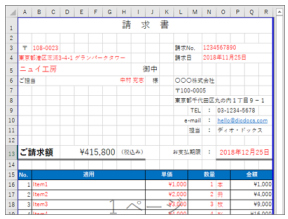
.NET向けExcel・PDF操作ライブラリ「DioDocs」の衝撃！ その魅力に迫る

開発ツール

中村 充志（リコージャパン株式会社）[著]

2018/12/17 11:00



2018年10月17日、グレープシティ社から「DioDocs（ディオドック） for Excel」および「DioDocs for PDF」がリリースされました。皆さんはすでにご覧になったでしょうか？ 事前に同社のブログやイベントで告知がありましたが、リリースされたサイトを見て私は大げさではなく衝撃を受けました。そしてこう思いました。「これは帳票生成ライブラリの勢力図を大幅に塗り替えかねないポテンシャルを秘めている」と。本稿ではその衝撃的なまでの魅力に迫りたいと思います。

- DioDocs（公式Webサイト）

筆者について

私は普段、主に金融業界向けの受託開発のアーキテクトを務めています。特に所属会社の特性上、電子と紙をまたがった領域を得意としています。

本稿はエンタープライズ領域でのアプリケーション開発者の視点で記載しています。したがって立場が変わればその見方は大きく変わるでしょう。しかし、異なる立場の方であっても、ExcelやPDFを扱う上で検討するに値する素晴らしい製品のはずです。ぜひ本稿を一読いただければと思います。

なお、本稿で扱ったコードはすべて**GitHubに公開しています**。部分的なコードで分かりにくい部分は併せてこちらもご覧ください。

DioDocsの魅力【1】

DioDocs for ExcelとDioDocs for PDFは、.NET向けのExcelファイルおよびPDFを操作するためのライブラリです。

具体的にサポートする機能については、それぞれのサイトを見てください。

- DioDocs for Excel
- DioDocs for PDF

私は先ほど、DioDocsに衝撃的な魅力を感じたと書きました。その魅力は、単に機能がどうのという話ではありません。多数の選定要因が非常にバランスよくまとまっている点にあると考えています。

これまでも.NETからExcelやPDFを操作する方法は多数存在しました。例えばExcelファイルを操作する方法を簡単に挙げてみると、次のような手段があります。

Microsoft.Office.Interop.Excel

Microsoft純正のExcelのCOMによるオートメーションライブラリ。

Visual Studio Tools for Office（VSTO）

Microsoft純正のExcelなどのOfficeアドイン作成用フレームワーク。

Open XML

Microsoftが主導して開発する、OSSのOfficeファイル操作用API。Excel以外にも対応しているが煩雑。

ClosedXML

Open XMLをラップしExcel準拠のオブジェクトモデルに寄せたExcelファイル操作ライブラリ。

NPOI

Java製のApache POIを.NETに移植したライブラリ。APIの独自性が高い。

EPPlus

xlsファイルもサポートしたライブラリ。比較的Excelのオブジェクトモデルに近い。

これらはすべて無償で利用でき、なおかつ実績のあるプロダクトです。

対してPDFは「主要なライブラリ」を挙げることも実は悩みます。例えばOSSのPDF製品を考えてみましょう。OSS系のPDFライブラリとなると、最近はiTextやPDFSharpをよく見かけます。

しかし後ほど詳細に説明しますが、PDFSharpはクライアントプロセスでの利用であれば問題ありませんがサーバープロセスでの利用は危険です。対してiTextはデュアルライセンスで、無償利用可能なライセンスはAGPLであることから利用シーンが限られますし、有償版は非常に高価です。

ではその他の商用製品はどうかと言いますと、実は多数存在します。しかし機能一覧だけ眺めていても決定的に優位な要素は見えてきません。

このようにExcelとPDFを取り囲むライブラリの環境はそれぞれ全く異なります。そういった状況の中で私が衝撃を受けた理由は、DioDocsの細かな機能の一覧にあったわけではありません。次のような複数の選定要因が、非常にバランスよく成り立っていることが大きな要因です。

- Excelファイルから PDFファイルの生成
- .NET Standard 2.0準拠
- ランタイムフリー
- Excel準拠のオブジェクトモデル
- 開発元は日本が本社の企業なので直接サポートが受けられる

これらの非常に絶妙なバランスが、衝撃的ともいえる魅力を生み出しています。そして実際に評価目的で利用して、次のもう1つの魅力に驚かされました。

- 軽快な動作速度

何がそこまで魅力的なのか？ 端的に言えば**「クラウド時代のユーザーメンテナンス可能な帳票生成サービスの構築に最適である」**ということです。なぜそうなのか、順番に説明していきましょう。

ExcelファイルからPDFファイルの生成

これまでも前述のOpen XML、ClosedXML、NPOI、EPPlusといったExcelファイル操作ライブラリを利用して、Excelをテンプレートとして帳票出力することは可能ではありました。しかしその場合、出力形式は必然的にExcelに限られました。

