、その一回一回の巡回では 1 つの要素の一部分しか生成されません。

PDF によるページ取り込み　タグ付き PDF 文書や、その他構造情報を含んだ PDF 文書の中のページはタグ付き PDF モードでは取り込むことができません。なぜなら取り込まれた文書構造が、生成される文書構造とぶつかってしまうためです。

逆に、構造のない文書の中のページは取り込むことができます。ただしそうしたページは、適切な *ActualText* でタグ付けされていない限り、Acrobat のアクセシビリティ機能では「あるがまま」に扱われます。

アーティファクト　著者の書いたもとの内容以外のグラフィックやテキストのことをアーティファクトといいます。アーティファクトは *Artifact* 擬似タグを用いてそのように識別指定する必要があり、また、以下のいずれかの種類に分類する必要があります。

- **ページネーション**：柱やノンブルのような機能。
- **レイアウト**：罫線、表の影のような組版・デザイン要素。
- **ページ**：トンボやカラーバーのような印刷工程補助。

アーティファクトの識別指定は厳密に必須ではありませんが、テキストの折り返しやアクセシビリティをうまく動作させるには強く推奨します。

インラインアイテム　PDF はブロックレベル構造要素（BLSE ＝ block-level structure element）とインラインレベル構造要素（ILSE ＝ inline-level structure element）を定義します（厳密な定義は ***PDFlib API* リファレンス**参照）。BLSE は他の BLSE を含むことも内容本体を含むこともできますが、ILSE は必ず内容を直接含みます。この他に、PDFlib では以下の違いがあります。

表 10.16 通常アイテムとインラインアイテム

	通常アイテム	インラインアイテム
対象	すべてのグループ化要素と BLSE	すべての ILSE と非構造タグ（擬似タグ）
通常かインラインかの設定を換えられる	×	ASpan アイテムだけ可
文書の構造ツリーの一部である	○	×
複数ページにわたることができる	○	×
他のアイテムで中断できる	○	×
一時停止・アクティブ化できる	○	×
任意の深さに入れ子できる	○	他のインラインアイテムによる入れ子のみ

ASpan アイテムを通常にするかインラインにするかはクライアント側で *PDF_begin_item()* の *inline* オプションで制御します。たとえば段落が複数ページにわたっていたり複数言語を含んでいたりするときなどは、その段落に対するアクセシビリティスパンを通常

（*inline=false*）に設定することを推奨します。または、アイテムをいったん閉じて、次のページでまた新しいアイテムを始めてもよいでしょう。インラインアイテムは、それを開いたページ上で閉じなければなりません。

推奨操作　タグ付き PDF 出力を生成する際に必須ではないが推奨されるすべての操作を表 10.17 に挙げます。これらの機能は厳密に必須ではありませんが、生成されるタグ付き PDF 出力の品質を向上させますので、推奨します。

表 10.17 タグ付き PDF 生成のために推奨される操作一覧

項目	タグ付き PDF 互換のために推奨される PDFlib 関数・オプション
ハイフネーション	ワードブレーク（単語を行末で二分する）をハードハイフン（U+002D）でなくソフトハイフン（U+00A0）で表現します。
単語間区切り	単語の間をスペース（U+0020）で区切ります。位置合わせ上厳密に必須でない場合にも、そうすること。autospace パラメタを使うと、show 関数群のうちのいずれかを呼び出すたびにスペースが自動生成されます。
アーティファクト	純粋な内容をページアーティファクトから区別するために、PDF_begin_item( ) で tag=Artifact を指定してアーティファクトをそのように識別指定します。
***Type 3*フォントのプロパティ**	タグ付き PDF 文書で使われている Type 3 フォントについてはすべて、PDF_begin_font( ) で familyname・stretch・weight オプションに適切な値を与える必要があります。
インタラクティブ要素	リンクなどのインタラクティブ要素は、文書構造に組み込む必要があり、かつ必要に応じて、代替テキストを与えるなどしてアクセシブルにする必要があります。インタラクティブ要素のタブ順は、PDF_begin/end_document( ) の taborder オプションで指定できます（インタラクティブ要素が文書構造に適切に組み込んであるなら、これは必須ではありません）。

10.5.2 直接テキスト出力・テキストフローによるタグ付き PDF の作成

最小限のタグ付き PDF の作成例　次の作成例コードは非常にシンプルなタグ付き PDF 文書を作成するものです。その構造ツリーはただ 1 個の *P* 要素だけを含んでいます。このコードでは *autospace* 機能を用いて、テキスト部分どうしの間に自動的にスペースを生成させています。

```
if (p.begin_document("hello-tagged.pdf", "tagged=true") == -1)
        throw new Exception("エラー：" + p.get_errmsg());

/* テキスト部分どうしの間にスペースを自動作成 */
p.set_parameter("autospace", "true");

/* 最初の構造要素を文書構造ルート(=0)の子として開く */
id = p.begin_item("P", "Title={Simple Paragraph}");

p.begin_page_ext(0, 0, "width=a4.width height=a4.height");
font = p.load_font("Helvetica-Bold", "unicode", "");

p.setfont(font, 24);
p.show_xy("Hello, Tagged PDF!", 50, 700);
p.continue_text("This PDF has a very simple");
p.continue_text("document structure.");

p.end_page_ext("");
```

```
p.end_item(id);
p.end_document("");
```

テキストフローによるタグ付き PDF の生成　テキストフロー機能（205 ページ「8.2 複数行のテキストフロー」参照）は、テキスト組版のためのさまざまな強力な機能を提供します。そこではテキストの各部分はもはやクライアントの制御下にはなく、PDFlib によって自動的に組版されるので、テキストフローによるタグ付き PDF の生成の際には以下のような特別な注意を払う必要があります。

- テキストフローの中で構造要素を入れることはできませんが、1 つのテキストフローはめ込み枠の内容をまるごと、1 つの構造要素の中に入れることはできます。
- 1 つのテキストフローの中のすべての部分（ある 1 つのテキストフローハンドルによる *PDF_fit_textflow()* に対するすべての呼び出し）は、ただ 1 つの構造要素の中に入れなければなりません。
- 1 つのテキストフローの中の各部分は複数のページにわたる可能性があり、その各ページには他の構造要素が含まれる可能性があるので、適切な親アイテムを選ぶよう注意を払う必要があります（親パラメタとして -1 を用いるのは、間違った親要素を指し示してしまうかもしれないので、避けるべきです）。
- 範囲枠機能を使ってテキストフロー内にリンクなどの注釈を作成する場合は、構造ツリー内における注釈の位置に対する制御を保つことは困難です。

10.5.3 複雑なレイアウトにおけるアイテムのアクティブ化

上から下へ素直に流れない複雑なレイアウトにおける構造情報の作成を容易にするため、PDFlib はアイテムのアクティブ化という機能に対応しています。この機能を利用すると、以前に作成した構造要素をアクティブ化することができます。この機能は、開発者が複数の構造の枝を同時に並行して処理していかなければならないような状況において活用することができます。その際、それぞれの枝は複数のページにわたってもかまいません。この技法が役立つ典型的状況は以下の通りです。

- 多段組のページ
- 本文の途中に挿入する内容。要約や挿入コラムなど。
- 段組の途中に配置する表・イラスト。

アクティブ化機能を利用すると、論理構造の複数の枝どうしの間を行ったり来たりすることができるようになるので、上記のような状況でページ内容を生成することが容易になります。これは、それぞれの枝を 1 つずつ完了させていくよりもずっと効率的です。アクティブ化機能の具体的な利用例として、図 10.1 に示すページレイアウトを例に挙げて見てみましょう。このページレイアウトには本文の段組が 2 つあり、その途中に表が 1 つと、注釈コラムの枠線（背景が影）が 1 つあって、さらにヘッダとフッタもついています。

ページ内容を論理順序で生成　論理構造の観点からいえばページ内容は次の順序で作成するべきです。左の段組、右の段組（ページの右下部分）、表、コラム、ヘッダ・フッタ。次の擬似コードはこの順序を実装しています。

```
/* ページレイアウトを論理構造順序で作成 */

id_art = p.begin_item("Art", "Title=Article");

id_sect1 = p.begin_item("Sect", "Title={First Section}");
        /* 1 左の段組の上の部分を作成 */
        p.set_text_pos(x1_left, y1_left_top);
        ...
```

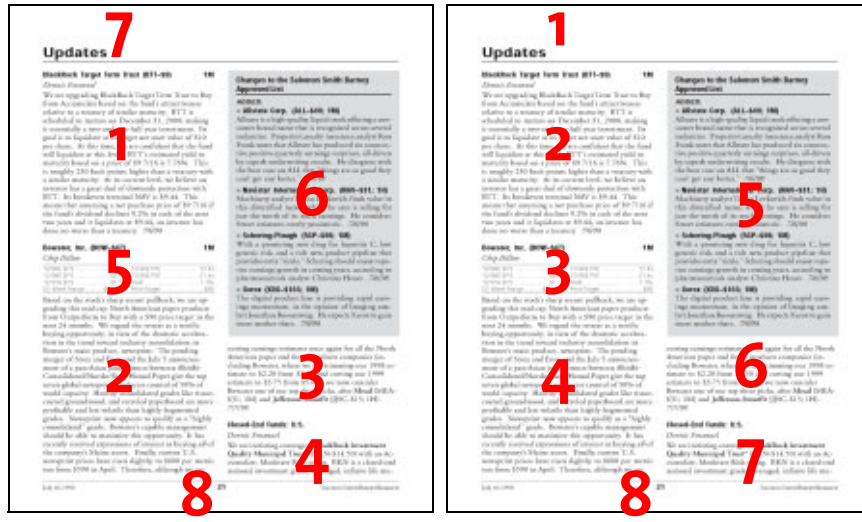


図 10.1
複雑なページレイアウトを論理構造順序で作成したもの（左）と視覚順序で作成したもの（右）。右側では 1 番目の項に対してアイテムのアクティブ化を用いてから 4 と 6 の部分を続けている。

```
        /* 2 左の段組の下の部分を作成 */
        p.set_text_pos(x1_left, y1_left_bottom);
        ...
        /* 3 右の段組の上の部分を作成 */
        p.set_text_pos(x1_right, y1_right_top);
        ...
p.end_item(id_sect1);

id_sect2 = p.begin_item("Sect", "Title={Second Section}");
        /* 4 右の段組の下の部分を作成 */
        p.set_text_pos(x2_right, y2_right);
        ...
        /* 2番目の項は次ページへ続く可能性あり */
p.end_item(id_sect2);

String optlist = "Title=Table parent=" + id_art;
id_table = p.begin_item("Table", optlist);
        /* 5 表の構造と内容を作成 */
        p.set_text_pos(x_start_table, y_start_table);
        ...
p.end_item(id_table);

optlist = "Title=Insert parent=" + id_art;
id_insert = p.begin_item("P", optlist);
        /* 6 挿入構造と内容を作成 */
        p.set_text_pos(x_start_table, y_start_table);
        ...
p.end_item(id_insert);

id_artifact = p.begin_item("Artifact", "");
        /* 7+8 ヘッダ・フッタを作成 */
        p.set_text_pos(x_header, y_header);
        ...
        p.set_text_pos(x_footer, y_footer);
```

```
        ...
p.end_item(id_artifact);

/* 記事は次のページへ続く可能性あり */
...
p.end_item(id_art);
```

ページ内容を視覚順序で作成 「論理順序」式では、作成者はページ内容を必ず論理順序で構築しなければなりません。しかし物によっては、次のような視覚順序で作成するほうが簡単な場合もあるでしょう。ヘッダ、左の段組の上の部分、表、左の段組の下の部分、コラム、右の段組、フッタ。*PDF_activate_item()* を使えば、この順序を次のように実装することができます。

```
/* ページレイアウトを視覚順序で作成 */

id_header = p.begin_item("Artifact", "");
        /* 1 ヘッダを作成 */
        p.set_text_pos(x_header, y_header);
        ...
p.end_item(id_header);

id_art = p.begin_item("Art", "Title=Article");

id_sect1 = p.begin_item("Sect", "Title = {First Section}");
        /* 2 左の段組の上の部分を作成 */
        p.set_text_pos(x1_left, y1_left_top);
        ...

String optlist = "Title=Table parent=" + id_art;
id_table = p.begin_item("Table", optlist);
        /* 3 表の構造と中身を作成 */
        p.set_text_pos(x_start_table, y_start_table);
        ...
p.end_item(id_table);

/* 1番目の項のつづき */
p.activate_item(id_sect1);
        /* 4 左の段組の下の部分を作成 */
        p.set_text_pos(x1_left, y1_left_bottom);
        ...

optlist = "Title=Insert parent=" + id_art;
id_insert = p.begin_item("P", optlist);
        /* 5 挿入構造と内容を作成 */
        p.set_text_pos(x_start_table, y_start_table);
        ...
p.end_item(id_insert);

/* 1番目の項のさらにつづき */
p.activate_item(id_sect1);
        /* 6 右の段組の上の部分を作成 */
        p.set_text_pos(x1_right, y1_right_top);
        ...
p.end_item(id_sect1);

id_sect2 = p.begin_item("Sect", "Title={Second Section}");
```

```
        /* 7 右の段組の下の部分を作成 */
        p.set_text_pos(x2_right, y2_right);
        ...
        /* 2番目の項は次ページへ続く可能性あり */
p.end_item(id_sect2);

id_footer = p.begin_item("Artifact", "");
        /* 8 フッタを作成 */
        p.set_text_pos(x_footer, y_footer);
        ...
p.end_item(id_footer);

/* 記事は次のページへ続く可能性あり */
...
p.end_item(id_art);
```

構造要素をこの順序で処理すると、本文（1 段半にわたる）は表とコラムで二度中断されます。そのため、*PDF_activate_item()* を用いた本文のアクティブ化も二度行う必要があります。

これと同じ技法は内容が複数ページにわたる場合にも使えます。たとえば、ヘッダやその他コラム等をまず作成したのち、ページの本文要素をふたたびアクティブ化させるとよいでしょう。

10.5.4 Acrobat におけるタグ付き PDF の利用

この項では、私達がタグ付き PDF 出力を Adobe Acrobat 8/9/X で試験していくなかで得た知見を述べます。以下の知見は主として、Acrobat のバグや動作不安定に関するものです。これらの知見は、別記ない限り Acrobat 8・9・X にあてはまります。回避法を見出したものについては併せて記載しています。

Acrobat の折り返し機能　Acrobat はタグ付き PDF 文書を折り返すことができます。すなわち、ページ内容をその時点のウィンドウサイズに合わせて表示させることが可能です。タグ付き PDF を試験していくなかで私達は Acrobat の折り返し機能に関して以下のようないくつかの知見を得ました。

- ページ上の内容の順序は、望ましい折り返しの順序に従う必要があります。
- 記号（非 Unicode フォント）は、Acrobat 8 では折り返しをクラッシュさせることがあり、Acrobat 9 では折り返しを無効化することがあります。このため、そうしたテキストを *Figure* 要素の中に入れることを推奨します。この問題は Acrobat X では修正されています。
- BLSE は子要素と直接内容の両方を含むことができます。折り返し機能（アクセシビリティチェッカと読み上げも）を正しく動作させるには、最初の子要素の前に直接内容を入れることを推奨します。
  子の種類が混在した（すなわち、ページ内容シーケンスと非インライン構造要素の両方）構造要素は、リフロー失敗の原因となることがありますので、避けるべきです。
- 表とイラストには *BBox* オプションを与える必要があります。*BBox* は正確でなければなりません。ただし、表については左下隅だけが正確に設定されていれば充分です。*BBox* 項目を与えるかわりに、グラフィックを *P・H* などの BLSE タグの中に作成することもできます。しかし、折り返しを有効にした時にベクトルグラフィックが表示されなくなります。クライアントが *BBox* オプションを与えない（すなわち自動 *BBox* 生成に依

存する）場合は、セル枠線などのすべての表グラフィックはその表要素の外で描く必要があります。

- 表要素はその子要素として表関連要素（*TR・TD・TH・THead・TBody* など）だけを含まなければならず、それ以外の要素を一切含んではいけません。たとえば、表内で *Caption* 要素を用いることはタグ付き PDF としては正しいとしても、折り返しには問題を生じます。
- Acrobat 8・9：*Private* タグの中の内容は他の形式へ書き出されません。しかし折り返しと読み上げの対象にはなるので、*Private* タグの中のイラストは代替テキストを持つ必要があります。
- *topdown*オプションで生成したPDF文書については折り返しには問題があるようです。
- アクティブ化したアイテムが内容だけを含み、子要素を含まない場合には、折り返しは失敗することがあります。特にそのアイテムが別のページでアクティブ化された場合に顕著です。この問題は、アクティブ化したアイテムを非インライン *Span* タグでラップすることで避けることができます。
- Acrobat は、フォームフィールド（電子署名フィールドを含む）を持つページを折り返しできず、その場合は警告が表示されます。
- Acrobat 8：折り返しで問題があるたび、リフロー機能は無効化され、そのメニュー項目は無効になります。

Acrobat のアクセシビリティチェッカ　Acrobat のアクセシビリティチェッカを利用すると、スクリーンリーダのような支援技術に対するタグ付き PDF 文書の適合性を調べることができます。いくつかのヒントを挙げます：

