PDFlib GmbH

PDFlib, PDFlib+PDI, PPS

A library for generating PDF on the fly

PDFlib 9.3.1

チュートリアル

C・C++・Java・.NET・.NET Core・Objective-C・
Perl・PHP・Python・RPG・Ruby 用

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

jp.sales@pdflib.com
jp.support@pdflib.com（ライセンス番号をお知らせください）

目次

o PDFlib ライセンスキーを適用

評価版の制約　PDFlib GmbH によって提供される PDFlib・PDFlib+PDI・PPS のすべてのバイナリバージョンは、商用ライセンスを取得したか否かにかかわらず、完全に動作する評価版として利用できます。ただし非ライセンス版は、すべての生成されるページ上に、*www.pdflib.com* というデモスタンプを横断印字し、また、内蔵の pCOS インタフェースは小容量の文書（10 ページ以下、ファイルサイズ 1 MB 以下）に制限されます。非ライセンスのバイナリは、業務目的に使用してはならず、この製品を評価するためにのみ使用できます。PDFlib GmbH 製品はいずれも、業務目的に使用するには有効なライセンスが必要です。

PDFlib ライセンスの取得を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（評価キーの提供要請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します。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。このインストーラは、そのライセンスキーをレジストリに追加します（後述）。

ライセンスキーを API 呼び出しで実行時に適用　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（C の場合：*PDF_new()* の後に）、*license* オプション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++・Java・.NET/C#・Python・Ruby の場合：

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

- C の場合：

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

- Objective-C の場合：

```
[pdflib set_option: @"license=...あなたのライセンスキー ..."];
```

- Perl・PHP の場合：

```
$p->set_option("license=...あなたのライセンスキー ...")
```

- RPG の場合：

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

ライセンスファイルを使用　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、その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。以下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、それを利用することもできます）。1 個の「#」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      9.3.1   ...あなたのライセンスキー ...
```

この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、以下の方法で構成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（13 ページ「デフォルトファイル検索パス」を参照）。
- *licensefile* オプションを *set_option()* API 関数で設定することもできます：

  ```
  p.set_option("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 System i では、ライセンスファイルは ASCII で符号化されている必要があります（*asciifile* オプションを参照）。ライセンスファイルを以下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('/PDFlib/9.3/licensefile.txt') LEVEL(*SYS)
  ```

レジストリ内へライセンスキー　Windows では、以下のレジストリ値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、以下のレジストリ値のうちのいずれか 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\PDFlib9\license
HKLM\SOFTWARE\PDFlib\PDFlib9\9.3.1\license
```

インストーラは、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には注意してください：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」ダイアログから以下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・macOS・IBM System i 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、以下のディレクトリが検索されます：

```
<rootpath>/PDFlib/PDFlib/9.3/resource/cmap
<rootpath>/PDFlib/PDFlib/9.3/resource/codelist
<rootpath>/PDFlib/PDFlib/9.3/resource/glyphlst
<rootpath>/PDFlib/PDFlib/9.3/resource/fonts
<rootpath>/PDFlib/PDFlib/9.3/resource/icc
<rootpath>/PDFlib/PDFlib/9.3
<rootpath>/PDFlib/PDFlib
<rootpath>/PDFlib
```

Unix・Linux・macOS 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。IBM System i では ***<rootpath>*** は空です。

ライセンスファイルとリソースファイルに対するデフォルトファイル名　デフォルトで、以下のファイル名が、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。

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）かアップグレード（PDFlib から PDFlib+PDI または PPS への、または PDFlib+PDI から PPS への切り換え）を購入された場合、あるいはご自分のサポート契約の一部として新しいライセンスキーを受け取った場合には、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利用できなくなります。

まだライセンスされていない機能を評価　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できます。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 の機能を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（ブロック関数群）を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

スの機能を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。

さまざまなライセンシングオプション　PDFlib の 1 台ないし複数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シングオプションを利用可能です。サポート契約やソースコード契約も提供しています。ライセンシング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。PDFlib ライセンスの入手にご関心をお持ちいただけた場合や、ご質問につきまして何なりと、当社までご連絡ください。

1 導入

1.1 各種文書とサンプルへのロードマップ

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。

すべての言語バインディング用のミニサンプル　*hello*・*image* ミニサンプルが、すべてのパッケージに入っており、すべての言語バインディングで利用可能です。最小限のコードで、テキスト出力・画像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どうかを試したり、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・PDF/VT・PDF/UA 作成、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（本マニュアル）は、すべてのパッケージに入っている 1 個の PDF 文書であり、重要なさまざまなプログラミング概念を詳しく説明しており、各種の小さなコード断片も含んでいます。コードをスタータサンプルよりも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 この 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。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、すべての言語バインディングについて同じです。

PDFlib API リファレンス　*PDFlib API* リファレンスは、すべてのパッケージに入っている 1 個の PDF 文書であり、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・オプションを簡明に記述しています。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 API 一覧は不完全です。PDFlib で作業する際にはかならず、この完全な PDFlib API リファレンスを利用してください。

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クエリすることができます。pCOS は PDFlib+PDI・PPS に内蔵されています。この 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の照応する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、さまざまな課題を達成するための PDFlib コーディング断片を数百個集めたものです。クックブックのトピック群は Java・PHP 用ですが、簡単に他のプログラミング言語に合わせて変えることができます。なぜなら PDFlib API は、対応するすべての言語バインディングについて等価だからです。この PDFlib クックブックは、以下の URL で入手可能です：

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断片を集めています。この 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クエリすることができます。以下の URL で入手可能です：

www.pdflib.com/pcos-cookbook/

TET クックブック　PDFlib TET（Text and Image Extraction Toolkit ＝テキスト抽出ツールキット）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 のためのコード断片を集めています。これは、TET と PDFlib+PDI の組み合わせを演示するサンプルのグループを含んでいます。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、目次を作成したりなどです。この TET クックブックは以下の URL で入手可能です：

www.pdflib.com/tet-cookbook/

1.2 PDFlib プログラミング

PDFlib とは　PDFlib は、Portable Document Format（PDF）形式のファイルを生成することを可能にする開発コンポーネントです。PDFlib は、あなた自身の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。そのディストリビューションパッケージは、さまざまな製品を 1 個のバイナリに含んでいます：

- PDFlib：テキスト・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、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 オブジェクトをクエリする（ページ上のすべてのフォントを列挙したり、メタデータをクエリしたり、その他さまざまなことをする）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（OS X/macOS 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 を使うには　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・macOS のいずれでも利用することができ、また、IBM System i・IBM Z 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。こうした言語や環境を言語バインディングといいます。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網羅しています。その API（アプリケーションプログラミングインタフェース）は学習が容易であり、かつ、すべてのバインディングについて同等です。現在、以下のバインディングに対応しています：

- C・C++
- Java
- .NET・.NET Core
- Objective-C
- Perl
- PHP
- Python
- RPG（IBM System i）
- Ruby

PDFlib の使い道　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

取得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。この PDFlib のアプローチはいくつかの利点を提供します：

- PDFlib を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・エラー処理が、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 をさまざまなオペレーティングシステム・開発環境で利用できます。

PDFlib を使うための要件　PDFlib を使えば、PDF の仕様にわずらわされずに 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越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、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 9.0 の新機能

PDFlib/PDFlib+PDI/PPS 9.0 と Block Plugin 5 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。詳しくは表 1.1 と PDFlib API リファレンスを参照してください。

PDF/A-2・PDF/A-3 を作成　PDFlib は、アーカイビングのための PDF/A 規格の 2 つの新たな部分に対応しました。PDF/A-2 は、PDF 1.7 に基づいており、透過・JPEG 2000 圧縮・レイヤーなど多くの機能をサポートしています。PDF/A-2 では PDF/A-1・PDF/A-2 文書の埋め込みが可能であり、PDF/A-3 では任意のファイル形式の埋め込みが可能です。

タグ付き PDF と PDF/UA を作成　タグ付き PDF の作成が、短縮タグ付けやページ装飾の自動タグ付けといったさまざまな簡便機能により、ずっと容易になっています。PDFlib の表組版機能が、自動的に表組みにタグ付けします。構造エレメントを含むタグ付き PDF 文書を PDI で取り込めます。

アクセシブル文書を、PDF/UA 規格（Universal Accessibility ＝ユニバーサルアクセシビリティ）に従って作成できます。PDF/UA は、PDF 1.7 に基づいており、タグ付き PDF をアクセシビリティのために改良したものです。

PDF/VT を作成　PDF/VT は、可変・トランザクション印刷に最適化された PDF のための規格です。PDFlib は、可変文書印刷（Variable Document Printing ＝ VDP）のための ISO 16612-2 に従った出力を作成できます。文書部分メタデータ（Document Part Metadata ＝ DPM）を、PDF/VT 規格に従って添付できます。

スケーラブルベクトルグラフィック（Scalable Vector Graphics ＝ SVG）を取り込む
PDFlib が、SVG 形式のベクトルグラフィックを取り込みます。SVG は、Web 上でのベクトルグラフィックのための標準規格です。

フォント処理とテキスト出力　PDFlib のフォントエンジンとテキスト処理が、いくつかの面で改良されています：

- 日中韓異体字グリフのための表意文字異体字シーケンス（ideographic variation sequences ＝ IVS）
- WOFF フォント（Web Open Font Format ＝ Web オープンフォント形式）、すなわち W3C が定めた、TrueType・OpenType フォントのための新たなコンテナ形式
- SVG フォント、すなわち SVG 形式で表されたベクトルフォント
- CEF フォント（Compact Embedded Font ＝コンパクト埋め込みフォント）、すなわち SVG グラフィック内にフォントを埋め込むために用いられる OpenType の変種
- 任意の数のディレクトリ内で見つかったすべてのフォントについて UPR フォント構成ファイルを自動的に作成

PDFlib+PDI で PDF 文書を取り込む　PDF 取り込みライブラリ PDI には以下の新機能があります：

- 構造エレメントを含むタグ付き PDF 文書を取り込めます。
- レイヤー定義を取り込めます。

PDFlib Personalization Server（PPS）と Block Plugin　PPS には以下の新機能があります：

- 新たなブロック種別**「グラフィック」**を用いて、PDFlib ブロックに SVG グラフィックを流し込めます。
- PDFlibブロックにPPSで流し込みを行えるだけでなく、PDFlibブロックを出力PDFへ取り込むこともできます。
- 新たなブロックプロパティがいくつか導入されています。

PDFlib ブロックをプログラム的に作成　PDFlib ブロックを PDFlib Block Plugin で対話的に作成できるだけでなく、PDFlib ブロックを PPS でプログラム的に作成することもできます。取り込んだ文書の中の既存の PDFlib ブロックを、生成 PDF 出力へコピーすることもできます。これらの機能は、PPS のためのテンプレート自体をプログラム的に組み立てる高度な文書構成ワークフローを可能にします。

PDF オブジェクト作成 API (PDF Object Creation API ＝ POCA)　POCA は、生成 PDF 出力内に含まれる低レベル PDF オブジェクトを作成するための手段の集合を提供します。POCA は以下の目的のために利用できます：
- PDF/VT のための文書部分メタデータ（Document Part Metadata ＝ DPM）を作成
- PPS で使用するための PDFlib ブロックをプログラム的に作成

マルチメディアコンテンツを埋め込む　PDFlib は、音声・映像・3D コンテンツを持つリッチメディア注釈を作成できます。このマルチメディアコンテンツを、JavaScript と PDF アクションで制御できます。以下の新たなマルチメディア機能があります：
- リッチメディア注釈
- リッチメディア実行アクション

暗号化アルゴリズムの向上　PDFlib は、Acrobat X/XI/DC に従った PDF 文書暗号化に対応しました。この暗号化方式は、AES-256 に基づいており、PDF 1.7 Adobe 拡張レベル 8 と、ISO 32000-2 に基づく PDF 2.0 で定められています。

その他の改良　以下の改良が実装されています：
- 表・テキストフロー組版機能の改良
- 幾何図形からパスオブジェクトを作成するための便利関数群
- JPEG 2000 ラスタ画像を取り込むためのサポートを向上
- PDFlib 仮想ファイルシステム（PDFlib Virtual Filesystem ＝ PVF）内のファイルの詳細をクエリ
- 関数スコープに関する多くの制約を除去。たとえば、ページ・パターン・テンプレートを任意にネストできるようになりました。

1.4 PDFlib/PDFlib+PDI/PPS 9.1 の新機能

PDFlib/PDFlib+PDI/PPS 9.1 では、カラー処理に関連するさまざまな新機能が導入されました：

- 任意の数のインキを有する *DeviceN・NChannel* 色空間に対応
- PDF/X-5n：n 色印刷ファイルの受け渡しに用いられます。パッケージング業界などで利用されます。
- ICC プロファイル、スポット・DeviceN カラー、およびグレー／ RGB ／ CMYK デバイスカラーのための SVG カラー拡張：印刷における SVG の有用性を向上させます。
- Pantone Extended Gamut Coated（XGC）スポットカラー群、および Pantone Plus 2016 アップデート
- 任意の数のストップカラーを用いたカラーシェーディング：柔軟なカラーブレンドが可能になります。
- 複数のスポットカラーの間のカラーシェーディング：Pantone カラーどうしのブレンドなどが可能になります。
- パターン・テンプレート・Type 3 フォントグリフのためのデフォルト色空間
- PDFlib チュートリアルとクックブックにおいてカラー関連のトピックの扱いを拡大

PDFlib/PDFlib+PDI/PPS 9.1 では、いくつかの言語バインディングへのサポートもアップデートされました。

1.5 PDFlib/PDFlib+PDI/PPS 9.2 の新機能

PDFlib/PDFlib+PDI/PPS 9.2 は、以下を含む、多くのバグフィックスと改良を含んでいます：

- 多くの言語バインディングへのアップデート
- 新たに .NET Core のための言語バインディング
- PDF 2.0 を見越した構造エレメントネスト化規則を明確化
- PDF/UA-1 の実装を最新の勧告とバリデータに整合させました
- タグ付き PDF ページの取り込みを改良
- ラスタ画像の色を変更するための新たなオプション群（*chromakey・decode*）
- SVG 内の非 sRGB カラーに対するカラー制御を改良
- 等しい CMYK プロファイルとの衝突を処理するための PDF/X-4/5 便利機能
- いくつかの非標準な種類の JPEG を識別
- フォーム XObject の PDF/VT カプセル化を、RIP パフォーマンス向上のために改良
- TrueType フォントのサブセット化を最適化したことにより、出力ファイルサイズを劇的に削減。とりわけ、使われていないグリフを多数含むフォントで顕著です
- 廃止の API 関数を特定。C・C++・.NET・Java ではコンパイル時に、Perl・PHP では実行時に行わせることができます（PDFlib Migration Guide 参照）
- すべての言語バインディングのためのコーディングサンプル群を見直し
- PDFlib クックブック内のサンプルアプリケーション群をアップデート・拡張
- 埋め込みシステム用の PDFlib Mini Edition（ME）でメモリ必要量を削減

1.6 PDFlib/PDFlib+PDI/PPS 9.3 の新機能

- PDF/A でフィールドを使用するための要請として、フォームフィールド書式を生成

- タグ付きPDFを取り込む際、ある種のまれな構造に対してPDF/UAバリテータがエラーを発していたのを、正しく扱えるよう改良
- PDF 協会発行『Tagged PDF Best Practice Guide』（タグ付き PDF ベストプラクティスガイド）に従って、タグ付き PDF と PDF/UA-1 を向上
- 将来の削除を見据え、廃止のAPI機能をすべて洗い出し（詳しくはPDFlib Migration Guide（PDFlib マイグレーションガイド）を参照してください）
- 各種オペレーティングシステム・開発環境の新バージョンに合わせた調整
- PDFlib クックブック内の各種サンプルアプリケーションを更新・拡張
- 大部分の言語バインディングを更新
- 多数の領域でのバグフィックスと改善
- Block Plugin（ブロックプラグイン）を Acrobat の最近のバージョンに合わせて調整するとともに、いくつかのバグフィックスを実装

1.7 PDFlib の諸機能

表 1.1 に、PDF を生成するための機能を挙げます。新機能・改良機能には脚注を付しています。

表 1.1 PDFlib の機能一覧

分類	機能
各種 PDF	PDF 1.4 ～ PDF 1.7 拡張レベル 8・PDF 2.0
	線形化（Web 最適化）PDF による、Web 上でのバイトサービング
	大容量出力、任意の PDF ファイルサイズ（10 GB 超も可）
PDF のための ISO 規格群	ISO 32000-1：PDF 1.7 の標準化バージョン ISO 32000-2：PDF 2.0（日付入りバージョン ISO 32000-2:2020 を含む） ISO 15930：PDF/X-3/4/5。グラフィックアート業界向け ISO 19005-1/2/3：PDF/A-1/2/3。アーカイビング用 ISO 16612-2：PDF/VT-1。可変・トランザクション印刷用 ISO 14289-1：PDF/UA-1。ユニバーサルアクセシビリティ用
フォント	TrueType（TTF・TTC）・PostScript Type 1 フォント
	PostScript か TrueType のアウトラインを持つ OpenType フォント（TTF・OTF・OTC）
	WOFF フォント（Web Open Font Format ＝ Web オープンフォント形式）
	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・スモールキャップス・オールドタイプ数字・スワッシュキャラクタ・簡体 / 繁体・縦書き字体
	Windows か macOS システムにインストールされているフォントにアクセス
	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 フォント（グリフレット）による日中韓外字キャラクタ
	フォールバックフォント（足りないグリフを他のフォントから使用）
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・グリフ名のいずれかで指定可能
	カーニングによる文字間隔の改善
	テキストを太字化・斜体化・影付き
	パス上テキスト
	見つからないグリフの代替を構成可能
アクセシビリティ	タグ付き PDF の作成によるアクセシビリティ
	注釈・フォームフィールド等インタラクティブ要素のタグ付け
	表・ページ装飾の自動タグ付け
	ユニバーサルアクセシビリティのための PDF/UA-1
	構造エレメント種別・属性の追加
国際化	Unicode 完全対応

表 1.1 PDFlib の機能一覧

分類	機能
	日本語・中国語・韓国語テキストのための日中韓フォントと CMap
	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
	日中韓異体字グリフのための表意文字異体字シーケンス（IVS）
	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
	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
SVG ベクトルグラフィック	SVG 形式のベクトル画像を取り込み。SVG・CSS 内の ICC プロファイル、CMYK・スポットカラー
画像	BMP・GIF・PNG・TIFF・JBIG2・JPEG・JPEG 2000・CCITT ラスタ画像を読み込み
	画像情報をクエリ（ピクセル寸法・解像度・ICC プロファイル・クリッピングパスなど）
	TIFF・JPEG 画像内のクリッピングパスを使用
	TIFF・PNG 画像内のアルファチャンネル（透過）を使用
	画像マスク（透過画像に色を適用）、スポットまたは DeviceN カラーを用いて画像に着色
色	グレースケール・RGB（数値・16 進・HTML カラー名）・CMYK・CIE $L^*a^*b^*$ カラー
	PANTONE®（PANTONE+ も）・HKS® カラー対応内蔵
	プロセスまたはスポットカラーに基づく DeviceN（n 色）色空間
	ユーザー定義スポットカラー
	複数のプロセスカラーまたはスポットカラーの間のカラーシェーディング（スムーズシェーディング）。パターン塗り・描線
カラーマネジメント	ICC プロファイルによる ICC ベースカラー
	テキスト・図形・ラスタ画像のレンダリングインテント
	PDF/A・PDF/X のための出力インテントとしての ICC プロファイル。PDF/X-5n に対する多インキプロファイル
アーカイビング	PDF/A-1a/1b・PDF/A-2a/b/u・PDF/A-3a/b/u
	PDF/A のための XMP 拡張スキーマ
グラフィックアート	PDF/X-3・PDF/X-4・PDF/X-4p・PDF/X-5n
	埋め込まれた、または外部参照された出力インテント ICC プロファイル
	PDF/X-5p・PDF/X-5pg のための外部グラフィカル内容（参照されたページ）
	オーバープリント・テキストノックアウト
可変文書印刷（VDP）	可変・トランザクション印刷のための PDF/VT-1
テキストフロー組版	テキストを、1 個ないし複数の長方形内に、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）、フォントと 色の変更、揃え方式、タブ、リーダを指定可能
	言語固有処理で高度な改行

表 1.1 PDFlib の機能一覧

分類	機能
	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・表列の寸法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、一行か複数行テキスト、画像、SVG グラフィック、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
ベクトルグラフィック	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・長方形等
	透過（不透明）・ブレンドモード
	再利用可能なパスオブジェクト、クリッピングパスを画像から取り込み
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
セキュリティ	PDF 文書または添付を暗号化
	Unicode パスワード
	印刷不可・コピー不可等、文書権限設定の指定
インタラクティブ要素	フォームフィールドを作成。すべてのフィールドオプションと JavaScript を設定可能
	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を作成
	しおりを作成。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、PDF 注釈（コメント）を作成
	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）を作成
マルチメディア	PDF 内に 3D アニメーションを埋め込み
	音声・映像を PDF 内に埋め込み、それを JavaScript で制御
地理参照付き PDF	地理空間参照情報を持つ PDF を作成
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
	文書情報フィールドか XMP ストリームから XMP メタデータを作成
	ユーザーが与えるカスタム XMP メタデータ
	TIFF・JPEG・JPEG 2000・SVG グラフィック内の XMP 画像メタデータを処理
プログラミング	C・C++・Java・.NET・.NET Core・Objective-C・Perl・PHP・Python・RPG・Ruby 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）

表 1.1 PDFlib の機能一覧

分類	機能
	PDF 文書をディスクファイルか直接メモリ上へ生成
埋め込みシステム	埋め込みシステムのための、必要リソースを削減した PDFlib Mini Edition（ME）

1.8 PDFlib+PDI で加わる諸機能

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

分類	機能
PDF 入力（PDI）	既存 PDF 文書からページを取り込む
	PDF 1.7 拡張レベル 8（Acrobat X/XI/DC）・PDF 2.0 までのすべてのバージョンの PDF を取り込み可能
	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
	取り込まれるページに関する情報をクエリ
	取り込まれるページのページ寸法を転写（BleedBox・TrimBox・CropBox 等）
	複数の取り込まれた 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
	取り込まれた PDF 文書から PDF/A・PDF/X 出力インテントをコピー
	構造エレメント群を含むタグ付き PDF 文書のページ群を取り込み
	レイヤー定義（オプショナルな内容）を取り込む
pCOS インタフェース	pCOS インタフェースで、取り込まれる PDF 文書の詳細をクエリ

1.9 PPS で加わる諸機能

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

分類	機能
可変文書印刷（VDP）	PDFlib ブロックにテキスト・画像・PDF データ・SVG ベクトルグラフィックを流し込んで PDF をパーソナライズ
	PPS で PDFlib ブロックをプログラム的に作成
	取り込まれた文書から PDFlib ブロックをコピー
PDFlib Block Plugin	PDFlib Block Plugin で、Windows・macOS 版 Acrobat 上で PDFlib ブロックを対話的に作成
	PPS ブロック流し込みを Acrobat 上でプレビュー
	ブロックをプレビューファイルへコピー
	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
	ブロックコンテナの PDF/X・PDF/A プロパティを転写
	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® スポットカラー名を内蔵
	macOS 上で Retina ディスプレイに対応

1.10 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_begin_document( ) の linearize オプション	○[1]	○	○
PDF を最適化（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]	○	○
PDF/A 準拠のために添付を解析	PDF/A-2 モードにおいて PDF_load_asset( ) で type=Attachment を使用	○[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	—	○	○
pCOS を用いて PDF から情報をクエリ	すべての pCOS 関数	—	○	○
ブロックに可変データを流し込み	PDF_fill_*block( )	—	—	○
ブロックをプログラム的に作成	PDF_poca_new( ) : オプション usage=blocks PDF_begin/end_page_ext( ) : オプション blocks	—	—	○
ブロックを生成出力へコピー	PDF_process_pdi( ) : オプション action=copyblock または action=copyallblocks	—	—	○
PPS で利用するための PDFlib ブロックを対話的に作成	Acrobat 用 PDFlib Block Plugin	—	—	○

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 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++ リンカでリンクを行う必要があります。なぜなら、PDFlib 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。

C におけるエラー処理　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以下のようになります：

```
PDF_TRY(p)
{
        ...PDFlib命令群...
}
PDF_CATCH(p)
{
        printf("PDFlib例外がhelloサンプル内で発生しました:\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。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切ではありませんので、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 ブロックをネストして例外を再 throw *PDF_TRY()* は、任意の深さにネストすることが可能です。エラー処理をネストした場合、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()* ブロックの先頭ステートメントへ移ります。

try ブロックを途中で抜ける *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、すなわちそれに照応する *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);
}
```

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。

Linux で静的ライブラリから共有ライブラリを作成するには、以下のコマンドを利用します：

```
mkdir tmp
cd tmp
ar x ../libpdf.a
g++ -shared -o libpdf.so *
```

これで共有ライブラリができますので、アプリケーションを C++ ランタイムライブラリにリンクする必要はもうありません。

2.2 C++ バインディング

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、照応する *.cpp* モジュールは必要ありません。

C++ における文字列処理　このテンプレートベースの C++ バインディングは、文字列処理に関して以下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 以降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（*pstring_utf8.cpp*）に含まれています。

この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。

注 EBCDIC ベースのシステム上では、wstring ベースのインタフェースのためのオプションリスト文字列の整形は、オプションリスト内での EBCDIC と UTF-16 wstring の混在を避けるために、追加の変換が必要になります。この変換のための便利コードと説明が、補足モジュール `utf16num_ebcdic.hpp` 内にあります。ユーザーは、C++ コード内の文字列リテラルの Unicode への変換を制御するコンパイラオプション `CONVLIT(,UNICODE)` を調べてみるべきです。

C++ におけるエラー処理　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 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。以下のコード断片は、PDFlib が発生させた例外をキャッチします：

```
try {
        ...PDFlib命令群...
catch (PDFlib::Exception &ex) {
        wcerr << L"PDFlib例外がhelloサンプル内で発生しました: " << endl
              << L"[" << ex.get_errnum() << L"] " << ex.get_apiname()
```

```
            << L": " << ex.get_errmsg() << endl;
}
```

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（33 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、以下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。

2.3 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 Java 版をインストール　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 は以下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、クラス ***pdflib・PDFlibException*** を内容としています。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=. hello
```

このプロパティの値をチェックするには以下のようにします：

```
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：***libpdflib_java.so*** ライブラリ（macOS の場合：***libpdflib_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：***pdflib_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。

Java におけるエラー処理　Java バインディングは、PDFlib のエラーをネイティブ Java 例外に翻訳します。例外が起きると PDFlib は、以下のクラスのネイティブ Java 例外を発生させます：

```
PDFlibException
```

Java 例外は通常の try/catch 技法で処理できます：

```
try {
```

```
...PDFlib命令群...

} catch (PDFlibException e) {
            System.err.print("PDFlib例外がhelloサンプル内で発生しました:\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・レガシエンコーディング変換　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。以下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 文字列を作成します：

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

以下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 文字列を作成します：

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

以下の 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 で設定するには以下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。

2.4 .NET バインディング

2.4.1 .NET バインディングの種類

PDFlib の .NET 用バインディングには 2 つの種類があります：

- C# Interop に基づく .NET Core バインディング
- C++ Interop に基づくクラシック .NET バインディング

これらの .NET バインディングは、表 2.1 に示すとおり、実装の詳細と、対応しているターゲット環境が異なっています。この情報に基づいて、自分のアプリケーションに適しているほうのバインディングを選ぶことができます。

表 2.1 クラシック .NET バインディングと .NET Core バインディングの比較

	C++ Interop に基づくクラシック .NET バインディング	C# Interop に基づく .NET Core バインディング
ダウンロードパッケージ	Windows インストーラ	プラットフォーム独自の zip または tar.gz パッケージ
パッケージ内容	アセンブリ・文書・サンプル	アセンブリ・文書・サンプルを含む NuGet パッケージ
実装	アンマネージドコードを用いた C++/CLI アセンブリ pdflib_dotnet.dll	マネージドコードを用いた C# アセンブリ PDFlib_dotnet.dll と、非マネージドコードを用いた補助 DLL PDFlib_dotnetcore_native.dll
.NET 統合	暗黙の PInvoke を通じた C++ Interop	明示的な PInvoke を通じた C# Interop
.NET Framework 対応	.NET Framework 4.x	.NET Framework 4.6.1 以上
.NET Core 対応	該当せず	.NET Standard 2.0
ターゲットオペレーティングシステム	Windows x86・x64	Windows x64・Linux x64・macOS・Alpine Linux x64
Windows のレジストリの扱い	インストーラが PDFlib を登録してライセンスキーをレジストリに追加	PDFlib の登録は必要なし。ライセンスキーのためのレジストリキーは手作業で追加する必要あり
クラス名	PDFlib_dotnet	PDFlib_dotnet
廃止の PDFlib 4 ～ 8 の API メソッドへの対応	あり	なし

2.4.2 .NET Core バインディング

注 この .NET Core バインディングは、対応しているすべてのプラットフォームのための汎用のパッケージとして提供されます。ただし、それでも、プラットフォームごとのライセンスが必要であり、このライセンスを他のプラットフォームへ転用することはできません。

.NET Core 用の PDFlib は .NET Standard 2.0 に対応しています。すなわち、.NET Core 2.0・.NET Framework 4.6.1・Mono 5.4 と、それぞれのもっと新しいバージョン群、その他多数の環境に対応しています。ターゲットにしたいプラットフォームのための .NET Core SDK が必要です。

.NET Core バインディングのために用いられているバージョン方式は .NET のバージョン付け規則に準拠しています。.NET Core バージョン番号はたとえば NuGet キャッシュや

.csproj プロジェクトファイルの中で見ることができます。これらのバージョン番号は、PDFlib のメジャー・マイナーリリース番号に等しくありません。2 つのバージョン付け方式の間のマッピングは *compatibility.txt* 内にあります。

この製品は NuGet パッケージとして提供されており、以下のいずれかの方式を用いてローカルにインストールできます：

- *dotnet* コマンドラインツール（全プラットフォーム）。この方式については次項で詳説します。
- Visual Studio の Package Manager UI（Windows・macOS）
- Visual Studio の Package Manager Console（Windows）
- *nuget* コマンドラインツール（全プラットフォーム）

提供しているサンプル群のためのプロジェクトファイル群はターゲットフレームワーク .NET Core 2.0 に対して作成されています（ターゲットフレームワークモニカ TFM= *netcoreapp2.0*）。他のフレームワークをターゲットにしたい場合は、このプロジェクトファイルの中の TFM を調整するとよいでしょう（*net471* 等）。

dotnet コマンドラインツールを用いて .NET Core 用 PDFlib をインストール 提供している *hello* プロジェクトを例に、*dotnet* ユーティリティを用いてインストール・構成・ビルドを行うプロセスを説明します：

- 圧縮されている製品パッケージを、好きなディレクトリへ解凍します。
- コマンドシェルで、hello プロジェクトディレクトリへ *cd* します：

```
cd <installdir>\bind\dotnetcore\csharp\hello
```

- （このステップは、提供しているサンプル群では必要ありません。ローカルの *NuGet.Config* ファイルを用いてパッケージを参照しているからです）NuGet パッケージを、アプリケーションのプロジェクトディレクトリへ複製します：

```
<installdir>/bind/dotnetcore/PDFlib_dotnet.X.Y.Z.nupkg
```

- （このステップは、提供しているサンプル群では必要ありません。すでに PDFlib への参照を含んでいるからです）以下のコマンドを、適切なバージョン番号とともに入力します：

```
dotnet add package PDFlib_dotnet.X.Y.Z
```

このコマンドは PDFlib 参照を *.csproj* プロジェクトファイルに追加します。これはまた、PDFlib をローカル NuGet パッケージ内に、もしまだ存在していなければインストールします。例：

```
~/.nuget/packages/pdflib_dotnet/X.Y.Z
```

このキャッシュ化があるので、**.nupkg* を複製する必要があるのは最初のプロジェクトに対してのみです。それより後のプロジェクト群では、パッケージファイルはキャッシュから取られるので必要ありません。

- これで、hello プロジェクトをビルドして実行してみることができます：

```
dotnet build
dotnet run
```

結果として、アプリケーションディレクトリに *hello.pdf* 出力文書が生成されているのが見つかるはずです。

2.4.3 クラシック .NET バインディング

注 クラシック .NET バインディングに関する詳細な情報が、PDFlib-in-.NET-HowTo.pdf 文書にあります。これはディストリビューションパッケージにあるほか、PDFlib Web サイトでも入手可能です。

クラシック .NET バインディングをインストール　与えられているWindowsインストーラで PDFlib をインストールします。これは PDFlib アセンブリと補助データファイル群・解説文書・サンプルを、マシンに対話的にインストールします。このインストーラは PDFlib の登録も行って、Visual Studio の「参照の追加」ダイアログボックスの .NET タブで PDFlib を簡単に参照できるようにします。

C# プロジェクト内で .NET バインディングを参照　C# プロジェクト内で .NET バインディングを使用するには、Visual C# .NET で以下のように PDFlib アセンブリへの参照を作成する必要があります :「**プロジェクト**」→「**参照の追加 ...**」→「**参照 ...**」→インストレーションディレクトリから ***pdflib_dotnet.dll*** を選択。コマンドラインコンパイラを用いて以下の例のように PDFlib を参照することもできます :

```
csc.exe /r:..\..\bin\pdflib_dotnet.dll hello.cs
```

2.4.4 .NET バインディングをアプリケーションで利用

この項は .NET バインディングの両方の種類にあてはまります。さまざまな完全な例が、すぐ使える構成とともに、すべてのパッケージに含まれています。

.NET バインディングを正しく参照した後は、***PDFlib_dotnet.PDFlib*** ・ ***PDFlib_dotnet.PDFlibException*** クラスを使用できます。

.NET におけるエラー処理　.NET バインディングは .NET の例外に対応しており、実行時に問題が発生すると、詳細なエラーメッセージを持った例外を発生させます。こうした例外をキャッチして適切に対処するのはクライアント側の役目です。そうしない場合、.NET Framework が例外をキャッチして、たいていはアプリケーションを停止させます。

PDFlib は、例外関連情報を伝えるために、メンバ ***get_errnum*** ・ ***get_errmsg*** ・ ***get_apiname*** を持つ自前の例外クラス ***PDFlib_dotnet.PDFlibException*** を定義しています。PDFlib は IDisposable インタフェースを実装しているので、クライアントは Dispose( ) メソッドを呼び出してクリーンアップが可能です。

クライアントコードでは、PDFlib が発生させる例外を、通常の ***try*** ～ ***catch*** 構造を用いて処理できます :

```
try {
        ...PDFlib命令群...
catch (PDFlibException e)
{
        // PDFlibが投げた例外をキャッチした
        Console.WriteLine("PDFlib例外がhelloサンプル内で発生しました:\n");
        Console.WriteLine("[{0}] {1}: {2}\n",
            e.get_errnum(), e.get_apiname(), e.get_errmsg());
} finally {
        if (p != null) {
            p.Dispose();
        }
}
```

Unicode・レガシエンコーディング変換　PDFlib ユーザーの便宜のために、便利な C# 文字列変換方法を示しましょう。詳しくは .NET の説明書を参照してください。以下のコンストラクタはバイト配列（指定された変位と長さでの）から、*Encoding* 引数で与えられたエンコーディングを用いて Unicode 文字列を作成します：

```
public String(sbyte*, int, int, Encoding)
```

2.5 Objective-C バインディング

C・C++ 言語バインディングを Objective-C で使うこともできますが、Objective-C 用の真正言語バインディングもあります。以下の種類の PDFlib フレームワークを利用可能です：

- PDFlib：macOS 用
- PDFlib_ios：iOS 用

どちらのフレームワークも C・C++・Objective-C 用の言語バインディングを含んでいます。

PDFlib Objective-C 版をインストール　PDFlib を自分のアプリケーションで使うためには、*PDFlib.framework* か *PDFlib_ios.framework* をディレクトリ */Library/Frameworks* へコピーする必要があります。PDFlib フレームワークを別の場所にインストールすることも可能ですが、Apple の *install_name_tool* の使用が必要です。このツールについてはここでは説明しません。PDFlib メソッド定義を持つ *PDFlib_objc.h* ヘッダファイルをアプリケーションソースコード内で取り込む必要があります：

```
#import "PDFlib/PDFlib_objc.h"
```

または

```
#import "PDFlib_ios/PDFlib_objc.h"
```

PDFlib フレームワークをアプリ内に埋め込むには、Xcode のコード署名はバージョン番号 *A* を持つフレームワークを受け付けるのに対して、PDFlib 製品は数字のバージョン番号を用いています。これを解決するには、以下のように適切に名づけたフレームワークフォルダを作成します：

```
cd PDFlib.framework/Versions
mv 9.2.0 A
rm Current
ln -s A Current
```

引数命名規則　PDF メソッド呼び出しにあたっては、引数を、以下の規則に従って与える必要があります：

- 1 番目の引数の値は、メソッド名の直後に、コロンキャラクタで区切って与えます。
- その後の各引数については、その引数の名前と値（これも互いにコロンキャラクタで区切って）を与える必要があります。引数名は、PDFlib API リファレンスか *PDFlib_objc.h* 内で見つけることができます。

たとえば、PDFlib API リファレンス内の以下の行は：

```
void begin_page_ext(double width, double height, String optlist)
```

以下の Objective-C メソッドに照応します：

```
- (void) begin_page_ext: (double) width height: (double) height optlist: (NSString *) optlist;
```

よって、アプリケーションから以下のような呼び出しを行う必要があることになります：

```
[pdflib begin_page_ext:595.0 height:842.0 optlist:@""];
```

コード補完のための Xcode Code Sense は PDFlib フレームワークで使用できます。

Objective-C におけるエラー処理　Objective-C バインディングは、PDFlib エラーを、ネイティブ Objective-C 例外へ翻訳します。実行時の問題が発生したときには、PDFlib は、クラス PDFlibException のネイティブ Objective-C 例外を発生させます。この例外は通常の try/catch 機構で処理できます：

```
@try {
        ...PDFlib命令群...
}
@catch (PDFlibException *ex) {
        NSString * errorMessage =
            [NSString stringWithFormat:@"PDFlib error %d in '%@': %@",
            [ex get_errnum], [ex get_apiname], [ex get_errmsg]];
        NSAlert *alert = [[NSAlert alloc] init];
        [alert setMessageText: errorMessage];
        [alert runModal];
        [alert release];
}
@catch (NSException *ex) {
        NSAlert *alert = [[NSAlert alloc] init];
        [alert setMessageText: [ex reason]];
        [alert runModal];
        [alert release];
}
@finally {
        [pdflib release];
}
```

get_errmsg メソッドのほかに、例外オブジェクトの reason フィールドを使ってエラーメッセージを取得することもできます。

2.6 Perl バインディング

Perl 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 版をインストール　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付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、***-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***（macOS の場合：***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、以下の 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/5.32/i686-linux
```

Windows　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、以下の 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\Perl5.32\site\lib
```

Perl におけるエラー処理　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 {
        ...PDFlib命令群...
};
if ($@) {
        die("$0: PDFlib例外が発生しました:\n$@");
```

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。以下のコードスニペット群は

この 3 つの場合すべてを演示しています。どの例も同じ日本語の出力を作成しますが、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_option("stringformat=utf8");
$font = $p->load_font("Arial Unicode MS", "unicode", "");
$p->setfont($font, 24.0);
$p->fit_textline(uhex("U+65E5 U+672C U+8A9E"), $x, $y, "");
```

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_option("textformat=utf16le");
$font = $p->load_font("Arial Unicode MS", "unicode", "");
$p->setfont($font, 24.0);
$p->fit_textline("\xE5\x65\x2C\x67\x9E\x8A", $x, $y, "");
```

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_option("searchpath={{../../../resource/cmap}}");
$font = $p->load_font("Arial Unicode MS", "cp932", "");
$p->setfont($font, 24.0);
$p->fit_textline("\x93\xFA\x96\x7B\x8C\xEA", $x, $y, "");
```

Unicode・レガシエンコーディング変換　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。以下のコンストラクタはバイト配列から UTF-16 Unicode 文字列を作成します：

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

以下のコンストラクタは Unicode キャラクタ名から Unicode 文字列を作成します：

```
$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
```

2.7 PHP バインディング

PDFlib PHP 版をインストール　PDFlib を PHP 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、*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。

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。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=php_pdflib.so        ; Unix・macOS用
extension=php_pdflib.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 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 *PDFlib* という見出しのセクションをチェックしてください。

- 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("php_pdflib.so");           Unix用
dl("php_pdflib.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 53 ページ「3 章 PDF 文書を作成」のさまざまなコード断片例では、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 におけるファイル名処理　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。なお、PDFlib は UTF-8 符号化されたファイル名を受け付けますが、PHP の *dirname()* 関数は通常は WinAnsi 等のホストエンコーディングを返すことに留意してください。この場合には、ディレクトリかファイルの名前を UTF-8 へ変換する必要があります。詳しくは 109 ページ「5.2.2 UTF-8 対応のある言語バインディング」を参照してください。例：

```
$searchpath = dirname(dirname(__FILE__)).'/data';
$searchpath = $p->convert_to_unicode("auto", $searchpath, "outputformat=utf8");
```

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（57 ページ「3.1.4 リソース構成とファイル検索」参照）。

PHP における例外処理　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。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 {
        $p = new PDFlib();
        ...PDFlib命令群...

} catch (PDFlibException $e) {
        print "PDFlib例外が発生しました:\n";
        print "[" . $e->get_errnum() . "] " . $e->get_apiname() . ": "
                $e->get_errmsg() . "\n";
}
catch (Throwable $e) {
        die("PHP例外が発生しました: " . $e->getMessage() . "\n");
}
```

Unicode・レガシエンコーディング変換　*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 を用いて開発　PHP Development Tools（PDT）は Eclipse と Zend Studio による PHP 開発に対応しています。PDT は、以下に示す手順で、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*」→「*Source Paths*」→「*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以下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。

2.8 Python バインディング

PDFlib Python 版をインストール　Python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 Python 用 PDFlib ライブラリ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・macOS：*pdflib_py.so*
- Windows：*pdflib_py.pyd*

PDFlib ライブラリのほかに、以下のファイルもライブラリと同じディレクトリに入れておく必要があります：

- *PDFlib/PDFlib.py*
- *PDFlib/__init__.py*（Python 2.7 の場合のみ）

Python におけるエラー処理　Pythonバインディングはエラー時に*PDFlibException*を発生させます。Python 例外は通常の try/catch で取り扱うことができます：

```
try:
        p = PDFlib()

        ...PDFlib命令群...
except PDFlibException:
        print("PDFlib例外が発生しました:
        print("[%d] %s: %s" % (ex.errnum, ex.apiname, ex.errmsg))

except Exception:
        print(ex)

finally:
        if p:
            p.delete()
```

2.9 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。このコードページは *CHGJOB CCSID(37)* を用いて設定できます。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 用 RPG プログラムをコンパイル・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 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、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。以下の例は、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))
```

これで、*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：

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

生成されたプログラムを実行する前に、以下のコマンドを適用する必要があります：

```
chgcurdir '/pdflib/pdflib/x.y/x.y.z/bind/rpg'
```

生成された PDF 文書は、同一ディレクトリ内に見つけることができます。

引数は、表 2.2 に挙げるデータ型に従って PDFlib API へ渡される必要があります。

表 2.2 RPG バインディングにおけるデータ型

API データ型	RPG バインディングにおけるデータ型
文字列データ型	Unicode 文字列（%ucs2 を使用）
バイナリデータ型	data

RPG におけるエラー処理 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
```

2.10 Ruby バインディング

PDFlib Ruby 版をインストール　Ruby 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 の場合：*PDFlib.so*。macOS の場合：*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/<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 におけるエラー処理　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、通常の ***rescue*** 技法で扱うことができます：

```
begin
        ...PDFlib命令群...
rescue  PDFlibException => pe
  print "PDFlib例外がhellpサンプル内で発生しました:\n"
  print "[" + pe.get_errnum.to_s + "] " + pe.get_apiname + ": " + pe.get_errmsg + "\n"
end
```

Ruby on Rails　Ruby on Rails 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。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 と Ruby on Rails をインストール。
- コマンドラインから新規コントローラをセットアップ：

```
$ rails new pdflibdemo
$ cd pdflibdemo
$ cp <PDFlibディレクトリ>/bind/ruby/<バージョン>/PDFlib.so vendor/  # .so/.dll/.bundleを使用
$ rails generate controller home demo
$ rm public/index.html
```

- *config/routes.rb* を編集：

```
...
# public/index.htmlを削除することを忘れないようにしてください
root :to => "home#demo"
```

- *app/controllers/home_controller.rb*を以下のように編集して、PDF内容を作成するための PDFlib コードを挿入。PDF 出力はメモリ内に生成しなければならないことに留意して

ください。すなわち、空のファイル名を *begin_document()* に与える必要があります。出発点として、*hello-rails.rb* サンプル内のコードを利用できます：

```
class HomeController < ApplicationController
def demo
require "PDFlib"
begin
     p = PDFlib.new
     ...
     ...PDFlibアプリケーションコード、hello-rails.rbを参照...
     ...
     send_data p.get_buffer(), :filename => "hello.pdf",
     :type => "application/pdf", :disposition => "inline"
     rescue PDFlibException => pe
     # エラー処理
end
end
end
```

- インストレーションを試験するには、以下のコマンドで WEBrick サーバを起動し、

  ```
  $ rails server
  ```

  そしてブラウザで *http://0.0.0.0:3000* を表示させます。生成された PDF 文書がブラウザ内に表示されます。

ローカル PDFlib インストレーション　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。

3 PDF 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：

- 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。

失敗した関数呼び出しの原因をクエリ 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。ただし、関数が、エラー値を返すことによって、致命的でない問題を報告した場合には、文書を続行することもできます。クライアントアプリケーションは自由に、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、文書を続行することができます。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取得し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 の場合：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* オプションで構成できます。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*：この設定は廃止ですが、以前のバージョンの PDFlib と互換な動作を保証します。エラーコードを返す関数もありますし、例外を投げる関数もあり、それは個々のAPIの説明に依存します。この*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、関数がエラー値 -1（PHP の場合：0）を返します。アプリケーション開発者はこの戻り値をチェックして問題を突きとめ、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。例外の後には、出力文書は不完全となり使えなくなります。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。

以下のコード断片群は、例外処理に関するさまざまな戦略を演示します。このさまざまな例は、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チェックする必要があります。それが失敗を示しているときは、失敗の原因をクエリすることで、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、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-programming/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しても、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。例外を発生させるのではなく、状況の説明がログ記録されます（ログ記録機能についてさらに詳しくは 55 ページ「3.1.2 ログ記録」を参照）。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 ログ記録

PDFlib は、以下の項目を記録するログファイルを生成できます：

- すべての API 呼び出し、およびその引数群とオプション群。なお、PDFlib 言語ラッパによって発されたさらなる API 呼び出しもログ記録に含まれる場合があります（文字列変換のため等）。
- API 関数によって返された戻り値とハンドル。
- 各呼び出しに対するタイムスタンプ。
- 廃止 API 機能の使用に関する情報。
- 例外を発生させるほどではないが、アプリケーションの開発フェーズの間に調べるべき警告メッセージ。
- サポートケースの調査に有用でありうる内部動作に関する詳細。

ログファイルの内容は、アプリケーション開発者にとって、プログラムフローの中の問題を特定するうえで重要な助けとなりえます。ログ記録は実行時に以下のように有効化できます：

```
p.set_option("logging={filename={mylogfile.log}}");
```

あるいは、ログ記録を環境変数を通じて有効化させることも可能です。ログ記録される情報の量は、さまざまなログ記録クラスと、各クラスに対するレベルを用いて制御できます。詳しくは PDFlib API リファレンスを参照してください。

ログ記録は通常、本番環境においては有効化するべきではなく、開発フェーズ中にのみ、かつ問題を解析する必要がある時に有効化されるべきものです。PDFlib GmbH のサポートでは、ユーザーの問題を話し合うためにログファイルを要請することがあります。

以下は、デフォルトのログ記録クラスを用いた典型的なログファイルから数行を抜粋したものです：

```
PDF_load_image(p_0x2201c20, "auto", "nesrin.jpg", /*c*/0, "")
[0]

PDF_begin_page_ext(p_0x2201c20, 10.000000, 10.000000, "")
[Begin page #1]
PDF_fit_image(p_0x2201c20, 0, 0.000000, 0.000000, "adjustpage")
PDF_close_image(p_0x2201c20, 0)
PDF_end_page_ext(p_0x2201c20, "")
[End page #1]

PDF_end_document(p_0x2201c20, "")
[Full product name: "PDFlib Personalization Server"]
[Document ID: <C98301CB2D4EAC2972D34CAAAE929021> <C98301CB2D4EAC2972D34CAAAE929021>]
```

```
[Size of document: 34842 bytes]
[End document #1]
```

3.1.3 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 の構成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*
- 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チェックします。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。

さまざまな戦略　PVF では、仮想ファイルのための必要メモリの管理アプローチは複数ありえます。どのアプローチ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_end_document()* 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照応するデータ構造が解放されるだけだということです。よって、以下のような戦略がありえます：

- メモリ使用を最小にする：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_end_document( )* 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：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、クライアントが照応する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内に作成　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 が作成した 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（62 ページ「3.1.5 PDF 文書をメモリ内に生成」参照）。

3.1.4 リソース構成とファイル検索

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・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、構成ファイルを作成しておくことができます。構成ファイルの中には、利用可能なさまざまなリソースとその照応するディスクファイル名を記述しておくのです。このような静的な構成ファイルだけではなくて、*PDF_set_option( )* によるリソースの追加を用いた動的な実行時構成を行うことも可能です。構成ファイルに関しては PDFlib は、*Unix PostScript Resource*（UPR）という簡単なテキスト形式を用いています。ただし元の UPR 形式を、独自の目的のために多少拡張してあります。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。

enumeratefonts オプションを用いると、PDFlib に、検索パス上に存在するすべてのフォントを収集させることができます（59 ページ「ファイル検索と SearchPath リソースカテゴリ」を参照）。*saveresources* オプションを用いると、PDFlib リソースのカレントリストをファイルへ書きだすことができます：

```
/* フォントディレクトリを検索パスに追加 */
p.set_option("searchpath={{C:/fonts}}");

/* 検索パス上のすべてのフォントをなめてUPRファイルを作成 */
p.set_option("enumeratefonts saveresources={filename={C:/fonts/pdflib.upr}}");
```

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。多くのカテゴリは、リソース名（PDFlib API 内でもちいるための）を、仮想かディスクベースのファイルへマップするものです。表 3.1 にあるもの以外のリソースカテゴリは無視されます。カテゴリ名は大文字・小文字区別されます。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（行頭に BOM 付き）で、あるいは IBM Z の場合には EBCDIC-UTF-8 で符号化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	pathname	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	cmapname=filename	日中韓エンコーディングのための CMap ファイル
FontAFM	fontname=filename	AFM 形式の PostScript フォントメトリックファイル
FontPFM	fontname=filename	PFM 形式の PostScript フォントメトリックファイル
FontOutline	fontname=filename	PostScript・TrueType・OpenType・WOFF・CEF のうちいずれかのフォントアウトラインファイル
Encoding	encodingname=filename	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	fontname=hostfontname	システムにインストールされているフォントの名前（通常、両方のフォント名は等しい）
FontnameAlias	aliasname=fontname	PDFlib がすでに知っているフォントに対してエイリアスを作成
ICCProfile	profilename=filename	ICC カラープロファイル名

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。
- すべてのエントリは、大文字・小文字を区別します。
- 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。リソースエントリに列挙されたファイルを 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・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*** 項目を蓄積させることもでき、それらは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。検索パスエントリを設定するには以下のようにします：

```
p.set_option("SearchPath={{/パス/パス/ディレクトリ1} {/パス/パス/ディレクトリ2}}");
```

検索パスを複数設定することもでき、また、複数のディレクトリ名を 1 回の呼び出しの中で与えることもできます。空白キャラクタを含んだディレクトリ名による問題を避けるために、エントリが 1 個だけの場合にも中カッコを二重に用いることを推奨します。空の文字列リスト（例：{{}}）は、既存のすべての検索パスを、デフォルト項目も含めて削除します。

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 はレジストリからの項目群で ***SearchPath*** を初期化します。以下のレジストリエントリ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。これらは以下の順序で検索されます：

```
HKLM\SOFTWARE\PDFlib\PDFlib9\9.3.1\SearchPath
HKLM\SOFTWARE\PDFlib\PDFlib9\SearchPath
HKLM\SOFTWARE\PDFlib\SearchPath
```

- IBM System i 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/9.3/resource/icc
/PDFlib/PDFlib/9.3/resource/fonts
/PDFlib/PDFlib/9.3/resource/cmap
/PDFlib/PDFlib/9.3
/PDFlib/PDFlib
/PDFlib
```

  これらのエントリの最後のものは特に、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。

デフォルトファイル検索パス　Unix・Linux・macOS・IBM System i システムでは、パス・ディレクトリ名を一切指定しなくても、いくつかのディレクトリがデフォルトでファイル検索されます。UPR ファイル（追加の検索パスを内容として持つ可能性のある）を検索して読み取る前に、以下のディレクトリが検索されます：

```
<rootpath>/PDFlib/PDFlib/9.3/resource/cmap
<rootpath>/PDFlib/PDFlib/9.3/resource/codelist
<rootpath>/PDFlib/PDFlib/9.3/resource/glyphlst
<rootpath>/PDFlib/PDFlib/9.3/resource/fonts
<rootpath>/PDFlib/PDFlib/9.3/resource/icc
<rootpath>/PDFlib/PDFlib/9.3
<rootpath>/PDFlib/PDFlib
<rootpath>/PDFlib
```

Unix・Linux・macOS では、***<rootpath>*** はまず ***/usr/local*** へ置き換えられ、ついで HOME ディレクトリへ置き換えられます。IBM System i では、***<rootpath>*** は空です。

デフォルトファイル名とライセンスファイル・リソースファイル　デフォルトでは、以下のファイル名が、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。MVSの場合: license)
pdflib.upr               (リソースファイル。MVSの場合: upr)
```

この機能を利用すると、ライセンスファイルを、環境変数や実行時オプションを一切設定しないで扱うことができます。

サンプル UPR ファイル　UPR 構成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
/users/kurt/my_images
.
FontOutline
ArialMT=Arial.ttf
.
HostFont
Wingdings=Wingdings
```

```
.
ICCProfile
highspeedprinter=cmykhighspeed.icc
.
```

UPR リソースファイルを検索　組み込みリソース（たとえば PDF コアフォント・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 構成ファイルは必要とはされません。なぜなら、とりたてて構成がなくても、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_option()***（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以下のとおりです：

- Unix システム・macOS・IBM System i 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、以下のディレクトリが検索されます（この順に）：

  ```
  <rootpath>/PDFlib/PDFlib/9.3/resource/icc
  <rootpath>/PDFlib/PDFlib/9.3/resource/fonts
  <rootpath>/PDFlib/PDFlib/9.3/resource/cmap
  <rootpath>/PDFlib/PDFlib/9.3
  <rootpath>/PDFlib/PDFlib
  <rootpath>/PDFlib
  ```

  Unix システム・macOS 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。IBM System i では ***<rootpath>*** は空です。この機能を利用すれば、環境変数や実行時オプション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 ***PDFLIBRESOURCEFILE*** 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 ***PDFLIBRESOURCEFILE*** が定義されていない場合、PDFlib は以下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 System i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 は以下のレジストリエントリを読み込もうとします。以下の順に検索されます：

  ```
  HKLM\Software\PDFlib\PDFlib9\9.3.1\resourcefile
  HKLM\Software\PDFlib\PDFlib9\resourcefile
  HKLM\Software\PDFlib\resourcefile
  ```

  そしてこれらのエントリの値を、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には注意が必要です：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」**ダイアログから以下によって起動することができます：

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

- クライアント側で *resourcefile* オプション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。以下のように記述します：

  ```
  p.set_option("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、リソースエントリが蓄積されていきます。

リソースを実行時に構成　UPR ファイルを使った構成だけではなく、ソースコード中で *PDF_set_option()* を使って直接個々のリソースを構成することもできます。この関数はカテゴリ名とそれに照応するリソースエントリをとります。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：

```
p.set_option("FontOutline={Foobar-Bold=foobb.otf}");
```

注 フォント構成の詳しい説明は 143 ページ「6.4.4 フォントを検索」を参照してください。

リソース値をクエリ　リソースエントリを設定するだけでなく、*PDF_get_option()* を使ってクエリすることもできます。カテゴリ名をキーとして、リソースの番号（1 から開始）をオプションとして指定します。たとえば以下の呼び出し：

```
idx = p.get_option("SearchPath", "resourcenumber=" + n);
sp = p.get_string(idx, "");
```

は、SearchPath リスト内の *n* 番目のエントリを取得します。要求されたカテゴリに対して利用可能なエントリの数よりも *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。

3.1.5 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 1 チャンクを送り出している間に、次の 1 チャンク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（57 ページ「PDF 出力を仮想ファイル内に作成」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作成するには、*PDF_begin_document()* で空のファイル名を指定し、*PDF_get_buffer()* でデータを取得します：

```
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、クライアント側で構成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、複数のチャンク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。

3.1.6 PDF 文書の最大サイズとその他の制限

PDF 文書のサイズ　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：

- 10 GB ファイルサイズ制限：PDF 1.4 の文書は、相互参照テーブルによって、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。10 GB を超える出力文書を作成しようとするなら、PDF 1.5 以上を使用する必要があります。このバージョンは、10 桁制限にもはや縛られない圧縮オブジェクトストリームをサポートしていますので、10 GB を超える PDF 文書の作成が可能です。
- 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・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。それ以外のモードではつねに、もっとオブジェクトの多い文書を作成できます。このチェックを、文書オプション *limitcheck=false* を用いて無効化することもできます。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、規格が要求する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。

PDF の制限　PDFlib は、特定の実体に制約を課すことによって、PDF Reference か Acrobat、または何らかの PDF 規格によって課せられる制限に準拠する PDF 出力を作成します。これらの制限を以下に記します。

以下の制限が、値を然るべく変更することによって強制されます：

- PDF 内における最小の絶対浮動小数点値：0.000015。これより小さな絶対値を持つ数は 0 へ置換されます。
- （PDF 1.4。ただしそれより新しい PDF バージョンにはあてはまりません）PDF 内において浮動小数点数として表現できる最大の絶対値：32767.0。これより大きな絶対値を持つ数は、その最も近い整数へ置換されます。

PDF 形式は特定の諸制限を課しています。以下のいずれか 1 つの制限を超えると例外が発生します：

- PDF 内において許容される最大の数値：2,147,483,647
- ハイパーテキスト文字列の最大長：65535
- ページ上のテキスト文字列の最大長：*kerning=false* かつ *wordspacing=0* の場合には 32,763 バイト（すなわち、CID フォントの場合には 16,381 キャラクタ）。そうでないなら 4095 キャラクタ
- 以下のオプションのリストエントリは最大 8191 個に制限されています：
  views, namelist, polylinelist, fieldnamelist, itemnamelist, itemtextlist, children, group
- PDF/A-1/2/3 と PDF/X-4/5 内における最大間接オブジェクト数：8,388,607

3.1.7 マルチスレッドプログラミング

PDFlib のスレッディング動作は、次のように特徴づけられます：PDFlib そのものはシングルスレッドですが、マルチスレッドアプリケーションで安全に使用することができます。1 つの PDFlib オブジェクトが 1 つのスレッド内でのみ使用されるというよくある状況においては、マルチスレッディングに関する特別な注意は一切必要ありません。同一の PDFlib オブジェクトが複数のスレッド内で使用される場合には、その PDFlib オブジェクトが複数のスレッドによって同時にアクセスされることのないよう、アプリケーションはそれらのスレッドを同期させる必要があります。典型的なシナリオは PDFlib オブジェクト群のプールを利用するもので、そこでは各スレッドが、新規 PDFlib オブジェクトを作成するのではなく、既存の PDFlib オブジェクトを 1 つ取り出して、そして文書を作成した後に、そのオブジェクトがもう必要ない場合には、それをプールへ返します。同一の PDFlib オブジェクトを、その出力文書が完了する前に別のスレッドで使用することは、アプリケーションにとっていかなる利点をも与えることはめったになく、推奨されません。

3.1.8 EBCDIC ベースのプラットフォームで PDFlib を使う

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、IBM System i・IBM Z 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、以下のアイテムは 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、IBM System i ではコードページ 037 で、IBM Z ではコードページ 1047 で）：

- PFA フォントファイル・UPR 構成ファイル・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を使いたい場合は、***asciifile*** オプションを ***true*** に設定します（デフォルトは、IBM Z では ***false***、IBM System i では ***true***）。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、以下のアイテム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。

メートル座標を用いる　メートル座標を、座標系を拡縮することによって簡単に使えます。縮尺は、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縮尺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・ファイル注釈の長方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、*usercoordinates* オプションを *true* に設定します：

```
p.set_option("usercoordinates=true");
```

リンク・フィールドの長方形としては、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長方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、その長方形を囲う、かつ縁がページの縁と平行な最小の長方形を計算します。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* オプションが *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。

座標を視覚化　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寸法の座標を視覚化するものです。望みの寸法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 では以下のようにして視覚化できます：

- カーソル座標を表示するには以下を用います：
  Acrobat X/XI/DC：**「表示」→「表示切り替え」→「カーソル座標」**
- 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 X/XI/DC 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 67 ページ「下向き座標を用いる」を参照してください。

オブジェクトを回転　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。以下の例は傾き 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** textflow/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。以下の例は、左下隅を *(200, 100)* に持つ、回転された長方形を作成します。描きたい長方形の隅へ座標原点を変更し、座標系を回転させて、長方形を *(0, 0)* に配置しています。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);              /* 原点を長方形の隅へ移動*/
        p.rotate(45.0);                     /* 座標を回転させる */
        p.rect(0.0, 0.0, 75.0, 25.0);       /* 回転された長方形を描く */
        p.stroke();
p.restore();
```

下向き座標を用いる　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、*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。

以下のような「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、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、長方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 ページ寸法

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。**

規格ページ寸法 *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寸法名を指定できます。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*）。

ページ寸法の限界 PDF や PDFlib では、利用できるページ寸法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、ページ寸法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。他の PDF インタプリタではもっと大きな寸法やもっと小さな寸法の文書を扱うこともできる可能性があります。Acrobat のページ寸法制限を表 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縮尺を指定することもできます。

さまざまなページ寸法枠 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 の枠エントリを記述したいときもあるでしょう。PDFlib では、PDF のすべての枠エントリに対応しています。PDFlib のクライアントで指定できるエントリを以下に挙げます（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：

表 3.2 Acrobat の最小・最大ページ寸法

PDF 表示ソフト	最小ページ寸法	最大ページ寸法
userunit オプションなし（デフォルト）	1/24″ = 3 pt = 0.106 cm	200″ = 14400 pt = 508 cm
userunit オプションあり	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、可能なページ寸法は最大 14 400×75 000 ＝ 1 080 000 000 ポイント＝ 381 km

- MediaBox: ページの幅と高さを指定するために用いられ、通常私達がページ寸法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 はこの寸法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です。
- 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、それ以外のエントリを生成しません。以下のコード断片は、新しいページを開始させた後、CropBox の 4 つの値を設定します：

```
/* 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}");
```

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・長方形・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。パスに対して適用できる操作はいくつかあります：

- 描線。パスに沿って線を描きます。クライアントが与えた、描画に関するオプション（たとえば色や線幅）を用います。
- 塗り。パスで囲われた領域全体を塗ります。クライアントが与えた、塗りに関するオプションを用います。
- 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わ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。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.set_graphics_option("strokecolor=red");
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さまざまな方法で作成できます：

- *PDF_add_path_point()* は、点および紐付いたパス要素をパスオブジェクトに追加します。この関数は、新規に構築されたパスに対して、既存のパスオブジェクトへの参照を追加することもできます。*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えます。この関数はまた、SVG パス記述を受け付けます。以下のコード断片は、円 1 個を持つシンプルなパスオブジェクトを作成し、それをページ上の別々の 2ヶ所に描線し、最後にそれを削除します：

```
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);
```

- ラスタ画像内に含まれているクリッピングパスを、*PDF_info_image()* とキーワード *clippingpath* を用いて取得できます：

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

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");
```

- 配置された PDF ページ・SVG グラフィック・パス・ラスタ画像・表組み・範囲枠・テキストフロー・テキスト行を内包する長方形（外接枠）を、照応する *PDF_info_*()* 関数を用いて抽出できます：

```
optlist = "boxsize={400 300} fitmethod=clip matchbox={name=border}";
p.fit_image(image, 200, 150, optlist);

int path = (int) p.info_matchbox("border", 1, "boundingbox");
```

```
p.draw_path(path, 0, 0, "close stroke linewidth=10 strokecolor=red");
p.delete_path(path);
```

パスオブジェクトを作成または取得した後には、それをさまざまな目的に使えます：

- *PDF_draw_path()* を用いて、ページ上でそのパスの塗り・描線を行なったり、パスをクリッピングパスとして使用したりすることができます。
- *PDF_fit_textflow()* を用いてそのパスに複数行テキストを回り込ませる：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（252 ページ「9.2.11 テキストをパス・画像に回り込ませる」参照）。
- *PDF_fit_textline()* を用いてそのパスの上にテキストを配置、すなわち、キャラクタ群がそのパスの曲線に沿って並びます（231 ページ「9.1.7 パス上テキスト」参照）。
- *PDF_add_table_cell()* を用いてパスオブジェクト群を表セル内に配置。

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。

3.2.4 テンプレート（フォーム XObject）

テンプレート（PDF 内のフォーム XObject） 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（ページ上で直接操作するのではなく）。テンプレートができたら、それは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推奨されます。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。テンプレートは、クリッピングパスを持ったラスタ画像を複数回配置する場合にも推奨されます。テンプレートを作成するには以下の方法があります：

- *PDF_begin_template_ext()* で直接。
- ベクトルグラフィックから *PDF_load_graphics()* と *templateoptions* オプションで間接的に。
- ラスタ画像から *PDF_load_image()* と *templateoptions* オプションで間接的に。このオプションがないと、*PDF_load_image()* は、画像 XObject という同様の PDF 構造を作成します。

注 *PDF_open_pdi_page()* で取り込まれた PDF ページも PDF フォーム XObject を作成しますが、これはテンプレート関数ではなく PDI 関数で扱われます。

テンプレートは以下のように利用できます：

- *PDF_fit_image()* を用いてそのテンプレートをページ上または他のコンテンツストリーム上に配置（後述）：
- そのテンプレートによって定義した輝度ソフトマスクを用いてグラフィックステートを作成（*PDF_create_gstate()* のオプション *softmask* のサブオプション *template*、101 ページ「4.9.2 ソフトマスクを用いて色を変える」参照）。

- SVG グラフィックを読み込む際のフォールバック（背景）として（*PDF_load_graphics()* のオプション *fallbackimage*）。
- 注釈の体裁として（*PDF_create_annotation()* のオプション *template* のサブオプション *normal/rollover/down*）。
- 押しボタンフォームフィールドの体裁として（*PDF_create_field()* のオプション *icon/icondown/iconrollover*）。

テンプレートを作成・利用　テンプレートは、ラスタ画像とまったく同様に、*PDF_fit_image()* 関数でページ上または他のテンプレート上に貼り付けることができます(218ページ「8.4 画像・グラフィック・取り込み PDF ページを配置」参照)。一般に、PDFlib でテンプレートを作成・活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、以下の関数は、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_begin_item()* と、さまざまな関数の *tag* オプション：構造エレメントはテンプレート内では作成できません。
- すべてのインタラクティブ関数：これらは、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。**

3.3 PDF のパスワードセキュリティ

3.3.1 PDF におけるパスワードセキュリティ

PDF のパスワードセキュリティは、以下の保護機能を提供します：

- ユーザーパスワード（開くパスワードとも呼ばれます）が、ファイルを閲覧するために開くために必要。ユーザーパスワードを持つファイルのみが、クラッキングから安全です！
- マスターパスワード（所有者または権限パスワードとも呼ばれます）が、権限、ユーザーまたはマスターパスワードといったセキュリティ設定を変更するために必要です。ユーザーパスワードとマスターパスワードを持ったファイルは、いずれかのパスワードを与えることによって、閲覧するために開くことができます。
- 権限設定が、PDF 文書に対する、印刷やテキスト抽出といった特定の操作を制限します。
- 添付パスワードを、文書自体の本体内容ではなく、ファイル添付だけを暗号化するために与えることができます。

PDF 文書がこれらの保護機能のいずれか一つでも使用している場合には、それは暗号化されます。文書のセキュリティ設定を Acrobat で表示したり変更したりするためには、それぞれ**「ファイル」→「プロパティ ...」→「セキュリティ」→「詳細を表示 ...」**または**「設定を変更 ...」**をクリックします。

暗号化アルゴリズムとキー長　PDF の暗号化は、以下の暗号化アルゴリズムを活用しています：

- RC4：対称ストリーム暗号（すなわち、暗号化と復号に同一のアルゴリズムを使用できる）。RC4は適切な安全性を提供しなくなっており、PDF 2.0では廃止とされています。
- AES（Advanced Encryption Standard）：規格 FIPS-197 で仕様化された形のものです。AES は、さまざまな応用に使用されている現代的ブロック暗号です。

実際の暗号化キーは扱いにくいバイナリ列ですので、それは、プレーンなキャラクタ群から成る、よりユーザーフレンドリーなパスワードから導出されます。PDF と Acrobat の開発の過程につれて、PDF の暗号化方式は、より強力なアルゴリズムと、より長い暗号化キー、より洗練されたパスワードを使用する方向へ改良されてきました。表 3.3 に、すべての PDF バージョンに対する、暗号化と、キー・パスワード特性を詳述します。

表 3.3 各 PDF バージョンにおける暗号化アルゴリズム・キー長・パスワード長

PDF と Actoba のバージョン、pCOS アルゴリズム番号	暗号化アルゴリズムとキー長	最大パスワード長とパスワードエンコーディング
PDF 1.1 ～ 1.3（Acrobat 2 ～ 4） pCOS アルゴリズム 1	RC4 40 ビット（脆弱。PDF 2.0 では廃止）	32 キャラクタ（Latin-1）
PDF 1.4（Acrobat 5） pCOS アルゴリズム 2	RC4 128 ビット（脆弱。PDF 2.0 では廃止）	32 キャラクタ（Latin-1）
PDF 1.5（Actobat 6） pCOS アルゴリズム 3	PDF 1.4 と同じ、ただし暗号化方式の異なる応用（脆弱。PDF 2.0 では廃止）	32 キャラクタ（Latin-1）
PDF 1.6（Actobat 7）・PDF 1.7 ＝ ISO 32000-1（Acrobat 8） pCOS アルゴリズム 4	AES-128（脆弱。PDF 2.0 では廃止）	32 キャラクタ（Latin-1）

表 3.3 各 PDF バージョンにおける暗号化アルゴリズム・キー長・パスワード長

PDF と Actoba のバージョン、pCOS アルゴリズム番号	暗号化アルゴリズムとキー長	最大パスワード長とパスワードエンコーディング
PDF 1.7ext3（Acrobat 9） pCOS アルゴリズム 9	パスワード処理に欠陥を持った AES-256（脆弱。PDF 2.0 では廃止）	127 UTF-8 バイト（Unicode）
PDF 1.7ext8（Acrobat X/XI/DC）・PDF 2.0 ＝ ISO 32000-2 pCOS アルゴリズム 11	パスワード処理が改善された AES-256	127 UTF-8 バイト（Unicode）

パスワード　PDF の暗号化は、内部的には、PDF のバージョンによって、40・128・256 ビットのいずれかの暗号化キーで処理されています。このバイナリ暗号化キーは、ユーザーが与えたパスワードから導出されます。このパスワードには、長さとエンコーディングに制約があります：

- PDF 1.7（ISO 32000-1）以前は、パスワードは最大長 32 キャラクタに制約され、かつ Latin-1 エンコーディングのキャラクタのみを内容とすることができます。
- PDF 1.7ext3 で、Unicode キャラクタが導入され、最大長も、パスワードの UTF-8 表現で 127 バイトへ拡張されました。UTF-8 では、キャラクタを 1 ～ 4 バイトの可変長で符号化しますので、パスワード内に許される Unicode キャラクタの数は、それが非 ASCII キャラクタを含む場合には 127 より少なくなります。たとえば、日本語のキャラクタは通常、UTF-8 表現では 3 バイトを必要としますので、日本語 42 キャラクタまでをパスワード内で使用できるわけです。

曖昧さを避けるために、Unicode パスワードは、***SASLprep***（RFC 3454 内の ***Stringprep*** に基づいた RFC 4013 で仕様化された）という処理によって正規化されます。この処理は非テキストキャラクタを除去し、かつ特定のキャラクタクラスを正規化します（たとえば、非 ASCII の空白キャラクタは ASCII の空白キャラクタ U+0020 へマップされます）。パスワードは Unicode 正規化形 NFKC に正規化され、パスワード内に右書きと左書きのキャラクタ群が混在していた場合に起こりうる曖昧さを避けるために特別な双方向処理が施されます。

PDF 暗号化の強度は、暗号化キーの長さ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。名前や単なる単語などは、パスワードとして使用するべきではないことが広く知られています。なぜならこれらは簡単に推測されたり、いわゆる辞書攻撃を使ってシステマチックに試行されうるからです。さまざまな調査によれば、かなりの数のパスワードとして、配偶者やペットの名前、ユーザーの誕生日、子供のあだ名などが選ばれており、従って容易に推測可能となっています。

権限制限　PDF は、文書の操作に関するさまざまな制約を符号化することができ、これは個別に許可したり禁止したりすることができます：

- **印刷を許可**：印刷が許可されていなければ、Acrobat の印刷ボタンは無効のままになります。Acrobat は、**低解像度印刷（*150 dpi*）**と**高解像度印刷**の区別に対応しています。低解像度印刷は、ページのラスタ画像を生成し、これは私的利用にのみ適していますが、高解像度での複製ができないようになっています。画像ベースの印刷は、低い出力品質となるのみならず、印刷処理をかなり遅くさせることにも留意してください。
- **変更を許可**：以下の一覧は、さまざまな文書変更操作に対する制御を提供します：

  ページの挿入、削除、回転
  フォームフィールドへの入力と既存の署名フィールドに署名
  注釈の作成、フォームフィールドの入力と既存の署名フィールドに署名
  ページの抽出を除くすべての操作

Adobe Reader が表示する権限制限は信頼できないことに留意してください。たとえば、それはアセンブリ機能を持っていないので、文書内の実際の権限設定にかかわらず、つねに「**文書アセンブリ：許可しない**」と表示します。

- 内容のコピーは、「**テキスト、画像、およびその他の内容のコピーを有効にする**」を通じて制御されます。これは、「**スクリーンリーダーデバイスのテキストアクセスを有効にする**」を用いて有効化することもできますが、PDF 2.0 では、PDF リーダはつねにアクセシビリティに対応するべきなので、この設定は廃止です。

文書に対して、「**印刷を許可：なし**」といったアクセス制限を指定すると、Acrobat 内の各機能も無効になります。ただし、これはサードパーティの PDF ビューア等のソフトウェアについてもそうであるとは限りません。アクセス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、権限設定をまるで無視することが知られており、有償で入手可能な PDF クラッキングツールを利用すれば、すべてのアクセス制限を無効にすることが可能です。これは、暗号をクラックする話とは関係がなく、PDF ファイルを表示可能とする限り、それが印刷されないようにする方法は存在しないのです。このことは ISO 32000-1 に以下のように説明されています：

「ひとたび文書が開かれて成功裡に復号されれば、準拠リーダは技術的には文書の内容全体へのアクセスを有する。暗号化辞書内に指定された文書権限群を強制する実質は、*PDF* 暗号化内には何も存在しない。」

暗号化文書構成要素　デフォルトでは、PDF 暗号化はつねに文書のすべての構成要素を網羅します。しかし、文書のいくつかの構成要素だけを暗号化し、それ以外は暗号化したくないという利用場面も存在します：

- PDF 1.5（Acrobat 6）で、プレーンテキストメタデータという機能が導入されました。この機能を使うと、暗号化文書は、暗号化されていない文書 XMP メタデータを内容として持つことができます。これは、検索エンジンが暗号化文書からも文書メタデータを取得できるようにするためです。
- PDF 1.6（Acrobat 7）以降、ファイル添付を暗号化して、それ以外は文書を保護しないということが可能になりました。この方法により、保護されていない文書を、機密の添付のための入れ物として利用することができます。

セキュリティ推奨事項　ユーザーパスワード（文書を開くために必要な）を持った PDF のみがクラッキングから安全であることに留意してください。クラックされるおそれのある暗号化をしてしまわないためには、以下の推奨事項を守るべきです：

- 1 ～ 6 キャラクタから成るパスワードは避けるべきです。なぜならそれは、すべての可能なパスワードを試行する攻撃（パスワードに対するブルートフォース攻撃）に対して脆いからです。
- パスワードは、プレーンテキストな単語には似ていないべきです。なぜならそのパスワードは、すべてのプレーンテキストな単語を試行する攻撃（辞書攻撃）に対して脆いからです。パスワードは非アルファベットのキャラクタを含むべきです。自分の配偶者やペットの名前、誕生日など、容易に突きとめられるアイテムを用いてはいけません。
- 現代的な AES アルゴリズムが、以前の RC4 アルゴリズムよりも望ましいです。
- PDF 1.7ext3（Acrobat 9）に従った AES-256 は避けるべきです。なぜなら、それはパスワードチェックアルゴリズムに脆弱性があり、パスワードに対するブルートフォース攻撃が可能になっているからです。この理由から、Acrobat X/XI/DC と PDFlib では、新規文書を保護するためには Acrobat 9 の暗号化を決して使用しません（既存の文書を復号する際にのみ使用）。

まとめると、PDF 1.7ext8/PDF 2.0 に従った AES-256 を使用するべきです。パスワードは、6 キャラクタよりも長くするべきであり、かつ非アルファベットキャラクタを含むべきです。

Web 上の PDF を保護　¥PDF が Web で提供される場合には、ユーザーは自分のブラウザで、その文書のローカルコピーを作ることがつねに可能です。ユーザーに PDF 文書のローカルコピーを保存させない方法は存在しません。

3.3.2 PDFlib を用いて PDF 文書をパスワード保護

PDFlib は、PDF 文書を生成する際に、標準のセキュリティ機能を適用することができます。保護された PDF 文書から、PDFlib+PDI か PDFlib Personalization Server（PPS）でページを取り込むには、マスターパスワードか ***shrug*** オプションが必要です。文書のプロパティを pCOS インタフェースでクエリする際には、pCOS モードで制御されます。たとえば、XMP 文書データ・文書情報フィールド・しおり・注釈内容は、その文書がユーザーパスワードを必要としない場合には（またはユーザーパスワードのみが与えられた場合には）、マスターパスワードなしで取得できます。これについては pCOS パスリファレンスでさらに詳しく説明しています。

注 PDF 文書を、PDFlib 製品で Reader 有効化する（たとえば Acrobat Reader で注釈を許す）ことはできません。

暗号化アルゴリズムとキー長　パスワードを用いて文書を保護するために使用される暗号化アルゴリズムとキー長は、生成される文書の PDF バージョンに依存し、この PDF バージョンはさらに、***PDF_begin_document()*** の ***compatibility*** オプションに依存します。暗号化アルゴリズムは以下のように選択されます：

- PDF 1.4・1.5：128 ビットキーによる RC4 暗号化のそれぞれの種類が使用されます。
- PDF 1.6・PDF 1.7・PDF 1.7ext3：AES-128 が使用されます。なお、PDF 1.7ext3（Acrobat 9）に従った AES-256 は、既知の脆弱性のために決して使用されません。
- PDF 1.7ext8・PDF 2.0：Acrobat X/XI/DC に従った AES-256 が使用されます。

パスワードを PDFlib で設定　パスワードは、***PDF_begin_document()*** の ***userpassword*・*masterpassword*** オプションで設定できます。PDFlib は、生成される文書のためにクライアントが与えたパスワードと、以下の方式で相互作用します：

- ユーザーパスワードか権限設定が与えられていながら、マスターパスワードが与えられていない場合には、通常のユーザーが簡単にセキュリティ設定を変更することができるので、いかなる保護も無効となります。この理由から、PDFlib はこの状況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の場合には、そのファイルのユーザーと所有者の区別がもはやつかないので、この場合も効果的な保護は無効となります。PDFlib はこの状況をエラーと見なします。
- Unicode パスワードは AES-256 で許されます。それより古いすべての暗号化アルゴリズムは、Latin-1 文字集合に制限されたパスワードを必要とします。古い暗号化アルゴリズムに対して、与えられたパスワードが Latin-1 文字集合外のキャラクタを含む場合には、例外を発生させます。
- パスワードは、AES-256 の場合には 127 UTF-8 バイトに、それより古い暗号化アルゴリズムの場合には 32 キャラクタに切り落とされます。

権限を PDFlib で設定　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 1 個ないし複数の文字列で構成されます。*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。以下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 3.4 に、使えるすべての操作制限キーワードを挙げます。

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

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	（noannots を暗黙に前提します）Acrobat が、noannots が指定されていなくても、フォームフィールドへの記入を拒みます。
noaccessible	（PDF 2.0 では廃止）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を拒みます。
noassemble	（nomodify を暗黙に前提します）Acrobat が、nomodify が指定されていなくても、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	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。
plain-metadata	（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。

暗号化ファイル添付　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。

4 色空間

クックブック 色空間は、塗り操作と描線操作に対してそれぞれ別個に指定できます。たとえば *fillcolor*・*strokecolor* オプションを使用します。PDF において色空間は画像に対しても使用されますが、PDFlib は通常、画像の色空間を自動的に決定します。ただしカラー関連のオプションが *PDF_load_image()* で与えられた場合を除きます。

クックブック **カラー処理のコードサンプルが PDFlib クックブックの** color **カテゴリにあります。**color/starter_color **サンプルにすべての色空間の使用が演示されています。**

4.1 デバイス色空間

デバイス独自の色空間は、おそらく、色を記述するために最も広く用いられる手段です。名前のとおり、これは、ある特定のデバイスによって解釈されるものとして色を記述します。デバイスの例としてはモニタやプリンタや印刷機などです。デバイス独自カラーは、どのデバイスで出力を行うかによって、見た目が異なります。この望ましくない性質があるので、デバイスカラーは、PDF/A と PDF/X においては、直接的には許されておらず、然るべき出力インテントICCプロファイルかデフォルト色空間とともにのみ使用が許されます。

グレースケール色空間　グレースケールカラーは、色空間キーワード *gray* を用いて要求することができ、1 個のグレー値です。グレー値は 0 ＝黒から 1 ＝白までの範囲です。例：