Excel形式のまま帳票として運用することもケースによっては可能でしょう。しかしExcelのままでは容易に編集が可能であることが、業務帳票として扱うには不適切なケースも多いはずです。

しかしDioDocs for ExcelはPDFを生成することができます。

Excelを帳票のテンプレートとして登録しておき、必要な値を設定した上でPDF化することで、容易にメンテナンス可能な帳票生成サービスを構築することができます。そこにDioDocs for PDFを併せて利用することで、デジタル署名を用いて改ざんを抑止することも可能になります。

先に挙げたExcel関連ライブラリの中でも、次の2つはExcelからPDFへ直接変換できます。

1. Microsoft.Office.Interop.Excel
2. Visual Studio Tools for Office（VSTO）

しかし、これらはいずれもサーバーサイドでの利用は不適切です。

1．はMicrosoftから明確に非推奨とされています。

- **Office のサーバーサイド オートメーションについて**

Microsoft Officeがインストールされている必要があることから、ライセンスからして不明確です（明確に可もしくは不可と書かれた文書の存在を知りません）し、単純に技術的な側面だけ見てもリスクが高すぎます。実際に非推奨であることを知らずに利用して、トラブルとなったプロジェクトが数多く存在します。特に次の点に大きな問題があります。

- COMオブジェクトのリソース開放漏れが起こりやすく、ゾンビExcelプロセスが発生しやすい
- 動作が非常に遅い
- マルチスレッド非対応
- 何らかのユーザーダイアログが起動した場合、処理がすべて停止する
- 動的にアセンブリをロードしない場合、Excelのバージョンに依存する
- 動的にアセンブリをロードすると、型安全性が失われる

また2．のVSTOはOfficeのAddInを作成するための仕組みであり、サーバーサイド オートメーションには利用できません。

その2つ以外のライブラリはサーバーサイドでの処理にも利用できますが、ExcelからPDFを生成することが可能なのはDioDocs for Excelのみです。

.NET Standard 2.0準拠

Microsoft.Office.Interop.ExcelとVSTO以外は.NET Standard 2.0に準拠しています。これは.NET Framework、.NET Core、Mono（つまりXamarinなど）といった、複数のプラットフォームの.NETランタイム上で動作が可能だということです。

特に.NET Core上で動作するということは、クラウドやDockerコンテナ上で利用しやすく、マイクロサービスやサーバレスアーキテクチャとも相性が良いというメリットがあります。

また.NET Standardに準拠しているということはSystem.Drawing.BitmapやSystem.Drawing.Graphics、つまりGDI+に依存していないことを意味します。サーバーサイドで画像操作をする場合、System.Drawingパッケージの利用は未サポートです。

サーバーサイドでの画像操作にはWindows Imaging Componentが推奨されていますが、これの標準の実装クラスはPresentationCoreというWPFのアセンブリに含まれており、実はそのアセンブリが同様にサーバーサイドでの利用がサポートされていません（C++のネイティブライブラリを自力でラップすることは可能でしょう）。

- **System.DrawingがWindowsサービスやASP.NETサービスで未サポート**

System.Drawing系のクラスをサーバーサイドで動かしても一見正しく動作します。しかし、まれに落ちます。私の経験上、月間数十万枚の画像を処理して年に1回落ちるかどうか？ 程度ですが、落ちるときには例外も発生せず、プロセスが無言で停止します。Windowsのイベントログを見ると、Bitmapクラスのコンストラクタを呼んだだけでプロセスが落ちたように見えました。

このようにSystem.Drawingパッケージは.NETで画像操作をする手軽な手段を提供してくれていますが、実のところサーバーサイドでの画像操作となると一筋縄ではいきません。特にPDFでは画像を扱う頻度が高いと考えています。

DioDocsはいずれも.NET Standard準拠で、その辺りの実装はSystem.Drawingに依存しない独自実装になっており、サーバーサイドでの動作もサポートしています。これは非常に重要なポイントだと考えます。

DioDocsの魅力【2】

ランタイムフリー

クラウド含め、リソースをオンデマンドに利用したいとなった場合、有償プロダクトの場合、ランタイムライセンスが気になることでしょう。

AWS LambdaやAzure Functionsで無限にスケールしたいといったケースは当然ながら、エンタープライズ領域の業務システムであっても業務ボリュームには波があります。年に1カ月もあるかないかの繁忙期に合わせてランタイムライセンスを購入しなくてはならないケースは避けたいところです。

DioDocsは、SaaSでの利用や開発環境への組み込みを除き、基本的には開発ライセンスのみでランタイムには課金されません。クラウド時代の有償コンポーネントとしては必須条件でしょう。

Excel準拠のオブジェクトモデル

DioDocs for ExcelはExcel VBAやMicrosoft.Office.Interop.Excel、VSTOといった公式のライブラリのオブジェクトモデルに準拠しています。このため非常に理解しやすく、いずれかの経験があればさらに利用しやすいと感じるでしょう。