- もっとも重要なこととして、すべてのページ内容にタグが付いている必要があります。タグ構造外の内容はアクセシブルになりませんので、Acrobat のアクセシビリティチェッカによってフラグされます。
- フォームフィールドをアクセシブルにするには、*PDF_create_field()*・*PDF_create_fieldgroup()* で *tooltip* オプションを使います。
- ページに注釈があると、Acrobat は「タブ順が構造順と整合しない可能性がある」と表示します。
- *Alt* タグは *Figure* タグについて無視されます。

Acrobat による他の形式への書き出し　タグ付き PDF を利用すると、PDF 文書を Acrobat で XML や RTF などの形式で保存するとき、結果が飛躍的に向上します。

- Acrobat 8/9：取り込んだPDFページが *Form* タグを持っている場合、*ActualText* オプションで与えたテキストは Acrobat で他の形式へ書き出されますが、*Alt* タグで与えたテキストは無視されます。しかし、読み上げ機能はどちらのオプションでも動作します。
- Acrobat X は内容を「ありのままに」抽出します：*Alt*・*ActualText* オプションは無視されますし、*Private*・*NonStruct* タグも無視されます。
- Acrobat 8/9のみ：*NonStruct* タグの内容はHTML 4.01 CSS 1.0には書き出されません（しかし HTML 3.2 書き出しには用いられます）。
- ILSE（たとえば *Code*・*Quote*・*Reference*）には代替テキストを与える必要があります。*Alt* オプションを用いた場合、読み上げ機能ではその与えたテキストが読まれますが、他の形式へ書き出されるのは実際の内容になります。*ActualText* オプションを用いた場合、その与えたテキストは読み上げと書き出しの両方に用いられます。

Acrobat の読み上げ機能　タグ付き PDF は、Acrobat の読み上げ機能を改善します。

- *Alt*・*ActualText* を与える際にはその先頭にスペースを入れると有用です。そうしておくと、読み上げ機能はテキストを直前の文から区別することができます。同じ理由で、末尾にピリオドキャラクタ「.」を入れるのも有用です。そうでないと、読み上げ機能は、前の文の最後の単語と、代替テキストの最初の単語とを、つなげて読もうとしてしまうでしょう。

11 PPS と PDFlib Block Plugin

PDFlib Personalization Server（PPS）は、可変データ処理のための、テンプレートを用いた PDF ワークフローに対応しています。ブロックという概念を用いて、取り込んだページに、外部情報源から引き出した任意量の 1 行ないし複数行のテキスト・画像・PDF グラフィックを入れ込むことができます。これを利用すれば、PDF 文書のカスタマイズを必要とするアプリケーションを容易に実装できます。たとえば：

- メールの連結
- ダイレクトメールの宛名印刷
- 納品書・請求書等の発行
- 名刺の項目内容を各人ごとに変更

PDFlib Block Plugin を使えば、ブロックを対話的に作成・編集することができ、フォームフィールド変換プラグインを使えば、既存の PDF フォームフィールドを PDFlib ブロックに変換することができます。ブロックへは、PPS を使って流し込みを行うことができます。Block Plugin には、内蔵バージョンの PPS が含まれていますので、PPS によるブロックへの流し込み結果を Acrobat 上でプレビューすることができます。

注 **ブロック処理を利用するには PDFlib Personalization Server（PPS）が必要です。PPS はすべての PDFlib 商用パッケージに含まれていますが、PPS に対するライセンスキーをご購入いただく必要があります。PDFlib や PDFlib+PDI のライセンスキーだけではご利用いただけません。PDF テンプレートに対話的にブロックを作成するには Adobe Acrobat 用 PDFlib Block Plugin が必要です。**

クックブック **可変データとブロックに関するコードサンプルが PDFlib クックブックの `blocks` カテゴリにあります。**

11.1 PDFlib Block Plugin をインストール

Block Plugin と、その姉妹製品である PDF フォームフィールド変換プラグインは、以下のバージョンの Acrobat で動作します：

- Windows 版 Acrobat 8/9/X Standard・Professional・Pro Extended
- Mac 版 Acrobat 8/9/X Professional

このプラグインは Acrobat Elements では動作せず、Adobe Reader のどのバージョンでも動作しません。

Windows 版 Acrobat 8/9/X に PDFlib Block Plugin をインストール PDFlib Block Plugin と PDF フォームフィールド変換プラグインを Acrobat にインストールするには、プラグインのファイルを Acrobat のプラグインフォルダのサブディレクトリに入れる必要があります。これは、プラグインのインストーラによって自動的に実行されますが、手作業でもできます。プラグインのファイル名は *Block.api* と *AcroFormConversion.api* です。プラグインフォルダの典型的な場所は下記のようになります：

```
C:\Program Files\Adobe\Acrobat 10.0\Acrobat\plug_ins\PDFlib Block Plugin
```

Mac 版 Acrobat 8/9/X に PDFlib Block Plugin をインストール Mac 版 Acrobat 8/9/X では、プラグインフォルダは Finder 内で直接見ることができません。プラグインのファイル

をプラグインフォルダにドラッグするのではなく、以下の手順を踏みます（Acrobat が動作していないことを確認してください）：

- ディスクイメージをダブルクリックして、プラグインファイルをフォルダへ抽出します。
- ***Adobe Acrobat*** アプリケーションアイコンを Finder 内で見つけます。通常は次のような名前のフォルダ内にあります。

```
/Applications/Adobe Acrobat 10.0
```

- この Acrobat アプリケーションアイコンをシングルクリックして、コンテキストアイコンを開き、**「パッケージの内容を表示」**を選択します。
- 現れる Finder ウィンドウで、***Contents/Plug-ins*** フォルダへ移動し、1 番目の手順でできた ***PDFlib Block Plugin*** フォルダをこのフォルダ内へ複製します。

マルチリンガルインタフェース　PDFlib Block Plugin は、ユーザーインタフェース内で複数言語に対応しています。Acrobat のアプリケーション言語に従って、Block Plugin はそのインタフェース言語を自動的に選択します。目下、英語・ドイツ語・日本語のインタフェースが利用可能です。Acrobat がこれ以外の言語モードで動作している場合には、Block Plugin は英語インタフェースを使用します。

トラブルシューティング　PDFlib Block Plugin が動作しないように見られる場合は、以下の点を確認してください：

- **「編集」→「環境設定」**（**→「一般 ...」**）**→「一般」**で **「承認されたプラグインのみを使用」**チェックボックスがオフになっていることを確認してください。Acrobat が承認済みモードで動作していると、プラグインは読み込まれません。
- Adobe Designerによって作成されたPDFフォームは、Block Pluginの適切な動作を妨げることがあります。他の Acrobat のプラグインの動作についても同様です。なぜならこうしたフォームは、Acrobat の内部セキュリティモデルと衝突するためです。ですので、Designer の静的な PDF フォームは利用せずに、動的な PDF フォームだけを Block Plugin への入力として用いることを推奨します。

11.2 PDFlib ブロックの概念あらまし

11.2.1 文書デザインとプログラムコードとの分離

PDFlib のデータブロックを利用すると、取り込んだページ上に、可変のテキストや画像やグラフィックを簡単に配置できます。単純な PDF ページと違って、データブロックを含むページは、後でサーバサイドで行われるべき処理についての情報を内部に持っています。PDFlib ブロックの概念は、以下の 2 種類の作業を完全に分離するものです：

- デザイナーはページレイアウトを作成し、可変ページ構成要素の位置を指定するとともに、その文字サイズ・色・画像縮尺といったプロパティも指定します。レイアウトは PDF 文書として作成し、その後デザイナーは、Acrobat 用 PDFlib Block Plugin を使って、可変データブロックとそのそれぞれのプロパティを指定します。
- プログラマーは、取り込まれる PDF ページ上の PDFlib ブロックに含まれる情報を、データベースのフィールドといった動的な情報と紐づけるコードを書きます。プログラマーは、ブロックの詳細については何も知らなくてよく（名前を含むのか ZIP コードを含むのか、ページ上の正確な位置、書式など）、そのため、どのようなレイアウト変更からも独立でいられます。ブロックに関連する詳細についてはすべて、ファイル内のブロックプロパティに基づいて PPS の側で処理します。

言いかえれば、プログラマーによって書かれるコードは「データ非依存」です。すなわちそれは汎用であり、ブロックのいかなる特性にも依存しません。たとえばデザイナーは、手紙の宛先を入れるブロックをページ上の別の場所へ移動させるかもしれませんし、あるいは、文字サイズを変えるかもしれません。一般的なブロック処理コードに変更を加える必要はなく、デザイナーがブロックプロパティを Acrobat プラグインで変更してラストネームのかわりにファーストネームを用いるようにしさえすれば、正しい出力が生成されます。

中間ステップとして、ブロックへの流し込みは Acrobat でプレビューできますので、開発と試験サイクルを迅速化することが可能です。ブロックプレビューには、ブロックの定義内で指定されたデフォルトデータ（文字列や画像ファイル名等）が用いられます。

11.2.2 ブロックプロパティ

ブロックの動作はブロックプロパティで制御することができます。プロパティは Block Plugin でブロックに割り当てます。

標準ブロックプロパティ　ブロックはページ上の矩形として定義され、名前・種類・その他自由なプロパティ群を割り当てられます。こうしたプロパティは後で PPS によって処理されます。名前は、ブロックを識別する任意の文字列であり、たとえば ***firstname*・*lastname*・*zipcode*** のように名づけることができます。PPS では、さまざまな種類のブロックを使うことができます：

- **テキスト行ブロック**は、1 行のテキストデータを持ちます。このデータは、PPS のテキスト行メソッドで処理されます。
- **テキストフローブロック**は、1 行ないし複数行のテキストデータを持ちます。複数行のテキストは PPS のテキストフローフォーマッタによって組版されます。複数のテキストフローブロックを連結して、前のブロックからあふれたテキストを次のブロックに入れることも可能です（302 ページ「テキストフローブロックを連結」参照）。
- **画像ブロック**は、ラスタ画像を持ちます。これは、DTP アプリケーションで TIFF や JPEG のファイルを貼り付けるのと似ています。

- ***PDF*ブロック**は、他のPDF文書のページから取り込んだ任意のPDFグラフィックを持ちます。これは、DTP アプリケーションで PDF ページを貼り付けるのと似ています。

ブロックは、その種類によって異なるさまざまな標準プロパティを持っています。プロパティは、Block Plugin で追加・変更することができます（288 ページ「11.3.2 ブロックプロパティを編集」参照）。標準ブロックプロパティの全一覧が 306 ページ「11.6 ブロックのプロパティ」にあります。たとえば、テキストブロックではテキストのフォントやサイズを指定することができ、画像ブロックや PDF ブロックでは拡縮倍率や回転を指定することができます。PPS はブロックの種類ごとに、それを処理するための専用の関数を提供しています（*PDF_fill_textblock()* 等）。こうした関数は、取り込まれた PDF ページの中でブロックを名前で検索し、そのプロパティを分析して、クライアントの与えたデータ（一行テキスト・複数行テキスト・ラスタ画像・PDF ページのいずれか）を、新しいページ上に、指定されたブロックプロパティに従って配置します。プログラマーは、ブロック流し込み関数の対応するオプションを指定することによって、ブロックプロパティをオーバライドすることもできます。

デフォルト内容に関するプロパティ　特殊なブロックプロパティを定義して、そのブロックのデフォルト内容を持たせることもできます。このデフォルト内容は、ブロック流し込み関数に可変データが与えられなかったときや、あるいはブロック内容が次回の印刷実行時には変わりうるけれども現時点では不変であるようなときに、そのブロックに配置されます。

デフォルトプロパティは、Block Plugin のプレビュー機能でも用いられます（296 ページ「11.4 Acrobat でブロックをプレビュー」参照）。

カスタムブロックプロパティ　標準ブロックプロパティを利用することにより、可変データ処理アプリケーションを手軽に実装することができますが、こうしたプロパティは、PPS の内部的に既知で自動処理可能なものに限られてしまいます。より高い柔軟性を与えるために、デザイナーは、カスタムプロパティをブロックに割り当てることもできます。カスタムプロパティを利用すれば、ブロックの概念を拡張して、より高度な可変データ処理アプリケーションの要請に応えることが可能です。

カスタムプロパティに関してはいかなる規則も存在しません。なぜなら PPS はカスタムプロパティに対してはいかなる処理も行わないからです。PPS はただ、カスタムプロパティをクライアントが利用できるようにするだけです。クライアントコードは、カスタムプロパティを取得し、適切に処理することができます。ブロックのカスタムプロパティに基づいて、アプリケーションがレイアウト関連やデータ抽出関連の決定を行えるようにすることも可能です。たとえば、科学アプリケーションのためのカスタムプロパティであれば、数値出力の桁数を指定することもできるでしょうし、あるいは、データベースのフィールド名をカスタムブロックプロパティとして定義しておいて、そのブロックのためのデータ抽出に用いることもできるでしょう。

11.2.3 PDF のフォームフィールドを利用しない理由は？

経験ある Acrobat ユーザーならば、なぜ我々は新たにブロックという概念を導入したのか、どうして PDF にすでにあるフォームフィールドのしくみを活用しないのか、疑問を抱かれるかもしれません。そもそもの違いは、PDF のフォームフィールドは対話的に記入されることを主眼として作られているのに対して、PDFlib のブロックは自動的に流し込まれることを目的としている点です。対話的記入と自動流し込みの両方を必要とするアプリケーションの場合であれば、フォームフィールド変換プラグインを用いて、PDF フォームと

PDFlib ブロックを併用することも可能です（291 ページ「11.3.4 PDF フォームフィールドを PDFlib ブロックに変換」参照）。

両概念の間には類似点も多くありますが、PDFlib ブロックには PDF フォームフィールドと比較して表 11.1 に示すようないくつかの利点があります。

表 11.1 PDF フォームフィールドと PDFlib ブロックの比較

機能	PDF フォームフィールド	PDFlib ブロック
設計の趣旨	対話的利用	自動流し込み
文字組版機能（フォント指定・文字サイズ指定よりも高度な）	―	カーニング・単語間隔・文字間隔・下線 / 上線 / 取り消し線
OpenType レイアウト機能	―	何ダースもの OpenType レイアウト機能（合字・スウォッシュ文字・オールドスタイル数字等）
複雑用字系への対応	制約あり	シェーピング・双方向テキスト（アラビア文字・デーヴァナーガリー等）
フォント制御	フォント埋め込み	フォント埋め込み・サブセット化・エンコーディング
テキスト組版制御	左・中央・右揃え	左・中央・右・両端揃え。各種組版アルゴリズム・制御。インラインオプションを用いてテキストの見映えを制御可能
テキストの途中でフォントその他のテキスト属性を変えられる	―	○
追加結果が PDF のページ記述に組み込まれる	―	○
ユーザーが追加フィールドの内容を編集可能	○	×
プロパティの拡張セット	―	○（カスタムブロックプロパティ）
画像ファイルを流し込める	―	BMP・CCITT・GIF・PNG・JPEG・JBIG2・JPEG 2000・TIFF
カラー対応	RGB	グレースケール・RGB・CMYK・Lab・スポットカラー（HKS・Pantone スポットカラーが Block Plugin に内蔵）
PDF/X・PDF/A	PDF/X：× PDF/A：制約あり	○（ブロックコンテナも追加結果も）
グラフィックやテキストのプロパティを流し込み時に上書き可能	―	○
透過内容	―	○
テキストブロック群を連結可能	―	○

11.3 PDFlib Block Plugin でブロックを編集

11.3.1 ブロックを作成

ブロックツールをアクティブにする　Block Plugin を使った PDFlib ブロックの作り方は、Acrobat のフォームツールと同様です。ページ上のすべてのブロックは、ブロックツールがアクティブな時に表示されます。Acrobat の他のツールが選択されるとブロックは見えなくなりますが、なくなったわけではありません。ブロックツールをアクティブにするには、以下のいずれかの操作を行います：

- **「ツール」→「プラグイン 高度な編集」**ペーン（Acrobat X）または**「高度な編集」**ツールバー（Acrobat 8/9）でブロックアイコン  をクリック。Acrobat でこのツールバーが表示されていないときは、**「表示」→「ツール」→「プラグイン 高度な編集」**（Acrobat X）または**「表示」→「ツールバー」→「高度な編集」**（Acrobat 8/9）を選択すれば表示させることができます。
- メニュー項目**「*PDFlib* ブロック」→「*PDFlib* ブロックツール」**を選択。

ブロックの作成と変更　ブロックツールをアクティブにしたら、十字ポインタをドラッグすれば、ページ上の希望の位置に希望の大きさのブロックを作成することができます。ブロックは必ず矩形で、その辺はページの辺と平行になります（ブロックの内容をページの辺と平行でなくするには *rotate* プロパティを用います）。ブロックの矩形をドラッグし終わると、**「*PDFlib* ブロックプロパティ」**ダイアログが現れるので、ブロックのさまざまなプロパティを編集することができます（288 ページ「11.3.2 ブロックプロパティを編集」参照）。ブロックツールはブロックの名前を自動生成しますが、この名前はプロパティダイアログで変更することもできます。ブロック名はページ内では一意である必要がありますが、別のページでは同じ名前も使えます。

ダイアログの上端では、ブロックの種類を**「*Textline*」**（テキスト行）・**「*Textflow*」**（テキストフロー）・**「*Image*」**（画像）・**「*PDF*」**のいずれかに変更できます。ブロックの種類ごとに異なる色が用いられています（図 11.1 参照）。タブは、どのブロックの種類を選択しているかに応じて、一度に 1 つだけがアクティブになっています。**「*PDFlib* ブロックプロパティ」**ダイアログは、ブロックの種類に応じて、プロパティを階層的にいくつかのグループやサブグループにまとめて表示します。