```
p.set_graphics_option("fillcolor={{ gray 0.5 }");
```

RGB 色空間　加法的な RGB 色空間の色は、赤・緑・青の成分を混ぜて作られます。RGB カラーは、色空間キーワード *rgb* を用いて要求することができ、赤・緑・青の割合を指定する範囲 0 ～ 1 の 3 個の RGB 値であり、*(0, 0, 0)* ＝黒、*(1, 1, 1)* ＝白になります。広く用いられている範囲 0 ～ 255 の RGB カラー値を、PDFlib が要求する範囲 0 ～ 1 へスケールするには、255 で割る必要があります。

数値で RGB 値を指定せず、その HTML 名または 16 進値で RGB カラーを指定することもできます。例：

```
p.set_graphics_option("fillcolor={{ rgb 1 0 0 }");
p.set_graphics_option("fillcolor=pink");
p.set_graphics_option("fillcolor={#FFC0CB})";
```

CMYK 色空間　減法的な CMYK 色空間の色は、シアン・マゼンタ・イエロー・黒（「キー」）の成分を混ぜて作られます。CMYK 値は、色空間キーワード *cmyk* を用いて要求することができ、シアン・マゼンタ・イエロー・黒を表す範囲 0 ～ 1 の 4 個の CMYK 値であり、0 ＝色なし、1 ＝色フルを表します。*(0, 0, 0, 0)* ＝白、*(0, 0, 0, 1)* ＝黒になります。CMYK カラーの極性はグレースケール・RGB カラーとは異なっていることに注意してください。例：

```
p.set_graphics_option("fillcolor={{ cmyk 0 1 0 0 }");
```

この CMYK 色空間の特質として、CMYK オブジェクトのオーバープリント動作を、*overprintmode* オプションを用いて制御できるということがあります（104 ページ「CMYK カラーに対するオーバープリントモード」参照）。

デバイスカラーをデフォルト色空間を用いてデバイス独立カラーへマップ　デバイス色空間は、デバイス独立色空間へ、デフォルト色空間を用いてマップすることが可能です：詳しくは 83 ページ「デバイスカラーを ICC ベースの色空間へマップ」を参照してください。これは、RGB または CMYK の画像またはグラフィックなど、デバイス独自カラーを有するデータが、PDF/A または PDF/X の中で、合致する出力インテントを持たずに使用されている場合に有用です。

4.2 ICC プロファイルによる色管理

PDFlib は、ICC プロファイルとレンダリングインテントによる色管理に対応しています。ICC プロファイルは、色管理されたワークフローと、PDF/X・PDF/A といった多くの PDF 規格において、重要な役割を担います。

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

ICC プロファイル International Color Consortium（ICC）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、すべての主要な色管理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、表 4.1 に挙げる用途について、ICC プロファイルによる色管理に対応しています。色管理は、色指定の中の成分の数を変えません（RGB から CMYK へ等）。

注 **よくある印刷条件のためのICCカラープロファイル群への推奨とリンクがwww.pdflib.comで入手可能です。**

表 4.1 ICC プロファイルのさまざまな用途

用途	使用する API 関数・オプション
ページ上のテキストとベクトルグラフィックに ICC ベースの色空間を設定	PDF_set_option( ) で iccprofilegray/rgb/cmyk　および PDF_setcolor( ) で colorspace=iccbasedgray/rgb/cmyk 色オプションで iccbased キーワード
取り込んだ画像に ICC プロファイルを適用	PDF_load_image( ) : オプション iccprofile
画像に埋め込まれた ICC プロファイルを処理または無視	PDF_load_image( ) : オプション honoriccprofile
画像に埋め込まれた ICC プロファイルをクエリ	PDF_info_image( ) で keyword=iccprofile
グレースケースか RGB か CMYK データを ICC ベースの色空間へマップするためのデフォルト色空間を設定	PDF_begin_page_ext( )・PDF_begin_template_ext( )・PDF_begin_pattern_ext( )・PDF_begin_font( ) : オプション defaultgray/defaultrgb/defaultcmyk
PDF/X または PDF/A の出力インテントを、参照または埋め込み ICC プロファイルで指定	PDF_load_iccprofile( ) : オプション usage=outputintent
透過・ブレンドモードのためのブレンド色空間を指定	PDF_begin/end_page_ext( )・PDF_open_pdi_page( )・PDF_begin_template_ext( )・PDF_load_graphics( ) で templateoptions : オプション transparencygroup、サブオプション colorspace
スポットカラーのための代替色空間	PDF_set_graphics_option( ) : 色オプションでキーワード spotname と iccbased 代替色、または PDF_set_graphics_option( ) でオプション fillcolor の後に PDF_makespotcolor( )。
DeviceN カラーのための代替色空間	PDF_create_devicen( ) : オプション alternate でサブオプション iccbased
ICC プロファイル内の色成分の数をクエリ	PDF_get_option( ) でキーワード icccomponents

受け入れ可能な ICC プロファイル カラープロファイルは、そのプロファイルの ICC バージョン番号、そのデバイスクラス、そのデータ色空間に関して、特定の条件を満たす必要があります。ICC バージョン番号は以下に限られています：

- PDF 出力互換性 1.4：ICC バージョン 2.x

- PDF 出力互換性 1.5 以上：ICC バージョン 2.x または 4.x

表 4.2 に、ICC プロファイルに関するデバイスクラスとデータ色空間に関する、その用途に応じた追加の要請を詳述します。

表 4.2 さまざまな用途について受け入れ可能な ICC プロファイル

ICC プロファイルの用途	デバイスクラス	データ色空間
PDF/X-3/4・PDF/X-5p/5pg のための出力インテント	prtr	グレー・RGB・CMYK
PDF/X-5n のための出力インテント	prtr	xCLR（n 色）
PDF/A のための出力インテント	prtr・mntr	グレー・RGB・CMYK
透過グループ色空間	prtr・mntr・scnr・spac	グレー・RGB・CMYK
ICC プロファイルのその他すべての用途	prtr・mntr・scnr・spac	グレー・RGB・CMYK・Lab

ICC プロファイルを検索　PDFlib は、*PDF_load_iccprofile()* に与えられた *profilename* 引数を用い、以下の手順を踏んで 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チェックします。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、以下の組み合わせを 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>          (Windows・macOSのみ)
<colordir>/<ファイル名>.icc      (Windows・macOSのみ)
<colordir>/<ファイル名>.icm      (Windows・macOSのみ)
```

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（たとえば *C:\Windows\System32\spool\drivers*）。macOS では *colordir* として以下のパスが試みられます：

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

sRGB 色空間と sRGB ICC プロファイル　PDFlib は、sRGB と呼ばれる工業規格の RGB 色空間に対応しています。これは、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色管理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構成する必要はなく、あえて構成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。便利なショートカットとして、*PDF_load_iccprofile()*

で作成されたICCハンドルが期待されるすべての場所において、かわりにキーワード *srgb* を与えることもできます。

sRGBプロファイルは、デバイスクラス *mntr*（出力デバイス）に属します。すなわち、これはPDF/Aに対する出力インテントとしては使用できますが、PDF/Xに対する出力インテントとしては使用できません。

画像（ICCタグ付き画像）内の埋め込みプロファイルを用いる　画像のなかには、その画像のカラー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ものがあります。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 は、JPEG・JPEG 2000・PNG・TIFF 画像ファイル形式の中に埋め込まれたICCプロファイルを処理します。*honoriccprofile* オプションが *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ます。

PDF_info_image() のキーワード *iccprofile* 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する必要がある場合に有用です。

未知のICCプロファイル内の色成分数をチェックするには *icccomponents* オプションを用います。

外部ICCプロファイルを画像に適用　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。

ページ記述に対してICCベース色空間　テキストやベクトルグラフィックのカラー値は、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、*iccprofilegray*・*iccprofilergb*・*iccprofilecmyk* のうちのいずれかのオプションの値として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、ICCベースのカラー値を、*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()* の色オプションに与えることができます：

```
p.set_option("errorpolicy=return");
icchandle = p.load_iccprofile("myCMYK", "usage=iccbased");
if (icchandle == -1)
{
        return;
}
p.set_graphics_option("fillcolor={iccbased=" + icchandle + " 0 1 0 0}";
```

デバイスカラーをICCベースの色空間へマップ　PDFには、デバイス依存なグレー・RGB・CMYKカラーをデバイス独立なICCベースの色へマップする機能があります。これを利用すると、そのままではデバイス依存なカラー値に対して、正確な測色指定を与えることができます。これを実現するには、*PDF_begin_page_ext()*・*PDF_begin_template_ext()*・*PDF_begin_pattern_ext()*・*PDF_begin_font()* の *defaultgray*・*defaultrgb*・*defaultcmyk* オプションに、然るべきICCプロファイルハンドルを与えます。以下の例は、sRGB色空間を、ページ上のテキスト・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ます：

```
p.begin_page_ext(595, 842, "defaultrgb=srgb");
```

デフォルト色空間が外部 ICC プロファイルに由来する場合には、プロファイルハンドルをまず作成する必要があります：

```
/* ICCプロファイルハンドルを作成 */
icchandle = p.load_iccprofile("myRGB", "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　出力デバイスのプロファイルを用いて、PDF/X または PDF/A 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、プリンタまたはモニタプロファイルを出力インテントとして指定でき、PDF/X ではプリンタプロファイルのみ許されます。詳しくは 341 ページ「12.4 PDF/X による印刷出力」と 329 ページ「12.3 PDF/A によるアーカイビング」を参照してください。

4.3 デバイス独立な CIE L*a*b* カラー

クックブック 完全なコードサンプルが starter_color サンプル内にあります。

デバイス独立なカラー値を、CIE 1976 $L^*a^*b^*$ 色空間（略して *Lab*）において指定することができます。この $L^*a^*b^*$ 色空間における色は、3 個の値 ***L・a・b*** によって指定されます（図 4.1 参照）。輝度（または明度）*L* は 0 ～ 100 の範囲にわたります。値 *a* と *b* は範囲 −128 ～ 127 で色彩を記述します。値 *a* は緑（−128）からマゼンタレッド（+127）までの範囲にわたり、値 *b* は青 （−128）から黄色（+127）までの範囲にわたります。正の数は暖色（黄色・マゼンタレッド）を記述し、負の数は寒色（緑・青）を記述します。*a・b* 軸の交点（*a=b=0*）は無彩色のグレー値を記述し、黒 *(0, 0, 0)* から白 *(100, 0, 0)* までの範囲にわたります。

この Lab 色空間の重要な特性として、RGB・CMYK と異なり、知覚的に一様であるということがあります。すなわち、色どうしの数値の差が等しければ、その視覚的な差も等しくなります。これによって、複数の色の混ぜ合わせの計算がはるかに簡単になります。各色の Lab 値の加重平均をとればよいからです。92 ページ「Lab 代替空間を持ったスポットカラー群に基づく DeviceN 色空間」ではこのことを利用して PostScript 変換関数を生成します。

Lab 値は、白色点もあわせて指定されている場合にのみ、絶対的な色を指定します。PDFlib は、白色点として、標準光源 D50（日中光 5000 K、2° 測定者）を使用します。

PDFlib は Lab 色空間に以下のように対応しています：

- Lab カラーをテキスト・ベクトルグラフィックに使用するには、色オプションと *PDF_setcolor( )* において、色空間キーワード *lab* と、*L・a・b* 値の 3 個の数値を用います。例：

```
p.set_graphics_option("fillcolor={lab 100 0 0}");
```

  PDF/A モードでは、デフォルトの黒の塗り・描線色は、出力インテントが指定されていなければ、デバイス独立な Lab *(0, 0, 0)* として指定されます。
- PDFlib は内蔵データベースにおいて、すべての HKS・Pantone スポットカラーの代替カラー値に Lab 色空間を使用しています。
- Lab 色空間を用いた TIFF 画像を取り込めます。
- Lab 色空間を用いた ICC プロファイルを、*PDF_load_iccprofile( )* を用いて読み込めます。ただし、出力インテントとして使用しようとする場合、または *transparencygroup* オプションに使用しようとする場合を除きます。なぜなら PDF はこれらの箇所では Lab を許さないからです。

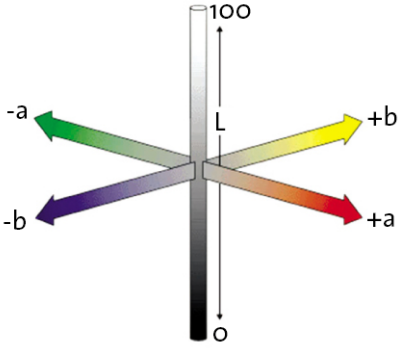


図 4.1 CIE L*a*b* 色空間

4.4 Pantone・HKS・カスタムスポットカラー

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

スポットカラーのための Separation 色空間 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 Separation 色空間として知られます）。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カラー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 では代替色を伴う必要があります。この代替色は、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。PDF ビューアや RIP 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。

PDFlib は、さまざまな組み込みスポットカラーライブラリに対応しており、かつ、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チェックします。もし見つかれば、PDFlib は代替色に関して、内蔵の Lab 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カラー値を、カラーオプション群の *spotname* カラーキーワードで（あるいは *PDF_setcolor( )* を使用の際にはカレント塗り色を通じて）与える必要があります。スポットカラーには濃度を指定することもできます。すなわち、0 ＝色なし＝白から 1 ＝最大濃度までの色濃度を指定するパーセント値とともに用いることができます。Separation 色空間はつねに減法的です（RGB のような加法的な色空間では 0 ＝黒になるのと異なり）。スポットカラーが定義されたら、それを用いて、テキスト・ベクトルグラフィックを描くこともできますし、グレースケール画像に着色することもできますし、シェーディングを構築することもできます。なお、画像にスポットカラーで着色する際には、PDFlib はその色の極性を自動的に反転させます（199 ページ「グレースケール画像にスポットカラーで着色」参照）。

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。この動作は、*PDF_set_option( )* の *spotcolorlookup* オプションを用いて変更することもできます。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。

注 組み込み Pantone®・HKS® スポットカラーのデータとその照応する商標については、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。

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®（名前付きカラー数千色を有する）に完全対応しています。表 4.3 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。

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。例示のように、必ず接頭辞 *PANTONE* をスウォッチ名につける必要があります。Pantone カラー名は次の方式に従って構成されます：

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

ここで ***<id>*** は色の識別子であり（たとえば 185）、***<paperstock>*** はペーパーストックなど指定の頭文字です（たとえば ***C*** は coated ＝コート紙）。スウォッチ名を構成する各成分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、警告がログ記録されます。以下のコードスニペットは、ある Pantone カラーを濃度値 70 パーセントで用いた例です：

```
p.set_graphics_option("fillcolor={ spotname {PANTONE 281 U} 0.7 }");
```

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., 2003-2016.

表 4.3 PDFlib に内蔵の Pantone スポットカラーライブラリ

カラーライブラリ名	カラー名の例	注
PANTONE ソリッド／コート紙	PANTONE 185 C	
PANTONE+ ソリッド／コート紙 –336 New	PANTONE 2071 C	2012 年追加の 336 色
PANTONE PLUS ソリッド／コート紙	PANTONE 2337 C PANTONE 3514 C	2014 年追加の範囲 2337 ～ 2427 の 84 色 2016 年追加の範囲 2428 ～ 3599 の 112 色
PANTONE ソリッド／上質紙	PANTONE 185 U	
PANTONE+ ソリッド／上質紙 –336 New	PANTONE 2071 U	2012 年追加の 336 色
PANTONE PLUS ソリッド／上質紙	PANTONE 2337 U PANTONE 3514 U	2014 年追加の範囲 2337 ～ 2427 の 84 色 2016 年追加の範囲 2428 ～ 3599 の 112 色
PANTONE ソリッド／マット紙	PANTONE 185 M	
PANTONE 拡張色域／上質紙（XGC）	PANTONE 185 XGC	2015 年追加の XGC 1729 色
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 年追加
廃止カラーライブラリ（廃止）		
PANTONE ヘキサクローム／コート紙	PANTONE H 305–1 C	2008 年廃止

表 4.3 PDFlib に内蔵の Pantone スポットカラーライブラリ

カラーライブラリ名	カラー名の例	注
PANTONE ヘキサクローム／上質紙	PANTONE H 305-1 U	2008 年廃止
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

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。表4.4に挙げるデジタルカラーライブラリにあるカラースウォッチ名がすべて利用可能です。

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。HKS カラー名は次の方式に従って構成されます：

```
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、警告がログ記録されます。以下のコードスニペットは、ある HKS カラーを濃度値 70 パーセントで用いた例です：

```
p.set_graphics_option("fillcolor={ spotname {HKS 38 E} 0.7 }");
```

表 4.4 PDFlib に内蔵の HKS スポットカラーライブラリ

カラーライブラリ名	カラー名の例	注
HKS K	HKS 43 K	グロスアート紙用 88 色（Kunstdruckpapier）
HKS N	HKS 43 N	ナチュラル紙用 86 色（Naturpapier）
HKS E	HKS 43 E	連続ステーショナリ / コート用 88 色（Endlospapier）
HKS Z	HKS 43 Z	ニュースプリント用 50 色（Zeitungspapier）

カスタムスポットカラー　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。これは、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）代替色を持つことができます。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。

スポットカラーは、*fillcolor/strokecolor* テキストまたはグラフィック書式オプションとその他の色関連オプションで設定できます。代替色は、スポットカラー定義内で直接与えることができます：

```
fillcolor={spotname={CompanyRed} 1.0 {cmyk 0 0.78 0.88 0}}
```

あるいは、カレント塗り色を用いてスポットカラーを定義することも可能です。この場合には、カレント塗り色が代替色として用いられます。カレント塗り色を代替色として設定するために呼び出しが 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、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：

```
/* 代替色として使いたいカレント塗り色を設定 */
p.setcolor("fill", "cmyk", 0 0.78 0.88 0);
/* カレント塗り色からスポットカラーを生成 */
spot = p.makespotcolor("CompanyRed");

/* 塗りにスポットカラーを設定 */
p.set_graphics_option("fillcolor={ spotname {CompanyRed} 0.7 }");
```

CMYK プロセスカラーに基づく Separation 色空間　スポットカラー名は一般に、任意に選ぶことができますが、インキ名 *Cyan*・*Magenta*・*Yellow*・*Black* はつねに CMYK プロセスカラーを指します。このことを利用して、特定の CMYK チャンネル（たとえばマゼンタ）のみに描画を行い、他のチャンネルに影響を与えないようにすることも可能です。これは、フル CMYK カラーで *Cyan=Yellow=Black=0* によって描画を行うのとは異なります。後者はシアン・イエロー・黒チャンネルの既存の内容を消去するからです（この動作はデバイス依存設定を用いて変えることも可能です。104 ページ「4.11 オーバープリント制御」参照）。たとえば、次の呼び出しは：

```
fillcolor={spotname=Magenta 0.5 {cmyk 0 1 0 0}}
```

次のように CMYK カラーを直接使用するのとは異なります：

```
fillcolor={cmyk 0 0.5 0 0}
```

なぜなら前者の呼び出しではシアン・イエロー・黒チャンネル内の既存の内容は変更されませんが、後者の呼び出しではそれらを 0% に設定するからです（オーバープリント設定に依存）。

4.5 DeviceN カラー

クックブック　完全なコードサンプルが、starter_color サンプルとクックブックの color/devicen_color トピックにあります。

DeviceN 色空間は、任意の数の名前付き色成分に対応しており、スポットカラーを一般化したものととらえることができます。そのインキ群は、プロセスカラーの集合から採ることもできますし（通常 CMYK ですが、それ以外の色空間も可）、任意のスポットカラー群とすることもできます。DeviceN カラーの応用は以下のとおりです：

- 4 色よりも多くのプロセスカラーを使うことによって色域（印刷可能な色の集合）を拡げている印刷系もあります。たとえば、CMYK に加えてオレンジ・緑・紫に基づく 7 色印刷系があります。
- CMYK プロセスカラー群の部分集合（たとえばシアンとマゼンタのみ）を内容とする DeviceN 色空間を用いると、CMYK カラーのオブジェクトを他の CMYK オブジェクト群にオーバープリントするべきときに有用です。
- DeviceN カラーを用いて、スポットカラーとプロセスカラーの間のシェーディング（なめらかな色遷移）を構築したり、複数のスポットカラーの間のシェーディングを構築したりすることも可能です。
- パッケージ印刷では、伝統的なスポットカラー群を使用できないので、n 色を使用することが多いです。追加のカラーチャンネルを用いて、目に見える色には直接関係しない情報を記録することも行われています。たとえばワニスやダイラインなどです。

PDF_create_devicen() 関数は、DeviceN 色空間ハンドルを返します。これを使って、***fillcolor/strokecolor*** などの色関連オプションを用いて描画操作を行うこともできますし、***PDF_shading()*** を用いてシェーディングを構築することもできますし、***PDF_load_image()*** の ***colorize*** オプションを用いてラスタ画像に着色することもできます。DeviceN カラーには濃淡をつけることもできます。すなわち、そのそれぞれのカラーチャンネルに対して、各カラーチャンネルの濃さを 0 ＝色なし＝白から 1 ＝最大濃度までの範囲で指定する割合値を持たせて使用することが可能です。DeviceN 色空間はつねに減法的です（RGB など加法的な色空間において 0 ＝黒となるのと異なります）。

PDF_create_devicen() には、インキ名のリストと、代替カラー名と、濃度変換関数を実装した PostScript コードが必須です。この変換関数は、DeviceN 色空間の N 個のカラー値を、代替色空間における照応するカラー値群へ変換するものである必要があります。この代替カラー値群は、出力デバイスがその DevicefN 色空間に対応していない場合（モニタ上など）に表現に用いられます。以下に、CMYK プロセスカラーかスポットカラー群から構築された DeviceN 色空間に対する、Lab を代替色として使用する場合の、然るべき PostScript 変換関数を示します。これ以外の組み合わせに対して然るべき PostScript 変換関数を作るには工夫が必要です。

PDF 1.6 以上に対する出力を生成する際には、PDFlib は DeviceN 属性辞書を出力します。この辞書には、すべての既知のスポットカラーに対する ***Colorants*** エントリが含まれています。これによって PDF ビューアは確実に、その名前付きインキ群を個別に表現するための充分な情報を得られますし、そのインキ群をブレンドすることもできるようになります。この手法の利点は、Acrobat においてオーバープリントプレビュー設定にかかわらず色表示が正しくなることにあります。かつ、Acrobat において PostScript 変換関数がなくてもそのページを表現できるようになります。ただし、この変換関数はいずれにせよ、与えなければなりません。なぜならサードパーティビューアや RIP は、DeviceN カラーを代替色空間で表現する際に PostScript 関数を頼るかもしれないからです。

PDF/X-4/5 モードにおいては、*PDF_create_devicen()* の前に、その DeviceN 色空間において使用されるすべてのカスタムスポットカラーについて、*PDF_makespotcolor()* を呼び出す必要があります。これは一般に画面表示を向上させますので、PDF/X-4-5 を生成しない場合であっても推奨されます。

NChannel 色空間　PDF 1.6 では、DeviceN 色空間の拡張として、NChannel というものが導入されました。NChannel 色空間は、PDF ビューアや RIP が正確な色表現を行うことを助ける情報を追加で内容として持ちます。この情報は、その DeviceN 色空間における個々のインキに対する代替色空間と（結合された濃度変換関数だけでなく）、プロセス色空間と、成分名群を含みます。NChannel はさらに、印刷インキの混ぜ合わせ動作を記述した情報を含むことも可能ですが、これは目下 PDFlib では対応していません。NChannel 色空間では、そのインキ群に対する個別の代替色を持つ *Colorants* エントリが必須です。これは PDFlib によって自動的に生成されます。NChannel 色空間ではさらに、プロセス色空間の名前とそのインキ名群も必須です。これらについては *process* オプションと *colorspace・components* サブオプションで与える必要があります。

NChannel 色空間を作成するには、*PDF_create_devicen()* でオプション *subtype=nchannel* を用います。この設定の利点は、このサブタイプを持った DeviceN 色空間は Acrobat の**「オーバープリントプレビュー」**設定に依存しなくなることです。場合によっては、スクリーンプレビューを正しくするためにこの Acrobat 設定が必要になることがあるのですが、NChannel は自動的に Acrobat をこの表示モードへ切り替えます。

CMYK プロセスカラーに基づく DeviceN 色空間　以下の例においては、マゼンタとイエローのインキで構成された、CMYK カラーの部分集合を使い、かつ代替色空間として DeviceCMYK を使います。その変換関数はシンプルです。なぜなら CMYK 代替色空間のシアン・黒成分に対するゼロ値を補足しているだけだからです：

```
devicen = p.create_devicen(
        "names={Magenta Yellow} alternate=devicecmyk transform={{0 0 4 1 roll}}");
p.set_graphics_option("fillcolor={devicen " + devicen + " 0.5 1}");
```

その色を設定する際には N 個のカラー値を与える必要があります（上の例では N=2）。上の例で作成した DeviceN 色空間は、マゼンタとイエローのチャンネルにのみ描画し、シアンと黒のチャンネルを変更しません。これは、フル CMYK カラーで *Cyan=Black=0* として描画を行うとシアンとブラックのチャンネルにおける既存の内容が消去される（オーバープリント設定に依存）のと異なります。

便宜のために、表 4.5 に、CMYK インキのすべての可能な部分集合に対する、CMYK を代替色空間とした場合の、DeviceN PostScript 変換関数を挙げます。

同様の技法を用いて、CMYK 画像チャンネルの部分集合のみを表現することも可能です。たとえば以下のコード断片は、黒チャンネルのみを使う、それ以外の 3 個のチャンネルを *None* へ置き換える DeviceN 色空間を作成します。その画像データ内には 4 個のカラーチャンネルがあるため、この PostScript 変換関数はシアンとマゼンタとイエローのチャンネルをゼロ値へ置き換えています（これは、表 4.5 の PostScript 関数が既存のカラー値群をゼロへ置き換えるのでなく足りない 0 値群を補っているのと異なります）：

```
devicen = p.create_devicen(
        "names={None None None Black} " +
        "alternate=devicecmyk transform={{4 1 roll pop pop pop 0 0 0 4 -1 roll}}");
optlist = "width=4000 height=3000 bpc=8 colorize=" + devicen;
image = p.load_image("raw", filename, optlist);
```

表 4.5 CMYK プロセスインキの部分集合群に対する、alternate=devicecmyk の場合の DeviceN PostScript 変換関数

CMYK 部分集合に対する DeviceN インキ名	PostScript 変換関数	CMYK 部分集合に対する DeviceN インキ名	PostScript 変換関数
Cyan	{0 0 0}	Magenta Black	{0 3 1 roll 0 exch}
Magenta	{0 0 0 4 1 roll}	Yellow Black	{0 0 4 2 roll}
Yellow	{0 0 0 4 2 roll}	Cyan Magenta Yellow	{0}
Black	{0 0 0 4 3 roll}	Cyan Magenta Black	{0 exch}
Cyan Magenta	{0 0}	Cyan Yellow Black	{0 3 1 roll}
Cyan Yellow	{0 exch 0}	Magenta Yellow Black	{0 4 1 roll}
Cyan Black	{0 0 3 -1 roll}	Cyan Magenta Yellow Black	{ }
Magenta Yellow	{0 0 4 1 roll}		

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/colorize_image_with_DeviceNトピックにあります。

Lab 代替空間を持ったスポットカラー群に基づく DeviceN 色空間 複数のスポットカラーを DeviceN インキとして用いることは、スポットカラーの間のシェーディングの基礎を提供します。デバイス依存な代替色空間に対しては、然るべき PostScript 変換関数を実装するのはかなり大変です。なぜなら RGB や CMYK は非線形な特性を有するからです。Lab 色空間の色を混ぜ合わせるのははるかに容易です。なぜなら知覚的に一様だからです。ですので、各色の Lab 値の加重平均を計算するだけで色を組み合わせることができます。以下の PostScript コードは、2 個の Pantone カラーに対してこの方式を実装しています。その 2 色の Lab 代替値をコードの先頭で与える必要があります：

```
% N=2かつLab代替を持つDeviceNに対するPostScript変換関数。
% 濃度値群を重みとして用いてL/a/b値の加重平均を計算することにより
% Labカラーどうしを混ぜ合わせます

80 28 75                    % 色1=PANTONE 123 UのLab値群
31.7647 0 -17               % 色2=PANTONE 289 UのLab値群

% L値どうしを混ぜ合わせ
7 index 6 index mul         % t1*L1
7 index 4 index mul         % t2*L2
add 9 1 roll                % ボトム: L = t1*L1 + t2*L2

% a値どうしを混ぜ合わせ
7 index 5 index mul         % t1*a1
7 index 3 index mul         % t2*a2
add 9 1 roll                % ボトム: a = t1*a1 + t2*a2

% b値どうしを混ぜ合わせ
7 index 4 index mul         % t1*b1
7 index 2 index mul         % t2*b2
add 9 1 roll                % ボトム: b = t1*b1 + t2*b2

% 2個の濃度値と2x3個の入力カラー値をポップ
pop pop pop pop pop pop pop pop pop
% 結果: 混ぜ合わされた色のLab値群
```

上記の PostScript 関数 ***transformFunc2*** を用いて、2 個の Pantone スポットカラーに基づいて DeviceN 色空間を作成するには、以下のようにします :

```
devicen = p.create_devicen(
        "names={{PANTONE 123 U} {PANTONE 289 U}} alternate=lab " +
        "transform={{" + transformFunc2 + "}}");
```

Lab 代替値を持った別々のスポットカラーに基づいてシェーディングが構築されているときには、PDFlib は自動的にそのような DeviceN 色空間を構築します。

クックブック さらにさまざまなコードサンプルと、Lab に基づくもっと大きな N 値に対する PostScript 変換関数が、クックブックの `color/devicen_color` トピックにあります。

4.6 シェーディングとシェーディングパターン

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

スムーズシェーディングは、カラーブレンドまたはグラデーションとも呼ばれ、同じ色空間にある複数の色の間の連続的な遷移を与えます。たとえば 2 個の RGB カラーの間や、1 個のスポットカラーの 2 個の濃度の間などです。シンプルなシェーディングは、2 個の色の間の遷移を定義します。1 番目の色は、*PDF_shading()* の *startcolor* オプションかカレント塗り色から採られます。2 番目の色は、*endcolor* オプションか *c1*・*c2*・*c3*・*c4* 引数で与えられます。任意の数の中間色を持つシェーディングを作成することもでき、そのためには *startcolor/endcolor* でなく *stopcolors* オプションを用います。

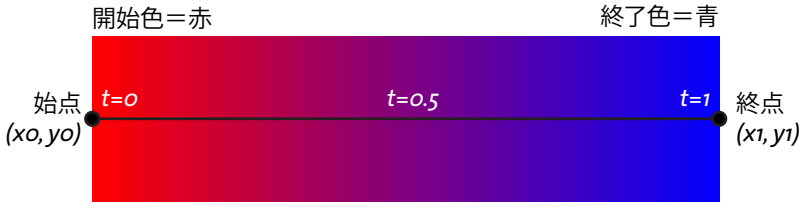
そのシェーディングカラーは変数 *t* によって制御されます。*t* は開始色における 0 から終了色における 1 まで線形的に変化します。この線形的な遷移を指数的な遷移へ変えることもでき、そのためには *N* オプションを指数として用います。RGB・CMYK 色空間は知覚的に一様ではありませんので、その得られるブレンドは意図どおり滑らかに見えない場合があります。Lab 色空間の開始・終了色を用いてブレンドを指定すれば、より滑らかな色遷移を得られる可能性があります。PDFlib ではシェーディングに 2 種類の形状を使用できます：

- 直線状シェーディング（*type=axial*）：始点と終点の間の直線に沿って色が変化します（図 4.2 参照）。そのシェーディングカラーは、始点 *(xo, yo)* から終点 *(x1, y1)* まで線形的に変化します。シェーディングを、始点・終点の向こうの境界色を用いて拡張することも可能で、そのためには *extend0*・*extend1* オプションを用います。
- 放射状シェーディング（*type=radial*）：2 個の円の間で色が変化します（図 4.3 参照）。放射状シェーディングを利用すると、球の 3 次元風な視覚表現を作れます。そのシェーディングカラーは、中心 *(xo, yo)* と半径 *ro* を持つ開始円から、中心 *(x1, y1)* と半径 *r1* を持つ終了円まで変化します。片方の円を縮めて点にすることも可能です。

PDF_shading() は、シェーディングオブジェクトへのハンドルを返します。このハンドルの使い道は 2 つあります：

- *PDF_shfill()* を用いて領域を直接塗る。塗りたいオブジェクトの形状がシェーディングの形状と同じである場合にはこの方式が良いでしょう。この関数は、その名前に反して、オブジェクトの内側を塗るだけでなく、外側にも影響します。この動作は、*PDF_clip()* を用いて変えることもできます。
- もっと複雑なオブジェクトを塗るために、シェーディングパターン（タイリングパターンと混同しないでください）を定義して使用する。具体的には、*PDF_shading_pattern()* を呼び出すことによってシェーディングに基づいてパターンを作成し、このパターンを使用して任意のオブジェクトを塗ったり描線したりします。

図 4.2
2 色の間の直線状シェーディングの主なパラメータ



2 個のプロセスカラーの間のシェーディング　以下のコードは、RGB 色空間における赤から青までの直線状シェーディングを作成し、それを使って円を塗ります：

```
sh  = p.shading("axial", 100, 100, 500, 500, 0, 0, 0, 0, "startcolor=red endcolor=blue");
shp = p.shading_pattern(sh, "");
p.set_graphics_option("fillcolor={pattern " + shp + "}");

p.circle(300, 300, 200);
p.fill();
```

複数のスポットカラーの間のシェーディング　シェーディングを使って作成できるのは、同一の色空間に属する色の間の遷移だけですので、開始色と終了色として、任意のスポットカラー群を使ったり、1 個のスポットカラーと 1 個のプロセスカラーを使ったりすることはできません。ただし、Lab 代替値を持ったスポットカラー群に対しては、92 ページ「Lab 代替空間を持ったスポットカラー群に基づく DeviceN 色空間」で示しているのと同様の適切な DeviceN 色空間を用いてシェーディングを実現できます。特定の条件が満たされていれば（PDFlib API リファレンス参照）、ストップカラーとしてのスポットカラー群に対してこのような DeviceN 色空間を PDFlib は自動生成します。以下のコード断片は、2 個のスポットカラーの間のシェーディングを作成します：

```
sh  = p.shading("axial", 100, 100, 500, 500, 0, 0, 0, 0,
        "stopcolors={ 0% {spotname {PANTONE 123 U} 1} 100% {spotname {PANTONE 289 U} 1}}"
shp = p.shading_pattern(sh, "");
p.set_graphics_option("fillcolor={pattern " + shp + "}");

p.circle(300, 300, 200);
p.fill();
```

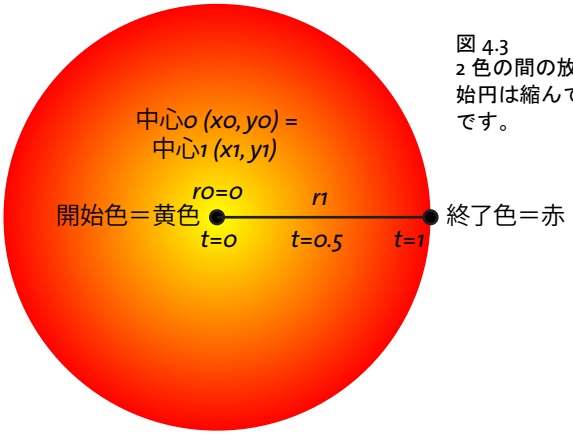


図 4.3
2 色の間の放射状シェーディングの主なパラメータ。開始円は縮んで点になっています。2 つの円の中心は同一です。

4.7 タイリングパターン

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

タイリングパターンは、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コピー（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_ext()* と *PDF_end_pattern()* の間で、描画オペレータ群を用いて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_ext()* によって返されたパターンハンドルは、*PDF_set_graphics_option()* または *PDF_setcolor()* でオプション *fillcolor/strokecolor* を用いて、パターンをカレントカラーとして設定するために用いることができます。

PDF_begin_pattern_ext() の *painttype* オプションによって、パターン定義がそれ自身の色指定群を含むことができるかどうかが決まります。*painttype=colored* 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=uncolored* 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。

図 4.4
タイリングパターンを使って円とテキストを塗る。さらに強調のためにテキストの輪郭を描線しています。

4.8 透過ブレンドモード

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

ブレンドモードは、透過イメージングモデル（各オブジェクトがその背後にあるすべてのオブジェクトを完全に覆い隠す不透明イメージングモデルではなく）において、オブジェクトの色を背景とどのように混ぜ合わせるかを制御します。背景色は、複数の他のオブジェクトをブレンドすることによって作成されている可能性もありますので、これを**バックドロップ**と呼びます。描画されるオブジェクトを**ソース**と呼びます。ソースとバックドロップの色は、何らかのブレンド機能を用いて互いに混ぜ合わされます。表 4.6 に挙げるブレンドモードは、さまざまな芸術的効果を生み出し、また、特殊な用途に利用することもできます（100 ページ「4.9.1 ブレンドモードを用いて色を変える」参照）。利用可能なブレンドモードは以下のように分類できます：

- デフォルトのブレンドモード *Normal*：ソースを完全に不透明にします。すなわち、ソースオブジェクトが完全に背景を置き換えます。ブレンドモードを *Normal* 以外に設定すると、いかなる不透過設定や画像透過にもよらず、ソースオブジェクトは暗黙的に透過になります。
- ブレンドモード *Darken*・*Multiply*・*ColorBurn*：暗くする効果を生み出します。
- ブレンドモード *Lighten*・*Screen*・*ColorDodge*：明るくする効果を生み出します。
- ブレンドモード *Overlay*・*SoftLight*・*HardLight*：明るい領域と暗い領域を加減することによってコントラストを強くします。
- 差分ブレンドモード *Difference*・*Exclusion*：カラー値の差をとります。白いオブジェクトを白バックドロップの上に描画して結果が白にならないのはこれらのモードだけです。
- ブレンドモード *Hue*・*Saturation*・*Color*・*Luminosity*：HSL カラー表現の次元群に対する効果によって定義されます。HSL モデルにおいて、色相（hue）は知覚される色を記述し（赤と緑は色相が異なる）、彩度（saturation）は色がどのくらいカラフルに見えるか（その色にグレーがどのくらい混ぜられているか）を記述し、輝度（luminosity）は色がどのくらい暗く、または明るく見えるかを記述します（白が最大輝度）。

表 4.6 ブレンドモードと、ソースオブジェクトを背景オブジェクト（バックドロップ）にブレンドさせる際の効果

ブレンドモード	ISO 32000-2 に従った説明
ブレンド効果のないブレンドモード	
None	PDFlib 独自のブレンドモード。ブレンドモード指定のない gstate を作成します。外側の gstate で設定されているブレンドモードを確実に効かせます。
Normal	ソース色を採り、バックドロップを無視します。
暗くするブレンドモード	
Darken[1]	バックドロップとソースのうち暗いほうの色を採ります。ソースのほうが暗い場所ではバックドロップはソースへ置き換えられます。そうでない場合には変更されません。
Multiply[1]	バックドロップとソースのカラー値を乗算します。結果の色は少なくともこの 2 色のどちらかと同じ暗さになります。加法的な色を扱う場合には、任意の色を黒と乗算すると結果は黒になり、白と乗算すると元の色は変更されません。減法的な色の場合には、その色空間のすべてのインキに対して使用されている濃度の最大値が、加法空間における黒と同じ役目をします。いくつものオブジェクトが連なって重なり合っている場合には、それらが黒でも白でもなければ、だんだん暗い色が描画されます。
ColorBurn	ソースカラーを反映してバックドロップカラーを暗くします。白で描画した場合には変化ありません。

表 4.6 ブレンドモードと、ソースオブジェクトを背景オブジェクト（バックドロップ）にブレンドさせる際の効果

ブレンドモード	ISO 32000-2 に従った説明
明るくするブレンドモード	
Lighten[1]	バックドロップとソースのうち明るいほうの色を採ります。ソースのほうが明るい場所ではバックドロップはソースへ置き換えられます。そうでない場合には変更されません。
Screen[1]	バックドロップとソースのカラー値の補数を乗算し、結果の補数をとります。結果の色は少なくともこの 2 色のどちらかと同じ明るさになります。加法的な色を扱う場合には、任意の色を白とスクリーンすると結果は白になり、黒とスクリーンすると元の色は変更されません。減法的な色の場合には、その色空間のすべてのインキに対して使用されている濃度の最大値が、加法空間における黒と同じ役目をします。複数のスライド写真を同時に 1 つのスクリーンへ映し出したような効果になります。
ColorDodge	ソースカラーを反映してバックドロップカラーを明るくします。黒で描画した場合には変化ありません。
コントラストを強めるブレンドモード	
HardLight	ソースカラー値が 0.5 より小さければ 2 色を乗算し、0.5 より大きければスクリーンします。バックドロップにきついスポットライトをあてたような効果になります。
SoftLight	ソースカラー値が 0.5 より小さければ 2 色のうち暗いほうの色を採り、0.5 より大きければ明るいほうの色を採ります。バックドロップにやわらかいスポットライトをあてたような効果になります。
Overlay	バックドロップカラー値に応じて 2 色を乗算かスクリーンします。ソースカラーが、バックドロップのハイライトと影を温存しつつその上にかぶさります。バックドロップカラーは置き換えられるのではなく、バックドロップの明るさや暗さを反映するようにソースカラーと混ぜ合わされます。
差分ブレンドモード	
Difference[1,2]	2 色のうち明るいほうの色から暗いほうの色を減算：白で描画するとバックドロップカラーが反転されます。黒で描画すると何も変化しません。減法的な色の場合には、その色空間のすべてのインキに対する濃度の最大値が、加法空間における黒と同じ役目をします。
Exclusion[1,2]	Difference モードと似た効果になりますが、コントラストはより弱くなります。白で描画するとバックドロップカラーが反転されます。黒で描画すると何も変化しません。減法的な色の場合には、その色空間のすべてのインキに対する濃度の最大値が、加法空間における黒と同じ役目をします。
HSL ブレンドモード	
Hue[2]	ソースカラーの色相を持ち、バックドロップカラーの彩度と輝度を持つ色を作成します。
Saturation[2]	ソースカラーの彩度を持ち、バックドロップカラーの色相と輝度を持つ色を作成します。バックドロップの純粋な灰色（彩度なし）の部分にこのモードで描画しても何も変化ありません。
Color[2]	ソースカラーの色相と彩度を持ち、バックドロップカラーの輝度を持つ色を作成します。これはバックドロップのグレーレベルを温存しますので、白黒の画像に色をつけたり、カラー画像に濃度をつけたりするのに有用です。
Luminosity[2]	ソースカラーの輝度を持ち、バックドロップカラーの色相と彩度を持つ色を作成します。これは Color モードと同じ効果を生み出しますが、ただしソースとバックドロップが入れ替わっています。

1. このモードは対称です。すなわち、ソースカラーとバックドロップカラーを入れ替えてもブレンド操作の結果は変わりません。
2. スポットカラーに対してはこのモードは何ら影響を与えず、ブレンドモード Normal がかわりに使われます。

ブレンド色空間　PDF ビューアは、すべての透過計算をブレンド色空間において行います。この色空間は重要な役割を果たします。なぜなら、どの色空間を選択するかによってブレンドの結果が変わってくるからです。これは以下のとおり決定されます：

- *transparencygroup* ページオプションの *colorspace* サブオプションがある場合には、そこで指定されている色空間。
- ない場合には、PDF/X または PDF/A 出力インテント ICC プロファイルがもしあるなら、それが使用されます。PDF/X では出力インテントは必ずあります。PDF/A 文書が出力インテントを含んでいない場合には、それは *transparencygroup* ページオプションの *colorspace* サブオプションを指定する必要があります。
- ないなら、出力デバイス（ソフトプルーフ／出力プレビューの場合には、シミュレートされた出力デバイス）のネイティブ色空間が使用されます。ブレンド色空間を明示的に指定することによってこの状況を避けることを推奨します。なぜなら、そうでないと透過オブジェクトの視覚表現が出力デバイスに依存してしまうからです。

以下のコード断片は、DeviceRGB をブレンド色空間として指定し、ブレンドモード *Multiply* を指定しています：

```
p.begin_page_ext(0, 0, "width=a4.width height=a4.height " +
                       "transparencygroup={colorspace=DeviceRGB}");

gstate = p.create_gstate("blendmode=Multiply");
p.set_gstate(gstate);
```

複数のブレンドモード　PDF では、いっときに 1 つのブレンドモードしかグラフィックステート内で有効にすることができません。複数のブレンドモードの効果を適用したい場合には、すでにブレンドされたバックドロップとオブジェクトの上に重ねて別のオブジェクトを描画することができます。たとえば、何かほかのブレンドモードの結果を色反転させたい場合には、まずバックドロップとオブジェクトを何らかのブレンドモードでブレンドしたのち、ブレンドモードを *Difference* へ切り替えて、そして最初のブレンド操作の結果の上に重ねて白い長方形を描画します。

4.9 オブジェクトの色を変更

PDFlib では、オブジェクトの色を変更する手段をいくつか用意しています。これによって、ページ上に配置されている取り込み PDF ページ、ラスタ画像、SVG グラフィック、任意のテキスト・ベクトル要素の色を変更することができます：

- ラスタ画像には着色することができます。199 ページ「8.1.5 画像にスポットカラーか DeviceN カラーで着色」参照。
- 任意のオブジェクトの色を、ブレンドモードを用いて変更できます。
- 輝度ソフトマスクを持った gstate を用いて、任意のオブジェクトの色をつけることができます。

4.9.1 ブレンドモードを用いて色を変える

クックブック **この項で述べるすべての効果のためのコードサンプルがクックブックの** color/blendmode_effects **トピックにあります。**

この項ではブレンドモードの有用な応用をいくつか紹介します。利用できるすべてのブレンドモードの説明は 97 ページ「4.8 透過ブレンドモード」にあります。すべての例において、オブジェクト（取り込み画像・PDF ページ・SVG グラフィックなど）の色が、何らかの形で変更されます。

すべての例は、以下の示す基本的なコード断片を用いて実装できます。このコード断片は、ページに対して透過グループ色空間を指定することによってデバイス独自の視覚表現を防いだうえで、何か特定のブレンドモードを設定しています。後続するオブジェクト群が、ここで指定したブレンドモードによって影響されることを防ぐために、このシーケンスは、*save/restore* ペアで囲むべきです：

```
p.begin_page_ext(width, height, "transparencygroup={colorspace=DeviceCMYK}");

// ラスタ画像か取り込みPDFページかSVGグラフィックを配置
p.fit_image(image, 200, 150, optlist);

// 望む色とブレンドすることにより既存内容を着色または脱色または色反転
p.save();
        // 望むブレンドモードを持ったgstateを作成
        gs_blendmode = p.create_gstate("blendmode=Color");
        p.set_graphics_option("fillcolor=red gstate=" + gs_blendmode);
        p.rect(0, 0, width, height);
        p.fill();
p.restore();
```

オブジェクトに任意の色で着色　ブレンドモード *Color* を使うと、オブジェクトに何か新しい色を着色できます。そのオブジェクトの、知覚されるグレーレベルが、追加される色の濃度値を制御し、かつ、そのオブジェクトの元の色は無視されます。オブジェクトに着色するには、ブレンドモード *Color* を適用し、色付きの長方形（または他の形）を、そのオブジェクトの上に重ねて配置します。透過オブジェクトに着色するには、配置オブジェクトの下に白い領域を配置することを推奨します。

逆にソースオブジェクトの白い領域を通してバックドロップカラーが見えるように（暗い領域は変更されないように）したい場合には、ブレンドモード *Multiply* を使用します。

オブジェクトを脱色または色反転　場合によっては、オブジェクトを脱色（彩度をなくす。褪色させる）したい場合もあるでしょう。すなわち、もともと色が付いていたオブ

ジェクトからグレースケール出力を作成することです。その際に、その各部分のグレーレベルは、そのオブジェクトの各部分のもともとの色の、知覚される明るさ（輝度）に照応するようにすることです。これを実現するには、ブレンドモード *Color* を用いて、そのオブジェクトの上に白い領域を重ねます。

ラスタ画像の色は *invert* 画像オプションを用いて反転できます。任意のオブジェクトの色を反転することも可能です。そのためには、ブレンドモード *Difference* を用いて、そのオブジェクトの上に白を重ねます。

4.9.2 ソフトマスクを用いて色を変える

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

画像アルファチャンネルは、一般化して任意のオブジェクトと色空間で利用できます。そのためには、輝度ソフトマスクをグラフィックステート（gstate）内に入れ込んで使用します。このソフトマスクは、取り込み PDF ページや SVG グラフィックなど任意の内容を持ったテンプレートをベースとして構築することができます。このテンプレート内容の輝度（知覚されるグレーレベル）によって、その後に描かれるオブジェクト群の可視性が決定されます：テンプレート内の明るい領域は透明になり（すなわち、描かれるオブジェクト群が見える）、暗い領域はオブジェクトを見えなくします。この技法を実現するには以下の手順に従います：

- テンプレートを、*transparencygroup* オプションとサブオプション *colorspace* を用いて作成します。このテンプレートの中で、任意の描画操作を行なっていきます。
- ソフトマスクを適用した時、ソフトマスクテンプレートの明るい（白い）領域はカレント色で着色され、一方、暗い（黒い）領域に色がつきません。結果として暗い領域と明るい領域が入れ替わります。暗い領域と明るい領域を温存するには、100 ページ「オブジェクトを脱色または色反転」で説明したように、ブレンドモード *Difference* を用いてテンプレートカラーを反転させるとよいでしょう。
- このテンプレートをベースとして、gstate を、*softmask* オプションリストとサブオプション *type=luminosity* を用いて作成します。デフォルトではソフトマスクの背景は黒すなわち不透明で初期化されます。これは *backdropcolor* オプションを白（テンプレートの *transparencygroup* オプションの *colorspace* サブオプションに従った然るべき数の色要素を持った）に設定することによって変えることができます。
- この gstate を設定します。save/restore シーケンスで囲むことが多いでしょう。そうすれば、このソフトマスクの効果をその中だけにとどめることができます。この gstate 内のソフトマスクテンプレートの幾何情報はカレント座標系に依存します。たとえば、ソフトマスクの位置を制御するには、gstate を設定する前に平行移動を適用します。以上が終わったら、ページ上でマスクされる内容を描きます。

以下のコード断片は、このシーケンスを実装しています。マスクの反転も行い、それを適用して、取り込み SVG グラフィックファイルに着色しています：

```
// ソフトマスクの定義に使うテンプレートを作成
tpl = p.begin_template_ext(595, 842, "transparencygroup={colorspace=devicecmyk}");
    // 任意の内容を配置。例：SVGグラフィック
    p.fit_graphics(graphics, 0, 0, "boxsize={595 842} position={center} fitmethod=meet");

    // テンプレートの輝度を反転させて、暗い領域と明るい領域が入れ替わるのを防ぐ
    // 色反転のためのブレンドモードを用いたgstateを作成
    gstate_invert = p.create_gstate("blendmode=Difference");
    p.set_graphics_option("fillcolor=white gstate=" + gstate_invert);
    p.rect(0, 0, 595, 842);
```

```
    p.fill();
p.end_template_ext(0, 0);


// ソフトマスクを用いたgstateを有効にしたうえで、色を適用
// 結果: SVGグラフィックが赤く着色されます
p.save();
    // このテンプレートをベースにしたソフトマスクを用いたgstateを作成
    // 「backdropcolor」オプションでテンプレート領域を透明に初期化
    gstate_softmask = p.create_gstate(
        "softmask={type=luminosity template=" + tpl + " backdropcolor={0 0 0 0} }");

    p.set_graphics_option("fillcolor=red gstate" + gstate_softmask);
    p.rect(0, 0, 595, 842);
    p.fill();
p.restore();
```

複数のソフトマスク　PDF は、1 つの合成操作について 1 つまでのソフトマスクにのみ対応しています。たとえば、画像が自分のアルファチャンネル（＝ソフトマスク）を持っている場合には、グラフィックステートで与えられたソフトマスクは何の効力も及ぼしません。この PDF の制約を回避するには、テンプレート内に内容を生成して、このテンプレートをページ上に配置する必要があります。これによって事実上、PDF ビューアに強制的に別々の合成操作を適用させ、ソフトマスクが使用可能になります。

4.10 レンダリングインテント

PDFlib アプリケーションはデバイス独立なカラー値を指定できますが、ある特定の出力デバイスにおいて、その色を正確に再現できないことがあるかもしれません。この場合には、何らかの妥協が行われる必要があります。この処理を色域圧縮といいます。これは、色の範囲を、そのデバイスによって再現可能な、より狭い範囲へ縮める処理です。レンダリングインテントは、この処理を制御するために使用するものです。レンダリングインテントは個別の画像に対して指定できます。そのためには *PDF_load_image()* に *renderingintent* オプションを与えます。また、レンダリングインテントをテキストとベクトルグラフィックに対して指定することもできます。そのためには *PDF_create_gstate()* に *renderingintent* オプションを与えます。利用可能なすべてのレンダリングインテントを表 4.7 に挙げます。

表 4.7 利用可能なレンダリングインテント

インテント	説明	典型的用途
Auto	レンダリングインテントを PDF ファイル内で一切指定せず、デバイスのデフォルトインテントをかわりに使用	ページ内容に関して充分な情報が得られません
AbsoluteColorimetric	デバイスの白色点（紙色の白など）に対し補正を一切行わない。色域外の色は、デバイスの色域の中で最も近い値へマップされます。	ベタ色の正確な再現。それ以外の用途には推奨されません
RelativeColorimetric	色データをデバイスの色域へ圧縮。白色点を互いにマップし、それ以外の色を若干シフトさせます。	ベクトルグラフィック
Saturation	色の彩度を温存。カラー値はシフトされる場合があります。	ビジネスグラフィック
PerceptualColor	色の関係を温存。見た目の美しさを優先して、色域内の色も色域外の色も変更します。	現実画像

4.11 オーバープリント制御

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

デフォルトでは、ある特定のカラーチャンネルに描画を行うと、他のカラーチャンネル群の照応する領域は置き換わります。たとえば、CMYK デバイス上の色 0/0/1/0、すなわちイエローは、そのページ上の同じ場所にそれまでに描画されていたオブジェクト群の既存のシアン・マゼンタ・黒チャンネルを消去します。この動作を変えるにはオーバープリントという処理を用います。この処理は、印刷デバイス上における色インキの混合を制御します。オーバープリントを利用して、黒を他の暗い色の上に重ねて印刷することにより、「リッチブラック」にすることもできます。

PDF のオーバープリントフラグは CMYK 印刷を対象としています。このフラグは、ある特定のチャンネルへの描画が他のチャンネル群の既存の内容に影響するかどうかを制御します。たとえば、シアンをイエローの上に重ねて印刷すると、その結果は、オーバープリントありであれば緑になります（シアンとイエローが混合される）。逆にオーバープリントなしで同じことをすると、その結果は、最後に印刷した色であるシアンとなります。なぜなら、純粋なシアンにはイエローの成分が一切ないので、0% イエローが既存のイエローの内容を置き換えるからです。

塗り・描線操作に対するオーバープリント設定 オーバープリント動作は、描線操作とそれ以外の操作とで別々に制御できます。そのためにはグラフィック書式オプション *overprintstroke*（描線操作に対して）または *overprintfill*（描線以外のすべての操作に対して。画像の配置に対しても）を用います。どちらのオプションもデフォルトは *false* です：

- *overprintfill=false* か *overprintstroke=false* の場合には、塗りまたは描線は、どのカラーチャンネルにおいても、指定されていないインキの、照応する領域を置き換えます：前面の色が勝ちます。
- これらのオプションが *true* の場合、かつ出力デバイスがオーバープリントに対応している場合には、指定されていないインキにおけるそれまでのマーキング群に対して、変更は行われません。オーバープリントは通常、かなり暗い色に対して使用されます。

CMYK カラーに対するオーバープリントモード CMYK オブジェクトのオーバープリント動作は、さらに、*overprintmode* グラフィック書式オプションを用いて変更することもできます。このオプションは、*overprintfill/overprintstroke=true* の場合のゼロ CMYK 成分の動作を制御します：

- *overprintmode=0*（デフォルト）：それぞれの色成分が、それまでに配置されたマーク群を置き換えます（「前面の色が勝つ」）。言い換えれば、前面のオブジェクトの色のすべてのチャンネルが、下に重なっているオブジェクト群をヌキにします。
- *overprintmode=1*（「Illustrator のオーバープリントモード」と呼ばれることもあります）：濃度値 0 は、それまでに描画された色の、照応する成分を、0 に設定するのではなく、変更しません（「前面の濃度値 0 は無視される」）。言い換えれば、0 以外の値を持った前面チャンネルのみが、下に重なっているオブジェクト群をヌキにします。これは、DeviceN 色空間において、0 でない値を持った CMYK 成分のみで描画するのと同じことです。

つまり、CMYK 濃度値がわずか違うだけで（たとえば 0% か 1% か）、オーバープリントモードによって結果に大きな違いが生じる場合があります：オーバープリントモード 0 の場合にはその結果はそれぞれ 0% と 1% になります。オーバープリントモード 1 の場合には、その結果はそれぞれ背景色（それが何であれ）か 1% になります。オーバープリントモードは、その定義により、CMYK 以外の色空間においては何も影響を与えません。

オーバープリントによって影響を受けるのは DeviceCMYK 色空間の色だけです。DeviceN 色空間で Cyan・Magenta・Yellow・Black 成分のいずれかが使われていても、この CMYK オーバープリント規則は適用されません。

デバイス依存なオーバープリント表現　PDF ビューアは、出力デバイスがオーバープリントに対応していない場合には、オーバープリント設定を無視しなければならないことになっていますので、オーバープリントオブジェクトを持った文書は本質的にポータブルではなく、意図したページ表現を実現するには、PDF ビューアや RIP を特別に構成する必要が生じる場合があります。Acrobat が画面表示においてオーバープリント設定に従うかどうかは、**「編集」→「環境設定」→「ページ表示」→「オーバープリントプレビューを使用」**を通じて構成可能です。オーバープリント設定を使って作業しているときには、この環境設定を**「常時」**に設定することによって正しい画面表示が得られるようにすることを推奨します。さらに、Acrobat の**「出力プレビュー」**パネルには項目**「オーバープリントをシミュレート」**が設けられています。

ページ上で透過かブレンドモードが使用されている場合には、Acrobat のオーバープリントプレビューは、カレントのブレンド色空間が DeviceCMYK 色空間を使用するよう設定されている場合のみ、期待どおり動作します。そのように設定するには ***PDF_begin_page_ext()*** で以下のオプションを用います：

```
transparencygroup={colorspace=devicecmyk}
```

オーバープリントプレビュー対応のないサードパーティ PDF ビューアは、期待と違う色を表示する場合があります。実際、シンプルな PDF ビューアの多くはオーバープリントに対応していません。

デバイス独立なオーバープリント効果　上述のビューア依存性により、オーバープリントの表現はもろくなる場合があります。オーバープリント効果をデバイス独立な方式で実現することもできます。そのためにはブレンドモード ***Darken*** を以下のように使用します：

```
gstate = p.create_gstate("blendmode=darken");
p.set_gstate(gstate);
```

オーバープリントシミュレーションが有用となる状況としては、一つには、色分版の再合成が挙げられます。画像やページのシアン・マゼンタ・イエロー・黒分版を持っているとして、これらを 1 個のコンポジットな PDF ページへ再合成する必要がある場合に、***Darken*** ブレンドモードを用います。

クックブック　**CMYK 分版を再合成しているコードサンプルが、クックブックの `color/recombine_color_channels` トピックにあります。**

5 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にも対応しています。

5.1 Unicode の重要な諸概念

キャラクタとグリフ　テキストを扱う際には、以下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を表現します。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 5.1 参照）。

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

- **基本多言語面**（*Basic Multilingual Plane ＝ 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**（*Private Use Area ＝ PUA*）：私用のために予約されている複数の Unicode 領域から成ります。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 PUA 領域 U+E000 ～ U+F8FF 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 5.1
グリフとキャラクタの関係

Unicode の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 ビット値のペアで表されます。このようなペアをサロゲートペアといいます。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。

Unicode の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 5.1 に示します。

表 5.1 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

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. 四角 ■ は null バイトを意味します。

5.2 Unicode 対応言語バインディング

PDFlib API の機能のなかには、使用する言語バインディングが Unicode 対応かどうかによって変化するものがあります。この概念をこの節と次の節で解説します。

5.2.1 ネイティブ Unicode 文字列のある言語バインディング

プログラミング言語が Unicode 文字列をネイティブに用いている場合に、そのバインディングを Unicode 対応と呼ぶことにします。以下の PDFlib 言語バインディングは Unicode 対応です：

- C++
- .NET・.NET Core
- Java
- Objective-C
- Python
- RPG

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、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 のオプションを自動的に設定します。このことから以下の結果が生じます：

- クライアントが与える文字列はすべて、Unicode エンコーディングかつ UTF-16 形式で PDFlib にもたらされます。
- API 解説内のいろいろな文字列種別（内容文字列・ハイパーテキスト文字列・名前文字列）どうしの違いは意味を持ちません。オプション ***textformat***・***hypertextformat***・***hypertextencoding*** は不必要であり、かつ許されません。テキストフローオプションは強制的に ***true*** になります。
- 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 レガシ CMap を用いることは（118 ページ「5.5 日本語・中国語・韓国語 CMap」参照）できません。なぜなら、ラッパがつねに Unicode を PDFlib カーネルに与えるからです。Unicode CMap のみが利用可能です。

要するに基本的には、クライアントはネイティブ Unicode 文字列を PDFlib API 関数に与えることができ、その際に別途構成は必要ないということです。

5.2.2 UTF-8 対応のある言語バインディング

ネイティブ Unicode 文字列データ型を持たないプログラミング言語であっても、Unicode 文字列を UTF-8 形式で取り扱うことが可能です。以下の PDFlib 言語バインディングは、***stringformat=utf8***オプションを設定することによってUnicode対応にすることができます：

- C
- Perl
- PHP
- Ruby

これらの言語バインディングのうちのいずれかで作業をする場合には、UTF-8 が推奨されます。新規 PDFlib オブジェクトを作成した直後に以下の関数呼び出しを用いれば、言語バインディングを Unicode 対応にすることができます：

```
p.set_option("stringformat=utf8");
```

アプリケーション内で Unicode 処理が必要な場合には、上記の呼び出しを用いて言語バインディングを UTF-8 に基づいて Unicode 対応にすることを推奨します。この呼び出しの後には、言語バインディングは Unicode 対応バインディングであるかのように動作します。ただし、クライアントは必ず UTF-8 文字列をすべての API 関数に与える必要があります。この呼び出しには以下の結果もあります：

- API におけるすべての文字列、すなわち名前文字列・内容文字列・ハイパーテキスト文字列・オプションリストは、BOM ありかなしの UTF-8 形式と見なされます。
- C 言語バインディングの場合には、***length*** 引数に 0 より大きな値が与えられたときには、関数引数としての名前文字列は依然として UTF-16 として解釈されます。

Unicode 変換　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、UTF-8 か UTF-16 形式の Unicode に変換する必要があります。これは、***PDF_convert_to_unicode()*** を用いて、あるいは言語独自の方式で実現できます。31 ページ「2 章 PDFlib の言語バインディング」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。

5.3 非 Unicode 対応言語バインディング

以下の PDFlib 言語バインディングは、デフォルトでは Unicode 対応ではありません：

- C
- Perl
- PHP
- Ruby

これらの言語バインディングを *stringformat* オプションで Unicode 対応にすることを推奨します（109 ページ「5.2.2 UTF-8 対応のある言語バインディング」参照）。この節の残りは、上記の言語のいずれか一つで書かれた、かつオプション *stringformat=utf8* を設定しないアプリケーションについてのみ意味を持ちます。

Unicode 変換　PDFlib は、UTF-8・UTF-16・UTF-32 文字列間の変換を行なったり、任意のエンコーディングから BOM の有無を選択して Unicode への変換を行ったりすることのできる *PDF_convert_to_unicode()* 関数を提供しています。

BOM 付き UTF-8 形式は、C ユーザーにとって、PDFlib がそのような文字列を BOM を通じて自動的に認識するという利点があります。これによって、フォントを *encoding=unicode* で読み込むこと、およびハイパーテキスト文字列を *hypertextencoding=unicode* で取り扱うこと、名前文字列を *usehypertextencoding=true* で取り扱うことが可能となり、ひいては完全な Unicode ワークフローが実現できます。

31 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。

Unicode 処理と文字列種別　Unicode 文字列は、非 Unicode 対応言語でも使用できますが、文字列処理は少し複雑になり、文字列の種別に依存します。PDFlib API は、内容文字列・ハイパーテキスト文字列・名前文字列（これらの呼称は歴史的な誤称です）という文字列種別を使用します。引数とオプションは、PDFlib API リファレンス内で、これらの種別のいずれか一つとして示されています。これらの文字列種別の取り扱いを、表 5.2 にまとめるとともに、以下の各項で解説します。

表 5.2　さまざまな文字列種別に対する文字列処理の概要

文字列種別	サンプル引数・オプション	関連オプション / 解釈
内容文字列	PDF_fit_textline( ) ・ PDF_add_textflow( ) の text 引数。	textformat encoding
ハイパーテキスト文字列	► 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	hypertextformat hypertextencoding
名前文字列	► PDF_begin_document( ) ・ PDF_create_pvf( ) の filename 引数 ► PDF_load_font( ) の fontname 引数 ► PDF_load_iccprofile( ) の profilename 引数	usehypertextencoding, filenamehandling
オプションリスト内の文字列		BOM 付き：UTF-8 BOM なし：文字列種別による

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。

内容文字列の解釈は、*PDF_load_font()* の *textformat* オプション（後述）と *encoding* 引数またはオプション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。C 言語では、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　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、PDFlib API リファレンスでは**「ハイパーテキスト文字列」**として示しています。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。

ハイパーテキスト文字列の解釈は *hypertextformat*・*hypertextencoding* オプション（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。C 言語では、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、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、PDFlib API リファレンスでは**「名前文字列」**として示しています。これはハイパーテキスト文字列とわずかに違います。
ファイル名は特殊な場合です：オプション *filenamehandling* は、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用できる文字列へどのように PDFlib が変換するかを指定します。

名前文字列の解釈は、名前文字列とわずかに異なっています。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、名前の先頭に UTF-8 BOM があるときは、それは 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=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非 Unicode 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。Unicode の名前文字列を作成するには、文字列を UTF-8 へ変換する必要があります。

内容文字列とハイパーテキスト文字列に対するテキスト形式　Unicode 文字列は、UTF-8 か UTF-16 か UTF-32 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* オプションでページ記述の全テキストに対して、*hypertextformat* オプション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。この両オプションで対応している値を表 5.3 に示します。*[hyper]textformat* オプションのデフォルトは *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* オプションを用います。*hypertextencoding* オプションのデフォルトは *auto* です。

表 5.3 textformat・hypertextformat オプションに対する値

[hyper]textformat	説明
bytes	文字列の 1 バイトが 1 キャラクタに照応します。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（IBM System i・IBM Z 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 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 付きの 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 オプション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* 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 5.4 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（133 ページ「グリフ置換」参照）。

オプションリスト　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。なぜなら非 Unicode 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以下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、その照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* 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 *auto* エンコーディングで解釈されます。

表 5.4 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	115 ページ「自動エンコーディング」の項を参照	
unicode と 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]
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。

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：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）。

5.4 シングルバイト（8 ビット）エンコーディング

注 この節の情報は Unicode ワークフローでは必要ではないでしょう。

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 つのキャラクタへマップします。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 は、以下のエンコーディングの内蔵定義を持っています：

winansi（cp1252と等しい。iso8859-1のスーパーセット），
macroman（オリジナルMacintosh文字集合），
macroman_apple（macromanとほぼ同じ、ただし通貨をユーロで置き換え），
ebcdic（EBCDICコードページ1047），ebcdic_37（EBCDICコードページ037），
pdfdoc（PDFDocEncoding），
iso8859-1，iso8859-2，iso8859-3，iso8859-4，iso8859-5，iso8859-6，iso8859-7，iso8859-8，
iso8859-9，iso8859-10，iso8859-13，iso8859-14，iso8859-15，iso8859-16,s
cp1250，cp1251，cp1252，cp1253，cp1254，cp1255，cp1256，cp1257，cp1258

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、以下の 8 ビットエンコーディングへマップされます：

- MVS か USS を持つ IBM System Z では *ebcdic* へマップされます。
- IBM System i では *ebcdic_37* へマップされます。
- Windows では *winansi* へマップされます。
- それ以外のすべてのシステムでは *iso8859-1* へマップされます。

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。

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 の場合：カレントのシステムコードページ（詳しくは後述）
- Unix・macOS の場合：*iso8859-1*
- IBM System i の場合：カレントジョブのエンコーディング（*IBMCCSIDoooooooooooo*）
- IBM System Z の場合：*ebcdic*（＝コードページ 1047）

記号フォントでは、キーワード *auto* は *builtin* エンコーディングへマップされます（138 ページ「6.4.2 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。

auto エンコーディングは、非 Unicode 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（111 ページ「5.3 非 Unicode 対応言語バインディング」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。

システムコードページを流用　PDFlib は、コードページ定義をシステムから取得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 181 ページ「7.5.1 TrueType・OpenType 日中韓フォントを用いる」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 へマップされます。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、119 ページ「カスタム日中韓フォントのためのコードページ」参照）。
- Linux では、***iconv* 機能**が対応しているすべてのコードセット識別子を使えます。
- IBM System i 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 System Z 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文字集合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 文字集合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 リソースとして（57 ページ「3.1.4 リソース構成とファイル検索」参照）構成する。
- 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。

エンコーディングファイルは、グリフ名とコードを 1 行ずつ列挙したものです。エンコーディング定義の冒頭部分は以下のようになります：

```
% 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。グリフ名でなく 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
...
```

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。
- 各行内の先頭エントリは PostScript グリフ名か 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、その照応する 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。

5.5 日本語・中国語・韓国語 CMap

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 PDF では、キャラクタコレクションとキャラクタマップ（*CMap*）という概念に対応してフォント内のキャラクタを組織化しています。