実際にコードを見比べてみましょう。

次のVBAのコードでは、アクティブなシートの「C3」セルに「Hello World!」と値を設定して、ファイルを保存しています。

```
Dim workbook As workbook
Dim worksheet As worksheet

' ワークブックの取得
Set workbook = ActiveWorkbook
' ワークシートの取得
Set worksheet = workbook.ActiveSheet
' セル範囲を指定して文字列を設定
worksheet.Range("C3").Value = "Hello, World."
' Excelファイルとして保存
workbook.SaveAs ("Result.xlsm")
```

続いてDioDocsのコードです。こちらはC#の例ですが、もちろんVB.NETでも利用可能です。

```
// DioDocs for Excelのライセンスの有効化
Workbook.SetLicenseKey("YOUR_KEY");
// ワークブックの作成
var workbook = new Workbook();
// ワークシートの取得
var worksheet = workbook.ActiveSheet;
// セル範囲を指定して文字列を設定
worksheet.Range["C3"].Value = "Hello World!";
// Excelファイルとして保存
workbook.Save("Result.xlsx");
```

VBAとC#の文法の差異はありますが、ほとんど同等のコードで記述できているのが見て取れます。言語の差異を除くと、次の差しかありません。

- DioDocsは有償ツールのためライセンスの有効化が必要
- VBAはすでにExcelのWorkbookが開かれているがDioDocs（やClosedXMLなど）はWorkbookを作成するところから開始する
- VBAはファイルにマクロを含むため、ファイルフォーマットを「xlsm」にする必要がある

他も見てみましょう。

Microsoft.Office.Interop.Excelでは、次の通り記述してファイルを作ること「は」できます。

```
var application = new Application { Visible = false };
var workbook = application.Workbooks.Add();
var worksheet = workbook.ActiveSheet as Worksheet;
worksheet.Range["C3"].Value = "Hello World!";
workbook.SaveAs(Path.Combine(Environment.CurrentDirectory, "Result.xlsx"));
workbook.Close();
application.Quit();
```

しかし、このようなコードを書いているとCOMオブジェクトがリークし、ApplicationをQuitしても実際にはExcelのゾンビプロセスが次々生み出される結果になります。

実際には次のように、オブジェクトを1つずつ丁寧に開放していく必要があります。

```
Application application = null;
Workbooks workbooks = null;
Workbook workbook = null;
Worksheet worksheet = null;
Range range = null;
try
{
    application = new Application { Visible = false };

    workbooks = application.Workbooks;
    workbook = workbooks.Add();
    worksheet = (Worksheet)workbook.ActiveSheet;
    range = worksheet.Range["C3"];
    range.Value = "Hello World!";
    workbook.SaveAs(Path.Combine(Environment.CurrentDirectory, "Result.xlsx"));
    workbook.Close();
    application.Quit();
}
finally
{
    GC.Collect();
    GC.WaitForPendingFinalizers();

    if(range != null) Marshal.ReleaseComObject(range);
    if(worksheet != null) Marshal.ReleaseComObject(worksheet);
    if(workbook != null) Marshal.ReleaseComObject(workbook);
    if(workbooks != null) Marshal.ReleaseComObject(workbooks);
    if(application != null) Marshal.ReleaseComObject(application);
}
```

ちょっと見ただけで読む気を失うコードだと思いますが、よく見ていただきたいのは次の点にあります。

- worksheet.Range["C3"].Valueのような呼び出しをすると、RangeでCOMオブジェクトが利用されているが開放されないためリークする。アクセスしたプロパティはすべて個別に開放する必要がある。
- tryの途中で例外が発生したことを考慮し、Marshal.ReleaseComObjectを呼び出して開放前にnullチェックをする必要がある。しないとNullReferenceExceptionが発生する。
- リソース開放前にガベージコレクタを呼び出さないとリークすることがある。
- 保存パスは絶対パスで指定する必要がある。

しかもSaveAsのところで同名のファイルが存在した場合、上書き確認ダイアログが開かれます。しかし、サーバーサイドプロセスではそもそもユーザーインターフェース自体表示されていませんから、実際には表示もされず選択もできず、そこで処理がストップします。

また上記のコードは、try-finallyやリソースの開放といった煩雑さを含んでいます。そして実際にExcelファイルを操作するとなると、多数のセルにアクセスします。そのため大量のRangeやCellオブジェクトにアクセスするコードが必要となり、ブロックのネストは深くなる一方です。可読性は落ちるばかりで、DioDocsのように簡潔に記述し、機能の実装のみに集中することはとてもできません。

同じC#から利用できるMicrosoft公式のVSTOはリークを気にすることなく利用でき、非常に快適に実装できるのですが、実行速度は速いとは言えません。そもそも先にも記述したように、あくまでOfficeのプラグインとして動作するため、サーバーサイドでの処理には不適切です。