注　PDF にブロックを追加したり、既存のブロックに変更を加えたりした後は、Acrobat の「名前を付けて保存 ...」コマンドを使うほうがファイルサイズが小さくなります（「上書き保存」よりも）。

注　Acrobat 用 Enfocus PitStop プラグインを使って、PDFlib ブロックを含んだ文書を編集する際、「This document contains PieceInfo from PDFlib. Press OK to continue editing or Cancel to abort.」というメッセージが表示されることがあります。このメッセージは気にしなくてかまいません。このような場合、OK をクリックしても安全です。

ブロックを選択　コピー・移動・削除・プロパティ編集といったいくつかのブロック操作は、選択した 1 個ないし複数のブロックに対して動作します。ブロックツールを用いてブロックを選択するには、以下のように操作します：

- 1 個のブロックを選択するには、単にそれをシングルクリックします。
- 複数のブロックを選択するには、Shift キーを押しながら 2 個目以降のブロックを選択します。

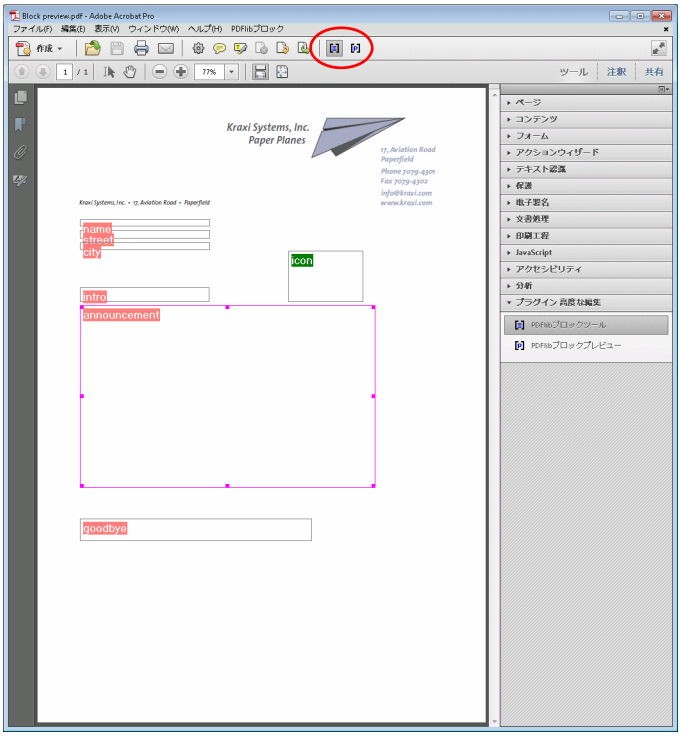


図 11.1
各種ブロックの表示

- ページ上のすべてのブロックを選択するには、Ctrl+A (Windows の場合) か Cmd+A (Mac の場合) を押すか、または**「編集」→「すべて選択」**を用います。

コンテキストメニュー　ブロックを選択（複数可）している時には、コンテキストメニューを開けば、ブロック関連のいろいろな機能（**「*PDFlib* ブロック」**メニューから利用できる諸機能と同じ）をすばやく実行することができます。コンテキストメニューを開くには、選択したブロック（複数可）を、Windows の場合はマウスの右ボタンでクリックし、Mac の場合は Ctrl+ クリックします。たとえば、ブロックを削除するには、それをブロックツールで選択したのち、***Delete*** キーを押してもよいですし、あるいはコンテキストメニューで**「編集」→「削除」**を用いることもできます。

ページ上でブロックのない領域を右クリック（Mac では Ctrl+ クリック）すると、コンテキストメニューの中身は、**「プレビューの生成」**と**「プレビューの設定 ...」**という項目になります。

ブロックの大きさと位置　ブロックツールを使うと、選択したブロック（複数可）を別の位置へ動かすことも可能です。Shift キーを押しながらブロックをドラッグすると、水平方向か垂直方向にだけ動きます。これはブロックを正確に整列させたいときに便利でしょう。ポインタがブロックの角の近くにある時は、ポインタは矢印に変わり、ブロックの大きさを変えることができます。複数のブロックの位置や大きさを調整したいときは、複数のブロックを選択して、**「*PDFlib* ブロック」**メニューかコンテキストメニューから**「整列」・「中央揃え」・「均等配置」・「サイズ」**のいずれかのコマンドを選択します。ブロック（複数可）の位置は、矢印キーを使って小刻みに変えることもできます。

図 11.2
ブロックプロパティダイアログ

あるいは、ブロックの座標を数値でプロパティダイアログに入力することもできます。座標系の原点はページの左上隅です。座標は、その時点で Acrobat で選択されている単位で表示されます：

- Acrobat 8/9/X で表示単位を変えるには、**「編集」→「環境設定」**（→**「一般 ...」**）→**「単位とガイド」**を選択し、インチ・センチメートル・パイカ・ポイント・ミリのいずれかを選びます。あるいは Acrobat 8 の場合は、**「表示」→「ナビゲーションパネル」→「情報」**を選択し、**「オプション」**メニューから単位を選ぶこともできます。
- カーソルの座標を表示するには、**「表示」→「表示切り替え」→「カーソル座標」**（Acrobat X）または**「表示」→「カーソル座標」**（Acrobat 9）または**「表示」→「ナビゲーションパネル」→「情報」**（Acrobat 8）を選択します。

ただしここで選択されている単位は ***Rect*** プロパティに対してのみ効力を持ち、それ以外の数値プロパティ（***fontsize*** 等）に対しては一切効力を持ちません。

グリッドを用いてブロックを位置合わせ　Acrobat のグリッド機能を活用して、ブロックの位置合わせやサイズ合わせを正確に行うこともできます：

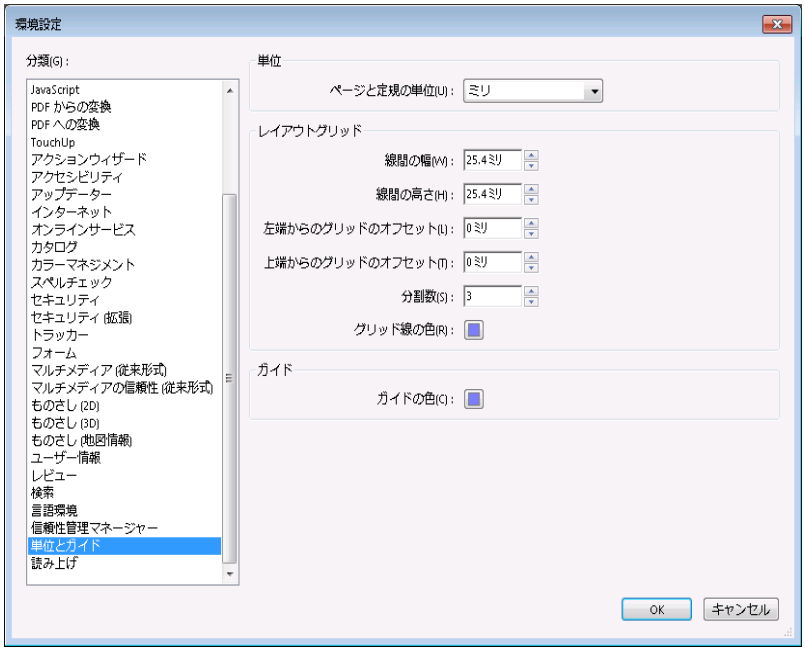


図 11.3
Acrobat の
グリッド設定

- グリッドを表示：**「表示」→「表示切り替え」→「定規とグリッド」→「グリッド」**（Acrobat X）または**「表示」→「グリッド」**（Acrobat 8/9）。
- グリッドスナップを有効に：**「表示」→「表示切り替え」→「定規とグリッド」→「グリッドにスナップ」**（Acrobat X）または**「表示」→「グリッドにスナップ」**（Acrobat 8/9）。
- グリッドを変更（図 11.3 参照）：**「編集」→「環境設定」**（→「一般 ...」）**→「単位とガイド」**を選択します。そこでグリッド線の間隔や位置や色を変えることができます。

「グリッドにスナップ」を有効にしていると、ブロックの大きさと位置は、設定したグリッドに揃います。**「グリッドにスナップ」**は、新規作成したブロックにも効きますし、既存のブロックをブロックツールで移動したり大きさを変えたりするときにも効きます。

画像やグラフィックを選択してブロックを作成　手動でブロックの矩形をドラッグするのではなく、既存のページ内容を使ってブロックの大きさを定義することもできます。まず、メニュー項目**「*PDFlib* ブロック」→「オブジェクトをクリックでブロックを定義」**を有効にします。これで、ブロックツールを使って、ページ上の画像をクリックして、その画像と同じ位置に同じ大きさのブロックを作成することができます。それ以外のグラフィックオブジェクトをクリックすることもでき、その場合、ブロックツールはそのグラフィック（たとえばロゴ）全体を選択しようとします。この**「オブジェクトクリック」**機能は、ブロック定義作業を補助するために設けてあるものです。できたブロックの位置や大きさを変更したければ、後から何ら制約なく行うことができます。ブロックは、画像やグラフィックに固定されてしまうのではなく、ただそれを位置や大きさの決定の補助として用いるだけです。

この「オブジェクトクリック」機能は、どのベクトルグラフィックや画像がページ上で論理的要素を形づくっているかを認識しようとします。いずれかのページ内容がクリックされると、その対象が白かったり非常に大きかったりしない限り、その外接枠（対象を囲む矩形）が選択されます。その次の段階として、この検知された矩形に一部入り込んでいる他の物が選択領域に追加され、これが繰り返されます。そうして最終的にできた領域

に基づいてブロックの矩形が生成されるのです。結局のところ「オブジェクトクリック」機能は、個々の線ではなくグラフィック全体を選択しようとすることになります。

フォントプロパティの自動検出　Block Plugin は、テキスト行またはテキストフローブロックの置かれた位置の背景にあるフォントを分析することができ、ブロックの以下のプロパティを自動的に書き込むことができます：

`fontname`・`fontsize`・`fillcolor`・`charspacing`・`horizscaling`・`wordspacing`・`textrendering`・`textrise`

フォントプロパティの自動検出は望ましくない結果をもたらすこともあるので、背景を無視させたい場合は、**「*PDFlib* ブロック」→「背景フォント・色の検出」**を用いて機能の有効・無効を切り替えることができます。デフォルトではこの機能は無効になっています。

ブロックをロック　ブロックは、うっかり移動したり大きさを変えたり削除したりされないよう、ロックして保護することができます。ブロックツールがアクティブな状態で、ブロックを選択し、そのコンテキストメニューから**「ロック」**を選びます。ブロックがロックされていると、移動させることも大きさを変えることも削除することもできず、そのプロパティダイアログを編集することもできません。

11.3.2 ブロックプロパティを編集

新規ブロックを作成した時や、既存ブロックをダブルクリックした時や、ブロックのコンテキストメニューから**「プロパティ」**を選択した時には、プロパティダイアログが現れて、その選択したブロックに関するすべての設定を編集することができます（図 11.2 参照）。306 ページ「11.6 ブロックのプロパティ」で詳述するように、プロパティにはブロックの種類に応じて、いくつかのグループがあります。

「適用」ボタンは、ダイアログ内の 1 個ないし複数のプロパティを変更した時にのみ有効になります。**「適用」**ボタンは、ロックされたブロックについては無効となります。

注 ブロックの種類やプロパティの設定によっては、表示されないプロパティもあります。たとえば、タブ設定を編集するためのプロパティサブグループ「`hortabmethod=ruler` におけるルーラタブ」は、テキストフローグループの `hortabmethod` プロパティが `ruler` に設定されているときにのみ表示されます。

プロパティの値を変更するには、入力したい数や文字列をそのプロパティの入力領域に入力するか（例：***linewidth***）、ドロップダウンリストから値を選ぶか（例：***fitmethod***・***orientate***）、またはダイアログの右側にある**「...」**ボタンをクリックしてフォントや色の値やファイル名を選択します（例：***backgroundcolor***・***defaultimage***）。***fontname*** プロパティの場合は、システムにインストールされているフォントの一覧から選ぶこともできますし、カスタムのフォント名を打ち込むこともできます。フォント名を入力した方式にかかわらず、そのフォントは、PPS によってブロックへ内容が流し込まれるシステム上において利用可能になっていることが必要です。

変更されたプロパティは、ブロックプロパティダイアログ内で太字で表示されます。ブロックのいずれかのプロパティが変更されているときには、その表示されているブロック名の後に接尾辞 ***(*)*** が付記されます。プロパティの編集が済んだら、**「適用」**ボタンをクリックしてブロックを更新します。定義したプロパティは、PDF ファイル内にブロック定義の一部として格納されます。

重なったブロック　重なりあうブロック群は選択しづらいことがあります。その領域をクリックすると必ず最前面のブロックが選ばれてしまうからです。このような場合には、コンテキストメニューの**「ブロックの選択」**項目を用いれば、ブロックのうちのいずれか1個を名前で選択することができます。1個のブロックを選んだ直後にその領域で行う操作は、その選択した1個のブロックに対してのみ効力を持ち、他のブロックに対しては効力を持ちません。たとえば *Enter* を押せば、選択したブロックのプロパティを編集できます。この方法を利用すれば、ブロックの上に他のブロックが部分的ないし完全にかぶさっていても、簡単にそのブロックのプロパティを編集することができます。

ブロックプロパティの値を繰り返し使用・リセット　キー入力やクリックの量をいくらか軽減できるよう、ブロックツールは、直前のブロックのプロパティダイアログで入力されたプロパティ値を記憶しています。こうした値は、新規ブロックの作成時に再利用されます。もちろんその値は、いつ別の値で上書きしてもかまいません。

プロパティのダイアログで**「全てリセット」**ボタンを押すと、多くのブロックプロパティがそれぞれのデフォルトにリセットされます。以下の項目は変更されません：

- *Name*・*Type*・*Rect*・*Description* プロパティ
- すべてのカスタムプロパティ

注 標準プロパティのデフォルト値は、プレビュー生成時のプレースホルダデータを保持する defaulttext・defaultimage・defaultpdf プロパティと混同しないようにする必要があります（296 ページ「ブロックのデフォルト内容」参照）。

複数のブロックを一度に編集　複数のブロックのプロパティを一度に編集すれば、大いに時間が節約できます。複数のブロックを選択するには以下のように操作します：

- メニュー項目**「*PDFlib* ブロック」→「*PDFlib* ブロックツール」**を選択してブロックツールをアクティブにします。
- 1個目のブロックをクリックして選択します。最初に選択したこのブロックがマスターブロックになります。他のブロック（複数可）を Shift+ クリックして、選択ブロック群に加えます。あるいは、**「編集」→「すべてを選択」**をクリックして、現在のページ上のすべてのブロックを選択することもできます。
- これらのブロックのうちいずれか1個をダブルクリックすると、ブロックプロパティダイアログが開きます。この時ダブルクリックしたブロックは、新たにマスターブロックになります。
- あるいは、1　個のブロックをクリックしてマスターブロックとして指定したうえで、*Enter* キーを押してブロックプロパティダイアログを開くこともできます。

プロパティダイアログには、選択されているすべてのブロックに適用されるプロパティの部分集合だけが表示されます。ダイアログには、マスターブロックから採られたプロパティ値が表示されます。ダイアログを**「適用」**で閉じると、その時点の内容が、選択されているブロックすべてに複製されます。すなわち、マスターブロックの値がそのまま、あるいはダイアログで手変更を加えていればそれが適用されます。この動作を利用して、ある1個のブロックのブロックプロパティを、他の1個ないし複数のブロックへ複製することも可能です。

以下の標準プロパティは共用できません。すなわち、これらは複数のブロックに対して一度に編集することはできません：

`Name`・`Description`・`Subtype`・`Type`・`Rect`・`Status`

カスタムプロパティも、ブロック間で共用することはできません。

11.3.3 ページ間・文書間でブロックをコピー

Block Plugin は、現在のページの中で、または現在の文書の中で、あるいは他の文書へと、ブロックを移動したりコピーしたりするための手段をいくつか提供しています：

- ブロックをマウスでドラッグして移動・コピー、または他のページや開いている文書へブロックを貼り付け
- ブロックを、通常のコピー / 貼り付け操作を用いて、同一文書内のページ（複数可）へ複製
- ブロックを、新しいファイル（ページが空白の）や、既存の文書（既存のページにブロックを適用）へ書き出し
- 他の文書からブロックを取り込み

ブロックの定義を維持したままページの内容を更新したい場合には、ブロックを保ったまま背景のページ（複数可）を置換することができます。Acrobat の**「文書」→「ページの置換 ...」**を用います。

ブロックを移動・複製　ブロックの位置を変えるには、ブロック（複数可）を選択して、新しい位置へドラッグします。ブロックの複製を作成するには、Ctrl キー（Windows の場合）か Alt キー（Mac の場合）を押しながら同様にドラッグします。キーを押している間は、マウスカーソルが変わります。複製されたブロックは元のブロックと同じプロパティを持ちますが、ただし名前と位置だけは自動的に変更されます。

また、コピー／貼り付けを使って、ブロックを、同一ページ内の他の場所へ、または同一文書内の他のページへ、あるいは Acrobat でその時点で開いている他の文書へ複製することもできます：