注 CMapは主にレガシ日中韓エンコーディングに対して用いられます。Unicodeベースのワークフローでは必要ありません。*PDF_convert_to_unicode()* 関数や、言語独自・システム独自の方法を用いて、レガシ日中韓エンコーディングの文字列を Unicode へ変換することができます。

注 Unicode 対応の言語バインディングは Unicode CMap（UTF16）にのみ対応しています。他の CMap を使用することはできません。

代表的な日中韓エンコーディングのための定義済み CMap 定義済みの日中韓 CMap を表 5.5 に示します。これらは、macOS・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 CMap もすべてのロケールについて利用可能です。

表 5.5 日本語・中国語・韓国語テキストのための定義済み CMap

ロケール	CMap 名
日本語	UniJIS-UCS2-H/V[1]・UniJIS-UCS2-HW-H/V・UniJIS-UTF16-H/V・83pv-RKSJ-H・90ms-RKSJ-H/V・90msp-RKSJ-H/V・90pv-RKSJ-H・Add-RKSJ-H/V・EUC-H/V・Ext-RKSJ-H/V・H/V
中国語 簡体字	UniGB-UCS2-H/V[1]・UniGB-UTF16-H/V・GB-EUC-H/V・GBpc-EUC-H/V・GBK-EUC-H/V・GBKp-EUC-H/V・GBK2K-H/V
中国語 繁体字	UniCNS-UCS2-H/V[1]・UniCNS-UTF16-H/V・B5pc-H/V・HKscs-B5-H/V・ETen-B5-H/V・ETenms-B5-H/V・CNS-EUC-H/V
韓国語	UniKS-UCS2-H/V[1]・UniKS-UTF16-H/V・KSC-EUC-H/V・KSCms-UHC-H/V・KSCms-UHC-HW-H/V・KSCpc-EUC-H/V

1. UCS2 CMap は廃止。PDF 1.4 を生成する必要がある場合以外は、照応する UTF-16 CMap を使用するべきです。

CMap 構成 定義済み CMap の 1 つを用いて日本語・中国語・韓国語（日中韓）テキストを作成するには、PDFlib は、その照応する 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 へマップするためです。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の中に置くことができ、実行時に *SearchPath* を設定することで、CMap ファイルを手動で構成する必要があります：

```
p.set_option("SearchPath={{/パス/パス/resource/cmap}}");
```

日中韓 CMap ファイルの検索先を構成する方式ではなく、適切な *SearchPath* 定義を含む UPR 構成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。

カスタム日中韓フォントのためのコードページ　PDFlib は表 5.6 に挙げるコードページに対応しています。Windows では PDFlib はさらに、システムにインストールされている任意の日中韓コードページに対応しています。

表 5.6 日中韓コードページ（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	文字集合
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

5.6 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*・*ebcdic* 等）で書くことを要求されます。この場合、8 ビット符号化されたテキストの中へ、一部分だけ Unicode キャラクタ群を含めるのは厄介な問題です。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。

5.6.1 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入れ込める方式をサポートしています。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。バックスラッシュは多くのプログラミング言語において特殊な意味を持ちますので、ソースコード内でリテラルに用いる際には二重にする必要があるかもしれません。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 5.7 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 5.7 エスケープシーケンスとバイト値一覧

シーケンス	長さ	macOS・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うには、*escapesequence* オプションを明示的に *true* に設定する必要があります。必要なところで選択的に *PDF_fit_textline()* 等関数内でこれを行うことも可能です。あるいはすべての内容文字列（すなわちすべてのテキスト出力操作）に対してエスケープシーケンスの展開を有効化することもでき、その方法は以下のとおりです：

```
p.set_text_option("escapesequence=true");
```

エスケープシーケンスの置換は、*PDF_set_option()* を用いてグローバルに有効化することも可能です。このグローバルオプションは、その後に使われるすべての名前文字列・ハイパーテキスト文字列・内容文字列に効力を及ぼします。環境変数の値は名前文字列として扱われますので、それもエスケープシーケンスの展開の対象となります。これはたとえばバックスラッシュキャラクタを含む Windows パスまたはファイル名などについて、望まない動作につながるおそれがあります。ですので、*PDF_set_option()* を用いてエスケープシーケンス置換をグローバルに有効化するのではなく、必要なところだけで選択的に関数を用いて（*PDF_fit_textline()* 等）、あるいは *PDF_set_text_option()* を用いて内容文字列に対してだけ有効化することを推奨します。

escapesequence オプションを設定するのではなく、*PDF_convert_to_unicode()* を用い、入力と出力のエンコーディングを同じにすることによって、文字列内のエスケープシーケンスを置き換えることもできます。例：

```
String s_plain = p.convert_to_unicode("utf16",
        s.character.getBytes("UTF-16"),
        "outputformat=utf16 escapesequence=true");
```

エスケープシーケンスは 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。

5.6.2 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以下の場所にあります：

`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?
```

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、以下のグループの 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 7.4 に挙げるデフォルトシェーピング動作をオーバライドするための制御キャラクタ群。
- 表 7.5 に挙げるデフォルト双方向組版をオーバライドするための制御キャラクタ群。
- 表 9.1 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　グリフ名は以下のソースから導かれます：

- 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以下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（スワ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、フォントが対応していれば OpenType 機能でもっと簡単に実装できます（167 ページ「7.3 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
```

文字参照を用いる　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用いるには、*charref* 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.fit_textline("Price: 500&euro;", x, y, "charref=true");
```

テキストオプションとして *charref* を与えると、すべての内容文字列に対して文字参照置換が有効化されます：

```
p.set_text_option("charref=true font=" + font + " fontsize=24");
p.fit_textline("Price: 500&euro;", x, y, "");
```

この *charref* オプションは、*PDF_set_option()* を用いてグローバルに設定することも可能です。しかし、これはすべての名前文字列・ハイパーテキスト文字列・内容文字列に影響しますので、望まない結果を引き起こすおそれがあることから、推奨されません。

charref オプションを設定するのではなく、*PDF_convert_to_unicode()* を用い、入力と出力のエンコーディングを同じにすることによって、文字列内の文字参照を置き換えることもできます。例：

```
String s_plain = p.convert_to_unicode("utf16",
        s.character.getBytes("UTF-16"),
        "outputformat=utf16 charref=true");
```

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* オプションには従いません。
- 非 Unicode 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）。
- 1 個のアンパサンドキャラクタは、文字参照の開始と見なされます。ただし、その後にセミコロン「;」が見つかり、かつ「&」と「;」の間のキャラクタ群が文字参照として有効である場合に限ります。そうでない場合には、そのアンパサンドキャラクタ「&」は変更されないままとなります。文字参照を開始させずにアンパサンドキャラクタ「&」を表したい場合には、文字参照 *&* を使用することが推奨されます。

- 文字参照の可能性のある文字列が無効の場合（たとえば**&#**の後に無効な10進値群が続いている場合や、「&」の後に未知の実体名が続いている場合）には、その動作はオプション *glyphcheck* に依存します：
  glyphcheck=none：その文字列はそのまま保たれます。
  glyphcheck=replace：その文字列は置換キャラクタで置換されます。
  glyphcheck=error：エラーが発生し、テキスト処理は停止します。*errorpolicy=exception* の場合には例外が発生します。

6 フォント処理

6.1 フォント形式

6.1.1 TrueType フォント

各種 TrueType フォント形式　PDFlib はベクトルベースの TrueType フォントに対応しています。PDFlib は TrueType フォントについて以下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、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用いられています。
- 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。
- macOS 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 フォント名　フォントファイルを扱う際には、フォントに任意の API 名を割り当てることもできます（143 ページ「フォントデータのソース」参照）。この名前は、フォントを読み込む際に用いられ、そのフォントのファイル名やそのフォントの内部名とは異なっていてもかまいません。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 名とは異なります（例：*TimesNewRomanPSMT* に対して *Times New Roman*）。

6.1.2 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、合字やスワ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（167 ページ「7.3 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（174 ページ「7.4 複雑用字系出力」参照）。なお、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 167 ページ「7.3 OpenType レイアウト機能」を参照してください。

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 には以下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「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「O」ロゴで表示します。
- OpenType コレクションは、OpenType 1.7 仕様で導入されました。これは、関連する OpenType フォント群を 1 個の合体ファイル内へまとめるという点において、TrueType コレクションと似ています。OpenType コレクションは **.ttf* または **.otc* 接尾辞を用います。
- OpenType フォントバリエーション。バリアブルフォントとも呼ばれます：OpenType フォントバリエーション機構を用いるフォントを用いると、1 つのフォントファミリに属する複数のフォントフェース（ライト・レギュラー・ボールド等）を、ただ 1 つのフォントリソースの中へパッケージすることができます。PDF はフォントバリエーションに対応していませんのでこの機構を利用できません。TrueType アウトラインを持った OpenType フォントバリエーションを PDFlib で読み込むことはできますが、バリエーションのないデフォルトフォントインスタンスのみが見えます。CFF2 テーブル内に PostScript アウトラインを持った OpenType フォントバリエーションは、PDF と非互換ですので、拒否されます。

6.1.3 WOFF フォント

WOFF（Web Open Font Format＝Webオープンフォント形式）は、TrueType フォントと OpenType フォントのための、単純な圧縮ファイル形式です。これは、既存のファイル形式のための新しいコンテナ形式として捉えることができますが、新たなタイポグラフィ機能を一切提供しません。WOFF は、Web 上での使用のために設計されたもので、小さなフォントファイルサイズを実現するために、圧縮機能とサブセット化機能を提供しています。WOFF は W3C 勧告で説明されています：WOFF の仕様は以下にあります：

```
www.w3.org/TR/WOFF
```

WOFF フォントは通常、ファイル名拡張子 *.woff* を用いています。

PDFlib は、背後の TrueType または OpenType フォントに対応している限りにおいて、WOFF フォントに対応しています。たとえば、TrueType ビットマップフォントが WOFF としてパッケージされたものには対応していません。Windows・Mac OS X オペレーティングシステムは WOFF フォントに対応していないので、これはホストフォントとしては使用できません。

6.1.4 PostScript Type 1 フォント

注 PostScript Type 1 フォントの使用は非推奨になりました。

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: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 のアウトラインデータとメトリックデータのための以下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。

6.1.5 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 で符号化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 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フォールバックフォントとして利用することを推奨します。詳しくは 182 ページ「7.5.3 EUDC・SING フォントによる外字キャラクタ」を参照してください。

6.1.6 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 フォントは以下の用途で有用です:

- 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。以下の例は、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_ext(-1, "glyphname=circle width=1000 boundingbox={0 0 1000 1000}");
p.arc(500, 500, 500, 0, 360);
p.fill();
```

```
p.end_glyph();

p.begin_glyph_ext(-1, "glyphname=ring width=400 boundingbox={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** type3_fonts/starter_type3font・type3_fonts/type3_bitmaptext・type3_fonts/type3_rasterlogo・type3_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：

- フォントが ***encoding=unicode*** で読み込まれている場合、そのグリフは、その Unicode 値で、あるいは ***&.<グリフ名>*** の形のグリフ名参照で指定できます。たとえば：***&.circle;***
- フォントが ***encoding=builtin*** で読み込まれている場合、文字コードを用いてグリフを指定できます。この場合、各グリフのコードは、グリフが作成された順序に照応します。***.notdef*** グリフはつねにコード 0 を持ちます。
- Unicode 値のみが指定され、グリフ名が指定されていない場合、PDFlib は、***GXXX*** の形のグリフ名を生成します。ここで ***XXX*** は、生成グリフの 10 進番号です。
- Type 3 フォント内でビットマップを定義するには、***inline*** 画像オプションを用いることを推奨します。***interpolate*** 画像に対するオプションは、Type 3 ビットマップフォントの画面上と印刷上の見映えを向上させるために有用でしょう。
- 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*** オプションを用います。
- 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：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_ext()*** 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、その照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。***.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。
- Type 3 グリフ定義は、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を一切提供しません。ただし、***PDF_load_font()*** でその照応するオプションを用いれば設定できます。

6.2 Unicode のキャラクタとグリフ

6.2.1 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* キーワードでクエリ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、フォント内のその照応する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（138 ページ「グリフ ID エンコーディング」参照）。直接 GID 指定は、グリフ数をクエル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。

6.2.2 グリフに対する Unicode マッピング

Unicode マッピングと多義グリフ　PDFlib は、1 つのフォントの中のすべてのグリフに対して Unicode 値を割り当てます。フォントによっては、ある 1 つのグリフを用いて複数の Unicode 値が表現されていることがあります。こういう多義グリフのよくある例は、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 値が同一のグリフで表されている場合には、PDFlib は、そのグリフを当該 Unicode 値群のうちの 1 つへマップする ToUnicode CMap を生成します。その他の Unicode 値は、同じグリフ ID を用いて出力されますが、その適切な Unicode 値を *ActualText* 属性で割り当てられます。これによって、正しいセマンティクスが、生成された PDF 出力の中でも保たれます。

Acrobat のテキスト選択の不具合　不幸なことに、Acrobat DC はときどき、*ActualText* 属性を持っているテキストのテキスト選択で不具合を起こします。選択・ハイライトされないグリフが生じるのです。見た目上のハイライトは欠けているのに、クリップボードへは、*ActualText* 属性のテキストが正しくコピーされます。この選択不具合を回避するには以下の手があります：

- そもそも多義グリフを避ける。
- テキストフィルタオプション *actualtext=false* を用いることによって *ActualText* 属性の生成を無効化する。

とはいえ、これらの手法は推奨されません。なぜなら、異なるキャラクタが同一 Unicode 値を持って出力されてしまうので、内容の再利用を妨げるからです。

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（107ページ「5.1 Unicodeの重要な諸概念」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* キーワードでクエリ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。ただし、この動作は変更することもでき、それは特に日中韓キャラクタに対して有用です。詳しくは183ページ「外字キャラクタに対するPUA値を温存」を参照してください。

PDFlibがPUA値をマップなしグリフに割り振っていく際には、以下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* テーブル内で見つかった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（スワ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、130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、130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。

1. AGLは *partners.adobe.com/public/developer/en/opentype/glyphlist.txt* にあります。

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）エントリ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（直接的に *uv* 引数を通じて、または間接的に *PDF_begin_glyph_ext()* の *glyphname* オプションを通じて）ということです。ですので、ユーザー定義 Type 3 フォント内のグリフに対しては、適切な Unicode 値か、あるいは適切な AGL グリフ名を与えることを強く推奨します。これによって、正しい Unicode 値が PDFlib によって必ず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。

6.2.3 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、以下のグループの Unicode 制御キャラクタを処理します：

- デフォルトシェーピング動作をオーバライドするための制御キャラクタ群（表 7.4 に挙げる）と、デフォルト双方向組版をオーバライドするための制御キャラクタ群（表 7.5 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 9.1 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 *replacementchar* 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。

6.3 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。

6.3.1 入力文字列を Unicode へ正規化

以下のステップが、*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（120 ページ「5.6.1 エスケープシーケンス」参照）を、その照応する数値へ置き換えます。
- 文字参照を解決し、それをその照応する Unicode 値へ置き換えます（121 ページ「5.6.2 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。
- *normalize* オプションに応じて、テキストを、Unicode 正規化形（NFC 等）のうちの一つへ正規化。

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、129 ページ「6.2.2 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、その照応する 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 121 ページ「5.6.2 文字参照」を参照してください。これらの参照は、その照応する 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チェックす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。

6.3.2 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、以下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 はグリフ ID 0 へ置換されます。*glyphcheck=error* のときは例外が発生します。

フォールバックフォントからのキャラクタを強制　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、その照応するフォールバックフォントのグリフ ID を決定します。詳しくは 149 ページ「6.4.6 フォールバックフォント」を参照してください。

異体字シーケンスを解決　フォントによっては、Unicode キャラクタの後に、そのキャラクタの特定のグリフ異体字を選択する異体字セレクタが続きうるものがあります（186 ページ「7.5.5 Unicode 異体字セレクタと異体字シーケンス」参照）。そのフォントがその異体字シーケンスのための異体字グリフを内容として持っている場合には、元のグリフ ID ではなくその異体字グリフのグリフ ID が用いられます。

グリフ ID へ変換　Unicode 値を、129 ページ「6.2.2 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 値に照応するグリフ 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。すなわち、*.notdef* グリフがテキスト出力内で用いられます。*.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作成せずに終了することを意味し、*PDF_add/create_textflow( )* が −1（PHP の場合：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フォールバックフォントの中で、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フォールバック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。フォールバックフォントのうちの 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?）
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、その照応する 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、その照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*** オプションが以下のように評価されます：

- ***replacementchar=auto***（これがデフォルトです）の場合には、キャラクタ 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、「ミッシンググリフ」記号が使われます（PDF/A・PDF/UA・PDF/X-4/5 では不可）。
- Unicodeキャラクタが***replacementchar***として指定されている場合には、それが元のキャラクタのかわりに用いられます。
- ***replacementchar=drop*** の場合には、そのキャラクタは入力ストリームから除去され、出力は作成されません。
- ***replacementchar=error*** の場合には、例外が発生します。これを利用すると、読めないテキスト出力を避けることができるでしょう。

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

6.3.3 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作成する前に、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・スワッシュキャラクタ・スモールキャップス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成することができます。OpenType 機能については 167 ページ「7.3 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（170 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（176 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。

6.4 フォントを読み込む

6.4.1 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以下を決定します：

- 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 出力を作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チェックします。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（149 ページ「6.4.6 フォールバックフォント」参照）。

非 Unicode 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チェックします。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チェックします。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（149ページ「6.4.6 フォールバックフォント」参照）。

非Unicode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115ページ「5.4 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。これはさまざまなソースから持って来ることができます：

- 115ページ「5.4 シングルバイト（8ビット）エンコーディング」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とさまざまなロケールで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 System i・IBM Z 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)* エントリに基づいて作成されます。
- 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、各グリフに照応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（138ページ「6.4.2 記号フォントに対するエンコーディングを選ぶ」参照）。

非 Unicode 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（118 ページ「5.5 日本語・中国語・韓国語 CMap」参照）。

これらの伝統的エンコーディングは、Unicode CMap を除き、非 Unicode 対応言語バインディングでのみ使うことができます。Unicode CMap は *encoding=unicode* と等価です。

非 Unicode 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チェックし、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チェックしない点です。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 値をそれに照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、129 ページ「6.2.1 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* キーを用いてクエリできます。

非 Unicode 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チェックします。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 によって作成されることもあるので、*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。

6.4.2 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 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 フォントはこれを行えず、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以下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 9.1 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。

6.4.3 例：Wingdings 記号フォント内のグリフを選択

記号フォント内のキャラクタを選択するにはさまざまな方法があり、なかには望みの出力が得られないものもありますので、例を見てみましょう。

フォント内のキャラクタを理解　まず、フォント内の、ターゲットとするキャラクタについていくつかの情報を収集しましょう。これには Windows の**「文字コード表」**アプリケーションを利用します（図 6.1 参照）：

- **「文字コード表」**は、Wingdings フォント内のグリフ群を表示しますが、それは**「詳細表示」**内で Unicode へのアクセスを一切与えません。これは、このフォントが内容として持っている記号グリフ群に対して、標準化された Unicode 値が一切登録されていないという事実による結果です。そのかわりに、このフォント内のグリフ群は私用領域（PUA）内のダミー Unicode 値を使用していますが、**「文字コード表」**アプリケーションはこれらの値を明らかにしません。
- **「文字コード表」**ウィンドウの左下隅を見ると、あるいはマウスを ***smileface*** キャラクタの上に乗せると、**「文字コード：*0x4A*」**と表示されます。これはこのグリフのバイトコードです。
  このコードは、Winansi エンコーディング内の大文字の ***J*** キャラクタに照応しています。たとえば、このキャラクタをクリップボードへコピーして、そのクリップボード内容を、テキストのみのアプリケーションへ貼り付けると、その結果は、その照応する Unicode 値 U+004A すなわちキャラクタ ***J*** となります。にもかかわらず、これはこのキャラクタの Unicode 値ではありませんので、Unicode ワークフロー内で U+004A または ***J*** を用いてこれを選択することはできません。
- このフォント内で内部的に使用されている Unicode キャラクタは**「文字コード表」**には表示されません。しかし、Microsoft が提供している記号フォントは、以下の単純な規則を用いています：

  ```
  Unicode値 = U+F000 + (「文字コード表」に表示される文字コード)
  ```

  このことから、***smileface*** グリフに対しては Unicode 値 U+F04A との結果が得られます。
- その照応するグリフ名は、フォントエディタや同様のツールで取得できます。この例ではそれは ***smileface*** です。

PDF_info_font() を使って、Unicode 値・グリフ名・コードのいずれかをクエリすることもできます。158 ページ「6.6.2 フォント依存のエンコーディング・Unicode・グリフ名クエリ」を参照してください。

記号キャラクタを PDFlib で指定　ターゲットとするキャラクタに関して得られる情報に応じて、Windings の ***smileface*** グリフを選択する方法はいくつかあります：

- そのフォント内のそのキャラクタに割り当てられている PUA Unicode 値がわかっているなら、数値文字参照を使えます（122 ページ「数値文字参照」参照）：

  ```
  &#xF04A;
  ```

  textformat=utf8 で作業している場合には、その照応する 3 バイト UTF-8 列を使えます：

  ```
  \xEF\x81\x8A
  ```

  Unicode 値は、***encoding=builtin*** と ***textformat=bytes*** の組み合わせでは使えません。
- 文字コードがわかっているなら、バイト値参照を使えます（123 ページ「バイト値参照」参照）：

  ```
  &.#x4A;
  ```

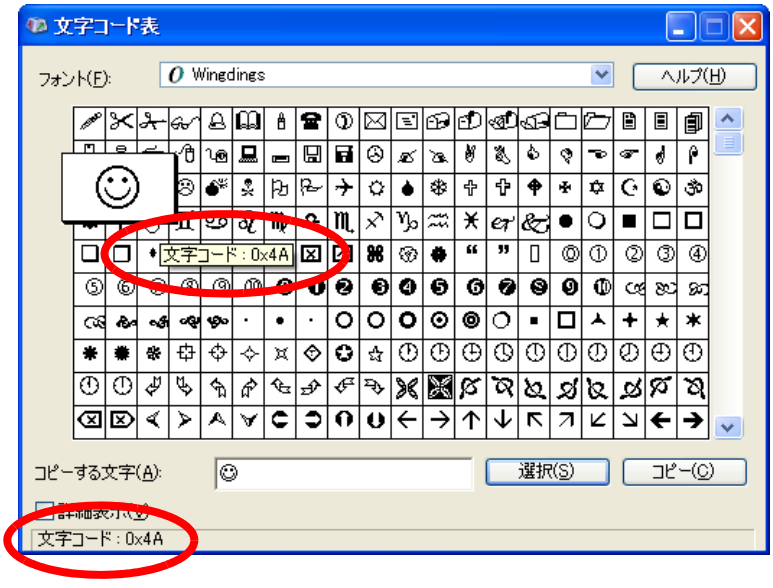


図 6.1
Windows の文字コード表で Wingdings フォント

非 Unicode 対応言語バインディングでは、*encoding=builtin* かつ *textformat=bytes* ならば、文字コードを直接指定できます：

```
J
\x4A
```

- グリフ名がわかっているなら、グリフ名参照を使えます（122 ページ「グリフ名参照」参照）：

```
&.smileface;
```

グリフ名は、*encoding=builtin* と *textformat=bytes* の組み合わせでは使えません。

表 6.1 に、Java・.NET など Unicode 対応言語バインディングの場合の方法を挙げます。

表 6.1 Unicode 対応言語バインディング（Java 等）で Wingdings フォント内の smileface グリフを指定

エンコーディング	追加オプション	入力文字列	ページ上の印字結果
unicode		\uF04A[1]	☺
	charref		☺
	charref	&.#x4A;	☺
	charref	&.smileface;	☺
		J[2]	（利用可能ならスペースグリフ、そうでないなら .notdef グリフ）
	escapesequence	\x4A	（利用可能ならスペースグリフ、そうでないなら .notdef グリフ）
builtin	（encoding=unicode で上記と同じ）		

1. U+F04A に対する、Java など多くの Unicode 対応言語での文字列文法

表 6.2 に、C など非 Unicode 対応言語バインディングの場合の方法を挙げます。

表 6.2 非 Unicode 対応言語バインディング（C 等）で Wingdings フォント内の smileface グリフを指定

エンコーディング	textformat	追加オプション	入力文字列	ページ上の印字結果
unicode	utf16		\xFO\x4A[1]	☺
	utf8	charref	&#xFO4A;	☺
		charref	&.#x4A;	☺
		charref	&.smileface;	☺
			ï•Š[2]	☺
		escapesequence[3]	\xEF\x81\x8A[4]	☺
			J[5]	（利用可能ならスペースグリフ、そうでないなら .notdef グリフ）
		escapesequence	\x4A	（利用可能ならスペースグリフ、そうでないなら .notdef グリフ）
	bytes	charref	&#xFO4A;	☺
		bytes	&.#x4A;	☺
		charref	&.smileface;	☺
			J	（利用可能ならスペースグリフ、そうでないなら .notdef グリフ）
		escapesequence	\x4A	（利用可能ならスペースグリフ、そうでないなら .notdef グリフ）
builtin	utf16, utf8	（encoding=unicode で上記と同じ）		
	bytes	charref	&#xFO4A;	🕮✂☒☞🗀🗐✌▭
		charref	&.#x4A;	🕮👈✂☒🗐✌▭
		charref	&.smileface;	🕮👈♦❍♓●♏➚♋♍♏▭
			J	☺
		escapesequence	\x4A	☺

1. バイト順序に応じて、\xFO\x4A か \x4A\xFO のいずれかで表す必要があります。なお、\x は C のエスケープ文法を示しています
2. 3 バイト列 \xEF \x81 \x8A に対する Winansi キャラクタ群
3. escapesequence オプションは、プログラミング言語が直接のバイト値のための文法を全く提供していない場合にのみ必要です。
4. U+FO4A に対する 3 バイト UTF-8 列
5. バイトコード \x4A に対する Winansi キャラクタ

6.4.4 フォントを検索

フォントデータのソース　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* キーでクエリすることができます。

同じフォント名で *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ソースとして以下に対応しています：

- ディスクベースまたは仮想のフォントファイル
- Windows か macOS 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。**

enumeratefonts オプションを用いると、検索パス上で得られるすべてのフォントを収集するよう PDFlib に命令することができます（59 ページ「ファイル検索と SearchPath リソースカテゴリ」参照）。*saveresources* オプションを用いると、PDFlib リソースのカレントリストをディスクファイルへ書き出すことができます：

```
/* フォントを検索パスに直接追加 */
p.set_option("searchpath={{C:/fonts}}");

/* 検索パス上のすべてのフォントをなめてUPRファイルを作成 */
p.set_option("enumeratefonts saveresources={filename=C:/fonts/pdflib.upr}");
```

フォント名エイリアス設定　フォントはそれぞれ、任意の数のエイリアス名を持つことができます。これは、物理フォントへマップする必要がある擬似または仮想名を通じてフォントを要求する場面で有用でしょう。フォント名エイリアスは、以下の例のように、*FontnameAlias* リソースカテゴリ（58 ページの表 3.1 参照）で作成できます：

```
p.set_option("FontnameAlias={sans Helvetica}");
```

左のエイリアス名は、任意に選ぶことができ、そしてこれを用いて、そのフォントをその新しいエイリアス名のもとに読み込むことができます。右の名前は、フォントの有効な API 名、たとえばホストフォントや、フォントリソースカテゴリ *FontOutline* 等のうちの一つでフォントリソースへ紐付けられているフォントの名前である必要があります。

フォントに対する検索順序　PDFlib に与えられるフォント名は名前文字列です。指定された名前がフォント名エイリアスである場合には、それはその照応する API フォント名へ置き換えられます。PDFlib は API フォント名を使って、さまざまな種類のフォントを後述の順序で探します。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：

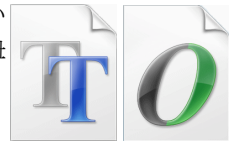
- フォント名が、これ以前に同一文書内で*PDF_begin_font()*で作成されたType 3フォントの名前と一致する（127 ページ「6.1.6 Type 3 フォント」参照）。
- フォント名が、フォント名をTrueTypeまたはOpenTypeフォントファイルの名前と紐づける *FontOutline* リソース内の名前と一致する。
- フォント名が、フォント名をPostScript Type 1フォントのメトリックファイルの名前と紐づける *FontAFM* または *FontPFM* リソース内の名前と一致する。
- フォント名が、フォント名を SVG フォントファイルの名前と紐づける *FontOutline* リソース内の名前と一致し、かつ、*HostFont* リソース内の名前と一致しない。
- 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（145 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（147 ページ「6.4.5 Windows・macOS 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以下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 57 ページ「3.1.4 リソース構成とファイル検索」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・WOFF 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_option("FontOutline={Arial=/usr/fonts/Arial.ttf}");
font = p.load_font("Arial", "unicode", "embedding");
```

等号の左側のフォント名（フォントの API 名といいます）は任意に選べます：

```
p.set_option("FontOutline={f1=/usr/fonts/Arial.ttf}");
font = p.load_font("f1", "unicode", "embedding");
```

実行時に *PDF_set_option()* で構成するのでなく、UPR ファイル内で *FontOutline* リソースを構成することもできます（57 ページ「3.1.4 リソース構成とファイル検索」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 ファイル内で構成することも可能です）。例：

```
p.set_option("SearchPath={{/usr/fonts}}");
p.set_option("FontOutline={f1=Arial.ttf}");
font = p.load_font("f1", "unicode", "");
```

TrueType・OpenType コレクション　TrueType または OpenType コレクション（TTC/OTC、181 ページ「7.5.1 TrueType・OpenType 日中韓フォントを用いる」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_option("FontOutline={MS-Gothic=msgothic.ttc}");
font = p.load_font("MS-Gothic", "unicode", "embedding");
```

フォント名は、TTC/OTC 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/OTC 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 API フォント名は任意に選べます：

```
p.set_option("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_option("FontPFM={lucidux=LuciduxSans.pfm}");
font = p.load_font("lucidux", "unicode", "");
```

PostScript フォントに対して *embedding* が必要な場合には、その名前を、さらにその照応するフォントアウトラインファイル（PFA または PFB）に対しても、*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_option("FontPFM={lucidux=LuciduxSans.pfm}");
p.set_option("FontOutline={lucidux=LuciduxSans.pfa}");
font = p.load_font("lucidux", "unicode", "embedding");
```

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）。コアフォントは以下の名前を持ちます：

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。以下のコード断片は、コアフォントの 1 つを構成なしに要求します：

```
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うちのひとつを埋め込むためには、フォントアウトラインファイルを構成する必要があります。SVG 内のフォント名など場合によっては、フォントアウトラインの構成の中でフォント名にスタイルキーワードを付加すると役立つことがあります：

```
p.set_option("FontOutline={Helvetica,Bold=/usr/fonts/HelvBd.ttf}");
```

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 か macOS 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 147 ページ「6.4.5 Windows・macOS 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（この文法は Latin コアフォントでも使えます）：

```
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。以下のコード断片は、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ます：

```
p.set_option("HostFont={Symbol=Symbol}");
font = p.load_font("Symbol", "unicode", "embedding");
```

等号の左側の API フォント名は任意に選べます。通常は、ホストフォントの名前が等号の両側で用いられます。

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エントリをなめながら、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：

- 以下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 .woff .cef .afm .pfm .ttc .otc, .svg .svgz
  .TTE .TTF .OTF .GAI .WOFF .CEF .AFM .PFM .TTC .OTC, .SVG .SVGZ
  ```

- PostScript フォントの埋め込みが要求されている場合は、以下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以下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構成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、手作業で一切構成をしなくても、もしもその照応するフォントファイル名が、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から成っており、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。

以下のステートメント群のグループ群は、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_option("FontOutline={Arial=/usr/fonts/Arial.ttf}");
font = p.load_font("Arial", "unicode", "");
```

と

```
p.set_option("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 187 ページ「7.5.6 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、もしも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し、かつ、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 つの名前であり、かつ、***embedding*** オプションが指定されなかったならば、フォント検索処理のいちばん最初の段階で発見します。内部リスト内で見つかった標準日中韓フォントは、フォントアウトラインファイルが構成されている場合にのみ埋め込まれます。

なお、標準日中韓フォントの概念は廃止です。PDFlib で使用するための適切なフォントを構成してください。

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（127 ページ「6.1.6 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。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", "");
```

6.4.5 Windows・macOS 上のホストフォント

macOS・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構成しなくても、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 126 ページ「6.1.4 PostScript Type 1 フォント」にあります。ホストフォント検索は、*PDF_set_option()* の *usehostfonts* オプションで無効にすることもできます。

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ウィンドウタイトルに表示される完全フォント名を見れば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。

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", "");
```

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せん）。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構成されたフォントに対しては働きません。

以下のキーワード（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)

以下の例はフォントの 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。

Windows のフォント代替 Windows は、特定のレジストリエントリ群に基づいて、自動的にフォントを代替することがあります。この種のフォント代替は、PDFlib のホストフォント機構にも影響を与えますが、Windows オペレーティングシステムの完全制御下にあります。以下のレジストリエントリによって、たとえば、Helvetica フォントが必要なときに Windows がかわりに Arial を届けることもありえます：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\FontSubstitutes
```

Windows のフォント代替に関する詳しい情報については、Microsoft の文書を参照してください。

macOS 上のホストフォント名 macOS 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 Font Tool Suite[1] を推奨します。このコマンドラインユーティリティの集合には ***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：

- 「一意」フォント名：これは、東アジアフォント等に対して Unicode でエンコードされている場合もあるフォント名です。一意フォント名を知るには、ターミナルウィンドウで以下のコマンドを実行します（「:」を含むエントリ群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
  ```

macOS 上でホストフォントを用い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。

6.4.6 フォールバックフォント

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。

1. *developer.apple.com/fonts* を参照。

フォールバックフォントは、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。複数の用字系のためのフォント群を合成するのにも役立ちます。たとえばアラビア文字用字系のためのフォントは、欧文キャラクタに対応していないことがありえます（Google Noto フォントファミリなどはこのように作られています）。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、フォールバックフォント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。フォールバックフォントは以下の機能を提供します：

- ベースフォント内で得られないグリフは自動的に、1 つないし複数のフォールバック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。複数のフォールバック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。
- ある特定のフォールバック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オーバライド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。

フォールバックフォントからのグリフの寸法と縦位置は、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フォールバック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）。これを利用すると以下のトリックが実装できます：

- ベースフォントそれ自身をフォールバック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寸法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。

フォールバックフォント機構は、*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フォールバック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、フォールバックフォント機構の結果をクエリすることができます（159 ページ「6.6.3 コードページ網羅性とフォールバックフォントをクエリ」参照）。

注意　フォールバックフォントを扱う際には以下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フォールバックフォントだけを使うよう注意を払う必要があります。
- フォールバック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、フォールバックフォントは自動的には埋め込まれません。

- フォールバックフォントは、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 つないし複数のフォールバック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、フォールバック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、フォールバック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フォールバック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、フォールバックフォントの重要な用途をいくつか解説し、その照応するオプションリストを演示します。

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以下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以下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。フォールバック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フォールバックフォントとします）、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キャラクタを含んでいません）で読み込んだとすると、*fallbackfonts* オプションに対して以下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（121 ページ「5.6.2 文字参照」参照）。

フォント内の一部ないし全部のグリフを大きくする　フォールバックフォントを使うと、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フォールバック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以下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以下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以下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 180 ページ「7.4.5 アラビア文字テキスト組版」で解説しています。

足りないグリフをつきとめる　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 キャラクタの 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以下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視覚化されるようこの特殊なフォールバックフォントで強化することができます：

```
fallbackfonts={{fontname={Unicode BMP Fallback SIL} encoding=unicode}}
```

フォントに外字キャラクタを追加　この用途は 182 ページ「7.5.3 EUDC・SING フォントによる外字キャラクタ」で解説しています。

6.5 フォントの埋め込みとサブセット化

6.5.1 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、かつ構成されていればそれを取り（**「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。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。

フォントを PDFlib で埋め込む　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 6.3 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 6.3 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、その照応する 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 6.3 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 かつ skipembedding={latincore} が設定されていない場合のみ
macOS か Windows にインストールされている TrueType・OpenType・Type 1 ホストフォント	×	×
非コア Type 1 フォント	○	embedding=true の場合のみ
TrueType・OpenType・SING・WOFF・SVG フォント	n/a	○

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書体工房もあります。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チェックしてください。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。

上記の法的警告は、Web フォントについては特に留意しておく必要があります。なぜなら、Web 上で利用するためのフォントの多くのベンダーは、そのようなフォントを PDF 文書内に埋め込むことを許していないからです。

6.5.2 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。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、以下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
- WOFF フォント
- Type 3 フォント（特別な扱いが必要、155 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*** オプションで制御されます。もしこのオプションが ***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*** オプション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。以下のフォントオプションはサブセット限界を 75% に設定します：

```
subsetlimit=75%
```

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。

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* オプションを用いると、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。

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後で 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、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_ext()* 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_ext()* 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。
  使われていないグリフ群の定義のための API 関数呼び出し群は警告なく無視されます。エラーコードを返せる関数（*PDF_load_image()* 等）はエラー値を返しますが、それをアプリケーション側は無視する必要があります。使われていないグリフを本当のエラーから区別するためには、*PDF_get_errnum()* によって返されるエラーメッセージ番号をチェックする必要があります : もしゼロであれば、そのグリフは無視されている、すなわち本当のエラーは起こっていないということです。

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** `type3_fonts/type3_subsetting` **トピックにあります。**

6.6 フォント情報をクエリ

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クエリすることができます。クエリの種類によっては、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 6.4 に挙げる変数を用いることにします。

表 6.4 PDF_info_font( ) の利用例で用いる変数

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 以外のとき、その照応する文字列を取得するには以下のようにします： gn = p.get_string(gn_idx, "");
String enc;	エンコーディング名
int font;	PDF_load_font( ) を用いて作成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。

6.6.1 フォント非依存のエンコーディング・Unicode・グリフ名クエリ

エンコーディングクエリ　エンコーディングクエリには、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 の場合：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 キャラクタか指名したグリフの、8 ビットエンコーディング内の 8 ビットコードをクエリ：

```
c = (int) p.info_font(-1, "code", "unicode=" + uv + " encoding=" + enc);
c = (int) p.info_font(-1, "code", "glyphname=" + gn + " encoding=" + enc);
```

8 ビットコードか指名したグリフの、8 ビットエンコーディング内の Unicode 値をクエリ：

```
uv = (int) p.info_font(-1, "unicode", "code=" + c + " encoding=" + enc);
uv = (int) p.info_font(-1, "unicode", "glyphname=" + gn + " encoding=" + enc);
```

8 ビットコードか 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クエリ：

```
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_string(gn_idx, "");
```

Unicode・グリフ名クエリ　*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クエリを行うことも可能です。これらのクエリ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 値をクエリ：

```
uv = (int) p.info_font(-1, "unicode", "glyphname=" + gn + " encoding=unicode");
```

Unicode 値の内部グリフ名をクエリ：

```
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_string(gn_idx, "");
```

6.6.2 フォント依存のエンコーディング・Unicode・グリフ名クエリ

以下のクエリは、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 ビットコードをクエリ：

```
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 値をクエリ：

```
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 をクエリ：

```
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 をクエリ：

```
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クエリ：

```
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_string(gn_idx, "");
```

グリフの入手可能性をチェック　*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チェックすることができます。たとえば、以下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チェックします：

```
/* "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チェックすることもできます。以下のコード断片は、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クエリします。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 */
}
```

6.6.3 コードページ網羅性とフォールバックフォントをクエリ

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作成するのにフォントが適しているかどうかをチェックすることもできます。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、158 ページ「6.6.2 フォント依存のエンコーディング・Unicode・グリフ名クエリ」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クエリ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チェック　以下のコード断片は、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チェックします：

```
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　以下のコード断片は、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クエリします：

```
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_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以下のリストを作成します：

```
cp1252 cp1250 cp1251 cp1253 cp1254 cp1255 cp1256 cp1257 cp1258 cp874 cp932 cp936 cp949
cp950 cp1361
```

予備グリフをクエリ　*PDF_info_font()* を利用すると、フォールバックフォント機構の結果をクエリすることができます（フォールバックフォントについて詳しくは 149 ページ「6.4.6 フォールバックフォント」を参照）。以下のコード断片は、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フォールバックフォントの名前を調べます：

```
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フォールバック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フォールバックフォントでも表示できない */
}
else
{
        idx = p.info_font(result, "fontname", "api");
        fontname = p.get_string(idx, "");
}
```

7 テキスト出力

7.1 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）。以下のコード断片は、低レベル関数群でテキスト出力を作成します：

```
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作成し、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。以下のコード断片は、テキスト行でテキスト出力を作成します。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");
```

詳しくは 225 ページ「9.1 テキスト行を配置・はめ込む」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作成し、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。以下のコード断片は、テキストフローを用いてテキスト出力を作成します：

```
tf = p.add_textflow(tf, text, optlist);
result = p.fit_textflow(tf, llx, lly, urx, ury, optlist);
p.delete_textflow(tf);
```

詳しくは 235 ページ「9.2 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。詳しくは 256 ページ「9.3 表の組版」を参照してください。

7.2 フォントメトリックとテキストバリエーション

7.2.1 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* オプションでクエリでき、後者は *currentx/currenty* でクエリできます。

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。

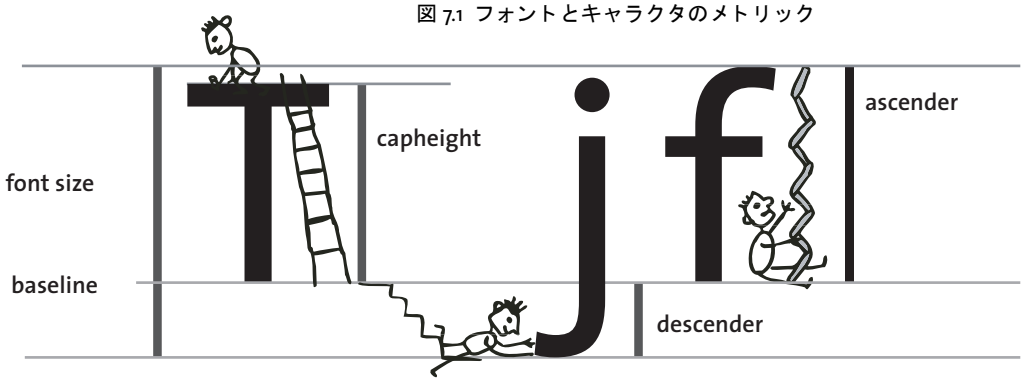
PDFlib のユーザーが指定する必要のある文字サイズというのは、テキストの行と行の間で文字が重なりあわないために必要な最小間隔のことです。文字サイズは一般にフォント内の各文字よりも大きくなっています。なぜならその中にはベースラインより上の部分も下の部分も含んでいるからであり、また、それに加えて行と行の間の間隔をもっと広くとっていることもあるからです。

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* をクエリします。あるフォントのキャラクタメトリックを PDFlib でクエリするには以下のように記述します：

図 7.1 フォントとキャラクタのメトリック



```
font = p.load_font("Times-Roman", "unicode", "");

capheight = p.info_font(font, "capheight", "");
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を、PDFlib を用いてクエリ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/font_metrics_info` トピックにあります。

7.2.2 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 には、カーニングの動作を制御する方式が 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ペア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。

図 7.2 カーニング

カーニングは、どのような字間・単語間隔・横伸縮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ペアの数についてはまったく無制限です。

カーニングは、フォントオプション *readkerning* とテキストオプション *kerning* によって制御されます。デフォルトではカーニングは有効になっています。

7.2.3 テキストバリエーション

擬似ボールドフォント PDFlib は、個別のテキスト文字列について、*fakebold* オプションを通じて擬似的なボールドテキストを作成する機構をサポートしています。この方式は、グリフの輪郭を描線することによってボールドフォントを擬似表現しています：Type 3 フォントの場合には、テキストがさまざまな変位で複数回配置されます。強調のためには本当のボールドフォントを使用することを強く推奨します。*fakebold* オプションは、本当のボールドテキストよりも劣るテキスト出力を作成しますし、また、テキスト抽出を阻害するおそれもあります。

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

注 `fontstyle=bold[italic]` **フォントオプションを使うと、ある特定のフォントで作成されるすべてのテキストについて fakebold 擬似表現を強制することができます。**

擬似斜体フォント *italicangle* 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成するために、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す。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* 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* オプションの値は普通 −12 から −15 度の範囲です。

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作成することができます。影付きテキストは *PDF_fit_textline()*・*PDF_add/create_textflow()* の *shadow* オプションで作成できます。影の色や、その主テキストに対する相対位置、図形ステータスオプション群は、サブオプションで指定できます。

下線・上線・取り消し線付きテキスト 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、横伸縮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_text_option()* で *underline*・*overline*・*strikeout* オプションを、またはテキスト出力関数群でその照応するオプションをオンオフします。*underlineposition*・*underlinewidth* オプションを使って微調整もできます。

線の色には *strokecolor* オプションが用いられます。しかし、カレントの linecap オプションは無視されます。*decorationabove* オプションは、線がテキストの上と下のいずれ

に引かれるかを制御します。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*** 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、***strokewidth*** ・***strokecolor*** といったテキストステータスのオプション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、***fillcolor*** オプションで指定された色で印字されます。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、***strokecolor*** と ***fillcolor*** はどちらもテキストに対して効力を持ちます。

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。**

7.3 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・text_output/opentype_feature_tester トピックにあります。

7.3.1 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ップス、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 は以下のグループの OpenType 機能に対応しています：

- 表 7.1 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 7.7 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 183 ページ「7.5.4 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（174 ページ「7.4 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（164 ページ「7.2.2 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以下にあります：

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以下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 機能を表示します。
- DTL OTMaster Light（*www.fontmaster.nl*）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。
- PDFlib の *PDF_info_font()* インタフェースを使って、対応している OpenType 機能をクエリすることもできます（172 ページ「OpenType 機能をプログラム的にクエリ」を参照）。

表 7.1 欧文タイポグラフィ用の対応 OpenType 機能（日中韓テキスト用の OpenType 機能は表 7.7 に挙げます）

キーワード	名前	説明
_none	全機能無効	表 7.1・表 7.7 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ップス	大文字キャラクタをスモールキャップスに変えます。
calt	コンテキスト字体	デフォルトグリフを、より良い連結動作を与える字体へ置き換え。一部ないし全部のグリフを連結させるようデザインされた用字系書体で使用されます。
case	ケースセンシティブ字体	さまざまなアクセント記号を上へずらして、全大文字シーケンスや横並び数字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。
ccmp	グリフ合成 / 分解	グリフ字体の数を最小化するために、1 個のキャラクタを 2 個のグリフへ分解したり、あるいはより良いグリフ処理のために 2 個のキャラクタを 1 個のグリフに合成したりすることが望ましい場合があります。この機能はそのような合成 / 分解を許します。
clig	コンテキスト合字	グリフ列を、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。他の合字機能群と異なり、clig は、その合字が推奨されるコンテキストを指定します。この能力は、いくつかの用字系デザインにおいて、またスワッシュ合字に対して重要です。
cswh	コンテキストスワッシュ	デフォルトグリフを、指定されたコンテキストにおいて、照応するスワッシュへ置き換え。
dlig	随意合字	グリフ列を、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。dlig は、ユーザーの好みによって、特殊効果のために用いることができる合字をカバーします。
dnom	分母	スラッシュの直後の数字を分母数字へ置き換え
falt	改行箇所語尾グリフ	行末グリフを、この目的のために特にデザインされた字体（必要に応じて前幅が短かったり長かったりします）へ置き換えることによって、テキストの均等割り付けを助けます。
fina	語尾形	単語の末尾にあるグリフを、この使用目的のためにデザインされた字体へ置き換え。これは、欧文の続け字において広く用いられており、また、アラビア文字等さまざまな非欧文用字系において必須です。
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	デフォルトの（現行の）合字を歴史的合字へ置き換え。
init	語頭形	単語の先頭にあるグリフを、この使用目的のためにデザインされた字体へ置き換え。これは、欧文の続け字において広く用いられており、また、アラビア文字等さまざまな非欧文用字系において必須です。
isol	単独形	グリフの基準形を、その単独形へ置き換え。
liga	標準合字	グリフ列を、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。liga は、デザイナー / 製造者が通常の条件において用いられるべきと判定した合字をカバーします。

表 7.1 欧文タイポグラフィ用の対応 OpenType 機能（日中韓テキスト用の OpenType 機能は表 7.7 に挙げます）

キーワード	名前	説明
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。
medi	語中形	単語の途中にあるグリフを、この使用目的のためにデザインされた字体へ置き換え。これは、単独で用いられるためにデザインされているデフォルト形とは異なります。これは、欧文の続け字において広く用いられており、また、アラビア文字等さまざまな非欧文用字系において必須です。
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、その照応する、数学的表記で広く用いられている字体へ置き換え。
numr	分子	スラ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、その照応する、数字の後に用いるための序数字体へ置き換え。通常、Numero（U+2116）キャラクタも作成します。
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、スワ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。
smcp	スモールキャップス	小文字キャラクタをスモールキャップスに変えます。
ss01 ... ss20	スタイリスティック集合 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、その文字集合の 1 個ないし複数の一部分に照応するスタイリスティック異体字グリフ群の集合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異体字。
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	スワッシュ	デフォルトグリフを、照応するスワ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、照応する、見出し用にデザインされた字体へ置き換え。
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ップス字体の混在する集合へマップして、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。

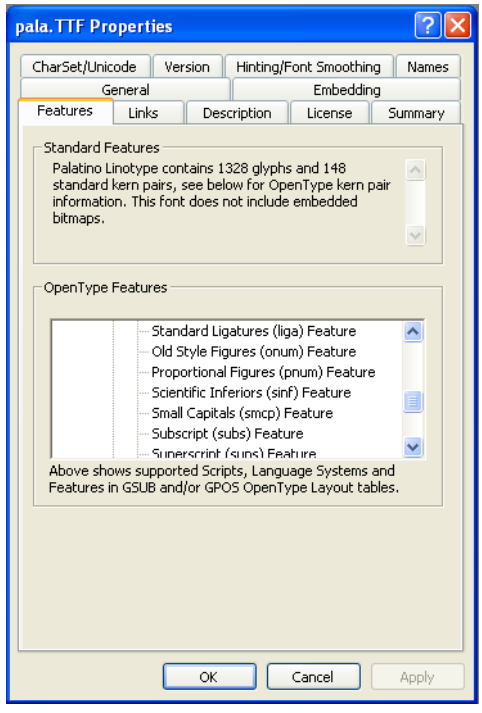


図 7.3
Microsoft の font property extension はフォント内の OpenType 機能一覧を表示します

7.3.2 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、*PDF_show()* 等の単純テキスト出力関数群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、以下の要件を満たす必要があります：

- フォントは、TrueType（**.ttf*）・OpenType（**.otf*）・TrueType/OpenType 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。
- フォントファイルは、使いたい OpenType 機能を持つ GSUB テーブルを含んでいる必要があります。
- 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。

PDFlib は、GSUB テーブルルックアップを持つ 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 機能には対応していません。

注意　OpenType 機能を扱う際には以下に留意してください：

- 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、フォールバックフォントを指定している場合は、ベースフォントと 1 個ないし複数のフォールバック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　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以下のようになります：

```
<features={liga}>ffi<features={noliga}
```

テキスト行に対しては OpenType 機能を以下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、かつその照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*）とスモールキャップス機能（*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（そのフォントが、照応する機能エントリ群を含んでいることを前提として）、スモールキャップス機能が適用され、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ップスを適用します。

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 7.4 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、以下のコードは *f+i* 合字を作成しません：

```
<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、以下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作成します：

```
<features={liga}>fi
```

しかし、以下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作成しません：

```
<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 176 ページ「7.4.2 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（174 ページ「7.4 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クエリ　フォント内の OpenType 機能を *PDF_info_font()* でクエリすることができます。以下のステートメントは、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_string(result, "");
}
else
{
        /* 対応機能は見つからなかった */
}
```

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チェックするには以下のステートメントを用います。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 */
}
```

7.4 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complex_scripts/starter_shapingトピックにあります。

7.4.1 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 7.2 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。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。

シェーピング　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。以下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以下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。フォールバック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 7.2 script オプションに対する複雑用字系とキーワード

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。全一覧は以下にあります： 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、シェーピングされたテキストの中での書式の変更は、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、以下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。
- 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 *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。

7.4.2 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 7.2 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クエリ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 7.2 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（171 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（251 ページ「9.2.10 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。

表 7.3 language オプションに対するキーワード

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。

7.4.3 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（例：U+FB50 から始まるアラビア文字表示形群 A）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 176 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オーバライド　場合によっては、ユーザーがデフォルトシェーピング動作をオーバライド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 7.4 参照）。

表 7.4　デフォルトシェーピング動作をオーバライドするための Unicode 制御キャラクタ

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

7.4.4 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ネストされ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、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。

1. *www.unicode.org/unicode/reports/trg/* を参照。

双方向アルゴリズムをオーバライド　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 7.5 参照）。これらの双方向組版コードは、以下のような場合にデフォルトの双方向アルゴリズムをオーバライドするのに有用です：

- 右書きの段落が左書きのキャラクタ群で始まる場合。
- 混在テキスト列がネストされている場合。
- 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 7.5 双方向アルゴリズムをオーバライドするための方向組版コード

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 の前の双方向ステータスへ復帰

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、以下のオプションを用います：

- テキスト行を、以下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作成します：

```
leader={alignment=left text=.}
```

- *PDF_begin/end_document()* の以下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以下も有用でしょう：

- *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 7.5 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。たとえば以下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}
```

7.4.5 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。

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 7.4 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、以下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　Google の Noto フォント群など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。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以下のフォント読み込みオプションリストを用いると、NotoNaskhArabic-Regular フォントへ NotoSerif-Regular フォントの欧文キャラクタ群を追加することができます：

```
fontname=NotoNaskhArabic-Regular encoding=unicode
fallbackfonts={ {fontname=NotoSerif-Regular encoding=unicode} }
```

7.5 日本語・中国語・韓国語テキスト出力

7.5.1 TrueType・OpenType 日中韓フォントを用いる

PDFlib は、TrueType・TrueType/OpenType Collections（TTC/OTC）・OpenType 形式の日中韓フォントに対応しています。日中韓フォントは以下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、BOM 付き UTF-8 形式か UTF-16 形式で ***PDF_load_font()*** に与えることができます。macOS では非欧文ホストフォント名に対応していません。

以下の例では、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 143 ページ「6.4.4 フォントを検索」に従って構成されている必要があります：

```
font = p.load_font("Arial Unicode MS", "unicode", "");
if (font == -1) { ... }
p.fit_textline("\u4e00\u500b\u4eba", x, y, "fontsize=24");
```

TrueType Collection 内の個別フォントにアクセス　TTC/OTC 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/OTC 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/OTC ファイル ***msgothic.ttc*** は複数のフォントを持っており、***PDF_load_font()*** で以下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。フォント名のエイリアス（143 ページ「フォントデータのソース」参照）を TTC/OTC 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/OTC フォントファイルは 1 回のみ構成しなければなりません。TTC/OTC 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構成するために充分なコードを示します（143 ページ「6.4.4 フォントを検索」参照）：

```
p.set_parameter("FontOutline", "msgothic=msgothic.ttc");
```

7.5.2 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：

- CMap 以外のエンコーディングを持つ TrueType・OpenType 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。

- 日中韓 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。

デフォルトでは縦書きではすべてのグリフが同じ高さを持ちます。ただし TrueType・OpenType フォントは、縦書きについてプロポーショナルなメトリックを内容とすることもできます。PDFlib では、フォントオプション ***readverticalmetrics***（デフォルトは ***false***）を設定することによって、そのようなプロポーショナルな縦書きメトリックを使用させることも可能です。

注 縦書きで文字の間をあけるには字間を負にする必要があります。

OpenType フォントは、表 7.6 に挙げる OpenType レイアウト機能を持っている場合があります。これらの機能は、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るものです。うち ***vkna*** 機能については、***features*** テキストオプションを用いて制御することが可能であり、また ***vrt2/vert*** は、縦書きでは自動的に有効になります。

表 7.6 縦書きテキストのための OpenType レイアウト機能

キーワード	名称	説明
vert[1]	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	縦組みかな切替	標準のかなを、縦書きのために特にデザインされた字形へ置き換え。
vrt2[1]	縦組み切替・回転	いくつかの等幅のグリフ（半角・三分の一幅・四分の一幅）とプロポーショナル幅のグリフ（主に欧文とカタカナ）を、縦書きに適した字形へ、すなわち 90°時計回りさせた形へ置き換え。この機能が存在しているときには、vrt2 の部分集合である vert 機能は無効化されます。

1. 縦書きにおけるフォントに対しては自動的に有効化されます

7.5.3 EUDC・SING フォントによる外字キャラクタ

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フォールバック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。

外字キャラクタに対してフォールバックフォントを用いる 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以下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フォールバックフォント構成で読み込めば、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}
}
```

外字キャラクタに対する PUA 値を温存　場合によっては、たとえば Windows EUDC フォント（後述）では、そのフォントによって外字キャラクタが私用領域（PUA）内の Unicode 値へマップされています。デフォルトでは、PDFlib は PUA 値を ToUnicode CMap 内の U+FFFD（Unicode 置換キャラクタ）へ置き換えます。結果として、そのようなキャラクタは、生成される PDF から正しく抽出できません。

この動作は、*preservepua* フォント読み込みオプションを用いて変えることもできます。あるフォントに対してこれが *true* に設定されている場合には、PUA 値を持つ外字キャラクタはその Unicode 値を温存しますので、その外字は、生成される PDF から正しく抽出できます。

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コピーします。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、143 ページ「6.4.4 フォントを検索」で示した方式のいずれかで構成します：

  ```
  p.set_option("FontOutline={EUDC=EUDC.TTE}");
  p.set_option("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構成をするのではなく、以下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構成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_option("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、以下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。

7.5.4 OpenType レイアウト機能と高度な日中韓テキスト出力

167 ページ「7.3 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 7.7 に、日中韓テキストのための OpenType 機能を挙げます（他に、一般的用途のための OpenType レイアウト機能を表 7.1 に、縦書きのための機能を表 7.6 に挙げています）。

表 7.7 日本語・中国語・韓国語テキスト用の対応 OpenType レイアウト機能

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、照応する、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	（ISO 14496-22:2015/Amd.2:2017 により廃止）韓文漢字キャラクタを、その照応するハングル（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、その照応するイタリック体のグリフへ置き換え。
jp04	JIS2004 字形	（nlck 機能の部分集合）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、その照応する 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、その照応する 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、その照応する 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	四分の一幅	他の幅のグリフを、em の 4 分の 1 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、その照応する簡体字へ置き換え。

表 7.7 日本語・中国語・韓国語テキスト用の対応 OpenType レイアウト機能

キーワード	名前	説明
tnam	名前旧字体	和文新字体を、その照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、その照応する旧字体 / 繁体字へ置き換え。
twid	三分の一幅	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。

7.5.5 Unicode 異体字セレクタと異体字シーケンス

Unicode キャラクタは、さまざまなグリフで表現されることができます。通常、そうした視覚的差異は、適切なフォント（レギュラーかイタリックか等）を用いて実現されます。場面によっては、グリフの選択がセマンティックに意味を持ち、フォント関連の組版情報一切なしでプレーンテキスト内でもそれを明確にする必要があります。Unicode は、この目的のための異体字セレクタという機構を提供しています。

異体字シーケンス　異体字シーケンスは、ベース Unicode キャラクタ 1 個と、後続する異体字セレクタ 1 個とから成ります。このシーケンスを、そのベースキャラクタの**異体字**といいます。Unicode 規格は 2 種類のシーケンスから成っています：

- 標準化異体字シーケンス：Unicode　　キャラクタデータベース内のファイル *StandardizedVariants.txt*[1] 内で定義されています。範囲 *U+FE00 ～ U+FE0F* 内の 16 種の異体字セレクタのうちの一つを使用します。標準化異体字シーケンスは、数学グリフの字体と、絵文字の異体字、モンゴル文字テキストを選択するために用いられます。
- 表意文字異体字シーケンス（IVS）：Unicode Technical Standard #37「Unicode Ideographic Variation Database」に従った登録プロセスによって定義されており、表意文字異体字データベース[2] 内にリストされています。IVS は、ベースキャラクタとしての統一表意文字キャラクタ 1 個と、範囲 *U+E0100 ～ U+E01EF* 内の 240 種の異体字セレクタのうちの一つから成ります。

そのフォントが、必要なグリフを含んでいないなどの原因で、ベースキャラクタに対する異体字セレクタに従えない場合、それは無視されます。

PDFlib で異体字グリフを作成　Unicode 異体字シーケンス（UVS）に対する適切なグリフは、フォントによって提供される必要があります。現状では OpenType が唯一、UVS を格納可能なフォント形式です。（フォーマット 14 cmap テーブルを用いて）。PDFlib は、OpenType フォント内の UVS テーブルを解釈します。ただし、これが *readselectors* フォントオプションで無効にされている場合は除きます。フォントは内容文字列に対してのみ利用可能であり、ハイパーテキスト・名前文字列に対してはそうではありませんので、これらの文字列種別に対しては異体字セレクタは無視されます。

フォントが、必要なグリフを含んでいることがわかっている場合には、異体字シーケンスに対する作業は、単に PDFlib のテキスト出力関数にそのシーケンスを与えればすみます。以下のコード断片は、Unicode ベースキャラクタのデフォルトグリフと、セレクタによって選択された異体字を印字します：

```
p.fit_textline("&#x2268; &#x2268;&#xFE00;", 50, 700,
        "fontname={Cambria Math} encoding=unicode fontsize=24 charref=true");

p.fit_textline("&#x3402;&#xE0100; &#x3402;&#xE0101;", 50, 650,
        "fontname={KozMinPr6N-Regular} encoding=unicode fontsize=24 charref=true");
```

得られる出力を図 7.4 に示します：各グリフペアの中の違いに留意してください。

注 異体字シーケンスは、`fallbackfonts` オプションの `forcechars` サブオプションでは対応していません。

フォント内の異体字セレクタをクエリ　*PDF_info_font()* を使って、フォントがそもそも異体字セレクタを含んでいるのかどうかをチェックすることもできます。*selector* キー

1. *www.unicode.org/Public/UNIDATA/StandardizedVariants.html* 参照
2. *www.unicode.org/ivd* 参照

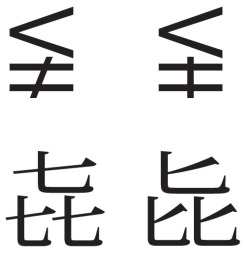


図 7.4 デフォルトグリフと異体字グリフ

ワードを *index* オプションとともに用いれば、そのフォント内で得られるすべての異体字セレクタの一覧を取得することができます：

```
for (i = 0; i < 256; i++)
{
        selectors[i] = (int) p.info_font(font, "selector", "index=" + i);
        if (selectors[i] == -1)
        {
            selectorcount = i;
            break;
        }
}
```

以下のコード断片を使うと、そのフォントが、特定のシーケンスに対する異体字グリフを含んでいるかどうかをチェックできます：

```
if (p.info_font(font, "unicode", "unicode=" + uv + "selector=" + s) == -1)
{
        /* このシーケンスに対してそのフォント内で得られる異体字グリフはない */
}
```

このクエリは、異体字が得られるかどうかをチェックすることだけを意図しています。得られる Unicode 自体は、PDFlib は異体字に PUA Unicode 値を割り当てますので、おそらく有用ではないでしょう。

7.5.6 標準日中韓フォント

注 標準日中韓フォントという概念は廃止です。外部的に構成されたフォントを、埋め込みありまたはなしで、かわりに使用してください。

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提供され、PDF ファイルに埋め込む必要がありません。標準日中韓フォントは、縦書きと横書きに対応しています。標準日中韓フォントの名前の一覧を、適用可能な CMap とともに表 7.8 に示します（日中韓 CMap についての詳細は 118 ページ「5.5 日本語・中国語・韓国語 CMap」参照）。

表 7.8 日本語・中国語・韓国語テキスト用の Acrobat の標準フォント・CMap（エンコーディング）

ロケール	フォント名	対応 CMap（エンコーディング）
日本語	KozMinPro-Regular-Acro KozGoPro-Medium[1] KozMinProVI-Regular[1]	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-UTF16-H[2], UniJIS-UTF16-V[2]

表 7.8 日本語・中国語・韓国語テキスト用の Acrobat の標準フォント・CMap（エンコーディング）

ロケール	フォント名	対応 CMap（エンコーディング）
中国語 簡体字	AdobeSongStd-Light[1]	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-UTF16-H[2], UniGB-UTF16-V[2]
中国語 繁体字	AdobeMingStd-Light[1]	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-UTF16-H[2], UniCNS-UTF16-V[2]
韓国語	AdobeMyungjoStd-Medium[1]	KSC-EUC-H, KSC-EUC-V, KSCms-UHC-H, KSCms-UHC-V, KSCms-UHC-HW-H, KSCms-UHC-HW-V, KSCpc-EUC-H, UniKS-UTF16-H[2], UniKS-UTF16-V[2]

1. PDF 1.6 以上を生成するときのみ利用可能
2. 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、レガシコード列は 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 機能とテキストフロー組版機能が利用できることです。

8 画像・SVG グラフィック・PDF ページを取り込む

8.1 ラスタ画像

8.1.1 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、その画像を PDFlib 関数で開く必要があります。この関数は、画像特性を解析してピクセルデータを PDF 出力へコピーします。*PDF_load_image()* 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、その際に位置付け・拡縮オプションも与えます：

```
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、画像の位置付け・拡縮・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 218 ページ「8.4 画像・グラフィック・取り込み PDF ページを配置」で詳しく説明します。

画像ファイルの取り込みが失敗した場合には、*PDF_load_image()* はエラーコードを返します。こ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、*PDF_get_errmsg()* を呼び出すと、詳細なエラーメッセージが得られます。

クックブック **画像処理のためのコードサンプルが PDFlib クックブックの images カテゴリにあります。**

画像データを再利用　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拡縮倍率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。拡縮倍率が 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か DeviceN 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。

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、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あり、インライン画像を他の場面で使用することは推奨しません。

インライン画像は *PDF_load_image()* と *inline* オプションで生成できます。インライン画像の再利用はできません。すなわちその照応する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()* の内部動作は以下のコードと等価になります：

```
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。

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。デフォルトでは PDFlib は、出力ファイルサイズを削減するため、TIFF・JPEG・JPEG 2000 画像形式内の画像メタデータを無視します。ただし、その XMP メタデータは、*PDF_load_image()* の以下のオプションで、出力 PDF 文書内の生成画像に添付することができます：

```
metadata={keepxmp=true}
```

8.1.2 対応画像ファイル形式

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（195 ページ「8.1.4 画像透過」参照）。

PNG 画像が sRGB チャンクを含んでいる場合には、その sRGB ICC プロファイルが画像に添付されます。ただし、*honoriccprofile* オプションが *false* である場合、または、*iccprofile* オプションで他の ICC プロファイルがその画像に割り当てられている場合を除きます。この sRGB チャンク内のレンダリングインテントが使用されます。ただし *renderingintent* オプションが与えられているときを除きます。

JPEG 画像　PDFlib は、以下の種類の JPEG 画像（ISO 10918-1）に対応しています：

- グレースケール・RGB（通常は YCbCr 符号化されていますが、直接 RGB にも対応）・CMYK カラー。
- ベースラインおよびプログレッシブ JPEG 圧縮。

これらの条件は、実用されているすべての JPEG 画像を網羅します。上記の制約は、広く用いられていない JPEG の機能を使用している画像を取り込むことはできないことを意味します。とりわけ算術符号、ロスレス圧縮、構成要素あたり 8 ビット以外のビット深度、および SmartScale・可逆 RGB 色変換など非標準的機能がこれにあたります。なお、類似形式 JPEG-LS（ISO 14495、「ロスレス JPEG」と呼ばれる時もある）・JPEG-XR（ISO 29199-2、以前 HD Photo と呼ばれていた）には対応していません。

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 ファイル形式と機能に PDFlib は対応しています：

- JFIF 1。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。
- 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。
- PDFlib は、*honoriccprofile* オプションが *false* に設定されていない限り、JPEG 画像内の埋め込み ICC プロファイルに従います。
- JPEG 画像が Exif マーカを含んでいる場合には、その Exif マーカ内の色空間情報は解釈されます。それが sRGB 色空間を示している場合には、その sRGB ICC プロファイルがその画像に添付されます（ただし、その画像が明示的に埋め込まれた ICC プロファイルを含んでいる場合、または *honoriccprofile* オプションが *false* である場合、またはその画像に *iccprofile* オプションで他の ICC プロファイルが割り当てられている場合は、この限りではありません）。
- 画像の望ましい向きを指定した、Exif マーカ内の Orientation エントリに従います。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。

JPEG 2000 画像　JPEG 2000 画像（ISO 15444-2）には PDF 1.5 以上が必要です。PDFlib は、JPEG 2000 画像を、以下の条件に従って受け付けます：

- 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。ただし以下の色空間条件に従う必要があります。範囲 1 ～ 38 ビットのすべての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 ベース色空間群・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。

- （非サポート）1 個・3 個・4 個の色要素を持つ、JPX ラッパのない生の JPEG 2000 コードストリーム（しばしば **.j2k*）を取り込むことができます。
- JPEG 2000 画像に埋め込まれた限定またはフルの ICC プロファイルは保持されます。すなわち、*honoriccprofile* オプションはつねに *true* です。

注 ISO 15444-6 に従った JPM 複合画像ファイル（通常 *.jpm）には対応していません。

PDF/X-4/5 に対する追加の JPEG 2000 の制約（JPEG 2000 は、PDF 1.4 に基づいた PDF/X-3 では許されません）：

- カラーチャンネルの数は 1・3・4 のうちのいずれかである必要があります。
- 各カラーチャンネルのビット深度は 1・8・16 のうちのいずれかである必要があります。
- すべてのカラーチャンネルが同一のビット深度を持っている必要があります。
- ちょうど 1 個の色空間定義が JPEG 2000 画像ファイル内に存在する必要があります。
- CMYK 画像は、出力条件が CMYK デバイスである場合、または *defaultcmyk* オプションが与えられている場合にのみ使用できます。

PDF/A-2 に対する追加の JPEG 2000 の制約（JPEG 2000 は、PDF 1.4 に基づいた PDF/A-1 では許されません）：

- カラーチャンネルの数は 1・3・4 のうちのいずれかである必要があります。
- すべてのカラーチャンネルが同一のビット深度を持っている必要があります。
- そのJPEG 2000画像内の色空間指定の数が1より大きい場合には、*APPROX* フィールド内に値 0x01 を持つ色空間指定がちょうど 1 個存在する必要があります。

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。これは以下のように行います：

- 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 へコピーします。*PDF_load_image()* に対して以下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>* を与えることによって、すでにページ 1 とともにコピーされているグローバルセグメントへの参照を PDFlib が作成できるようにする必要があります。*PDF_load_image()* に対して以下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コピーします。

GIF 画像　PDFlib は、あらゆる種類の GIF とあらゆるパレットサイズに対応しています。GIF 画像はつねに Flate 圧縮で再圧縮されます。GIF ファイルは複数の画像を内容として

いる場合があります（190 ページ「複数ページ画像」参照）: GIF ファイル内の特定の画像を選択するには *page* オプションを使用します。

TIFF 画像　PDFlib は、すべての関連する種類の TIFF 画像を処理します：

- 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。取り込まれる TIFF 画像内の色空間は、以下の例外を除き、変更されずに保持されます：CIELab カラーを持つLZW圧縮されたTIFF画像はRGBへ変換され、CIELab色空間を保持しません。
- 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。
- 元の TIFF 形式を、4GB を超えて拡張した BigTIFF 形式。

画像を取り込む際、以下の TIFF 機能は処理されます：

- 複数の画像を含んだ TIFF ファイル（190 ページ「 複数ページ画像」参照）。TIFF ファイル内の特定の画像を選ぶには *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（195 ページ「8.1.4 画像透過」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画像内の埋め込み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。
- JPEG画像がExifマーカを含んでいる場合には、そのExifマーカ内の色空間情報は解釈されます。それが sRGB 色空間を示している場合には、その sRGB ICC プロファイルがその画像に添付されます（ただし、その画像が明示的に埋め込まれた ICC プロファイルを含んでいる場合、または *honoriccprofile* オプションが *false* である場合、またはその画像に *iccprofile* オプションで他の ICC プロファイルが割り当てられている場合は、この限りではありません）。
- 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。

BMP 画像　PDFlib は、以下の種類の BMP 画像に対応しています：

- 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）。
- 白黒または RGB カラー（インデックスカラーも直接カラーも）。
- 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、非圧縮処理なしにパススルーされます。この形式は実は、プレーンな 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。

CCITT 画像データを与えるには、***width*・*height*** オプションを与える必要があります。なぜなら PDFlib はデータからその画像の寸法を割り出せないからです。CCITT 圧縮の具体的な種別を与えるには ***K*** オプションを用います。

RAW 画像　非圧縮処理された RAW 画像データは、さまざまな特殊な用途に有用でしょう。なお、この形式は、Adobe アプリケーション群が生成する Camera Raw ファイルとは関係ありません。

RAW 画像データを与えるには、***width*・*height*・*bpc*** オプションを与える必要があります。なぜなら PDFlib は画像データからその画像の寸法を導出できないからです。その色空間は ***components*** オプションから導出できます：1 成分ならグレースケール画像を暗示し、3 成分なら RGB 画像を、4 成分なら CMYK 画像を暗示します。あるいは、***colorize*** オプションを通じて、スポットカラーか DeviceN カラーを画像に適用することもできます。この場合には、その色空間と、色成分の数は、その色空間ハンドルから導出されます。与える画像データ長は