その他のライブラリのコードはGitHubに公開しています（以下のリンク）。良かったら参考にご覧ください。DioDocsが細部にわたって気を配って設計されていることが見て取れるかと思います。

- **Open XML**：公式SDKで安心感はあるが煩雑すぎて直接利用するのは現実的ではない
- **ClosedXML**：Open XMLをラップしているためファイル操作に安心感が得られ、かつ使いやすい

- **NPOI**：ややAPIに癖があり、個別の習熟が求められる
- **EPPlus**：おおむねExcelライクに操作できるが、DioDocsほどではない

開発元は日本が本社の企業なので直接サポートが受けられる

有償のプロダクトを利用する場合、当然サポートにも大きく期待したいところでしょう。

有償プロダクトのサポートを受ける場合、提供会社によってサポートの品質に差異がありますが、そもそもの問題として、開発元の直接サポートを受けられるかどうか？ というのも注意すべき点であると私は考えています。

海外のプロダクトを利用する場合、購入・サポートが開発会社の日本支社によるものなのか、それとも輸入代理店を経由するものなのかによって、やはりある程度の差が発生するように感じます。十分なコミュニケーションが取れれば、代理店でも必要なサポートが得られることが多いのです。しかし1次窓口から最終的に必要な情報へ到達するまでのスピードなどに差があると感じることも確かにあります。

有償プロダクトはサポートへの期待が大きい以上、プロダクトに差異がないのであれば開発元のサポートが受けられるDioDocsはその分優位にあると考えています。

軽快な動作速度

さて、ここまで読まれた方はDioDocsに興味を持たれたものと思います。私はグレープシティ社の別製品の情報を調べるために訪れた公式サイトでリリースを知り、これまでExcel操作がらみで苦慮してきた経験から、一気に興味を掻き立てられ試用させていただきました。そして2度目の驚きを得ました。

と言うのは、動作が非常に軽快だからです。実際にベンチマークを取得したので見ていただこうと思います。

まず新しく100×100の範囲のセルに文字を設定したExcelファイルを作成してみました。DioDocsの実際のコードは以下のようなものです。

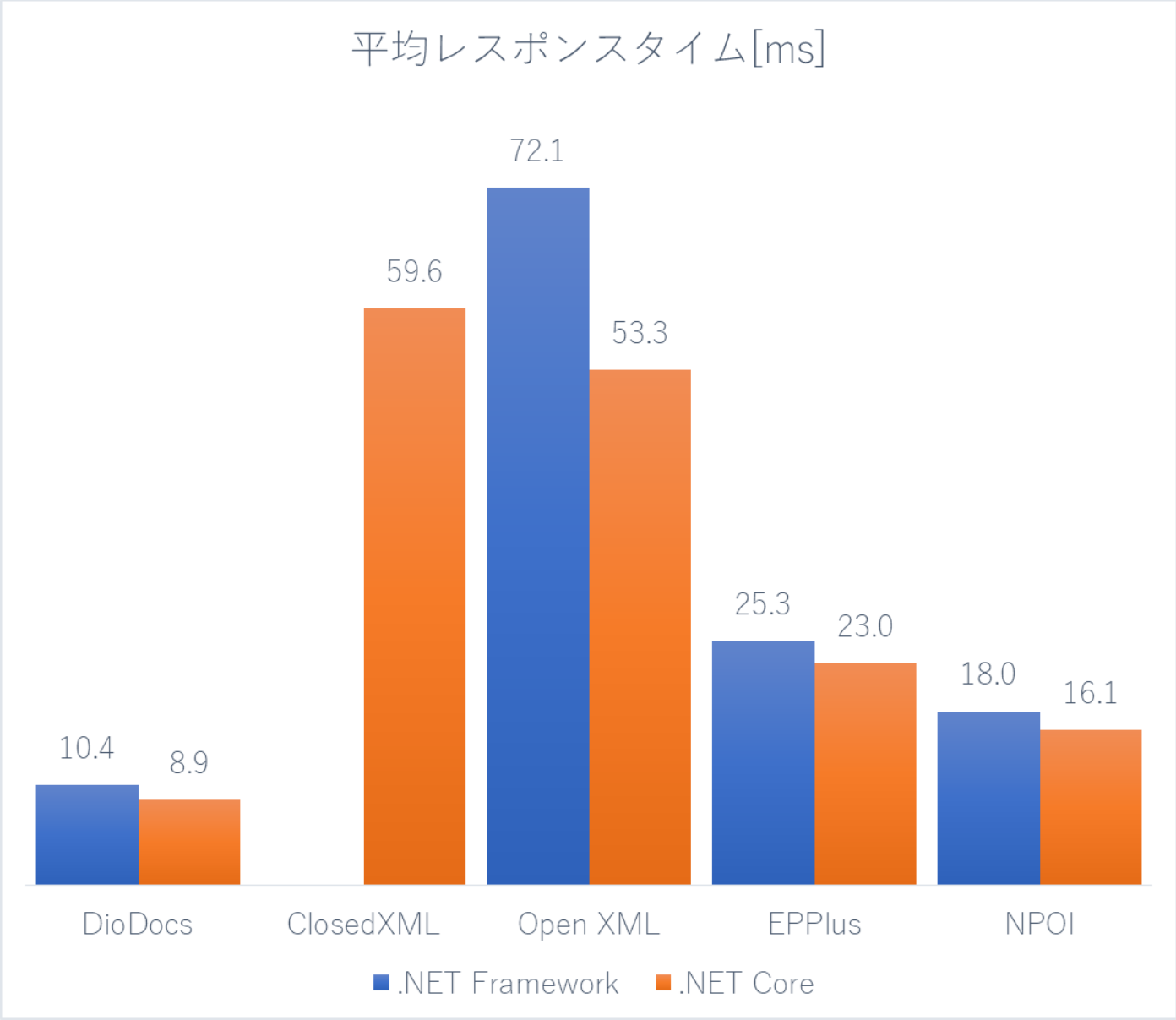
```
var workbook = new Workbook();
var worksheet = workbook.ActiveSheet;
for (var i = 1; i <= ColumnNum; i++)
{
    for (var j = 1; j <= RowNum; j++)
    {
        worksheet.Range[i, j].Value = "Hello World!";
    }
}

workbook.Save(Stream.Null);
```