- ブロックツールをアクティブにしてから、複製したいブロック群を選択します。
- Ctrl+C（Windows の場合）か Cmd+C（Mac の場合）を、または**「編集」→「コピー」**を使って、選択したブロックをクリップボードへコピーします。
- 複製先ページへ移動します（必要なら）。
- Ctrl+V（Windows の場合）か Cmd+V（Mac の場合）を、または**「編集」→「貼り付け」**を使って、クリップボードからブロックを現在のページへ貼り付けます。

ブロックを他のページ群へ複製　ブロック（複数可）の複製を、現在の文書の中の任意の数のページ上に一度に作成することもできます：

- ブロックツールをアクティブにしてから、複製したいブロック群を選択します。
- **「*PDFlib* ブロック」**メニューかコンテキストメニューから**「取り込みと書き出し」→「複製 ...」**を選びます。
- どのブロックを複製するかを選び（**「選択されているブロック」**または**「このページ上の全ブロック」**）、この選んだブロック群を複製したい複製先ページの範囲を選びます。

ブロックを書き出し・取り込み　ブロックの書き出し／取り込み機能を使うと、ある 1 つのページ上のブロックの定義や、ある文書内のすべてのブロックの定義を、複数の PDF ファイル間で共用することが可能です。これは、既存のブロック定義を維持したままページ内容を更新したいときに便利です。ブロック定義を別ファイルとして書き出すには以下のように操作します：

- ブロックツールをアクティブにしてから、書き出したいブロック群を選択します。
- **「*PDFlib* ブロック」**メニューかコンテキストメニューから**「取り込みと書き出し」→「書き出し ...」**を選択します。ページ範囲と、ブロック定義を持たせたい新規 PDF ファイル名を入力します。

ブロック定義を取り込むには「*PDFlib* ブロック」→「取り込みと書き出し」→「取り込み ...」を選択します。ブロック取り込みの際には、取り込んだブロックを文書内の全ページに適用するか、それともあるページ範囲にのみ適用するかを選ぶことができます。複数のページを選択した場合、ブロック定義は変更されずに各ページへコピーされます。取り込むブロック定義よりも取り込み先範囲のほうがページ数が多い場合には、**「テンプレートを繰り返す」**チェックボックスを用いることもできます。これをチェックすると、取り込みファイル内のブロックのシーケンスが、現在の文書の中で、文書の終わりに達するまで繰り返されます。

書き出しでブロックを他の文書へ複製　ブロックを書き出す際には、そのブロック群を他の文書内のページ群へ直接適用することもできます。結果として、ある文書から別の文書へブロック群を転写することが可能です。そのためには、ブロックの書き出し先として既存の文書を選びます。**「既存のブロックを削除」**チェックボックスをチェックすると、書き出し先の文書にブロックが存在していてもすべて削除され、その後に、新しいブロック群がその文書へコピーされます。

11.3.4 PDF フォームフィールドを PDFlib ブロックに変換

PDFlib ブロックを手動で作成するのではなく、PDF フォームフィールドをブロックへ自動変換させることもできます。これは、複雑な PDF フォームがあって PPS で自動流し込みさせたい場合や、既存の大量の PDF フォームを自動流し込みできるように変換したい場合などに特に便利です。1 つのページ上のすべてのフォームフィールドを PDFlib ブロックに変換するには、**「*PDFlib* ブロック」→「フォームフィールドの変換」→「現在のページ」**を選択します。文書内のすべてのフォームフィールドを変換したい場合は**「全ページ」**を選びます。選択したフォームフィールドだけを変換するには（1 個または複数のフォームフィールドを選択するには、Acrobat のフォームツールかオブジェクト選択ツールを選びます）、**「選択されているフォームフィールド」**を選択します。

フォームフィールド変換の詳細　自動フォームフィールド変換では、**「*PDFlib* ブロック」→「フォームフィールドの変換」→「変換オプション ...」**ダイアログで選択されている種類のフォームフィールドが、テキスト行ブロックかテキストフローブロックに変換されます。デフォルトでは、すべての種類のフォームフィールドが変換されます。変換されたフィールドの属性は、表 11.3 に従って各ブロックプロパティへ変換されます。

同名の複数のフォームフィールド　同じページ上にある複数のフォームフィールドは、同じ名前を持つことが許されていますが、それに対してブロック名はページ上で一意でなければなりません。このため、フォームフィールドがブロックに変換される際には、生成されるブロックの名前に数の接尾辞が付加され、一意なブロック名が作成されます（291 ページ「フォームフィールドの対応ブロックを見つける」も参照）。

なお、Acrobat の内部的な問題のため、複数のフォームフィールドが同じ名前を持つ場合のフィールドの属性は正しく報告されません。複数のフィールドが同じ名前を持っていて、しかし属性が異なっている場合には、生成されるブロックにはこうした属性の違いは反映されません。変換処理はこの場合、警告メッセージを表示して、関係するフォームフィールド群の名前を示します。この場合は、生成されたブロックのプロパティを注意深く調べる必要があります。

フォームフィールドの対応ブロックを見つける　同じ名前のフィールドが複数あった場合（ラジオボタン等）、変換されたフォームフィールドの名前は変更されてしまっていますから、どのフォームフィールドがどのブロックになったのかを正しく知ることは困難で

す。このことは特に、FDF または XFDF ファイルを用いてブロックへの流し込みを行い、その結果をフォームへの記入と同じにしたい場合に問題となります。

この問題を解決するため、AcroFormConversion プラグインは、元のフォームフィールドに関する情報を、それに対応するブロックを生成する際に、カスタムプロパティ群として記録します。表 11.2 に、ブロックを正しく同定するために利用できるカスタムプロパティの一覧を示します。プロパティの型はすべて**文字列**です。

表 11.2 ブロックの元のフォームフィールドを同定するためのカスタムプロパティ一覧

カスタムプロパティ	意味
PDFlib:field:name	フォームフィールドの完全修飾名。
PDFlib:field:pagenumber	元の文書でフォームフィールドが存在していたページの番号（文字列で）。
PDFlib:field:type	フォームフィールドの種類。pushbutton・checkbox・radiobutton・listbox・combobox・textfield・signature のうちのいずれか。
PDFlib:field:value	（type=checkbox の場合のみ）フォームフィールドの出力値。

ブロックを対応するフォームフィールドへバインド　PDFlib フォームフィールドと生成 PDFlib ブロックを同期させるために、生成されたブロックは、対応するフォームフィールドにバインドさせておくことができます。言い換えれば、ブロックツールが内部的にフォームフィールドとブロックとの紐付けを保持するということです。変換処理が再実行される際、バインドされたブロックは、対応する PDFlib フォームフィールドの属性を反映して更新されます。ブロックがバインドされていると、作業の二度手間が省けて便利です。フォームが対話的利用のために更新された時には、対応するブロックも自動的に更新されるからです。

ブロック生成後に変換元フォームフィールドを残したくない場合は、**「*PDFlib* ブロック」→「フォームフィールドの変換」→「変換オプション ...」**ダイアログの**「変換されたフォームフィールドを削除」**オプションを選びます。このオプションを選ぶと、フォームフィールドは変換処理後に完全に削除されます。削除されたフィールドに関連づけられていたアクション（JavaScript など）も、すべて文書から削除されます。

バッチ変換　フォームフィールドを PDFlib ブロックに変換したい PDF 文書が多数ある場合には、バッチ変換機能を利用して、任意の数の文書を自動処理することもできます。**「*PDFlib* ブロック」→「フォームフィールドの変換」→「バッチ変換 ...」**を選択すれば、バッチ処理ダイアログが現れます。

- 入力ファイルは個別に選ぶこともできますし、1 個のフォルダの中身をすべてまとめて処理させることもできます。
- 出力ファイルは、入力ファイルと同じフォルダへ書き出すこともできますし、別のフォルダへ書き出すこともできます。出力ファイルには、入力ファイルと区別するためにプレフィックスを追加することもできます。
- 大量の文書を処理する際には、ログファイルを指定することを推奨します。変換後、ログファイルには、処理されたすべてのファイルの一覧と、それぞれの変換結果に関する詳細が書き込まれており、エラーが起きた場合にはエラーメッセージも書き込まれます。

変換処理の間、変換される PDF 文書は Acrobat で表示されますが、変換が完了するまで、Acrobat のメニュー機能やツールは一切使用できません。

表 11.3 PDF フォームフィールドから PDFlib ブロックへの変換

以下の PDF フォームフィールド属性は …	… 以下の PDFlib ブロックプロパティに変換される
全フィールド	
位置	Rect

表 11.3 PDF フォームフィールドから PDFlib ブロックへの変換

以下の PDF フォームフィールド属性は …	… 以下の PDFlib ブロックプロパティに変換される
名前	Name
ツールヒント	Description
一般→一般プロパティ→表示と印刷	Status： 表示 =active 非表示 =ignore 表示 / 印刷しない =ignore 非表示 / 印刷する =active
一版→一般プロパティ→向き	orientate：0=north、90=west、180=south、270=east
表示方法→テキスト→フォント	fontname
表示方法→テキスト→フォントサイズ	fontsize：文字サイズ auto は、ブロックの高さの 3 分の 2 の固定文字サイズに変換され、fitmethod は auto に設定されます。複数行のブロック／フィールドにおいては、この組み合わせでは結果として自動的に適切な文字サイズになるので、ブロックの高さの 3 分の 2 という初期値よりも小さくなることがあります。
表示方法→テキスト→テキストの色	strokecolor・fillcolor
表示方法→境界線と色→境界線の色	bordercolor
表示方法→境界線と色→塗りつぶしの色	backgroundcolor
表示方法→境界線と色→幅	linewidth：細 =1、標準 =2、太 =3
テキストフィールド	
オプション→整列	position： 左揃え ={left center} 中央 ={center center} 右揃え ={right center}
オプション→デフォルト	defaulttext
オプション→複数行	チェックありならテキストフローブロックが生成 チェックなしならテキスト行ブロックが生成
ラジオボタン・チェックボックス	
「デフォルトでチェック」がオンの場合： オプション→チェックボックススタイル、 オプション→ボタンスタイル	defaulttext： チェックマーク =4 円形 =l 十字形 =8 ひし形 =u 四角形 =n 星形 =H （文字は ZapfDingbats フォントにおける各記号を表します）
リストボックス・コンボボックス	
オプション→選択されている（デフォルト）項目	defaulttext
ボタン	
オプション→アイコンとラベル→ラベル	defaulttext

11.3.5 Block Plugin のユーザーインタフェースを XML でカスタマイズ

Block Plugin のユーザーインタフェースの以下の点は、XML 設定ファイルで制御することができます。この XML ファイルは、Block Plugin ディレクトリに置く必要があります。デフォルト設定ファイル *default.PPSoptions* は起動時に読み込まれます。PDFlib Block Plugin とともにインストールされるこのデフォルト設定ファイルを参照してください：

- 要素 */Block_Plugin/MainDialog/CloseOnApply* は、ブロックプロパティダイアログの「**適用したらダイアログを閉じる**」チェックボックスの初期状態を制御します。このチェックボックスは、ブロックプロパティダイアログを、ブロックを作成した後も、またはブロックプロパティを変更した後も開いたままにしておくかどうかを決定します。
- 要素 */Block_Plugin/FontDialog/ShowBaseFonts* は、ブロックプロパティダイアログのフォント一覧（「**書式**」プロパティグループの *fontname* プロパティ）に、システムにインストールされているフォント群に加えて、ベース 14 フォントも表示するかどうかを制御します。
- 要素 */Block_Plugin/Command/ControlByClick* は、メニュー項目「*PDFlib* **ブロック**」→「**オブジェクトをクリックでブロックを定義**」の初期状態を制御します。
- 要素 */Block_Plugin/Command/DetectFonts* は、メニュー項目「*PDFlib* **ブロック**」→「**背景フォント・色の検出**」の初期状態を制御します。

11.4 Acrobat でブロックをプレビュー

注 PDFlib ディストリビューションの中の block_template.pdf 文書で、プレビュー機能を試すことができます。必要なリソース（フォント・画像等）もディストリビューションに含まれています。

PDFlib ブロックは PPS によって処理されます。PPS を用いることで、ブロックへの流し込み処理について、そのデータ源（データベース内のテキスト、ディスク上の画像ファイル等）や、生成される文書の書式・対話的性質をカスタマイズすることができます。この処理について詳しくは 301 ページ「11.5 PPS でブロックへ流し込み」で解説します。

しかし Block Plugin には内蔵バージョンの PPS が含まれており、これを用いて、プログラミングを一切必要とせずに Acrobat 上で、流し込まれたブロックのプレビューバージョンを生成することができます。このプレビュー機能は、カスタムプログラミングと同等の柔軟性を提供することはできませんが、ブロックへの流し込み結果を手軽に眺めるには適しています。ブロックプレビューを活用すれば、ブロックの位置や大きさを改善したり、ブロックのプロパティ（フォント名・文字サイズ等）をチェックしたりすることができます。プレビューの表示結果に満足するまで、ブロックを変更し、プレビューを新たに生成することができます。プレビューは、現在のページについても、文書全体についても生成できます。

プレビューはつねに、新しい PDF 文書として表示されます。元の文書（ブロックを含んでいる）は、プレビューを生成しても変更を受けません。プレビュー文書は、必要に応じて保存することも捨てることもできます。元のブロックコンテナ文書は、プレビューによって影響を受けません。

ブロックのデフォルト内容　プラグインでは、サーバサイドのデータ源（データベース等）からブロックのテキスト・画像・PDF 内容を入手することは望むべくもありませんので、プレビュー機能ではつねに、ブロックのデフォルト内容、すなわち *defaulttext*・*defaultimage*・*defaultpdf* プロパティで指定されているデータが用いられます。通常、PPS で使われる実際のブロック内容を代表するようなサンプルデータセットが、デフォルトデータとして用いられます。デフォルト内容を持たないブロックは、プレビュー生成時には無視されます。*Status=ignoredefault* のブロックについても同様です。

新規ブロックでは、デフォルトプロパティは空です。プレビュー機能を使う前には、**「デフォルト内容」**プロパティグループの *defaulttext*・*defaultimage*・*defaultpdf* プロパティ（ブロックの種類による）に書き込む必要があります。

注 デフォルトテキストを記号フォントで入力する方法はややトリッキーです。詳しくは 299 ページ「デフォルトテキストに記号フォントを用いる」を参照してください。

ブロックプレビューを生成　ブロックプレビューを生成するには、以下のいずれかの方法を用います：

- メニュー項目**「*PDFlib* ブロック」→「プレビュー」→「プレビューの生成」**で。
- **「ツール」→「プラグイン 高度な編集」**ペーン（Acrobat X）または**「高度な編集」**ツールバー（Acrobat 8/9）で PDFlib ブロックプレビューアイコン  をクリック。Acrobat でこのツールバーが表示されていないときは、**「表示」→「ツール」→「プラグイン 高度な編集」**（Acrobat X）または**「表示」→「ツールバー」→「高度な編集」**（Acrobat 8/9）を選択すれば表示させることができます。

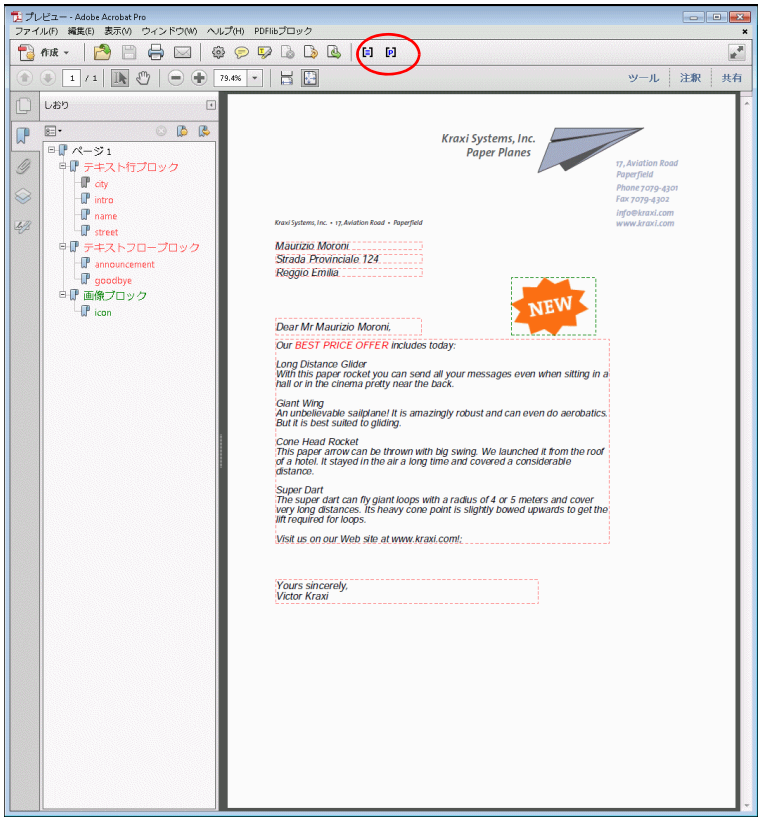


図 11.4
図 11.1 で示したコンテナ文書に対するプレビュー PDF。ブロック情報レイヤー群と注釈群を含んでいます

- ブロックツールがアクティブのときは、どのブロックもない所を右クリックすれば、コンテキストメニューに項目**「プレビューの生成」**と**「プレビュー設定 ...」**が現れます。