```
[width x components x bpc / 8] x height
```

バイトに等しい必要があります。ここで、カッコ内の小数は切り上げです。画像サンプルは、上から下へ、かつ左から右への順序と見なされます（座標変換は行われていないと前提されます）。16 ビットサンプルは、最上位バイトを最初に与える必要があります（ビッグエンディアンバイト順序）。***bpc*** が 8 より小さい場合には、各ピクセル行はバイト境界で開始し、カラー値はバイト内で左から右へ詰め込まれる必要があります。カラーチャンネル群はつねに交互配置されます。すなわち、最初のピクセルに対するすべてのカラー値を最初に与える必要があり、その後に 2 番目のピクセルに対するカラー値群を与え、以下同様とする必要があります。画像に合致するオプション値群を与えるのはユーザー側の役割です。そうしない場合には、壊れた PDF 出力が生成される場合があり、Acrobat がメッセージ「***Insufficient data for an Image***」を発するおそれがあります。

カラー値群の極性は、色関連オプション群に対する場合と同じです（79 ページ「4 章 色空間」参照）。なお、Adobe Photoshop は、期待される PDF 極性を、グレースケール画像と RGB 画像に対してのみ用いており、RAW CMYK 画像と追加カラーチャンネルに対しては反転極性を用いています：PDF と PDFlib は 0＝強度なしと期待するのに対し、Photoshop は 0 ＝最大強度と見なします。この極性を調整するには、その画像を読み込む際に ***invert*** オプションを適用します。

8.1.3 クリッピングパス

PDFlib は、Adobe Photoshop や互換ソフトで作成された 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：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 8.1 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。

クリッピングパスを記述するためのベクトル演算は、*PDF_fit_image()* が呼び出されるたびに、PDF 出力へ書き出されます。クリッピングパスを持った画像 1 個を、その文書内に複数回配置する場合には、出力ファイルサイズを削減するために、その画像をテンプレート内にラップすることを強く推奨します。これは *PDF_load_image()* の *templateoptions* オプションで実現できます。

8.1.4 画像透過

画像透過は、さまざまな芸術的効果などに有用です。たとえば、画像のどうでもいい部分を無視して、関心の対象である人物や物だけを見せることができます。PDFlib は、画像透過のためのいくつかの手法に対応しています：

- アルファチャンネル（ソフトマスクとも呼ばれる）は、各画像ピクセルに対して透過値を指定するものです。このアルファチャンネルは、取り込む画像ファイルに含まれていることもありますし、別のグレースケール画像として指定することもできます。ピクセルあたり 1 ビットより多いソフトマスクは PDF/A-1・PDF/X-1a/3 では許されていません。
- クロマキーマスク処理は、単色値またはカラー値範囲を透過と指定します。この透過カラー値（1 つないし複数）は、取り込む画像から来ることもありますし、*chromakey* オプションを通じて指定することもできます。
- ステンシルマスク処理は、透過領域を指定する別のビットマップ画像を用います。

表 8.1 に、アルファチャンネル・クロマキーチャンネル・ステンシルマスクを用いて画像をマスクする場合に利用可能な手法をまとめました。このマスクの極性は、アルファ方式とステンシル方式とで異なります：アルファマスクでは黒い領域を通じて背景が見えますが、ステンシルマスクでは白い領域を通じて背景が見えます。

これ以外にも、*colorize* オプション（199 ページ「8.1.5 画像にスポットカラーか DeviceN カラーで着色」参照）・ブレンドモード・gstate ソフトマスク（100 ページ「4.9 オブジェクトの色を変更」参照）を用いて、画像の色のさまざまな効果を実現できます。比較のため、画像にスポットカラーか DeviceN カラーで着色する効果も表 8.1 に挙げています。

図 8.1
クリッピングパスを利用して前景と背景を分離

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。

表 8.1 アルファマスク・クロマキーマスク・ステンシルマスク・画像着色で使える方式

マスク／着色手法	ベース画像	補助マスク画像	マスクまたはベース画像内のピクセル群の効果
画像または masked オプションからのアルファチャンネル（ソフトマスク）			
内蔵アルファチャンネルによる透過画像	内蔵アルファチャンネルを持つ TIFF・PNG・JPEG 2000 画像	（画像ファイルから）	（TIFF/PNG/JPEG 2000 画像形式に従った透過）
画像および別のアルファチャンネル画像	任意の画像。オプション masked で補助画像内のソフトマスクを参照	ビットマップまたはグレースケール画像	黒マスク＝背景 グレーマスク＝透け 白マスク＝ベース画像[1]
画像または chromakey オプションからのクロマキーマスク処理			
内蔵クロマキー値を持った画像	内蔵クロマキーエントリを持った GIF または PNG 画像	—	クロマキー色＝透過 画像のその他の色＝ベース画像
画像および別のクロマキー値または範囲	JPEG・JPEG 2000 以外の任意の画像。オプション chromakey で透過カラー値（1 つないし複数）を指定	—	クロマキー色（1 つないし複数）＝透過 画像のその他の色＝ベース画像
mask オプションを用いたステンシルマスク処理			
ステンシルとしてのマスク画像を通して描画	—	オプション mask を用いて読み込まれたビットマップ画像	黒マスク＝塗り色 白マスク＝背景 [1]
ステンシル画像を用いてベース画像をマスク	任意の画像。オプション masked で補助画像内のステンシルマスクを参照	オプション mask を用いて読み込まれたビットマップ画像	黒マスク＝ベース画像 白マスク＝背景 [1]
colorize オプションを用いた画像着色			
スポットカラーを用いて画像に着色	グレースケール画像（ピクセルあたり 1 ビット以上）。オプション colorize でスポットカラーを参照	—	黒画像＝ベタスポットカラー グレー画像＝薄めたスポットカラー 白画像＝白
DeviceN カラーを用いて画像に着色	n 色画像。オプション colorize で DeviceN カラーを参照	—	画像内の 0 ＝白 中間＝薄めた DeviceN 画像内の 1 ＝ベタ DeviceN カラー

1. ステンシルまたはアルファ画像に invert オプションを適用することによってマスクの効果を反転させることもできます。

内蔵アルファチャンネルを持った画像 PDFlib は、以下の画像形式から内蔵アルファチャンネルを読み取ります（*ignoremask=true* の場合を除きます）：

- TIFF 画像：1 個のアルファチャンネルを内容として持っていることがあり、PDFlib はそれを使用します。あるいは TIFF 画像は、名前で識別される複数のアルファチャンネルを含むこともできます。複数のチャンネルが TIFF 画像内に存在する場合には、PDFlib はデフォルトでは、その先頭のアルファチャンネルを使用します。他のチャンネルを明示的に選択することも可能です。そのためには、*alphachannelname* オプションを用いてその名前を与えます：

```
image = p.load_image("tiff", filename, "alphachannelname={apple}");
```

- PNG 画像：1 個のアルファチャンネルを内容として持っていることがあり、PDFlib はそれを使用します。
- JPEG 2000 画像：1 個のアルファチャンネルを内容として持っていることがあり、PDFlib はそれを使用します。

注 Photoshop は、完全なアルファチャンネルのかわりに、TIFF 画像内に透過背景を作成することもできます。これに使用される独自形式に PDFlib は対応していません。このような透過画像を PDFlib で使うには、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」を選択する必要があります。

別のグレースケール画像をアルファチャンネルとして使用　画像ファイル内の内蔵アルファチャンネルのかわりに、第二の画像をアルファチャンネルのソースとして使うこともできます。グレースケール画像は、すべての種類が、アルファチャンネルとして使用するのに適しています。このマスクが埋め込み ICC プロファイルを内容として持っている場合には、*honoriccprofile=false* を用いてこれを無視する必要があります。TIFF 画像に対しては、*PDF_load_image()* に *nopassthrough* オプションを与えることによってマルチストリップ画像を避けることを推奨します。BMP 画像は他の画像種別とは向きが異なっています。ですので BMP 画像をマスクとして使用できるようにするには、まず ***x*** 軸を軸として鏡映させる必要があります。

アルファ画像内の白いピクセルの部分においては、ベース画像の照応する領域が描画され、黒いマスクピクセルの部分においては背景を見通せます。マスクがピクセルあたり 1 ビットよりも多くを使用している場合には、中間値は、前景画像を背景に対してブレンドさせることによって、透け効果を生みます。

マスクを読み込んだ後、それをベース画像に適用するには、*masked* オプションを用います：

```
// アルファチャンネルとして使いたいマスク画像を読み込む
mask = p.load_image("png", maskfilename, "");
if (mask == -1)
        throw new Exception("エラー：" + p.get_errmsg());

// ベース画像を読み込んでそれをマスク
image = p.load_image(type, filename, "masked=" + mask)
if (image == -1)
        throw new Exception("エラー：" + p.get_errmsg());

p.fit_image(image, x, y, "");
```

画像アルファチャンネルを視覚化　ときには画像アルファチャンネルをグレースケール画像として見せたいこともあるでしょう。これはデバッグや、アルファチャンネルの再利用に役立つでしょう。これを実現するには、マスクへの画像ハンドルをメイン画像から取得します。それには *PDF_info_image()* とキーワード *imagemask* を用います：

```
image = p.load_image("auto", "image.jpg", "");
if (image == -1)
        throw new Exception("エラー： " + p.get_errmsg());

alpha = (int) p.info_image(image, "imagemask", "");
if (alpha != -1)
        p.fit_image(alpha, 0.0, 0.0, "");
```

ここでアルファチャンネルはグレースケール画像として扱われます：透過領域が黒く見えます。

クロマキーマスク処理　画像は、マスクされるべき単色または色範囲を指定することができます。指定された範囲の色を持つピクセルは描画されず、したがって背景が透けて見えます。映像技術方面ではこの技法はクロマキーまたはブルースクリーンマスク処理と呼ばれています。これは ***chromakey*** 画像オプションで使えます。このオプションは、画像の色空間の色要素の数を n としたとき、範囲 0 ～ $2^{\text{色要素あたりビット数}}$ の n 個または 2×n 個の整数を受け付けます。

このリストの中の各ペアは、色要素範囲の、境界を含む下限と上限を内容とします。すべての色要素（***decode*** 値群も ***invert*** オプションを用いた色反転もまだ適用していない段階の）がこの指定範囲群に収まっているところのピクセルは透過として扱われ、すなわち描画されずに背景が透けて見えます。

n 個のペアでなく n 個の値を与えた場合には、各要素範囲は 1 個の単色値のみが内容となります。すなわち、各リスト値は色要素に対する下限と上限の両方を記述していることになります。

chromakey オプションのさまざまな例を表 8.2 に挙げます。

表 8.2　chromakey 画像オプションの使用例

クロマキーオプション	画像の色空間	効果
`chromakey={255 255 255}`	RGB	白ピクセルを透過として扱う
`chromakey={0 255 128 255 0 255}`	RGB	50% を超えるグリーンを持つすべてのピクセルを透過として扱う
`chromakey={242 255 242 255 242 255}`	RGB	95% より明るいすべての色を透過として扱う
`chromakey={242 255 242 255 242 255}` `decode={0 0.95 0 0.95 0 0.95}`	RGB	95% より明るいすべての色を透過として扱い、かつ、95% で急に切り落とされるのを防ぐために残りの色を拡散させる

クロマキーマスク処理は、以下の場合には自動的に適用されます（***ignoremask=true*** の場合を除きます）：

- GIF 画像は、1 個の透過カラー値（パレットエントリ）を持っていることがあり、PDFlib はそれに従います。
- PNG 画像は、1 個の透過カラー値を持っていることがあり、PDFlib はそれに従います。複数のカラー値が、照応するアルファ値とともに存在している場合には、50% 未満のアルファ値を持つ最初のもののみが用いられます。

ステンシルマスク　ステンシルマスクは、白いピクセルが透過として扱われる、ビット深度 1 のビットマップ画像です：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。0 ピクセル値を持つ領域は、カレント塗り色で描くこともできますし、それを用いて別の画像の一部を見せることもできます。

ステンシルマスクをカレントの塗り色で着色するには、その画像を ***mask*** オプションを用いて読み込み、そしてその画像を配置する前にカレント塗り色を設定する必要があります：黒いピクセルがカレント塗り色で着色され、白い領域は変更されません：

```
mask = p.load_image("tiff", maskfilename, "mask");
if (mask == -1)
        throw new Exception("エラー：" + p.get_errmsg());
```

```
p.set_graphics_option("fillcolor=red);
p.fit_image(mask, x, y, "");
```

もしもこのステンシル画像を使って他の画像を *masked* オプションでマスクした場合には、ステンシルマスクの黒い領域においてはそのベース画像のピクセルが見え、白い領域においては背景が見えます。

8.1.5 画像にスポットカラーか DeviceN カラーで着色

ステンシルマスクでは、ビットマップ画像の不透過部分に色が適用されましたが、これと同様に、PDFlib では、画像にスポットカラーか DeviceN カラーで着色することもできます。

グレースケール画像にスポットカラーで着色　白黒またはグレースケール画像にスポットカラーで着色することができます。画像にスポットカラーで着色するには、その画像を読み込む際に *colorize* オプションを与える必要があります。このオプションは、*PDF_makespotcolor()* を用いて作成されたスポットカラーハンドルを内容とします：

```
spot = p.makespotcolor("PANTONE Reflex Blue CV");

String optlist = "colorize=" + spot;
image = p.load_image("tiff", "image.tif", optlist);
```

86 ページ「4.4 Pantone・HKS・カスタムスポットカラー」で説明しているように、スポット色空間は通常、0 ＝色なし＝白と期待します。これに対し、グレースケール色空間では 0 ＝黒です。白を白のままにするために、PDFlib は、画像にスポットカラーで着色する際には、画像の極性を反転させます。すなわち、画像の暗い領域が最大のスポットカラー強度になります。この色極性を反転させるには *invert* 画像オプションを用います。

n チャンネル画像を DeviceN カラーで着色　*PDF_create_devicen()* を用いて作成された DeviceN カラーハンドルを、*colorize* 画像オプションで与えると、DeviceN カラーを n 色画像に適用することができます。画像を DeviceN カラーで着色するためには、その画像データが、ちゃんと N 個のカラーチャンネルを内容として持っている必要があります。

90 ページ「4.5 DeviceN カラー」で説明しているように、DeviceN 色空間は 0 ＝色なし＝白と期待します。これに対し、グレースケール色空間では 0 ＝黒です。画像をスポットカラーで着色した場合と異なり、PDFlib は、画像を DeviceN カラーで着色する際には、画像の極性を反転させ<u>ません</u>。ですので、スポットカラーで着色する場合と、*N=1* の DeviceN カラーで着色する場合とでは、期待される色極性が異なります。この色極性を反転させるには *invert* 画像オプションを用います。

画像に DeviceN 色空間で着色するコード断片が 91 ページ「 CMYK プロセスカラーに基づく DeviceN 色空間」にあります。

注 DeviceN カラーで着色できるのは RAW 画像だけです。

クックブック コードサンプルがクックブックの `images/colorize_image_with_DeviceN_color` トピックにあります。

8.1.6 復号配列を用いてカラー値を変える

画像のカラー値は、線形の復号関数を用いてさらに変えることもできます。この方式は、*decode* 画像オプションを用いて使えます。これは、画像の色空間の要素数を *n* としたときに、*2*×*n* 個の float またはパーセント値を受け付けます。ですので、復号配列を適用す

るには、画像の色空間に関する情報が必要です。復号配列内の数値の各ペアは、色要素値の下限と上限である 0 と $2^{\text{色要素あたりビット数}}$ がマップされるターゲットカラー値を記述します：中間の値は線形に内挿されます。デフォルトでは、たいていの色空間の場合、画像の色要素値は範囲 0 ～ 1 へマップされます。復号配列を利用すると、カラー値の範囲を縮めたり拡げたり限ったりすることによって、特定のカラー効果を適用することが可能です。この技法は、画像が着色されている場合や、マスクとして用いられている場合にも役立つでしょう。

復号値は、加法混色の色空間の場合と、減法混色の色空間の場合とで、効果が異なります。たとえば、値を大きくするほど、RGB カラーでは明るくなり、CMYK カラーでは暗くなります。たいていの色空間では（Lab・インデックスを除き）、典型的な値は範囲 0 ～ 1 の中にあります。この範囲の外の値も許されますが、ビューアは結果のカラー値を、その色空間で適切な範囲へ切り捨てます。これを利用すると特定の効果が得られます。たとえば、ある範囲の明るさの色を強制的に白にするなどです。

別の画像へのソフトマスクとして用いられている画像に *decode* オプションを適用すると、そのマスクの効果を変えることができます。たとえばそのマスクを和らげたりすることができます。

以下の画像オプションは、ブラックのターゲット値を、デフォルト範囲 0 ～ 1 から、新しい範囲 0.2 ～ 1.2 へずらさせています。すなわち、CMYK 画像のブラックチャンネルを 20% 増加させ、画像を強めています：

```
decode={0 1  0 1  0 1  0.2 1.2}
```

decode オプションのさまざまな例を表 8.3 に挙げます。

表 8.3 decode 画像オプションの使用例

復号オプション	画像の色空間	効果
decode={1 0 1 0 1 0}	RGB	画像の色を反転。invert と同じ
decode={-0.5 1.5 -0.5 1.5 -0.5 1.5}	RGB	コントラストを強める
decode={0.5 1.5 0.5 1.5 0.5 1.5}	RGB	すべての色を 50% 明るくする
decode={-0.2 0.8 -0.2 0.8 -0.2 0.8}	RGB	すべての色を 20% 暗くする（100% ＝白なので）
decode={-0.2 0.8 -0.2 0.8 -0.2 0.8 -0.2 0.8}	CMYK	すべての色を 20% 明るくする（100% ＝暗なので）
decode={0 1 0 1 0 1 0 0}	CMYK	ブラックチャンネルを除去。鈍い画像になります
decode={0 1 0 1 0 1 0.2 1.2}	CMYK	ブラックチャンネルを 20% 大きくする。強い画像になります
decode={1 0 0 1 0 1 0 1}	CMYK	シアンチャンネルを反転させ、他のチャンネルは変更しない。芸術向け以外に有用性はなさそう

8.2 SVG グラフィック

8.2.1 対応 SVG 種別

PDFlib は、W3C の述語によれば「規格準拠した高品位な静的 SVG ビューア」です。PDFlib は、SVG グラフィックを以下のように受け付けます：

- PDFlibは、W3Cが発行したSVG 1.1（Second Edition）を実装しています。
- 以下の Unicode 形式とエンコーディングに対応しています：
  UTF-8・UTF-16・ISO 8859-1 ～ ISO 8859-15・ASCII
- CSS スタイル付けが可能ですが、ただし対応していない CSS 要素もあります。
- プレーンテキスト形式のSVGファイルのほかに、Flate圧縮されたSVGファイル（**.svgz*）にも対応しています。
- CEF 形式のフォントに対応しています。CEF フォントは、SVG 仕様の一部分ではありませんが、いくつかの Adobe アプリケーションによって SVG グラフィック内に埋め込まれます。
- 色は、SVG Color 1.2 ドラフトに従った追加の色空間を用いて指定することも可能です。スポットカラーの濃度値のための PDFlib 独自拡張の色空間も利用可能です（207 ページ「8.2.6 SVG カラー拡張」参照）。

制約について 210 ページ「8.2.8 対応していない SVG 機能」を参照してください。

8.2.2 SVG 処理上の考慮事項

基本的な SVG の取り扱い　ベクトルグラフィックを PDFlib で埋め込むことは容易に実現できます。まず、そのグラフィックファイルを PDFlib 関数で開く必要があります。この関数はそのグラフィックを解釈して、内部表現をメモリ内に格納します。この関数 *PDF_load_graphics()* は、グラフィック記述子の役割を持つハンドルを返します。このハンドルは、*PDF_fit_graphics()* への呼び出しで、位置付け・拡縮オプションとともに使うことができます：

```
graphics = p.load_graphics("auto", "graphics.svg", "");
if (graphics == -1)
        throw new Exception("エラー: " + p.get_errmsg());

if (p.info_graphics(graphics, "fittingpossible", optlist) == 1)
        p.fit_graphics(graphics, 0.0, 0.0, "");
else
        System.err.println("グラフィックを配置できません: " + p.get_errmsg());

p.close_graphics(graphics);
```

PDF_fit_graphics() の最後の引数は、位置付け・拡縮・回転のためのさまざまなオプションをサポートしているオプションリストです。これらのオプションに関する詳細は、218 ページ「8.4 画像・グラフィック・取り込み PDF ページを配置」で説明します。

クックブック　SVG の取り扱いのためのコードサンプルが、PDFlib クックブックの graphics カテゴリにあります。

SVG 内容をインライングラフィックとするかフォーム XObject（テンプレート）とするか　PDFlib は、ベクトルグラフィックを取り込むための、以下の方式をサポートしています：

- デフォルトでは、グラフィックデータは、ページ・パターン・テンプレート・グリフ記述の内容ストリーム内にインラインに書き込まれます。これがデフォルトの動作であり、これは、そのグラフィックがその文書内にちょうど 1 回だけ配置され、かつ取り込んだグラフィックの不透明度を変えたくない場面に対して推奨されます。もし *PDF_fit_graphics()* が複数回呼び出されると、そのグラフィックデータは PDF 出力へ何度も書き込まれ、出力ファイルサイズが増大します。
- そのグラフィックをその文書内に複数回配置するつもりの場合には、*PDF_load_graphics()* の *templateoptions* オプションを推奨します。取り込んだグラフィックの不透明度を変えるつもりの場合には、オプション *templateoptions= {transparencygroup={isolated=true}}* が必要です。これは PDF フォーム XObject（テンプレート）を作成します。すなわち、そのグラフィックデータはその PDF 文書内に、任意の回数参照されることのできる別個の実体として格納されます。テンプレートのためのグラフィックデータは、文書の最後に、あるいは *PDF_close_graphics()* が呼び出されたときに、PDF 出力へ書き出されます。この方法なら、出力ファイルサイズは最適化されます。ただし、グラフィック内のリンクは PDF 注釈へは変換されなくなります。

同一のグラフィックを複数の文書内で使用　グラフィックは、カレント出力文書とは独立に読み込んだり閉じたりすることもできます。*PDF_load_graphics()* が呼び出された時点で、そのグラフィックの内部表現が作成されます。これは、その照応する *PDF_close_graphics()* への呼び出しまでメモリ内に保持されます。グラフィックを、文書をまたいでメモリ内に保持することは、同一のグラフィックが多数の出力文書内に配置される場面において、そのグラフィックを 1 回だけ読みこめば済むので、パフォーマンス上の利点があります。たとえば、アプリケーションが、シンボルや背景アートワークや企業ステーショナリを持つグラフィックを 1 回読み込んで、そのグラフィックが必要とされる各文書内で *PDF_fit_graphics()* を呼び出すことができるでしょう。

SVG の処理上の問題をチェック　*PDF_load_graphics()* は SVG グラフィックを読み込みますが、完全な処理と分析はその後、フォーム XObject の作成と、読み込みのスコープに応じて、*PDF_fit_graphics()*・*PDF_close_graphics()*・*PDF_end_document()* のうちのいずれかの時点でしか行われません。エラー状況のなかには、完全処理が行われている間にしか検出できないものもありますので、これらの関数は、問題が発見されれば例外を発生させる可能性があります（なぜならこれらはエラー値を一切返すことができないので）。そのような例外を回避するには、グラフィックを *PDF_info_graphics()* の *fittingpossible* キーワードでチェックすることができます。これは、すべての処理ステップを実行しつつも、出力を一切作成せず、SVG 処理の成功（か否か）を報告します。もしこのチェックが成功すれば、その画像が配置される際に *PDF_fit_graphics()* は例外を発生させません。もしこの *fittingpossible* チェック中にエラーが発生すれば、*errorpolicy* にかかわらず、*PDF_info_graphics()* は 0 を返します。まとめると：

- この *fittingpossible* チェックは、後の *PDF_fit_graphics()*・*PDF_close_graphics()*・*PDF_end_document()* での例外を回避します。PDF 出力は例外の後では使用不能になりますので、これは推奨されるアプローチです。
- この *fittingpossible* チェックを省くと、SVG の読み込みは速くなりますが、そのSVGデータが引き起こす例外が後で発生するおそれがあります。この設定は、*PDF_fit_graphics()* での例外が許容できる場合に、SVG の読み込みを速めるために使用できます。たとえ

ば、アプリケーションが 1 個の SVG グラフィックファイルを 1 個の PDF 文書へ変換し、それ以外のページ内容を追加しない場合には、これは許容できるアプローチです。

この *fittingpossible* チェックは、カレントでアクティブなグローバル・文書・ページオプション群と、カレント出力インテントを使用します。ですのでこのチェックは、*PDF_fit_graphics( )* への実際の呼び出しの直前にのみ走らせることを推奨します。

8.2.3 SVG グラフィックの視覚的寸法

SVG グラフィックは、その幅と高さを、その SVG グラフィックのターゲットビューポート（ブラウザウィンドウや、PDF ページの一部分など）へのマッピングを定義している *svg* エレメント内で指定します。しばしば、このビューポートの寸法は絶対単位で指定されます。例：

```
<svg xmlns="http://www.w3.org/2000/svg" width="640mm" height="480mm">
```

PDFlib は、この *width・height* 属性をポイントへ変換し、そしてそれらを、*PDF_info_graphics( )* の *graphicswidth・graphicsheight* キーワードを通じて得られるようにします。寸法がピクセル（*px*）で指定されている場合には、PDFlib は 1pt=1px を用います。これらの値は、はめ込み操作のためのオブジェクト枠を計算するためにも使われます。*svg* エレメントは、ビューポート内のウィンドウを指定する *viewBox* 属性を内容として持つ場合もあります。

SVG グラフィック内で指定されている寸法値を、*forcedwidth・forcedheight* オプションを用いてオーバライドすることもできます。*matchbox* オプションのサブオプション *clipping* を用いると、グラフィックをオブジェクト枠の一部へ切り出すこともできます。

絶対寸法値のない SVG グラフィック　SVG グラフィックのなかには、*width・height* を欠いているか、あるいは以下の例のように相対値のみを内容としているために、絶対寸法情報を内容として持たないものもあります：

```
<svg xmlns="http://www.w3.org/2000/svg" width="100%" height="100%">
```

これはスタンドアロン用途を想定されていない SVG グラフィックで用いられています。PDFlib はこの場合にも対応します。それにはユーザーがオプション *boxsize* と *fitmethod=nofit* を用いてはめ込み枠を指定しておく必要があります。このはめ込み枠はオブジェクト枠として用いられます。その SVG グラフィックはこのはめ込み枠の辺で切り抜かれます。はめ込み枠を指定されていない場合には PDFlib は、*viewBox* 属性（あれば）をオブジェクト枠として用います。SVG グラフィック内で絶対寸法情報が得られないときは、*PDF_load_graphics( )* の *fallbackwidth・fallbackheight* 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が与えられていないときは、PDFlib は、その SVG グラフィックの外接枠を、*PDF_fit_graphics( )* か *PDF_info_graphics( )* への最初の呼び出しの際に算出します。算出される外接枠は、線幅と過大なグリフが考慮されていないので、小さすぎる場合があります。この場合には、この算出された枠を、*bboxexpand* オプションを用いて拡げることもできます。デフォルトでは、この算出された外接枠は、そのグラフィックが座標系の中の元の場所に位置付けられるよう移動されます（すなわち、この枠は必ずしも原点を隅として用いるわけではありません）。

8.2.4 フォント選択

フォント選択アルゴリズム　SVG におけるフォント選択は、以下のプロパティによって制御されます：

```
font-family
font-style
font-weight

font-stretch
font-variant
font-size
font-size-adjust
```

これらのプロパティのうち、最初の 3 つだけが、外部フォントの選択に関連します。適切なフォントを選択するため、PDFlib は以下のフォント名を構築します：

```
<font-family>,<font-weight>,<font-style>
<font-family>-<font-weight><font-style>
<font-family>,<font-normweight>,<font-style>
<font-family>,<font-weight>,<font-normstyle>
<font-family>,<font-normweight>,<font-normstyle>
```

ここで ***<font-normweight>*** は

```
Regular, Thin, Extralight, Light, Medium, Semibold, Bold, Extrabold, Black
```

のうちのいずれか一つであり、また ***<font-normstyle>*** は

```
Italic
```

たとえば、以下の SVG フォント指定に対して：

```
font-family="Tahoma" font-weight="Bold" font-style="Italic"
```

PDFlib は、Windows ホストフォントを指定するためにも用いることができるフォントスタイルを指定するためのこのカンマ区切り PDFlib 文法を用いて、フォント ***Tahoma,Bold,Italic*** を検索します。

PDFlib はその後、処理が成功してフォントが読み込めるまで、上に挙げたフォント名を順次読み込み試行します。このリスト内のフォント名は、フォントリソース指定でも用いることができます。例：

```
p.set_option("FontOutline={<fontname>=<filename>}")
p.set_option("FontNameAlias={<fontname>=ArialMT}")
```

すべての試みが失敗した場合には、PDFlib は、名前 ***<font-family>*** を持つフォントを読み込み試行して、必要に応じて ***Bold・Italic*** プロパティを擬似表現します。

ブラウザによっては、指定されたフォントファミリが見つからない場合には、フォント選択プロパティ群を無視するものがあります。PDFlib はそういうことをしませんので、PDFlib のフォント構成機構を通じて適切なフォントが得られるように配慮する必要があります（136 ページ「6.4 フォントを読み込む」参照）。

この ***font-family*** プロパティは、複数のフォントファミリ名を内容とする場合もあります。例：

```
font-family="Georgia, 'Minion Web', 'Times New Roman', Times, 'MS PMincho', serif"
```

この場合、PDFlib は、このリスト内の特定のフォントが読み込めなかったときには、その次のフォントを読み込み試行します。1 つの *font-family* リストに対して何らかのフォントが読み込めた場合には、PDFlib は、このリスト内の残りの *font-family* 群を、最初に読み込んだフォント（マスタフォント）に対するフォールバックフォント（149 ページ「6.4.6 フォールバックフォント」参照）として読み込もうと試みます。もしこのマスタフォントが、もっと前に読み込まれていたためにすでにフォールバックフォント群を持っているときには、新しいフォールバックフォント群は、既存のフォールバックフォントのリストの末尾に追加されます。

PDFlib が SVG グラフィックのためのフォントを読み込むと試みる際には、以下のオプションでデフォルトで用いられます：

```
embedding subsetting skipembedding={fstype latincore}
```

これらのオプションをオーバライドするには、*PDF_load_graphics()* のオプション *defaultfontoptions* を用います。

フォント構成 Windows システムでは、PDFlib は、システムにインストールされているすべてのフォントにアクセスできます（147 ページ「6.4.5 Windows・macOS 上のホストフォント」参照）。たとえば、SVG フォント指定

```
font-family="Verdana" font-weight="bold"
```

は、PDFlib フォント名 *Verdana,Bold* となります。他のオペレーティングシステムでは、PDFlib は、*FontOutline* リソースが以下の方式で指定されていればフォントを発見します：

```
<fontnamepattern>=<filename.xxx>
```

ここで、*<fontnamepattern>* は、上述のフォント名パターン群のうちの一つであり、また *xxx* は、その照応する、そのフォントアウトラインファイルのフォント名拡張子です。

上述のフォント名パターン群のうちの一つないし複数に整合するフォント名を持つ適切な *FontOutline* リソースは、*PDF_set_option()* の *enumeratefonts* オプションで自動的に作成することもできます。フォント構成に関して詳しくは 143 ページ「6.4.4 フォントを検索」を参照してください。

総称 SVG フォントファミリを PDF コアフォントへマップ PDFlib は、以下の形の *FontNameAlias* リソースを用いて、総称 SVG フォントファミリ *monospace*・*sans-serif*・*serif* を、総称 SVG フォントファミリが最初に出現した時点で、自動的に Latin コアフォントへマップします：

```
p.set_option("FontNameAlias={monospace=Courier}")
p.set_option("FontNameAlias={monospace,Bold=Courier-Bold}")
```

総称フォント名マッピングの完全なリストは以下のとおりです（総称フォントファミリ *cursive* と *fantasy* に対してはデフォルトマッピングはありません）：

```
monospace               Courier
monospace,Bold          Courier-Bold
monospace,Italic        Courier-Oblique
monospace,Bold,Italic   Courier-BoldOblique

sans                    Helvetica
sans,Bold               Helvetica-Bold
```

```
sans,Italic              Helvetica-Oblique
sans,Bold,Italic         Helvetica-BoldOblique

sans-serif               Helvetica
sans-serif,Bold          Helvetica-Bold
sans-serif,Italic        Helvetica-Oblique
sans-serif,Bold,Italic   Helvetica-BoldOblique

serif                    Times-Roman
serif,Bold               Times-Bold
serif,Italic             Times-Italic
serif,Bold,Italic        Times-BoldItalic
```

これらのマッピングは、以前にユーザーによって他の適切なリソースが一切指定されていない場合にのみ実行されます。

8.2.5 見つからないフォント、見つからないグリフを扱う

見つからないフォントとデフォルトフォント　すべてのフォント読み込み試行が失敗した場合、PDFlib は、*PDF_load_graphics()* の *defaultfontfamily* オプションで定義された *font-family* 名を持つデフォルトフォントを読み込もうと試みます。デフォルトでは、*Arial Unicode MS* フォントがもし得られるならそれが、そうでないなら Helvetica が用いられます。PDF のフォント構成の中で *Arial Unicode MS* フォントが得られるようにしておくか、あるいは、大きなグリフ集合を持つ別のフォントを *defaultfontfamily* で指定しておくことを強く推奨します。たとえば、*NotoSans-Regular* フォントを最終手段フォントとして構成するには、以下のオプションを用います：

```
defaultfontfamily={NotoSans-Regular}
```

個別のフォントを置換　入手できない、あるいは望ましくない（たとえば、充分なグリフを含んでいないという理由で。後述）特定のフォントを避けるために、より適切な別のフォントへそれをマップすることができます。フォント名エイリアス機能と *PDF_set_option()* をこの目的に使います（詳しくは 143 ページ「フォント名エイリアス設定」を参照）。たとえば、中国語のテキストが不適切に、中国語のグリフを一切含んでいない *Trebuchet MS* フォントで設定されている場合に、これを以下のようにして *Arial Unicode MS* へマップすることができます：

```
p.set_option("FontnameAlias={ {Trebuchet MS}={Arial Unicode MS} }");
```

フォント属性は自動的に追加されないことに留意してください。たとえば、もしこの *Trebuchet MS* フォントが属性 *font-weight="bold"* で使われているならば、このフォントのボールド版へのエイリアスを作成する必要があります：

```
p.set_option("FontnameAlias={ {Trebuchet MS,Bold}={Arial Unicode MS} }");
```

見つからないグリフを視覚化　選択されたフォントが、必要なグリフを含んでいない場合には、デフォルト置換グリフがかわりに用いられますので、テキストは全く見えなくなります。見つからないグリフを視覚化するには、*defaultfontoptions* オプションを用いて、見える置換グリフを指定することができます。たとえば、*PDF_load_graphics()* に対する以下のオプションは、すべての見つからないグリフに対して疑問符を表示します：

```
defaultfontoptions={replacementchar=?}
```

見つからないグリフに対して特定のフォールバックフォントを指定　SVG内で指定されているフォントが、そのテキストのための適切なグリフを含んでいない場合には、テキストが全く見えなくなります。よくある例として、中国語のテキストを、中国語のグリフを一切含んでいない欧文フォントで見せようとしている例を考えてみましょう。もちろん最善の解決策は、SVG内でそもそも適切なフォントを用いることでしょう。しかし、SVG内の不適切なフォントを扱わねばならない場合には、PDFlibでフォールバックフォントを指定することができます。このフォールバックフォントは、元のフォントが特定のグリフを与えないときに用いられます。

PDF_load_graphics() に対する以下のオプションは、Arial Unicode MSをフォールバックフォントとして指定しています：

```
defaultfontoptions={fallbackfonts={{fontname={Arial Unicode MS} encoding=unicode}}}
```

なお、さきに説明した *defaultfontfamily* オプションで指定されたフォントは、フォントが見つからないときに使用されるのに対して、この予備技法は、フォントは得られるが必要なグリフを全部は含んでいない場合にあてはまります。

グローバルなフォールバックフォントファミリを指定　*fallbackfontfamily*・*fallbackfontoptions* オプションを用いると、フォールバックフォント群のファミリと、照応するオプション群を指定することができます。

defaultfontoptions 内の *fallbackfonts* オプションは、ただ1つのフォントをフォールバックフォントとして使用するよう選択するのに対して、*fallbackfontfamily* を用いると、フォールバックフォント群のファミリを指定することができます。すなわち、スタイル属性群がこのフォントファミリ名に適用されます。ただし、指定されたフォントのスタイル変種が実際に得られることを前提としています。例：

```
fallbackfontfamily={Arial} fallbackfontoptions={encoding=unicode}
```

8.2.6 SVG カラー拡張

クックブック　**非sRGBカラーのためのコードサンプルがPDFlibクックブックのトピックcolor/svg_color_extensionにあります。**

SVG 1.1はデフォルトでは、テキストとベクトルグラフィックに対してsRGB色空間を使用しています。ですのでPDFlibは、PDF/XとPDF/Aに対しては、SVGグラフィックをデバイス依存カラーとして扱います。しかし、SVG内に埋め込まれている画像は他の色空間を使用している場合もあります。たとえば、埋め込まれているか参照されているJPEG画像がCMYK色空間を使用しているかもしれず、それはICCプロファイルを有しているかもしれないし、有していないかもしれません。RGBカラーに対するデフォルトsRGBプロファイルは、*PDF_load_graphics()* の *iccprofilergb* オプションを用いてオーバライドすることもできます。

sRGBに加えて、PDflibは、SVG Color 1.2ドラフト仕様に従った文法拡張を使用した追加の色空間に対応しています。表8.4に、使えるSVG色空間を、文法例とともに挙げます。これらの色空間に関する一般的な情報と、PDFlibにおけるそれらの扱いについては、79ページ「4章 色空間」を参照してください。

SVG 文法は、すべての種類に対して、sRGB フォールバックカラーを必須としています。PDFlibは、文法エラーのときと、スポットカラー定義が見つからないときと、ICCプロファイルが見つからないとき、またはオプション *forcesrgb* が与えられているときに、このフォールバックカラーを使用します。

ブレンドモードを援用するとSVG画像を着色または脱色することもできます(100ページ「4.9.1 ブレンドモードを用いて色を変える」参照)。

ICC ベースのカラー *icc-based* 色指定は、指定された ICC プロファイルを適用します。これはローカルファイルとして参照されていない場合には、*ICCprofile* リソースカテゴリに従って検索されます。*honoriccprofile=false* が与えられている場合には ICC プロファイルは無視されます。SVG グラフィック内で指定されているプロファイルを他のプロファイルでオーバライドするにはオプション *iccprofilegray/iccprofilergb/iccprofilecmyk* を使用します。

グレー・RGB・CMYK デバイス色空間 デバイス独自色空間 *device-gray・device-rgb・device-cmyk* は、一般的用途においては推奨されません。なぜならそれらの視覚表現は、PDF/A または PDF/A 出力インテント ICC プロファイルか然るべきデフォルト色空間によって制御されるのでない限り、特定の出力デバイスに依存してしまうからです。かわりに ICC ベースの色空間を用いて作業することを強く推奨します。例外として挙げられるのは、CMYK コントロールストリップを使って印刷機における濃度値を直接制御する場合や、トンボを印刷する場合です。

device-gray/rgb/cmyk 属性か参照画像によって生成されたデバイス独自カラーについては、*defaultimageoptions* の *iccprofile* サブオプションか *templateoptions* の *defaultgray/rgb/cmyk* サブオプションを通じて、ICC ベースのカラーへマップすることができます。

スポットカラー・DeviceN カラー *icc-named-color* 構文を使うと、スポットカラーを指定できます。ただし、名前付きカラーの ICC プロファイルには対応していないので、そのスポットカラーは PDFlib が知っているものである必要があります。そうでないと、sRGB フォールバックカラーが代替色として使用されます。したがって、カスタムのスポットカラーについては、*PDF_load_graphics()* を呼び出す前に自分のコードの中で定義しておく必要があります（88 ページ「カスタムスポットカラー」参照)。例：

```
// スポットカラー「CompanyRed」をLab代替値とともに定義
p.set_graphics_option("fillcolor={spotname {CompanyRed} 1.0 {lab 60 65 65}}");
```

device-nchannel 構文を使うと、DeviceN カラーを選択できます。この DeviceN カラーは、*PDF_create_devicen()* を用いて作成されたうえで（90 ページ「4.5 DeviceN カラー」参照)、*PDF_load_graphics()* に *devicencolors* オプションを通じて与えられている必要があります。すなわち、使える DeviceN 色空間は ***N*** の値ごとに 1 個だけであることになります。

シェーディング 表 8.4 に挙げる色空間は、SVG のグラデーションの中で使用されていることもありえます。ただし、PDF は、1 つのグラデーションの中ではすべてのストップカラーが同一の色空間から採られていることを要請しています。1 つのグラデーションの中で複数の色空間からのストップカラーを使っている場合には、すべての色に対してその sRGB フォールバックカラーが用いられます。

スポットカラーから作られるシェーディングは、スポットカラーが Lab 代替値とともに作成されていれば（PDFlib 内部で既知のスポットカラーはすべてこれに該当)、スポットカラーを温存します。CMYK など他の色空間で作成されたスポットカラーは温存されず、PDF 出力内で sRGB シェーディングとなります。

注 SVG における非 sRGB カラーは、まだ完全には標準化が済んでいませんので、その文法は将来変わる可能性もあります。

表 8.4 SVG 内の拡張色空間（forcesrgb を用いて無効化させることもできます）

色空間	SVG 構文例	注
デバイス独立色空間		
sRGB	`#FF0000`	SVG 1.1 のデフォルトの色空間

ICC ベースカラー	`#7F7F7F icc-color(gray.icc, 0.5)`
	`#CC6633 icc-color(rgb.icc, 0.8, 0.4, 0.2)`
	`#2B7FAB icc-color(cmyk.icc, 0.8, 0.4, 0.2, 0.0)`
		グレースケール・RGB・CMYK のプロファイルに対して、それぞれ 1 個・3 個・4 個のカラー値を与える必要があります。

CIELab	`#598237 cielab(50, -25, 35)`	
CIELch	`#FF007E cielch(50, 127, 0)` `#FF007E cielchab(50, 127, 0)`	明度・彩度・色相（度単位で）CIELab へ変換されます。

周知のスポットカラー	`#FFB12D icc-named-color(nmcl.icc, 'PANTONE 123 U')`
	`#FFCE4C icc-named-color(nmcl.icc, 'PANTONE 123 U', 0.5)`
	`#994F5B icc-named-color(nmcl.icc, 'HKS 18 Z')`
		プロファイル名は無視されます。SVG 文法への拡張として、PDFlib では、オプショナルな濃度値を使えます。濃度値がない場合は 1.0 と見なされます。

カスタムスポットカラー	`#FFE560 icc-named-color(nmcl.icc, 'CompanyColor')` `#FFF36D icc-named-color(nmcl.icc, 'CompanyColor', 0.5)`	そのスポットカラーがそれまでにすでに定義されていない限り、sRGB フォールバックカラーが代替色として使用されます。プロファイル名は無視されます。濃度値がない場合は 1.0 と見なされます。
n 色	`#FFE560 device-nchannel(0, 0.7, 0.2)` `#FFF36D device-nchannel(1 0 0 0 0.5 0.2 0)`	DeviceN 色空間ハンドルを devicencolors オプションで与える必要があります。n に対する DeviceN 色空間ハンドルが与えられていない場合には、sRGB フォールバックカラーが使用されます。
デバイス依存色空間		
DeviceGray	`#7F7F7F device-gray(0.5)`	PDF/A と PDF/X のデバイス色空間に対する要件に従う必要があります。
DeviceRGB	`#0000FF device-rgb(0, 0, 255)`	PDF/A と PDF/X のデバイス色空間に対する要件に従う必要があります。
DeviceCMYK	`#00AEEF device-cmyk(1, 0, 0, 0)`	PDF/A と PDF/X のデバイス色空間に対する要件に従う必要があります。

8.2.7 ベクトルグラフィック・テキスト以外の SVG 内容

SVG 内に埋め込まれた画像　PDFlib は、SVG 内の *image* エレメントを処理して、189 ページ「8.1 ラスタ画像」で解説したすべての画像形式と、ネストされた SVG グラフィックを受け付けます。画像データは、SVG ファイル内に埋め込まれていても、外部ファイル内に存在していてもかまいません。

SVG グラフィック内の画像は、自動的に処理されます。ただし、場合によっては、特定の画像処理オプション群を与えるほうがよい場合もあります。これは *PDF_load_graphics()* の *defaultimageoptions* オプションで実現できます。たとえば、以下のオプションを用いると、低解像度画像の見栄えを向上させるアンチエイリアスを適用できます（これは、画像にアンチエイリアスを適用する多くのブラウザの SVG 表示と整合します）：

```
defaultimageoptions={interpolate}
```

画像が入手できない場合（参照された外部画像ファイルが見つからないなどの原因で）、かつ、オプション *fallbackimage={}*（すなわち空のオプションリスト）が与えられている場合には、PDFlib は透明な灰色の市松模様を作成します。オプション *fallbackimage* の各種サブオプションを用いると、このパターンをカスタマイズしたり、カスタムの画像かテンプレートをフォールバック視覚効果として与えたりすることができます。見つからない画像が自動的に置換されることを避けたい場合には以下のオプションを用います：

```
errorconditions={references={image}}
```

SVG のリンク　SVG グラフィック内のリンクは、通常、生成される PDF 出力内のインタラクティブなリンク注釈へ変換されます：ただし、リンクの作成を無効化する条件がいくつかあります（PDFlib API リファレンス参照）。グラフィックの外部に位置するリンクは無視されます。PDF 注釈の *contents* オプションに、SVG リンクの *xlink:title* 属性がもしあればそれが、なければターゲット URI が記入されます。タグ付き PDF モードでは、生成されたリンクに対して、*Link* エレメント 1 個と、関連する OBJR エレメント 1 個が作成されます。ただし、カレントでアクティブなアイテムがページ装飾か擬似エレメントである場合を除きます。

SVG リンクの PDF リンクへの変換は、*PDF_fit_graphics()* の *convertlinks* オプションで無効にすることもできます。なお、そのグラフィックに対してテンプレート（フォーム XObject）が作成されるか、そのグラフィックがテンプレート上に配置される場合には、リンクは作成できません。

SVG 内のメタデータ　SVG グラフィックは、XMP メタデータを含んでいることもあります。デフォルトでは PDFlib は、出力ファイルサイズを削減するために、グラフィック内の SVG メタデータを無視します。ただし、SVG からテンプレートが作成される場合には、*PDF_load_graphics()* の以下のオプションで、その XMP メタデータを、生成される XObject に添付することもできます：

```
templateoptions={metadata={keepxmp=true}}
```

SVG グラフィックの *metadata・desc・title* エレメントの内容は、*PDF_info_graphics()* で、以下のパターンに従って取得できます：

```
idx = (int) p.info_graphics(svg, "description", "");
if (idx != -1)
        description = p.get_string(idx, "");
```

8.2.8 対応していない SVG 機能

対応していない機能の扱い　デフォルトでは、対応していない SVG 機能は無視されます。結果として、出力は作成されますが、そのグラフィックのいくつかの側面は失われているか、あるいは誤った状態となります。この動作は、*PDF_load_graphics()* の *errorconditions* オプションで変更できます。そのサブオプション群は、無視されるのでなくエラーを発生させる条件を指定します。たとえば、以下のオプションリストを用いると、*PDF_load_graphics()* は、SVG グラフィックがアニメーションエレメントまたはスクリプティングエレメントを含む場合には失敗します。例：

```
errorconditions = {elements={animate script}}
```

一般的制限　以下の制限が、多くのエレメントに影響を与えます：

- 外部 URL への参照は解決されません（画像・フォントなど）

対応していない SVG エレメント　以下の SVG エレメントは、対応されておらず、無視されます：

- アニメーションとスクリプティングのためのエレメント群：

  `animate, animateColor, animateMotion, animateTransform, script, mpath, set`

- SVG フィルタのためのエレメント群：

  `feBlend, feColorMatrix, feComponentTransfer, feComposite, feConvolveMatrix, feDiffuseLighting, feDisplacementMap, feDistantLight, feFlood, feFuncA, feFuncB, feFuncG, feFuncR, feGaussianBlur, feImage, feMerge, feMergeNode, feMorphology, feOffset, fePointLight, feSpecularLighting, feSpotLight, feTile, feTurbulence, filter`

- その他のエレメント群：

  `cursor, foreignObject, vkern`

制約のある SVG エレメント・属性・プロパティ　以下の属性とプロパティには制約があります：

- いくつかの CSS 規則に対応していません。例：*@import*・*@font-face*
- フォント選択プロパティ *font-variant* には、キーワード *small-caps* でのみ、かつ OpenType 機能 *smcp* を含むフォントに対してのみ対応しています。
- テキスト体裁プロパティ *text-decoration* の複数の値の組み合わせには対応していません。PDFlib は文字飾り要素を、別個の塗り・描線色を持つ領域として描くのではなく、文字飾り要素を直線として描きます。この直線は、塗り色があればそれで、なければ描線色で描かれます。
- *textPath* エレメントに対する *rotate* 属性には対応していません。
- *use* エレメントの属性 *xlink:href* はローカルの IRI 参照にのみ対応しています。たとえば *use* エレメント内における外部 SVG ファイルへの参照には対応していません。
- プロパティ *unicode-bidi* には、双方向テキストレイアウトのために必要なテーブルを含む TrueType/OpenType フォントについてのみ従います。PDFlib は、*PDF_fit_textline()* のオプションリスト内で、オプション *shaping* と *script=_auto* を設定します。
- プロパティ *glyph-orientation-vertical* は SVG 1.1 に従って対応していますが、ただし角度 180°・270° には対応していません。
- *view* エレメントに対する属性 *preserveAspectRatio* は無視されます。
- フォーム XObject（テンプレート）が、オプション *templateoptions* を用いて作成されている場合には、*svg* エレメントのプロパティ *overflow* は値 *visible* に対応しません。

対応していない SVG プロパティ　以下の SVG プロパティは、対応されておらず、無視されます：

`alignment-baseline, color-interpolation, color-interpolation-filters, color-rendering, cursor, dominant-baseline, enable-background, filter, flood-color, flood-opacity, glyph-orientation-horizontal, image-rendering, lighting-color, pointer-events, shape-rendering, text-rendering`

対応している SVG エレメントの対応していない属性　対応している SVG エレメントの、以下の属性は、対応されておらず、無視されます：

baseProfile (svg)
contentScriptType (svg)
contentStyleType (svg)
externalResourcesRequired (すべてのエレメント)
method (textPath)
on* (すべてのエレメント)
spacing (textPath)
縦書きテキストに対するtextLength・lengthAdjust (text・textpath・tref・tspan)
version (svg)
zoomAndPan (svg)
xlink:role (すべてのエレメント)
xlink:show (すべてのエレメント)
xlink:type (すべてのエレメント)

8.3 PDF ページを PDI で取り込む

注 この節で解説するすべての関数は、PDFlib+PDI か、または PDFlib Personalization Server PPS（これは PDI を含んでいます）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 のすべてのバイナリ版に内蔵されていますが、それを利用するには PDFlib+PDI か PPS のためのライセンスキーが必要です。

8.3.1 PDI の機能と用途

PDI（PDF 取り込みライブラリ）が得られる場合には、既存 PDF 文書内のページを取り込むことができます。PDI は、既存 PDF 文書内のページを PDFlib で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、取り込まれたラスタ画像と同様に扱われます : 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ます。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・斜形化させることもできます。取り込んだページは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 のファイルを作成する。
- 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クエリ（詳しくは pCOS パスリファレンスを参照）。

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（367 ページ「13 章 PPS と PDFlib Block Plugin」参照）。

8.3.2 PDFlib+PDI を使用

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、その照応する PDFlib 関数で生成することができます。

以下のアイテムを取り込むこともできます :

- 構造エレメントタグを取り込めます(詳しくは 216 ページ「タグ付き PDF 文書内のページを取り込む」を参照)
- レイヤ定義を取り込めます(詳しくは 216 ページ「レイヤーを持つ PDF ページを取り込む」を参照)
- (PPS のみ)PDFlib ブロックを、*PDF_process_pdi()* とオプション *action=copyallblocks* または *copyblock* で取り込めます(411 ページ「13.9.2 PDFlib ブロックを取り込む」参照)。

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 内で構成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 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 ページを取り込むためのコード断片 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。以下のコードスニペットは、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, pagecount, 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());

/* 入力文書内のページ数。全ページを取り込むのに有用 */
pagecount = (int) p.pcos_get_number(doc, "length:pages");

page = p.open_pdi_page(doc, pageno, "");
if (page == -1)
        throw new Exception("エラー: " + p.get_errmsg());

/* ダミーのページ寸法。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付け・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 218 ページ「8.4 画像・グラフィック・取り込み PDF ページを配置」で解説しています。

8.3.3 文書・ページ関連のチェック

文書関連のチェック　PDFlib+PDI は、Acrobat で開くことのできるすべての種類の PDF 文書を、PDF バージョン番号や、そのファイル内で使用されている機能にかかわらず、適切に処理します。

PDFlib+PDI は、特定の種類の破損した文書でも開くことができるよう、破損 PDF のための修復モードを実装しています。ただし、まれに、PDF 文書が、あるいは文書の特定のページが、PDI によって拒絶されることもあります。

PDF文書またはページがうまく取り込めないときは、*PDF_open_pdi_document()* と *PDF_open_pdi_page()* はエラーコードを返します。失敗に関してもっと詳しく知る必要がある場合には、その理由を *PDF_get_errmsg()* でクエリすることができます。あるいは、*errorpolicy* オプションを *exception* に設定することによって、文書を開くことができなかったときには例外が発生するようにすることもできます。

ページ関連のチェック　*PDF_open_pdi_page()* 内で以下のチェックが行われます：

- 生成されつつある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文書の中のページは、取り込めません。その理由は、より高いバージョン番号を持つ PDF が取り込まれた後には、その出力が本当に、要求された PDF バージョンに準拠しているかどうか、PDFlib は確信が持てなくなるからです。解決策：出力 PDF のバージョンを、*PDF_begin_document()* 内の *compatibility* オプションを用いて、必要なレベルに設定します。

  PDF 1.7ext 3（Acrobat 9）と *PDF 1.7ext8*（Acrobat X/XI）の文書は、PDI に関する限り、PDF 1.7 と互換です。

  PDF/A モードでは、入力 PDF バージョン番号は意味を持ちません。なぜなら PDF バージョンヘッダは PDF/A では無視される必要があるからです。

  文書が、より古い PDF バージョンに準拠していることがわかっているにもかかわらず、より高い PDF バージョンヘッダを使用している場合には、*PDF_open_pdi_document()* の *ignorepdfversion* オプションを用いることができます。
- PDF/A・PDF/X・PDF/VT・PDF/UA　の文書が、その照応する、カレント出力文書の PDF/A・PDF/X・PDF/VT・PDF/UA ステータスと非互換の場合には、拒否されます。詳しくは以下の項を参照してください：
  ▸337 ページ「12.3.7 PDF/A 文書を PDI で取り込み」。
  ▸349 ページ「12.4.6 PDF/X 文書を PDI で取り込む」。
  ▸358 ページ「12.5.7 PDF/X・PDF/VT 文書を PDI で取り込む」。
  ▸364 ページ「12.6.4 PDF/UA 文書を PDI で取り込む」。
- 文書が、矛盾した PDF/A または PDF/X 出力インテントを含んでいる場合には、ページは一切取り込まれません。

8.3.4 取り込んだ PDF 文書の具体的特徴

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。68 ページ「3.2.2 ページ寸法」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コピーするよう指示し、その結果、すべてのページ寸法情報が転写されます。

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 エントリ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寸法を決めるよう PDI に指示することもできます。

そして、*PDF_open_pdi_page( )* の *transform* オプションを用いて、取り込みページに対して、拡大・縮小や回転など、任意の変換を適用することもできます。

色の扱い　PDFlib+PDI は、取り込んだ PDF 文書の色を一切変更しません。たとえば、PDF が ICC カラープロファイルを含んでいる場合、これは出力文書内に保持されます。ページが透過グループエントリを含んでいる場合、それは、生成されるフォーム XObject へ複製されます。ただし、*PDF_open_pdi_page( )* の *transparencygroup* オプションが別の扱いを要請している場合を除きます。必要に応じ、取り込んだページに対して然るべき透過グループが作成されます。その際には、PDF/A・PDF/X・PDF/VT のあてはまる要請がすべて考慮されます。

タグ付き PDF 文書内のページを取り込む　デフォルトでは、タグは、入力文書と出力文書の両方がタグ付けされていれば取り込まれます。ただし、タグ取り込みは、*PDF_open_pdi_document( )* と *PDF_open_pdi_page( )* の *usetags* オプションで無効にすることもできます。詳しくは 320 ページ「11.4.5 タグ付き PDF ページを PDI で取り込む」を参照してください。

レイヤーを持つ PDF ページを取り込む　PDFlib はつねに、ページ上のすべてのレイヤー（技術的には「オプショナル内容」といいます）の内容を取り込みます。レイヤーの可視ステートを含むレイヤー定義も、そのレイヤーが、取り込んだページ群のうちのいずれかで使用されていれば、取り込まれます。ただし、レイヤー定義の取り込みは、*PDF_open_pdi_document( )* の *uselayers* オプションで無効にすることもできます。*uselayers=false* で作業をするためには、生成される文書はレイヤーを一切含んでいてはいけません。すなわち、レイヤーを持った PDF 文書はすべて *uselayers= false* を用いて開く必要があり、かつ、*PDF_define_layer( )* を呼び出してはいけません。

取り込んだレイヤー群の整頓をさらに制御するには、取り込んだレイヤー群に付加されるレイヤーリストの中に階層構造的なタイトルレイヤーを作成する *PDF_open_pdi_document( )* の *parenttitle* を用いることもできます（たとえばファイル名を与えるために）。*parentlayer* オプションは、同様に動作しますが、ただしユーザー定義レイヤーのハンドルをとります。

地理参照付き PDF を取り込む　地理参照付き PDF を PDI で取り込む際には、地理空間情報は、それが以下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、それが以下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。

複数の取り込み文書をまたぐ最適化　PDFlib 自体は、高度に最適化した PDF 出力を作成しますが、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、場合によっては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。

暗号化された PDF 文書と「シュラッグ」機能　暗号化文書内（すなわち、権限設定またはパスワードを持ったファイル）のページを取り込むためには、その照応するマスターパスワードを与える必要があります。暗号化された PDF 文書でパスワードがないものは、デフォルトでは拒絶されます。ただしこれは、*PDF_open_pdi_document()* の *infomode* オプションを *true* に設定することにより、pCOS で情報をクエリするために（ページを取り込むためではなく）開くことはできます。（この *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文書は、情報モードでも開くことができません）。

「シュラッグ」機能を利用すると、保護された文書の中のページを、その文書の作成者の権利を尊重する責任をユーザーが受け入れることを前提として、マスターパスワードなしで取り込むことができます。**「シュラッグ」**機能を利用することにより、ユーザーは、彼または彼女がいかなる文書作成者の権利をも侵害していないことを宣明します。PDFlib GmbH の契約条件は、ユーザーが文書作成者の権利を尊重することを求めています。

以下のすべての条件が真であるときに、**「シュラッグ」**機能は有効になります：

- *PDF_open_pdi_document()* に *shrug* オプションが与えられている。
- その文書はマスターパスワードを必要としているが、それが *PDF_open_pdi_document()* に与えられていない。
- その文書がユーザー（開く）パスワードを必要としている場合には、それが *PDF_open_pdi_document()* に与えられている必要があります。

「シュラッグ」機能には以下の効果があります：

- マスターパスワードが与えられていなくてもページを取り込むことができる。
- pCOS 擬似オブジェクト *shrug* が *true*/1 に設定される。
- pCOS が完全モードで動作する（限定モードではなく）。すなわち、*pcosmode* 擬似オブジェクトが 2 に設定される。

8.4 画像・グラフィック・取り込み PDF ページを配置

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、グラフィックを配置するための関数 *PDF_fit_graphics()* と、取り込み PDF ページを配置するための関数 *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。

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グラフィックでも取り込み 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_load_graphics()* 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繰り返しません。

クックブック 画像・グラフィック・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・graphics・pdf_import カテゴリにあります。

8.4.1 単純にオブジェクトを配置

画像を参照点に位置付け デフォルトでは、オブジェクトはその元の寸法で、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*(o, o)*に配置してみましょう：

```
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。

画像を拡縮して配置 コードを以下のように変えると、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");
```

このコード断片は、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* 方向に拡縮倍率 0.5 で拡縮されるので、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。

8.4.2 オブジェクトを点上か線上か枠内に配置

オブジェクトを位置付けるために、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。以下の図内の灰色の枠と線は、枠の大きさを視覚化するために描き足してあるだけで、実際の出力にはありません。

オブジェクトを枠内に配置することは、*fitmethod=nofit* の場合には意味がありません。なぜならこの場合、そのオブジェクトは位置付けられるだけであり、拡縮は行われないからです。*boxsize* オプションを用いて、オブジェクト配置のための横線か縦線、あるいは実際の枠を指定することができます：

```
boxsize={100 0}          横線
boxsize={0 100}          縦線
boxsize={100 200}        枠
```

以下のさまざまな例では、オブジェクトを枠内に、さまざまなはめ込み方式ではめ込んでいきましょう。

枠内へ自動はめ込み　*fitmethod=auto* を用いると、PDFlib は画像を、枠内にはめ込めるように、ただしそれをつぶさずに拡縮します：もしそれがその枠内にはめ込めた場合には、拡縮は一切行われません。そうでない場合には、幅と高さの縦横比を保ちながら、寸法が縮小されます。

図 8.2a・図 8.2b・図 8.2c では、はめ込み枠の寸法が最初 *boxsize={70 45}*、次に *boxsize={70 30}*、さらに *boxsize={30 30}* と減少していくにつれて、PDFlib が画像寸法を縮小していくさまを演示しています。

図 8.2　さまざまなはめ込み方式に従って画像を枠内にはめ込む

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 fitmethod=auto （拡縮は必要でない）
b) 	boxsize={70 30} position=center fitmethod=auto （枠の高さが小さくなったのに合わせて縮小される）
c) 	boxsize={30 30} position=center fitmethod=auto （枠の高さと幅が小さくなったのに合わせて縮小される）
d) 	boxsize={70 45} position=center fitmethod=meet
e) 	boxsize={35 45} position=center fitmethod=meet
f) 	boxsize={70 45} position=center fitmethod=entire
g) 	boxsize={30 30} position=center fitmethod=clip
h) 	boxsize={30 30} position={right top} fitmethod=clip

画像を枠の中央へはめ込み　あらかじめ定義した長方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 8.2d 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 8.2e 参照）。

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、枠内に、できるだけ大きく配置されます。

画像を枠全体へはめ込み　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おそれがあるので、有用な場合はまれでしょう（図 8.2f 参照）。

画像を枠へ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位置付けてみましょう（図 8.2g 参照）。

画像を枠の中央に位置付けることによって、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}* で位置付ければよいでしょう（図 8.2h 参照）。

8.4.3 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 *(0, 0)* 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 8.5a 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へはめ込み　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 8.5b 参照）。東に向けた画像を図 8.5c に、南向きを図 8.5d に示します。

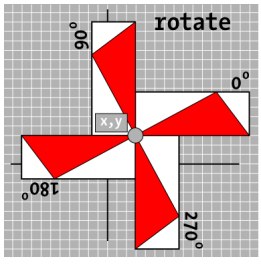
orientate オプションは、図 8.4 に示すとおり、向きのキーワードとして *north*・*east*・*west*・*south* に対応しています。

なお、*orientate* オプションは、座標系全体には一切影響せずに、配置するオブジェクトにだけ影響を及ぼします。

図 8.5 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`[1]

図 8.3
rotate オプション



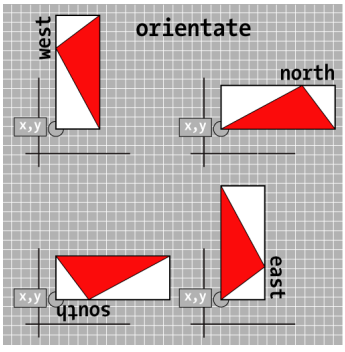


図 8.4
orientate オプション

生成される出力	PDF_fit_image に与えるオプションリスト
b) 	`boxsize={70 45} orientate=west fitmethod=meet`
c) 	`boxsize={70 45} orientate=east fitmethod=meet`
d) 	`boxsize={70 45} orientate=south fitmethod=meet`
e) 	`boxsize={70 45} position={center bottom} orientate=east fitmethod=clip`

1. デフォルト `fitmethod=nofit` ですので、この `boxsize` オプションは実際には必要ありません。

向きを変えた画像を枠へ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付けましょう（*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 8.5e 参照）。

8.4.4 オブジェクトを回転

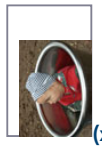
rotate オプションは、参照点を中心に座標系を回転させることによって、オブジェクトを回転させます。結果として、はめ込み枠も回転されます。図 8.3 に、*rotate* オプションの一般的な動作を図示します。

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 8.6a 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 8.6b 参照）。

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 8.6c 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 8.6d 参照）。

図 8.6 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`[1]
b) 	`boxsize={70 45} rotate=30`[1]
c) 	`boxsize={70 45} rotate=90 fitmethod=meet`
d) 	`boxsize={70 45} rotate=30 fitmethod=meet`

1. デフォルト `fitmethod=nofit` ですので、この `boxsize` オプションは実際には必要ありません。

8.4.5 ページ寸法の調整

ページ寸法を画像に合わせる 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寸法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。ページ寸法を大きめにすると（図 8.7 参照）、画像のまわりにふちが残ります。ページ寸法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、ページ寸法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");
```

次のコード断片は、ページ寸法を *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作成されます：

図 8.7
ページ寸法の調整。左から、ちょうど・大きめ・小さめ

```
p.fit_image(image, 40, 40, "adjustpage");
```

次のコード断片は、ページ寸法を *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長方形の領域です。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 ページのページ枠群を転写　取り込み PDF ページのすべてのページ枠（MediaBox・CropBox 等）を、カレント出力ページへコピー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、ページ枠エントリ群を転写 */
inpage = p.open_pdi_page(indoc, 1, "cloneboxes");
...
/* 出力ページをダミーページ寸法で開く */
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転写。これは、begin_page_ext()で用いた
* ダミー寸法をオーバライドします。
*/
p.fit_pdi_page(inpage, 0, 0, "cloneboxes");
```

この技法を活用すると、生成 PDF のページ寸法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。

8.4.6 配置された画像と PDF ページに関する情報をクエリ

配置された画像とテンプレートに関する情報　*PDF_info_image()* 関数を用いて、画像・テンプレート情報をクエリすることができます。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）。

以下のコード断片は、ピクセル寸法と、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寸法の両方を取得します：

```
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("画像寸法（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("画像寸法（ポイント単位）: " + width + " x " + height);
```

配置された PDF ページに関する情報　*PDF_info_pdi_page()* 関数を用いて、配置 PDF ページに関する情報をクエリすることができます。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）。

以下のコード断片は、取り込みページの元の寸法と、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寸法の両方を取得します：

```
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("元のページ寸法: " + pagewidth + " x " + pageheight);

width  = p.info_pdi_page(page, "width", optlist);
height = p.info_pdi_page(page, "height", optlist);
System.err.println("配置ページ寸法: " + width + " x " + height);
```

9 テキストと表の組版

9.1 テキスト行を配置・はめ込む

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、218 ページ「8.4 画像・グラフィック・取り込み PDF ページを配置」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付けを決定します：この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。このテキスト枠を変更するには *matchbox* オプションを用います。

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・fitmethod・position・rotate*）で指定される長方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。**

9.1.1 単純なテキスト行配置

テキストを参照点に位置付け　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。以下のコード断片は、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 9.1 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。

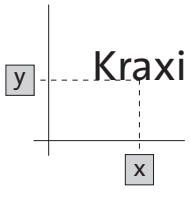


図 9.1
テキストの
中央揃え

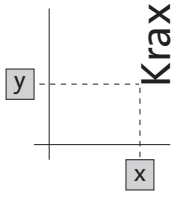


図 9.2
西向きの
単純なテキスト

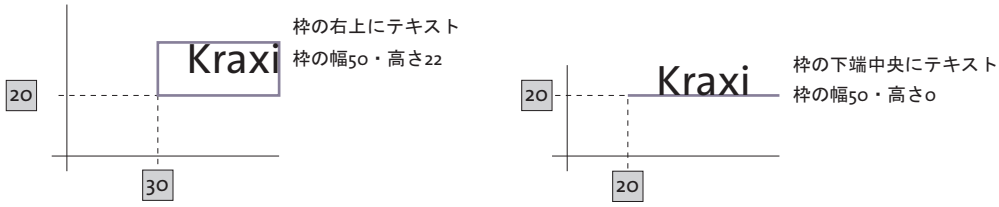


図 9.3 テキストを枠内に位置付け

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。以下のコード断片は、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 9.2 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。

9.1.2 テキストを枠内に位置付け

テキストを位置付けるには、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、この枠に対して相対的に位置付けることが可能です。図 9.3 に、その一般的なはたらきを図示します。

テキストを枠内に位置付け　長方形の枠を定義して、この枠内の右上にテキストを配置しましょう。以下のコード断片は、幅 50 単位・高さ 22 単位の枠を、参照点 *(30, 20)* に定義します。図 9.4a 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 9.4b 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 9.4c 参照）。

なお、図中の枠や線は、枠の寸法を視覚化するために描いたものであり、実際の出力には現れません。

図 9.4 さまざまな位置付け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　テキストの位置付けを、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 9.4d 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位置付けましょう。この場合を図 9.4e に示します。

9.1.3 テキストを枠へはめ込み

この項では、さまざまなはめ込み方式を用いて、テキストを枠へはめ込みましょう。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：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 9.5a 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 9.5b 参照）。

テキストを小さな枠内へ縦横比を保ってはめ込み　それでは、テキスト全体を枠の中へ、縦横比を保ってはめ込んでみましょう。これはオプション *fitmethod=auto* で実現できます。図 9.5c 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、変わらずに枠内に配置されています。

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 9.5d は、テキストが元の長さの 75 パーセントに縮んだ様子を示します。

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。これは、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 9.5e です。

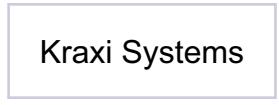
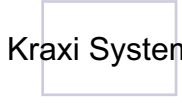
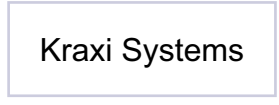
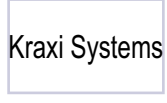
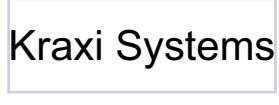
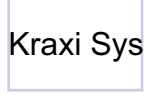
テキストの文字サイズを上げて枠へはめ込み　テキストを枠の幅（または高さ）いっぱいに拡げて、ただし縦横比は保ったまま、はめ込みたい時があるかもしれません。テキストより大きい枠に対して *fitmethod=meet* を用いると、テキストの幅が枠の幅と一致するまでテキストが大きくなります。この場合を図示したのが図 9.5f です。

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 9.5g 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 9.5h 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 9.5 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へはめ込む

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位置付けると、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 9.6a 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（272 ページ「9.4 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 9.6b 参照）。図 9.6c では、テキストが *xheight* に基づいて縦中央揃えされるように、その照応する高さの範囲枠を定義しています。

図 9.6d ～ 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の中で中央揃えされたテキストを示します。

図 9.6 さまざまな範囲枠高さに従ってテキストを枠へはめ込む

生成される出力		PDF_fit_textline( ) に与えるオプションリスト
a)	Kraxi Systems	`boxsize={80 20} position=center fitmethod=auto`
b)	Kraxi Systems	`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={capheight none} fillcolor=mistyrose}`
c)	Kraxi Systems	`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={xheight none} fillcolor=mistyrose}`
d)	Kraxi Systems	`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={ascender none} fillcolor=mistyrose}`
e)	Kraxi Systems	`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={ascender descender} fillcolor=mistyrose}`
f)	Kraxi Systems	`boxsize={80 20} position=center fitmethod=auto matchbox={boxheight={fontsize none} fillcolor=mistyrose}`

9.1.4 テキストを文字で揃える

テキストを文字で揃える　テキストを、ある特定の文字で揃え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 9.7a に示すように、テキストははめ込み枠の中央に位置付けられます。*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 9.7b 参照）。一般に、揃え文字はその右下隅が参照点に配置されます。

図 9.7 テキスト行を点で揃える

生成される出力		PDF_fit_textline( ) に与えるオプションリスト
a)	127.123 12.01 123.0 4025.20	`boxsize={70 8} position={center bottom} alignchar=.`
b)	127.123 12.01 123.0 4025.20	`boxsize={70 8} position={left bottom} alignchar=.`

9.1.5 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。*fitbox* オプションは無視されます。図 9.8 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 9.8 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む

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

9.1.6 リーダを用いる

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、目次の各エントリとその照応する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 9.9a 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 textflow/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効果を作成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 9.96b 参照）。

図 9.9 リーダを用いたテキスト行をはめ込む

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={+ }}

9.1.7 パス上テキスト

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のハンドルを取得することによって作成できます。以下のコード断片は、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 9.10 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 strokecolor=dodgerblue");
```

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。

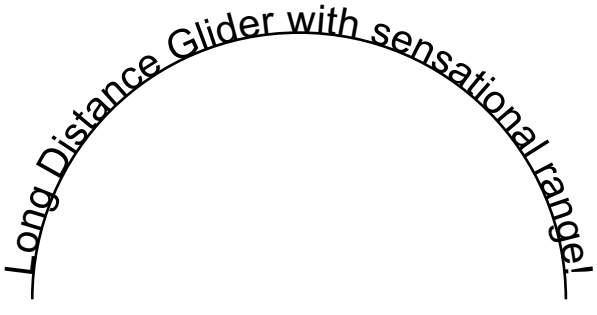


図 9.10
パス上テキスト

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にアキを作る　デフォルトでは、PDFlib は個々のキャラクタをパス上に直接配置します。すなわち、グリフとパスの間にはアキは全くありません。パスとテキストの間にアキを作りたいときは、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実現できます。以下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 9.11 参照）：

```
p.fit_textline("Long Distance Glider with sensational range!", x, y,
    "textpath={path=" + path + "} position={center bottom} " +
    "matchbox={boxheight={capheight descender}}");
```

9.1.8 影付きテキスト

shadow オプションを用いると、テキストの影付き効果を生み出すことができます。影の色と、メインテキストからの横・縦距離を、サブオプション内で指定できます：

```
p.fit_textline("Long Distance Glider", x, y,
    "fillcolor=rosybrown shadow={offset={3, -3}}");
```

9.1.9 Acrobat で編集できる透かし

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/watermark トピックにあります。

テキスト行機能を使うと、テキストに対してさまざまな組版オプションを適用することができます。このテキストは、ページに組み込まれた一部分となり、最終 PDF 文書内で簡

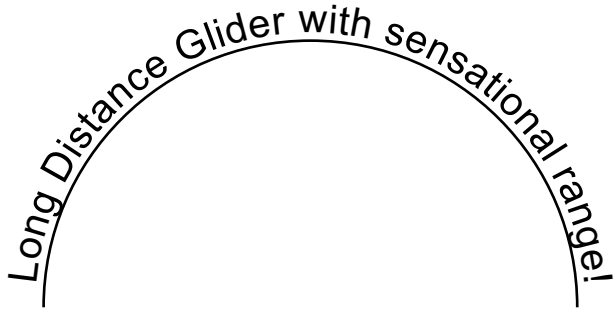


図 9.11
パス上テキストで、パスとテキストの間をあけた

単に編集することはできません。しかし、Acrobat は「透かし」機能を提供しています。この機能を利用すると、Adobe Acrobat で後で変更や削除できるページ内容を作成できます（無料の Reader では不可）。ただし、この機能は PDF 標準 ISO 32000 に含まれているものではなく、Acrobat に実装されている独自拡張です。ですのでこの機能は、すべての標準準拠 PDF ビューアで動作することが保証されているものではありません。

Acrobat の透かしは、1 行ないし複数のテキスト行か、1 つの画像か、1 つの PDF ページを、1 つのテンプレート（フォーム XObject）内にラップして、その透かしの組版特性群の XML 記述を加えることによって動作します。既存の透かしを Acrobat で編集するには以下のように操作します：

- Acrobat DC：**「ツール」→「*PDF* を編集」→「透かし」→「更新 ...」**または**「削除 ...」**をクリック。
- Acrobat XI：**「ツール」→「ページ」→「ページデザインの編集」→「透かし」→「更新 ...」**または**「削除 ...」**をクリック。

すると**「透かしを更新」**ダイアログが開きますので（図 9.12 参照）、そこで透かしのテキストや、その配置・体裁や、対象とするページ範囲を指定できます。このテキストの体裁は、このダイアログを用いた後に変わる可能性もあります。なぜなら、このダイアログは、文字間隔や下線、テキスト表現モードなど、あらゆるテキスト組版オプションに対応しているわけではないからです。

この透かしは、画面表示上で、または印刷ページ上で、あるいは両方で印字されるよう指定することが可能です（この Acrobat のダイアログの中の**「表示方法オプション ...」**）。これは、適切な画面・印刷設定を施された**「透かし」**というレイヤーを通じて実現されます。ただし PDF/A・PDF/X・PDF/UA においては、レイヤーとレイヤーオプションにはさまざまな制約が課せられています。

PDFlib は、そのテンプレート機能を拡張することによってこの透かし機能に対応しています。*PDF_begin_template_ext()* を用いてテンプレートを作成する際に、その *watermark* オプションを用いてこれに標識することが可能です。その場合、このテンプレートは、

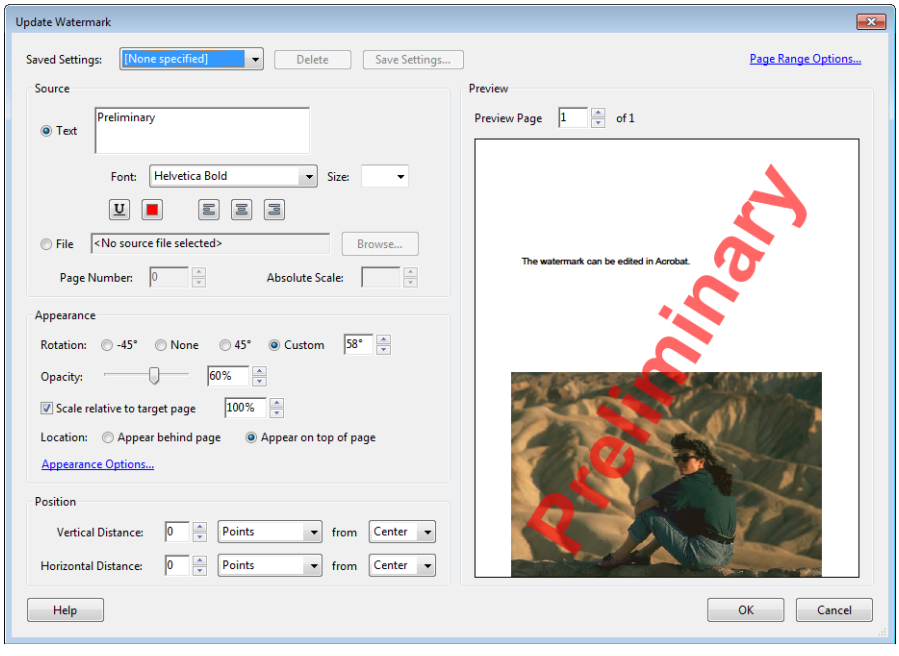


図 9.12
Acrobat の透かしダイアログ

PDF_fit_textline() を用いて作成されたただ 1 行のテキストのみを内容としている必要があります。そうして生成された透かしは、自動的にすべてのページに加えられ（あるいは一部のページにも可）、後で Acrobat を用いて変更または削除することが可能です。