すべてのベンチマークコードは**こちら**にあるので、興味のある方はご覧ください。

テスト環境は以下の通りです。

- Intel Core i7-7700T CPU 2.90GHz
- Memory 16G
- Windows 10.0.18252
- .NET Framework 4.7.2（CLR 4.0.30319.42000）、64bit RyuJIT-v4.7.3190.0
- .NET Core 2.1.5（CoreCLR 4.6.26919.02, CoreFX 4.6.26919.02）、64bit RyuJIT
- BenchmarkDotNet v0.11.3

そして実際に計測した結果が以下の通りです。平均レスポンスタイムなので、グラフが短い方が速いことを表します。



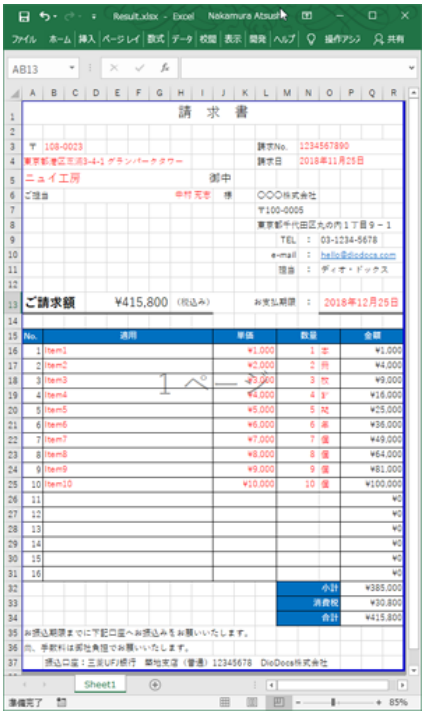
各ライブラリの平均レスポンスタイム

DioDocs速いですね！ この結果からおおむね次のように解釈できます。

- 総じてDioDocsが非常に高速で動作している。
- .NET Coreが.NET Frameworkに対し、すでに有意なレベルで高速である。
- Open XMLがそれほど速くなく、結果ClosedXMLも同様の傾向にある。

ClosedXMLがなぜか.NET Frameworkで動作していませんが、ここでは詳細は解析していません。BenchmarkDotNetとの相性問題のような気もします。また、Microsoft.Office.Interop.Excelは、Excelの起動時間を含めて91秒程度でした。ミリ秒ではなく秒です。Excelプロセスの起動が入るため、極端に遅くなります。起動したままにすることで高速化は図れますが、Microsoft.Office.Interop.Excelはサーバーサイドでは扱わず、クライアントプロセスで利用することを想定すると、Excelの起動時間を完全に無視するのは、あまり現実的とは言えないでしょう。