プレビューは、ディスク上の PDF ファイルに基づいて生成されます。Acrobat 上で変更を行なっていた場合、ブロック PDF を**「ファイル」→「上書き保存」**または**「ファイル」→「名前を付けて保存 ...」**を用いてディスクに保存してはじめて、その変更はプレビューに反映されます。変更を受けたブロックは、ブロックの名前の後にアスタリスクが付いていることで識別できます。プレビュー機能を設定して、プレビュー生成前にブロック PDF が自動保存されるようにすることもできます。そうすれば、対話的に行なった変更が確実に、ただちにプラグイン内で反映されます。

プレビューを設定　ブロックプレビューの生成と、その背後の PPS の動作については、いくつかの性質を、**「*PDFlib* ブロック」→「プレビュー」→「プレビュー設定 ...」**で設定することができます：

- 現在のページをプレビューするか、それとも文書全体をプレビューするか。
- 生成されるプレビュー文書の出力ディレクトリ。
- ブロック PDF をプレビュー生成前に自動保存。
- ブロック情報レイヤーと注釈を追加。
- PDF/A-1b または PDF/X 状態を複製。これらの規格では、レイヤーと注釈の使用は制限されていますので、**「ブロック情報レイヤーと注釈を追加」**オプションとこのオプションは同時には使えません。

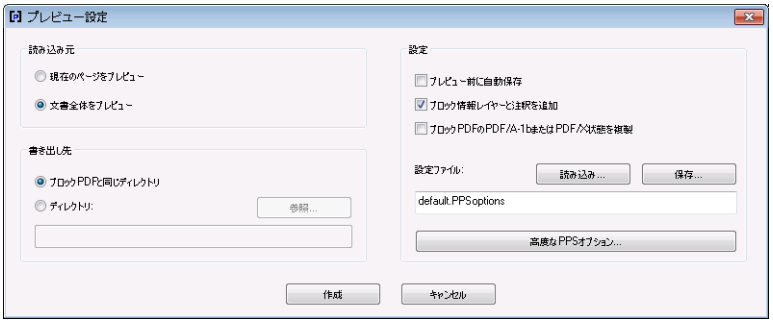


図 11.5
ブロック
プレビュー設定

- **「高度な *PPS* オプション」**ダイアログを利用して、PPS API に従って PPS 関数のオプションリストを追加指定することもできます。たとえば、***PDF_set_option()*** の ***searchpath*** オプションを用いて、ブロックへの流し込みに使うフォントや画像が置かれているディレクトリを指定することができます。高度なオプションを指定する際には、PPS コードを書くプログラマーとの連携のもとに行うことを推奨します。

プレビュー設定は、ディスクファイルに保存して、後で再読み込みすることも可能です。

プレビューで提供される情報　生成されるプレビュー文書には、元のページ内容（背景）と、流し込みが行われたブロックのほかに、さまざまな情報も含まれることがあります。この情報は、ブロックや PPS 設定のチェックや改善に役立つものです。デフォルト内容を持つアクティブなブロックそれぞれについて、以下の項目が生成されます：

- **エラーマーカ**：流し込みが成功しなかったブロックは、打ち消し線の付いた矩形として図示され、容易に識別することができます。エラーマーカは、ブロックが処理できなかったときにはつねに生成されます。
- **しおり**：ブロックの処理結果はしおりにまとめられます。このしおりは、ページ番号とブロックの種類に従って、かつエラーが起きたときはエラーにも従って、構造化されています。しおりは**「表示」→「ナビゲーションパネル」→「しおり」**で表示できます。しおりはつねに生成されます。
- **注釈**：処理されたブロックごとに、ブロック内容そのものに加えて、ページ上に注釈が生成されます。この注釈の矩形は、元のブロックの枠を図示しています（デフォルト内容と流し込みモードによっては、これはブロック内容の枠とは異なる場合があるのです）。この注釈の中にはブロックの名前が入っており、ブロックへの流し込みができなかったときにはエラーメッセージも入っています。注釈はデフォルトで生成されますが、プレビュー設定で無効化することもできます。PDF/A-1・PDF/X 規格では注釈の使用は制限されていますので、**「ブロック *PDF* の *PDF/A-1b* または *PDF/X* 状態を複製」**オプションを有効にすると、注釈は生成されません。
- **レイヤー**：ページの内容は、分析とデバッグが容易になるよう、複数のレイヤーに分けて配置されます。ページ背景（すなわち元のページの内容）、ブロックの各種類、流し込みができなかったエラーブロック、ブロック情報を持った注釈について、それぞれ別々のレイヤーが生成されます。空のままになるレイヤーについては生成されません（エラーが何も起きなかった場合等）。レイヤー一覧は**「表示」→「ナビゲーションパネル」→「レイヤー」**で表示できます。デフォルトでは、ページ上のすべてのレイヤーが表示されます。いずれかのレイヤーの内容を見えなくするには、そのレイヤーの名前の左の目のアイコンをクリックします。レイヤーの生成は、プレビュー設定で無効化することも可能です。PDF/A-1・PDF/X 規格ではレイヤーの使用は制限されていますので、**「ブロック *PDF* の *PDF/A-1b* または *PDF/X* 状態を複製」**オプションを有効にすると、レイヤーは生成されません。

PDF/A または PDF/X 状態を複製　「ブロック *PDF* の *PDF/A-1b* または *PDF/X* 状態を複製」設定は、PDF/A 規格か PDF/X 規格に従った PDF 出力を生成する必要があるときに有用です。複製モードは、入力が以下のいずれかの規格に準拠しているときに有効にできます：

PDF/A-1b:2005
PDF/X-1a:2001・PDF/X-1a:2003
PDF/X-3:2002・PDF/X-3:2003
PDF/X-4・PDF/X-4p
PDF/X-5g・PDF/X-5pg

プレビューが複製モードで生成されるときは、PPS は、ブロック PDF の以下の性質を、生成するプレビューへ複製します：

- PDF 規格識別。
- 出力インテント条件。
- ページ寸法。すべてのページボックスを含みます。
- XMP 文書メタデータ。

規格準拠の PDF 文書を複製する際には、すべてのブロック流し込み操作が、それぞれの規格に準拠している必要があります。たとえば、出力インテントがなければ、ICC プロファイルのない RGB 画像は使うことができません。同様に、使用しているフォントはすべて埋め込む必要があります。要請の全一覧は、253 ページ「10.3 PDF/X による印刷出力」と、261 ページ「10.4 PDF/A によるアーカイビング」に示しています。PDF/A または PDF/X 複製モードでのブロック流し込み操作が、選択されている規格に違反するときには（デフォルト画像が RGB カラースペースを用いているにもかかわらず、文書が適切な出力インテントを含んでいない場合等）、エラーメッセージが表示され、プレビューは生成されません。これにより、ユーザーは作業フローの非常に早い時点で、規格違反の危険を感知することができます。

デフォルトテキストに記号フォントを用いる　ブロックのデフォルトテキストを記号フォントで与えるには、2 つの方法があります：

- Windowsの文字コード表アプリケーションなどに示されている8ビットレガシコードを用いる：***defaulttext*** に対して 8 ビットコードを、その対応する 8 ビットキャラクタをリテラルに入力するか（Windows の文字コード表からコピー/ 貼り付けするなどして）、あるいは数値エスケープシーケンスとして与えます。この場合には、**「テキスト作成」**プロパティグループ内の ***charref*** プロパティのデフォルト値を ***false*** にしておく必要があり、また、文字参照を用いることはできません。たとえば、下記のデフォルトテキストは、***charref=false*** のとき、記号フォント Wingdings の「スマイリー」グリフを生成します：

  ```
  J
  \x4A
  \112
  ```

- フォント内で用いられている Unicode 値かグリフ名を用いる：**「テキスト作成」**プロパティグループの ***charref*** プロパティを ***true*** に設定したうえで、記号の文字参照かグリフ名参照を与えます（112 ページ「4.5.2 文字参照」参照）。たとえば、下記のデフォルトテキストは、***charref=true*** のとき、記号フォント Wingdings の「スマイリー」グリフを生成します：

  ```
  &#xF04A;
  &.smileface;
  ```

なお、どちらの方法でも、ブロックプロパティダイアログ上は、実際の記号グリフではなく文字化けした状態で表示されます。

11.5 PPS でブロックへ流し込み

PPS でブロックへ流し込みを行うには、まずブロックを含むページを、*PDF_fit_pdi_page()* 関数で出力ページ上に貼り付ける必要があります。ページを貼り付けた後、その上のブロックへ *PDF_fill_*block()* 関数群で流し込みを行うことができます。

簡単な例：可変テキストをテンプレートに追加　PDF テンプレートへの動的テキストの追加は、非常に頻繁に必要となる動作です。以下のコードでは、入力 PDF 文書（テンプレート、ブロックコンテナ）の中のページを開き、それを出力ページ上に配置し、そして *firstname* というテキストブロックに可変テキストを入れ込んでいます：

```
doc = p.open_pdi_document(filename, "");
if (doc == -1)
        throw new Exception("エラー: " + p.get_errmsg());

page = p.open_pdi_page(doc, pageno, "");
if (page == -1)
        throw new Exception("エラー: " + p.get_errmsg());

p.begin_page_ext(width, height, "");
/* 取り込んだページを貼り付け */
p.fit_pdi_page(page, 0.0, 0.0, "");

/* 貼り付けたページ上のブロック1個へ流し込み */
p.fill_textblock(page, "firstname", "Serge", "encoding=winansi");

p.close_pdi_page(page);
p.end_page_ext("");
p.close_pdi_document(doc);
```

クックブック 完全なコードサンプルがクックブックの blocks/starter_block トピックにあります。

ブロックのプロパティを上書き　場合によってプログラマーは、ブロックの定義が与えているプロパティ群を一部だけ採用し、その他のプロパティをカスタムの値で上書きしたいことがあります。これはさまざまな場合に有用です：

- 業務上の要請である種の上書きが必要と判断される場合に対応。
- 画像・PDF ページの拡縮倍率を、ブロック定義から採らずに、アプリケーションで算出。
- ブロックの座標をプログラムで変える。生成したい請求書のデータ項目数が一定でない場合等。
- 別々のスポットカラー名を与えることも可能。プリントサービス業務で、顧客ごとの要請に合わせるため。

プロパティを上書きするには、*PDF_fill_*block()* 関数群のオプションリストにプロパティの名前とその値を与えます。例：

```
p.fill_textblock(page, "firstname", "Serge", "fontsize=12");
```

これは、ブロックの内部の *fontsize* プロパティを、与えた値 12 で上書きします。ほとんどすべてのプロパティ名を、オプションとして用いることが可能です。

プロパティの上書きは、それぞれの関数呼び出しにのみ適用されます。ブロック定義内に保持されるわけではありません。

流し込んだブロックの上に取り込んだページを配置　取り込んだページは、どのブロック流し込み関数を使うよりも前に、出力ページ上に配置しておく必要があります。ということは元ページは通常、ブロック内容よりも下に配置されることになります。しかし場合によっては、流し込みが行われたブロックよりも上に元ページを配置したいこともあるでしょう。これを実現するには、*PDF_fit_pdi_page()* の *blind* オプションを用いてページを一度貼り付けることにより、そのページ上のブロックとその位置を PPS に知らせておき、ブロックへの流し込みが済んだ後にページを再び貼り付けることにより、実際にページ内容を表示させます：

```
/* ブロックを用意するためにページをblindモードで配置してページを見えなくする */
p.fit_pdi_page(page, 0.0, 0.0, "blind");

/* ブロックへ流し込み */
p.fill_textblock(page, "firstname", "Serge", "encoding=winansi");
/* ... いろいろなブロックへ流し込み ... */

/* ページを再度配置、今度は見えるように */
p.fit_pdi_page(page, 0.0, 0.0, "");
```

クックブック　**完全なコードサンプルがクックブックの** blocks/block_below_contents **トピックにあります。**

ブロックへ流し込む際にコンテナページを無視　取り込んだブロックは、そのブロックの背景のページ内容を一切参照せずに、プレースホルダとして使ってもよいでしょう。ブロックを持つコンテナページをブラインドモードで、すなわち *PDF_fit_pdi_page()* で *blind* オプションを指定して、1 個ないし複数のページ上に取り込んだうえで、ブロックへの流し込みを行うという方法です。こうすれば、出力ページ上にコンテナページを貼り付けることなく、ブロックやそのプロパティの利点を活用することができ、また、ブロックを複数のページ上に（あるいは同一出力ページ上にさえ）複製することが可能になります。

クックブック　**完全なコードサンプルがクックブックの** blocks/duplicate_block **トピックにあります。**

テキストフローブロックを連結　テキストフローブロックは、前のブロックからあふれたテキストが次のブロックに入るよう、連結することが可能です。たとえば、長い可変テキストがあって、別のページへ続かせる必要が想定される場合、2 個のブロックを連結しておけば、1 個目のブロックがいっぱいになっても、残りは 2 個目のブロックへ流し込まれます。

PPS は、*PDF_fill_textblock()* とブロックプロパティに与えられたテキストから、1 個のテキストフローを内部的に生成します。連結されていないブロックの場合は、このテキストフローはそのブロック内に配置され、そのテキストフローハンドルは呼び出しが終わった時点で削除され、あふれたテキストは失われます。

連結されたテキストフローの場合は、最初のブロックへ流し込んだ後に余っているあふれテキストを、その次のブロックへ流し込むことができます。最初のテキストフローの余りがブロック内容として使われ、新たなテキストフローは作成されません。テキストフローブロックの連結は以下のように動作します：

- 連結されたテキストブロックのチェーンの中の最初の *PDF_fill_textblock()* を呼び出す時は、*textflowhandle* オプションに値 -1（PHP では 0）を与える必要があります。内部的に生成されたテキストフローハンドルを *PDF_fill_textblock()* が返しますので、アプリケーション側でこれを保持しておく必要があります。

- *PDF_fill_textblock()* への次の呼び出しでは、前段で返されたテキストフローハンドルを *textflowhandle* オプションに与えることができます（このとき *text* 引数にテキストを与えても無視されるので、空にするべきです）。ブロックへ、テキストフローの余りが流し込まれます。
- この処理を、さらなるテキストフローブロック群に対して繰り返すことができます。
- 返されたテキストフローハンドルは、*PDF_info_textflow()* に与えれば、ブロック流し込みの結果を知ることができます。終了状況やテキストの終了位置などがわかります。

なお、*fitmethod* プロパティは *clip* に設定する必要があります（*textflowhandle* を与えているときはどのみちこれがデフォルトです）。テキストフローブロックを連結する基本的なコードの骨組みは以下のようになります：

```
p.fit_pdi_page(page, 0.0, 0.0, "");
tf = -1;

for (i = 0; i < blockcount; i++)
{
        String optlist = "encoding=winansi textflowhandle=" + tf;
        tf = p.fill_textblock(page, blocknames[i], text, optlist);
        text = null;

        if (tf == -1)
            break;

        /* いちばん最近のfit_textflow()呼び出しの結果をチェック */
        reason = (int) p.info_textflow(tf, "returnreason");
        result = p.get_parameter("string", (float) reason);

        /* テキストが全部配置されたならループを抜ける */
        if (result.equals("_stop"))
        {
            p.delete_textflow(tf);
            break;
        }
}
```

クックブック **完全なコードサンプルがクックブックの** blocks/linked_textblocks **トピックにあります。**

ブロックの流し込み順序　ブロック関数群 *PDF_fill_*block()* は、プロパティとブロック内容を、以下の順序で処理します：

- 背景：*backgroundcolor* プロパティが存在し、かつ *None* 以外のカラースペースキーワードを持っているときは、ブロック領域は指定された色で塗られます。
- 枠線：*bordercolor* プロパティが存在し、かつ *None* 以外のカラースペースキーワードを持っているときは、ブロックの枠は指定された色と線幅で描線されます。
- 内容：与えられたブロック内容と、*bordercolor*・*linewidth* 以外のすべてのプロパティが処理されます。
- テキスト行・テキストフローブロック：テキストもデフォルトテキストも与えられていないときは、何の出力も行われません。背景色やブロックの枠線もありません。

入れ子になったブロック　ブロックへ流し込みを行う前には、そのブロックを含むページを出力ページ上にまず貼り付ける必要があります（そうでないと、ページを拡縮・回転・平行移動した後のブロックの位置を PPS が知りえないため）。ページをブロックのコ

ンテナとしてのみ使っており、静的内容を新ページへ複製しなくてよい場合には、取り込んだページを、*blind* オプションを用いて貼り付けることができます。

取り込んだページを、どのような方法で出力ページ上に貼り付けても、ブロックへの流し込みは行うことができます：

- ページは、*PDF_fit_pdi_page()* で直接貼り付けることができます。
- ページは、テーブルセル内に *PDF_fit_table()* で間接的に貼り付けることができます。
- ページは、他の PDF ブロックの内容として *PDF_fill_pdfblock()* で貼り付けることができます。

この 3 番目の方法、すなわち PDF ブロックへ、ブロックを含む他のページを流し込むという方法を用いると、ブロックコンテナを入れ子にすることができます。これを活用すると、面白い使い方を簡単に実装できます。たとえば、2 段階のブロック流し込み処理で、組み付けとパーソナライゼーションの両方を実装することができます：