注 複数行・画像・PDF ページを内容とする透かしには現状対応していません。

以下のコード断片は、編集可能な透かしの定義を演示しています。この透かしは、この透かしを定義した後に作成されるすべてのページに自動的に追加されます。なお、指定している文字サイズは意味を持ちません。なぜなら、こうしてできたテンプレートはいずれにせよそのページ内に収められるからです：

```
p.begin_template_ext(0, 0, "watermark={location=ontop opacity=60%}");

p.fit_textline("Preliminary", 0, 0,
        "fontsize=12 fontname=Helvetica-Bold encoding=unicode fillcolor=red " +
        "boxsize={595 842} stamp=ll2ur");

p.end_template_ext(0, 0);
```

9.2 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 9.13・図 9.14 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。

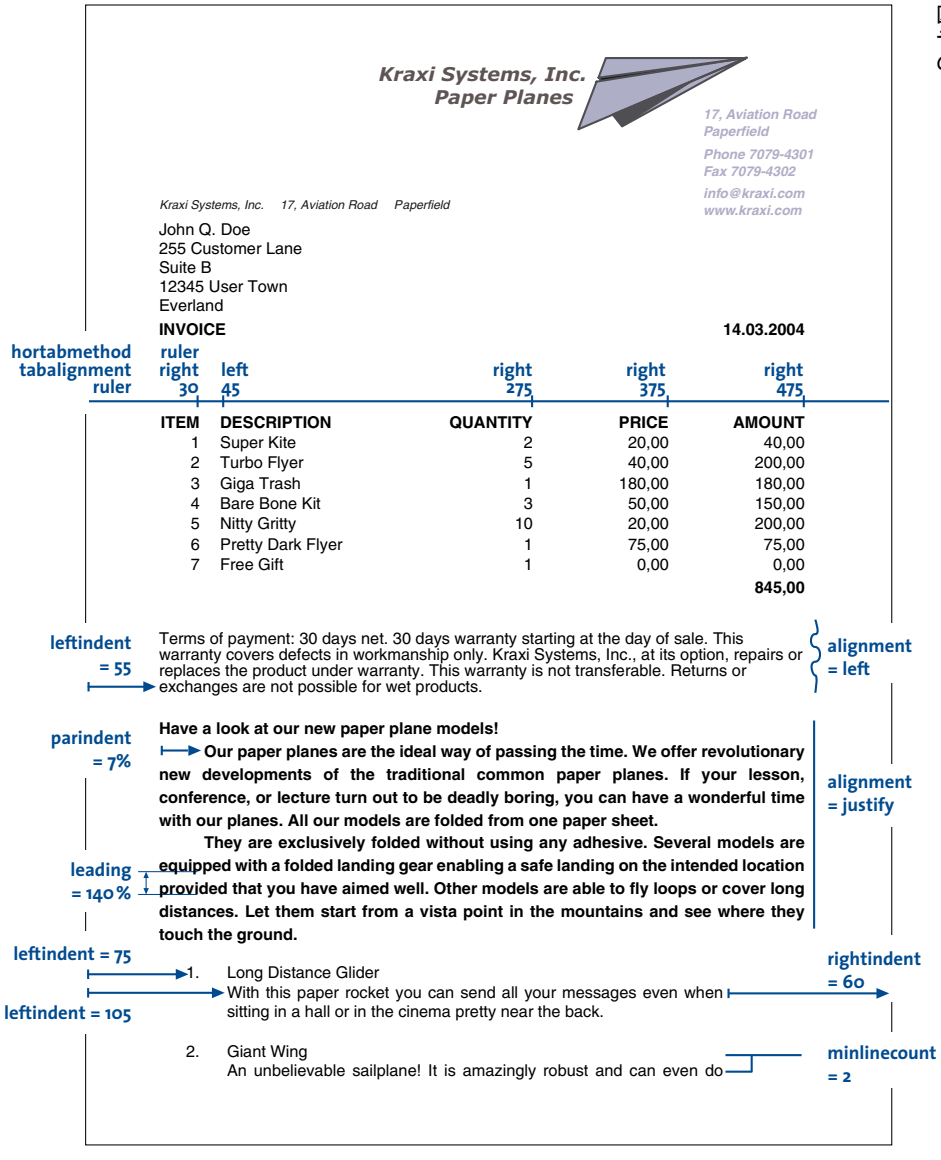


図 9.13
テキストフローの組版

複数行のテキストフローは、1 つの長方形（はめ込み枠という）内にも複数の長方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、その照応する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クエリ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 225 ページ「9.1 テキスト行を配置・はめ込む」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。**

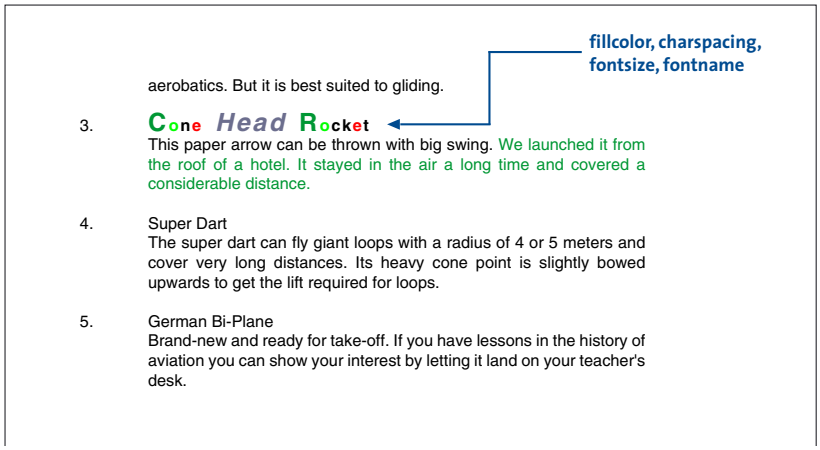


図 9.14
テキストフローの組版

9.2.1 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* 引数で指定される長方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。以下のコード断片は、*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。次のコード断片は、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演示しています（図 9.15 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 textflow/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"));

/* エラーかどうかをチェック */
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);
```

9.2.2 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以下のように拡張します（図 9.16 参照）：

図 9.15
テキストフローを２つのはめ込み枠に配置

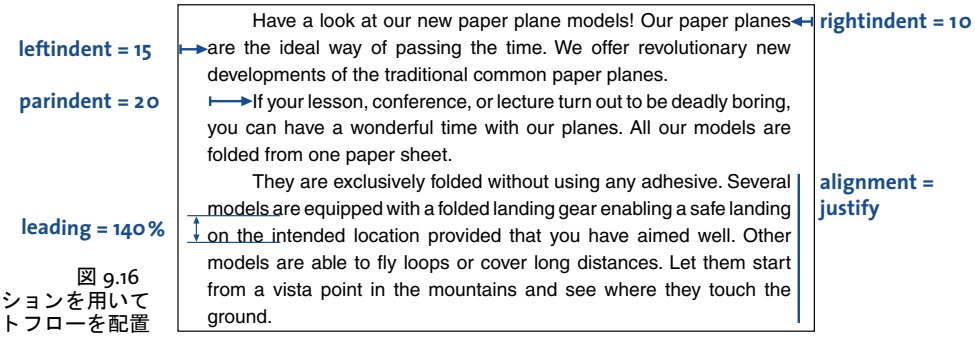


図 9.16
オプションを用いて
テキストフローを配置

```
String  optlist =
        "leftindent=15 rightindent=10 parindent=20 alignment=justify " +
        "leading=140% fontname=Helvetica fontsize=8 encoding=unicode";
```

9.2.3 インラインオプションリストとマクロ

図 9.16 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。改段落キャラクタを矢印で視覚化します。

```
‹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、文字参照（*<* 等）を回避策として用いることはできません。

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、照応するインラインオプション群を書かなくて済むようにするには、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 9.17 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。以下のコード断片は、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%
fontname=Helvetica fontsize=8 encoding=winansi}
```

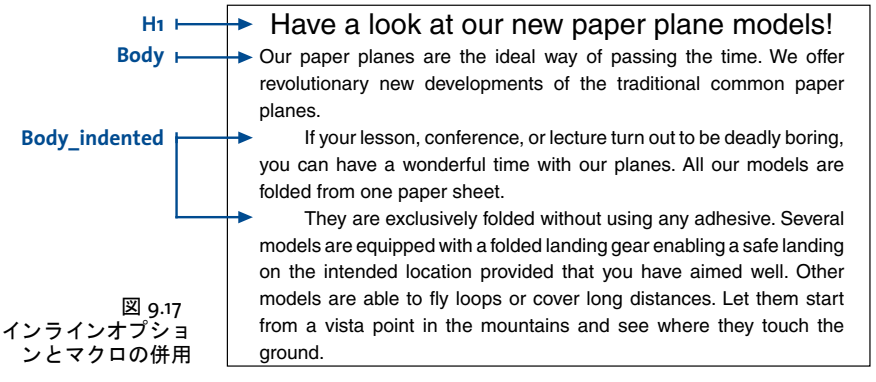


図 9.17
インラインオプションとマクロの併用

```
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ソース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ソースから来るでしょう。現実的には 2 通りの状況が考えられます：

- テキスト内容が外部ソースから来て、組版オプションはプログラム内にある場合：実行時に、外部ソース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ソースから来る場合：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ソースから来ます。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：テキスト内容は外部ソース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 9.18
テキストを
表として配置

9.2.4 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。表は以下のような複数行のテキストを持ち、個々のエントリ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 9.18 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 textflow/tabstops_in_text トピックにあります。

注 複雑な表を作成するには PDFlib の表機能を推奨します（256 ページ「9.3 表の組版」参照）。

9.2.5 番号付きリストと段落間隔

インラインオプション*leftindent*を用いて番号付きリストを組む方法を以下に示します（図 9.19 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.
```

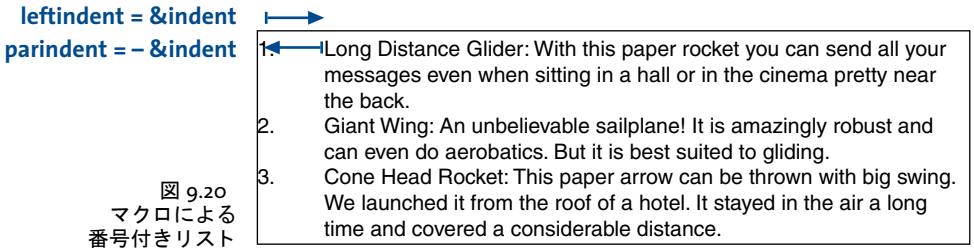


図 9.20
マクロによる
番号付きリスト

クックブック 箇条書きリストと番号付きリストの完全なコードサンプルがクックブックの `textflow/bulleted_list`・`textflow/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。以下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* に照応したタブ位置を指定しています。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 9.20 に *parindent*・*leftindent* オプションの作用を示します。

２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

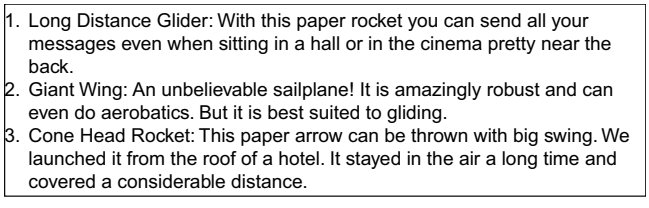


図 9.19
番号付きリスト

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。以下の例は、2 つの段落の間に 80% のアキを作ります（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 textflow/distance_between_paragraphs トピックにあります。

9.2.6 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、これはその照応する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 9.1 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。たとえば以下の並びは空行を作成します：

```
<nextline><nextline>
```

キャラクタかキャラクタ列をマッピング / 除去 *charmapping* オプションを使うと、テキスト内のキャラクタを、別のキャラクタへマップしたり、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へマップしてみましょう。これを行うには *charmapping* オプションを以下のようにします：

```
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。複数のマッピングを一度に行うには、以下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}
```

以下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}}
```

タブ 1 つにつきスペース 4 つに置換します：

```
charmapping={hortab {space 4}}
```

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}}
```

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}}
```

表 9.1 テキストフロー内の制御キャラクタとその意味

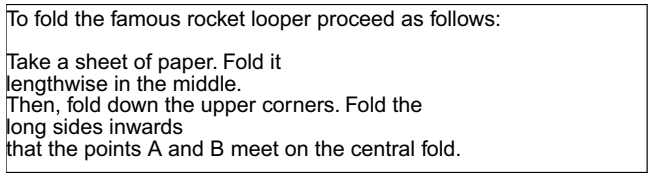
Unicode キャラクタ	実体名	等価なテキストフローオプション	テキストフロー内における意味
U+0020	SP, space	space	単語揃え・改行
U+00A0	NBSP, nbsp	（なし）	（no-break space）改行しない空白文字
U+202F	NNBSP, nnbsp	（なし）	（narrow no-break space）改行しない、かつ組版オプションに従ってその幅を変えることのない固定幅空白文字
U+0009	HT, hortab	（なし）	水平タブ：ruler・tabalignchar・tabalignment オプションに従って処理されます
U+002D	HY, 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	next-paragraph	（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。

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以下のようなものだとします：

```
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.
```

以下のコード断片は、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演示しています：

```
/* オプションリストを組み立て */
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());
```



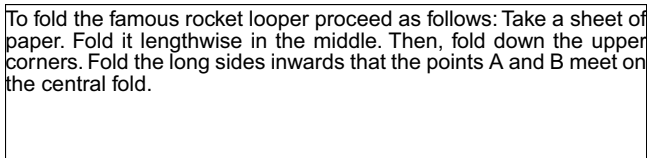


図 9.21
上：無駄な改行のあるテキスト

下：charmapping オプションで改行を置換したもの

```
result = p.fit_textflow(textflow, left_x, left_y, right_x, right_y, "");
if (!result.equals("_stop"))
        { /* ... */ }

p.delete_textflow(textflow);
```

図 9.21 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。

9.2.7 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、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。
- 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*）、文字実体を用いることができます：**

グリフ *U+00AD* がそのフォント内で得られる場合にはそれが、そうでない場合には *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　以下の例では、以下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（ここではダッシュで視覚化しています）：

```
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
```

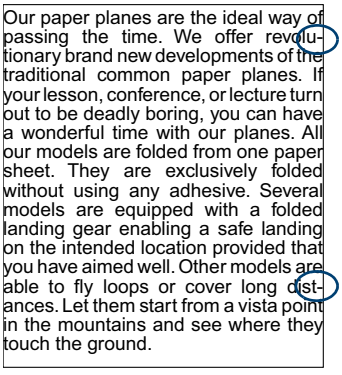


図 9.22
テキスト両端揃えでソフトハイフンあり。
デフォルト設定と広いはめ込み枠を使用。

図 9.23
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。

```
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 9.22 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：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 9.23 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以下のようになります：

```
fontname=Helvetica fontsize=9 encoding=winansi alignment=justify
```

9.2.8 ウィドー行・オーファン行

1 つの段落の先頭の 1 行（ないし複数行）だけが段またはページの下端に現れるとき、これをオーファンといいます。同様に、1 つの段落の末尾の 1 行（ないし複数行）が次の段またはページの先頭に現れるとき、これをウィドーといいます。高品質な文字組版において、孤立したオーファン行またはウィドー行は望ましくないと考えられています。

オーファン制御　テキストフローオプション *minlinecount* は、はめ込み枠の末尾段落の最少行数を指定します。行がそれよりも少ない場合には、それらは次のはめ込み枠の中に配置されます。値 *minlinecount=2* を用いると、はめ込み枠の末尾の段落の 1 行のオーファン行を避けることができます。

ウィドー制御　PDFlib は、将来のテキストフロー配置について何も知りませんので、ウィドー行を直接制御することはできません。しかし、クライアントコード内に以下の仕組みでウィドー制御を実装することは可能です：

- テキストフローの先頭部分を、先頭はめ込み枠内へブラインドモード（オプション *blind=true*）で、すなわち実際の出力を生成せずにはめ込みます。
- テキストフローの次の部分を、2 番目のはめ込み枠内へブラインドモードではめ込みます。*PDF_info_textflow()* をキーワード *firstparalinecount* とともに呼び出すことによって、この 2 番目のはめ込み枠の先頭段落の行数をクエリします。もしこの結果が 1 ならば、1 行ウィドーを見つけたことになります。
- そうであれば、先頭はめ込み枠へ *rewind* オプションを用いて戻ることができますので、アルゴリズムで、ウィドー行を避けるためにはめ込みオプション群を調整する必要が

あります。たとえばこれは、先頭はめ込み枠内の行数を *maxlines* オプションを用いて減らすことによって実現できます。

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

9.2.9 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 9.2 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、引用符のペアを数えていく仕組みになっています。

表 9.2 改行アルゴリズムを制御するためのオプション

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	以下の方式が順に適用されます：shrink・spread・nofit・split。
	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。	

表 9.2 改行アルゴリズムを制御するためのオプション

オプション	説明
charclass	（ペアのリスト。ここでペアの 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よって分類されることにより、そのキャラクタ（群）の改行時動作を決定します： ***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）。
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み出しません。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。

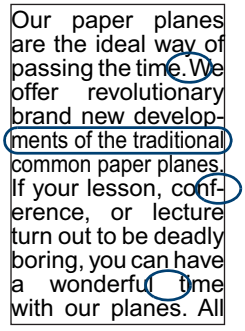


図 9.24 狭いはめ込み枠でテキスト両端揃え。デフォルト設定を用いています

改行を防止　*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止することができます。たとえば、以下のオプションはキャラクタ / の直後での改行を防止します：

```
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** `textflow/avoid_linebreaking` **トピックにあります。**

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、以下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作成されます：

```
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 9.24 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 9.24 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 9.25a 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 9.25b 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 9.25 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 brand new 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 9.25c 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。

9.2.10 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 7.2 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、以下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。以下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 7.3 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。

9.2.11 テキストをパス・画像に回り込ませる

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、明示的な長方形 / 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。

範囲枠を持つ画像にテキストを回り込ませる　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、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}");
```

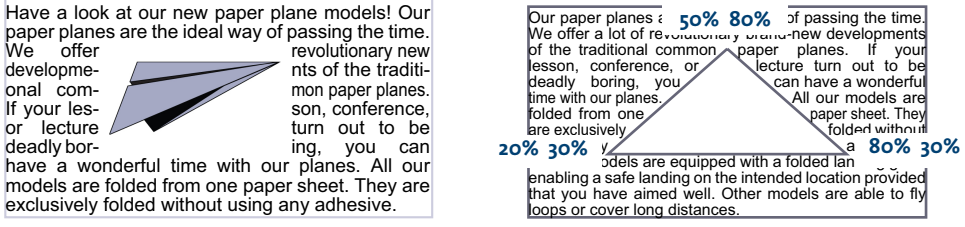
テキストフローを追加します。そしてそれを、以下のように、画像のはめ込み枠 *img* を回り込むべき領域として *wrap* オプションを使って配置しましょう（図 9.26 参照）：

```
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 `textflow/wrap_text_around_images` トピックにあります。**

図 9.26　範囲枠を持つ画像にテキストを回り込ませる



任意のパスにテキストを回り込ませる　パスオブジェクトを作成して（69 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。下記のコード断片は、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", "");

/* 望むならパスを視覚化 */
p.draw_path(path, x, y, "stroke");

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作るのではなく、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%} }" +
        "}}");
```

画像のクリッピングパスにテキストを回り込ませる　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。こうすれば、クリッピングパスが抽出される際と、画像が実際に配置される際に、必ず同一の変換（拡縮等）が適用されます。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。

非長方形形状にテキストを回り込ませる　パスオブジェクトを回りこみ輪郭として作成するのではなく、パス要素群を直接 Textflow オプション群で指定することもできます。

テキストは、範囲枠で指定される長方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。たとえば以下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 9.27 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。以下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　横でも縦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 `textflow/wrap_text_around_polygons` トピックにあります。

非長方形の輪郭へ流し込む　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

図 9.28
ひし形へテキストを流し込む

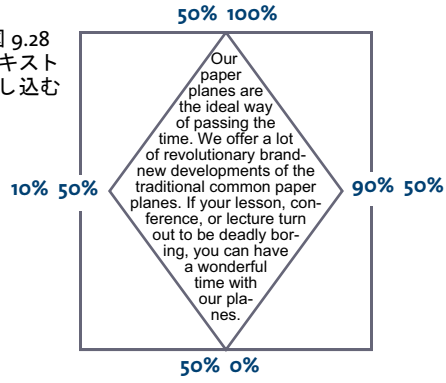
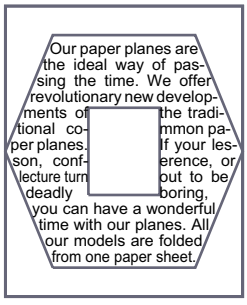


図 9.29
重なり合う輪郭へ流し込む



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、1 個ないし複数の輪郭の中にテキストが配置されます。以下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、はめ込み枠の長方形に対するパーセント値として与えています（図 9.28 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む　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 9.29 参照）：

```
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：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** `textflow/fill_polygons_with_text` **トピックにあります。**

9.3 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、一行か複数行のテキストか、画像か、SVG グラフィックか、PDF ページ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 table カテゴリにあります。

表の一般的特徴　表組版機能の説明は、以下の概念と用語にもとづきます（図 9.30 参照）：

- **表**は、長方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、表内の長方形の領域であり、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（266 ページ「9.3.5 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 1 個ないし複数の表行のかたまりであり、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。
- オプショナルなキャプション（図 9.30 では示されていません）は、表の説明を配置するために使える追加の要素です。これは表の任意の辺に配置することができます。

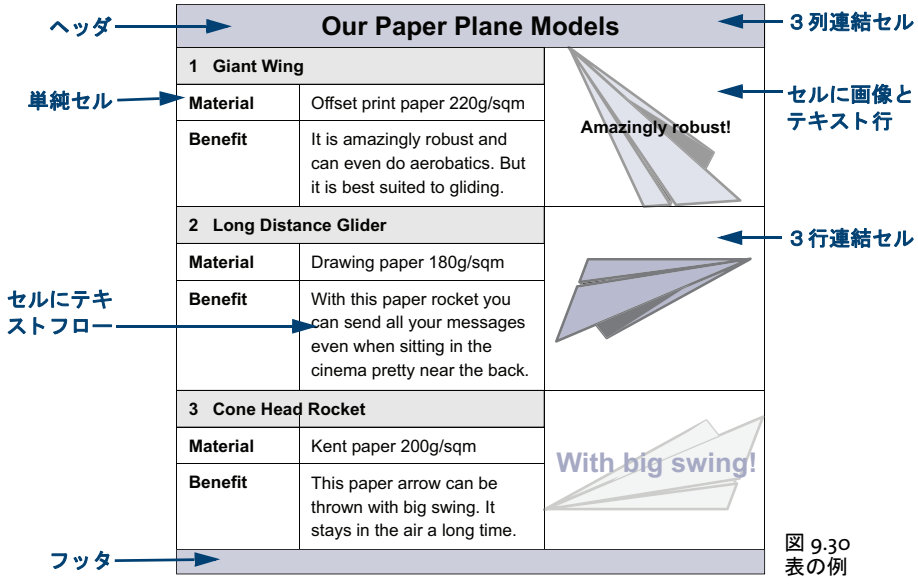


図 9.30
表の例

例として、図 9.30 の表を作成するすべての側面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 272 ページ「9.4 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、カレント図形ステータスからは独立です。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かパターン / テンプレート / グリフ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 `table/starter_table` トピックにあります。

表に関する組版上の問題を分析 セルの数と、表組版オプションによっては、PDFlib の表組版機能の結果が、自分の期待とは一致しないことがあります。ほとんどの場合において、これは適切なオプションによって矯正できます。しかし、問題の起きているセル群や、セルのグループを、誤ったオプションを用いて特定するのは、難しい場合もあります。表に関する組版上の問題のデバッグを実施可能にするため、PDFlib は、*PDF_fit_table()* に以下のオプションを提供しています：

- オプション *showcells* は、各セル内枠の境界を視覚化します。この関数がページスコープで呼ばれ、かつ PDF/A モードが有効でない場合には、各表セルの中央に、そのセル内容に関する詳細を持った注釈が配置されます。
- オプション *debugshow* が *true* の場合には、高すぎる、あるいは幅が広すぎる、あるいはセルが小さくなりすぎる表に関するエラーがすべて抑制され、ログ記録されます。結果として作成される表インスタンスは、その表は破損していますが、デバッグの助けとして生成されます。
- オプション *showgrid* が *true* の場合には、すべての列と表行の縦・横境界が描線されます。すなわち、基礎をなす表グリッドが視覚化されます。

9.3.1 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を黒い罫線で視覚化しましょう。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 `table/vertical_text_alignment` トピックにあります。

以下のコード断片は、この単純な表を作成する方法を示します。結果を図 9.31a 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;

/* エラー時には抜け出す */
p.set_option("errorpolicy=exception");

/* フォントを読み込む */
font = p.load_font("Helvetica", "unicode", "");

/* 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);

/* 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);

/* 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);

/* 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);

/* 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);

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);
```

```
/* 結果をチェック。「_stop」はすべてOKを意味します */
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ります */
        }
}
p.end_page_ext("");

/* これは、表内で使われたテキストフローハンドル群も一緒に削除します */
p.delete_table(tbl, "");
```

セルの内容の縦位置を調整　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、それらは、その辺からのまちまちの距離に位置付けられます。図 9.31a において、4 つのテキスト行セルは以下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 9.31 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 9.31b 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* オプションと照応します。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** `table/vertical_text_alignment` **トピックにあります。**

9.3.2 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 9.32 に示す要素があります。

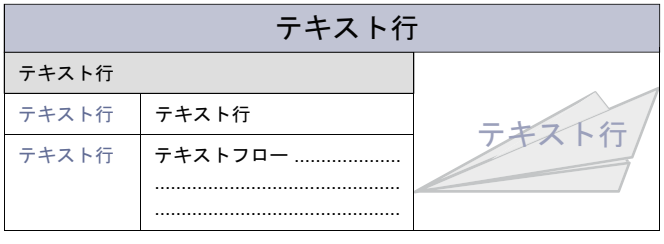


図 9.32
表セルのさまざまな内容

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* オプションリストで、*PDF_fit_textline( )* のすべての組版オプション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。

テキストに色をつけるには *fittextline* オプションリストで *fillcolor* オプションを使用してください。

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。これはすなわち：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。その画像または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 / テンプレートが、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。

ベクトルグラフィック　グラフィックは、*PDF_add_table_cell( )* を呼び出す前に *PDF_load_graphics( )* で読み込んでおく必要があります。そのグラフィックのハンドルは、*graphics* オプションで与えます。*fitgraphics* オプションで、*PDF_fit_graphics( )* 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。すなわちグラフィックが、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、グラフィック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 ページが、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）に照応する *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。セル枠が注釈長方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）に照応する *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長方形として用いられます。

セル内枠内でセル内容を位置付け　デフォルトでは、セル内容はセル枠に合わせて位置付けられます。*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。その結果できる長方形を、**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 9.33 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。263 ページ「9.3.4 さまざまな種類の内容を持った表」で説明します。

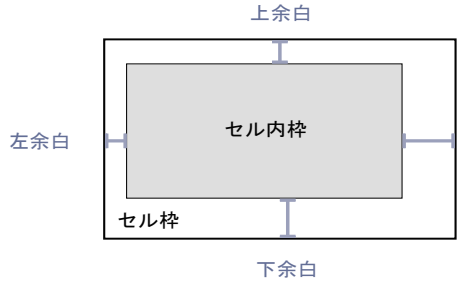


図 9.33
内容をセル内枠にはめ込み

9.3.3 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 9.34 に、3 列・4 表行の表の例を示します。

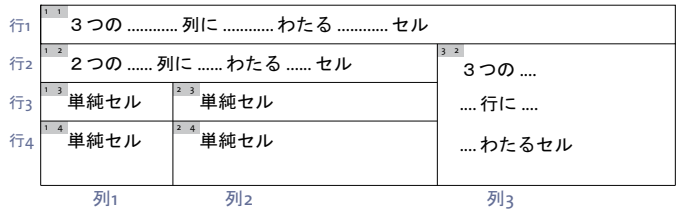


図 9.34
単純セルと、複数の表行や列を連結したセル

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。なお、与えた *colwidth* が効力を及ぼすのは最初の列に対してだけであり、*colspan* グループ内のすべての列に対してではないことに留意してください。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 9.35 に例を示します。

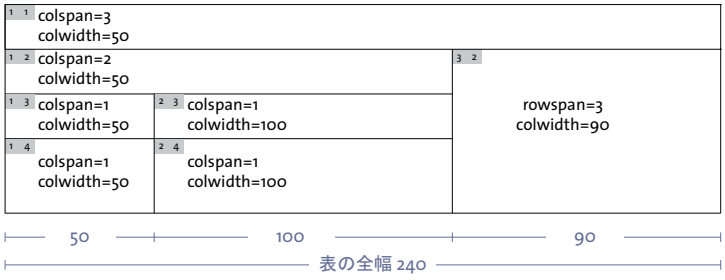


図 9.35
列幅を足し合わせると表全体の幅に

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 9.36 参照）。

列拡大縮小グループ

	Max. Load	Range	Weight	Speed
Giant Wing	12g	18m	14g	8m/s
Long Distance Glider	5g	30m	11.2g	5m/s
Cone Head Rocket	7g	7m	12.4g	6m/s

図 9.36
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。

9.3.4 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 9.37 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 `table/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 寸法で開始しましょう：

```
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。以下のコード断片は、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} " +
          "fillcolor=red margin=4 colspan=3 colwidth=" + c1;

tbl = p.add_table_cell(tbl, 1, 1, "Our Paper Plane Models", optlist);
```

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位置付け、すべての端で余白を 4 ポイントとります。259 ページ「セルの内容の縦位置を調整」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、セルが覆う長方形を灰色の背景色で塗りましょう。範囲枠機能について詳しくは 272 ページ「9.4 範囲枠」で説明しています。以下のコード断片は、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);
```

図 9.37　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：テキスト行付き画像セル追加
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　以下のコード断片は、「*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位置付け、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);
tbl = p.add_table_cell(tbl, 1, 4, "Benefit", optlist);

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);
```

手順４：テキストフローセルを追加　以下のコード断片は、「*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作成しましょう。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);
```

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);
```

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", "");

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);
```

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);

page = p.open_pdi_page(doc, pageno, "");

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);
```

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("");
```

9.3.5 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 9.38 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。

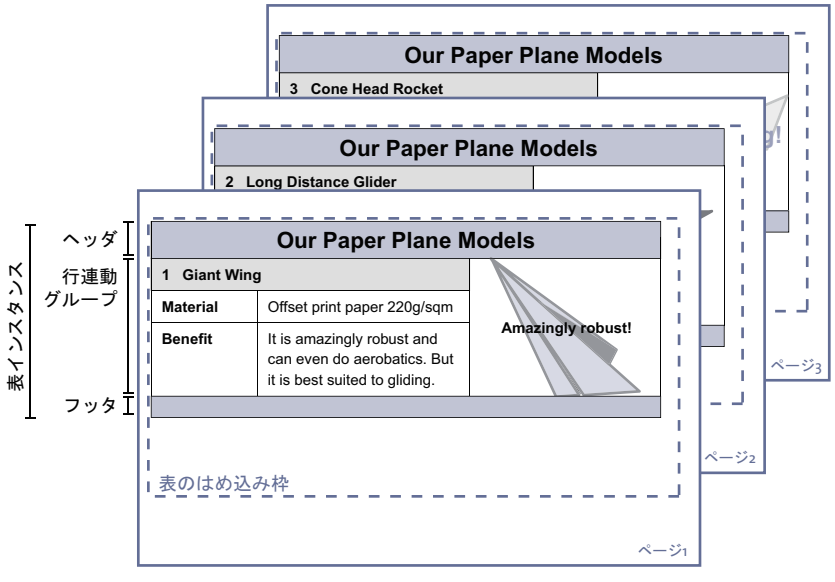


図 9.38
表は複数の表インスタンスに分解され、各はめ込み枠に 1 つずつ配置されます。

以下のコード断片は、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"));

/* 結果をチェック。「_stop」はすべてOKを意味します */
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です。*/
            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ます */
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：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。

セルを分割　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・SVG グラフィック・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 9.39 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 9.40 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。

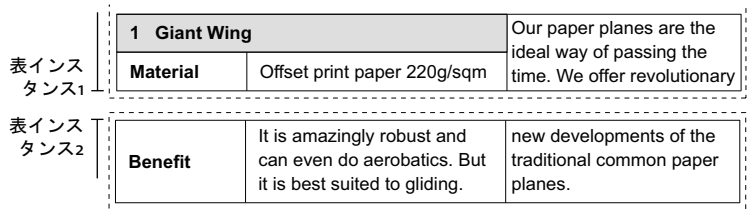


図 9.39
セルの分割

表行を分割　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 9.40 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。「*Benefit*」セルも繰り返されます。

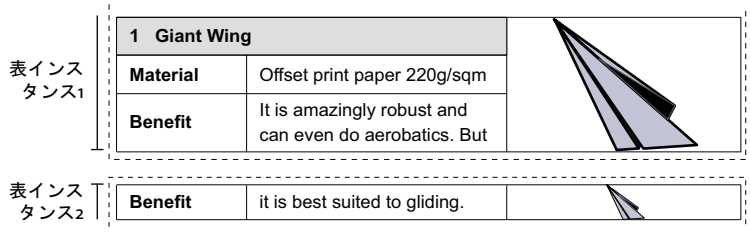


図 9.40
表行の分割

9.3.6 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、テキスト行・テキストフロー内容、先頭はめ込み枠の幅に基づいて算出されます。はめ込み枠の高さは無限と見なされます。先頭表インスタンス（すなわち先頭はめ込み枠内の表先頭部分の配置）は *PDF_fit_table()* の *fitmethod* オプションに従って算出されます。

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寸法を決定します。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：枠高さの 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。

その後、この表セルの残余の幅と高さが、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按分されます。

仮の表寸法を算出　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、仮の表寸法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。

rowheightdefault オプションを用いて、はめ込み枠の高さを完全に満たすか（キーワード *auto* と *distribute*）、それとも広さを節約するか（キーワード *minimum*）を指定することもできます。*rowheight*　オプションで明示的に表行の高さを指定した場合には、*rowheightdefault* 設定はつねにオーバライドされます。

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9.33参照)。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 9.36 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クエリできます。

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・グラフィック・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へ続ける　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・SVG グラフィック・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 9.39 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 9.38 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 9.40 参照）。

PDF_info_table() の *rowsplit* キーワードを用いると、表行が分割されているかどうかをチェックすることができます。

表をはめ込み枠の幅に縮める　与えた表列幅を合計した表幅は、それまでの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。たとえばテキスト行の横方向をはめ込んだ後などです。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。一切の横縮小を避けるには *horshrinklimit=100%* を用います。

PDF_info_table() の *horshrinking* キーワードを用いると横縮小率をクエリできます。

horshrinklimit の閾値を超えたとき、すなわち、表を横に過度に縮めなければならない場合には、以下のエラーメッセージが現れます：

```
Table width $1 exceeds fitbox width $2 (required shrinking factor $3 is smaller than
'horshrinklimit'
```

ここで $1 は算出された表幅を示し、$2 は *PDF_fit_table()* で与えられたはめ込み枠の幅、$3 は算出された縮小率です。このエラーが出た場合には、はめ込み枠の幅にもっと大きい値を与える必要があります。

表をはめ込み枠の高さに縮める　ヘッダ・フッタ行に少なくとも 1 つの表本体行または表行連動グループを加えた表の高さが、はめ込み枠の高さより大きくなることもありえます。この場合には、表の高さがはめ込み枠の高さと等しくなるまですべての表行が均等に低くされます。この縮小過程は *vertshrinklimit* オプションによって制限されます。一切の縦縮小を避けるには *vertshrinklimit= 100%* を用います。

PDF_info_table() の *vertshrinking* キーワードを用いると縦縮小率をクエリできます。

vertshrinking の閾値を超えたとき、すなわち、表を縦に過度に縮めなければならない場合には、以下のエラーメッセージが現れます：

```
Table height $1 exceeds fitbox height $2 (required shrinking factor $3 is smaller than
'vertshrinklimit'
```

ここで $1 は算出された表高さを示し、$2 は *PDF_fit_table()* で与えられたはめ込み枠の高さ、$3 は算出された縮小率です。このエラーが出た場合には、はめ込み枠の高さにもっと大きい値を与える必要があります。

後続するはめ込み枠の幅が異なる場合　後続するはめ込み枠が最初のはめ込み枠より広い場合、表セルはそのはめ込み枠の幅を完全に満たすように拡がるわけではありません。結果として、最初のはめ込み枠よりも広いはめ込み枠群を持つ表インスタンスはすべて、デフォルトでは等しい幅を持ちます。はめ込み枠を完全に満たすには、オプション *fitmethod=auto* を与える必要があります。*fitmethod=meet* を用いると、セル高さがそのアスペクト比を温存するよう適切に拡大されます。

最初のはめ込み枠よりも後続のはめ込み枠のほうが狭い場合には、表セルの幅は、狭いはめ込み枠に収まるよう縮まります。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。

9.4 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 API 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、1 個ないし複数の注釈を *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領域を定義します。
- 範囲枠の座標やその他のプロパティは、*PDF_info_matchbox( )* でクエリして、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照応する長方形として、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長方形から成ることもあります。

範囲枠の長方形（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、範囲枠の長方形を、*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。範囲枠の長方形の座標は、*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。

注 範囲枠は *blind* モードにおいては、すなわち *blind* オプションを用いて組版する際には、サポートされていません。

9.4.1 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、以下のコード断片は、範囲枠を青の背景色で塗ります（図 9.41a 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 9.41b 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 9.41c 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、範囲枠のまわりに長方形を描きましょう（図 9.41d 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 textflow/text_on_color トピックにあります。

図 9.41 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

9.4.2 テキストフロー内で範囲枠を用いる

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囲います（図 9.42 参照）：

図 9.42 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=red 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長方形として使いましょう（*PDF_create_annotation( )* の長方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 textflow/weblink_in_text 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 9.43 に示します。

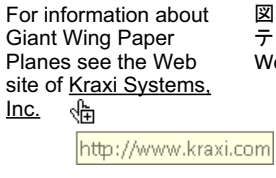


図 9.43
テキストフローの一部分に
Web リンクを追加

9.4.3 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 273 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、画像を以下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);
```

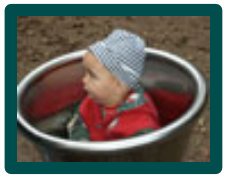
クックブック　完全なコードサンプルがクックブックの interactive/link_annotations トピックにあります。

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 9.44 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 9.44 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}}

テキストを画像に揃える　以下のコード断片は、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 9.45 参照）。

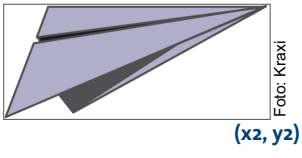
クックブック　完全なコードサンプルがクックブックの　images/align_text_at_image　トピックにあります。

```
/* 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)
{
        x2 = p.info_matchbox("giantwing", 1, "x2");
        y2 = 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 9.45 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

10 インタラクティブ機能

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。

10.1 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 10.1 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、その照応する 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 1 個ないし複数のインタラクティブ要素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、

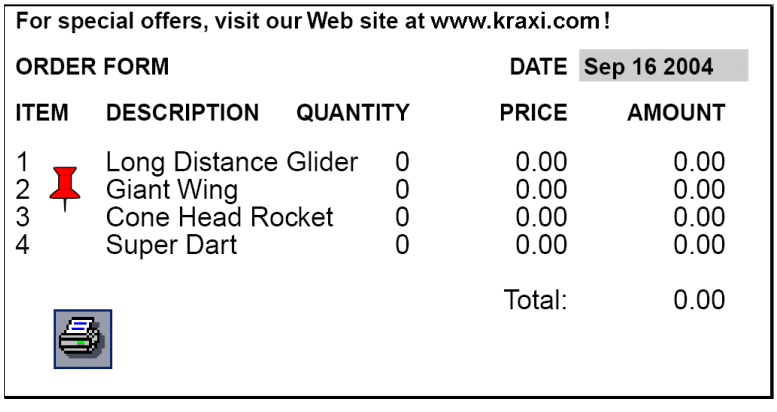


図 10.1
いろいろなハイパーテキスト要素を持つ文書

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付け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}";
/* 長方形座標の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 272 ページ「9.4 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。**

別のファイルへ移動するしおり　今度は、別のPDFファイルへ移動するしおり「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=blue* 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=blue";
```

```
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。**

ファイル添付を持つ注釈　次に、ファイル添付を作成しましょう。まず *FileAttachment* 型の注釈を作成しましょう。*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=red*）、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=red 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寸法のまま、指定した長方形の左上隅に表示されます。

10.2 フォームフィールドと JavaScript

印刷のための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しょう。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=yellow* オプションで背景を黄色に指定し、*bordercolor=black* 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、ラベルの寸法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、その照応する画像ファイル *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** `form_fields/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** `form_fields/form_textfield_layout`・`form_fields/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。まず、*JavaScript* 型（Acrobat では：**「*JavaScript* を実行」**）のアクションを作成しましょう。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、ページを作成しましょう。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);
```

最後に、上と同様にテキストフィールドを作成しましょう。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** `form_fields/form_textfield_fill_with_js` **トピックにあります。**

テキストフィールドのための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さまざまな簡単な JavaSctipt コード断片を *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 スニペットをそのフィールドの ***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。

以下のサンプルでは、***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** `form_fields/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 10.1 に挙げます。表 10.2 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 10.1 テキストフィールドのための JavaScript フォーマット関数

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 10.2 JavaScript フォーマット関数に対する引数

付け	説明・とりうる値
nDec	小数点以下の桁数
sepStyle	桁区切りのスタイル： ***0*** 1,234.56 ***1*** 1234.56 ***2*** 1.234,56 ***3*** 1234,56

表 10.2 JavaScript フォーマット関数に対する引数

付け	説明・とりうる値
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ます： *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
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ます： *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	いくつかの追加の形式を記述します： *0* ZIP コード *1* ZIP コード + 4 *2* 電話番号 *3* 社会保障番号

フォームフィールド入力を検証　以下のサンプルは、JavaScript をフォームフィールドに検証アクションとして紐付けて、テキストフィールドへのユーザー入力が、求められる形式 *mm/dd/yyyy* に合致しているかどうかをチェックします：

```
optlist =
"script={" +
```

```
        "// JavaScript code for date mask format MM/DD/YYYY\n" +
        "var re = /^[0-9]{2}\\/[0-9]{2}\\/[0-9]{4}$/\n" +
        "if (event.value !=\"\") {\n" +
        " if (re.test(event.value) == false) {\n" +
        " app.alert ({\n" +
        "    cTitle: \"Incorrect Format\",\n" +
        "    cMsg: \"Please enter date using mm/dd/yyyy format\"\n" +
        " });\n" +
        " }\n" +
        "}\n" +
"}";
validate_action = p.create_action("JavaScript", optlist);
textfield_font = p.load_font("Helvetica", "unicode", "");
optlist = "action={validate=" + validate_action + "} " +
          "backgroundcolor={gray 0.8} font=" + textfield_font;
p.create_field(llx, lly, urx, ury, "startdate", "textfield", optlist);
```

10.3 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 geospatial/starter_geospatial トピックにあります。

10.3.1 地理空間 PDF を Acrobat で利用

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 では、地理空間参照付き PDF 文書でいくつかのことができます（Acrobat・Acrobat Reader DC では、**「ツール」→「ものさし」**を有効にする必要があります。Acrobat X/XI 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ります。Adobe Reader X/XI では、**「編集」→「分析」→「地図位置ツール」**）：

- マウスカーソルの下の地図上の位置の座標を表示：**「地図位置ツール」**。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：**「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：**「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：**「ものさしツール」**。

Acrobat Reader では上記のうち最初の 2 つの機能だけが利用可能です。地理空間測定のため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）」**で変更できます。たとえば座標読み上げのための望ましい座標系などです。

PDFlib の地理空間機能は以下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（とサブオプション *georeference*）を用いて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。この方式は、地理空間 PDF への対応を有するすべての PDF ビューアで動作します。

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

- *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。

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

- *PDF_open_pdi_page()*・*PDF_load_graphics()*・*PDF_begin_template_ext()* の *georeference* オプションを用いて地球ベースの座標をフォーム XObject に割り当てることもできます。しかしこの方式は、Acrobat DC を含むいずれの既知のビューアにおいても対応していないため、推奨しません。

10.3.2 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、以下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ー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。これは以下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。特に、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。以下の Web サイトで、特定の EPSG コードに照応する WKT を示しています：

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

10.3.3 座標系の例

地理座標系の例 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。これは以下のように *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。これは以下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）。

投影座標系の例　投影は、その背景にある地理座標系に基づいています。以下の例では、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、以下のように *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]]
}}
```

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

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

10.3.4 Acrobat における地理空間 PDF の制約

地理空間 PDF を Acrobat X/XI/DC で扱うなかで、私たちは以下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、その照応する WKT を使うこと。
- 地理空間データをフォーム XObject に紐付けても動作しません。この理由から、*PDF_open_pdi_page()*・*PDF_begin_template_ext()*・*PDF_load_graphics()* に対して *georeference* オプションを指定することは推奨しません。PDF Reference に従えばこれは動作するべきなのですが。
  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ます：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。

11 文書交換

11.1 XMP メタデータ

文書情報フィールドのかわりとして、あるいはそれに加えるものとして、PDFlib はメタデータを指定するためのフレームワークとして、XMP（*Extensible Metadata Platform*）に対応しています。XMP は ISO 16684-1:2012 として標準化されています。PDFlib における XMP 対応にはいくつかの側面がありますので、以下説明します。

クックブック **シンプルな XMP サンプルが、クックブックトピック** `interchange/embed_xmp` **内にあります。**

多くの場合、XMP は文書全体にメタデータを紐付けるために用いられます。文書レベルメタデータのほかに、XMP はページ・フォント・ICC プロファイル・画像・グラフィック・画像・取り込み PDF ページに対して与えることもできます。これは、さまざまな関数の *metadata* オプションで実現できます。たとえば：

```
metadata={filename=info.xmp inputencoding=winansi}
```

この *metadata* オプションは、完全な XMP メタデータストリームないしはその一部分を受け付けます。PDFlib は、ユーザーが与えた XMP メタデータを、XML 規則と XMP/RDF 規則に従って検証します。PDF/A の場合には、カスタム XMP プロパティに対する追加規則が適用されます：338 ページ「12.3.8 PDF/A のための XMP 文書メタデータ」を参照してください。

内部・予約 XMP プロパティ　PDFlib は、いくつかの XMP プロパティを内部的に作成します。たとえば *CreationDate* です。他に、PDF/A や PDF/X といったさまざまな PDF 企画への準拠をシグナルするために必須の XMP プロパティ群があります。内部プロパティと規格関連識別プロパティは、ユーザーが与える XMP でオーバライドすることはできません。

文書情報フィールドに対する自動 XMP 同期　*PDF_begin/end_document()* の *autoxmp* オプションが *true* の場合には、PDFlib は、*PDF_set_info()* に与えられた文書情報フィールド群と、いくつかの内部的に生成されたエントリ（*CreationDate* 等）とを、その照応する、文書レベル XMP メタデータ内のエントリ群へ同期します。

標準 XMP スキーマ群のうちの 1 つの中にあるよく知られたプロパティに照応する文書情報フィールドは、適切なスキーマ内に配置されます。未知の情報フィールドは、通常、拡張 PDF（*pdfx*）スキーマ内に配置されますが、ただし PDF/A では無視されます。

XMP メタデータを転写　PDF 文書の大半ないしすべてのページが取り込まれたものである場合には、XMP メタデータがもし入力の中に存在していれば、それを転写することを推奨します。XMP メタデータを転写するには、以下のコード断片を使います：

```
if (p.pcos_get_string(indoc, "type:/Root/Metadata").equals("stream"))
{
        xmp = p.pcos_get_stream(indoc, "", "/Root/Metadata");
        p.create_pvf("/xmp/document.xmp", xmp, "");
        optlist += " metadata={filename=/xmp/document.xmp}";
}
```

```
p.end_document(optlist);
p.delete_pvf("/xmp/document.xmp");
```

11.2 Web 最適化（線形）PDF

PDFlib は、線形化という処理を PDF 文書に対して適用することができます（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 ファイル内部のオブジェクト群を再配列するとともに、アクセス高速化のために活用されうる補足情報を追加します。

非線形化 PDF は、クライアントへまるごと転送される必要がありますが、Web サーバは線形化 PDF 文書を、バイトサービングという処理を用いて、1 ページずつ転送することが可能です。これによって、Acrobat（ブラウザのプラグインとして動作している）は、PDF 文書の個別のページを別々に取得することが可能になります。その結果として、その文書の最初のページが、文書全体がサーバからダウンロードされ終わるのを待つことなく、ユーザーに提示されます。これはユーザー体験の向上をもたらします。

ただし、PDF データをブラウザへストリームするのは Web サーバであって、PDFlib ではありません。PDFlib は、バイトサービングのための PDF ファイルを作成します。バイトサービング PDF の利点を活用するには、以下のすべての要件が満たされる必要があります：

- PDF文書が線形化されている必要があります。これは*PDF_begin_document()*の*linearize*オプションで以下のように実現できます：

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

  Acrobat では、ファイルが線形化されているかは、その文書プロパティを見ればチェックすることができます（「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 ブラウザに実装されています。
- ユーザーがAcrobatをブラウザプラグインとして用いており、かつAcrobatでページごとのダウンロードを有効にしている必要があります（**「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。

線形化とファイルサイズ　線形化は、大きな PDF 文書の Web ベースでの表示を向上させることを目的としていますので、1 ページしかない文書に対してはあまり意味がありません。Acrobat 内のバグのため、小さな線形化文書は、線形化されているという扱いを受けないことがあります。たとえば Acrobat は、4KB 未満の文書を、その実際の線形化ステータスにかかわらず、非線型化であると見なします。

Acrobat はまた、2 GB より大きな PDF 文書を、線形化されていると見なしません。

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。

11.3 タグ付き PDF の基礎

タグ付き PDF は、ISO 標準 PDF/UA・PDF/A-1a・PDF/A-2a・PDF/A-3a、米国のセクション 508、ドイツの BITV、その他多くの法規における必須事項です。タグ付き PDF は、以下の利点を提供する文書構造情報で PDF を向上させます：

- アクセシビリティ：タグ付き PDF は、たとえば Acrobat 内蔵の読み上げ機能や、より高度なスクリーンリーダソフトウェア（図 11.1 参照）などを通じて、障碍を持つユーザーのためにアクセシブルです。
- 他の文書形式への信頼性高い書き出し・変換：タグ付き PDF を、RTF・XML・HTML といった他の形式へ変換すると、より正確な出力が得られます。

PDF/UA は、タグ付き PDF を、文書タグに関する要件を仕様化することで改良したものです。アクセシブルな PDF 文書を作成したいなら、PDF/UA のための追加規則群に従うことを推奨します：詳しくは 360 ページ「12.6 PDF/UA によるユニバーサルアクセシビリティ」を参照してください。

すべての PDF/UA の要件には従えない場合には（たとえば PDF/UA に準拠していない既存の PDF を元に文書を組み立てなければならないなどの理由で）、PDF/UA モードを無効にして、できる限り多くの PDF/UA 規則に従うことを推奨します。

クックブック　タグ付き PDF を生成するためのコードサンプルが、PDFlib クックブックの pdfua カテゴリにあります。クックブック内のタグ付き PDF サンプルはすべて PDF/UA を作成します。

11.3.1 論理構造ツリー（構造ヒエラルキー）

タグ付き PDF は、クライアントがその文書の内部構造に関する情報を提供し、かつ PDF 出力を生成する際に特定の規則に従った場合にのみ作成できます。タグ付き PDF を作成

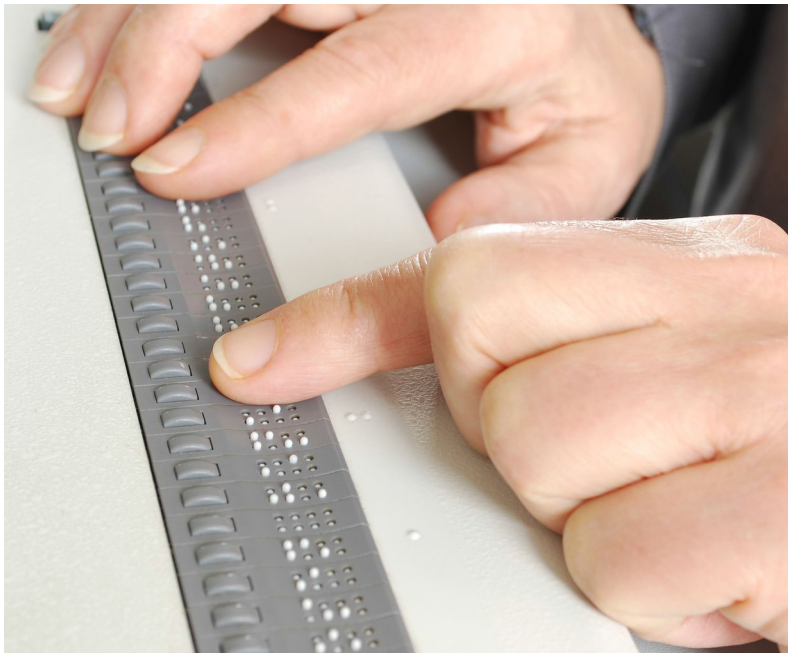


図 11.1
画面上のテキストをキャプチャして点字デバイス上に表示するスクリーンリーダ

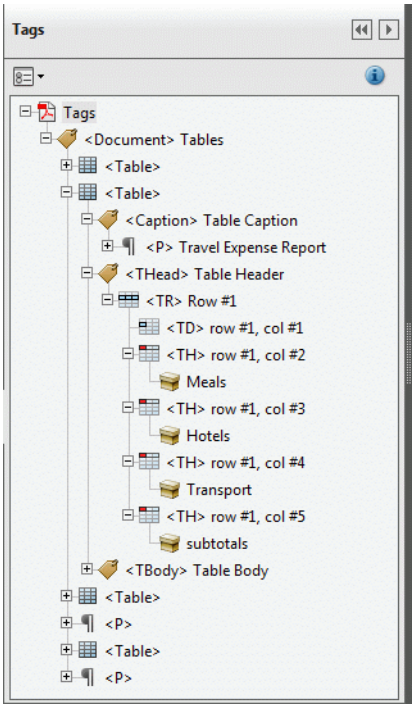


図 11.2 Acrobat のタグパネルが文書の論理構造ツリーを表示

するには、*tagged* 文書オプションを *true* に設定する必要があり、また、*lang* オプションが推奨されます：

```
if (p.begin_document("tagged.pdf", "tagged=true lang=en") == -1) {
        throw new Exception("エラー: " + p.get_errmsg());
```

タグ付き PDF 文書内の論理構造は、エレメントのヒエラルキーによって記述されます。これを構造ヒエラルキー、または論理構造ツリーといいます。ルートレベルから始まって、この構造ヒエラルキーは任意の数のレベルから成ります。各レベルにおいては、1 個のエレメントは、以下の種類のアイテムを 0 個以上含むことができます：

- 他の構造エレメント。たとえば *Document*（文書）エレメントは、複数の *Art*（アーティクル）エレメントを含むことができ、各 *Art* エレメントはさらに、複数の *P*（段落）エレメントを含むことができます。
- コンテンツアイテム。すなわち、ページ上のテキスト列とグラフィックや、取り込まれた画像から作成された XObject や、注釈・フォームフィールドへの OBJR 参照。こうしたアイテムは、構造エレメントに関連づけられたグラフィカルなコンテンツを表現します。

Acrobat における構造ヒエラルキー　タグ名と構造ヒエラルキーは、Acrobat X/XI/DC で以下のように表示できます：

- **「表示」→「表示切り替え」→「ナビゲーションパネル」→「タグ」**を選択（図 11.2 参照）

タグネスト化関数　構造エレメントは、*PDF_begin_item()* 関数を用いて明示的に作成できます。この関数は必ず、照応する *PDF_end_item()* への呼び出しとペアにする必要があります。たとえば、以下のコード断片は、見出し 1 個と段落 1 個を内容とするセクション 1 個から成るヒエラルキーを作成します。階層関係をインデントで視覚化しています：

```
/* 種別「Sect」（セクション）の構造エレメントを作成 */
id_sect = p.begin_item("Sect", "Title={来し方行く末}");

        /* 種別「H1」（見出し）の構造エレメントを作成 */
        id_h1 = p.begin_item("H1", "Title={企業沿革}");
            p.fit_textline(...);
        p.end_item(id_h1);

        /* 種別「P」（段落）の構造エレメントを作成 */
        id_p = p.begin_item("P", "");
            p.fit_textline(...);
        p.end_item(id_p);

/* 「Sect」を閉じる */
p.end_item(id_sect);
```

デフォルトでは、構造エレメントは、カレントでアクティブなアイテムの子として、他の子アイテムがすでに存在すればそれらすべての後に、挿入されます。エレメントが論理的順序で作成されるならば、これによって正しいツリー構造が得られます。もっと高度な技法については 318 ページ「11.4.4 コンテンツを順序にとらわれず作成」を参照してください。

構造エレメントは、オプションを通じて与えられる 1 個ないし複数の属性を保持することができます。たとえば、コンテンツエレメントの自然言語を指定するための *lang* や、画像に対する代替テキストのための *Alt* です。利用可能なオプションの集合は、構造エレメントの種別によって異なります。

短縮タグ付け　多くの場合において、構造エレメントのコンテンツは、PDFlib はめ込み関数を 1 回呼び出すことで作成することができますので、結果として典型的な並びは *PDF_begin_item()/PDF_fit_*()/PDF_end_item()* となります。この並びは、短縮タグ付けを用いて削減することが可能です。ページコンテンツを作成するための多くの関数が対応している *tag* オプションを用いることによって、コンテンツを配置することと、タグ付け情報を与えることとを、ただ 1 回の関数呼び出しにまとめることができます。上記のコード断片は、テキスト・画像配置関数に *tag* オプションを与えることによって単純化できます：

```
/* 種別「Sect」（セクション）の構造エレメントを作成 */
id_sect = p.begin_item("Sect", "Title={来し方行く末}");

        /* 種別「H1」（見出し）の構造エレメントを作成 */
        p.fit_textline(..., "tag={tagname=H1 Title={企業沿革}}");

        /* 種別「P」（段落）の構造エレメントを作成 */
        p.fit_textline(..., "tag={tagname=P}");

/* 「Sect」を閉じる */
p.end_item(id_sect);
```

タグネスト化関数は *PDF_begin_item()* にも対応していますので、両方の方式を合わせることも可能です。これは、タグの下にタグをネストする必要があり、そしてネストされたタグがさらに複数のコンテンツアイテムを内容とする場面において役立ちます。たとえば、*Caption* エレメントの中に *P* エレメントがあり、その中にさらに複数のテキストアイテムがあるようにしたければ、以下のように生成できます。

```
id_caption = p.begin_item("Caption", "tag={tagname=P}");
        p.fit_textline(...);
        p.fit_textline(...);
p.end_item(id_caption);
```

tag オプションのオプションリストの中のサブオプションとして 2 番目の *tag* オプションをネストすることもできます。実際、*tag* オプションは任意のレベルまでネストさせることが可能です。ネストされた構造エレメントの内容がコンテンツアイテム 1 個だけである場合には、上記のコード断片は、テキストをページ上に配置するための呼び出しの中で短縮タグ付けを直接適用することにより、さらに単純化することが可能です。

```
p.fit_textline(..., "tag={tagname=Caption tag={tagname=P}");
```

短縮タグ付けは、生成された構造エレメントのハンドルを露出しませんので、*PDF_activate_item()* で、または *tag* オプションの *parent* サブオプションで使うことはできません。これらは構造エレメント ID を必要とするからです。*tag* オプションの動作は詳しくは以下のとおりです：

- 生成されたコンテンツのために新規の構造エレメントが作成され、そして呼び出しから返る前に閉じられます。以下の状況はこの規則から除外されます：
  - *PDF_begin_document()* で *tag* オプションを用いて作成された構造エレメントは、*PDF_end_document()* で閉じられます。
  - *PDF_begin_item()* の *tag* オプションを用いて複数のタグが与えられたときは、これらすべてのタグは、照応する *PDF_end_item()* への呼び出しで閉じられます。
- *PDF_fit_table()*：*tagname=Table* または *PDF_fit_table()* で *Table* にロールマップされているタグ名は、PDFlib に対して、必要なテーブルタグ群を作成するよう指示します（310 ページ「11.4.1 自動表タグ付け」参照）。表セルに対して自動的に生成された *TH・TD* タグは、*PDF_add_table_cell()* の *tag* オプションでさらに修飾することもできます。
- *PDF_fit_textflow()*：完全なテキストフローインスタンスは新規の構造エレメントを形成します。
- 生成されるエレメントは、カレントでアクティブなアイテムの、または *parent* オプションで与えられているアイテムの子です。
- グループ化エレメントは、*PDF_begin_item()* でのみ作成することができ、*PDF_begin_document()* 以外の短縮タグ付けを用いて作成することはできません。
- *PDF_begin_document()* と *PDF_add_table_cell()* の場合を除いて、短縮タグ付けは**ページ**スコープ内でのみ使用できます。

11.3.2 標準・カスタムエレメント種別

標準エレメント種別　PDF は、幅広い文書分類に対応するよう設計されたさまざまな標準エレメント種別に対応しています。PDFlib は、表 11.1 に従ってこれらすべての標準エレメント種別に対応しています。この表の中で示している説明は、適切な種別を選ぶ助けとなることを意図しています。表 11.1 はまた、301 ページ「通常エレメントと直接エレメントの違い」で説明する BLSE/ILSE の区別も示しています。

グループ化エレメントは、他のエレメント群を保持するコンテナです。これは直接ページ要素を内容とすることはできません。PDF が完全な文書を内容とする場合には、*Document* エレメントを構造ツリーのルートとして用いるべきです。PDF が文書の一部分を内容とする場合には、*Part* か *Sect* をルートとして用いるべきです。ルートエレメントは、*PDF_begin_document()* の *tag* オプションで簡便に与えることができます。

擬似エレメント種別は、いかなる構造エレメントをも作成することなく、コンテンツを特定の特性でマークアップするために用いられます。これは主に、ページ装飾をマークアップするために用いられます（302 ページ「11.3.3 ページ装飾」参照）。

表 11.1 タグ付き PDF 内の標準エレメント種別（タグ）と、PDFlib によって追加された擬似エレメント種別

種別	説明
グループ化（コンテナ）エレメント（直接ページコンテンツを内容とすることはできない）	
Document	完全な文書。構造ツリーのルートエレメントとして推奨します
Part	ヒエラルキーを考慮せずに構造エレメント群をグループ化したものを内包（セマンティクス界の Div）
Sect	（セクション）ヒエラルキーを考慮して構造エレメント群をグループ化したものを内包
Div	（分割）一般的ブロックレベルエレメントまたはエレメント群のグループ。これはセマンティクスに基づかないグループ化に対して使用するべきです。たとえばブロックレベルエレメントのシーケンスに lang 属性を紐付けたり、他のグループ化要素がどれもふさわしくないカスタムグループ化構造エレメントをロールマップしたりする場合がこれにあたります。
Caption	（キャプション）テーブルかリストか図を説明するテキストの短い部分
見出し・段落エレメント（*BLSE*）	
H	（見出し。PDF/UA-1 では structuretype=strong の場合のみ）文書のコンテンツの下位分割部分のための見出し。これは、その親の最初の子であるべきです。H エレメントは、階層的なネスト化のために意図されています。このエレメントは推奨されません。H1 ～ H6 を用いた弱い構造化を用いるべきです。
H1～H6	弱く構造化された文書のための特定のレベルを持つ見出し。これは、アプリケーションがセクション群を階層的にネストできないために見出しのレベルをそのネスト化のレベルから判定できない場合に用いるべきです。
P	（段落）一般的段落エレメント、すなわち、見出しではないテキストの低レベルな分割部分
ラベル・リストエレメント（*BLSE*）	
L	（箇条書き）同様の意味・重要性を持った項目の列
LI	（箇条書き項目）箇条書きの個々のメンバ
Lbl	（ラベル）同一箇条書き内で、あるいは同様の項目群の他のグループ内で、ある特定の項目を他から区別する名前または番号。たとえば、ビュレットリストまたは番号付きリストでは、ビュレットキャラクタまたはその箇条書き項目の番号および伴う約物。
LBody	（箇条書きボディ）箇条書き項目の説明コンテンツ
テーブルエレメント（すべてのテーブルタグは自動的に作成可能です。*310*ページ「*11.4.1* 自動表タグ付け」参照）	
Table	（テーブル。BLSE）長方形セルの 2 次元レイアウトで、複雑な下部構造を持つこともあります
TR	（テーブル行）テーブル内の見出しまたはデータの表行
TH	（テーブルヘッダセル）テーブルの 1 個ないし複数のテーブル行または列を記述するヘッダテキストを内容とするテーブルセル
TD	（テーブルデータセル）テーブルのコンテンツの一部であるデータを内容とするテーブルセル

表 11.1 タグ付き PDF 内の標準エレメント種別（タグ）と、PDFlib によって追加された擬似エレメント種別

種別	説明
THead	（テーブルヘッダ行グループ：PDF 1.5）テーブルのヘッダを成すテーブル行のグループ
TBody	（テーブルボディ行グループ：PDF 1.5）テーブルの本体部を成すテーブル行のグループ
TFoot	（テーブルフッタ行グループ：PDF 1.5）テーブルのフッタを成すテーブル行のグループ
インラインエレメント	
Span	（スパン。テーブルのテーブル行または列の連結とは関係ありません）特定の固有の特性を何ら持たない一般的なテキストのインライン部分。
インタラクティブ要素のためのエレメント（*313*ページ「*11.4.2* インタラクティブ要素にタグ付け」参照）	
Link	（リンク）そのエレメントのコンテンツの一部分と、1 個ないし複数の照応するリンク注釈との間の関連付け
Annot	（注釈。PDF 1.5）そのエレメントのコンテンツの一部分と、1 個の照応する PDF 注釈との間の関連付け。これは、Link または Form のほうが適切である場合以外、すべての注釈に対して用いられるべきです。
Form	（フォーム）インタラクティブフォームフィールド。
イラストレーションエレメント	
Figure	（図）グラフィカルなコンテンツのアイテム
Formula	（数式）数式。このエレメント種別は、コンテンツエレメント全体を数式として識別します。数式が画像として表されている場合には、Formula エレメントが依然用いられるべきです（Figure ではなく）。
和文ルビ・割注のためのエレメント（*PDF 1.5*）	
Ruby	より小さなテキストサイズで書かれ、それが参照する親文字テキストに隣接して配置されたサイドノート。この Ruby エレメントは、ルビ組立構造全体を囲むラッパとしての働きをします。
RB	（ルビ親文字テキスト）ルビ注釈が施される対象となるフルサイズのテキスト。
RT	（ルビ注釈テキスト）ルビ親文字テキストに隣接して配置されるべき、より小さなサイズのテキスト。
RP	（ルビ約物）ルビ注釈テキストを囲う約物。これは、ルビ注釈がルビスタイルで適切に組版できずに、通常のコメントとして組版されるか割注として組版される場合にのみ用いられます。
Warichu	（割注）より小さなテキストサイズで、それを含むテキスト行の高さの中に、より小さな 2 行として組版され、それが参照するベーステキストに後続して（インラインに）配置される、コメントまたは注釈。
WT	（割注テキスト）割注コメントの、2 行として組版され、それを囲う WP エレメントの間に配置される、より小さなサイズのテキスト。
WP	（割注約物）WT テキストを囲う約物
非構造	
NonStruct	（非構造エレメント）固有の構造特性を何も持たないエレメントをグループ化。これはグループ化のみを目的とするものです。
擬似エレメント種別	
Artifact	実のページコンテンツから区別されるべきページ装飾（302 ページ「11.3.3 ページ装飾」参照）。

表 11.1 タグ付き PDF 内の標準エレメント種別（タグ）と、PDFlib によって追加された擬似エレメント種別

種別	説明
ASpan	（アクセシビリティスパン。PDF には Span として書き込まれますが、インラインアイテム Span とは区別する必要があります）構造エレメントに属しない、あるいは構造エレメントの一部分であるようなコンテンツに対して、アクセシビリティ特性群を紐付けます。この ASpan 擬似エレメントは、Alt・ActualText・Lang・E といったアクセシビリティ属性を持った Span として書き込まれます。ASpan は、いかなる構造エレメントとも紐付きません。
Reversed-Chars	（廃止）右書き用字系の中の、反転したキャラクタ群によるテキストを指定。
Clip	（廃止）マークされた切り抜き列を指定。これは、クリッピングパスまたはテキスト表現モード 7 のテキストのみを内容とする列であり、目に見えるグラフィックまたは PDF_save( )/PDF_restore( ) を一切含みません。

PDF 2.0 で廃止になった標準エレメント種別　に挙げるエレメント種別は、PDF 1.7/ISO 32000-1 では利用可能でしたが、PDF 2.0/ISO 32000-2 では廃止となりましたので使用するべきではありません。

表 11.2 PDF 2.0 において廃止の標準エレメント種別（タグ）。これらの種別は使用するべきではありません

種別	説明
Art	（アーティクル）単独の物語または解説を構成する、比較的自己完結しているテキストの本文
BibEntry	（書誌項目）何らかの引用コンテンツの外部ソースを特定する参照
BlockQuote	（ブロック引用）周囲のテキストの作成者以外の誰かに帰する 1 個ないし複数の段落から成るテキストの部分
Code	コンピュータプログラムテキストの断片
Index	（インデックス）文書の主本文内における指定されたテキストの出現を指し示す Reference エレメントを伴った識別テキストを内容とする項目の列
Note	脚注や後注のように、文書の本体の中から参照される説明テキストのアイテム。このエレメントは子として Lbl を持つことができます。
Private	（プライベートエレメント）アプリケーションに属するプライベートコンテンツを内容とするグループ化エレメント。この種別の構造上の意義は仕様化されていません。この Private エレメントも、その子孫エレメントのいずれも、他の文書形式へは解釈も書き出しもされません。
Quote	（引用）周囲のテキストの作成者以外の誰かに帰するテキストのインライン部分。この引用されたテキストは、単一の段落内にインラインで含まれているべきです。
Reference	文書内の他の所にあるコンテンツの引用。これはローカルリンクのために使用されるべきです。
TOC	（目次）目次項目群（エレメント種別 TOCI）および / または他のネストされた目次エントリ群（TOC）から成るリスト。
TOCI	（目次項目）目次のメンバ。これは適切な Link エレメントを内容とするべきです。

構造エレメントに対するネスト規則　構造エレメントを作成するにあたっては、さまざまな規則に従う必要があります。これらの規則を表 11.3 にまとめてあります。この規則は、列挙した標準エレメント種別と、各標準種別へロールマップされたカスタムエレメント種別（301 ページ「カステムエレメント種別とロールマップ」参照）に対して適用されます。PDF/UA-1 には追加の規則群が適用されます（360 ページ「12.6 PDF/UA によるユニバーサルアクセシビリティ」参照）。

新規構造エレメントに対するネスト規則は、*PDF_begin_document()* の *checktags* オプションを用いて無効にしたり、緩和したりすることもできます。しかしこれは、無効な構造ヒエラルキーを生み出す可能性がありますので推奨しません。このオプションは、レガシアプリケーションのための移行の助けとして意図されているものです。表 11.3 内のいくつかの規則を「厳格規則」として標識しています。オプション *checktags=relaxed* は、この厳格規則群以外のすべての規則を強制します。

表 11.3 PDF_begin_item( ) と、さまざまな関数の tag オプションに対するタグネスト規則

アイテム	規則
コンテンツアイテム	以下のエレメントは、ページコンテンツ、すなわちテキストか画像かベクトルグラフィックを内容として持つことができます： H・H1・H2・... P Lbl・LBody TH・TD Span・Quote[1]・Note・Reference・BibEntry[1]・Code[1] Link・Annot Figure・Formula RB・RT・RP・WT・WP Artifact・ASpan・ReversedChars・Clip これ以外のすべてのエレメントは、直接ページコンテンツを追加する前に、中間構造エレメントを必須とします。コンテンツアイテムを追加しようと試みたときには、PDFlib は例外を発生させます。
グループ化エレメント	グループ化エレメントは、コンテンツアイテムか ASpan か ILSE を子として持ってはいけません。すなわち、コンテンツが作成できる前に、BLSE が作成される必要があります： Document・Part・Art[1]・Sect・Div・BlockQuote[1]・Caption・TOC[1]・TOCI[1]・Index[1]・NonStruct[1]・Private[1] 以下のエレメントに対しては、グループ化エレメントの子である場合、オプション Placement=Block を推奨します：Figure・Formula・Form・Link・Annot
ブロックレベルエレメント	以下のブロックレベルエレメントは、コンテンツアイテムを子として持ってはいけません。すなわち、コンテンツが作成できる前に、然るべきグループ化エレメントか BLSE が作成される必要があります： L・LI・Table・TR・THead・TFoot・TBody 厳格規則：P エレメントはグループ化エレメントを内容として持つことができません。
擬似・インラインエレメント	擬似エレメント（すなわち Artifact・ASpan・ReversedChars・Clip）と、以下の ILSE は、direct=true の場合には子孫を一切持ってはいけません： Code[1]・BibEntry[1]・Note・Quote[1]・Reference[1]・Span ただしこれらのエレメントは、direct=false の場合には子を持つことができます。 擬似エレメントと、以下の ILSE の中には、direct=true の場合にはページ装飾を作成できません：Span,Quote[1]・Note[1]・Reference[1]・BibEntry[1]・Code[1]

表 11.3 PDF_begin_item( ) と、さまざまな関数の tag オプションに対するタグネスト規則

アイテム	規則
表エレメント	Table エレメントは、1 個ないし複数の TR エレメントを、あるいは 1 個のオプショナルな THead に続けてその後の 1 個ないし複数の TBody エレメントと 1 個のオプショナルな TFoot とを、内容として持つことができます。TBody は Table の唯一の子であるべきではありません。これに加えて Table エレメントは、1 個の Caption エレメントを、その最初または最後の子として持つこともできます。 TH・TD エレメントは、TR・TH・TD・THead・TBody・TFoot を内容として持つことはできません。 THead・TBody・TFoot エレメントは、TR エレメントのみを内容とすることができ、かつ Table のみを親とすることができます。 TR は、Table・THead・TBody・TFoot のみを親とすることができます。
リストエレメント	L エレメントは、オプショナルに 1 個の Caption エレメントを、また 1 個ないし複数の LI エレメントを内容として持つことができます。 LI エレメントは、1 個ないし複数の Lbl または LBody エレメントを、あるいは両方を内容として持つことができます。LI は L のみを親とすることができます。LBody は LI のみを親とすることができます。
目次[1]	TOC[1] エレメントは、1 個のオプショナルな Caption エレメントを最初の子として、また 1 個ないし複数の TOCI および TOC エレメントを（組み合わせでも）、内容として持つことができます。 TOCI[1] エレメントは、Lbl・Reference・NonStruct・P・TOC エレメントのみを内容とすることができます。TOCI は TOC のみを親とすることができます。
ラベルエレメント	Lbl は、Annot・LI・Link・TOCI・BibEntry・Note[1] のみを親とすることができます。
インタラクティブ要素：リンク・フォームフィールド・注釈	以下のエレメント種別は、PDF_create_field( )・PDF_create_annotation( ) の tag オプションに対しては、あるいは Link・Annot・Form に対する親としては許容されません：表エレメント、ILSE、ルビ・割注エレメント、擬似エレメント。 Annot エレメントはネストできません。 Link エレメントはネストできません。 Form エレメントは、PDF_create_field( ) によって自動的に作成される OBJR エレメント以外のエレメントを内容として持ってはいけません。
和文ルビ・割注	Ruby は RB・RT・RP を子として持つことができますが、それ以外のエレメント種別を子として持ってはいけません。 Warichu は WT・WP を子として持つことができますが、それ以外のエレメント種別を子として持ってはいけません。
***PDF 2.0*整合性規則**	まれにしか使われない PDF 1.7 の親子タグ組み合わせに対して、PDF 2.0 のネスト化規則との整合性のために、以下のネスト化規則が導入されています： 非互換的変更（ネスト化規則が厳しくなったもの）： ► RB・RT・RP は Ruby の子にのみなれる ► WT・WP は Warichu の子にのみなれる ► Document は Caption を内容とすることができない ► Caption は今後 Document・Caption を内容とすることが許されない ► H・H1 等は今後 Document・Part・Div を内容とすることが許されない ► TH・TD は今後 Document・Caption を内容とすることが許されない ► direct=false によるインラインエレメントは今後 L・Table・Document を内容とすることが許されない

1. PDF 2.0 では廃止

通常エレメントと直接エレメントの違い　たいていの構造エレメントは、照応するテキストかグラフィックをマークアップし、かつ文書構造ツリー内へ照応するエントリを追加することによって、PDF へ出力されます。このようなエレメントは Acrobat のタグペーンに表示されます。これを通常エレメントと呼びます。

これに対し、マーク付きコンテンツのみから成り、構造ツリー内にエントリを持たないエレメントというものもあります。このようなエレメントは Acrobat のタグペーンに表示されません。このような直接エレメントの利点は、PDF 出力内で必要なバイト数が少ない点です。構造種別のなかには、*direct* オプションを用いてそのステータスを変えることができるものがあります（デフォルトは *true*）。通常の構造エレメントと直接エレメントとの比較を表 11.4 に示します。デフォルトで直接なエレメントが別の直接でないエレメントを内容としている場合にはオプション *direct=false* が必須です。たとえば *Reference* が *Link* を内容としている場合がこれにあたります。

表 11.4 通常アイテムと直接アイテム

	通常アイテム（direct=false）	直接アイテム（direct=true）
対象アイテム	その他すべてのエレメント種別	Code・BibEntry・Note・Quote・Reference・Span BibEntry・TOCI・Note の子として Lbl 疑似アイテム ASpan・ReversedChars・Clip
構造ツリーの一部である	○	×
ページ境界をまたぐことが可能	○	×
他のアイテムで割り込むことが可能	○	×
PDF_activate_item( ) で有効にすることが可能	○	×
子エレメントに対するネスト規則	通常・直接エレメントを内容とすることが可能	その親が直接エレメントを内容とすることが可能な場合に限り、直接エレメントのみを内容とすることが可能

空の構造エレメント　一般的には、子エレメントもコンテンツアイテムも内容としない構造エレメントは避けるべきです。しかし、以下の例のように、この規則にはいくつかの例外があって、そこでは空の構造エレメントが、何らかの他の構造の中にあって、意味的な役割を伝えます：

- 空の表セルに対しては空の *TD* エレメントが必要です。PDFlib の自動表組版機能はこの規則に従います（310 ページ「11.4.1 自動表タグ付け」参照）。
- リスト構造の中の空の *LI* エレメント。

プログラミングスコープとページ境界　多くの構造エレメントは、**ページ**スコープ内でのみ作成できます。グループ化エレメントは、複数のページにわたる場合があり、**文書**スコープ内でも作成できます。疑似アイテムと直接アイテムは、ページを終了または一時停止する前に閉じる必要があります。

カスタムエレメント種別とロールマップ　表 11.1 に挙げた定義済み標準構造エレメントに加えて、カスタムエレメント種別名を用いることもできます。カスタムエレメント種別

名は通常、エレメント種別名をローカライズしたり（たとえばドイツ語 ***Abbildung*** が ***Figure*** にマップ）、アプリケーション独自の種別名で作業したり（たとえば ***Normal*** が ***P*** にマップ）するために用いられます。カスタムエレメント種別名を含んだ文書の再利用を可能にするためには、そのカスタム名を、標準構造エレメント種別の集合の中の、それと正確にないしおおむね同等のものへマップする必要があります。また、標準エレメント種別を、その意味付けを変更するために他の標準種別へ再マップすることも可能です。カスタムエレメント種別は、インラインエレメント・擬似エレメントへはマップできません。エレメントマッピングは ***rolemap*** 文書オプションで実現できます。たとえば