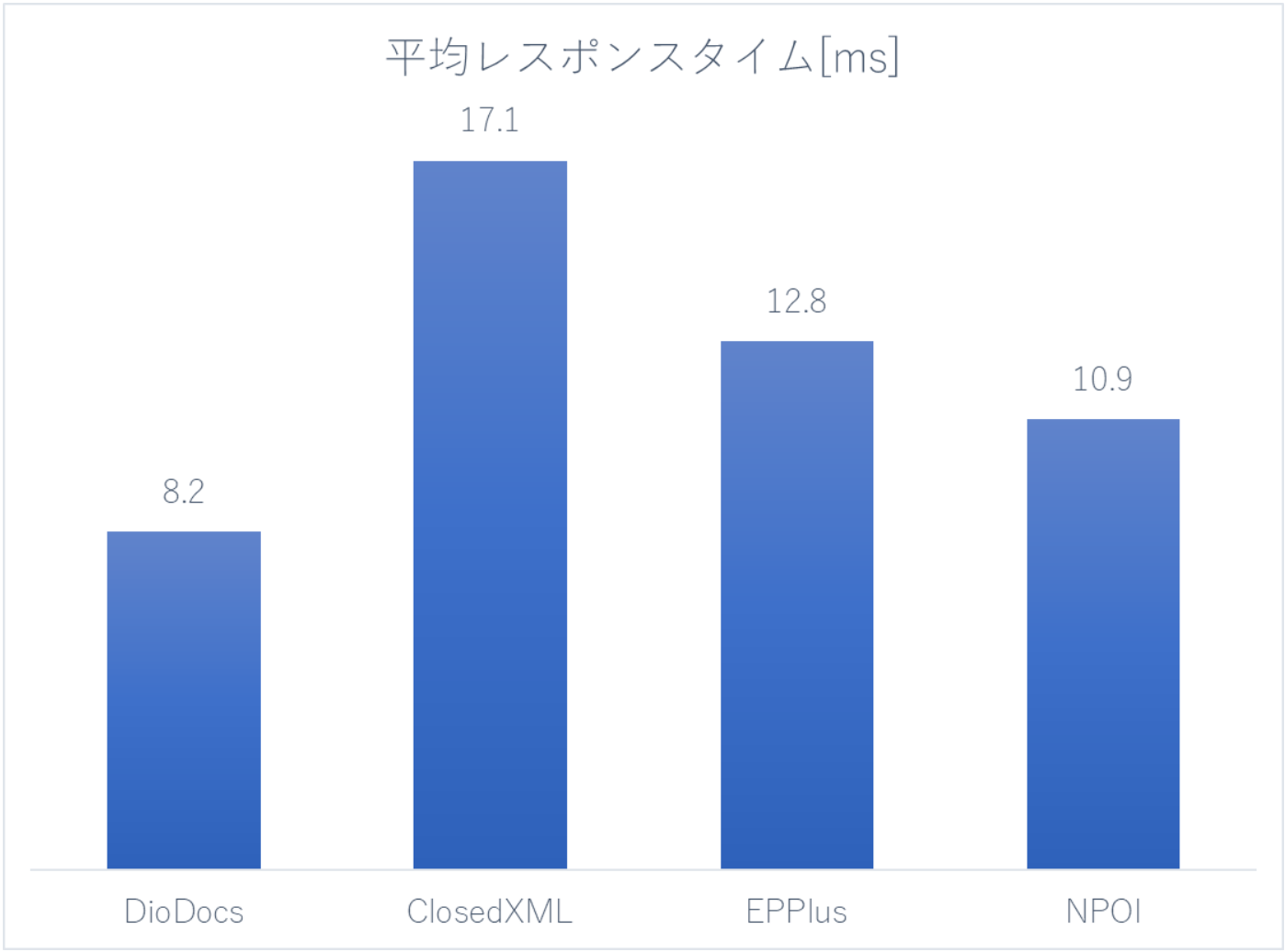
ただ、こちらのデータは実用とは少し離れている利用方法に思えます。そこでテンプレートとなるExcelファイルを事前に用意しておき、値を埋め帳票として実際に利用できるものを用意してみました。次の例をご覧ください。



帳票のサンプル

赤字の部分がExcelライブラリによって値を入れられた部分で、それ以外はテンプレートとして作成したExcelに元々含まれているものです。

この操作の実行結果が以下の通りです。なお、環境は先ほどと同様になります。



帳票のサンプルに対する各ライブラリの平均レスポンスタイム

やはりDioDocsが速い結果となりました。

今回の評価とは直接関係ない点ですが、NPOIについては、やや動作が想定通りではない点がありました。他のライブラリで作成したExcelは「金額」列がテンプレートのExcelに埋め込まれた式で計算された状態で表示されました。しかしNPOIは0円と表示されており、フォーカスをあててEnterキーを押下することで計算結果が表示されました。私の実装に問題があるのかもしれませんが、注意が必要でしょう。

さて、ここまでで私が衝撃を受けたDioDocsの魅力について理解していただけたのではないかと思います。ここではもう少し、DioDocs for ExcelとDioDocs for PDFの利用方法を、実際の利用シーンを想定して解説してみたいと思います。

DioDocs for Excel

それではもう少し具体的な領域に踏み込んで見ていきたいと思います。まずはDioDocs for Excelです。

DioDocs for Excelの魅力についてはすでに十分理解していただけているはずなので、ここでは実際に利用例を見ていただきたいと思います。実用的な複雑さを持ったレイアウトのPDF帳票が、いとも簡単に作れることに驚かれることと思います。