- 第一層のブロックコンテナページには、いくつかの大きな PDF ブロックを置きます。これらは、印刷する紙の上の主要な領域を表しています。PDF ブロックの配置は、想定している紙の後工程を反映しています（折り・断裁等）。
- この第一層の PDF ブロックそれぞれへ、第二層のコンテナ PDF ページを流し込みます。この PDF ページには、テキスト・画像・PDF ブロックを置いておき、それらへ可変テキストを流し込んでパーソナライゼーションを行います。

この方法で、ブロックコンテナは入れ子にすることができます。ブロックの入れ子は何重でも可能ですが、三重以上の入れ子が必要になることはまれでしょう。

この第二層のブロックコンテナは、各組み付けページで同じにすることもできますし、別のものにすることもできます。もし同じにした場合は、すべての第二層ブロックへの流し込みが済んでから、次の第一層ブロックへの流し込みを行う必要があります。なぜなら、第二層ブロックをページ上に配置するための情報は、次のインスタンスの第二層コンテナページが配置された時点で、利用不可能になるからです。

クックブック **完全なコードサンプルがクックブックの** `blocks/nested_blocks` **トピックにあります。**

ブロックの座標　ブロックの矩形の座標は、PDF のデフォルト座標系を参照しています。ブロックを含んだページを PPS で出力ページに配置するときには、*PDF_fit_pdi_page()* に対していくつかの配置オプションや拡縮オプションを与えることができます。これらのオプションは、そのブロックが処理される際に考慮されます。これを利用すると、1 つのテンプレートページを出力ページ上に何度でも配置して、そのたびにそのブロック群へデータを流し込むことができます。たとえば 1 枚の組み付け紙上に、1 つの名刺テンプレートを 4 回配置するといったことが可能です。ブロック関数群は、座標系の変換を正しく行い、すべてのブロックに対して、それがページ上に配置されるたびに、正しくテキストを配置します。クライアントに求められるのはただ、ページを配置して、そしてその配置したページ上のすべてのブロックを処理することだけです。以後はそのページを、出力ページ上の他の場所に配置したうえで、その新しい場所に対してさらにブロック処理操作を行うことができ、これを繰り返していくことが可能です。

Block Plugin におけるブロック座標の表示のされかたは、PDF ファイル内に格納されているものとは異なっています。プラグインでは Acrobat の方式を用いて、座標の原点をページの左上隅に置いていますが、内部座標（ブロック内に格納されているもの）では PDF の方式を用いて、座標の原点をページの左下隅に置いているためです。プロパティダイアログの座標表示は、Acrobat で指定されている単位にも従います（285 ページ「ブロックの大きさと位置」参照）。

ブロックプロパティでスポットカラー　ブロックプロパティで特色（スポットカラー）を使うには、「...」をクリックすれば、HKS・PANTONE スポットカラーの全一覧を表示させることができます。これらの色名は PPS に内蔵されており（80 ページ「3.5.2 Pantone・HKS・カスタムスポットカラー」参照）、それ以上の準備なしに使用できます。カスタムスポットカラーに対しては、Block Plugin で代替色を定義することが可能です。ブロックプロパティで代替色を指定していないときは、PPS アプリケーションで *PDF_makespotcolor()* を用いてカスタムスポットカラーをあらかじめ定義しておく必要があります。そうでないとブロックへの流し込みは失敗します。

11.6 ブロックのプロパティ

PPS と Block Plugin では、どの種類のブロックに対しても適用することのできる一般プロパティ群が用意されています。そのほかに、ブロックの種類**「テキスト行」・「テキストフロー」・「画像」・「*PDF* ブロック」**にそれぞれ特有のプロパティ群もあります。

プロパティは、ハンドルとアクションリストを除いて、オプションリストと同じデータ型に対応しています。

ブロックプロパティの名前は一般に、***PDF_fit_image( )*** に対するオプションと同じです（***fitmethod*・*charspacing*** 等）。その場合、それぞれの動作は、対応するオプションの解説に書いてあるものとまったく同じです。

11.6.1 管理プロパティ群

管理プロパティ群は、すべての種類のブロックに適用されます。必須の項目は、Block Plugin によって自動生成されます。表 11.4 に、管理プロパティの一覧を示します。

表 11.4 管理プロパティ一覧

キーワード	とりうる値・解説
Description	（文字列）ブロックの機能に関する、人が読める説明。エンコーディングは PDFDocEncoding か Unicode（後者の場合は先頭 BOM）。このプロパティは、ユーザーへの情報提供のためだけにあり、PPS には無視されます。
Locked	（論理値）true なら、ブロックとそのプロパティは Block Plugin で編集できません。このプロパティは PPS には無視されます。デフォルト：false
Name	（文字列、必須）ブロックの名前。ブロック名は、ページごとに一意である必要がありますが、文書内では一意でなくてもかまいません。3 種のキャラクタ [ ] / は、ブロック名には使えません。ブロック名は最長 125 文字です。
Subtype	（キーワード、必須）ブロックの種類によって、Text・Image・PDF のいずれか。テキスト行ブロックとテキストフローブロックは、ともに Subtype が Text になることに留意してください。両者は textflow プロパティによって識別されます。
textflow	（論理値）一行処理か複数行処理かを制御します。このプロパティは、Block Plugin のユーザーインタフェースでは表示されず、それぞれテキスト行ブロックとテキストフローブロックとして表されます（デフォルト：false）： ***false*** テキスト行ブロック：テキストが一行で組まれ、PDF_fit_textline( ) で処理されます。 ***true*** テキストフローブロック：テキストが複数行にわたる場合があり、PDF_fit_textflow( ) で処理されます。標準テキストプロパティ群に加え、テキストフロー関連のプロパティ群も指定できます（表 11.9 参照）。
Type	（キーワード、必須）つねに Block

11.6.2 矩形プロパティ群

矩形プロパティ群は、すべての種類のブロックに適用されます。これらは、ブロックの矩形本体の書式を記述します。必須の項目は、Block Plugin によって自動生成されます。表 11.5 に、矩形プロパティの一覧を示します。

表 11.5 矩形プロパティ一覧

キーワード	とりうる値・解説
background-color	（色）このプロパティが存在し、かつ None 以外の色空間キーワードを持つならば、矩形が描かれ、与えられた色で塗られます。既存のページ内容をおおい隠したいときに有用です。デフォルト：None
bordercolor	（色）このプロパティが存在し、かつ None 以外の色空間キーワードを持つならば、矩形が描かれ、与えられた色で描線されます。デフォルト：None
linewidth	（float。正の値でなければならない）ブロックの矩形を描くのに用いる線の描線幅。bordercolor 設定時のみ有効です。デフォルト：1
Rect	（矩形、必須）ブロックの座標。座標原点はページ左下隅。ただし Block Plugin では、座標は Acrobat の方式で、すなわちページ左上隅を原点として表されます。座標の単位は、Acrobat でその時点で選択されている単位で表示されますが、PDF ファイル内部ではつねにポイント単位で格納されています。
Status	（キーワード）PPS とブロック機能がブロックを処理する方法を記述します（デフォルト：active）： ***active*** ブロックは、そのプロパティに従って完全に処理されます。 ***ignore*** ブロックは無視されます。 ***ignoredefault*** defaulttext/image/pdf プロパティが無視される、すなわち可変内容がないとき（特にプレビューで）ブロックが空のままになる点を除き、active と同じです。これは特に、ブロックがプレビュー生成のためのデフォルト内容を持っているかもしれないけれども、サーバサイドでブロックへ流し込みが行われる際にはそのブロックのデフォルト内容が用いられないようにしたいときに有用です。また、ブロックをプレビューする際にも、ブロックプロパティからデフォルト内容を削除することなくデフォルト内容を無効化するために用いることができます。 ***static*** 可変内容が配置されません。ブロックにデフォルトのテキスト・画像・PDF 内容があれば、それが用いられます。

11.6.3 書式プロパティ群

書式プロパティ群は、組版の詳細を指定します：

- 表 11.6 に、透過書式プロパティの一覧を示します。これらは、すべての種類のブロックに適用されます。
- 表 11.7 に、テキスト書式プロパティの一覧を示します。これらは、テキスト行ブロックとテキストフローブロックに適用されます。

表 11.6 透過書式プロパティー覧（すべてのブロックに適用）

キーワード	とりうる値・解説
blendmode	（キーワードリスト。PDF 1.4。PDF/A モードで用いられるときは値 Normal を持つ必要があります）ブレンドモードの名前：None・Color・ColorDodge・ColorBurn・Darken・Difference・Exclusion・HardLight・Hue・Lighten・Luminosity・Multiply・None・Normal・Overlay・Saturation・Screen・SoftLight。デフォルト：None
opacityfill	（float。PDF 1.4。PDF/A モードで用いられるときは値 1 を持つ必要があります）塗り操作の不透明度を範囲 0 ～ 1 で表したもの。値 0 は完全透過を意味し、1 は完全不透過を意味します。
opacitystroke	（float。PDF 1.4。PDF/A モードで用いられるときは値 1 を持つ必要があります）描線操作の不透明度を範囲 0 ～ 1 で表したもの。値 0 は完全透過を意味し、1 は完全不透過を意味します。

表 11.7 テキスト書式プロパティー覧（テキスト行・テキストフローブロックに適用）

キーワード	とりうる値・解説
charspacing	（float またはパーセント値）文字間隔。パーセント値は fontsize に対する割合。デフォルト：0
decoration-above	（論理値）true の場合、underline・strikeout・overline オプションで有効にされたテキスト装飾がテキストの前面に描かれ、そうでなければテキストの背面に描かれます。描画順序を変えると、装飾線の書式に影響を与えます。デフォルト：false
fillcolor	（色）テキストの塗り色。デフォルト：gray 0（= 黒）
fontname[1]	（文字列）フォントの名前。PDF_load_font( ) が求めるものと同じです。Block Plugin では、システムで利用可能なフォントの一覧が表示されます。ただしこうしたフォント名は、Mac・Windows・Unix システム間で互換とは限りません。fontname の先頭が「@」キャラクタである場合は、そのフォントは縦書きモードで適用されます。 ブロックへの流し込み時には、PDF_fill_textblock( ) にオプションとしてテキストのエンコーディングを与える必要があります。ただし、font オプションが与えられている場合は除きます。
fontsize[1]	（float）文字のサイズをポイント単位で指定します
fontstyle	（キーワード）フォントスタイル。normal・bold・italic・bolditalic のいずれかでなければなりません
horizscaling	（float またはパーセント値）テキストの水平倍率。デフォルト：100%
italicangle	（float）テキストの斜体角度を度単位で表したもの。デフォルト：0
kerning	（論理値）カーニングの動作。デフォルト：false
monospace	（整数：1 ～ 2048）フォントの全キャラクタを強制的に等幅にします。デフォルト：無指定（フォントのメトリックが用いられます）
overline	（論理値）上線のモード。デフォルト：false
strikeout	（論理値）取り消し線のモード。デフォルト：false
strokecolor	（色）テキストの描線色。デフォルト：gray 0（= 黒）
strokewidth	（float・パーセント値・キーワードのいずれか。textrendering が袋文字に設定されているときのみ意味を持ちます）袋文字の線幅（ユーザー座標で、または percentage に対するパーセント値で）。キーワード auto または値 0 は内蔵のデフォルトを用います。デフォルト：auto
textrendering	（整数）テキスト表現モード。Type 3 フォントでは値 3 のみ意味を持ちます（デフォルト：0）： ***0***  テキストを塗る ***1***  テキストを描線（袋文字） ***2***  テキストを塗って描線 ***3*** 不可視テキスト ***4***  テキストを塗り、クリッピングパスに追加 ***5***  テキストを描線し、クリッピングパスに追加 ***6***  テキストを塗って描線し、クリッピングパスに追加 ***7***  テキストをクリッピングパスに追加（ブロックでは不可）
textrise	（float またはパーセント値）テキストの縦方向のオフセット。パーセント値は fontsize に対する割合。デフォルト：0
underline	（論理値）下線のモード。デフォルト：false
underline-position	（float・パーセント値・キーワードのいずれか）下線テキストのベースラインに対する描線の相対位置。パーセント値は fontsize に対する割合。デフォルト：auto

表 11.7 テキスト書式プロパティー覧（テキスト行・テキストフローブロックに適用）

キーワード	とりうる値・解説
underline-width	（float・パーセント値・キーワードのいずれか）下線テキストの線幅。パーセント値は fontsize に対する割合。デフォルト：auto
wordspacing	（float またはパーセント値）単語間隔。パーセント値は fontsize に対する割合。デフォルト：0

1. このプロパティは、テキスト行・テキストフローブロックでは必須です。Block Plugin はこれを自動生成します。

11.6.4 テキスト作成プロパティ群

テキスト作成プロパティ群は、テキスト行・テキストフローブロックの前処理工程を指定します。表 11.8 に、テキスト行・テキストフローブロックに適用されるテキスト作成プロパティの一覧を示します。

表 11.8 テキスト作成プロパティ一覧（テキスト行・テキストフローブロックに適用）

キーワード	とりうる値・解説
charref	（論理値）true なら、数値参照・文字参照・グリフ名参照の置き換えを行います。デフォルト：グローバルな charref なパラメタ
escape-sequence	（論理値）true なら、内容文字列・ハイパーテキスト文字列・名前文字列内のエスケープシーケンスの置き換えを行います。デフォルト：グローバルな escapesequence パラメタ
features	（キーワードのリスト）script・language オプションに従って、OpenType フォントのどのタイポグラフィ機能をテキストに適用するかを指定します。フォント内に存在しない機能に対するキーワードは、エラーを出さずに無視されます。以下のキーワードを与えることができます： ***_none*** フォント内のどの機能も適用しません。ただし vert 機能だけは、novert キーワードで明示的に無効化する必要があります。 **< 名前 >** 4 文字の OpenType タグ名を与えてその機能を有効にします。よく用いられる機能名は liga・ital・tnum・smcp・swsh・zero です。利用できるすべての機能の名前と説明の全一覧は、154 ページ「6.3.1 対応している OpenType レイアウト機能」に示しています。 ***no*< 名前 >** 機能名の前に接尾辞 no をつけると（noliga 等）その機能が無効化されます。 デフォルト：横書きモードでは _none。縦書きモードでは vert が自動的に適用されます。 OpenType 機能を利用するには、PDF_load_font( ) で readfeatures オプションが必須です。
language	（キーワード。script が与えられているときのみ意味を持ちます）指定された言語に従ってテキストが処理されます。これは features・shaping オプションに対して意味を持ちます。キーワードの全一覧は 163 ページ「6.4.2 用字系と言語」に示しています。例：ARA（アラビア語）・JAN（日本語）・HIN（ヒンディー語）。デフォルト：_none（言語未定義）
script	（キーワード。shaping=true のときは必須）指定された用字系に従ってテキストが処理されます。これは features・shaping・advancedlinebreak オプションに対して意味を持ちます。用字系としてもっともよく用いられるキーワードは次のとおりです：_none（未定義用字系）latn・grek・cyrl・armn・hebr・arab・deva・beng・guru・gujr・orya・taml・thai・laoo・tibt・hang・kana・han。キーワード _auto を指定すると、テキスト内のキャラクタの多数が属する用字系が選ばれますが、ただし _latn と _none は無視されます。キーワードの全一覧は 163 ページ「6.4.2 用字系と言語」に示しています。デフォルト：_none
shaping	（論理値）true なら、script・language オプションに従って文字の字形選択（シェーピング）が行われます。script オプションが _none 以外の値を持つ必要があり、かつ、必要なシェーピングテーブルがフォント内に存在している必要があります。デフォルト：false

11.6.5 テキスト組版プロパティ群

表 11.9 に、テキストフローブロックに対してのみ用いることができるテキスト組版プロパティの一覧を示します。ただし *stamp* プロパティだけは、テキスト行ブロックに対しても用いることができます。これらは、テキストフローを処理するための初期オプションリストを構築するために用いられます（*PDF_create_textflow()* の *optlist* パラメタと対応しています）。テキストフローで用いるインラインオプションリストは、プラグインでは指定することができず、サーバ上で *PDF_fill_textblock()* によるブロックへの流し込みの際に、またはブロックの *defaulttext* プロパティ内で、テキスト内容の一部として与えることができます。

表 11.9 テキスト組版プロパティー覧（主にテキストフローブロックに適用）

キーワード	とりうる値・解説
adjust-method	（キーワード）minspacing・maxspacing オプションで指定された制限内で単語間隔を詰めたり拡げたりしてもテキストの一部が行内に収まらない時、行の調整に用いる方式（デフォルト：auto）： ***auto*** 次の方式を順に適用：shrink・spread・nofit・split。 ***clip*** はめ込み枠の右端（rightindent オプションを考慮）からはみ出した部分を切り落とす点を除いて、nofit と同じ。 ***nofit*** 最後の単語を次行へ送ります。ただし、残される（短い）行が、nofitlimit オプションで指定されたパーセント値よりも短くならない場合に限ります。均等配置の段落であっても、若干がたついて見えることがあります。 ***shrink*** 単語が行内に収まらないとき、テキストを shrinklimit の制限内で圧縮します。それでも収まらなければ nofit 方式を適用します。 ***split*** 最後の単語を次行へ送らず、強制的にハイフネーションします。テキストフォントならハイフンキャラクタを挿入しますが、記号フォントなら挿入しません。 ***spread*** 最後の単語を次行へ送り、残された（短い）行を均等配置するよう単語内の文字間の間隔を spreadlimit の制限内で拡げます。それで均等配置できなければ nofit 方式を適用します。
advanced-linebreak	（論理値）複雑用字系で必要とされる高度な改行アルゴリズムを適用します。これはタイ語等、単語間の区切りを空白キャラクタを用いて示さない用字系での改行に必須です。locale・script オプションに従います。デフォルト：false
alignment	（キーワード）段落内の行の書式を指定。デフォルト：left。 ***left*** 左揃え（leftindent を始点として）。 ***center*** 中央揃え（leftindent から rightindent までの間で）。 ***right*** 右揃え（rightindent を終点として）。 ***justify*** 両端揃え。
avoid-emptybegin	（論理値）true なら、はめ込み枠先頭の空行が削除されます。デフォルト：false
fixedleading	（論理値）true なら、各行内で最初に見つかった行送り値を用います。そうでないなら、行内のすべての行送り値のうちの最大値を用います。デフォルト：false
hortab-method	（キーワード）テキスト内の水平タブの扱い。算出位置がカレントテキスト位置より左のときは、そのタブは無視されます（デフォルト：relative）： ***relative*** hortabsize で指定された分、位置を進めます。 ***typewriter*** hortabsize の次の倍数まで位置を進めます。 ***ruler*** ruler オプション内の n 番目のタブ値まで位置を進めます。ここで n は、その行内でそれまでに見つかったタブの数です。n がタブ位置の数を超える分については、relative 方式を適用します。