```
p.begin_document("tagged.pdf",
        "tagged=true lang=en rolemap={ {Heading H1} {Subhead H2} {Paragraph P} }");
```

Acrobat におけるロールマップ　ロールマップは、Acrobat X/XI/DC で以下のように表示・編集できます：

- **「表示」→「表示切り替え」→「ナビゲーションパネル」→「タグ」**を選択し、**「タグ」**パネルの上端にあるメニューボタンをクリックして、ドロップダウンリストから**「ロールマップを編集」**を選択

11.3.3 ページ装飾

実質的なコンテンツとページ装飾　ページのさまざまなコンテンツは、以下のカテゴリのいずれかにあてはまります：

- 実質的なコンテンツ。文書作成者によって、その文書の意味を伝えるために作成されています。その文書の論理構造ツリーは、実際のコンテンツを成すオブジェクト群を記述しているほか、注釈も含む場合があります。
- 実質的なページコンテンツに貢献しておらず、ページネーションまたはレイアウト目的のために作成されているグラフィックまたはテキストオブジェクトを、ページ装飾といいます。ページ装飾は、構造ツリー内に含まれておらず、スクリーンリーダによって読み上げられません。

ページ装飾を標識することは、アクセシビリティを向上させるために強く推奨され、PDF/UA-1 では必須です。典型的なページ装飾は、反復されるヘッダ・フッタ、ページ番号、背景画像、その他各ページ上で反復されるアイテムです。

Acrobat におけるページ装飾　ページ装飾は、Acrobat X/XI/DC で以下の方式のいずれかでチェックできます：

- **「ツール」→「アクセシビリティ」→「読み上げ順序」**（Acrobat DC）または**「*TouchUp* 読み上げ順序」**（Acrobat X/XI）を選択すると、ページ上のコンテンツエレメントを表示または編集できます。ページ装飾は、Acrobat の「**読み上げ順序**」ツール内では**「背景」**と呼ばれています。これは構造エレメントとは異なり、このツールをアクティブにした際にふちとタグ名を用いて視覚化されません。
- ページ装飾を識別するには、**「表示」→「表示切り替え」→「ナビゲーションパネル」→「コンテンツ」**を選択します。**「コンテンツ」**パネルには、すべてのページコンテンツが、それぞれの構造エレメント種別名または**「ページ装飾」**のいずれか適切なほうとともに一覧表示されます。ページ装飾は読み上げられませんので、ページ上に、照応する番号付きブロックはありません。ただし、この一覧内のページ装飾をクリックすると、その照応する、ページ上のコンテンツエレメントがハイライトされます。
- ページ装飾を検索：**「表示」→「表示切り替え」→「ナビゲーションパネル」→「タグ」**を選択し、**「タグ」**パネルの上端にあるメニューボタンをクリックして、**「検索 ...」**の

後、ドロップダウンリストから**「ページ装飾」**を選択します。ページ装飾は文書構造の一部分ではありませんので、タグナビゲーションパネルにも順序パネルにも、照応するエントリはありません。

- **「表示」→「読み上げ ...」**を有効にすると、Acrobat がページ上の構造エレメントを読み上げることができるようになります。ページ装飾は読み上げられません。

コンテンツをページ装飾として指定　ページ装飾は、PDFlib で、*PDF_begin_item()* 内で *Artifact* タグ名で指定することができます（ページ装飾は構造エレメントの意味においては本当はタグではないということはさておき）：

```
id = p.begin_item("Artifact", "");
```

あるいは、ページ装飾は、短縮タグ付けで、すなわちさまざまな関数の *tag* オプションで指定することもできます：

```
p.fit_textline(text, x, y, "tag={tagname=Artifact}");
```

ページ装飾は、BLSE がカレントでアクティブでないときにのみ作成することを推奨します。しかし、アプリケーションによってはこれはつねに可能とは限りませんので、ページ装飾が作成される際には、PDFlib はカレントでアクティブなエレメントを自動的に中断し、そのページ装飾の後でそれを再びアクティブ化します。なお、タグネスト規則（298 ページ「構造エレメントに対するネスト規則」参照）は、ILSE 内にページ装飾を許容しません。

ページ装飾を分類　非実質的なページコンテンツは、*Artifact*　擬似タグで識別し、*artifacttype* オプションの以下のキーワードのうちのいずれか 1 つに従って分類する必要があります：

- *Pagination*（ページネーション）ページ装飾：ランニングヘッドやページ番号といったページ機能群。ページネーションページ装飾は、*artifactsubtype* オプションと、キーワード *Header*・*Footer*・*Watermark* のうちのいずれか 1 つでさらに分類できます。
- *Layout*（レイアウト）ページ装飾：罫線や表シェーディングといったタイポグラフィまたはデザイン要素。
- *Page*（ページ）ページ装飾：補助、たとえばトンボやカラーバーなど。
- *Background*（背景）ページ装飾：ページの幅または高さいっぱいに、あるいは構造エレメントの寸法いっぱいに伸びる画像または色付き領域。

ページネーションページ装飾と背景ページ装飾では、どのページ辺ないしページ辺群（*Top/Bottom/Left/Right*）にそのページ装飾が付着しているかを指定する*Attached*オプションが使えます。

以下の例は、下位種別 *Header* を持つページネーションページ装飾を作成します：

```
id = p.begin_item("Artifact",
        "artifacttype=Pagination artifactsubtype=Header Attached={Top Left}");
```

自動ページ装飾タグ付け　すべてのページコンテンツは、構造エレメントかページ装飾のいずれかとしてタグ付けされているべきですので、PDFlib は自動的に、ある種のグラフィッカルな要素群にタグ付けを行います。以下の装飾要素は、タグ付き PDF モードでは自動的に、*artifacttype= Layout* を持つ *Artifact* としてタグ付けされます：

- ブロック装飾：*PDF_fill_*block()*　によって作成されるすべての装飾要素、すなわち *backgroundcolor*・*bordercolor* プロパティに従って作成される描線と塗り。

- 範囲枠装飾：範囲枠オプション *fillcolor*・*shading*・*strokecolor* に従って作成される範囲枠長方形。
- 表装飾：自動表タグ付けが有効の場合には（310 ページ「11.4.1 自動表タグ付け」参照）、表の罫線と網掛け、すなわちオプション *fill*・*stroke*・*showborder*・*showgrid* に従った描線と塗り。
- テキスト行ページ装飾：*leader*・*shadow*・*showborder*
- テキストフローページ装飾：*leader*・*shadow*・*showborder*・*showtabs*
- テキスト装飾：*underline*・*overline*・*strikeout*

すべてのタグ付き PDF 機能同様、自動ページ装飾タグ付けは**ページ**スコープ内でのみ動作します。

11.3.4 テキスト処理

言語指定　タグ付き PDF では、テキストの自然言語は明示的に指定されるべきです：これによって、スクリーンリーダがその文書を読み上げる際に適切な言語へ切り替われるようになります。この自然言語は、さまざまなレベルで指定することができます：

- *PDF_begin_document()* の *lang* オプションを、主要言語、すなわちその文書の全体としての自然言語を指定するために設定するべきです。この指定は、ページ内容のみならず、しおりや注釈といったインタラクティブ要素をもカバーします。
- この文書言語は、任意の構造レベル上にある個別のアイテムについて、*PDF_begin_item()* の *lang* オプションで、またはさまざまな関数の *tag* オプションでオーバライドすることもできます。
- ハイパーテキスト文字列は、言語を指定するための Unicode エスケープシーケンスを含むこともできます（後述）。

テキストとして符号化されながら、しかし自然言語の一部分ではないコンテンツ、たとえばプログラミングコード、音符、書体サンプル、数式などは、空の言語コードを用いるべきです。例：*lang={}*

Unicode 言語識別子は、以下の並びから成っています：

- Unicode 値 U+001B（2 バイト）
- ISO 639 言語コード 1 個（2 ASCII バイト）。例：*en*・*ja*・*de*
- オプショナルに、ISO 3166 国コード 1 個（2 ASCII バイト）。例：*US*・*JP*
- Unicode 値 U+001B（2 バイト）

16 進表記での例：

```
001B656E5553001B          (=enUS)
001B7A68001B              (=zh)
001B6465001B              (=de)
```

Java のような Unicode 対応言語では、この 2 個の ASCII キャラクタはまとめて 1 個の Unicode 値とする必要があります：

```
\u001B\u6465\u001B        (=de)
```

C 言語では、この 2 個の ASCII キャラクタはまとめて 1 個の Unicode 値とする必要があります。*charref=true* の場合、この並びは以下のように表現できます：

```
&#x001B;&#x6465;&#x001B;    (=de)
```

テキストフローを持つタグ付き PDF を生成　テキストフロー機能（235 ページ「9.2 複数行のテキストフロー」参照）は、テキスト組版のための強力な機能群を提供します。個別のテキストフラグメントはクライアント制御下ではなくなり、PDFlib によって自動的に組版されますので、テキストフローを持つタグ付き PDF を生成する際にはいくらか注意を払う必要があります：

- 単一のテキストフローはめ込み枠の内容全体を構造エレメントの一部分とすることはできます。しかし、テキストフロー枠が個別の構造エレメントを内容とすることはできません。
- 1 個のテキストフローのすべての部分（ある特定のテキストフローハンドルでの *PDF_fit_textflow( )* へのすべての呼び出し）は、単一の構造エレメントの中に含まれるべきです。
- 1 個のテキストフローの各部分が複数のページにわたる場合があり、それらのページは他の構造アイテム群を含んでいる可能性がありますので、適切な親アイテムを選択することに注意を払うべきです（*parent* オプションとして -1 を用いるのではなく。そうしてしまうと、誤った親エレメントを指し示すおそれがあります）。
- テキストフロー内にリンクやその他の注釈を作成するために範囲枠機能を用いる場合、この注釈を構造ツリー内に正しく位置付けることは困難です。

単語間を空白キャラクタで区切る　単語は空白キャラクタ（U+0020）で区切るべきです。*autospace* オプションを用いると、テキスト出力関数群のうちのいずれかへのそれぞれの呼び出しの後に、自動的に空白キャラクタを生成させることができます。

ハイフネーション　ハイフネーション、すなわち 1 つの単語を行末で 2 つの部分に分割することは、ハードハイフン（U+002D）ではなく、ソフトハイフンキャラクタ（U+00AD）を用いて表現される必要があります。このソフトハイフンキャラクタが用いられていれば、Acrobat は、テキストを検索する際に、ハイフネーションされた単語を正しく再結合（デハイフネーション）することができます。テキストがソフトハイフン U+00AD を含んでいるときには、PDFlib のテキストエンジンは、U+00AD に対するグリフがそのフォント内で得られる場合にはそれを、そうでないなら U+002D を用います。そのフォントが U+00AD と U+002D に対して別々のグリフを持っている場合には、ソフトハイフン U+00AD をテキスト内で使用すれば、ハイフネーションのためのタグ付き PDF の要件を満たすには充分です。

そのフォントが U+00AD に対する別個のグリフを持っていない（あるいは別のハイフネーションキャラクタが用いられている）場合には、そのハイフンを、U+00AD を内容とする *ActualText* 属性を持った *Span* か *ASpan* でタグ付けする必要があります（ただし PDF 1.4 ではこれは可能ではありません）。テキストフローを用いて複数行テキストが作成される際には、この *ActualText* は、指定された *hyphenchar* に自動的に割り当てられます（かつ *autospace* は抑止されます）。

テキスト行に対して必要な *ActualText* を付けることは以下のように達成できます：

- *PDF_fit_textline( )* でオプション *tagtrailinghyphen* を指定すれば、然るべき *ActualText* が付き、かつ *autospace* が抑制されます。このオプションのデフォルトは U+00AD ですので、そのテキストがハイフンとして U+00AD を用いている場合には、正しいタグ付けが自動的に行われます。
- フォントが U+00AD に対する別個のグリフを持っていない場合には、然るべきフォールバックフォントからのソフトハイフンを用いてこれを修正することもできます。以下のフォント読み込みオプションを用います：

```
fallbackfonts={{fontname=AuxiliaryFont encoding=unicode embedding forcechars=x00AD}}
```

- 然るべき *ActualText* を手動で割り当てるには以下のようにします。ただし、これが必要なのは、そのフォントが U+00AD と U+002D に対して 2 個の別々のグリフを持っていない場合のみであることに留意してください（このコードは PDF 1.5 以上でのみ動作します。PDF 1.4 の場合にはオプション *direct=false* を追加してください）：

```
p.set_option("charref=true");
p.fit_textline("-", x, y, "tag={tagname=ASpan ActualText=&#xOOAD;}");
```

11.3.5 代替記述・置換テキスト・略語拡張

タグ付き PDF は、追加情報なしでは容易に読めない画像・テキストのアクセシビリティを向上させる機能群を提供しています。

代替記述（Alt） テキストへ自然に翻訳されないアイテムに、代替記述を *Alt* オプションを通じて割り当てることができます。たとえば画像・数式や、*contents* オプションを持たない注釈などです。

この代替記述は、スクリーンリーダが読める 1 個のまるごとの単語ないし句から成るべきです。この記述の末尾は、スクリーンリーダがそれを後続テキストと合体させてしまわないよう、ピリオドか空白キャラクタかいずれか適切なほうとするべきです。**「この画像の内容は ...」**といった先頭句を代替記述に入れることは避けることを推奨します。この *Alt* 値は、その構造エレメントとそのすべての子の記述を提供します。*ASpan* 擬似エレメントを用いると、構造エレメントの一部分に *Alt* を割り当てることができます。

たとえば、企業ロゴの画像を、*Alt* オプションを通じて記述してもよいでしょう：

```
p.fit_image(image, x, y, "tag={tagname=Figure Alt={Kraxi企業ロゴ }}")
```

置換テキスト（ActualText） テキストへ変換されるアイテムであっても、それが何らかの非標準的な方式で表現されている場合には、それに代替テキストを *ActualText* オプションを通じて割り当てることができます。たとえばスワッシュキャラクタやドロップキャップを持った図や、ピクセル群を用いて単語を表現している画像などです。一方で、スキャンされたページに対する OCR テキストは、*ActualText* ではなく不可視テキスト（すなわち *textrendering=3*）として与えるべきです。

この代替テキストは、人がそのコンテンツを見た時に見えるものと同等の 1 個ないし複数のキャラクタを内容とするべきです。この *ActualText* 値は、その構造エレメントとそのすべての子に対する代替としての役割を果たします。*ASpan* 擬似エレメントを用いると、構造エレメントの一部分に *ActualText* を割り当てることができます。

たとえば、対応する Unicode 値が全く得られない記号 *flower* グリフに対して、適切な *ActualText* を割り当てることで、このグリフが実際にはビュレットキャラクタ U+2022 として用いられていることを明らかにすることができます：

```
p.fit_textline("&.flower;", x, y, "tag={tagname=ASpan ActualText={&#x2022;} } ...");
```

代替テキストと置換テキストに対するネスト規則 *Alt*・*ActualText* 属性を用いる際には、以下の規則に従う必要があります：

- *Alt* と *ActualText* は、対象構造エレメント配下の下位ヒエラルキー全体を覆いますので、その構造エレメントの構造ヒエラルキー内のいずれかの祖先がすでに *Alt* 属性か *ActualText* 属性を含んでいる場合には、どちらの属性も許容されません。
- *Alt*か*ActualText*属性を持つエレメントは、コンテンツアイテムか、1個ないし複数の*Link* エレメントを内容として持つ必要があります。この属性が *Link* エレメントに適用され

る場合には、それは、コンテンツアイテムか、*PDF_create_annotation()* によって作成された 1 個ないし複数の *OBJR* エレメントを内容として持つ必要があります（313 ページ「リンクとその他の注釈種別」参照）。そうでないと、その属性がどのページ上で読み上げられるべきなのかを決定することが不可能になります。この規則は当該エレメント自体に適用されます：コンテンツアイテムを持つ子エレメントを持っていても充分ではありません。

Acrobat における代替・置換テキスト　構造エレメントの *Alt*・*ActualText* 属性は、Acrobat X/XI/DC で以下のように表示できます：

- **「表示」→「表示切り替え」→「ナビゲーションパネル」→「タグ」**を選択し、ヒエラルキー内の構造エレメントを右クリックして、**「プロパティ ...」**を選択すると、オブジェクトプロパティダイアログが表示されます。**「タグ」**タブに、**「実際のテキスト」**・**「代替テキスト」**属性が表示されています。

略語拡張（E）　略語と頭字語には、拡張テキストを、*tag* オプションの *E* サブオプションで割り当てることができます。この拡張テキストは、スクリーンリーダが読める 1 個のまるごとの単語ないし句から成るべきです。略語が拡張テキストを何も持たない場合でも、*E* 属性を、テキストから読み上げへの変換処理を支援するために与えることができます。たとえば、用語 *IBM* は、拡張テキスト *I B M*（空白キャラクタをはさんで）をそれに割り当てられて持つことができます。

以下のコード断片では、拡張テキスト *Mister* が略語 *Mr.* に割り当てられています：

```
p.fit_textline("Mr.", x, y, "tag={tagname=ASpan E={Mister} } ...");
```

11.3.6 印刷ストリーム順序と論理読み取り順序

PDF には、内容の順序付けについて、根本から異なる 2 種類の概念があります。図 11.3 に、2 段組のメインテキストに、表 1 個がはさまり、灰色背景上のサマリ 1 個が挿入された、さらにヘッダとフッタもあるサンプルページを図示します。

PDFlib API 関数呼び出しの順序は、ページコンテンツストリーム内の PDF テキスト・描画演算子の順序（Acrobat では「生の印刷ストリーム順序」といいます）を決定付けます。ページコンテンツは、制御アプリケーションに都合のいい任意の方式で作成されうるものですので、これはただの技術的な順序付けであり、意味付け上の重要性は必ずしもありません。この印刷ストリーム順序は Acrobat の**「順序」**・**「コンテンツ」**パネルで表示されます。

論理読み取り順序は、人がテキストを読む順序です。これは、スクリーンリーダと Acrobat の読み上げ機能によって用いられる順序付けを決定します。この論理読み取り順序は、論理構造ツリーによって決定されます。タグ付き PDF では、すべてのコンテンツが意味付け上正しい順序でタグ付けされていることが必須であり、すなわち、その構造ヒエラルキーはそのページコンテンツを人が読む順に含んでいる必要があります。正しいタグ付けにより、スクリーンリーダがコンテンツを論理順に表現できるようになります。ページ装飾は構造ツリーの一部分ではありませんので、論理読み取り順序からは除外されています。

Acrobat における読み取り順序と印刷ストリーム順序　論理読み取り順序は、Acrobat X/XI/DC で以下の方式でチェックできます：

- **「表示」→「表示切り替え」→「ナビゲーションパネル」→「タグ」**を選択し、エレメントの順序を上から下へチェックします。この順序付けは、望む読み取り順序を正確に反映しているべきです。

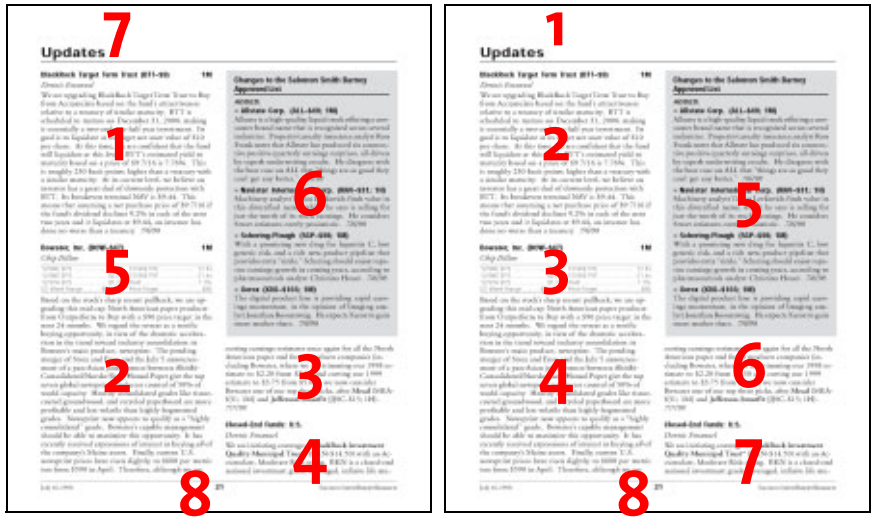


図 11.3
論理読み取り順序（左）と印刷ストリーム順序（右）

- **「表示」→「読み上げ ...」**を有効にすると、Acrobat にページ内容を、その文書内で指定されている読み取り順序で読み上げさせることができます。

「順序」・「コンテンツ」パネルでは、ページコンテンツを印刷ストリーム順序で一覧表示しています。

コンテンツを論理読み取り順序で作成　多くの状況においてうまくいく自然な方式は、1個の構造エレメントのすべての構成部分をシーケンシャルに生成し、ついで論理シーケンス内の次のエレメントへ移っていくことです。専門的にいうと、構造ツリーは単一の深さ優先トラバーサルによって作成され、ページ内容を作成する PDFlib 関数は、内容が読まれる順に呼び出されます。

PDFlib では、コンテンツを任意の順序で作成しながらもなお構造ヒエラルキーを論理順に作成するためのいくつかの方法が可能です。これらの方式は 318 ページ「11.4.4 コンテンツを順序にとらわれず作成」で説明します。

11.3.7 Adobe Acrobat におけるタグ付き PDF の不具合

この項では、私達がタグ付き PDF 出力を Adobe Acrobat で試験していくなかで得た知見を述べます。表 11.5 に、Acrobat DC におけるバグと動作不安定を、以下の機能ごとにまとめて挙げます：

- 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。
- Acrobat の読み上げ機能：タグ付き PDF は、Acrobat の読み上げ機能を向上させます。
- Acrobat のタグパネル内の**「検索 ...」**機能を使ってページ装飾とマークなしコンテンツを検索することもできます。

表 11.5 タグ付き PDF に関連する Acrobat DC の不具合

説明・推奨・回避策
***Acrobat*のアクセシビリティチェッカ**
`Alt` 属性が `Figure` タグについて無視される。
アクセシビリティチェッカが、チェックマーク記号を持った種類チェックボックスのフォームフィールドについて「Character encoding – failed」と警告を発するが、読み上げ機能はそのフィールド内容を完璧に読み上げる。
***Acrobat*の読み上げ機能**
タグ付きのページが PDI で配置されており、かつページ装飾のみを内容としている場合にも、読み上げ機能がその配置されているページのコンテンツを読み上げる。
その他の*Acrobat*の機能
タグパネル内の「検索 ...」機能が、ページ装飾を誤ってマークなしコンテンツとして報告する。例：表装飾。対照的に、これらのアイテムはコンテンツパネル内では「コンテナ〈ページ装飾〉」として正しく表示される。

11.4 タグ付き PDF の高度なトピック

11.4.1 自動表タグ付け

PDF_fit_table() は、表コンテンツのために *PDF_add_table_cell()* に与えられた情報に基づいて、生成される表のために適切なタグを自動的に作成することができます。*PDF_fit_table()* の tag オプションで *tagname=Table* を用いると、表 11.6 に説明するとおりの自動表タグ付けが行われます。

表 11.6 PDF_fit_table( ) の tag オプションと自動タグ付け

PDF_fit_table( ) の tag オプション	結果
tagname=Table	自動表タグ付けが行われます。tagname=Table のかわりに、Table へロールマップされたカスタムタグを用いることもできます。
tagname=Artifact	キャプションを含め、表コンテンツ全体がページ装飾としてマークされます。BBox 属性が自動的に追加されます。
他の任意の *tagname*	Table 構造は一切作成されず、そのセルコンテンツは tag オプションで指定されたエレメントの子として追加されます。擬似アイテムとインラインアイテムは PDF_fit_table( ) に対しては許容されません。
***tag*オプションが与えられていない**	自動タグ付けなし。tag オプションが個別の PDF_add_table_cell( ) への呼び出しに対して与えられた場合には（fit* オプションのうちのいずれかの対するサブオプションとして）、そのセルコンテンツに対する、照応する構造エレメントが作成されますが、表構造は一切作成されません。

注 自動表タグ付けは、PDFlib の表エンジンが用いられる場合にのみ威力を発揮します。手作業で作成された表に正しくタグを付けることは可能ではありますが、この処理は、クライアントアプリケーション側で表構造に関する詳しい知識を必要とします。表行 / 列構造のみならず、ヘッダセルと表行 / 列スパンに関する関連情報が必須です。空セルにもタグを付ける必要があります。

Acrobat でテーブルタグを視覚化　テーブルエレメントの構造は、Acrobat X/XI/DC で以下のように視覚化できます：

- **「表示」→「アクセシビリティ」→「読み上げ順序」**（Acrobat DC）または **「*TouchUp* 読み上げ順序」**（Acrobat X/XI）を選択。テーブルエレメントがハイライトされ、テーブルの左上隅近くの小さな番号で標識されます。テーブルサマリが存在する場合には、それはこの小さな番号の後に表示されます。
- テーブルの左上隅にある番号または構造種別名を選択し、**「読み上げ順序」**ダイアログ内の **「テーブルエディター」**をクリック。表構造が縦横の線で視覚化されます。**「テーブルエディターオプション」**ダイアログ内で、Acrobat に、表セルの種別に従って *TH/TD* アイコンを表示させることができます（図 11.4 参照）。
- 表セルを右クリックして、**「テーブルセルのプロパティ ...」**を選択すると、そのセル種別（ヘッダセルなら *TH*、データセルなら *TD*）、*scope* 属性、*rowspan*・*colspan* 属性、*header/ID* 値をチェックできます。

図 11.4
タグ付き PDF 表のための Acrobat のテーブルエディターがヘッダ（TH）・データ（TD）セルを表示

なお、Acrobat においてテーブルエディターで作業するにあたっては以下の制約があります：

- 表セルを視覚化する線が、誤った位置に表示されることがあります。
- 表が、*Table* へロールマップされたカスタム構造エレメント（標準エレメント *Table* ではなく）を用いていると、Acrobat はテーブルエディターをアクティブにしません。
- テーブルエディターは、表内に *Caption* エレメントが存在していても表示しません。
- 表セルが縦書きテキスト（たとえば *orientate=east* または *west* によるテキスト行）を含んでいると、このセルとその右隣セルはテーブルエディターで表示されませんが、ただしそれらは論理構造ツリー内には存在し、それらの内容はページ上で見えます。

自動作成される表タグと属性　自動表タグ付けは、**ページ**スコープ内でのみ働き、以下のように動作します：

- それぞれの表インスタンスに対して別々の *Table* エレメントが作成されます。たとえば、1 個の表が 2 個以上のインスタンスに分割されている場合、複数の *Table* エレメントが作成されます。*PDF_fit_table( )* の *tag* オプションに *Summary* サブオプションが与えられている場合には、*Table* 要素に *Summary* 属性が追加されます。
- *PDF_fit_table( )* で *caption* オプションが指定された場合には、*Caption* エレメントが作成されます。グループ化エレメントとして、*Caption* はいかなるコンテンツアイテムをも許しませんので、この *caption* オプションの *tag* サブオプションを与えることによって、*Caption* の子として構造エレメントを指定する必要があります。このエレメントがこのキャプションの実際のコンテンツを保持できます。
- それぞれの表行に対して 1 個の *TR* エレメントが作成されます。*PDF_fit_table( )* の *header* オプション内で指定されている表行は *THead* でラップされ、*footer* オプション内で指定されている表行は *TFoot* でラップされます。それ以外のすべての表行は、ヘッダかフッタが存在する場合には *TBody* でラップされます。
- 表セルはそれぞれ、*PDF_add_table_cell( )* の *tag* オプションの *tagname* サブオプションに従って、*TH*（テーブルヘッダ）か *TD*（テーブルデータ）のいずれかのエレメントでラップされます。このオプションが指定されていない場合には、そのセル種別は以下のように選択されます：
  - セルに対する *Scope* 属性は *TH* を強制します（たとえ *tagname=TD* が指定されていても）。

- そのセルの *id* を指定した *Headers* オプションを別のセルが含んでいる場合には、このターゲットセルは強制的に *TH* になります（たとえ *tagname=TD* が指定されていても）。
- そのセルが、*PDF_fit_table()* の *header* オプションで指定されているとおりの表ヘッダの一部分である表行に含まれている場合には、それは *TH* でラップされ、また、*Scope=Column* が追加されます。

- *PDF_add_table_cell()* が呼び出されていない表セルそれぞれについて、空の *TD* ダミーエレメントが作成されます。
- *TH*・*TD* エレメントは、*PDF_add_table_cell()* の *rowspan*・*colspan* オプションに従って、適切な *RowSpan*・*ColSpan* 属性を得ます。*tag* オプションの *RowSpan*・*ColSpan* サブオプションは使えません。
- *Table*・*TH*・*TD* エレメントには、適切な *Width*・*Height* 属性が割り当てられます：*Table* エレメントには *BBox* 属性も割り当てられます。
- 他の表セル属性は、*PDF_add_table_cell()* の *tag* オプションに対してサブオプションとして与えることが可能です。以下のオプションは許容されません：*RowSpan*・*ColSpan*・*Height*・*index*・*parent*・*Width*。
- 表行とセルは、*PDF_add_table_cell()* への呼び出しの順序にかかわらず、左上セル（すなわち列 1・行 1）から右下セルへジグザグ順に出力されます。
- 装飾的な各種表要素は、自動的に *artifacttype=Layout* の *Artifact* としてタグ付けされます。すなわち、表セル・表行・列・表全体の罫線とシェーディング（塗り / 描線）、範囲枠の塗りと罫線、*showborder* の罫線、*debugshow*・*showcells*・*showgrid* を通じて制御される視覚化支援です。

クックブック **自動表タグ付けのためのコードサンプルが、PDFlib クックブックの** `pdfua` **カテゴリ内の** `tagged_table`・`invoice_pdfua1` **トピックにあります。**

表セルにタグと属性を追加　短縮タグ付けを、表キャプションか表セルか、または表セルのコンテンツに適用することができます。*PDF_add_table_cell()* に *tag* オプションを与えると、以下の状況において有用です：

- 表セルが、ヘッダ行に含まれていないか、*Scope* 属性を持たない場合には、*tagname=TH* を用いてそれを強制的にヘッダセルにすることもできます（TD データセルではなく）。
- リンクに対して必要な正しいタグ構造を作成：詳しくは 315 ページ「表セル内でリンクにタグ付け」を参照してください。

PDF_add_table_cell() の *tag* オプションのサブオプションに対しては、以下の制約が課されます：

- 以下のオプションは使えません：*ColSpan*・*Height*・*index*・*parent*・*RowSpan*・*Width*。
- *tagname* オプションは、表セルの種別を指定するために、値 *TH* か *TD* のいずれかのみを持つことができます。しかし、*tag* オプションをネストすることによって子孫タグを指定することもできます。
- *id* オプションは、1 個の文書内で一意でなければなりませんので、このオプションは、複数の表インスタンス内で反復される表セル、たとえば、複数の表インスタンスを作成する表のためのヘッダ行かフッタ行の中のセルに対しては許容されません。

表セルのコンテンツにタグと属性を追加　*PDF_add_table_cell()* の以下のオプション（または *caption* オプションの照応するサブオプション）にサブオプションとして *tag* を与えることもできます：

fitannotation・fitfield・fitgraphics・fitimage・fitpath・
fitpdipage・fittextflow・fittextline

これは以下の状況において有用です：

- 表セルのコンテンツに対して下位構造を指定。この *tag* オプションは、その表セルの *TH* か *TD* エレメントの子エレメントを作成します。自動表タグ付けが有効の場合には、*tag* オプションに対して、*tagname* について以下の値は許容されません（言い換えれば、表のネストには対応していません）：

  Table・TR・TH・TD・THead・TBody・TFoot

- 以下に挙げる表の属性は、自動的には作成されることができませんが、アクセシビリティ目的のために必要な場合があります。これらはユーザーによって与えられる必要があります：
  - 表内の 1 個ないし複数の *TH* セルを参照する *TD* セル（すなわちこれは、*PDF_fit_table()* の *header* オプションの一部分ではないヘッダセルを参照します）に対しては、*Headers* オプションを与える必要があります。
  - いずれかの *Headers* オプション内で参照されている *TH* セルに対しては、*Id* オプションと *Scope=Row* オプションを与える必要があります（TH 列ヘッダセルに対しては *Scope=Column* は自動的に作成されます）。
- キャプションは任意のコンテンツを内容とすることができ、そのコンテンツ自体もタグ付けされることができます。たとえば、以下のオプションリストは、ネストされた *P* エレメントの中のテキスト行 1 個を内容とする *Caption* エレメントを作成します：

```
caption={ fittextline={tag={tagname=P title={出張旅費報告書}} ... } ... }
```

11.4.2 インタラクティブ要素にタグ付け

リンク・注釈・フォームフィールドも、アクセシブルにする必要があります。その照応するインタラクティブエレメントを、構造ツリー内で、構造ツリー内の正しい位置に表現する必要があります。インタラクティブ要素をページ装飾として作成することはできません。

クックブック **タグ付きのリンクを作成するためのコードサンプル群が starter_pdfua1 サンプルにあります。image_with_link_pdfua1 トピックは、背景画像付きのリンクを生成します。table_of_contents_pdfua1 トピックは、目次とともに、TOC・TOCI 構造エレメントと、動作するリンクを生成します。**

リンクとその他の注釈種別　注釈は、アクセシビリティのために以下のアイテムを要請します（図 11.5 参照）：

- *Link* エレメント（リンク注釈に対して）または *Annot* エレメント（その他すべての注釈種別に対して）は、次の 2 つのアイテムのためのコンテナとして働きます。*Alt* または *ActualText* オプションを与えることによって代替記述か置換テキストを与えることもできます。*Link* エレメントの *Alt* 属性はそのリンクの目的を記述するべきです。リンクのターゲットがカレント文書内に位置している場合には（*GoTo* アクション）、その *Link* エレメントはさらに *Reference* エレメントの内容となるべきです（オプション *direct=false* を用いて）。
- インタラクティブ要素を表現するテキストを、このコンテナエレメントの内側に作成するべきです。テキストが全く必要ない場合には、このエレメントはスキップするこ

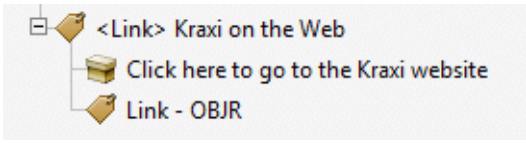


図 11.5
構造ツリー内における
アクセシブルなリンクの表現

とができます。たとえばテキスト注釈の場合などです。注釈がラスタ画像かベクトルグラフィックによって表現されている場合には、これは Artifact としてタグ付けされるべきです。この場合、このリンクの代替テキストは、そのグラフィックとリンクの両方を記述することになります。さまざまなコンテンツ作成関数の *matchbox* オプションを用いることによって、次のエレメントのための形状情報を用意することを推奨します。*matchbox* オプションを与える時点におけるカレントのアクティブなエレメントは構造アイテムであるべきです。すなわち、*structureitem=false* オプションは許されません。

- 注釈は、*PDF_create_annotation( )* を用いて作成する必要があります。注釈に加えて、この関数は、その注釈のための、種別 *OBJR*（オブジェクト参照）の、照応する構造エレメントを生成します。リンク注釈に対しては、*PDF_create_annotation( )* の *contents* オプションを与えるべきです（これは PDF/UA-1 では必須です）。その他の注釈種別については、*PDF_create_annotation( )* の *contents* オプション、または *ActualText* タグ付けオプションを用いて作成するべきです。*usematchbox* オプションを用いると、2 番目のステップで作成された視覚コンテンツの形状を簡便に与えることができます。

この 2 番目と 3 番目のアイテムは、どちらを先に作成してもかまいません。上記の注釈の要件は、*PDF_add_table_cell( )* のオプション *fitannotation* を用いて表セル内に作成される注釈に対してもあてはまります。

以下のコード断片は、この必須の 3 つのアイテムを持ったインタラクティブリンクを作成します（生成されるタグ構造を図 11.5 に示します）：

```
/* 親Linkエレメントを作成 */
id_link = p.begin_item("Link", "Title={Kraxi on the Web} Alt={Kraxi on the Web}");

/* このリンクを表現する可視コンテンツを作成 */
p.fit_textline("Click here to go to the Kraxi website", x, y,
        "matchbox={name={kraxi}} fontsize=14 font=" + font);

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

/* 名前付き範囲枠「kraxi」上にLink注釈を作成 */
p.create_annotation(0, 0, 0, 0, "Link",
        "action={activate=" + action + "} "
        "usematchbox={kraxi} contents={Kraxi Inc. Webサイトへのリンク}");

p.end_item(id_link);
```

注 インタラクティブ要素のために必要なこのタグ付け手順は、テキストフローと範囲枠を用いた場合には実現できません。なぜならタグはテキストフロー内に作成できないからです。また、テキストフローを配置した後にインタラクティブ要素のためのタグを作成すると読み取り順序がおかしくなります。

表セル内でリンクにタグ付け　表セル内のリンクは、上述のタグ構造を必要とします。ただしこれは、構造要素 *TH/TD・Link・OBJR* と、さらにおそらくコンテンツを、正しくネストする必要がありますので、やや複雑かもしれません。これを達成するには、*PDF_add_table_cell()* に *tag* オプションを与えてそのネスト機能を利用する必要があります。以下のオプションリストは、表セルにテキスト 1 行とリンク注釈 1 個を入れます。それを囲う *TD* エレメントは外側の *tag* オプションで与えられ（*TD* は表エンジンによって自動的に作成されますので、外側の *tagname* サブオプションは省略できます）、そして *Link* 要素は内側の *tag* オプションで与えられます。最後に、必要な *OBJR* エレメントは、*PDF_create_annotation()* と同等の働きをする *fitannotation* オプションによって自動的に作成されます：

```
fittextline={font=... fontsize=25 fillcolor=blue}
annotationtype=Link fitannotation={contents={Kraxi home page} action={activate ...}}
tag={tagname=TD tag={tagname=Link}}
```

フォームフィールドにタグ付け　フォームフィールドはアクセシビリティのために以下の構造エレメントを必要とします：

- 関連するフォームフィールドのグループを、*Part* 構造エレメントを用いて内包することができます。
- 個々のフォームフィールドに引き続く構成要素群を内容とする *Div* 構造要素を推奨します。
- ページ上でフィールドの目的を記述するテキストを内包する *P* 構造エレメントを持った *Caption* 構造エレメント。
- チェックボックスとラジオボタンに対しては、そのチェックボックス／ラジオボタンとその照応するキャプションを内包するもう 1 つの *Div* エレメントを作成することを推奨します。
- *Form* 構造エレメントは、次のエレメントのためのコンテナとして働きます。*Alt* または *ActualText* オプションを与えることによって代替記述か置換テキストを与えることもできます。なお、擬似エレメント、テーブルエレメント、ILSE、ルビ・割注エレメントを *Form* の親にすることは許容されないことに留意してください。ラジオボタンを作成する際には、*Form* エレメントは、そのラジオボタングループのための *PDF_create_fieldgroup()* では必要ではなく、個々のラジオボタンを作成するための *PDF_create_field()* でのみ必要です。
- そのフォームフィールドのための、種別 *OBJR*（オブジェクト参照）の構造エレメントが、*Form* 構造エレメント内にネストされて、*PDF_create_field()* によって自動的に作成されます。*PDF_create_field()* の *tooltip* オプションを与えることにより、フィールドのアクセシビリティを向上させるべきです（これは PDF/UA-1 では必須です）。

以下のコード断片は、テキストフィールドに対して推奨される構造を、図 11.6 に示すとおりに生成します。*PDF_fit_textline()* の *tag* オプションを使用することによるネストされた短縮タグ付けを用いて、ネストされた *Caption/P* 構造エレメントが生成されます：

```
id_Div = p.begin_item("Div", "");

labeltext = "Enter name:";
p.fit_textline(labeltext, x1, y1, "tag={tagname=Caption tag={tagname=P}}");

optlist = "tag={tagname=P tag={tagname=Form}} tooltip={" + labeltext + "} " +
        "bordercolor={gray 0} font=" + font;
```

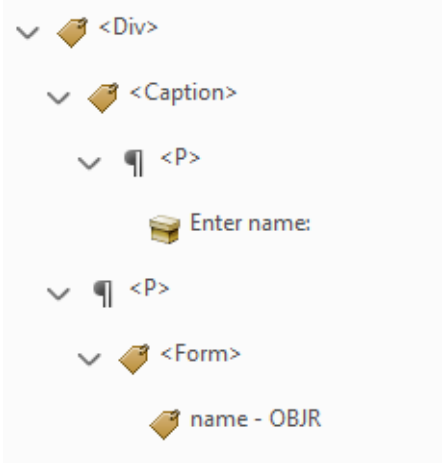


図 11.6
構造ツリー内におけるフォームフィールドの表現

```
p.create_field(x2, y2, x3, y3, "name", "textfield", optlist);

p.end_item(id_Div);
```

上記のフォームフィールドの必要事項は、*PDF_add_table_cell()* のオプション *fitfield* を用いて表セル内に生成されるフォームフィールドにもあてはまります。

クックブック **すべての種類のタグ付きフォームフィールドを作成するコードサンプルが、PDFlib クックブックの `pdfua` カテゴリの `form_fields` トピックにあります。**

タグ付きしおり　しおりには、通常の移動先に加えて、構造エレメントを割り当てることもできます。このようなしおりを**タグ付きしおり**といい、Acrobat はこれのために追加の機能を提供しています。Acrobat でタグ付きしおりを右クリックすると、機能**「ページを削除」**と**「ページを抽出」**が利用可能となっており、これらはその構造化エレメントを含んでいるページないしページ群に対して動作します。タグ付きしおりは、しおりと構造エレメントとの間に接続を作成します。この接続を作成できる方法は 2 つあります：

- *PDF_create_bookmark()*　でしおりを作成し、そのハンドルを、*PDF_begin_item()*　の *bookmark* オプションに、またはさまざまな関数の *tag* オプションに与える：

```
bm = p.create_bookmark("第1章", "");
id = p.begin_item("H1", "Title={第1章} bookmark=" + bm);
p.fit_textline(text, x, y, "");
p.end_item(id);
```

この方式は短縮タグ付けでも用いることができます：

```
bm = p.create_bookmark("第1章", "");
p.fit_textline(text, x, y,
     "tag={tagname=H1 Title={第1章} bookmark=" + bm + "}");
```

この方式の難点は、しおりテキストが、その構造エレメントとそのコンテンツが作成される前には得られていなければならないという点です。これは、参照される構造エレメントのほうが構造ツリー内で上にあった場合には困るでしょう。

- *PDF_begin_item()* で構造アイテムを作成し、そのハンドルを *PDF_create_bookmark()* の *item* オプションに与えます。アイテムのハンドルのかわりに、キーワード *current* を、*PDF_create_bookmark()* が呼び出される時点におけるカレントの構造エレメントを指し示すショートカットとして与えることができます：

```
id = p.begin_item("H1", "Title={第1章} ");
bm = p.create_bookmark("第1章", "item=current");
p.fit_textline(text, x, y, "");
p.end_item(id);
```

この方式には、しおりテキストが、その構造エレメントとそのコンテンツが作成される時点で得られればよいという利点があります。ただし、これは短縮タグ付けでは使えません。

タグ付きしおりは、オープン構造エレメントを参照することのみ可能であり、擬似アイテムまたはインラインアイテムを参照することはできません。しおりの移動先が構造エレメントと一致するようにするのはクライアントコード側の役割です（そうしないと、Acrobat でしおりをクリックしたらそのエレメントへジャンプせず、その文書内の別の場所へジャンプしてしまいます）。関連付けられる構造エレメントが複数のページにわたっている場合には、そのしおりはこの範囲内の最初のページを指し示すべきです。

11.4.3 箇条書き

箇条書きは、関連する項目群をまとめるために使われます。これは以下の構造エレメントによって表現されます（図 11.7 参照）：

- 以下のすべての構造エレメントを内容とする *L* エレメント 1 個。*ListNumbering* オプションを用いると、*Lbl* エレメント内で用いられる付番系列を指定することができます。この *ListNumbering* オプションは、*Lbl* エレメントがない場合でも、スクリーンリーダのために有用でしょう。
- オプショナルな *Caption* エレメント 1 個。*Caption* はグループ化エレメントですので、コンテンツアイテムを内容とすることはできず、他の構造要素のみを内容とすることができます（*P* など）。
- 以下を内容とする、1 個ないし複数の箇条書き項目（*LI*）：
  - オプショナルな、ビュレットや番号などを持ったラベル（Lbl）1 個。
  - その箇条書き項目の実際のコンテンツを持った *LBody* エレメント 1 個。*LBody* は、コンテンツアイテムか、ネストされた箇条書きも含めて他の構造エレメントのいずれかを内容とすることができます。

以下のコード断片は、1 個のキャプションと 3 個の項目を持った箇条書きを作成します。それぞれの箇条書き項目の頭には、ラベルとしてタグ付けされるビュレットキャラクタ U+2022 が付きます（生成されるタグ構造を図 11.7 に示します）：

```
id_list = p.begin_item("L", "ListNumbering=Disc");

/* Captionエレメントと Pエレメントの両方を一回で生成 */
p.fit_textline("The following kinds of fruit are available:",
        x1, y, "tag={tagname=Caption tag={tagname=P}}");
    y -= leading;

id_listitem = p.begin_item("LI", "");
    p.fit_textline("&#x2022;", x1, y, "tag={tagname=Lbl}");
```

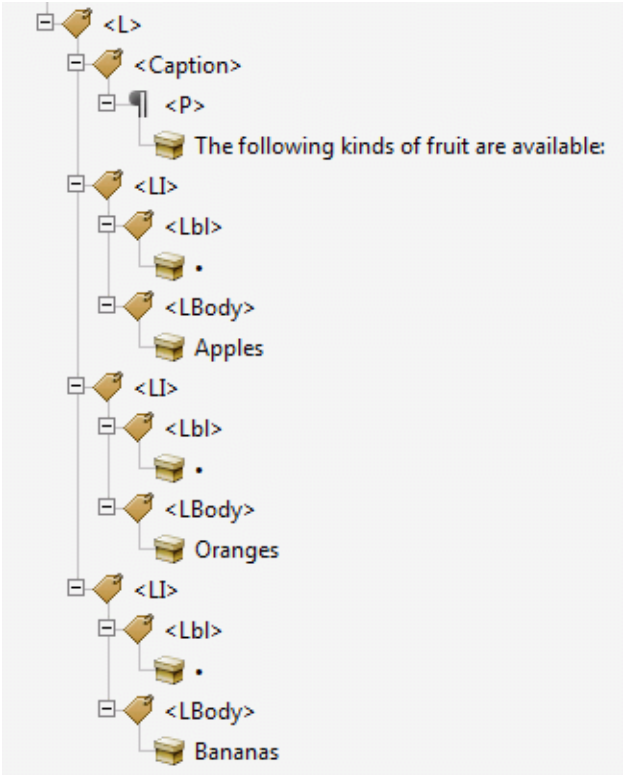


図 11.7
アクセシブルな箇条書きの構造ツリー内における表現

```
    p.fit_textline("Apples", x2, y, "tag={tagname=LBody}"); y -= leading;
p.end_item(id_listitem);

id_listitem = p.begin_item("LI", "");
    p.fit_textline("&#x2022;", x1, y, "tag={tagname=Lbl}");
    p.fit_textline("Oranges", x2, y, "tag={tagname=LBody}"); y -= leading;
p.end_item(id_listitem);

id_listitem = p.begin_item("LI", "");
    p.fit_textline("&#x2022;", x1, y, "tag={tagname=Lbl}");
    p.fit_textline("Bananas", x2, y, "tag={tagname=LBody}"); y -= leading;
p.end_item(id_listitem);

p.end_item(id_list);
```

クックブック タグ付き PDF 箇条書きを作成するためのコードサンプルが、PDFlib クックブックの pdfua カテゴリ内のトピック list_pdfua1 にあります。

11.4.4 コンテンツを順序にとらわれず作成

307 ページ「11.3.6 印刷ストリーム順序と論理読み取り順序」で述べたように、構造ヒエラルキー内の要素群を作成する際に論理読み取り順序のとおりにすることは不可欠です。アプリケーションが、論理読み取り順序とは異なる順序でページコンテンツを（たとえば、段組の相互関係にかかわらずつねに上から下へ）処理する場合には、いくつかの PDFlib 機能を用いて、構造ヒエラルキー内で正しい順序付けを保つことができます：

- 子エレメント群を順序にとらわれず作成するために *index* オプションを用いる。これは、新規構造エレメントがその親の中へ挿入される位置を変更します。
- 構造エレメント群を順序にとらわれず作成するために *parent* オプションを用いる。これは、新規構造エレメントが挿入される親エレメントを変更します。
- 構造ヒエラルキー内で前後へジャンプするために *PDF_activate_item()* 関数を用いる。これを用いると、構造ヒエラルキー内のいずれかのエレメントに対して、構造エレメントまたはコンテンツアイテムをさらに追加することができます。

これらの方式について詳しくは後述します。

クックブック **エレメントを順序にとらわらずタグ付けするためのコードサンプルが、PDFlib クックブックの** `pdfua` **カテゴリ内のトピック** `out_of_order`・`parallel_columns_pdfua1` **内にあります。**

子エレメント群を順序にとらわれず作成　構造エレメント内の子エレメント群を順序にとらわれず作成するためには、構造ツリー内の場所を指定するために、*PDF_begin_item()* の *index* オプションか、あるいはさまざまな関数の *tag* オプションの *index* サブオプションを用いることができます。以下のコード断片は、テキストフラグメント群を逆順に出力しながら、構造ツリー内における順序を正しくするために、それぞれの新規テキストフラグメントを親エレメントの新たな最初の子（*index=0*）として挿入します。それぞれの新規エレメントが親の新たな最初の子として挿入されていますから、結果として論理順は作成順の反対になります：

```
p.fit_textline("三", x, y, "tag={tagname=P index=0}");
y += leading;
p.fit_textline("二", x, y, "tag={tagname=P index=0}");
y += leading;
p.fit_textline("一", x, y, "tag={tagname=P index=0}");
```

カレントでアクティブなタグの、その親エレメント内における番号をクエリするために、*PDF_get_option()* とその *activeitemindex* か *activeitemkidcount* キーワードを用いることができますので、後で構造ツリー内のこの場所へ戻ってくることが可能です。以下のコード断片は、新規エレメントを、保管していた番号の位置にあるエレメントの後に挿入します：

```
nextindex = p.get_option("activeitemindex", "") + 1;

...同一レベル上のエレメント群をさらに作成...

p.fit_textline(text, x, y, "tag={tagname=P index=" + nextindex + "}");
```

構造エレメント群を順序にとらわれず作成　子エレメント群を、カレント位置ではなく、構造ツリー内のどこか他の場所に作成するためには、*parent* オプションを用います。これは、まだ閉じられていない構造エレメントを指し示す必要があります。短縮タグ付けで作成されたエレメントは、同一の関数呼び出しの中で作成されて閉じられてしまいますので、これは *parent* オプションのターゲットとしては使えません。カレントでアクティブなタグの ID をクエリするために、*PDF_get_option()* とその *activeitemid* キーワードを用いることができますので、後で構造ツリー内のこの場所へ戻ってくることが可能です：

```
parent_id = p.get_option("activeitemid", "");
...
p.fit_textline(text, x, y, "tag={tagname=P parent=" + parent_id + "}");
```

さらに柔軟性を得る方法として、*parent* オプションと *index* オプションを組み合わせて使うこともできます。*PDF_suspend/resume_page()* を用いると、ページに割り込んで、別のページで作業を続け、その後その一時停止したページに戻ってさらにコンテンツを追加することができます。

複雑なレイアウトのためにアイテムをアクティブ化　複雑な非線形のページレイアウトのための構造情報の作成を支援するために、PDFlib はアイテムのアクティブ化という機能を提供しています。これを呼び出すと、開発者が複数の構造分枝の進捗を同時進行させ、かつそれぞれの分枝が 1 個ないし複数のページにわたる可能性があるような状況において、以前に作成された構造エレメントをアクティブ化することができます。この技法によって恩恵を受ける典型的な状況は以下のとおりです：

- 1 つのページ上に複数の段組
- メインテキストに割り込む挿入部、たとえばまとめやその他非線形のテキストアイテム
- 段組どうしの間に配置される表・イラスト

PDF_activate_item() 関数を用いると、構造ツリーの別々の分枝どうしの間を行ったり来たりすることが可能になります。「論理順」アプローチでは、クライアントアプリケーションはページコンテンツを論理順に構築する必要があり、これはたとえそれを視覚順に作成するほうが簡単であっても変えることはできません。これと対照的に、アイテムのアクティブ化を用いると、コンテンツを視覚順で作成することが可能になります（あるいはアプリケーションにとって便利ないかなる順序でも）。この技法は、コンテンツが複数ページにわたる場合にも適用できます。

Acrobat での諸問題を回避するため、*PDF_activate_item()* を呼び出した直後には、コンテンツアイテムを追加するべきではなく、他の構造エレメントだけを追加するべきです。

カレントでアクティブな構造エレメントをクエリ　*parent*・*index* オプションや *PDF_activate_item()* を使うためには、カレントでアクティブな構造エレメントとその子に関する知識がいくらか必要になります。このステータス情報は、アプリケーション側で把握しておくこともできますが、PDFlib からクエリすることも可能です。関数 *PDF_get_option()* でキーワード　*activeitemid*・*activeitemindex*・*activeitemkidcount*・*activeitemname*・*activeitemstandardname* を用いると、カレントエレメントの ID・番号・子エレメント数・名前・標準名（それがロールマップされている場合）が得られます。

11.4.5 タグ付き PDF ページを PDI で取り込む

クックブック　**タグ付き PDF 文書からページを取り込むためのコードサンプルが、PDFlib クックブックの pdfua カテゴリ内のトピック `clone_pdfua`・`merge_and_stamp_pdfua1` にあります。**

タグ付き PDF モードでは、タグ付き PDF 文書からのページは、その構造エレメントタグ群と一緒に取り込まれます。略称として、タグ付き PDF モードで *usetags=true* を用いて取り込まれたタグ付き PDF 文書からのページのことを「タグ付きページ」と呼ぶことにしましょう。このステータスをクエリするには、*PDF_info_pdi_page()* の *tagged* キーワードを用います。タグ付きページの取り込みは、以下の説明のように動作します。

タグ付き PDF 文書を開く　*PDF_open_pdi_document()* が、取り込まれる文書がカレントの PDF/A-1a/2a/3a または PDF/UA モードと互換かどうかをチェックしたうえで、取り込まれる文書の構造ツリーを読み取ります。

usetags オプションが *false* の場合には、その文書の構造情報は無視され、その文書からは何のタグも取り込まれません。なお、*PDF_open_pdi_document()* がオブジェクトスコープ内で、すなわち *PDF_begin_document()* の前に呼び出された場合には、*usetags* のデフォルトは *false* です。

注 属性クラスとクラスマップは取り込まれません。

入力文書言語を転写 もし PDF 文書の大半ないしすべてのページが、PDI を用いて取り込まれたものである場合には、その入力文書内にその文書言語エントリが存在しているならば、それを転写することを推奨します。この文書言語を転写するには、pCOS と以下のコード断片を用います：

```
if (p.pcos_get_string(indoc, "type:/Root/Lang").equals("string"))
{
        inputlang = p.pcos_get_string(indoc, "/Root/Lang");
        optlist += " lang=" + inputlang;
}

p.begin_document(filename, optlist);
```

タグ付きページを開く *PDF_open_pdi_page()* が、取り込まれたページの内容を構成する構造エレメント群を選択し、そのページ上に存在するタグをフィルタします。たとえば、しおりのためのタグは除去されます。なぜなら PDI はインタラクティブ要素を取り込まないからです。最後に、取り込まれたページ上の、取り込まれた構造下位ツリーの最上位を形成する 1 個ないし複数の構造エレメントが選択されます。*usetags* オプションが *false* の場合には、そのページの構造情報は完全に無視されます。

取り込まれた文書のロールマップ内のエントリ群は、もしそのページ上で、その照応するエレメントが用いられているならば、出力文書のロールマップへコピーされます。衝突しあうロールマップエントリ群（すなわち、あるカスタムタグがすでに、生成文書のロールマップ内で、あるいはそれ以前に取り込まれた文書内で、別の標準タグへマップされている）は無視されます。ただし PDF/UA モードでは、衝突しあうロールマップエントリを持つページは拒絶され、すなわち *PDF_open_pdi_page()* への呼び出しは失敗します。

無効なタグ構造を持つ文書を取り込む PDFlib は、298 ページ「構造エレメントに対するネスト規則」で説明した通りの、ISO 32000-1 で課されているタグネスト規則に関する厳格なチェックを実装しています。これらのチェックは、取り込まれる文書に対しても、そして取り込まれるページから作成されるタグ構造に対しても適用できます。取り込まれたページ内のネスト規則は、デフォルトではチェックされません。しかし、これらのチェックは、*PDF_open_pdi_document()* の *checktags* オプションで有効にすることも可能です。*checktags=strict* の場合には、*PDF_open_pdi_page()* ですべてのタグネスト規則が強制されます。もしも取り込まれたページの構造ヒエラルキーが、構造エレメントに対するネスト規則に違反しているときは、*PDF_open_pdi_page()* への呼び出しは失敗し、*PDF_get_errmsg()* はたとえば以下のようなエラーを報告します：

```
Grouping element type 'Document' cannot contain content items
(but only other structure elements)
```

多数の既存の現実世界のタグ付き PDF 文書がこのタグネスト規則に違反していますので、取り込んだ文書内のこれらの問題への対処として、以下の方式のいずれかを用いることができます：

- 取り込まれた構造ヒエラルキーのてっぺんに追加のタグを挿入する（たとえば *PDF_fit_pdi_page()* の *tags* オプションで）ことは、取り込まれたページがその文書ルートの直下にコンテンツアイテムを内容としているありがちな問題を正すために有用です。
- 他の諸問題は、追加のタグを挿入しても解決できません。たとえば表や箇条書きのための構造エレメントが不完全なときなどです。可能であれば入力文書を直すことを考慮するべきです。これが実行可能な解決策でない場合には、*PDF_open_pdi_document()* で *checktags=none* と設定することによって、非準拠のタグ構造を持つタグ付き PDF ページを取り込むこともできます。
- もしも取り込まれた構造自体は正しくて、ただ出力文書の生成された新しいタグ群と衝突しているという場合には、その新しいタグ群を適切に調整することを試みるべきです。もしこれが現実的でないならば、*PDF_begin_document()* で *checktags=none* と設定することによって、生成されたタグ付け構造内の衝突を無視することもできます。これは PDF/UA-1 モードでは許容されません。

いずれの形の *checktags=none* によって処理された非準拠入力も、非準拠の PDF 出力を生成するおそれがありますので、この設定は推奨しません。

取り込まれたページ内のタグをクエリおよびチェック　状況によっては、取り込まれた構造エレメント群を、生成される新たな構造ヒエラルキー内に正しく統合させることが難しい場合もあります。この状況を支援するために、ページを開くことが成功した後に *PDF_info_pdi_page()* を用いると、取り込まれたタグ付き PDF ページのいくつかの特性をクエリすることができます：

- キーワード *fittingpossible* は、そのページがカレントコンテキスト内に配置できるかどうかを報告します。その *tag* オプションを与えると、そのページを追加の最上位レベルタグとともに配置できるかどうかをチェックできます。この *tag* オプションの *tagname* サブオプションだけが評価されます：他のサブオプションは与えるべきではありません。このキーワードを用いることは、取り込まれるページ内の構造情報に関してよくわかっておらず、取り込まれた構造エレメント群が不適合であるせいで *PDF_fit_pdi_page()* で例外が発生することを避けたい状況において推奨されます。もしもページがこの *fittingpossible* テストで拒絶された場合には、その *tag* オプションを通じて追加のタグを挿入することを試みることもできます。
- キーワード *topleveltagcount* は、最上位レベル構造エレメントの数を報告します。なぜなら最上位レベルタグが複数ある場合もあるからです。なお *topleveltagcount* は、そのページ内容が何の構造エレメントによっても覆われていない稀な場合において、0 となることもあります。このようなページは、タグ付きでないページのように動作します。他のコンテンツアイテムと同様、これはグループ化エレメントの子として配置することはできず、そのページのてっぺんに追加のタグが必要です。
- キーワード *topleveltag* は、その取り込まれたページの最上位レベル構造エレメント（群）を報告します。これは、取り込まれたページ構造にかぶせて追加の構造エレメントを挿入することができるか、あるいは挿入する必要があるかどうかを決定するために有用でしょう。
- キーワード *lang* は、すべての最上位レベルの取り込まれた構造エレメント（群）の *lang* 属性を報告します。これは、より高い構造エレメント内で *lang* 属性が必要かどうかを決める助けとなります。

タグ付きページを配置　*PDF_fit_pdi_page()* が、取り込まれたページを新たなページ上に配置し、その構造ヒエラルキーを、生成文書の構造ツリー内に統合します。その際には、

カレントでアクティブなエレメントを、取り込まれた構造ツリーの親として用います。追加の構造エレメントを、*PDF_fit_pdi_page()* の *tag* オプションで作成することもできます。これは、取り込まれた構造ヒエラルキーのための新たな親としての役割を果たします。

取り込まれたヒエラルキー内に *Alt* または *ActualText* 属性が存在する場合には、生成される文書構造内のもっと上にある既存の属性との衝突が見出されたときは、この *Alt* または *ActualText* 属性は除去されます（306 ページ「代替テキストと置換テキストに対するネスト規則」参照）。

そのページが、*PDF_open_pdi_document()* と *PDF_open_pdi_page()* で *usetags=true* を用いて開かれている場合には、それは出力文書内に 1 回しか配置できません。なぜなら、その取り込まれた構造は、出力文書の構造ヒエラルキーの一意な位置に統合される必要があるからです。ただし、ページが複数の位置において内容と構造に寄与する場合には、それを複数回開くことができます（すなわち、別々のページハンドルが配置されます）。

未知の文書構造を持つページを取り込む際には、以下の戦略を推奨します：

- *PDF_info_pdi_page()*の*topleveltagcount*キーワードがタグカウント0を報告したならば、そのページは空であるか、またはページ装飾のみを内容としているかのどちらかです。アプリケーションによっては、このようなページは実質的な内容に全く寄与しないのでスキップすると決めることもできるでしょう。それでもそのページを取り込む場合には、Acrobat の問題を回避するために追加の *Artifact* タグを与えることを推奨します。
- *PDF_info_pdi_page()* の*fittingpossible*キーワードがこのページについて1を返したならば、これは *PDF_fit_pdi_page()* で安全に配置できます。このテストは、そのページがページ装飾として配置される場合には不要です。
- あるいは、取り込まれたページの構造のてっぺんに追加のタグを挿入することもできます。このタグを何にするかの選択は、アプリケーションによって異なり、とりわけ、取り込まれたページが構造ツリー内に挿入される場所にある構造エレメントの種別に依存します。
- *PDF_info_pdi_page()* の *fittingpossible* キーワードが、追加のタグを用いてもなおそのページを拒絶するならば、アプリケーションは別のタグを試すか、あるいはそのページを諦めるかのどちらかでしょう。

以下のコード断片は、上述の戦略を実装したものです。ページが直接配置できない場合に、追加の *P* エレメントを挿入しています：

```
fittingpossible = true;
additionaltag = "";

topleveltagcount = (int) p.info_pdi_page(page, "topleveltagcount", "");

if (topleveltagcount == 0)
{
        /* このページは構造エレメントを何も含んでいません、
         * すなわち、これは空であるか、ページ装飾のみを内容としています。
         * Acrobatのバグを回避するために「Artifact」タグを追加しましょう。
         */
        additionaltag = "tag={tagname=Artifact} ";
}
else
/*
* ページを追加のタグなしで配置してみます。
* これがうまくいかないならば、別のタグを挿入しましょう。
*/
if (p.info_pdi_page(page, "fittingpossible", "") == 0)
```

```
{
        additionaltag = "tag={tagname=P} ";

        if (p.info_pdi_page(page, "fittingpossible", additionaltag) == 0)
        {
            fittingpossible = false;
        }
}

if (fittingpossible)
{
        p.fit_pdi_page(page, 0, 0, "adjustpage " + additionaltag);
}
else
{
        System.err.println("ページをスキップします: " + p.get_errmsg());
}
```

さまざまな用途　タグ付き PDF モードでページを取り込むことの用途は、以下のように分類することができます：これらの用途に対するオプションとその他要件を表 11.7 にまとめます：

- タグを温存：タグ付き PDF 文書からページを取り込んで、そのタグ群を出力へコピーします。そのページを構成する構造エレメント群は、新たな構造ヒエラルキーの一部分となります。オプショナルに、取り込まれたヒエラルキーのてっぺんに追加のタグを配置することもできます。
- タグを破棄：タグ付きまたはタグなし PDF 文書からページを取り込んで、そのページ全体に 1 個の新たなタグでタグ付けします。既存の構造エレメント群は破棄され、取り込まれたページのコンテンツは単一のタグを構成し、内部構造を一切持たなくなります。
- ページ装飾として配置:タグ付きまたはタグなし PDF 文書からページを取り込んで、そのページ全体をページ装飾としてマークします。たとえば背景グラフィックに使う場合などです。既存の構造エレメント群は破棄されます。

表 11.7 タグ付き PDF モードでタグ付き・タグなし PDF ページを取り込み

用途	取り込まれるページ	オプション	要件とコメント
タグを温存	タグ付き	PDF_open_pdi_document( ) と PDF_open_pdi_page( ) で usetags=true	▸ 親[1] は、インライン要素か擬似要素であってはいけません ▸ 取り込まれたページを配置できるのは 1 回だけ ▸ 取り込まれたページの最上位レベルエレメント（群）はネスト規則を守る必要があります
タグを破棄	タグ付きかタグなし[2]	PDF_open_pdi_document( ) か PDF_open_pdi_page( ) で usetags=false	▸ 取り込まれたページのコンテンツは、カレントでアクティブなアイテムの一部分になります ▸ 非グループ化親エレメントのみ許容されます
ページ装飾として配置	タグ付きかタグなし	PDF_fit_pdi_page( ) で tag={tagname=Artifact}	▸ 取り込まれたページのコンテンツはページ装飾になります

1. PDF_begin_item( ) で、または PDF_fit_pdi_page( ) の tag オプションで指定された、カレントでアクティブなタグ
2. この状況は、実際に何らかのページ内容を指し示している構造エレメントが全くないタグ付き PDF ページについても起こりえます。

12 PDF のバージョンと規格

12.1 Acrobat・PDF のバージョン

ユーザー側での選択に従い、PDFlib は以下の PDF バージョンに従った出力を生成します:

- PDF 1.4（Acrobat 5、2001 年リリース）
- PDF 1.5（Acrobat 6、2003 年リリース）
- PDF 1.6（Acrobat 7、2005 年リリース）
- PDF 1.7（Acrobat 8、2006 年リリース）。技術的には ISO 32 000-1:2008 と同等
- PDF 1.7 Adobe 拡張レベル 3（Acrobat 9、2008 年リリース）
- PDF 1.7 Adobe 拡張レベル 8（Acrobat X/XI/DC、2010/2012/2015 ～ 2019 年リリース）
- ISO 32000-2 に従った PDF 2.0:2017。日付入りリビジョン ISO 32000-2:2020 を含みます

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 12.1 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（215 ページ「8.3.3 文書・ページ関連のチェック」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/XI/DC）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作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。メニュー項目**「ファイル」→「その他の形式で保存 ...」→「最適化された *PDF*...」**（Acrobat XI/DC）を、あるいは**「ファイル」→「名前を付けて保存 ...」→「最適化された *PDF*...」**（Acrobat X）を用いれば、PDF バージョンを変えることができます。

表 12.1 特定の PDF 互換モードを要する PDFlib 機能

機能	PDFlib API 関数・オプション
PDF 2.0 ＝ ISO 32000-2 か特定の PDF/A または PDF/VT 規格を要する機能	
文書部分ヒエラルキー	PDF_begin/end_dpart( )：文書部分ヒエラルキーは PDF 2.0 か PDF/VT を要します。
ファイル添付の関係	PDF_load_asset( ) で type=Attachment を用いた場合と PDF_add_portfolio_file( ) に対するオプション relationship、および PDF_begin/end_document( ) の attachments オプションと PDF_create_annotation( ) の attachment オプションに対するサブオプション relationship：関係指定は PDF 2.0 か PDF/A-3 を要します。

表 12.1 特定の PDF 互換モードを要する PDFlib 機能

機能	PDFlib API 関数・オプション
連携ファイル	PDF_end_document( )・PDF_begin/end_page_ext( )・PDF_begin/end_dpart( )・PDF_begin_template_ext( )・PDF_load_image( )・PDF_open_pdi_page( )・PDF_load_graphics( ) に対するオプション associatedfiles：連携ファイルは PDF 2.0 か PDF/A-3 を要します。
PDF 1.7 拡張レベル 8（Acrobat X/XI/DC）以上を要する機能	
256 ビットキーを用いた AES 暗号化	PDF_begin_document( )：compatibility=1.7ext8 を用いると、masterpassword か userpassword か attachmentpassword か permissions オプションが与えられている場合には、256 ビットと、拡張レベル 3 のものより強力な暗号化アルゴリズムとを用いた AES 暗号化が自動的に用いられます。
PDF 1.7 拡張レベル 3（Acrobat 9）以上を要する機能	
マルチメディア	PDF_load_asset( ) PDF_create_annotation( )：オプション type=RichMedia PDF_create_action( )：オプション type=RichMediaExecute
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 ビットキーを用いた AES 暗号化	PDF_begin_document( )：compatibility=1.7ext3 を用いると、masterpassword か userpassword か attachmentpassword か permissions オプションが与えられている場合には、PDF 1.7ext3 に従った AES-256 暗号化アルゴリズムの中にある脆弱性を回避するために、PDF 1.7 に従った 128 ビットを用いた AES 暗号化が自動的に用いられます。
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 ＝ ISO 32000-1（Acrobat 8）以上を要する機能	
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）以上を要する機能	
NChannel カラー	PDF_create_devicen( )：オプション subtype=nchannel
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( )：compatibility=1.61.7 の場合、masterpassword か userpassword か attachmentpassword か permissions オプションが与えられているときは、自動的に AES 暗号化が用いられます。
ファイル添付のみを暗号化	PDF_begin/end_document( )：オプション attachmentpassword

表 12.1 特定の PDF 互換モードを要する PDFlib 機能

機能	PDFlib API 関数・オプション
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( ) : オプション type=3D PDF_create_action( ) : オプション type=GoTo3DView
PDF 1.5（Acrobat 6）以上を要する機能	
インキ 8 色を超える DeviceN 色空間	PDF_create_devicen( ) : オプション names
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 3.4 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	compatibility=1.5 以上の場合には、自動的に圧縮オブジェクトストリームが生成されます。ただし、PDF_begin_document( ) で objectstreams=none と設定されている場合を除きます。

12.2 PDF 標準 ISO 32000

ISO 32000-1　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 Adobe の PDF 1.7 リファレンスと等価です。これは Acrobat 8 のファイル形式です。PDFlib を用いて作成される PDF 文書は ISO 32000-1 に準拠しています。この規格の複製を以下の場所で無料で入手可能です：