ここでは、テンプレートとなるExcelをもとに値を適用し、PDF帳票を生成する例を見ていただきたいと思います。

テンプレートとなるExcelをもとに値を適用し、PDF帳票を生成する

今回は請求書を題材として取り扱います。先に見ていただいた帳票です。設計・実装するにあたり、次の手順で実施します。

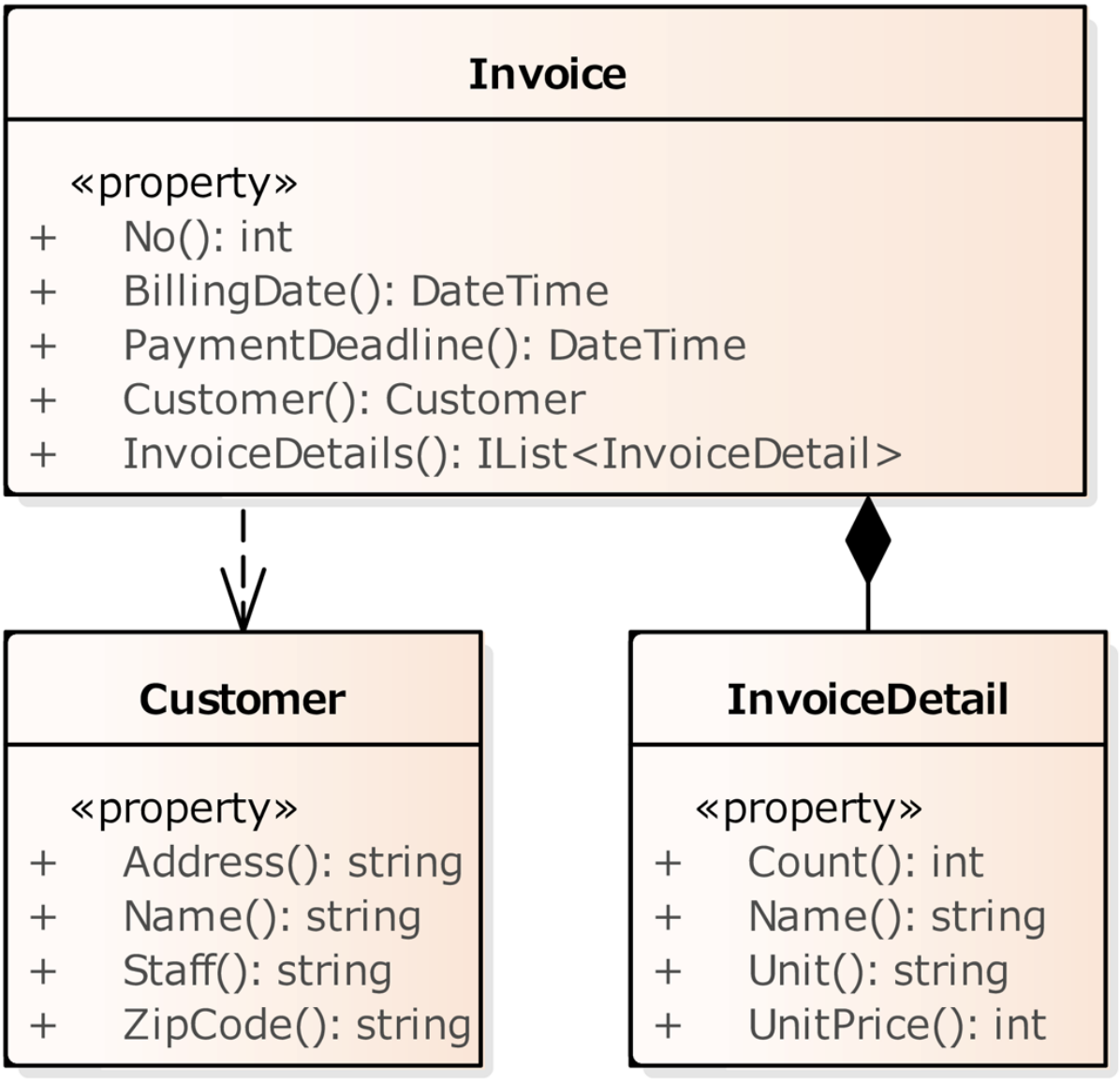
1. 帳票に出力するデータのオブジェクトを設計する
2. テンプレートとなるExcelファイルを設計する
3. テンプレートに値を設定し、PDFとして保存する

帳票に出力するデータのオブジェクトを設計する

請求書には大きく3種類の情報を表示します。

- 請求情報
- 請求先顧客
- 請求明細

具体的には以下のクラス図の通りです。



クラス図

- 請求情報（Invoice）には請求番号・請求日・支払期日が存在し、1つの請求先顧客と複数の請求明細を持ちます。
- 請求先顧客（Customer）には郵便番号・住所・名称・担当者を持ちます。
- 請求明細（InvoiceDetail）には名称・単価・数量・単位を持ちます。