表 11.9 テキスト組版プロパティー覧（主にテキストフローブロックに適用）

キーワード	とりうる値・解説
hortabsize	（float またはパーセント値）水平タブの幅[1]。その解釈は hortabmethod オプションに依存します。デフォルト：7.5%
lastalignment	（キーワード）段落内の最終行の書式。alignment オプションのすべてのキーワードを用いることができるほか、以下のキーワードも用いることができます（デフォルト：auto）： ***auto*** alignment オプションの値を用います。ただしそれが justify のときは left を用います。
leading	（float またはパーセント値）テキストのベースライン間の間隔。ユーザー座標で、または文字サイズに対するパーセント値で指定します。デフォルト：100%
locale	（キーワード）advancedlinebreak=true のとき、用字系特有の改行方式で用いられるロケール。キーワードは、以下の構成要素（複数可）から成り、オプショナルな構成要素は下線キャラクタ「_」で区切られます（その文法は、NLS/POSIX のロケール ID とは若干異なっています）： ▸（必須）ISO 639-2 に従った、小文字 2 文字または 3 文字の言語コード（www.loc.gov/standards/iso639-2 参照）。例：en（英語）・de（ドイツ語）・ja（日本語）。これは language オプションとは異なっています。 ▸（オプション）ISO 15924 に従った、4 文字の用字系コード（www.unicode.org/iso15924/iso15924-codes.html 参照）。例：Hira（ひらがな）・Hebr（ヘブライ文字）・Arab（アラビア文字）・Thai（タイ文字）。 ▸（オプション）ISO 3166 に従った、大文字 2 文字の国コード（www.iso.org/iso/country_codes/iso_3166_code_lists 参照）。例：DE（ドイツ）・CH（スイス）・GB（イギリス）。 ロケールを指定することは、高度な改行のために必須ではありません。キーワード _none は、ロケール独自の処理が行われないことを指定します。デフォルト：_none。 例：de_DE・en_US・en_GB
maxspacing ***minspacing***	（float またはパーセント値）単語間の最大間隔・最小間隔（ユーザー座標で表すか、空白キャラクタの幅に対するパーセント値で指定）。算出される単語間隔が、与えられた値までに制限されます（ただし、wordspacing オプションはなお加算されます）。デフォルト：minspacing=50%、maxspacing=500%
minlinecount	（整数）はめ込み枠の最終段落の最小行数。これより行が少ないときは、次のはめ込み枠内に配置されます。値 2 を用いると、段落の中の 1 行だけがはめ込み枠末尾に配置される（「オーファン」）のを避けられます。デフォルト：1
nofitlimit	（float またはパーセント値）nofit 方式における行の長さの下限（ユーザー座標で表すか、はめ込み枠の幅に対するパーセント値で指定）。デフォルト：75%
parindent	（float またはパーセント値）段落の先頭行の左インデント[1]。leftindent にこの値が加算されます。このオプションを行内で指定すると、タブのように動作します。デフォルト：0
rightindent ***leftindent***	（float またはパーセント値）全テキスト行の右インデント・左インデント[1]。leftindent が行内で指定された場合、決定された位置がカレントテキスト位置より左のときは、このオプションはカレント行については無視されます。デフォルト：0
ruler[2]	（float のリスト、またはパーセント値のリスト）hortabmethod=ruler に対する絶対タブ位置のリスト[1]。このリストは、非負値を昇順で最大 32 個持てます。デフォルト：hortabsize の整数倍
shrinklimit	（パーセント値）shrink 方式におけるテキスト長体の下限。算出される縮小率が、与えられた値までに制限されます。ただし、horizscaling オプションの値が乗算されます。デフォルト：85%
spreadlimit	（float またはパーセント値）spread 方式における 2 文字間の間隔の上限（ユーザー座標で表すか、文字サイズに対するパーセント値で指定）。算出された文字間隔が、charspacing オプションの値に加算されます。デフォルト：0

表 11.9 テキスト組版プロパティー覧（主にテキストフローブロックに適用）

キーワード	とりうる値・解説
stamp	（キーワード。テキスト行・テキストフローブロック）このオプションを使うと、ブロック矩形内の対角線上にスタンプを作成することができます。スタンプのテキストは可能な限り拡大されて印字されます。スタンプのテキストを枠内に配置する際には、position・fitmethod・orientate（north・south のみ）オプションに従います。デフォルト：none。 ***ll2ur*** 左下隅から右上隅へ向かう対角線上にスタンプが配置されます。 ***ul2lr*** 左上隅から右下隅へ向かう対角線上にスタンプが配置されます。 ***none*** スタンプは作成されません。
tabalignchar	（整数）タブの小数点揃えの整列位置にしたいキャラクタの Unicode 値。デフォルト：キャラクタ「.」（U+002E）
tabalignment[2]	（キーワードのリスト）タブ位置の整列方式。リスト内の各項目はそれぞれ、ruler オプション内で対応する項目の整列方式を定義します（デフォルト：left）： ***center*** テキストはタブ位置で中央揃えされます。 ***decimal*** 最初に現れる tabalignchar をタブ位置で左揃えされます。tabalignchar が見つからないときは右揃えが適用されます。 ***left*** テキストはタブ位置で左揃えされます。 ***right*** テキストはタブ位置で右揃えされます。

1. ユーザー座標で、またははめ込み枠の幅に対するパーセント値で指定します。
2. タブ設定は、ブロックプロパティダイアログのプロパティサブグループ「hortabmethod=ruler におけるルーラタブ」で編集することができます。

11.6.6 オブジェクトはめ込みプロパティ群

はめ込みプロパティ群は、すべての種類のブロックで利用できますが、いくつかのプロパティは、特定の種類のブロックでのみ利用できます。これらは、ブロック内に内容が配置される方法を管理します：

- 表 11.10 に、テキスト行・画像・PDF ブロックで利用できるはめ込みプロパティの一覧を示します。
- 表 11.11 に、テキストフローブロックで利用できるはめ込みプロパティの一覧を示します（主に縦方向のはめ込みに関するものです）。

オブジェクトはめ込みアルゴリズムは、ブロック矩形をはめ込み枠として用います。*fitmethod=clip* の場合を除き、切り落としは行われません。ブロック内容がブロック矩形からはみ出さないようにしたいときは、*fitmethod=nofit* を避けてください。

表 11.10 はめ込みプロパティ一覧（テキスト行・画像・PDF ブロックに適用）

キーワード	とりうる値・解説
alignchar	（Unichar またはキーワード。テキスト行ブロックにのみ適用）指定されたキャラクタをテキスト内に見つけたとき、その左下隅を、ブロック矩形の左下隅に整列させます。横書きテキストで orientate=north か south の場合には、position オプションの 1 番目の値で位置を決定します。縦書きテキストで orientate=west か east の場合には、position オプションの 2 番目の値で位置を決定します。指定された位置整列キャラクタがテキスト内にないときは、このオプションは無視されます。値 0 かキーワード none ならば、位置整列キャラクタを無効にします。fitmethod は指定されれば適用しますが、alignchar で強制的に位置整列されるので、はめ込み枠内にテキスト配置することはできません。デフォルト：none
dpi	（float のリスト。画像ブロックにのみ適用）縦方向・横方向の画像解像度を表す 1 個または 2 個の値。単位は pixels per inch。値 0 の場合、画像内部に解像度が格納されていればそれを用い、なければ 72 dpi とします。fitmethod プロパティが指定されていて、そのキーワードが auto・meet・slice・entire のいずれかならば、本プロパティは無視されます。デフォルト：0
fitmethod	（キーワード）与えられた内容がブロック矩形に収まりきらないときの解決方式。とりうる値は auto・nofit・clip・meet・slice・entire。テキスト行・画像・PDF ブロックについては、このプロパティは標準的に解釈されます。デフォルト：auto。テキストフローブロックについては、ブロックがテキストに対して小さすぎるときは、以下のように解釈されます： ***auto*** テキストが収まるまで、fontsize と leading を縮めます。 ***nofit*** テキストをブロック下端からはみ出させます。 ***clip*** テキストをブロックの端で切り落とします。
margin	（float のリスト。テキスト行ブロックにのみ適用）テキスト枠の縦・横方向の長さ差し引きを記述する 1 個または 2 個の値。デフォルト：0
orientate	（キーワード）内容を配置する向きを指定します。とりうる値は north・east・south・west。デフォルト：north
position	（float のリスト）内容の中における参照点の位置を指定する、1 個または 2 個の値。ブロック内でのパーセント値として位置を指定します。テキスト行ブロックのみ：キーワード auto を、リストの 1 番目の値として用いることもできます。これは、テキストの筆記方向が右書きの場合（アラビア文字・ヘブライ文字等）には right を意味し、そうでない場合（欧文等）には left を意味します。 デフォルト：{0 0}、すなわち左下隅
rotate	（float）回転角を度単位で表したもの。処理が始まる前にブロックが反時計回りに回転されます。参照点が回転の中心となります。デフォルト：0

表 11.10 はめ込みプロパティー覧（テキスト行・画像・PDF ブロックに適用）

キーワード	とりうる値・解説
scale	（float のリスト）縦方向・横方向の拡縮倍率を表す 1 個または 2 個の値。fitmethod プロパティが指定されていて、かつそのキーワードが auto・meet・slice・entire のいずれかならば、本プロパティは無視されます。デフォルト：1
shrinklimit	（float またはパーセント値。テキスト行ブロックにのみ適用）fitmethod=auto でテキストを収める際に適用される縮小倍率の下限。デフォルト：0.75

表 11.11 はめ込みプロパティー覧（テキストフローブロックに適用）

キーワード	とりうる値・解説
firstlinedist	（float・パーセント値・キーワードのいずれか）ブロック矩形上端とテキスト先頭行ベースラインとの間隔を、ユーザー座標で表すか、そこの文字サイズ（fixedleading=true なら行の先頭の文字サイズ、そうでないなら行内のすべての文字サイズのうちの最大値）に対するパーセント値で表すか、キーワードで表したもの。デフォルト：leading。 ***leading*** 先頭行について決定された行送り値。Ęのような、読み分け記号つきの文字は普通、はめ込み枠上端に接するでしょう。 ***ascender*** 先頭行について決定されたアセンダ値。d や h のような、大きなアセンダを持つ文字は普通、はめ込み枠上端に接するでしょう。 ***capheight*** 先頭行について決定されたキャップハイト値。H のような大文字は普通、はめ込み枠上端に接するでしょう。 ***xheight*** 先頭行について決定された x ハイト値。x のような小文字は普通、はめ込み枠上端に接するでしょう。 fixedleading=false なら、先頭行内で見出されたすべての leading・ascender・xheight・capheight 値のうちの最大値が用いられます。
fitmethod	（キーワード）与えられた内容が枠に収まりきらないときの解決方式。とりうる値は auto・nofit・clip。デフォルト：auto。テキストフローブロックの場合、ブロックがテキストに対して小さすぎるときは、以下のように解釈されます： ***auto*** テキストが収まるまで、fontsize と leading を縮めます。 ***nofit*** テキストをブロック下端からはみ出させます。 ***clip*** テキストをブロックの端で切り落とします。
lastlinedist	（float・パーセント値・キーワードのいずれか。fitmethod=nofit のときは無視されます）テキスト最終行ベースラインとはめ込み枠下端との間隔を、ユーザー座標で表すか、文字サイズ（fixedleading=true なら行の先頭の文字サイズ、そうでないなら行内のすべての文字サイズのうちの最大値）に対するパーセント値で表すか、キーワードで表したもの。デフォルト：0、すなわちはめ込み枠下端をベースラインとして用い、ディセンダは普通、はめ込み枠の下へはみ出すでしょう。 ***descender*** 最終行について決定されたディセンダ値。g や j のような、ディセンダを持つ文字は普通、はめ込み枠下端に接するでしょう。 fixedleading=false なら、最終行内で見出されたすべてのディセンダ値のうちの最大値が用いられます。
linespread-limit	（float またはパーセント値。verticalalign=justify の場合のみ）上下合わせの場合に行送りを増やす際の最大値を、ユーザー座標で表すか、行送りに対するパーセント値で表したもの。デフォルト：200%
maxlines	（整数またはキーワード）はめ込み枠内の最大行数。あるいはキーワード auto を指定して、できるだけ多くの行をはめ込み枠内に入れさせることもできます。最大行数が入ったとき、PDF_fit_textflow( ) は文字列 _boxfull を返します。

表 11.11 はめ込みプロパティー覧（テキストフローブロックに適用）

キーワード	とりうる値・解説
minfontsize	（float またはパーセント値）shrinklimit を超えたときに、fitmethod=auto のブロック矩形に収まるようテキストが縮小される際に許される最小文字サイズ。下限値を、ユーザー座標で、またはブロックの高さに対するパーセント値で指定します。下限に達した場合には、テキストは、この指定した minfontsize を文字サイズとして生成されます。デフォルト：0.1%
orientate	（キーワード）テキストを配置する向きを指定します。とりうる値は north・east・south・west。デフォルト：north
rotate	（float）回転角を度単位で表したもの。はめ込み枠の左下隅を中心として、座標系を回転させます。これによって、枠とテキストが回転されます。テキストが配置された時点で回転はリセットされます。デフォルト：0
verticalalign	（キーワード）はめ込み枠内のテキストの縦揃え。デフォルト：top。 ***top*** 先頭行から下へ順に組版。テキストがはめ込み枠に満たないときは、テキストの下に余白があきます。 ***center*** はめ込み枠内の縦方向の中央にテキストを配置。テキストがはめ込み枠に満たないときは、テキストの上下に余白があきます。 ***bottom*** 最終行から上へ順に組版。テキストがはめ込み枠に満たないときは、テキストの上に余白があきます。 ***justify*** はめ込み枠の上端と下端にテキストを合わせます。それを実現するために、行送りを増やします。ただし、linespreadlimit で指定された限界までしか増やしません。先頭行の高さは、firstlinedist=leading の場合のみ増やします。

11.6.7 デフォルト内容に関するプロパティ群

デフォルト内容に関するプロパティ群は、内容が特に与えられなかったときのブロックへの流し込みの方法を指定します。これらはとりわけプレビュー機能で有用です。なぜならプレビューではブロックにへそのデフォルト内容を流し込むからです。表 11.12 に、デフォルト内容に関するプロパティの一覧を示します。

表 11.12 デフォルト内容に関するプロパティ一覧

キーワード	とりうる値・解説
defaultimage	（文字列。画像ブロックにのみ適用）置き換え画像がクライアントアプリケーションから与えられなかったときに用いられる画像のパス名。ファイル名には絶対パスを付けず、SearchPath 機能を利用するよう、PPS クライアントアプリケーションを作っておくほうがよいでしょう。そうすればブロック処理を、プラットフォームやファイルシステムの細かい違いから切り離すことができます。
defaultpdf	（文字列。PDF ブロックにのみ適用）置き換え PDF がクライアントアプリケーションから与えられなかったときに用いられる PDF 文書のパス名。ファイル名には絶対パスを付けず、SearchPath 機能を利用するよう、PPS クライアントアプリケーションを作っておくほうがよいでしょう。そうすればブロック処理を、プラットフォームやファイルシステムの細かい違いから切り離すことができます。
default-pdfpage	（整数。PDF ブロックにのみ適用）デフォルト PDF 文書内のページのページ番号。デフォルト：1
defaulttext	（文字列。テキスト行・テキストフローブロックにのみ適用）可変テキストがクライアントアプリケーションから与えられなかったときに用いられるテキスト[1]

1. テキストは winansi エンコーディングか Unicode で解釈されます。

11.6.8 カスタムプロパティ

カスタムプロパティは、すべての種類のブロックに適用されます。PPS とプレビュー機能からは無視されます。表 11.13 に、カスタムプロパティの命名規則を示します。

表 11.13 カスタムブロックプロパティ（すべての種類のブロックに適用）

キーワード	とりうる値・解説
3 種類のキャラクタ [ ] / を含まないあらゆる名前	（文字列・名前・float のいずれか、または float のリスト）各カスタムプロパティの値をどう解釈するかは、全くクライアントアプリケーションの領分です。PPS からは無視されます。