```
http://www.adobe.com/devnet/pdf/pdf_reference.html
```

ISO 32000-2　ISO 32000-2:2017 は、PDF 2.0 を仕様化し、以下の諸グループからの機能を取り込んでいます：

- Acrobat 9の諸機能。PDFlibでは*compatibility=pdf1.7ext3*文書オプションで対応しています。たとえば地理参照付き PDF・ヒエラルキー型ポートフォリオ・AES-256 暗号化など。詳しい一覧については表 12.1 を参照してください。
- Acrobat Xの諸機能。PDFlibでは*compatibility=pdf1.7ext8*文書オプションで対応しています。とりわけ、より強力な暗号化アルゴリズムを用いた AES-256 暗号化。
- PDF/A-3・PDF/VT 規格で導入された諸機能、すなわち連携ファイルや文書部分ヒエラルキーなど。
- その他、新しい構造エレメント種別群など、Acrobat DC 以降が対応していない諸機能。

PDF 2.0 についてはさらに詳しい説明が PDFlib ウェブサイトにあります。

PDF 2.0 の廃止機能　PDF 2.0 では、表 12.2 に挙げる、それより前のバージョン群において利用可能であった機能のいくつかが廃止になっています。これらのいずれかの機能が使用されている場合には、PDFlib は警告を発します。PDF 1.x モードにおいてもこれらの機能については避けることを推奨します。

表 12.2 ISO 32000-2 で廃止になった PDF 機能

機能	PDFlib API 関数・オプション
アクセシビリティ制限	PDF_begin_document( )：オプション permissions、キーワード noaccessible
表示・印刷のための ビューア環境設定	PDF_begin/end_document( )：オプション viewerpreferences、サブオプション printarea・printclip・viewarea・viewclip
分版辞書	PDF_begin_page_ext( )・PDF_end_page_ext( )：オプション separationinfo
プラットフォーム独自のアクションパラメータ	PDF_create_action( )：type=Launch に対するオプション defaultdir・parameters・operation
ムービーアクション	PDF_create_action( ) で type=Movie（かわりに type=RichMediaExecute を使用してください）
ムービー注釈	PDF_create_annotation( ) で type=Movie（かわりに type=RichMedia を使用してください）
Flash ベースのマルチメディア	PDF_load_asset( ) で type=Flash
ポートフォリオで Flash ナビゲータ	PDF_end_document( )、オプション portfolio、サブオプション custom・navigator
ブレンドモード配列	PDF_create_gstate( )：オプションはキーワード 1 個だけを受け付け、複数の値のリストを受け付けません
バーコードフィールド	PDF_create_field( )・PDF_create_fieldgroup( )：barcode

12.3 PDF/A によるアーカイビング

12.3.1 各種の PDF/A 規格

注 PDF/A 規格に関する一般情報が PDF ウェブサイトにあります。

ISO 19005 規格シリーズ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 の部分集合を提供します。

PDF 協会における PDF/A PDFlib GmbH は PDF 協会（the PDF Association）の創立メンバーです。PDF 協会は、そのさまざまな活動の 1 つとして、PDF/A 技術ワーキンググループ（TWG）をホストしています。この業界組織の目的は「オープン標準ベースの PDF 技術を利用した電子文書実装を、教育・専門技術・経験共有を通じて、全世界のステークホルダーのために促進する」です。詳しくは PDF 協会のウェブサイト *www.pdfa.org* を訪れてください。

ISO 19005-1 に従った PDF/A-1a:2005・PDF/A-1b:2005 PDF/A-1 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さまざまな制約を課しています。PDF/A-1 には 2 つの種類があります：

- 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 から知られた諸特性を追加します：これは、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、信頼のおけるテキスト意味付けを必須とします。PDF/A-1a は、その文書が将来において処理される際にそれが同じに見えることを保証するのみならず、そのコンテンツを信頼性を持って解釈でき、身体障碍を持つユーザーに対してもアクセシブルであることを保証するものです。

PDFlib 内の PDF/A-1 対応は以下の文書に基づいています：

- PDF/A-1 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 協会が発行した TechNote 0010「Clarifications of ISO 19005」。

PDF/A-1 と（準拠レベルを添えずに）言うときは、PDF/A-1a と PDF/A-1b の両方の準拠レベルを意味します。

ISO 19005-2 に従った PDF/A-2a・PDF/A-2b・PDF/A-2u 各種の PDF/A-2 規格は、ISO 32000-1（すなわち PDF 1.7）に基づいています。このことは、これらが PDF/A-1 よりも多くの機能をサポートしていることを意味します。PDF/A-1 と異なり、より新しい PDF/A-2 規格は、透過・レイヤー・JPEG 2000 圧縮・PDF/A ファイル添付・PDF パッケージやその他の PDF 諸機能を許しています。PDF/A-2 は以下の種類を定義しています：

- ISO 19005-2 レベル B 準拠（PDF/A-2b）。文書の視覚的体裁を確保します。

- ISO 19005-2 レベル A 準拠（PDF/A-2a）。信頼のおける Unicode テキスト意味付けと、構造情報を持つタグ付き PDF を追加しています。タグは、PDF/A-2a 文書が完全にアクセシブルであることを確実にします。
- ISO 19005-2 レベル U 準拠（PDF/A-2u）。PDF/A-2a と PDF/A-2b の中間に位置付けられます。なぜならこれは、信頼のおける Unicode テキスト意味付けについては必須としますが、構造情報については必須としないからです。PDF/A-2u は、そのページ群が忠実に再現できることを、また、そのテキストが抽出・検索できることを保証します。

PDFlib 内の PDF/A-2 対応は以下の文書に基づいています：
- PDF/A-2 規格（ISO 19005-2:2011）
- PDF 協会が発行した TechNote 0010「Clarifications of ISO 19005」。

PDF/A-2 と（準拠レベルを添えずに）言ったときは、PDF/A-2a・PDF/A-2b・PDF/A-2u の 3 種類すべての準拠レベルを意味します。

ISO 19005-3 で定義された PDF/A-3a・PDF/A-3b・PDF/A-3u　PDF/A-3 は、以下の違いを除いて PDF/A-2 と同様です：
- PDF/A-2 では、PDF/A-1 か PDF/A-2 に準拠したファイル添付しか許容しませんが、PDF/A-3 では任意のファイル種別を添付として許容します。
- 添付されたファイルは、文書全体か、ページか、あるいはその文書のその他の要素と関連付けされます。ファイル添付と、文書のそれに照応する部分との間の関係が、ソース・代替・補足データなど、明示的に指定される必要があります。

PDFlib 内の PDF/A-3 対応は以下の文書に基づいています：
- PDF/A-3 規格（ISO 19005-3:2012）
- PDF 協会が発行した TechNote 0010「Clarifications of ISO 19005」。

PDF/A-3 と（準拠レベルを添えずに）言ったときは、PDF/A-3a・PDF/A-3b・PDF/A-3u の 3 種類すべての準拠レベルを意味します。

電子インボイスのための ZUGFeRD 標準は、PDF/A-3 に基づいた重要な応用です。これは、そのインボイスの、機械が読める XML 版を、PDF/A-3 に準拠した、人が読める文書の中へ埋め込みます。ZUGFeRD に関するさらに詳しい情報が PDFlib ウェブサイトにあります。

12.3.2 一般的要件

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 クライアントプログラムが、この節で記す規則に従うなら、有効な PDF/A 出力が保証されます。PDFlib は、PDF/A 規則への違反を検出したときには例外を発生させますので、アプリケーション側でそれを処理する必要があります。この場合には PDF 出力は作成されません。表 12.3 に、PDF/A 準拠出力を作成するための一般的要件を挙げます。

PDF/A・PDF/UA-1 両立文書を作成　PDF/A 文書が同時に PDF/UA-1 に準拠することも可能です。実際、PDF/A-1a/2a/3a を作成したい場合には、生成文書のアクセシビリティを向上させるために、PDF/UA の要件に従うことを推奨します。詳細と制約については、360 ページ「PDF/UA-1・PDF/A 両立文書を作成」を参照してください。

PDF/A・PDF/X 両立文書を作成　PDF/A 文書が同時に PDF/X-1a か PDF/X-3 か PDF/X-4 に準拠することも可能です。しかし PDF/X-4p か PDF/X-5 とは無理です。このような両

表 12.3 PDF/A 準拠レベル A・B・U のための一般的要件

項目	PDF/A 準拠（すべての準拠レベル）のための PDFlib の要件
***PDF/A*準拠レベルと *PDF*互換性**	PDF_begin_document( )：pdfa オプションを、必要な PDF/A 準拠レベルに設定する必要があります。例：pdfa=PDF/A-2b PDF/A-1：PDF 1.5 以上を要する操作（レイヤーなど）を避ける必要があります。 PDF/A-2/3：PDF 1.7ext3 以上を要する操作（PDF ポートフォリオなど）を避ける必要があります。
フォント	フォントオプション embedding を true にする必要があります。オプション unicodemap=false と dropcorewidths=true は許容されません。 PDF コアフォントについても埋め込みが必須です。この埋め込み必須は、不可視テキスト（主として OCR の生成結果のために有用）に対してのみ用いられているフォントについてだけはあてはまりません。これは、optimizeinvisible オプションを用いて制御することもできます。
ページ寸法	（PDF/A-2/3）PDF_begin/end_page_ext( )：PDF/A には、厳密なページ寸法制限はありません。しかし、ページ寸法（幅と高さ、およびすべての枠エントリ）を、PDF/A-1 では範囲 3 ～ 14400 ポイント（508 cm）、PDF/A-2/3 では範囲 3 ～ 14400 ユーザー単位に収めることが推奨されます。
レイヤー	PDF/A-1：PDF_define_layer( ) と PDF_set_layer_dependency( ) を避ける必要があります。 PDF/A-2/3：レイヤーを使うことはできますが、PDF_define_layer( ) のいくつかのオプションを避ける必要があります。
セキュリティ	PDF_begin_document( )：userpassword・masterpassword・attachmentpassword・permissions オプションを避ける必要があります。
外部コンテンツ	PDF_begin_template_ext( )・PDF_load_graphics( )・PDF_open_pdi_page( )：reference オプションを避ける必要があります。 PDF_load_asset( )：external オプションを避ける必要があります。
ファイルサイズ	生成される PDF 文書のファイルサイズが 2 GB を超えてはならず、かつ、PDF オブジェクトの数が 8.388.607 未満でなければなりません。これらの制約について詳しくは 63 ページ「3.1.6 PDF 文書の最大サイズとその他の制限」を参照してください。
***PDF*取り込み**	PDF_open_pdi_document( ) が制約されます。ただし infomode=true の場合を除きます。337 ページ「12.3.7 PDF/A 文書を PDI で取り込み」を参照してください。

立ファイルを作成するには、*PDF_begin_document( )* の *pdfa*・*pdfx* オプションに対して適切な値を与えます。例：

```
ret = p.begin_document("combo.pdf", "pdfa=PDF/A-2b pdfx=PDF/X-4");
```

12.3.3 色と画像の要件

PDF/A は、忠実な色再現を保証するために、デバイス独立な色指定を必須としています。色空間は以下のソースから来る可能性があります：

- *PDF_load_image( )* と *PDF_fill_imageblock ( )* を用いて直接的に、および *PDF_load_graphics( )* を通じて間接的に読み込まれた画像
- *PDF*_set_graphics_option( ) か *PDF_setcolor( )* を用いた明示的な色指定
- オプションリストを通じた色指定。たとえばテキストフロー内などにおいて。
- 透過グループのためにブレンドする色空間：*PDF_begin/end_page_ext( )*・*PDF_begin_template_ext( )*・*PDF_load_graphics( )*：オプション *transparencygroup* でサブオプション *colorspace*

- スポットカラーか DeviceN カラーの代替色空間
- 注釈・フォームフィールドは枠・背景・内容に色を指定することができます

表 12.4 に、上に挙げた操作すべてにおいて従う必要がある、色処理のための PDF/A 要件を挙げます。

表 12.4 PDF/A 準拠レベル A・B・U のための色と画像の要件

項目	PDF/A（すべての準拠レベル）のための PDFlib の要件
出力条件（出力インテント）	もしもデバイス依存色空間 Gray・RGB・CMYK のうちのいずれかがその文書内で用いられており、かつ然るべきデフォルト色空間がそのページに対して存在していないならば、PDF_begin_document( ) の直後に、PDF_load_iccprofile( ) で usage=outputintent として呼び出すか、PDF_process_pdi( ) で action=copyoutputintent として呼び出す必要があります。
グレースケールカラー	グレースケールカラーを使えるのは、グレースケールか RGB か CMYK の出力インテントが存在する場合か、defaultgray オプションが設定されている場合だけです（フォームフィールドに対しては defaultgray オプションは利用できません）。
***RGB*カラー**	RGB カラーを使えるのは、RGB 出力インテントが存在する場合か、defaultrgb オプションが設定されている場合だけです。例外として、フォームフィールドに対しては RGB カラーはつねに使用できます。
***CMYK*カラー**	CMYK カラーを使えるのは、CMYK 出力インテントが存在する場合か、defaultcmyk オプションが設定されている場合だけです（フォームフィールドに対しては defaultcmyk オプションは利用できません）。
***Separation*（スポット）・*DeviceN*カラー**	▸ その代替色は上述の諸規則に従う必要があります。 ▸ PDF/A-2/3：PDF_create_devicen( ) の前に、その DeviceN 色空間内のすべてのカスタムスポットカラーについて PDF_makespotcolor( ) を呼び出す必要があります。
透過とオーバープリント	PDF/A-1：透過を避ける必要があります。このことは以下の API 機能に影響を与えます： ▸ PDF_load_image( )：masked オプションを避ける必要があります。ただし、そのマスクが 1 ビット画像を指し示している場合を除きます。 ▸ PDF_load_image( )：内在透過（アルファチャンネル）を持つ画像が許容されません。これを読み込むには ignoremask オプションを用いる必要があります。 ▸ PDF_load_graphics( )：透過要素を含んだ SVG グラフィックを避ける必要があります。 ▸ PDF_create_gstate( )：opacityfill・opacity オプションを避ける必要があります。ただし、それが値 1 を持つ場合を除きます。blendmode を用いる場合には、それが Normal である必要があります。softmask を用いる場合には、それが none である必要があります。 ▸ PDF_create_annotation( )：opacity オプションを避ける必要があります。 PDF/A-2/3：透過は許容されますが、PDF_create_gstate( ) では以下の規則に従う必要があります：もしもカレント色空間が ICC ベース CMYK カラーであり、かつ overprintfill か overprintstroke が true の場合には、overprintmode=1 は許容されません。
透過グループ	PDF_begin/end_page_ext( )・PDF_begin_template_ext( )・PDF_open_pdi_page( )・PDF_load_graphics( )：オプション transparencygroup が以下のとおり制限されます： PDF/A-1：オプション transparencygroup は許容されません。 PDF/A-2/3：transparencygroup オプションのサブオプション colorspace が、上でグレースケール・RGB・CMYK カラーについて述べた要件を満たす必要があります。PDF_open_pdi_page( ) と PDF_load_graphics( ) に対しては transparencygroup=auto が強制されます。 PDF/A-2/3：transparencygroup オプションのサブオプション colorspace が、上でグレースケール・RGB・CMYK カラーについて述べた要件を満たす必要があります。PDF_begin/end_page_ext( ) に対してオプション transparencygroup=none とサブオプション colorspace=none は許容されません。

表 12.4 PDF/A 準拠レベル A・B・U のための色と画像の要件

項目	PDF/A（すべての準拠レベル）のための PDFlib の要件
画像とテンプレート	PDF_load_image( )：interpolate=true オプションを避ける必要があります。 PDF/A-2/3：JPEG 2000 画像は特定の条件を満たす必要がありますので、詳しくは 191 ページ「JPEG 2000 画像」を参照してください。

出力インテント　出力条件は、意図する出力先デバイスを定義します。これは、一貫性のある色表現のために重要です。PDF/X では常に出力インテントが必須ですが、PDF/A ではこれと異なり、出力インテント ICC プロファイルの使用はオプショナルです。出力インテントは、RGB 等デバイス依存カラーがその文書内で使われている場合にのみ必須です。ICCベースカラー等デバイス独立カラーのみがその文書内で使われている場合には出力インテントは必要ありません。PDF/X は出力インテントとしてプリンタ ICC プロファイルにのみ対応していますが、PDF/A ではモニタプロファイルも許容されます。これにより、広く使われている sRGB プロファイルを出力インテントとして使うこともできるようになっています。出力インテントを指定するには、以下のように ICC プロファイルを用います：

```
icc = p.load_iccprofile("sRGB", "usage=outputintent");
```

ICC プロファイルを読み込むのではなく、出力インテントを、取り込んだ PDF/A 文書からコピーすることもできます（338 ページ「取り込んだ文書から PDF/A 出力インテントをコピー」参照）。生成される出力文書の出力インテントを設定するのはちょうど 1 回だけにする必要があります。これを設定するのは ***PDF_begin_document( )*** の直後にするべきです。

PDF/A を作成するためのカラー戦略　表 12.5 に挙げるカラー戦略の概略が、さまざまな PDF/A アプリケーションを計画するうえで役立つかもしれません。多くの状況でうまくいく最も簡単な手は、***sRGB*** 出力インテント ICC プロファイルを使うことです。なぜならこれはグレースケールと RGB カラーに対応しているからです。さらに、***sRGB*** は PDFlib に内部的に知られていますので、外部プロファイルデータや構成を要しません。

黒いテキストを、出力インテントプロファイルを必要とせず作成したい場合には、CIELab 色空間を使えます。その ***Lab*** カラー値 ***(0, 0, 0)*** は、純粋な黒をデバイス独立な形で指定しており、かつ出力インテントプロファイルなしで PDF/A に準拠しています（これと異なり DeviceGray では出力インテントプロファイルが必須）。PDFlib は各ページの先頭でカレントカラー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どうかに応じて、PDFlib は DeviceGray 色空間か Lab 色空間のいずれかを黒に対して用います。以下の呼び出しを用いれば ***Lab*** 黒色を手動で設定できます：

```
p.set_graphics_option("fillcolor={lab 0 0 0}");
```

表 12.5 準拠レベル A・B・U のための PDF/A のカラー戦略

出力インテント ICC プロファイル	その文書内で使える色空間					
	CIELab	ICC ベース	Separation・DeviceN	グレースケール[1]	RGB[1, 2]	CMYK[1]
なし	○	○	○	−	−	−
グレースケール	○	○	○	○	−	−
RGB*。例：*sRGB	○	○	○	○	○	−

表 12.5 準拠レベル A・B・U のための PDF/A のカラー戦略

出力インテント ICC プロファイル	その文書内で使える色空間					
	CIELab	ICC ベース	Separation・DeviceN	グレースケール[1]	RGB[1, 2]	CMYK[1]
CMYK	○	○	○	○	–	○

1. ICC プロファイルを持たないデバイス色空間か、ページ・パターン・テンプレートのためのデフォルト色空間
2. RGB カラーはフォームフィールドに対してはつねに許されます。

表 12.5 に挙げた色空間のみならず、カスタムスポットカラーを、その照応する代替色空間に従って使うこともできます。PDFlib は CIELab を、内蔵の HKS・PANTONE スポットカラーに対する代替色空間として用いていますので、これらは常に PDF/A 規格とともに使えます。カスタムスポットカラーに対しては、その代替色空間を、それが出力インテントと互換になるように選び取る必要があります。DeviceN 色空間内で使用されるすべてのカスタムスポットカラーについて *PDF_makespotcolor( )* を呼び出す必要があります。

12.3.4 インタラクティブ機能に対する要件

表 12.6 に、PDF/A 準拠出力を生成する際に制約されるすべての操作を示します。禁じられたいずれかの関数を PDF/A モード内で呼び出すと例外が発生します。

表 12.6 すべての PDF/A 準拠レベルのためのインタラクティブ機能に対する要件

項目	PDF/A（すべての準拠レベル）のための PDFlib の要件
注釈	PDF/A-1：PDF_create_annotation( ) が以下の制約に束縛されます： ► type=FileAttachment・Movie を持つ注釈を避ける必要があります。 ► テキスト注釈に対しては zoom・rotate オプションを true に設定する必要があります。 ► annotcolor・interiorcolor オプションを使えるのは、RGB 出力インテントが指定されている場合のみです。fillcolor オプションを使えるのは、RGB か CMYK 出力インテントが指定されている場合のみであり、かつ照応する rgb か cmyk 色空間を用いる必要があります。 ► opacity オプションを用いてはいけません。 PDF/A-2/3：PDF_create_annotation( )：type=Link のみが許容されます。
添付	PDF/A-1：PDF_begin/end_document( )：attachments オプションを避ける必要があります。 PDF/A-2：PDF_begin/end_document( )：attachments オプションの参照先が PDF/A-1 文書か PDF/A-2 文書である必要があります。 PDF/A-3：associatedfiles オプションを用いて任意のファイル種別を添付できますが、attachments オプションを避ける必要があります。以下の条件に従う必要があります： ► 添付は、その文書のさまざまな部分と関連付けることができます。そのためには、PDF_end_document( )・PDF_begin/end_page_ext( )・PDF_begin/end_dpart( )・PDF_begin_template_ext( )・PDF_load_image( )・PDF_open_pdi_page( )・PDF_load_graphics( ) の associatedfiles オプションを用います。それぞれの添付を、その文書のちょうど 1 つの部分に関連付ける必要があります。すなわち、PDF_load_asset( ) を用いて作成されたそれぞれのアセットハンドルを、ちょうど 1 つの associatedfiles オプションに与える必要があります。 ► mimetype・relationship サブオプションが必須です。 ► description サブオプションが推奨されます。 ► external=true サブオプションを避ける必要があります。

表 12.6 すべての PDF/A 準拠レベルのためのインタラクティブ機能に対する要件

項目	PDF/A（すべての準拠レベル）のための PDFlib の要件
アクションと *JavaScript*	PDF_create_action( ) : type=Hide・Launch・Movie・ResetForm・ImportData・JavaScript を持つアクションを避ける必要があります。type=name に対しては、NextPage・PrevPage・FirstPage・LastPage のみが許容されます。 PDF_begin/end_document( ) : オプション action は、トリガイベント open とともにのみ使用できます。 PDF_begin/end_document( )・PDF_begin/end_page_ext( ) : オプション action を避ける必要があります。
フォームフィールド	PDF_create_field/fieldgroup( ) は以下の制約に縛られます : ▸ 使用されているすべてのフォントが埋め込まれている必要があります。 ▸ オプション backgroundcolor・bordercolor・fillcolor・strokecolor : RGB カラーはつねに許され、グレースケールカラーは出力インテント（どの種別でも）付きの場合にのみ許され、CMYK カラーは CMYK 出力インテント付きの場合にのみ許されます（表 12.4 も参照）。 ▸ action オプションは許されません。

12.3.5 レベル U 準拠のための追加の PDF/A の要件

PDF/A-2u と PDF/A-3u のための、規格の要件の多くは、PDFlib によって自動的に満たされます。レベル U 準拠の文書を生成する際には、表 12.7 に挙げる操作のみが制限されます。言い換えれば、アプリケーションがすでに PDF/A-2b か PDF/A-3b を作成しており、かつ表 12.7 の制約に従っていれば、生成された文書はそれぞれ、PDF/A-2u か PDF/A-3u としても宣言できます。

表 12.7 PDF/A 準拠レベル U のための追加の制約

項目	PDF/A-2u/3u 準拠のための PDFlib の要件
フォント	フォントオプション unicodemap=false を避ける必要があります。

12.3.6 レベル A 準拠のための追加の PDF/A の要件

PDF/A-1a・PDF/A-2a・PDF/A-3a を作成する際には、292 ページ「11.3 タグ付き PDF の基礎」に従ったすべてのタグ付き PDF の要件を満たす必要があります。表 12.8 に、レベル A に従った出力を生成するための必須操作と推奨操作を挙げます。一般的なタグ付き PDF の諸規則のほかにも、PDF/UA の要件にも従うことによって、生成文書のアクセシビリティを向上させることを強く推奨します。詳しくは 360 ページ「12.6 PDF/UA によるユニバーサルアクセシビリティ」を参照してください。

正しい構造情報を作成することはユーザー側の役割です。文書のテキスト全体を 1 個の構造エレメントに入れたら、技術的には正しい PDF/A ですが、忠実な意味付け再生というゴールに違反しています。

表 12.8 PDF/A 準拠レベル A のための追加の要件

項目	PDF/A-1a/2a/3a 準拠のための PDFlib の要件
フォント	フォントオプション unicodemap=false を避ける必要があります。
タグ付き *PDF*	タグ付き PDF のためのすべての要件を満たす必要があります（292 ページ「11.3 タグ付き PDF の基礎」参照）。文書の構造ヒエラルキーは、その文書の論理構造を、できるだけ正確に反映するべきです。

表 12.8 PDF/A 準拠レベル A のための追加の要件

項目	PDF/A-1a/2a/3a 準拠のための PDFlib の要件
タグ付き*PDF*の推奨事項	以下の項目が強く推奨されます： ▸ すべての PDF/UA 規則に従うこと。詳しくは 360 ページ「12.6 PDF/UA によるユニバーサルアクセシビリティ」を参照してください。 ▸ テキストの自然言語を指定するべきです。Lang オプションを用います（304 ページ「言語指定」参照）。 ▸ 画像等非テキストコンテンツアイテムが代替記述を与えるべきです。PDF_begin_item( ) の Alt オプションを用いるか、PDF_fit_image( )・PDF_fit_pdi_page( ) などの tag オプションを用います。 ▸ 略語と頭字語が適切な拡張テキストを持つべきです。PDF_begin_item( ) の E オプションで指定するか、tag オプションで指定します。 ▸ ヘッダ・フッタ・ページ番号といったページネーション機能をページ装飾としてマークするべきです。artifacttype=pagination を用います。
単語境界	単語間を空白キャラクタ（U+0020）で区切る必要があります。autospace オプションを用いるとこの作業を単純化できます。
テキスト出力と*PUA Unicode*キャラクタ	PDF/A-2a/3a：PUA Unicode キャラクタ（ロゴや記号など）が、適切な置換テキストを持つ必要があります。それを囲うコンテンツアイテムに対する PDF_begin_item( ) の ActualText オプションで指定するか、その照応する出力関数の同等の tag オプションで指定します（詳しくは後述）。
注釈	PDF_create_annotation( )：テキストを全く表示しない注釈に対しては contents オプションが推奨されます。

PUA キャラクタ　PDF/A-2a と PDF/A-3a は、私用領域、略して PUA、すなわち主として範囲 U+E000 ～ U+F8FF（詳しくは 107 ページ「BMP と PUA」を参照）の中にある Unicode 値を持ったキャラクタに対して、追加の要件を含んでいます。PUA キャラクタは通常、装飾的・記号グリフ、または企業ロゴ等カスタムグリフです。PDF/A-2a/3a は、PUA キャラクタが、そのキャラクタのテキスト表現を内容とする *ActualText* 属性を伴うことを必須とします。この *ActualText* は、個別の PUA キャラクタに対して割り当てることもできますし、ある 1 個の PUA キャラクタを包含するもっと長いキャラクタ列に対して割り当てることもできます。この *ActualText* を、*Span* インラインレベルエレメントとともに与えることを推奨します。

*PDF_info_font()*を使って、ある特定のコードの、ある指定したフォントに対するUnicode値をチェックできます（158 ページ「6.6.2 フォント依存のエンコーディング・Unicode・グリフ名クエリ」参照）：

```
uv = (int) p.info_font(font, "unicode", "code=" + c);
```

この生成される Unicode 値 *uv* が PUA に帰する場合には、これは *ActualText* 属性を必要とします。以下のコード断片は、PDFlib 企業ロゴのグラフィカル表現を包含した *PDFlibLogo* というフォントを想定しています。このロゴをページ上に配置する際には、テキスト「*PDFlib Logo*」を内容とする然るべき *ActualText* サブオプションを持った *Span* エレメントが *tag* オプションで与えられます：

```
p.fit_textline(text, 50, 700,
        "fontname=PDFlibLogo encoding=unicode embedding fontsize=24 " +
        "tag={tagname=Span ActualText={PDFlib Logo}}");
```

そのグリフに関して何ら情報を持たない場合、ゆえに然るべき *ActualText* がたやすくは得られない場合には、そのフォント内におけるそのグリフの名前を使うという手もあります。これを知るには以下のようにします：

```
gn_idx = (int) p.info_font(font, "glyphname", "code=" + c);
glyphname = p.get_option(gn_idx, "");
```

このグリフ名を *ActualText* で使う際、定常的な句と組み合わせるのも手でしょう。たとえば *Wingdings* フォント内のコード *0x1A* は、コンピュータキーボードの絵を内容とし、グリフ名 *keyboard* を持ちます。このグリフは U+F037 へマップします。すなわち PUA 値です。実テキストとして「*keyboard* **の記号**」を用いれば、この記号に対して意味を成すでしょう。ただし、プログラム的に構築される *ActualText* は、当座しのぎの解決策と考えるべきです。人の選ぶテキストは常に、機械生成の *ActualText* よりも望ましいものです。

12.3.7 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 文書を取り込もうとするときは、追加の規則が適用されます（PDF 取り込みについて詳しくは 213 ページ「8.3 PDF ページを PDI で取り込む」を参照）。あらゆる取り込み文書は、表 12.9 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、任意の入力 PDF 文書を PDF/A へ変換することもできません。

ある特定の PDF/A 準拠レベルが PDFlib で構成されていて、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 準拠レベルに従ってい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 12.9 さまざまな PDF/A 出力レベルに対する互換 PDF/A 入力レベル

	取り込まれる文書の PDF/A レベル				
PDF/A 出力レベル	**PDF/A-1a:2005**	**PDF/A-1b:2005**	**PDF/A-2a・PDF/A-3a**	**PDF/A-2b・PDF/A-3b**	**PDF/A-2u・PDF/A-3u**
PDF/A-1a:2005	許容	―	―	―	―
PDF/A-1b:2005	許容	許容	―	―	―
PDF/A-2a・PDF/A-3a	許容	―	許容	―	―
PDF/A-2b・PDF/A-3b	許容	許容	許容	許容	許容
PDF/A-2u・PDF/A-3u	許容	―	許容	―	許容

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/clone_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 12.10 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。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 の検証に利用されるようにも、また、取り込んだ文書に 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。

表 12.10 PDF/A 文書（すべての準拠レベル）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。

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 状況を以下のようにクエリすることができます：

```
pdfalevel = p.pcos_get_string(doc, "pdfa");
```

このステートメントは、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準拠レベルをクエリ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、それをコピーしようとする前に、まず pCOS を使って、出力インテントが存在するかどうかをチェックする必要があります。

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/clone_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。

12.3.8 PDF/A のための XMP 文書メタデータ

PDF/A 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。PDF/A では、2 つの種類の XMP 文書レベルメタデータに対応しています：1 つは、定義済みスキーマというよく知られたメタデータスキーマの集合です。これは、XMP 仕様の基礎をなすバージョンから採られたものです。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 準拠エントリ群を自動的に作成するほか、いくつかのよく使われるエントリ（*CreationDate* 等）も自動的に作成します。

文書レベル XMP　XMP文書メタデータは、*PDF_begin_document()*か*PDF_end_document()* または両方の *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。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。

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。

コンポーネントレベル XMP　文書全体に対してだけではなく、XMP メタデータは、PDF 文書内のそれ以外のページや画像といった構成要素に対しても与えることができます。コンポーネントレベルのメタデータに対しては、PDF/A-1 の要請はありませんが、PDF/A-2 と PDF/A-3 では、コンポーネントレベル XMP 内のカスタムプロパティ群をも文書レベル XMP と同様の拡張スキーマ記述によって記述することを義務付けています。

コンポーネントレベル XMP メタデータは、*PDF_begin/end_page_ext()* や *PDF_load_image()* などの関数の *metadata* オプションで与えることができます。

定義済み XMP スキーマ　PDF/A 内の文書メタデータに対する XMP の使用は、以下の仕様に基づいています：

- PDF/A-1：XMP 2004 仕様
- PDF/A-2・PDF/A-3：XMP 2005

それぞれの XMP 仕様に記述されたスキーマを定義済みスキーマといい、それらの名前空間 URI とその望ましい名前空間接頭辞とともに表 12.11 に挙げています。定義済みスキーマのプロパティのみを PDF/A では使えます。ただし、拡張スキーマ記述が存在する場合は例外です（後述）。PDF/A-1 のための定義済み XMP 2004 スキーマ群内のプロパティの完全な一覧が、PDF 連合の PDF/A 技術センターからの TechNote 0008 内にあります。PDF/A-2/3 は、XMP 2005 からの定義済みスキーマ群を追加していますが、この追加スキーマは画像と動的メディアに関連したものですので、文書メタデータのために有用とはなりにくいでしょう。

表 12.11 PDF/A-1 のための定義済み XMP スキーマ（詳しくは XMP 2004 と XMP 2005 を参照）

スキーマの名称と説明	名前空間 URI	望ましい名前空間接頭辞
PDF/A-1・PDF/A-2・PDF/A-3 で使うための XMP 2004 スキーマ群		
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 独自プロパティ群用 EXIF スキーマ	http://ns.adobe.com/exif/1.0/	exif
TIFF プロパティ群用 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
PDF/A-2・PDF/A-3 で使うための追加 XMP 2005 スキーマ群		
カメラ Raw スキーマ	http://ns.adobe.com/camera-rawsettings/1.0/	crs
追加 EXIF プロパティ群用 EXIF スキーマ	http://ns.adobe.com/exif/1.0/aux/	aux
XMP 動的メディアスキーマ	http://ns.adobe.com/xmp/1.0/DynamicMedia/	xmpDM

XMP 拡張スキーマ記述　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 では、カスタムスキーマを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 12.12 に、1 個ないし複数の拡張スキーマとそのプロパティ群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）。

コンポーネントレベル XMP（ページレベル等）に対してカスタム XMP プロパティが用いられる場合には、それに照応する拡張スキーマ記述をそのカスタム XMP プロパティ群とともに同一関数（*PDF_begin_page_ext()* 等）内で与えることもできます。あるいは、コンポーネントレベル XMP に対する拡張スキーマ記述を文書レベル XMP とともに *PDF_begin_document()* で与えることも可能です。

XMP 拡張スキーマ記述の構築に関するさらなる詳細と例が、PDF/A 技術センターからの TechNote 009 にあります。

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 12.12　PDF/A 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。

12.4 PDF/X による印刷出力

12.4.1 PDF/X 規格ファミリ

注 PDF/X に初めてふれる方には、PDF 協会が発行した文書「PDF/X in a Nutshell」を導入として推奨します（www.pdfa.org/pdfx-in-a-nutshell/ 参照）。

PDF/X 形式群は、ISO 15930 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 の部分集合を提供することを目指しています。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。表 12.13 に、各種 PDF/X の主な違いを挙げます。

表 12.13 各種 PDF/X の比較

	PDF/X-3	PDF/X-4	PDF/X-4p	PDF/X-5n
PDF バージョン	PDF 1.4	PDF 1.6	PDF 1.6	PDF 1.6
CMYK・スポットカラー	可	可	可	可
デバイス独立カラー（ICCBased・Lab）	可	可	可	可
透過・レイヤー	—	可	可	可
外部参照出力インテント ICC プロファイル	—	—	可	可
n 色出力インテント ICC ロファイル	—	—	—	可
外部参照内容	—	—	—	—

ISO 15930-4 で定義された PDF/X-1a:2003 この規格は、PDF 1.4 に基づいており、そのいくつかの機能（透過など）が禁止されています。PDF/X-1a は、CMYK・スポットカラーにのみ対応しており、ICC プロファイルを用いたモダンなカラーマネジメントには対応していません。ですので PDF/X-1a は時代遅れと考えられています。

ISO 15930-6 で定義された PDF/X-3:2003 この規格は、PDF 1.4 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ワークフローに対応しています。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。いくつかの PDF 1.4 機能、とりわけ透過が禁止されています。PDF バージョンが古いので、また、透過が除外されているので、PDF/X-3 は時代遅れと考えられています。

ISO 15930-7 で定義された PDF/X-4 この規格は、PDF 1.6 に基づいています。PDF/X-4 では、透過とレイヤーは許されますが、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。その変種である 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 15930-7:2010 を実装しています。2008 バージョンと比べて、この 2010 バージョンでは、レイヤーの取り扱いに関して変更が加えられています。

ISO 15930-8 で定義された PDF/X-5 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）、以下の種類から成ります：

- PDF/X-5g（廃止）：グラフィック内容を PDF 文書の外に許しています。
- PDF/X-5pg（廃止）：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:n色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。このプロファイルは ***xCLR*** プロファイルとも呼ばれます。

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。なぜならこれらのほうが一般的な規格だからです。

PDFlib は、ISO 15930-8:2010 を、2011 年に発行された正誤表 1 も含めて実装しています。

12.4.2 一般的要件

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。

PDFlib クライアントプログラムが、この項に記す諸規則に従えば、有効な PDF/X 出力が保証されます。PDFlib は、PDF/X 規則への違反を検出すると例外を発生させ、PDF 出力は何も生成されません。表 12.14 に、PDF/X 準拠出力を生成するための一般的要件を挙げます。

表 12.14 PDF/X 準拠のための一般的要件

項目	PDF/X 互換のための PDFlib の要件
***PDF/X*準拠レベルと *PDF*互換性**	PDF_begin_document( )：pdfx オプションを、必要な PDF/X 準拠レベルに設定する必要があります。例：pdfx=PDF/X-4。 PDF/X-1a・PDF/X-3：PDF 1.5 以上を必要とする操作を避ける必要があります。 PDF/X-4・PDF/X-5：PDF 1.7 以上を必要とする操作を避ける必要があります。
フォント	フォントオプション embedding が true である必要があります。埋め込みは PDF コアフォントについても必須です。
ページ寸法	PDF_begin/end_page_ext( )：ページ枠群を cropbox・bleedbox・trimbox・artbox オプションを通じて設定可能です。これらが以下の要件を満たす必要があります： ► TrimBox か ArtBox を設定する必要がありますが、ただしこれらの枠エントリを両方とも設定してはいけません。TrimBox も ArtBox も見つからないときは、PDFlib は CropBox（もしあれば）を TrimBox として採り、CropBox も見つからないときは MediaBox を採ります。 ► BleedBox が、存在するときは、ArtBox と TrimBox を完全に包含している必要があります。 ► CropBox が、存在するときは、ArtBox と TrimBox を完全に包含している必要があります。
レイヤー	PDF/X-1a・PDF/X-3：レイヤーを避ける必要があります。 PDF/X-4・PDF/X-5：レイヤーを使えますが、ただし PDF_define_layer( ) のいくつかのオプションを避ける必要があります。
文書情報フィールドと *XMP*メタデータ	Creator・Title 情報フィールドを、空でない値に設定する必要があります。PDF_set_info( ) を用いるか、（PDF/X-4・PDF/X-5 では）PDF_begin/end_document( ) の metadata オプションで xmp:CreatorTool・dc:title XMP プロパティを用います。 PDF_set_info( ) で Trapped 情報フィールドに、または PDF_begin/end_document( ) の metadata オプションでその照応する XMP プロパティ pdf:Trapped には、値 True か False 以外を避ける必要があります。
セキュリティ	PDF_begin_document( )：userpassword・masterpassword・permissions オプションを避ける必要があります。

表 12.14 PDF/X 準拠のための一般的要件

項目	PDF/X 互換のための PDFlib の要件
外部グラフィック内容（参照）	PDF/X-1a/3/4：PDF_begin_template_ext( ) と PDF_load_graphics( ) と PDF_open_pdi_page( ) で reference オプションを避ける必要があります。 PDF/X-5g・PDF/X-5pg（廃止）：PDF_begin_template_ext( ) か PDF_load_graphics( ) か PDF_open_pdi_page( ) で reference オプションで与えられる参照先が、以下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・PDF/X-3・PDF/X-4・PDF/X-4p・PDF/X-5g・PDF/X-5pg。かつ、同一の出力インテントのために作成されている必要があります。特定の XMP メタデータエントリ群が参照先内で必須ですので、すべての PDF/X 文書が参照先として受け付けられるとは限りません。PDFlib 8 以上を用いて生成された PDF/X 文書は参照先として使えます。
ファイルサイズ	PDF/X-4・PDF/X-5：生成される PDF 文書のファイルサイズが 2 GB を超えてはならず、かつ、PDF オブジェクトの数が 8.388.607 未満でなければなりません。これらの制限に関して詳しくは 63 ページ「3.1.6 PDF 文書の最大サイズとその他の制限」を参照してください。
***PDF*取り込み**	PDF_open_pdi_document( ) が制約されます。ただし infomode=true の場合を除きます。349 ページ「12.4.6 PDF/X 文書を PDI で取り込む」を参照してください。

12.4.3 出力インテントと色の要件

PDF/X は、忠実な色再現を保証するために、デバイス独立な色指定を必須としています。色空間は以下のソースから来る可能性があります：

- *PDF_load_image( )* と *PDF_fill_imageblock( )* を用いて直接的に読み込まれた画像と、*PDF_load_graphics( )* を通じて間接的に読み込まれた画像
- *PDF_set_graphics_option( )* か *PDF_setcolor( )* を用いた明示的な色指定
- オプションリストを通じた色指定。たとえばテキストフロー内において。
- スポットまたは DeviceN カラーの代替色空間
- 透過グループのためのブレンドする色空間：*PDF_begin/end_page_ext( )*・*PDF_begin_template_ext( )*・*PDF_load_graphics( )*：オプション *transparencygroup* でサブオプション *colorspace*

上記の操作は、以下に詳しく示すように、PDF/X 出力を生成する際、さまざまな制約に従う必要があります。

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、意図される出力先デバイスか印刷条件を定義します。これは主に、校正の信頼性を得るために有用です。PDF/X の出力インテントは通常、グレースケール・RGB・CMYK いずれかの ICC プリンタプロファイルによって記述されます。ただし PDF/X-5n は n 色 ICC プロファイルに対応しています。詳細は PDF/X の種類によって異なります：

- PDF/X-1a/3/4/5g：出力インテントを、出力先デバイスか印刷条件のための ICC プロファイルを埋め込むことによって指定できます。
- PDF/X-4p・PDF/X-5pg・PDF/X-5n：出力インテントのための外部 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ます）。この出力インテント ICC プロファイルは、名前で参照されるだけでなく、チェックサムによって保護された強力な参照が作成されます。参照 ICC プロファイルはその PDF 出力内に埋め込まれはしないのですが、それでも PDF 作成時にこれが利用可能である必要があります。*urls* オプションを、1 個ないし複数の、その ICC プロファイルを発見できる場所の有効な URL とともに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 */
}
```

PDF/X-5n：その PDF 文書の受信者がその出力インテント ICC プロファイルを利用できるようにするために、ICC プロファイルを PDF 文書内へ含めることもできます。そのためには ***embedprofile*** オプションを用います。これは、ICC プロファイルを PDF 文書に、ファイル添付のように添付します。このようにプロファイルを埋め込む場合には、***urls*** オプションは必要ありません。

- PDF/X-1a・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。ただし、その照応する ICC プロファイルを埋め込みません。ただしこの機能は廃止です。

すべての PDF/X 種別における出力インテント処理の比較を表 12.15 に示します。出力インテントに対する ICC プロファイルは、意図する印刷プロセスと用紙に基づいて、プリントプロバイダによって指定されるべきです。印刷工程がわからない場合、またはプリントプロバイダが出力インテントを何も指定していない場合には、然るべき ICC プロファイルを選ぶためのいくつかの推奨事項が www.pdflib.com にあります。

表 12.15 さまざまな PDF/X 種別に対する出力インテント処理

	PDF/X-1a	PDF/X-3	PDF/X-4	PDF/X-4p	PDF/X-5n
出力インテントの色空間	グレースケールか CMYK	グレースケールか RGB か CMYK	グレースケールか RGB か CMYK の ICC プロファイル		n 色 ICC プロファイル（xCLR）
出力インテント ICC プロファイルの埋め込み	それ以外では必須		常に埋め込まれる	外部参照、ただし ICC プロファイルを添付することも可能	
その他の制約	—	—	CMYK 出力インテントプロファイルを他の目的に使用することは不可		出力インテントプロファイルを他の目的に使用することは不可

PDF/X-1a に対する色の要件　PDF/X-1a は、CMYK ベースのワークフローを対象としています。これは色空間としてグレースケール・CMYK と Separation・DeviceN のみを許容します。表 12.16 に、この CMYK ベースの PDF/X-1a 規格に対する要件を挙げます。

表 12.16 PDF/X-1a 準拠のための色の要件

項目	PDF/X-1a 互換のための PDFlib の要件
出力条件（出力インテント）	PDF_begin_document( ) の直後に、PDF_load_iccprofile( ) を usage=outputintent として、または PDF_process_pdi( ) を action=copyoutputintent として呼び出す必要があります。グレースケールか CMYK の出力インテントを与える必要があります。
グレースケールカラー	defaultgray オプションを避ける必要があります。
***RGB*カラー**	RGB カラーと、defaultrgb オプションを避ける必要があります。
***CMYK*カラー**	defaultcmyk オプションを避ける必要があります。

表 12.16 PDF/X-1a 準拠のための色の要件

項目	PDF/X-1a 互換のための PDFlib の要件
***ICC*ベースカラー**	PDF_set_graphics_option( ) か PDF_setcolor( ) で iccbasedgray/rgb/cmyk 色空間を避ける必要があり、また、iccprofilegray/rgb/cmyk オプションを避ける必要があります。
***Lab*カラー**	PDF_set_graphics_option( ) か PDF_setcolor( ) で Lab 色空間を避ける必要があります。
***Separation*（スポット）・*DeviceN*カラー**	代替色はグレースケールか CMYK である必要があります。内蔵の Pantone データベースは Lab 代替色に基づいているので、これらの色は使用できません。

PDF/X-3・PDF/X-4/4p・PDF/X-5g/pg に対する色の要件　PDF/X-3/4 と PDF/X-5g/5pg は、ICC プロファイルを用いたデバイス独立なワークフローに対応しています。表 12.17 に、これらの規格のための要件を挙げます。PDF/X-5n は、別のワークフローを対象にしていますので、その要件は次の項で別途挙げます。

PDF/X-4/5 には特殊な規則が適用されます：CMYK 出力インテントプロファイル（すなわち *usage=outputintent* を用いて読み込まれた）を、同一文書内の ICC ベース色空間（すなわち *usage=iccbased* を用いて読み込まれた）のために用いることはできません。この要件は、PDF/X 規格によって必須とされているものであり、CMYK プロファイルに対してのみ適用され、グレースケール・RGB プロファイルには適用されません。この衝突が生じた場合には、単に ICC プロファイルを省略してかわりにプレーンな CMYK カラーを使用するのがよいでしょう。なぜなら出力インテント CMYK プロファイルはいずれにせよ適用されるからです。不必要なエラーメッセージを避けるため、PDFlib は、画像か SVG グラフィックに埋め込まれている CMYK ICC プロファイルが PDF/X 出力インテントと等しい場合にはそれを無視します。

この制約は、取り込まれる PDF 文書にも適用されます：取り込まれるページが、生成される文書の出力インテントと同じ CMYK ICC プロファイルを用いている場合には、それは *PDF_open_pdi_page( )* によって拒絶されます。

表 12.17 PDF/X-3・PDF/X-4/4p・PDF/X-5g/5pg 準拠のための色の要件

項目	PDF/X-3・PDF/X-4/4p・PDF/X-5g/5pg 互換のための PDFlib の要件
出力条件（出力インテント）	PDF_begin_document( ) の直後に、PDF_load_iccprofile( ) を usage=outputintent として、または PDF_process_pdi( ) を action=copyoutputintent として呼び出す必要があります。グレースケールか RGB か CMYK の出力インテントを与える必要があります。 PDF/X-3：HKS または Pantone スポットカラーか、ICC ベースカラーか、Lab カラーが使用されている場合には、出力デバイス ICC プロファイルを埋め込む必要があります。この場合には、標準出力条件の使用は許容されません。
グレースケールカラー	グレースケールカラーを使えるのは、グレースケールか CMYK の出力インテントが存在するか、defaultgray オプションが設定されているときだけです。
***RGB*カラー**	RGB カラーを使えるのは、RGB 出力インテントが存在するか、defaultrgb オプションが設定されているときだけです。
***CMYK*カラー**	CMYK カラーを使えるのは、CMYK 出力インテントが存在するか、defaultcmyk オプションが設定されているときだけです。
***Separation*（スポット）・*DeviceN*カラー**	▸ PDF/X-3/4・PDF/X-5g/pg：代替色空間は上記の諸規則に従う必要があります。 ▸ PDF/X-4・PDF/X-5g/pg：PDF_create_devicen( ) の前に、その DeviceN 色空間内のすべてのカスタムスポットカラーについて PDF_makespotcolor( ) を呼び出す必要があります。 ▸ PDF/X-4・PDF/X-5g/pg：PDF_create_devicen( ) の process オプションの colorspace サブオプションは、PDF/X 出力インテントと合致している必要があります。

PDF/X-5n に対する出力インテントと色の要件　PDF/X-5n に対する出力インテント ICC プロファイルは、n 色印刷デバイスを記述している必要があります。このようなプロファイルは色空間指定 *xCLR* を内容として持ちます。ここで *x* は、2 から F までの 16 進数で、すなわち *2CLR* ～ *FCLR* となり、2 ～ 15 個のインキを表します。*xCLR* プロファイルは、プレーンな PDF では対応されておらず、PDF/X-5n の出力インテントとしてのみ使用可能です。PDF はこのようなプロファイルに直接対応していませんので、このようなプロファイルは、出力インテントとして埋め込むことはできず、外部ファイルとして参照する必要があります。ただしこれは外部オブジェクトとして、ファイル添付にして埋め込むことは可能ですので、そうすると入手可能性を向上させることができます。

xCLR プロファイルは、名前付きインキ群の名前とカラー値を持ったインキテーブルを内容として持ちます。PDF/X-5n 文書内で許される色空間と関連して、*xCLR* 出力インテントプロファイルがインキ *Black* を内容として持っているか、または場合によってはすべての CMYK インキ *Cyan・Magenta・Yellow・Black* を内容として持っているかは重要です。たとえば、CMYK・オレンジ・グリーン・バイオレットインキ（CMYKOGV）を用いた印刷処理のための *7CLR* プロファイルは、両方の要件を満たします。

表 12.18 に、PDF/X-5n に対する要件を挙げます。

表 12.18 PDF/X-5n 準拠のための色の要件

項目	PDF/X-5n 互換のための PDFlib の要件
出力条件（出力インテント）	PDF_begin_document( ) の直後に、PDF_load_iccprofile( ) を usage=outputintent として、または PDF_process_pdi( ) を action=copyoutputintent として呼び出す必要があります。n 色（xCLR）プロファイルを与える必要があります。
グレースケールカラー	グレースケールカラーを使えるのは、出力インテントがインキ Black を内容として持っているか、defaultgray オプションが設定されているときだけです。
***RGB*カラー**	RGB カラーを使えるのは、defaultrgb オプションが設定されているときだけです。
***CMYK*カラー**	CMYK カラーを使えるのは、出力インテントがインキ Cyan・Magenta・Yellow・Black をすべて内容として持っているか、defaultcmyk オプションが設定されているときだけです。
***Separation*（スポット）・*DeviceN*カラー**	► 代替色空間は上記の諸規則に従う必要があります。 ► PDF_create_devicen( ) の前に、その DeviceN 色空間内のすべてのカスタムスポットカラーについて PDF_makespotcolor( ) を呼び出す必要があります。出力インテント内のインキリストにあるスポットカラーはこの要件から除外されます。 ► PDF_create_devicen( ) のオプション subtype=nchannel と process は許容されません。

然るべき 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、然るべき ICC 出力インテントプロファイルをインターネットで見つけることができます。

PDF/X-3/4/5 を作成するためのカラー戦略　色処理については、以下の 2 種類の戦略が可能であり、場合によっては組み合わせることもできます：

► デバイス独立カラー：出力インテント ICC プロファイルの種類にかかわらず、デバイス独立色空間を、すなわち ICC ベースか CIELab を使えます。Lab カラー値 *(0, 0, 0)* は、

純粋な黒をデバイス独立な形で指定します。Lab 黒色を手作業で設定するには、以下の呼び出しを用います：

```
p.set_graphics_option("fillcolor={lab 0 0 0}");
```

- デバイス依存カラー：デバイス独自のグレースケールか RGB か CMYK カラーを使えます。グレースケールカラーはどんな種類の出力インテントに対しても使えますが、RGB・CMYK カラーは整合する出力インテントとしか使えません。

スポット・DeviceN カラーを、その照応する代替色空間に従って使うこともできます。PDFlib は CIELab を、内蔵の HKS・Pantone スポットカラーに対する代替色空間として用いていますので、これらは常に PDF/X-3/4/5 で使えます。カスタムスポットカラーに対しては、その代替色空間を、それが出力インテントと互換になるように選び取る必要があります。DeviceN 色空間内で使用されているすべてのカスタムスポットカラーについて *PDF_makespotcolor( )* を呼び出す必要があります。

表 12.19 内のカラー戦略の要約は、PDF/X のアプリケーションを計画するために有用でしょう。

表 12.19 PDF/X-3/4/5 のカラー戦略

出力インテント ICC プロファイル	その文書で使える色空間					
	CIELab	ICC ベース	Separation・DeviceN	グレースケール[1]	RGB[1]	CMYK[1]
グレースケール	○	○	○	○	–	–
RGB[2]	○	○	○	○	○	–
CMYK	○	○	○	○	–	○
PDF/X-5nのための xCLR	○	○	○	○	–	–

1. ICC プロファイルを、またはそのページのためのデフォルト色空間を持たないデバイス色空間
2. PDF/X 規格はプリンタプロファイルを必須とします。sRGB などモニタプロファイルを使うことはできません。RGB プリンタプロファイルは非常に稀です。

12.4.4 画像と透過の要件

ラスタ画像は、348 ページ「12.4.5 インタラクティブ機能のための要件」で説明した色関連の諸要件に従う必要があります。に、PDF/X 準拠のための画像関連の追加の要件を挙げます。

表 12.20 PDF/X 準拠のための画像要件

項目	PDF/X 準拠のための PDFlib の要件
画像	► PDF/X-1a：RGB・ICC ベース・YCbCr・Lab のうちのいずれかの色を用いた画像とグラフィックは避ける必要があります。 ► PDF/X-1a/3：JBIG2 画像は避ける必要があります。 ► PDF/X-4/5：JPEG 2000 画像は特定の諸要件を満たす必要があります。詳しくは 191 ページ「JPEG 2000 画像」参照。

表 12.20 PDF/X 準拠のための画像要件

項目	PDF/X 準拠のための PDFlib の要件
透過	PDF/X-1a・PDF/X-3：透過は避ける必要があります。これは以下の API 機能に影響します： ► PDF_load_image( )：masked オプションは、そのマスクが 1 ビット深度の画像を参照しているのでない限り、避ける必要があります。 ► PDF_load_image( )：内在透過（アルファチャンネル）を持った画像は許容されません。それを読み込むには ignoremask オプションを用いる必要があります。 ► PDF_load_graphics( )：透過要素を内容として持つ SVG グラフィックは避ける必要があります。 ► PDF_create_gstate( )：opacityfill・opacitystroke オプションは、それが値 1 を持つのでない限り、避ける必要があります。blendmode が使用される場合には、それは Normal である必要があります。softmask が使用される場合には、それは none である必要があります。 PDF/X-4/5：透過画像・グラフィックを使用できます。
透過グループ	PDF_begin/end_page_ext( )・PDF_begin_template_ext( )・PDF_open_pdi_page( )・PDF_load_graphics( )：オプション transparencygroup に以下の制約が課せられます： ► PDF/X-1a・PDF/X-3：オプション transparencygroup は許容されません。 ► PDF/X-4/5p/5pg：transparencygroup オプションのサブオプション colorspace は、表 12.17 のグレースケール・RGB・CMYK カラーに対する要件を満たす必要があります。PDF_open_pdi_page( ) と PDF_load_graphics( ) に対しては transparencygroup=auto が強制されます。 ► PDF/X-5n：透過オブジェクトを内容として持つすべてのページに対してオプション transparencygroup が必要です。transparencygroup オプションのサブオプション colorspace は、表 12.18 に示されているグレースケール・RGB・CMYK カラーに対する要件を満たす必要があります。PDF_begin/end_page_ext( ) に対しては、オプション transparencygroup=none とサブオプション colorspace=none は許容されません。

12.4.5 インタラクティブ機能のための要件

表 12.21 に、PDF/X 準拠出力を生成する際に制限されるすべての操作を挙げます。PDF/X モードでこの禁じられた関数のうちのいずれかを呼び出すと例外が発生します。

表 12.21 インタラクティブ機能のための PDF/X の要件

項目	PDF/X 互換のための PDFlib 関数・オプションの要件
注釈とフォームフィールド	PDF_create_annotation( )・PDF_create_field( )：BleedBox（BleedBox が存在しない場合には TrimBox/ArtBox）内部には注釈を避ける必要があります。
ファイル添付	PDF/X-1a/3：PDF_begin/end_document( )：オプション attachments を避ける必要があります。PDF_create_annotation( ) で type=FileAttachment を避ける必要があります。
アクションと *JavaScript*	PDF_create_action( )：すべてのアクションを、JavaScript も含めて避ける必要があります。
ビューア環境設定/表示・印刷領域	PDF_begin/end_document( )：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場合には、media か bleed 以外の値は許容されません。

12.4.6 PDF/X 文書を PDI で取り込む

既存の PDF 文書のページを PDF-X 出力文書へ取り込もうとする際には、追加の規則が適用されます（詳しくは 213 ページ「8.3 PDF ページを PDI で取り込む」参照）。取り込まれる文書はすべて、表 12.22 に従ってカレント PDF/X モードに互換な PDF/X レベルに準拠している必要があります。許容されるすべての PDF/X-4/5 出力との組み合わせに対して、以下の追加の規則に服する必要があります：取り込まれるページが、生成される文書の出力インテントと同じ CMYK ICC プロファイルを用いている場合には、それは PDF/X-4/5 規格に違反するので、*PDF_open_pdi_page( )* によって拒絶されます。

ある特定の PDF/X 準拠レベルが PDFlib で構成されていて、かつ、取り込まれた文書もその互換レベル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。カレント PDF/X レベルに互換でない文書は *PDF_open_pdi_document( )* で拒絶されます。

表 12.22 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

	取り込まれる文書の PDF/X レベル						
PDF/X 出力レベル	**PDF/X-1a**	**PDF/X-3**	**PDF/X-4**	**PDF/X-4p**	**PDF/X-5g**	**PDF/X-5pg**	**PDF/X-5n**
PDF/X-1a	許容						
PDF/X-3	許容	許容					
PDF/X-4	許容	許容	許容	許容			
PDF/X-4p	許容	許容	許容	許容[1]			
PDF/X-5g	許容	許容	許容	許容	許容[2]	許容[2]	
PDF/X-5pg	許容	許容	許容	許容[1]	許容[2]	許容[1,2]	
PDF/X-5n							許容[3]

1. PDF_process_pdi( ) で action=copyoutputintent を指定すると、外部出力インテント ICC プロファイルへの参照がコピーされます。
2. 取り込まれたページが、参照された XObject を含んでいるときは、PDF_open_pdi_page( ) は代理と参照の両方をコピーします。
3. 取り込まれる文書が、同一の出力インテント（チェックサムが等しい）を使用している必要があります。

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。

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/X の検証を行なったり、取り込み文書に 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 ステータスをクエリ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。

取り込まれた文書から PDF/X 出力インテントをコピー　PDF/X 準拠レベルをクエリするだけでなく、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。

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/clone_pdfx` **トピック内にあります。**

これは、*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。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。

12.5 PDF/VT による可変・トランザクション印刷

12.5.1 PDF/VT 規格

注 PDF/VT 規格に関する一般情報が PDFlib Web サイトにあります。PDF/VT に初めてふれる方には、PDF 協会が発行した文書「PDF/VT Application notes」も推奨します（www.pdfa.org/publication/pdfvt-application-notes/ 参照）。

PDF/VT 規格は、2010 年に ISO 16612-2 として発行されています。これは「さまざまな環境で可変文書印刷（VDP）を可能にするよう設計された」ものです。PDF/VT 文書は、最終的なコンテンツエレメント群と、関連するメタデータとを内容としますが、変数やテンプレートを含んではいません。PDF/VT は、PDF/X-4/4p と PDF/X-5g/pg に基づいており、さらに透過とレイヤーを含めて PDF 1.6 の諸機能に対応しています。PDF/X の要件に加えて、PDF/VT 規格は、大量パーソナライズ印刷の要請を満たす補足機能を PDF に追加しています。PDF/VT は、電子印刷ファイルの高パフォーマンスレンダリングを可能にするために、効果的なリソース管理を PDF に追加しています。

PDF/VT-1 準拠レベル ISO 16612-2 は、いくつかの PDF/VT 準拠レベルを仕様化しており、これらはすべて PDF 1.6 に基づいています：

- PDF/VT-1：単一ファイル交換のために設計されており、PDF/X-4 に基づいています（PDF/X-4p は許容されません）。1 個の PDF 文書を表現するために必要なリソースのすべてが、単一の PDF/VT-1 ファイル内に含まれています。
- PDF/VT-2（廃止）：複数ファイル交換のために設計されており、PDF/X-4p・PDF/X-5g・PDF/X-5pg のうちの 1 つに基づいています。PDF/VT-2 文書は、外部 ICC プロファイルか外部ページコンテンツまたは両方を参照することができます。PDF/VT 文書と、そのすべての参照された PDF ファイルと外部 ICC プロファイルを、まとめて PDF/VT-2 ファイル集合といいます。

12.5.2 PDF/VT の諸概念

この項では、PDF/VT が基づいている技術的諸概念のあらましを提供します。

文書部分ヒエラルキー 文書部分（DPart）ヒエラルキーは、1 個の PDF/VT ファイル内の文書群ないし文書の部分群の並びと関係を指定します。代表的なシナリオでは、この PDF/VT ファイルは、多くの受領者のための下位文書群を内容としており、そしてそれぞれの文書部分は、個々の受領者のための下位文書を構成するページ群に照応しています。受領者にページを割り当てるだけでなく、この文書部分ヒエラルキーは、もっと複雑な構造を反映することもできます。たとえば、受領者をその住所内の郵便番号に従ってグループ化したり、郵便番号を州に従ってまとめたり、州を国に従ってまとめたりすることができます。この種の文書組織化は、その文書内のすべてのページを包含する 1 個の樹状構造を作成します。このツリーの要素群を DPart ノードといい、そこではそれぞれの内部ノードは他の DPart ノード群を内容とし、それぞれの葉ノードは 1 つの受領者のための 1 個ないし複数のページを指定します。

PDF/VT ファイル内の文書部分ヒエラルキーを用いて、ページにアクセスすることもできます。これは、ページ番号やページラベルによるアクセスといった他の方式のかわりに使えます。この DPart ヒエラルキーは、PDF/VT ファイル内では必須です。そのオプショナルなレコードレベル値は、個別の受領者に対するレコードに照応する DPart ヒエラルキー内のレベルを選択します。これはスコープヒントのために意味を持ちます（後述）。

文書部分メタデータ　文書ツリー内のルートノードから葉群まで下る文書部分ヒエラルキー内の各ノードは、文書部分メタデータ（DPM）を含むことができます。これを使うと、1 つの受領者の文書とその部分群に関する情報を伝えることができます。とりわけ、印刷のために意味を持つ諸特性（ある文書部分の印刷部数など）や、受領者に関する情報（その照応する郵便番号など）を DPM 内に符号化することが可能です。

JDF（または他の）印刷メタデータは、PDF/VT では必須ではありませんが、JDF 対応ワークフローにおいては実用的な価値を加えます。PDF/VT 規格は、文書部分メタデータを外部 XML 文書として表現するための方式も仕様化しています。

反復するグラフィカル内容に対する最適化　印刷要素が複数のページ上で再利用されることはよくあります。たとえば、1 つの送付文書のすべてのページ上に現れる企業ロゴや製品画像です。反復するグラフィカル内容の処理を最適化することは、印刷ファイルのファイルサイズと処理速度を向上させるための重要な戦略です。PDF は、ファイルサイズを最適化するための手段として、つねに XObject をサポートしてきました。これは、1 個の印刷要素のために必要なデータをそのファイル内に 1 回だけ入れ込んで、複数のページ（または同一ページ上の複数のインスタンス）からこのデータへの参照を許すものです。XObject は、ラスタ画像か、任意のテキスト・ベクトルグラフィックコンテンツを内容とすることができます。PDF における XObject は、1 個の文書の全体のサイズを最適化することに主眼を置いていますが、PDF は現在までのところ、反復するページコンテンツのレンダリングを最適化するための手段を一切含んでいません。たとえばある特定のページ上のある画像が、後で同一文書内の別のページ上に再び現れるということや、あるいは次の印刷ジョブ内に現れるかもしれないということを、消費ソフトウェアに対して告げることのできるものが、PDF 内には何もないのです。PDF/VT は、PDF における XObject の既存の概念を拡張して、印刷パフォーマンスを最適化するための以下の手段を追加します：

- **一意識別**：XObject に対して、すべての文書を通じて一意である識別子（*GTS_XID* という）を割り当てることができます。この識別子はキャッシングの実装のために利用可能です。なぜならこの実装は、同等な XObject を識別する必要があるからです。簡単な言葉で言えば、ジョブ 1 のためにすでに処理されている、かつジョブ 2 内で再利用されることが発見されたグラフィックについては、再度リップする必要がなく、そのレンダリング結果をキャッシュから採ることができます。
- **スコープヒントと環境コンテキスト**：XObject が、そのグラフィカル内容が再利用されるページ群か文書群の範囲に関する情報（*GTS_Scope* という）を含むことができます。これによって XObject は、キャッシュ内のそのレンダリング結果の有用継続期間に関する情報を伴うことができます：このコンテンツはカレント受領者のためにだけ再利用されるのか、それとも同一ファイルないしファイルストリーム内のどこか別の所で再利用されるのか、あるいは全く再利用されないのか。環境コンテキスト（*GTS_Env* という）が与えられると、XObject はグローバル用途を、すなわち、それが複数の PDF/VT インスタンス内で再利用されるということを指定することができます。環境コンテキスト文字列について制限は全くありません。たとえば、顧客名やジョブ名を用いてその環境を識別できます。
- **カプセル化ヒント**：RIP 内の XObject キャッシングアルゴリズムは、XObject の、その呼び出しコンテキストと、その同一ページ（または他のページ群。たとえば複数のページを同一シート上に重ね合わせる場合）上の既存の印刷要素群との、相互作用を考慮に入れる必要があります。たとえばもしも、ある XObject が色や線幅を指定しておらず、それが参照される時点において有効な色と線幅に基づいてその体裁が変わるならば、レンダリング結果をキャッシュしても、この変わる体裁のせいで効果的ではありません。もしもその XObject が透過要素を含んでおり、その既存の背景をその XObject とブレンドしなければならない状態の場合にも、同様の状況が起こります。RIP 内にお

ける XObject のキャッシングを支援するために、PDF/VT はカプセル化 XObject という概念を導入しており、それがそうであることを標識できるようにしています（***GTS_Encapsulated*** キーを用いて）。カプセル化 XObject は、RIP 内におけるキャッシングを支援する特定の諸規則を満たす必要があります。

これらのエントリはすべてオプショナルです：PDF/VT は、反復するグラフィカル内容の最適化をいずれも必須としていませんが、それらを利用すれば、PDF/VT 対応 RIP を用いたときに目ざましい印刷パフォーマンス向上が実現します。

12.5.3 PDF/VT-1 と PDF/VT-2 を生成するための諸規則の要約

クックブック **PDF/VT-1 を生成するためのコードサンプルが、PDFlib クックブックの pdfvt カテゴリ内にあります。**

PDFlib を使って PDF/VT-1・PDF/VT-2 を作成することは、以下の手段で実現できます：

- PDF/VTはPDF/Xに基づいていますので、基礎をなすPDF/X準拠レベルのためのすべての要件を満たす必要があります。***pdfvt・pdfx*** 文書オプションで、然るべき値を用いる必要があります。
- 文書部分群を指定する必要があります。***DPart*** API を用います。その文書部分ヒエラルキー内の各ノードが、オプショナルに DPM メタデータを伴うこともできます。その DPart ツリー内のレコードレベルを指定することもできます。
- 反復するグラフィカル内容のためのスコープヒントを与えるべきです。
- その文書が透過を含む場合には、カプセル化 XObject のための諸条件を満たすために追加の努力を払うべきです。
- 既存の PDF 文書からページを取り込む際には、追加の諸規則が適用されます（358 ページ「12.5.7 PDF/X・PDF/VT 文書を PDI で取り込む」参照）。

表 12.23 に、342 ページ「12.4.2 一般的要件」で挙げた、その照応する PDF/X の要件に加えて、PDF/VT 準拠出力を生成するための必須操作とオプショナルな操作を要約します。これらの項目は、特記ない限り、PDF/VT-1 と PDF/VT-2 の両方に対して適用されます。特定の諸側面については、以下に続く各項で詳しく説明します。

表 12.23 PDF/VT-1・PDF/VT-2 準拠のための必須操作とオプショナルな操作

項目	PDF/VT 準拠のための PDFlib の要件
準拠レベル	PDF_begin_document( ) の pdfvt オプションを、必要な PDF/VT 準拠レベルに設定する必要があります。例：pdfvt=PDF/VT-1 PDF/VT-1：オプション pdfx=PDF/X-4 が自動的に設定されます。pdfx オプションに対して他の値を与えることはエラーです。 PDF/VT-2：pdfx オプションにも、値 PDF/X-4p・PDF/X-5g・PDF/X-5pg のいずれか 1 つを指定する必要があります。
文書部分ヒエラルキー	文書部分ヒエラルキーを指定する必要があります。これは以下の操作を必要とします： ► PDF_begin_document( )：nodenamelist オプションが、DPart ツリーのすべてのレベルのための名前を指定する必要があります。recordlevel オプションを用いて、受領者レコード群に照応する DPart ツリーのレベルを指定することもできます。 ► PDF_begin_dpart( )・PDF_end_dpart( )：文書部分ヒエラルキーを構築する必要があります。 ► 文書部分群が、オプショナルに文書部分メタデータ（DPM）を伴うこともできます。DPM を生成するには、POCA インタフェースを用いて DPM 辞書を構築します。

表 12.23 PDF/VT-1・PDF/VT-2 準拠のための必須操作とオプショナルな操作

項目	PDF/VT 準拠のための PDFlib の要件
反復するグラフィカル内容のためのスコープヒント	PDF_load_image( )・PDF_open_pdi_page( )・PDF_begin_template_ext( )、および PDF_load_graphics( ) の templateoptions オプション：pdfvt オプションに scope サブオプションを、値 unknown・singleuse・record・file・stream・global のうちの 1 つとともに与えることによって、反復する画像・テンプレート・取り込み PDF ページのための用途ヒントを与える必要があります。 scope=stream と scope=global に対しては、サブオプション environment が PDF/VT 環境コンテキストを指定する必要があります。これはすなわち、PDF/VT 準拠のリーダが、関連する XObject 群を管理するための管理インタフェースを提供するために使える識別子です。たとえば顧客名やジョブ名を用いてその環境を識別できます。
外部グラフィック内容（参照）	PDF/VT-1 と、PDF/X-4p に基づいた PDF/VT-2：PDF_begin_template_ext( )・PDF_open_pdi_page( )・PDF_load_graphics( )：reference オプションを避けるべきです。なぜならこれは、基礎をなす PDF/X 規格群で許容されないからです。
カプセル化*XObject*	PDF_begin_document( )：その文書が透過要素を一切含んでいない場合には、オプション usestransparency=false を与えることによって、透過グループを持たないすべてのフォーム XObject を PDFlib がカプセル化できるようにするべきです。 PDF_load_image( )：画像が、renderingintent か mask オプションを用いるべきです。 そうでなければ、357 ページ「12.5.6 カプセル化 XObject」の諸規則に従うことによって、できるだけ多くの XObject がカプセル化されていると標識されることを可能にするべきです。

PDF/VT・PDF/A 両立文書を作成　PDF/VT 印刷文書でありながら、同時にアーカイビングのための PDF/A にも準拠している文書を作成すると、有用なこともあるでしょう（PDF/A に関する詳しい情報は 329 ページ「12.3 PDF/A によるアーカイビング」を参照してください）。このような二重用途の文書を作成する際には以下に留意してください：

- PDF/A は外部参照を一切許容しませんので、PDF/VT-1 のみが PDF/A と両立可能であり、PDF/VT-2 は不可能です。以下の文書オプションを用いれば、PDF/VT・PDF/A 両立文書を作成できます：

```
ret = p.begin_document("combo.pdf", "pdfx=PDF/X-4 pdfvt=PDF/VT-1 pdfa=PDF/A-2b");
```

- このマニュアル内で PDF/VT と PDF/A に対して述べた制約に従う必要があり、かつ、両方の規格で許容されている機能だけを使えます。
- 取り込まれる PDF 文書が、受け付け可能であるためには、PDF/X 規格と PDF/A 規格の両方を遵守している必要があります。

12.5.4 文書部分ヒエラルキーと文書部分メタデータ（DPM）

クックブック　文書部分ヒエラルキーを持った PDF/VT を作成するためのコードサンプルが、PDFlib クックブックの pdfvt カテゴリにあります。

文書部分ヒエラルキーを作成　文書内のすべてのページを、文書部分ヒエラルキー内に組織化する必要があります。インボイス等シンプルな場合においては、このツリー構造は、*root* レベルと *recipient* レベルの 2 つのレベルから成ります。この文書内のページ群は、受領者レコード群の線形配列を構成しています。ここで各レコードは 1 個ないし複数のページを内容としています。この文書部分ヒエラルキーの構造を、文書オプションリストで指定する必要があります。例：

```
if (p.begin_document(outfile,
        "pdfvt=PDF/VT-1 nodenamelist={root recipient} recordlevel=1") == -1)
```

もっと複雑な文書は、もっと深い文書ヒエラルキーを必要とするかもしれません。たとえば、カスタマイズされた小冊子が、前付・本文・後付部から成る場合です。これらは *docpart* レベル内で保持されます：

```
if (p.begin_document(outfile,
        "pdfvt=PDF/VT-1 nodenamelist={root recipient docpart} recordlevel=1") == -1)
```

この *recordlevel* オプションは、その受領者かレコードレベルを発見できる場所の、*nodenamelist* 内でのゼロベースの番号を指定します。これは *scope=record* オプションに対して意味を持ちます。

ヒエラルキーの構造を定義した後は、ページ群を文書ヒエラルキーノード群内でグループ化する必要があります。これを達成するには *PDF_begin/end_dpart()* を用います：

```
/* DPartヒエラルキーのルートノードを作成 */
p.begin_dpart("");
        p.begin_dpart("");          /* 受領者レベルに新規ノードを作成 */
            p.begin_page_ext(...);  /* 1個ないし複数のページを作成 */
            ...
            p.end_page_ext(...);
        p.end_dpart("");            /* 受領者ノードを閉じる */

        p.begin_dpart("");          /* 受領者レベルに次のノードを作成 */
            p.begin_page_ext(...);  /* 1個ないし複数のページを作成 */
            ...
            p.end_page_ext(...);
        p.end_dpart("");            /* 受領者ノードを閉じる */
/* ルートノードを閉じる */
p.end_dpart("");
```

この *PDF_begin/end_dpart()* への呼び出し群が、*nodenamelist* オプションに従ったツリー構造を作成する必要があります。すなわち、その最大ネストレベルが、この *nodenamelist* 配列内のエントリの数に照応する必要があります。

PDF_begin_document() の *groups* オプションと、*PDF_begin/end_page_ext()* の *group*・*pagenumber* オプションは、PDF/VT モードでは許容されません。なぜなら、これらは DPart 構造を阻害するからです。

文書部分メタデータ（DPM）を作成　文書部分ヒエラルキー内の各ノードに対して、1 つの葉ノードに照応するページ群に、またはこのノード配下の下位ツリー全体に適用されるメタデータ情報を指定できます。PDF/VT は特定の種類のメタデータを何ら規定していませんが、これは CIP4 機構によって発行された JDF 規格とともに用いられることを意図しています。とりわけ、「ICS − Common Metadata for Document Production Workflows」（CIP4 の Web サイトから入手可能）という文書が、印刷業務ワークフローで用いるためのメタデータを記述しています。この CIP4 メタデータ形式は、代表的な PDF データ型を活用しており、そのメタデータは、PDF/VT 規格で仕様化されている形で PDF 出力へ書き出されます。

PDFlib ユーザーは、POCA（PDF Object Creation API ＝ PDF オブジェクト作成 API）インタフェースを使って、PDF 辞書・配列やその他のデータ型から成る任意のデータ構造を

作成することもできます。DPM メタデータに対する最上位レベル辞書を、*PDF_begin/end_dpart()* の *dpm* オプションへ渡すことができます：

```
dpm           = p.poca_new("type=dict usage=dpm");
cip4_root     = p.poca_new("type=dict usage=dpm");
cip4_metadata = p.poca_new("type=dict usage=dpm");

p.poca_insert(dpm,           "type=dict   key=CIP4_Root value=" + cip4_root);
p.poca_insert(cip4_root,     "type=dict   key=CIP4_Metadata value=" + cip4_metadata);
p.poca_insert(cip4_metadata, "type=string key=CIP4_Conformance value=base");
p.poca_insert(cip4_metadata, "type=string key=CIP4_Creator value=starter_pdfvt1");
p.poca_insert(cip4_metadata, "type=string key=CIP4_JobID value={Kraxi Systems invoice}");

/* DPartヒエラルキー内にノードを作成してDPMメタデータを追加 */
p.begin_dpart("dpm=" + dpm);

p.poca_delete(dpm, "recursive=true");
```

クックブック PDFlib pCOS クックブックに、PDF/VT 文書から DPM を取得して、その照応する XML 表現を作成するためのコードサンプルがあります。

12.5.5 反復するグラフィカル内容のためのスコープヒント

スコープヒント *pdfvt* オプションの *scope* サブオプションを、然るべき値とともに、*PDF_load_image()*・*PDF_load_graphics()*・*PDF_open_pdi_page()*・*PDF_begin_template_ext()* に与えるべきです。これを達成するには、カレント PDF 文書内で、または（なるべくは）複数文書にわたって、画像・ページ・テンプレートが再利用される場所と回数をクライアントアプリケーションが知る必要があります。この *scope* オプションは必須ではありませんが、これは強く推奨されます。なぜならこれは、RIP のための重要な最適化情報を提供するからです。そのような情報が得られない場合には、間違っているおそれのある値を与えるよりも、この *scope* オプションを省略するか、値 *unknown* とともに指定するべきです。

スコープ *singleuse*・*record*・*file* に対しては、PDFlib は、そのスコープ値がその XObject の実際の使用と整合しているかどうかをチェックして、もしもそのスコープが大きすぎるときは、ログファイル内にたとえば以下のような警告を出力します：

```
XObject with handle 9 was assigned PDF/VT scope 'record', but was used only once
(use 'scope=singleuse' instead)
```

このような警告を得たときには、もっと然るべきスコープ値を、その照応する画像かグラフィックか取り込み PDF ページかテンプレートに割り当てることによって、RIP 内の不必要なキャッシングを回避し、ひいては印刷パフォーマンスを向上させることができるかどうかをチェックするべきです。

ある XObject への参照が少なすぎる場合（その *scope* オプションと比較して）には、クライアントコードに改善の余地がありうることを示唆する警告が発生するだけですが、ある XObject への参照が多すぎる場合にはエラーになります。なぜならこれは PDF/VT 規格に違反するからです。より具体的には、以下の状況で例外が発生します：

- オプション *scope=singleuse* が与えられており、かつ、その XObject がその文書内で複数回使用されている。
- オプション *scope=record* が与えられており、かつ、その XObject が複数の受領者レコード内で使用されている。

scope=stream と *scope=global* に対しては、*environment* サブオプションを与える必要があります。これは、キャッシュされたオブジェクトを複数の文書にわたって識別するために用いられる、然るべき文字列を内容とするべきです。ワークフローに応じて、顧客参照やジョブ参照をこのオプションのために使うとよいでしょう。たとえば、複数の印刷ジョブにわたって頻繁に柄笑える企業ロゴを、この方法で識別するとよいでしょう。

一意な ID　PDFlib は、すべての取り込まれた PDF ページ・画像・グラフィックに対して、一意な ID を自動的に割り当てます（グラフィックに対しては、*templateoptions* オプションを与える必要があります）。等価な画像・ページ・グラフィック群に等しい ID 値が割り当てられることにより、RIP 内での効果的なキャッシングを可能にしています。テンプレートに対しては、一意な ID を、*pdfvt* オプションの *xid* サブオプションを通じてユーザーが与えるべきです。テンプレートに対する識別子は、PDF/VT に従って等価な PDF フォーム XObject を作成するテンプレート定義群（すなわち、同一の視覚的出力を持つテンプレート）に対して等しくするべきです。等価でないテンプレートは、異なる識別子を持つか、識別子を全く持たないようにする必要があります。

アプリケーションが、PDFlib によって作成された一意な ID 文字列を必要とする場合、これを取得するには、画像とテンプレートに対しては *PDF_info_image()* の、グラフィックに対しては *PDF_info_graphics()* の、取り込まれた PDF ページに対しては *PDF_info_pdi_page()* の、*xid* オプションを用います。

12.5.6 カプセル化 XObject

XObject をカプセル化すると、リッピングのパフォーマンスが大いに向上します。ただし XObject をカプセル化できるのは、完全に不透明な文書内か、（透過が使われている文書内では）特定の要件が満たされた場合のみです。

文書内の透過　その文書内で透過が使われるかどうかを PDFlib はあらかじめ知りませんので、ヒントとして *PDF_begin_document()* に *usestransparency* オプションを与えることもできます。クライアントがこのオプションを用いて、透過が一切使われていないと宣明した場合には、すべての XObject をカプセル化できます。その文書内で透過が使われている場合には、その XObject 群がカプセル化されるようにするには、クライアントは特定の諸規則に従う必要があります。*usestransparency=false* が与えられたにもかかわらず、その文書が透過要素を含んでいた場合には、PDFlib は例外を発生させます。

以下の条件のうちの 1 つないし複数が真であれば、その生成文書内で透過が使われていると見なされます：

- *PDF_load_image()* か *PDF_fill_imageblock()* が、以下のいずれかのオプションとともに呼び出されている：
  - その画像がアルファチャンネルを含んでおり、かつ *ignoremask* オプションが *false*。
  - ピクセルあたりビット数が複数である画像のハンドルを持った *masked* オプション。
- *PDF_load_graphics()* を用いて取り込まれたグラフィックが、何らかの透過要素を含んでいる。
- *PDF_create_gstate()* が、以下のいずれかのオプションとともに呼び出されている：
  - オプションリストを持った（すなわち、キーワード *none* を持っているのではない）*softmask* オプション。
  - 値 1 以外を持ったオプション *opacityfill* か *opacitystroke* のいずれか 1 つ。
  - 値 *None*・*Normal* 以外を持った *blendmode* オプション。

- *PDF_open_pdi_page()*か*PDF_fill_pdfblock()*を用いて取り込まれたPDFページが、何らかの透過要素を含んでいる。取り込まれた PDF ページ内の透過を識別するには、pCOS インタフェースと *usespagetransparency* 擬似オブジェクトを用います（詳しくは pCOS パスリファレンスを参照してください）。

XObject をカプセル化　PDFlib は、多くの XObject を以下の規則に従ってカプセル化します：

- *PDF_load_image()* か *PDF_fill_imageblock()* を用いて作成される画像 XObject は、以下の条件のいずれかが真であれば、カプセル化されます：
  - *PDF_load_image()* の *renderingintent* オプションが値 *Auto* 以外とともに与えられている。
  - *PDF_load_image()* の *mask* オプションが値 *true* とともに与えられている。
- *PDF_begin_template_ext()* を用いて作成されるフォーム XObject は、もしも文書オプション *usestransparency=false* が与えられているか、*transparencygroup* オプションがサブオプション *colorspace* と *isolated=true* とともに与えられている場合には、カプセル化されます。
- *PDF_load_graphics()*か*PDF_load_image()*で*templateoptions*オプションを付けて生成されたフォームXObjectはつねにカプセル化されます。文書オプション *usestransparency=true*が与えられている場合には、然るべき色空間を持った透過グループがそのフォーム XObject に紐付けられます。そうでない場合にはグループは一切生成されません。
- *PDF_open_pdi_page()* を用いて生成されたフォーム XObject は、その取り込み PDF ページがレイヤーを一切含んでいないか、または *PDF_open_pdi_document()* が *uselayers=false* とともに呼び出されている場合には、カプセル化されます。文書オプション *usestransparency= true* が与えられている場合には、然るべき色空間を持った透過グループがそのフォーム XObject に紐付けられます。そうでない場合にはグループは一切生成されません。取り込むページが、すでにカプセル化されている XObject を含んでいる場合には、そのページを取り込む際に、このプロパティは変更されずに温存されます。

XObject をカプセル化できないときは、警告がログファイルへ書き出されます。

カプセル化 XObject に対するスコープ　*PDF_load_image()*・*PDF_fill_imageblock()*・*PDF_load_graphics()*・*PDF_begin_template_ext()* を用いて作成するすべてのカプセル化 XObject に対して、*pdfvt* オプションの *scope* サブオプションを与えることを推奨します。この *scope* サブオプションは、非カプセル化 XObject に対して与えることも許容されていますので、もしも画像かグラフィックか取り込まれるページかテンプレートの継続期間がわかっているならば、その生成される XObject のカプセル化ステータスにかかわらず、すべての関連する API 呼び出しにこれを与えることもできます。

12.5.7 PDF/X・PDF/VT 文書を PDI で取り込む

既存の PDF 文書からのページを PDF/VT 準拠の出力文書内へ取り込もうとする際には、特別な諸規則が適用されます（PDI について詳しくは 213 ページ「8.3 PDF ページを PDI で取り込む」を参照してください）。すべての取り込まれる文書は、表 12.24 に従って互換な PDF/X・PDF/VT 準拠レベルに準拠している必要があります。もしもある特定の PDF/VT 準拠レベルが PDFlib 内で構成されており、かつ取り込まれる文書がその互換 PDF/X レベルのうちの 1 つを遵守しているならば、その生成出力が、その選択された PDF/VT 準

拠レベルに準拠することが保証されます。取り込まれる文書は、受け付け可能な PDF/X レベルのうちの 1 つを遵守していない場合には、拒絶されます。

表 12.24 さまざまな PDF/VT 出力レベルに対する互換 PDF/X・PDF/VT 入力レベル

	取り込まれる文書の PDF/X・PDF/VT レベル			
PDF/VT 出力レベル	**PDF/X-1a、PDF/X-3、PDF/X-4、PDF/VT-1**	**PDF/X-4p、PDF/X-4p に基づく PDF/VT-2**	**PDF/X-5g、PDF/X-5g に基づく PDF/VT-2**	**PDF/X-5pg、PDF/X-5pg に基づく PDF/VT-2**
***PDF/VT-1*（つねに*PDF/X-4*に基づく）**	許容	許容		
PDF/X-4p*に基づく*PDF/VT-2	許容	許容[1]		
PDF/X-5g*に基づく*PDF/VT-2	許容	許容	許容[2]	許容[2]
PDF/X-5pg*に基づく*PDF/VT-2	許容	許容[1]	許容[2]	許容[1,2]

1. PDF_process_pdi( ) で action=copyoutputintent を指定すると、外部出力インテント ICC プロファイルへの参照がコピーされます。
2. 取り込まれたページが、参照された XObject を含んでいるときは、PDF_open_pdi_page( ) は代理と参照の両方をコピーします。

PDF/VT 取り込みでの DPart と DPM の制約 PDF/VT 文書を取り込む際には、以下の制約に留意してください：文書部分ヒエラルキー（DPart）と文書部分メタデータ（DPM）は取り込まれません。それらを、pCOS を用いてクエリして、*PDF_begin/end_dpart()* と POCA 関数群を用いて再構築することが可能です。

12.6 PDF/UA によるユニバーサルアクセシビリティ

12.6.1 PDF/UA-1 規格

注 PDF/UA 規格に関する一般情報が PDFlib ウェブサイトにあります。

注 PDF/UA に初めてふれる方には、PDF 協会が発行した文書「PDF/UA in a Nutshell」を推奨します（www.pdfa.org/publication/pdfua-in-a-nutshell/ 参照）。

PDF/UA-1 は、PDF 文書のユニバーサルアクセシビリティを向上させるために、タグ付き PDF のエレメント群や、その他の文書の諸側面の詳細を仕様化したものです。PDF/UA-1 は、WCAG 2.0（Web コンテンツアクセシビリティガイドライン群[1]）を PDF 文書に適用したものと捉えることができます。

PDF/UA-1 は、PDF 1.7 と ISO 32000-1 で定義されているタグ付き PDF に基づいています。これは PDF ファイル形式に何らかの新たな機能を加えたのではなく、主に、いくつかのアクセシビリティとタグ付けの側面が PDF 1.7 ではオプショナルであるのを必須としたものです。またこれは、構造エレメントのさまざまな種別どうしの関係を明確化しています。

ISO 14289-1 で定義された PDF/UA-1 PDF/UA-1 は、PDF 文書のアクセシビリティを向上させるために以下の手段を用いています：

- 文書構造に関する特定の要件を強制。例：タグ付け規則など。
- 特定の補足情報を必須化。例：メタデータ、グラフィックのための代替テキスト。
- アクセシビリティの目的を阻害する特定の PDF 要素を防止。

PDFlib の PDF/UA-1 の実装は、以下の文書に基づいています：

- PDF/UA-1 規格（ISO 14289-1:2014）

PDF/UA-1・PDF/A 両立文書を作成 PDF/UA 文書でありながら、同時にアーカイビングのための PDF/A にも準拠している文書を作成すると、有用なこともあるでしょう（329 ページ「12.3 PDF/A によるアーカイビング」を参照）。実際、PDF/A-1a/2a/3a を作成したい場合には、PDF/UA の要件を遵守することによって、生成される文書のアクセシビリティを向上させることを推奨します。PDF/A・PDF/UA-1 両立文書を作成するには、*PDF_begin_document()* の *pdfa*・*pdfua* オプションに対して適切な値を与えます。例：

```
ret = p.begin_document("combo.pdf", "pdfa=PDF/A-2a pdfua=PDF/UA-1 lang=en");
```

このような二重用途の文書を作成する際には、両方の規格に準拠する両立ファイルは、両方の規格によって課せられる要件に従う必要がありますので、留意してください。その PDF 互換レベルは、関与する規格群の PDF 互換性の最小値です：取り込まれる PDF 文書が、PDF/UA 規格と PDF/A 規格の両方を遵守している必要があります。

タグ付き PDF 出力のためには、PDF/A-1a を避けて、もっと新しい PDF/A-2a か PDF/A-3a 規格のほうで作業することを推奨します。なぜなら、PDF/UA-1 と PDF/A-1a の間には小さな衝突が 1 つあるからです：

- PDF/UA-1 は、注釈の存在下では、ページオプション *taborder=structure* を必須とします。しかし、この *taborder* オプションは PDF 1.5 を必要としますので、PDF/A-1a では使えません。結果として、PDF/A-1a・PDF/UA-1 両立文書内では注釈が使えません。

1. *www.w3.org/TR/WCAG20/* 参照

クックブック PDFlib クックブック内の invoice_pdfua1 サンプルが、PDF/UA-1・PDF/A-2a 両立文書を作成する方法を演示しています。

PDFlib は、技術的な PDF/UA の要件への違反を検出した場合には、例外を発生させます。この場合には PDF 出力は何も作成されません。表 12.25 に、PDF/UA 準拠出力を作成するための一般的要件を挙げます。

表 12.25 PDF/UA 準拠のための一般的要件

項目	PDF/UA-1 準拠のための PDFlib の要件
一般的文書要件	PDF_begin/end_document( ) : pdfua オプションを PDF/UA-1 に設定する必要があります。これはタグ付き PDF を必須とします（tagged オプションは自動的に設定されます）。 PDF 1.7ext3 以上を要する操作（リッチメディア注釈・ポートフォリオなど）を避ける必要があります。lang オプションが必須です。 オプション viewerpreferences : サブオプション displaydoctitle に対して true のみが許容されます。 オプション permissions : キーワード noaccessible が許容されません。
タグ付け	すべてのタグ付け規則（298 ページ「構造エレメントに対するネスト規則」参照）に従う必要があります。checktags 文書オプションを none に設定してはいけません。
フォント	フォントオプション embedding を true にする必要があります。オプション unicodemap=false と dropcorewidths= true が許容されません。埋め込みは PDF コアフォントについても必須です。
テキスト出力と*PUA Unicode*キャラクタ	PUA Unicode キャラクタ（ロゴや記号など）が、適切な置換テキストを持つ必要があります。それを囲うコンテンツアイテムに対する PDF_begin_item( ) の ActualText オプションで指定するか、その照応する出力関数の同等の tag オプションで指定します（336 ページ「PUA キャラクタ」参照）。
不可視テキスト	埋め込み必須は、不可視テキスト（主として OCR の生成結果のために有用）に対してのみ用いられているフォントについてだけはあてはまりません。不可視とは textrendering=3 です。これは、optimizeinvisible オプションを用いて制御することもできます。 不可視テキストは、レンダリングされる等価物を一切持たない場合には、Artifact としてマークする必要があります。
レイヤー	レイヤーを使えますが、ただし PDF_define_layer( ) のいくつかのオプションを避ける必要があります（PDFlib API リファレンス参照）。
外部コンテンツ	PDF_begin_template_ext( )・PDF_load_graphics( )・PDF_open_pdi_page( ) : reference オプションを避ける必要があります。
***PDF*取り込み**	PDF_open_pdi_document( ) : 取り込まれる文書が PDF/UA に準拠している必要があります。364 ページ「12.6.4 PDF/UA 文書を PDI で取り込む」を参照してください。
メタデータ	PDF_set_info( ) で key=Title を、または PDF_begin/end_document( ) で metadata に与える XMP 内で dc:title を、空でない値とともに与える必要があります。

12.6.2 タグ付けの要件

PDF/UA は、タグ付き PDF に基づいていますので、292 ページ「11.3 タグ付き PDF の基礎」で説明したタグ付き PDF の要件に従う必要があります。ただし、PDF/UA-1 はこれに加えて、タグ付けの要件を多数課することによって、アクセシビリティの向上を図っています。

意味付けの要件　ユーザーは、文書ヒエラルキーを作成し、以下に挙げる意味付け規則に従う必要があります。適切な構造エレメントを選ぶことは、PDF/UA 規格準拠の決定的な構成要素です。これらの側面に責任を負うのはアプリケーション側であることを理解することが重要です。なぜなら PDFlib はこれらをチェックできないからです：

- タグ付けは、その文書構造に対して適切な構造エレメントを用いる必要があります：それが見出しであるなら、それを見出しとしてタグ付けする必要があります。それが表であるなら、それを表としてタグ付けする必要があります。
- その文書にとって意味をなさないコンテンツを、その文書ヒエラルキー内に含めてはいけません。それは *Artifact* としてタグ付けする必要があります。
- 構造エレメント群を、論理読み取り順序に配列する必要があります。これは、そのタグ群を読み順に作成することによって最も容易に達成できます。ただし、複雑なレイアウトに対しては、*PDF_activate_item( )* を用いてこれを達成することもできます（318 ページ「11.4.4 コンテンツを順序にとらわれず作成」参照）。
- コンテンツの色や形式やレイアウトのせいで、意図する情報がそのままではアクセシブルでないコンテンツに、適切にタグ付けする必要があります。
- グラフィック内で表現されているテキストは、もしそれがテキストを自然言語で含まない場合には（フォントサンプルやスクリプトサンプルなど）、説明を持った *Alt* オプションを必須とします。
- 画像のキャプションを *Caption* タグでマークする必要があります。
- コンテンツが箇条書きとして読まれることを意図している場合には、箇条書きエレメント（*L*）を作成する必要があります。
- ヘッダとフッタを *Artifact* として、*artifacttype=Pagination* と *artifactsubtype=Header* か *Footer* とともにタグ付けする必要があります。
- 論理的に一体であるグラフィカル要素群のグループに対しては、ただ 1 つの *Figure* を作成する必要があります。
- 脚注・後注・注ラベルと、その文書内の場所への参照に、適切に *Note* か *Reference* としてタグ付けする必要があります。

注 PDF 協会が発行した文書「Tagged PDF Best Practice Guide: Syntax」は、PDF/UA-1 のためのタグ付き PDF 構造エレメントの正しい使い方に関するガイダンスを提供しています。

タグごとの要件　表 12.26 に、PDF/UA-1 準拠を達成するための、タグごとに対する要件を挙げます。ここに挙げた標準種別へマップされたカスタムエレメント種別についても、これらの規則に従う必要があります。たとえば、*rolemap* オプション内でカスタムタグ *Illustration* が *Figure* へマップされているならば、それも *Figure* に対する条件下に置かれます。

表 12.26 PDF/UA-1 準拠のためのタグごとの要件

エレメント種別	PDF/UA-1 準拠のための PDFlib の要件
標準エレメント種別	PDF_begin_document( )：rolemap オプションが標準エレメント種別をマップしてはいけません。
Figure	オプション Alt か ActualText のうちの 1 つを与える必要があります。
Formula	オプション Alt を与える必要があります。
Table	表エレメントは、論理的な表に対しては作成する必要がありますが、レイアウト目的のために作成される表に対しては作成してはいけません。PDFlib によって組版される表は自動的にタグ付けできます。310 ページ「11.4.1 自動表タグ付け」を参照してください。

表 12.26 PDF/UA-1 準拠のためのタグごとの要件

エレメント種別	PDF/UA-1 準拠のための PDFlib の要件
TH	表はヘッダを含むべきです。TH エレメントに対してはオプション Scope が推奨されます（これは、PDF_fit_table( ) の header オプションが用いられている場合には、自動表タグ付けによって常に生成されます）。
L	エレメント種別 L（箇条書き）は ListNumbering オプションを必須とします。その子 LI（箇条書き項目）群がいずれも Lbl（ラベル）エレメントを含まない場合には、ListNumbering を None にする必要があります。一方、もしも ListNumbering=None でありながら、可視な箇条書きラベルがある場合には、それらをページ装飾としてマークする必要があります。
Note	Note としてタグ付けされる脚注と後注に対しては id オプションが必須です（id はどこでも参照されていませんので、これは何の得ももたらしませんが）。

見出し　すべての見出しに対して、適切な見出しタグを用いる必要があります。PDF 文書内における見出しの階層ネストについては 2 通りのアプローチがあります：

- 強く構造化された文書：グループ化エレメント群が、必要なだけ深くネストすることによって、アーティクル・セクション・下位セクションなどを持ったコンテンツの組織を反映します。各レベルにおいて、そのグループ化エレメントの子群は、1 個の見出し *H* と、そのレベルのコンテンツのための 1 個ないし複数の段落 *P* と、ネストされた下位セクション群のためのさらなるグループ化エレメント群から成るべきです。強い構造は通常、XML 文書において用いられます。
- 弱く構造化された文書：その文書の構造ヒエラルキーが比較的フラットであり、ほんの 1 ～ 2 個のレベルのグループ化エレメント群を持ち、すべての見出し・段落やその他 BLSE 群をそれらの直接の子とします。そのコンテンツの組織が、その論理構造として反映されず、特定の見出しレベル *H1*・*H2*・*H3* などによって表現できます。見出しタグは子孫を一切持てません。弱い文書構造は通常、HTML 内において用いられます。

PDF/UA-1 モードにおいては、この区別を明示するために、*PDF_begin_document( )* で *structuretype* オプションを用いる必要があります。PDFlib は、文書構造の種別に応じて、見出しエレメントの使用に関する以下の規則を強制します。

- すべての文書：
  - すべての見出しタグ内で *Title* オプションを用いることによって、文書のセクションを表示する必要があります（例：「第 1 章」）。
  - 見出しエレメント *H*・*H1*・*H2* などは子孫を一切持ってはいけません。
- 弱く構造化された文書、すなわち *structuretype=weak*（デフォルト）：
  - 見出しの連鎖は、*H1* から始まり、数値レベルを一切スキップしてはいけません。たとえば、*H1 H3* という連鎖は許容されません。
  - 6 個を超える見出しレベルが必要な場合には、追加の見出しレベル *H7*・*H8* などを使うこともできます。これらは標準エレメント種別ではありませんので、*rolemap* オプション内にエントリが必要です。推奨されるマッピングは *P* です。
  - 番号なしの見出しエレメント *H* を使ってはいけません。
- 強く構造化された文書（*structuretype=strong*）：
  - 見出しのために *H* を用いる必要がありますが、ただしその構造ヒエラルキーの各ノード内に複数の *H* タグがあってはいけません。
  - 番号付きの見出しエレメント *H1*・*H2* などを使ってはいけません。

12.6.3 コンテンツ種別ごとの追加の要件

表 12.27 に、さまざまなコンテンツ種別とインタラクティブ要素に関連した PDF/UA-1 の要件と推奨事項を挙げます。

表 12.27 コンテンツ種別とインタラクティブ要素ごとに対する PDF/UA-1 の要件と推奨事項

コンテンツ種別	PDF/UA-1 準拠のための PDFlib の要件
テキスト	ページ上のすべてのテキストの自然言語を宣言する必要があります。テキスト列内の自然言語の変化も宣言する必要があります。この自然元とを宣言するには、PDF_begin_document( ) の lang オプションやその他の手段を用います。詳しくは 304 ページ「言語指定」を参照してください。エレメントの言語属性はそのすべての子孫へ継承されます。
ベクトルグラフィックとラスタ画像	ラスタ画像とベクトルグラフィックに、Artifact か Figure か Formula としてタグ付けする必要があります。これは、PDF_rect( ) などのような低レベルパス構築関数と、PDF_draw_path( ) を用いたパスオブジェクトと、PDF_fit_image( ) や同様の関数にあてはまります。ベクトルグラフィックかラスタ画像を内容として持つ SVG グラフィックも、この要件の対象となります。
取り込まれた *PDF*ページ	PDF_fit_pdi_page( ) を用いて配置された、グラフィックを含んだ PDF ページに、Artifact か Figure としてタグ付けする必要があります。
注釈	ページ上に注釈が存在する場合：PDF_begin/end_page_ext( )：オプション taborder に対して値 structure のみが許容されます。 PDF_create_annotation( ) で type=Link とする場合： ► その注釈は Link 構造エレメント内に含まれている必要があります。 ► PDF_create_action( ) の ismap オプションが、リンク注釈内のアクションに対しては許容されません。 PDF_create_annotation( ) で種別を Link・Popup 以外とする場合： ► その注釈は Annot 構造エレメント内に含まれている必要があります。 ► 可視注釈[1]のためには、サブオプション ActualText を持った contents オプションか tag オプションが必須です。
フォームフィールド	ページ上にフォームフィールドが存在する場合：PDF_begin/end_page_ext( )：オプション taborder に対して値 structure のみが許容されます。 PDF_create_field( ) と、PDF_add_table_cell( ) のオプション fieldname・fieldtype：PDF_create_field( ) か tag オプションを用いて Form タグを作成する必要があります。PDF_create_field( ) と PDF_create_fieldgroup( ) のオプションが必須です。
ページラベル	PDF_begin/end_document( ) で labels オプションを用いて、および PDF_begin/end_page_ext( ) で label オプションを用いて作成されるページラベルは、意味付け的に適切であるべきです。
しおり	PDF_create_bookmark( ) を用いてしおりを生成することが推奨されます。このしおり群は、正しい読み取り順序と、そのコンテンツのネストを反映するべきです。
添付	PDF_load_asset( )：オプション description が推奨されます。添付はそれ自体でアクセシブルであるべきです。

1. 注釈が可視と見なされるのは、その長方形の少なくとも一部分がそのページの CropBox 内にあり、かつ PDF_create_annotation( ) の display オプションが hidden・noview 以外の場合です。

12.6.4 PDF/UA 文書を PDI で取り込む

既存の PDF 文書からのページを PDF/UA 準拠出力文書へ取り込もうとする際には、追加の諸規則が適用されます（PDF 取り込みについて詳しくは 213 ページ「8.3 PDF ページを PDI で取り込む」を参照してください）。既存の PDF 文書からページを取り込むためには、

その取り込まれる文書とページが、以下のクライテリアに従ってカレント文書に互換である必要があります（そうでない場合にはそれは拒絶されます）：

- *PDF_open_pdi_document( )*：PDF/UA-1 文書のみを取り込むことができ、かつ *usetags* オプションを *true* にする必要があります。XMP 文書メタデータ内の標準識別に従って PDF/UA-1 に準拠していない文書は *PDF_open_pdi_document( )* で拒絶されます。
- *PDF_open_pdi_page( )*：取り込まれる文書のロールマップが、*PDF_begin_document( )* の *rolemap* オプションによって与えられたマッピングと互換である必要があります。これはすなわち、カスタムエレメント種別が、*rolemap* オプションと、取り込まれる文書のロールマップ（またはそれ以前に取り込まれた文書のロールマップ）とで、別々の標準種別へマップされていてはいけないということです。
- *PDF_open_pdi_page( )*：取り込まれるページの見出し構造が、生成される文書の構造種別と互換である必要があります。すなわち、*structuretype=weak* の場合には *H1*・*H2* などのみ（*H* は不可）がそのページ上で使われている必要があります：*structuretype=strong* の場合には *H* のみ（*H1*・*H2* は不可）が、その取り込まれるページ上で使われている必要があります：番号付きと番号なしの見出しを両方持ったページは拒絶されます。

PDFlib 内で PDF/UA-1 準拠が構成されており、かつ取り込まれた文書が上記の要件を遵守しているならば、生成される文書も PDFUA-1 に準拠していることが保証されます。

注 PDFlib は、PDF 入力文書の PDF/UA 準拠に関する検証は行わず、任意の入力 PDF 文書を PDF/UA へ変換することもできません。

13 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 ページ・ベクトル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・明細等の発行
- 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。**

13.1 PDFlib Block Plugin をインストール

Block Plugin は、以下のバージョンの Acrobat で動作します（Adobe Reader では動作しません）：

- Windows：Acrobat 8/9/X/XI/DC 32 ビット
- Windows：Acrobat DC 64 ビット
- macOS：Acrobat DC

Acrobat DC には 32 ビット版と 64 ビット版がありますことから、2 種類のインストーラがあります。インストールされている Acrobat のバージョンに合致しているインストーラを使うことが重要です。

Windows で 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。

64 ビット Windows 上の Acrobat 32 ビットの場合、プラグインフォルダは典型的には以下のとおりです：

```
C:\Program Files (x86)\Adobe\Acrobat DC\Acrobat\plug_ins\PDFlib Block Plugin
```

Acrobat 64 ビットの場合、プラグインフォルダは典型的には以下のとおりです：

```
C:\Program Files\Adobe\Acrobat DC\Acrobat\plug_ins\PDFlib Block Plugin
```

macOS で PDFlib Block Plugin をインストール　このプラグインをすべてのユーザーのためにインストールするには以下のように操作します：

- ディスクイメージをダブルクリックすることによってマウントします。プラグインファイル群が入ったフォルダが現れます。
- このプラグインフォルダを、システムの**ライブラリ**フォルダの中の、以下のパスへ複製します（*Plug-ins* フォルダがまだない場合には作成します）：

```
/Library/Application Support/Adobe/Acrobat/XXX/Plug-ins
```

あるいは、以下のように操作することによって、このプラグインを、単独のユーザーのためだけにインストールすることもできます：

- デスクトップをクリックすることによって Finder 内に確実にいるようにしてから、*Option* キーを押しながら**「移動」→「ライブラリ」**を選択することによって、そのユーザーの**ライブラリ**フォルダを開きます。
- プラグインフォルダを、そのユーザーの**ライブラリ**フォルダの中の、以下のパスへ複製します（*Plug-ins* フォルダがまだない場合には作成します）：

```
/Users/<ユーザー名>/Library/Application Support/Adobe/Acrobat/XXX/Plug-ins
```

多言語インタフェース　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。

Windows 版 Acrobat DC のためのサンドボックス保護　Acrobat DC 2020 では、**サンドボックス保護**という新しいセキュリティモデルが導入されました。これを有効にするには**「環境設定」→「セキュリティ（拡張）」→「起動時に保護モードを有効にする（プレビュー）」・「保護されたビュー」**を選択します。これが有効になっていると、さまざまな操作が制限され、文書ウィンドウの上端の黄色い帯にセキュリティメッセージが表示されます。サンドボックス保護について詳しくは：

```
helpx.adobe.com/acrobat/using/whats-new/2020-august.html
www.adobe.com/devnet-docs/acrobatetk/tools/AppSec/sandboxprotections.html
```

サンドボックスが有効になっていると、PDFlib Block Plugin のプレビュー機能に影響します。保護ビューは、デフォルトでは、Acrobat の *AppData* ディレクトリと、temp ディレクトリなどいくつかのディレクトリへのアクセスを許しますが、任意のユーザーディレクトリへのアクセスを許しません。Block Plugin が読み書きできるディレクトリは、保護ビューのデフォルトディレクトリリストに含まれているか、以下の場所にあるポリシーファイルの中で構成（ホワイトリスト）されているものに限られます（Acrobat の 32 ビット版の場合と 64 ビット版の場合）：

```
C:\Program Files (x86)\Adobe\Acrobat DC\Acrobat\PDFlibBlockCustomPolicies.txt
C:\Program Files\Adobe\Acrobat DC\Acrobat\PDFlibBlockCustomPolicies.txt
```

デフォルトではこのポリシーファイルは、以下のディレクトリへのアクセスを許していますが、管理者はこれにディレクトリ名を追加することもできます：

```
; Protected Path Section
FILES_ALLOW_ANY = C:\Users\<username>\*.*
FILES_ALLOW_ANY = C:\Users\Public\*.*
```

もし保護モードか保護ビューが有効にされていて、かつ、使用されているディレクトリがホワイトリストされていない場合には、Block Plugin の機能のうち、プレビューや、ブロックの取り込み／書き出しなどは、失敗する可能性があります。

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、以下をチェックしてください：

- **「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 または Adobe Experience Manager 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。

13.2 ブロック概念の概要

13.2.1 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 PDF ページやベクトル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・画像拡縮といった関連特性群も指定します。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。

13.2.2 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。

定義済みブロックプロパティ　ブロックはページ上の長方形として定義され、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（392 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 内容を持ちます。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。
- **グラフィックブロック**は、ベクトルグラフィックを持ちます。これは、レイアウトアプリケーション内で SVG ファイルを貼り付けるのに似ています。

ブロックは、その種類によって異なるさまざまな定義済みプロパティを持っています。プロパティは、Block Plugin で作成・変更することができます（377 ページ「13.3.2 ブロックプロパティを編集」参照）。定義済みブロックプロパティの全一覧が 395 ページ「13.7 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・ベクトルグラフィックのいずれか）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、その照応する、ブロック流し込み関数内の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。すなわち、ブロック流し込み関数に可変データが与えられていないときや、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る、テキスト・画像・PDF・グラフィックいずれかの内容です。

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（385 ページ「13.5 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　定義済み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、そのブロックに照応するデータを取得するために用いることもできるでしょう。

13.2.3 PDF のフォームフィールドを利用しないのはなぜか

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（382 ページ「13.4 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 13.1 に示すようないくつかの利点があります。

表 13.1 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・スワッシュ文字・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
ベクトルグラフィックを流し込める	―	SVG
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）・DeviceN
PDF 各種規格	―	PDF/A・PDF/X・PDF/VT・PDF/UA
グラフィックやテキストのプロパティを流し込み時にオーバライド可能	―	○
透過内容	―	○
テキストブロック群を連結可能	―	○

13.3 PDFlib Block Plugin でブロックを編集

13.3.1 ブロックを作成

ブロックツールをアクティブにする　PDFlib ブロックを作成するための Block Plugin は、Acrobat における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：

- ブロックアイコン  をクリック。これはAcrobat DC で以下の場所にあります:**「ツール」→「*PDF* を編集」**をクリック。
- メニュー項目**「*PDFlib* ブロック」→「*PDFlib* ブロックツール」**を選択。

ブロックを作成・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サイズのブロックを作成することができます。ブロックは必ず長方形で、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）。ブロックの長方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（377 ページ「13.3.2 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的に作成しますが、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*」**・**「*Graphics*」**（グラフィック）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 13.1 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）。

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。
- 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（macOS 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　1 個ないし複数のブロックを選択している時には、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 1 個ないし複数のブロックを、Windows の場合はマウスの右ボタンでクリックし、macOS 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、

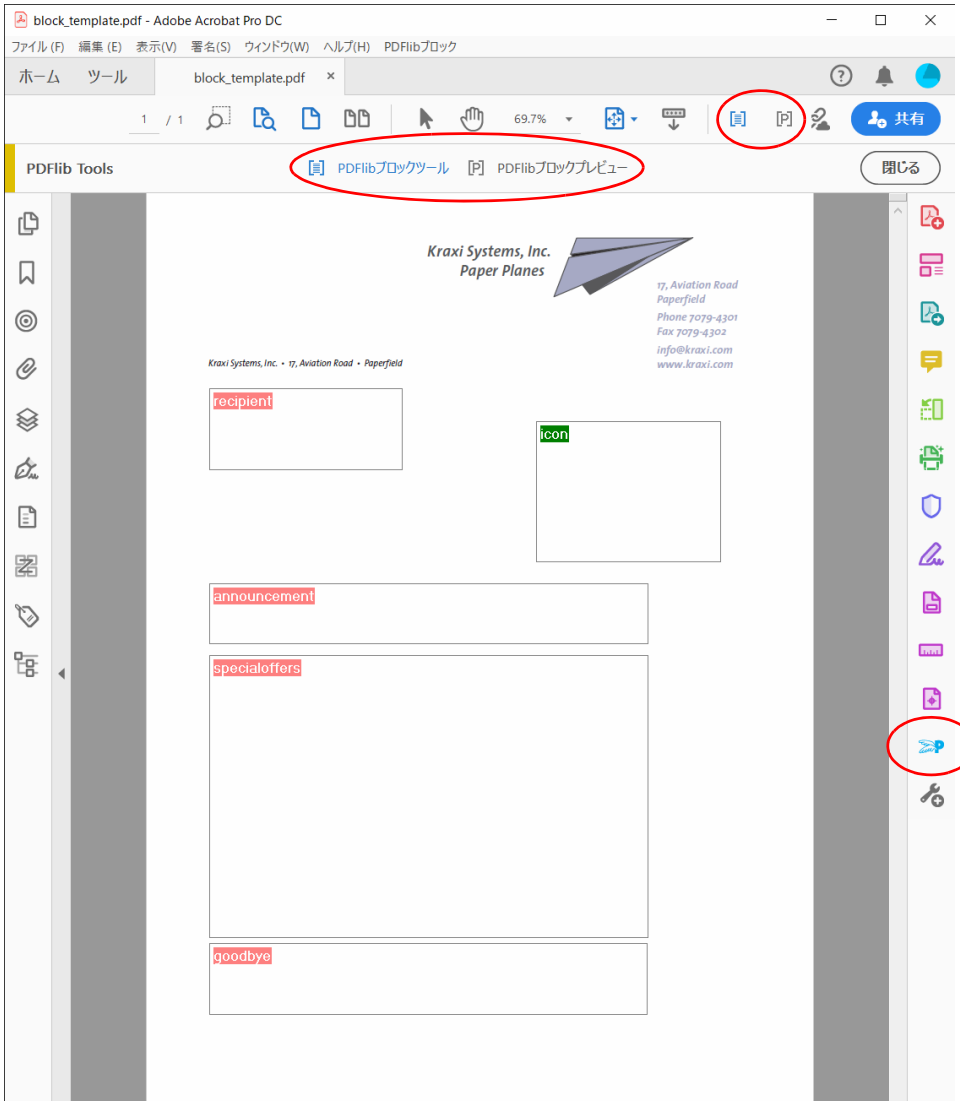


図 13.1
各種ブロックの視覚化

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（macOS では Ctrl+ クリック）すると、そのコンテキストメニューの中には、ブロックプレビューを作成するための項目と、ブロック機能を構成するための項目があります。

ブロックのサイズと位置　ブロックツールを使うと、選択した 1 個ないし複数のブロック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、横方向か縦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

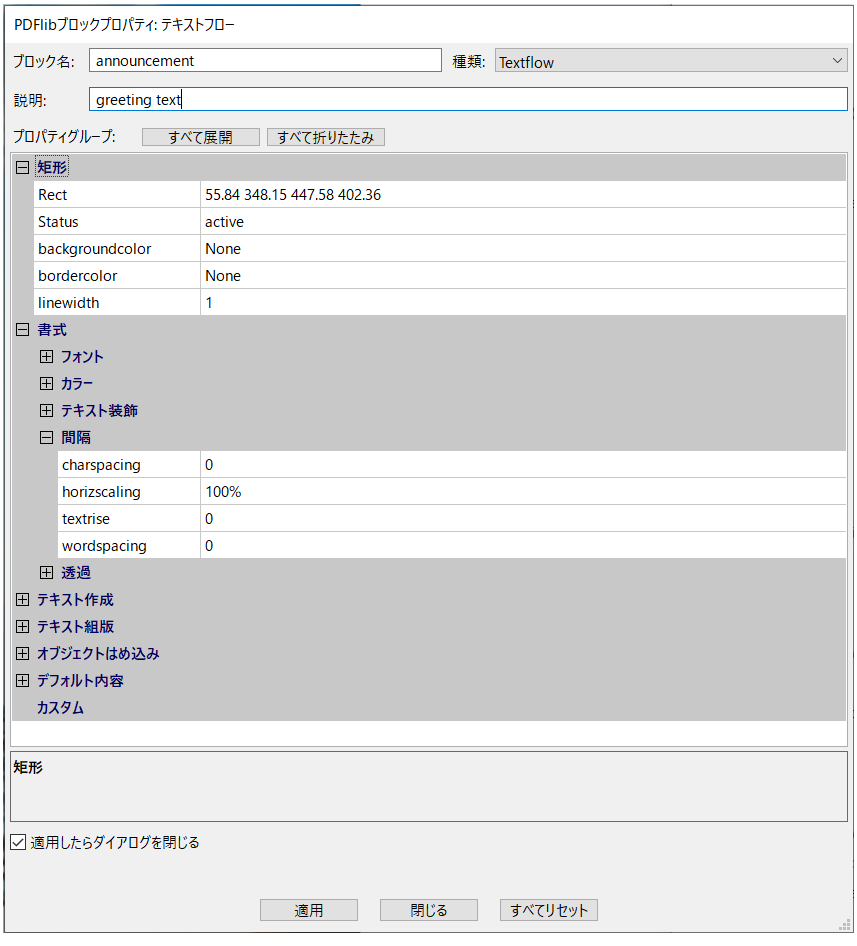


図 13.2
ブロックプロパティダイアログ

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、ブロックのサイズを変えることができます。

複数のブロックの位置やサイズ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。1 個ないし複数のブロックの位置を、矢印キーを使って小刻みに変えることもできます。

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 DC で表示単位を変更するには、以下のように操作します：**「編集」→「環境設定」（→「一般 ...」）→「単位とガイド」→「ページと定規の単位」**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ぶ。
- 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」**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付け　Acrobat のグリッド機能を活用して、ブロックの位置付けやサイズ変更を正確に行うこともできます：

- 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」**。
- 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」**。
- グリッドを変更：**「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、ブロックのサイズと位置は、構成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サイズ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することによってブロックを作成　手動でブロックの長方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サイズ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サイズ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サイズを変更したければ、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、ただそれを位置やサイズ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（対象を囲む長方形）が選択されます。その次の段階として、この検知された長方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長方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を自動検出　Block Plugin は、テキスト行またはテキストフローブロックの位置付けられた場所の背景にあるフォントを分析することができ、ブロックの照応する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サイズ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、移動させることもサイズ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。

13.3.2 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 13.2 参照）。395 ページ「13.7 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。プロパティの変更をブロックに適用した後には、ブロックの名前の中にアスタリスクが表示されて、ブロックが変更されたけれどもまだディスクには保存されていないということを示します。**「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「hortabmethod=ruler におけるルーラタブ」プロパティサブグループは、「テキスト組版」→「ルーラタブ」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。

注 ブロックプロパティにテキストを入力すると、ストレート引用符がスマート引用符に置換されるなど、キャラクタの置換が起こることがあります。この置き換えはオペレーティングシステムが行なっていることですので、無効にするには「システム環境設定」→「キーボード」→「ユーザ辞書」→「スマート引用符とスマートダッシュを使用」を選択します。

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、いつ別の値でオーバライド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。以下のアイテムは変更されません：

- *Name*・*Type*・*Rect*・*Description* プロパティ
- すべてのカスタムプロパティ

注 定義済み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・defaultgraphics **プロパティと混同しないようにする必要があります（385 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です。他の 1 個ないし複数のブロックを 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。この時、選択されているすべてのブロックに対して、プロパティ群を以下のように適用することができます：

- チェックボックス「**マスターブロックのすべてのプロパティを適用**」がチェックされていない場合：「**適用**」をクリックすると、ダイアログ内で手変更を加えられたプロパティ群（黒でハイライトされている）のみが、選択されているすべてのブロックへコピーされます。
- チェックボックス「**マスターブロックのすべてのプロパティを適用**」がチェックされている場合：**「適用」**を押すと、マスターブロックのすべてのカレントプロパティと、ダイアログ内で手変更を加えられたすべてのプロパティが、選択されているすべてのブロックへコピーされます。これを利用して、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コピーすることも可能です。

以下の定義済みプロパティ、およびカスタム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

13.3.3 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1個ないし複数のページへ複製
- 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 1 個ないし複数のページを置換することができます。そのためには、**「ツール」→「ページを整理」→「置換」**を用います。

ブロックを移動・コピー　ブロックの位置を変えるには、1 個ないし複数のブロックを選択して、新しい位置へドラッグします。ブロックのコピーを作成するには、Ctrl キー（Windows の場合）か Alt キー（macOS 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。コピー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コピー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、コピーしたいブロック群を選択します。
- Ctrl+C（Windows の場合）か Cmd+C（macOS 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。
- コピー先ページへ移動します（必要なら）。
- ブロックツールがアクティブであることを確認して、Ctrl+V（Windows の場合）か Cmd+V（macOS 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と文書へ貼り付けます。

ブロックを他のページ群へ複製　1 個ないし複数のブロックの複製を、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。

ブロックを書き出す・取り込む　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コピー　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。

13.3.4 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いくつかの設定は、各 Acrobat セッションごとに保管 / 再読み込みされており、XML 構成ファイルを通じて制御することが可能です。サンプル構成ファイル ***factory settings.xml*** をディストリビューションに同梱しています。構成が変更されたとき、新しい設定は ***user settings.xml*** に格納されます。変更された構成は、Acrobat が起動されるたびに読み込まれ、Acrobat が閉じられるたびに書き込まれます。この構成ファイルは、以下のような場所に格納されています（システムディレクトリの名前はローカライズされている可能性があります。**「*DC*」**は必要に応じて他の Acrobat トラック名へ置き換えてください）：

```
Windows : C:\<user>\AppData\Local\Adobe\Acrobat\DC\PDFlib\Block Plugin 5
macOS : /Users\<user>/Library/Application Support/Adobe/Acrobat/DC/PDFlib/Block Plugin 5
```

以下の XML エレメントを用いて、構成を手変更することが可能です：

- エレメント ***/Block_Plugin/MainDialog/CloseOnApply*** : 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。
- エレメント ***/Block_Plugin/MainDialog/ApplyAllProps*** : ブロックプロパティダイアログの**「マスターブロックのすべてのプロパティを適用」**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、マスターブロックのすべてのプロパティが選択された複数のブロックへコピーされるか、それともダイアログ内の変更が加わったプロパティ群のみかを指定します。
- エレメント ***/Block_Plugin/FontDialog/ShowBaseFonts*** : 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。
- エレメント ***/Block_Plugin/Command/ControlByClick*** : 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。
- エレメント ***/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。
- エレメント ***/Block_Plugin/Command/KeyAccelerator*** : ***control***（Windows では Ctrl キーを、macOS では Command キーを示します）・***shift***（Shift キーを示します）・***control+shift*** ・

none のいずれかの値をとり、以下のキーボードショートカットに対するアクセラレータキーを指定します：

A（すべてを選択），C（コピー），I（ブロックプロパティダイアログ），V（貼り付け），X（切り取り）

変更は、Acrobat を次に起動した際に有効になります。なぜならキーボードショートカットは実行時には変更できないからです。このエントリがない場合は、アクセラレータは一切利用できません。デフォルトは *control* です。

- エレメント *Configuration/Preferences/PreviewStatusMessage* は、各プレビュー操作後にステータスメッセージダイアログ（「10 個のブロックが処理されました：…」など）が表示されるかどうかを制御します。

13.4 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**「ツール」→「リッチメディア」**から**「オブジェクトを選択」**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 13.3 に従って、その照応する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（382 ページ「フォームフィールドを照応するブロックに関連付け」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チェックする必要があります。

フォームフィールドを照応するブロックに関連付け　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、特定のフォームフィールドに照応するブロックを正しく同定することは困難です。このことは特に、FDF または XFDF ファイルをソースとして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、その照応するブロックを作成する際に、カスタムプロパティ群として記録します。表 13.2 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。

ブロックを照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を、その照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、その照応する 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

表 13.2 ブロックに照応する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

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。

表 13.3 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
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を作成 チェックなしならテキスト行ブロックを作成
ラジオボタン・チェックボックス	

表 13.3 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
「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・コンポボックス	
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

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、その照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使用できません。

13.5 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 PDFlib 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、そのデータソース（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 390 ページ「13.6 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、ブロックの位置やサイズを改善したり、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。

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ソース（データベース等）からブロックのテキスト・画像・ベクトルグラフィック・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が、すなわち *defaulttext*・*defaultimage*・*defaultpdf*・*defaultgraphics* プロパティで指定されている値が用いられます。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集合が、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*・*defaultgraphics* プロパティ（ブロックの種類による）に書き込むか、あるいは**「高度な *PPS* オプション ...」**ダイアログ内の同じ名前のオプションに対して適切な値を与える必要があります。

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 389 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作成するには、以下のいずれかの方法を用います：

- PDFlib ブロックプレビューアイコン [P] をクリック。これは Acrobat DC で以下の場所にあります：**「ツール」→「*PDF* を編集」**をクリック。
- 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。
- 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。

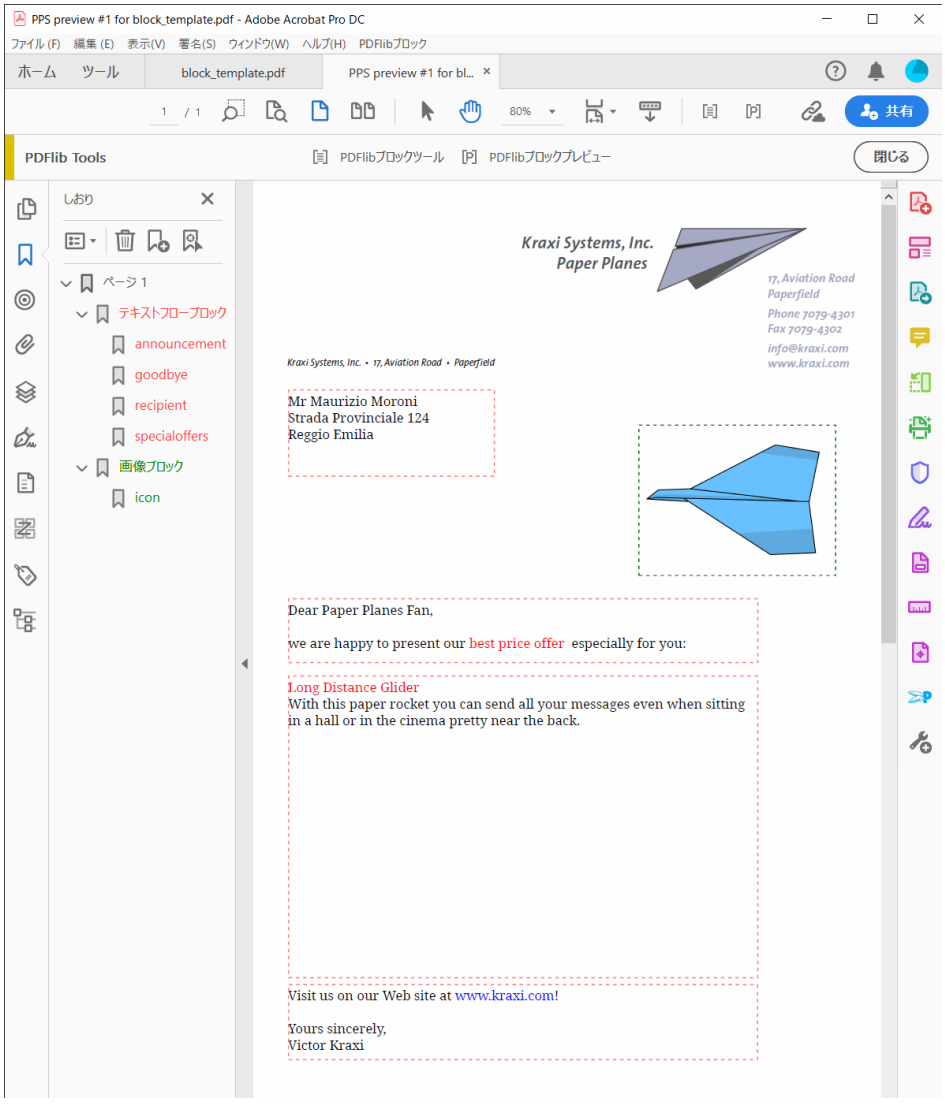


図 13.3 図 13.1 で示したコンテナ文書に対するプレビュー PDF。ブロック情報レイヤー群と注釈群を含んでいます

プレビューは、ディスク上の PDF ファイルに基づいて作成されます。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。プレビュー機能を構成して、プレビュー作成前にブロック 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、ただちにプレビュー内で反映されます。

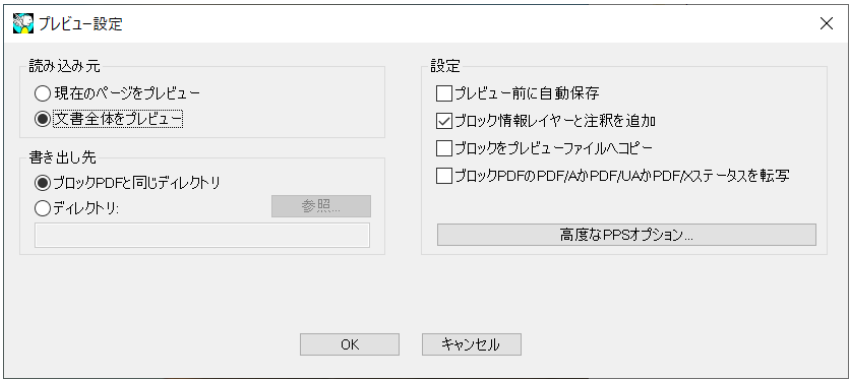


図 13.4 ブロックプレビュー構成

プレビューを構成 ブロックプレビューの作成と、その基礎をなす 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 ...」**で構成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 をプレビュー作成前に自動保存。
- ブロック情報レイヤーと注釈を追加。
- 生成される出力へブロック群をコピー。
- **「ブロック *PDF* の *PDF/A* か *PDF/UA* か *PDF/X* ステータスを転写」**: 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。
- **「ブロックをプレビューファイルへコピー」**を利用すると、流し込みの際に、PDF ブロック群を、生成されたプレビューへコピーすることができます。ブロックへの流し込みが成功したかどうかにかかわらず、すべてのブロックがコピーされます。
- **「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 コードの開発者との連携のもとに行うことを推奨します。

ブロックの順序 文書が Acrobat の「別名で保存 ...」で保存される際には、そのブロック群は、そのブロック名に従ってアルファベット順に並べ替えされます。ブロック群がプレビューによって処理される順序も、pCOS によって報告される順序も、これと同じです。とはいえ、各種アプリケーションでは通常、ブロックへの流し込みを、その名称に基づいて行いますので（ファイル内での格納順に従ってではなく）、PDF 文書内での順序が問題となることは通常ありません。

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 構成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、以下のアイテムが作成されます：

- **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長方形として視覚化されていますので、容易に識別できます。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作成されます。
- **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」→「表示切り替え」→「ナビゲーションパネル」→「しおり」**（Acrobat X/XI/DC）で表示できます。しおりはつねに作成されます。
- **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、ページ上に注釈が作成されます。この注釈の長方形は、元のブロックの枠を視覚化しています（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、プレビュー構成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* か *PDF/UA* か *PDF/X* ステータスを転写」**オプションを有効にすると、注釈は作成されません。
- **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作成されます。空のままになるレイヤーについては作成されません（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。レイヤーの作成は、プレビュー構成で無効化することも可能です。PDF/A-1・PDF/X-3 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* か *PDF/UA* か *PDF/X* ステータスを転写」**オプションを有効にすると、レイヤーは作成されません。

PDF/A か PDF/UA か PDF/X ステータスを転写　**「ブロック *PDF* の *PDF/A* か *PDF/UA* か *PDF/X* ステータスを転写」**構成は、これらの規格に従った PDF 出力を作成する必要があるときに有用です。転写モードは、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-1a:2005, PDF/A-1b:2005
PDF/A-2a, PDF/A-2b, PDF/A-2u
PDF/A-3a, PDF/A-3b, PDF/A-3u

PDF/UA-1

PDF/X-3:2003
PDF/X-4・PDF/X-4p
PDF/X-5n
```

プレビューが転写モードで作成されるときは、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。すべてのページ枠を含みます。
- タグ付き PDF：文書言語（もしあれば）。
- XMP 文書メタデータ。

規格準拠の PDF 文書を転写する際には、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、329 ページ「12.3 PDF/A によるアーカイビング」と、341 ページ「12.4 PDF/X による印刷出力」に示しています。PDF/A または PDF/X 転写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、その照応する 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、以下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（121 ページ「5.6.2 文字参照」参照）。たとえば、以下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。

13.6 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。以下のコード断片は、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みます：

```
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オーバライド　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オーバライド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：

- 業務上の要請で特定のオーバライド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。

プロパティをオーバライドするには、*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、その照応する値を与えます。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 でオーバライドします。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。

プロパティのオーバライドは、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。

流し込みながらテキストフローブロックを移動　固定サイズのテキストフローブロックは、そのいろいろなテキスト内容に合わない時があります。テキストが少ないと、2 つのブロックの間にアキが発生するかもしれませんし、テキストが多すぎると、そのブロックの長方形に収まらないかもしれません。こういう場合には、テキストフローへのはめ込みの結果をクエリすることによって、次のブロックの位置を調整することもできます：

- デフォルトの *fitmethod* は *auto* ですので、テキストは、ブロックの長方形にむりやりはめ込まれます。テキストが多い時にブロックからあふれてもよいことにするには、*fitmethod* を *nofit* に設定する必要があります。これを指定するには、デザイン時にブロックテンプレート内でブロックプロパティで指定するか、*PDF_fill_textblock()* に *fitmethod* オプションを与えます。
- *PDF_fill_textblock()* にダミーオプション *textflowhandle=-1*（PHP では *textflowhandle=0*）を与えます。するとこのメソッドは、ブロックの内容へのテキストフローハンドルを返します。
- この返されたテキストフローを *PDF_info_textflow()* に与えることによって、キーワード *textendy* を用いて、テキストの末尾の位置をクエリします。
- ブロックの下辺の垂直位置を、*PDF_pcos_get_number()* と pCOS パス *pages[..]/blocks/<ブロック名>/Rect[1]* を用いてクエリします。
- 2 つの値の差を計算します。このオフセットが正なら、テキストフローはブロックいっぱいになってはいませんし、負なら、テキストはブロックからあふれています。いずれの場合も、次のブロックを、このオフセットだけ上か下へ動かすとよいでしょう。これを実現するには、*PDF_fill_textblock()* の *refpoint* オプションで *Rect* プロパティをオーバライドします。このオプションは絶対座標をとりますので、ブロックの垂直位置をクエリしたうえで（前のステップを参照）、この元の位置にオフセットを加えた値をこの *refpoint* オプションに与える必要があります。
- この方式は、テキストフローブロックが何個あっても、ブロックごとのオフセットを足し合わせていけば、適用していくことができます。各ブロックの内容に応じて、その次のブロックを、上か下へ、適切な量だけ動かしていきます。

クックブック **完全なコードサンプルが** starter_block **サンプルにあります。**

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貼り付けたうえで、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へ（または同一出力ページ上へも）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作成します。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、その照応する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 の場合：0）を与える必要があります。内部的に作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断片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;
        int reason;
        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_string(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。

ネストされたブロック 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コピー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ネスト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・グラフィックのうちのいずれかのブロッ

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ネストすることができます。ブロックのネストは何重でも可能ですが、三重以上のネスト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（レターのテンプレートページなど）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、まずレターテンプレート上のブロック群への流し込みを行なってから、そのレターテンプレート自体を次の第一層ブロック内に貼り付ける必要があります。なぜなら、PPS はつねに、テンプレートページがもっとも最近に配置された位置を用いるからです。

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　ブロックの長方形の座標は、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位置付け・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 (374 ページ「ブロックのサイズと位置」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。これらのカラー名は PPS に内蔵されており、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()*** か然るべきカラーオプションリスト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。

13.7 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*」・「グラフィック」**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_textline()*・*PDF_fit_image()*** といった API 関数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。

13.7.1 管理プロパティ

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。必須エントリ群は Block Plugin によって自動生成されます。表 13.4 に、管理プロパティの一覧を示します。

表 13.4 管理プロパティ

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・Graphics 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へマップされます（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 13.9 参照）。
Type	（キーワード、必須）つねに Block

13.7.2 長方形プロパティ

長方形プロパティ群は、すべての種類のブロックに適用されます。これらは、ブロックの長方形本体の書式を記述します。必須エントリ群は Block Plugin によって自動生成されます。表 13.5 に、長方形プロパティの一覧を示します。

表 13.5 長方形プロパティ

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、長方形が描かれ、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、長方形が描かれ、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）ブロックの長方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	（長方形、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/graphics プロパティ・オプションが無視される、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・グラフィック内容があれば、それが用いられます。

13.7.3 書式プロパティ

書式プロパティ群は、組版の詳細を指定します：

- 表 13.6 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 13.7 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 13.6 すべてのブロック種別に対する透過書式プロパティ

キーワード	とりうる値・解説
blendmode	（キーワードリスト。PDF/A-1 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/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/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 13.7 テキスト行・テキストフローブロックに対するテキスト書式プロパティ

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、macOS・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
horizscaling	（float またはパーセント値）テキストの横伸縮。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
overline	（論理値）上線のモード。デフォルト：false
shadow	（複合）影付き効果を生み出します（デフォルト：影なし）。以下のサブプロパティが使用可能です： ***fillcolor*** （色）影の色。デフォルト：{gray 0.8} ***offset*** （float 2 個かパーセント値 2 個のリスト）テキストの参照点からの影のオフセットを、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表したもの。デフォルト：{5% -5%}
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、またはそれと等価な値 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

表 13.7 テキスト行・テキストフローブロックに対するテキスト書式プロパティ

キーワード	とりうる値・解説
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
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。

13.7.4 テキスト作成プロパティ

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 13.8 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 13.8 テキスト行・テキストフローブロックに対するテキスト作成プロパティ

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 オプション
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 オプション
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、167 ページ「7.3.1 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 176 ページ「7.4.2 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 176 ページ「7.4.2 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

13.7.5 テキスト組版プロパティ

表 13.9 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* 引数に照応しています）。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 13.9 テキスト組版プロパティ（主にテキストフローブロック用）

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します：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 13.9 テキスト組版プロパティ（主にテキストフローブロック用）

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 1 個ないし複数の構成要素から成り、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、非負エントリを昇順で最大 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 13.9 テキスト組版プロパティ（主にテキストフローブロック用）

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、ブロック長方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。このリスト内の各エントリはそれぞれ、ruler オプション内で、その照応するエントリ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「テキスト組版」グループの「hortabmethod=ruler におけるルーラタブ」サブグループで編集することができます。

13.7.6 オブジェクトはめ込みプロパティ

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制御します：

- 表 13.10 に、テキスト行・画像・PDF・グラフィック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 13.11 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、ブロック長方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長方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 13.10 テキスト行・画像・PDF・グラフィックブロックに対するはめ込みプロパティ

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、ブロック長方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 で強制的に位置付けられるので、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長方形に収まりきらないときの解決方式：auto・clip・entire・meet・nofit・slice。（デフォルト：meet）。
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
minfontsize	（float またはパーセント値。テキスト行にのみ適用）fitmethod=auto で shrinklimit を超えたとき、テキストがブロック長方形に収まるよう縮小される際に、許される最小の文字サイズ。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。下限に達したときは、テキストは指定された minfontsize を文字サイズとして作成されます。デフォルト：0.1%
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 13.10 テキスト行・画像・PDF・グラフィックブロックに対するはめ込みプロパティ

キーワード	とりうる値・解説
scale	(float のリスト。画像・PDF・グラフィックブロックのみ) 縦方向・横方向の、求める伸縮倍率を表す 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 13.11 テキストフローブロックに対するはめ込みプロパティ

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) ブロック長方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。Ḷのような、読み分け記号付きの文字は普通、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) ブロックがテキストフローに対して小さすぎるときの解決方式： ***auto*** テキストが収まるまで、fontsize と leading を縮めます。 ***clip*** テキストをブロックの端で切り落とします (テキストフローブロックを連結する場合に有用です)。 ***nofit*** テキストをブロック下端からはみ出させます (ブロックを移動させる場合に有用です)。 デフォルト：textflowhandle オプションが与えられているなら clip、そうでないなら auto
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 13.11 テキストフローブロックに対するはめ込みプロパティ

キーワード	とりうる値・解説
minfontsize	(float またはパーセント値) 特に fitmethod=auto のとき、はめ込み枠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、またははめ込み枠の高さに対するパーセント値で指定します。下限に達してもなおテキストが収まりきらないときは、文字列 _boxfull が返されます。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。

13.7.7 デフォルト内容のためのプロパティ

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 13.12 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 13.12 デフォルト内容のためのプロパティ

キーワード	とりうる値・解説
default-graphics	（文字列。グラフィックブロックにのみ適用）グラフィック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グラフィックファイルのパス名。[1]
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。[1]
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。[1]
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[2]

1. 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。
2. テキストは winansi エンコーディングか Unicode で解釈されます。

13.7.8 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 13.13 に、カスタムプロパティの命名規則を示します。

表 13.13 すべてのブロック種別に対するカスタムブロックプロパティ

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。

13.8 pCOS でブロック名とプロパティをクエリ

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クエリ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クエリ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を知る 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。なぜならクエリすることもできるからです。以下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");
```

以下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クエリしたり、ブロックへテキスト・画像・PDF・グラフィック内容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）。

ブロックが存在するかどうかをチェック クライアントアプリケーションコードにさらに柔軟性を加えるために、ブロックに流し込みを行う前に、そのブロックが存在するかどうかをチェックすることもできます。こうしておけば、デザイナーが別のページへブロックを移動させても、そのブロックへ流し込みを行うアプリケーションを破壊せずにすみます。

以下のコードは、*foo* という名前のブロックがページ上に存在するかどうかをチェックします：

```
/* pCOSオブジェクト種別「辞書」はそのブロックが存在することを意味します */
if (pcos_get_string(doc, "type:pages[" + pagenum + "]/blocks/" + "foo").equals("dict"))
{
        /* ブロック「foo」はそのページ上に存在 */
}
```

ブロックを番号か名前で特定

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、番号 6 のブロックが、その *Name* プロパティを *foo* に設定されているとします：

```
pages[...]/blocks[6]
pages[...]/blocks/foo
```

ブロックの座標をクエリ 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、以下のようにクエリできます：

```
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）。一方 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。

pCOS 擬似オブジェクト ***rect***（すべて小文字）を用いてクエリされる値は、関連するいかなる CropBox/MediaBox・Rotate エントリをも考慮に入れ、かつ座標の順序を正規化します。これに対し、ネイティブ PDF キー ***Rect*** を用いてクエリされる値は、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。

topdown オプションはブロック座標をクエリする際には考慮されないことに留意してください。

カスタムプロパティをクエリ　カスタムプロパティは、以下の例のようにクエリ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*** をクエリしています：

```
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、以下のようにします：

```
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、定義済みプロパティの場合、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　さまざまなソースからの 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（395 ページ「13.7 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。

13.9 ブロックをプログラム的に作成・取り込む

13.9.1 POCA で PDFlib ブロックを作成

PDFlib ブロックは、PPS に内蔵されている POCA インタフェースでプログラム的に作成することも可能です。POCA を用いると、ブロックのために必要な PDF データ構造を作成したうえで、*PDF_begin/end_page_ext()* の *blocks* オプションに与えることができます。ブロック定義を作成する際には、412 ページ「13.10 PDFlib ブロックの仕様」の要請に従う必要があります。ブロックプロパティは、395 ページ「13.7 ブロックのプロパティ」に挙げたデータ型に従って作成する必要があります。

クックブック **PDFlib ブロックを PPS で作成するためのコードサンプルが PDFlib クックブックの blocks カテゴリにあります。**

PDFlib ブロックの仕様には、1 つのブロックの名前が 2 回記録されているという残念な冗長性があります：ページのメイン *Blocks* 辞書内に 1 回と、特定のブロック辞書内の *Name* エントリ内にもう 1 回です。この 2 個の名前は、そのブロックに PPS で流し込みを行う際や、そのブロックを Block Plugin でプレビューする際に問題が起こることを避けるために、同一でなければなりません。*PDF_begin/end_page_ext()* はそのため、*blocks* オプションで与えられた辞書がこの「同一ブロック名」規則に違反するブロック定義を含んでいる場合には、例外を発生させます。以下のコードサンプルでは、その照応するペアを青色で示しています。

以下のコード断片では、412 ページ「ブロック辞書のキー」で示すブロック定義を POCA 関数群を用いて作成する様子を演示しています：

```
/* ブロック辞書を作成 */
blockdict = p.poca_new("containertype=dict usage=blocks");

/* ------------------------------------------------------
 * テキストブロックを作成
 * ------------------------------------------------------
 */
textblock = p.poca_new("containertype=dict usage=blocks type=name key=Type value=Block");

container1 = p.poca_new("containertype=array usage=blocks " +
        "type=integer values={70 640 300 700}");

p.poca_insert(textblock, "type=array key=Rect value=" + container1);
p.poca_insert(textblock, "type=name  key=Name value=job_title");
p.poca_insert(textblock, "type=name  key=Subtype value=Text");
p.poca_insert(textblock, "type=name  key=fitmethod value=auto");
p.poca_insert(textblock, "type=name  key=fontname value=Helvetica");
p.poca_insert(textblock, "type=float key=fontsize value=12");

/* このブロックをページのブロック辞書内に挿入 */
p.poca_insert(blockdict, "type=dict key=job_title direct=false value=" + textblock);

/* ------------------------------------------------------
 * 画像ブロックを作成
 * ------------------------------------------------------
 */
imageblock = p.poca_new("containertype=dict usage=blocks " +
        "type=name key=Type value=Block");
```

```
container2 = p.poca_new("containertype=array usage=blocks " +
        "type=integer values={70 440 300 600}");

p.poca_insert(imageblock, "type=array key=Rect value=" + container2);
p.poca_insert(imageblock, "type=name  key=Name value=logo");
p.poca_insert(imageblock, "type=name  key=Subtype value=Image");
p.poca_insert(imageblock, "type=name  key=fitmethod value=auto");

/* このブロックをページのブロック辞書内に挿入 */
p.poca_insert(blockdict, "type=dict key=logo direct=false value=" + imageblock);

/* ------------------------------------------------------
 * PDFブロックを作成
 * ------------------------------------------------------
 */
pdfblock = p.poca_new("containertype=dict usage=blocks " +
        "type=name key=Type value=Block");

container3 = p.poca_new("containertype=array usage=blocks " +
        "type=integer values={70 240 300 400}");

p.poca_insert(pdfblock, "type=array key=Rect value=" + container3);
p.poca_insert(pdfblock, "type=name key=Name value=pdflogo");
p.poca_insert(pdfblock, "type=name key=Subtype value=PDF");
p.poca_insert(pdfblock, "type=name key=fitmethod value=meet");

/* このブロックをページのブロック辞書内に挿入 */
p.poca_insert(blockdict, "type=dict key=pdflogo direct=false " + "value=" + pdfblock);


/* ------------------------------------------------------
 * このブロック辞書をカレントページ内に挿入
 * ------------------------------------------------------
 */
p.end_page_ext("blocks=" + blockdict);


/* クリーンナップ */
p.poca_delete(blockdict, "recursive");
```

13.9.2 PDFlib ブロックを取り込む

入力文書から1個ないし複数のPDFlibブロックを、*PDF_process_pdi()*と*action=copyallblocks*か*action=copyblock*で以下のようにカレント出力ページへコピーすることも可能です：

```
if (p.process_pdi(p, doc, 0, "action=copyallblocks block={pagenumber=1}") != 1)
{
        /* エラー */
}
```

このようにすると、マルチレベルのブロック流し込みワークフローを実装することができます。ただし、各ページ上でブロック名は一意でなければならないことに留意してください。すなわち、同じ名前を持つ複数のブロックを同じページ上へ取り込むことはできません。ブロックをコピー時に名称変更するには*outputblockname*サブオプションを用います。

13.10 PDFlib ブロックの仕様

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 に準拠しています。ここでは PDFlib ブロックの文法を説明します。Block Plugin か PDFlib でブロックを作成するユーザーにはこの情報は必要ではありません。

PDFlib ブロックの PDF オブジェクト構造　ページ辞書は *PieceInfo* エントリを含んでおり、このエントリは値として別の辞書を持っています。ページ辞書は、ブロック構造の作成または最終更新のタイムスタンプを内容とするキー*LastModified* をも含む必要があります。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 13.14 に挙げる 2 個の標準キーを含んでいます。

表 13.14 PDFlib アプリケーションデータ辞書内のエントリ

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。このエントリは、POCA インタフェースでブロックを作成した場合には PDFlib が作成します。
Private	（辞書、必須）ブロックリスト（表 13.15 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 13.15 に、ブロックリスト辞書の中のキーの一覧を示します。

表 13.15 ブロックリスト辞書内のエントリ

キー	値
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。その照応する値は、そのブロックに対するブロック辞書です（表 13.17 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。
BlockProducer[1]	（文字列）ブロックをプログラム的に作成するのに用いられたソフトウェアの名前。このエントリは、POCA インタフェースでブロックを作成した際に PDFlib によって作成されます。
PluginVersion[1]	（文字列）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[1]	（論理値）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。
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 10 のブロック仕様を解説しています。

1. キー BlockProducer・PluginVersion・pdfmark のうちのいずれか 1 つが、かつ 1 つのみが存在する必要があります。

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 13.16 に、これらの型が PDF データ型へどのようにマップされているかを示します。

ブロック辞書のキー　ブロック辞書は、表 13.17 に挙げるキーを含むことができます。

表 13.16 ブロックプロパティに対するデータ型

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）
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 の数値では点を必要とします。
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 番目の要素はカラー値を指定。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。配列の 1 番目の要素に対しては以下のエントリを指定できます： ***/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。
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

表 13.17 ブロック辞書内のエントリ

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 13.4 に従った管理プロパティ群
長方形プロパティ群	（いくつかのキーは必須）表 13.5 に従った長方形プロパティ群
書式プロパティ群	（いくつかのキーは必須）表 13.6 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 13.7 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 13.8 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 13.9 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 13.10 に従った、テキスト行・画像・PDF・グラフィック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 13.11 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

表 13.17 ブロック辞書内のエントリ

プロパティグループ	値
デフォルト内容に関するプロパティ群	（オプション）表 13.12 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 13.13 に従った、カスタムプロパティに対するキー / 値ペアを含んだ辞書

A 改訂履歴

日付	更新点
2021 年 6 月 06 日	▸ PDFlib 9.3.1 に関する更新
2020 年 8 月 16 日	▸ PDFlib 9.3.0 に関する更新
2019 年 2 月 01 日	▸ PDFlib 9.2.0 に関する更新
2018 年 2 月 01 日	▸ PDFlib 9.1.2 に関する更新
2017 年 7 月 24 日	▸ PDFlib 9.1.1 に関する更新
2017 年 5 月 09 日	▸ PDFlib 9.1.0 に関する更新
2016 年 8 月 14 日	▸ PDFlib 9.0.7 に関する更新
2016 年 1 月 13 日	▸ PDFlib 9.0.6 に関する更新
2015 年 5 月 24 日	▸ PDFlib 9.0.5 に関する更新
2014 年 12 月 16 日	▸ PDFlib 9.0.4 に関する更新
2014 年 5 月 14 日	▸ PDFlib 9.0.3 に関する更新
2013 年 12 月 17 日	▸ PDFlib 9.0.2 に関する更新
2013 年 6 月 06 日	▸ PDFlib 9.0.1 に関する更新
2013 年 3 月 12 日	▸ PDFlib 9.0.0 に関する更新
2011 年 6 月 09 日	▸ PDFlib 8 VT Edition（内部的には 8.1.0）に関する更新
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

日付	更新点
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
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

す

せ

そ

た

ち

つ

て

と

な

に

ぬ

ね

は

ひ

PDFlib GmbH

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

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）