11.7 pCOS でブロック名とプロパティを取得

PPS による自動ブロック処理に加えて、内蔵の pCOS 機能を使うと、ブロック名を評価したり、標準・カスタムプロパティを取得したりすることができます。

クックブック **取り込んだPDFの中に含まれているブロックのプロパティを取得するための完全なコードサンプルがクックブックの** `blocks/query_block_properties` **トピックにあります。**

ブロックの数と名前の取得　クライアントコード側では、取り込んだページ上のブロックの名前も数も知らなくてかまいません。なぜなら取得することもできるからです。下記のステートメントは、ページ番号 ***pagenum*** のページ上のブロックの数を返します：

```
blockcount = (int) p.pcos_get_number(doc, "length:pages[" + pagenum + "]/blocks");
```

下記のステートメントは、ページ ***pagenum*** 上の ***blocknum*** 番目のブロックの名前を返します（ブロックとページの番号は 0 から始まります）：

```
blockname = p.pcos_get_string(doc,
        "pages[" + pagenum + "]/blocks[" + blocknum + "]/Name");
```

返されたブロック名はその後、ブロックのプロパティを取得したり、ブロックへテキスト・画像・PDF を流し込んだりするために利用することができます。指定されたブロックが存在しないときは、例外が発生します。これを避けるには、***length*** 接頭辞を用いて、ブロックの数を知り、ひいては ***Blocks*** 配列の最大添字を知ることができます（配列の添字が 0 から始まるため、ブロックの数は最大可能添字より 1 大きいことに留意してください）。

ブロックプロパティを特定するパス文法において、以下の表現は等価です。ここで、通し番号 **< 番号 >** のブロックが、その ***Name*** プロパティを **< ブロック名 >** に設定されているとします：

```
pages[...]/blocks[<番号>]
pages[...]/blocks/<ブロック名>
```

ブロックの座標の取得　名前 ***foo*** のブロックの左下隅と右上隅を記述する 2 個の座標ペア ***(llx, lly)*** および ***(urx, ury)*** は、以下のようにして取得できます：

```
llx = p.pcos_get_number(doc, "pages[" + pagenum + "]/blocks/foo/Rect[0]");
lly = p.pcos_get_number(doc, "pages[" + pagenum + "]/blocks/foo/Rect[1]");
urx = p.pcos_get_number(doc, "pages[" + pagenum + "]/blocks/foo/Rect[2]");
ury = p.pcos_get_number(doc, "pages[" + pagenum + "]/blocks/foo/Rect[3]");
```

上記の座標はデフォルトのユーザー座標系で与えられていることに注意してください（左下隅が原点。ただし、そのページの CropBox によって変更されている可能性もあります）。これに対して Block Plugin は、ページ左上隅に原点を持つ Acrobat のユーザーインタフェース座標系に従って座標を表示します。ブロックの座標に優先させるための ***Rect*** オプションは、CropBox エントリによって適用されているいかなる変更をも考慮には入れないので、元のブロックから取得された座標は、CropBox が存在する場合には新しい座標としてそのまま受け渡すことはできません。これを回避するには、***refpoint*** ・ ***boxsize*** オプションを用いることができます。

また、***topdown*** パラメタはブロック座標取得の際には考慮されないことにも留意してください。

カスタムプロパティの取得　カスタムプロパティは、下記の例のように取得することができます。ここでは、ページ *pagenum* 上の *b1* というブロックからプロパティ *zipcode* を取得しています：

```
zip = p.pcos_get_string(doc, "pages[" + pagenum + "]/blocks/b1/Custom/zipcode");
```

ブロック内に具体的に何というカスタムプロパティがあるかわからなければ、実行時にその名前を得ることもできます。*b1* というブロックの最初のカスタムプロパティの名前を得るには、下記のようにします：

```
propname = p.pcos_get_string(doc, "pages[" + pagenum + "]/blocks/b1/Custom[0].key");
```

番号を 0 のかわりに 1 つずつ増やしていけば、すべてのカスタムプロパティの名前を得ることができます。*length* 接頭辞を用いれば、カスタムプロパティの数を知ることができます。

存在しないブロックプロパティとデフォルト値　ブロックまたはプロパティが実在するかどうかを知るには、*type* 接頭辞を用います。パスに対する型が 0 か *null* ならば、そのオブジェクトは PDF 文書内に存在していません。なお、標準プロパティの場合、これはプロパティのデフォルト値が用いられることを意味します。

カスタムプロパティの名前空間　さまざまな場所で作成された PDF 文書をやり取りする際に混乱が生じることを避けるため、カスタムプロパティ名をつけるときには必ず、インターネットドメイン名を企業固有の接頭辞として用い、その後にコロン「:」とプロパティ名本体を続けることを推奨します。たとえば、ACME 社であれば以下のようなプロパティ名を使用するのです：

```
acme.com:digits
acme.com:refnumber
```

標準プロパティとカスタムプロパティはブロック内で異なる格納のされ方をしているので、標準 PPS プロパティ名（306 ページ「11.6 ブロックのプロパティ」で定義されているもの）がカスタムプロパティ名と衝突することは決してありません。

11.8 PDFlib ブロックの仕様

PDFlib ブロックの文法は、PDF ページを構成するデータ構造にアプリケーションが独自データを追加格納できるようにする拡張のしくみを定めた PDF Reference に完全準拠しています。ここでは、PDFlib ブロックの文法を詳述して、ブロックを Block Plugin 以外の方法で作成したいユーザーの助けとします。プラグインのユーザーはこの節は飛ばしてかまいません。

11.8.1 PDFlib ブロックの PDF オブジェクト構造

ページ辞書は *PieceInfo* 項目を含んでおり、この項目は値として別の辞書を持っています。この辞書はキー *PDFlib* を含んでおり、このキーはアプリケーションデータ辞書を値として持っています。このアプリケーションデータ辞書は、表 11.14 に挙げる 2 個の標準キーを含んでいます。

表 11.14 アプリケーションデータ辞書内項目一覧

キー	値
LastModified	（日付文字列、必須）ページ上のブロックが作成された、または最近更新された日時。
Private	（辞書、必須）ブロックリスト（表 11.15 参照）

ブロックリストとは、ブロック処理に関する一般的な情報に加えて、ページ上のすべてのブロックの一覧をも含んだ辞書です。表 11.15 に、ブロックリスト辞書の中のキーの一覧を示します。

表 11.15 ブロックリスト辞書内項目一覧

キー	値
Version	（数、必須）ファイルが準拠するブロック仕様のバージョン番号。本文書では、バージョン 9 のブロック仕様を解説しています。
Blocks	（辞書、必須）キーはそれぞれ、ブロック名の名前オブジェクト。対応する値はそれぞれ、そのブロックに対するブロック辞書です（表 11.17 参照）。ブロック辞書内の Name キーは、この辞書内のブロック名と同じでなければなりません。
PluginVersion	（文字列。pdfmark キーがないなら必須[1]）ブロックの作成に使われた Block Plugin のバージョン識別を表す文字列。
pdfmark	（論理値。PluginVersion キーがないなら必須[1]）ブロックリストが pdfmark を用いて生成されている場合は true でなければなりません。

1. PluginVersion キーと pdfmark キーは、どちらか一方しか存在できず、かつ、どちらかが必ず存在しなければなりません。

ブロックプロパティのデータ型　プロパティはオプションリストと同じデータ型に対応しています。ただしハンドルと、アクションリストのような特化されたリストには対応していません。表 11.16 に、これらの型と PDF のデータ型との対応づけを示します。

表 11.16 ブロックプロパティのデータ型一覧

データ型	PDF での型および注釈
論理値	（論理値）
文字列	（文字列）

表 11.16 ブロックプロパティのデータ型一覧

データ型	PDF での型および注釈
キーワード（名前）	（名前）そのプロパティが対応するキーワードのリストにないキーワードを与えるとエラー。
***float*・整数**	（数値）オプションリストでは点もカンマも小数点として対応しているのに対し、PDF の数値では点のみ対応。
パーセント値	（要素 2 個の配列）配列の 1 番目の要素は数、2 番目の要素はパーセントキャラクタを持った文字列。
リスト	（配列）
色	（要素 2 個か 3 個の配列）配列の 1 番目の要素は色空間を指定し、2 番目の要素は下記の色値を指定。配列の 1 番目の要素に対しては以下の項目が指定できます： ***/DeviceGray*** 2 番目の要素はグレー値 1 個。 ***/DeviceRGB*** 2 番目の要素は RGB 値 3 個の配列。 ***/DeviceCMYK*** 2 番目の要素は CMYK 値 4 個の配列。 ***[/Separation/ スポットカラー名]*** 1 番目の要素は、キーワード /Separation とスポットカラー名 1 個を持った配列。2 番目の要素は濃度値。 オプションとして 3 番目の要素で、スポットカラーの代替色を指定します。代替色はそれ自体が 1 個の色配列であり、/DeviceGray・/DeviceRGB・/DeviceCMYK・/Lab のいずれかの色空間で表されます。代替色を指定しないときは、スポットカラー名は、PPS が内部的に知っている色か、またはアプリケーションで動作時に定義しておいた色でなければなりません。 ***[/Lab]*** 1 番目の要素はキーワード /Lab を持った配列。2 番目の要素は Lab 値 3 個の配列。 色なしを指定するには、各プロパティを省略する必要があります。
unichar	（テキスト文字列）UTF-16 BOM U+FEFF で始まる utf16be 形式の Unicode 文字列

11.8.2 ブロック辞書のキー

ブロック辞書は、表 11.17 に挙げるキーを含むことができます。

表 11.17 ブロック辞書内項目一覧

プロパティグループ	値
管理プロパティ群	（いくつかのキーは必須）表 11.4 に従った管理プロパティ群
矩形プロパティ群	（いくつかのキーは必須）表 11.5 に従った矩形プロパティ群
書式プロパティ群	（いくつかのキーは必須）表 11.6 に従った、すべての種類のブロックに適用される書式プロパティ群と、表 11.7 に従った、テキスト行・テキストフローブロックに適用されるテキスト書式プロパティ群
テキスト作成プロパティ群	（オプション）表 11.8 に従った、テキスト行・テキストフローブロックに適用されるテキスト作成プロパティ群
テキスト組版プロパティ群	（オプション）表 11.9 に従った、テキスト行・テキストフローブロックに適用されるテキスト組版プロパティ群
オブジェクトはめ込みプロパティ群	（オプション）表 11.10 に従った、テキスト行・画像・PDF ブロックに適用されるオブジェクトはめ込みプロパティ群と、表 11.11 に従った、テキストフローブロックに適用されるはめ込みプロパティ群
デフォルト内容に関するプロパティ群	（オプション）表 11.12 に従った、デフォルト内容に関するプロパティ群
カスタム	（辞書、オプション）表 11.13 に従った、カスタムプロパティに対するキー・値の対を含んだ辞書

作成例　以下に示す PDF コードは、*job_title* というテキストブロックと、*logo* という画像ブロックの、2 個のブロックを扱っています。テキストブロックには *format* というカスタムプロパティが含まれています：

```
<<
        /Contents 12 0 R
        /Type /Page
        /Parent 1 0 R
        /MediaBox [ 0 0 595 842 ]
        /PieceInfo << /PDFlib 13 0 R >>
>>

13 0 obj
<<
        /Private <<
            /Blocks <<
                /job_title 14 0 R
                /logo 15 0 R
            >>
            /Version 9
            /PluginVersion (4.1)
        >>
        /LastModified (D:20110813200730)
>>
endobj

14 0 obj
<<
```

```
        /Type /Block
        /Rect [ 70 740 200 800 ]
        /Name /job_title
        /Subtype /Text
        /fitmethod /auto
        /fontname (Helvetica)
        /fontsize 12
        /Custom << /format 5 >>
>>
endobj

15 0 obj
<<
        /Type /Block
        /Rect [ 250 700 400 800 ]
        /Name /logo
        /Subtype /Image
        /fitmethod /auto
>>
```

11.8.3 pdfmark で PDFlib ブロックを生成

このプラグインでブロックを作成するのではなく、PostScript ストリームの中に適切な *pdfmark* コマンドを挿入して、それを Distiller で PDF に変換することによってブロックを作成することも可能です。*pdfmark* オペレータについては詳しくは Acrobat の解説文書に記載されています。以下に示す *pdfmark* オペレータ群を用いれば、前節と同じブロック定義を生成することができます：

```
% ---------- Setup for the Blocks on a page ----------
[/_objdef {B1} /type /dict /OBJ pdfmark         % Blocks dict

[{ThisPage} <<
        /PieceInfo <<
            /PDFlib <<
                /LastModified (D:20110813200730)
                /Private <<
                    /Version 9
                    /pdfmark true
                    /Blocks {B1}
                >>
            >>
        >>
>> /PUT pdfmark

% ---------- text Block ----------
[{B1} <<
        /job_title <<
            /Type /Block
            /Name /job_title
            /Subtype /Text
            /Rect [ 70 740 200 800 ]
            /fitmethod /auto
            /fontsize 12
            /fontname (Helvetica)
            /Custom << /format 5 >>
        >>
>> /PUT pdfmark
```

```
% ---------- image Block ----------
[{B1} <<
        /logo <<
            /Type /Block
            /Name /logo
            /Subtype /Image
            /Rect [ 250 700 400 800 ]
            /fitmethod /auto
        >>
>> /PUT pdfmark
```

A 改訂履歴

日付	更新点
2011 年 7 月 11 日	▸ PDFlib 8.0.3 に関するさまざまな更新・修正
2010 年 12 月 09 日	▸ PDFlib 8.0.2 に関するさまざまな更新・修正
2010 年 9 月 22 日	▸ PDFlib 8.0.1p7 に関するさまざまな更新・修正
2010 年 4 月 13 日	▸ PDFlib 8.0.1 に関するさまざまな更新・修正
2009 年 12 月 07 日	▸ PDFlib 8.0.0 に関する更新
2009 年 3 月 13 日	▸ PDFlib 7.0.4 に関するさまざまな更新・修正
2008 年 2 月 13 日	▸ PDFlib 7.0.3 に関するさまざまな更新・修正
2007 年 8 月 08 日	▸ PDFlib 7.0.2 に関するさまざまな更新・修正
2007 年 2 月 19 日	▸ PDFlib 7.0.1 に関するさまざまな更新・修正
2006 年 10 月 03 日	▸ PDFlib 7.0.0 に関する更新と再構成
2006 年 2 月 21 日	▸ PDFlib 6.0.3 に関するさまざまな更新・修正。Ruby の節を追加
2005 年 8 月 09 日	▸ PDFlib 6.0.2 に関するさまざまな更新・修正
2004 年 11 月 17 日	▸ PDFlib 6.0.1 に関する小規模な更新・修正 ▸ 8 章に言語別関数プロトタイプ用新書式導入 ▸ 3 章にハイパーテキストの例を追加
2004 年 6 月 18 日	▸ PDFlib 6 に関する大規模な変更
2004 年 1 月 21 日	▸ PDFlib 5.0.3 に関する小規模な追加・修正
2003 年 9 月 15 日	▸ PDFlib 5.0.2 に関する小規模な追加・修正。ブロックの仕様を追加
2003 年 5 月 26 日	▸ PDFlib 5.0.1 に関する小規模な更新・修正
2003 年 3 月 26 日	▸ PDFlib 5.0.0 に関する全面的変更と書き直し
2002 年 6 月 14 日	▸ PDFlib 4.0.3 に関する小規模な変更と .NET バインディングに関する追加
2002 年 1 月 26 日	▸ PDFlib 4.0.2 に関する小規模な変更と IBM eServer エディションに関する追加
2001 年 5 月 17 日	▸ PDFlib 4.0.1 に関する小規模な変更
2001 年 4 月 1 日	▸ PDFlib 4.0.0 の PDI・他機能を解説
2001 年 2 月 5 日	▸ PDFlib 3.5.0 のテンプレート・CMYK 機能を解説
2000 年 12 月 22 日	▸ ColdFusion 解説と PDFlib 3.03 に関する追加。COM エディションを別マニュアルに
2000 年 8 月 8 日	▸ Delphi 解説と PDFlib 3.02 に関する小規模な追加
2000 年 7 月 1 日	▸ PDFlib 3.01 に関する追加・説明の明瞭化
2000 年 2 月 20 日	▸ PDFlib 3.0 に関する変更
1999 年 8 月 2 日	▸ PDFlib 2.01 に関する小規模な変更・追加

日付	更新点
1999 年 6 月 29 日	► 言語バインディングごとに節を分離 ► PDFlib 2.0 に関する追加
1999 年 2 月 1 日	► PDFlib 1.0 に関する小規模な追加（公開せず）
1998 年 8 月 10 日	► PDFlib 0.7 に関する追加（一つのお客様用のみ）
1998 年 7 月 8 日	► PDFlib 0.6 の PDFlib スクリプティングサポートを初めて記述
1998 年 2 月 25 日	► PDFlib 0.5 に関して説明を若干追加
1997 年 9 月 22 日	► PDFlib 0.4 と本マニュアルを初めて公開

索引

PDFlib GmbH

PDFlib GmbH
Franziska-Bilek-Weg 9
80339 München, Germany
www.pdflib.com
電話 +49 • 89 • 452 33 84-0
FAX +49 • 89 • 452 33 84-99

ご質問があるときは、PDFlib メーリングリストと
tech.groups.yahoo.com/group/pdflib にあるアーカイブをチェックしてください

ライセンス発行のお問い合わせ
sales@pdflib.com

サポート
support@pdflib.com（お持ちのライセンス番号をお書きください）