これらの情報を請求書に出力します。

テンプレートとなるExcelファイルを設計する

請求書を出力する、元となるテンプレートファイルをExcelで作成します。設計した帳票のテンプレートは以下の通りです。

テンプレートファイル

このテンプレートにはいくつかの工夫が（という程のものではありませんが）なされています。

1. 請求日・支払期限に日付フォーマットを設定している
2. ご請求額・単価・金額・小計・消費税・合計のセルの書式設定に通貨を設定している
3. 金額列は、単価が入力されていた場合に単価×数量を求める計算式を設定している
4. 小計には金額の縦計を求める計算式を設定している
5. 消費税には小計から消費税を求める計算式を設定している
6. 合計には小計と消費税の合計を求める計算式を設定している

これらをExcelのテンプレート側に設定することで、プログラムコードからの設定を最低限に抑えることを可能としています。こう見るとExcelはまさにリアクティブプログラミングですね。

テンプレートに値を設定し、PDFとして保存する

では実際にPDFを出力するコードをご覧いただきましょう。コードの全体は**こちら**に公開しています。

まずはライセンスを有効化します。製品版またはトライアル版のキーを指定してください。このコードを記述しない場合は制限付きのライセンスなし版として動作します。

```
Workbook.SetLicenseKey("YOUR_KEY");
```

続いて請求書へ出力する請求データを作成します。

```
var invoice = new Invoice
{
    No = 1234567890,
    BillingDate = DateTime.Today,
    PaymentDeadline = DateTime.Today.Add(TimeSpan.FromDays(30)),
    Customer = new Customer
    {
        ZipCode = "108-0023",
        Address = "東京都港区芝浦3-4-1 グランパークタワー",
        Name = "ニュイ工房",
        Staff = "中村 充志"
    },
    InvoiceDetails = new List<InvoiceDetail>
    {
```

```
        new InvoiceDetail{ Name = "Item1", UnitPrice = 1000, Count = 1, Unit = "本"},
        new InvoiceDetail{ Name = "Item2", UnitPrice = 2000, Count = 2, Unit = "冊"},
        new InvoiceDetail{ Name = "Item3", UnitPrice = 3000, Count = 3, Unit = "枚"},
        new InvoiceDetail{ Name = "Item4", UnitPrice = 4000, Count = 4, Unit = "ｾﾝ"},
        new InvoiceDetail{ Name = "Item5", UnitPrice = 5000, Count = 5, Unit = "束"},
        new InvoiceDetail{ Name = "Item6", UnitPrice = 6000, Count = 6, Unit = "年"},
        new InvoiceDetail{ Name = "Item7", UnitPrice = 7000, Count = 7, Unit = "個"},
        new InvoiceDetail{ Name = "Item8", UnitPrice = 8000, Count = 8, Unit = "個"},
        new InvoiceDetail{ Name = "Item9", UnitPrice = 9000, Count = 9, Unit = "個"},
        new InvoiceDetail{ Name = "Item10", UnitPrice = 10000, Count = 10, Unit = "個"},
    }
};
```

BillingDateやPaymentDeadlineにはDateTime型をフォーマットした文字列ではなく、DateTime型が設定され、UnitPriceにもint型がそのまま設定されていることをご覧ください。

続いてExcelを開き、作成したinvoiceの情報を適用していきます。

```
var workbook = new Workbook();
workbook.Open("Invoice.xlsx");
var worksheet = workbook.ActiveSheet;
worksheet.Range["N3"].Value = invoice.No;
worksheet.Range["N4"].Value = invoice.BillingDate;
worksheet.Range["O13"].Value = invoice.PaymentDeadline;
worksheet.Range["B3"].Value = invoice.Customer.ZipCode;
worksheet.Range["A4"].Value = invoice.Customer.Address;
worksheet.Range["A5"].Value = invoice.Customer.Name;
worksheet.Range["C6"].Value = invoice.Customer.Staff;

for (var i = 0; i < invoice.InvoiceDetails.Count; i++)
{
    worksheet.Range[15 + i, 1].Value = invoice.InvoiceDetails[i].Name;
    worksheet.Range[15 + i, 9].Value = invoice.InvoiceDetails[i].UnitPrice;
    worksheet.Range[15 + i, 12].Value = invoice.InvoiceDetails[i].Count;
    worksheet.Range[15 + i, 14].Value = invoice.InvoiceDetails[i].Unit;
}
```

一点注意があります。Excelの場合、通常配列のインデックスは1始まりで利用しますが、DioDocsでは0始まりで扱います。この点がExcelオブジェクトに完全に準拠していない点になります。しかしプログラムの都合上、配列などの添え字が0始まりであることを考慮すると、その選択肢もアリなのかもしれません。

そして最後に、値の設定されたExcelをPDFへ保存します。

```
workbook.Save("Result.pdf", SaveFileFormat.Pdf);
```

たったこれだけ！ 簡単すぎる。でき上がった帳票は、次の通りです。

でき上がった帳票

Amazing！ 素晴らしい！ こんなに簡単な帳票の実装は、過去に記憶がありません。

ちゃんと日付は「yyyy年mm月dd日」で、金額はカンマ区切りでフォーマットされているのが見て取れるでしょう。そして金額関係の式が正しく計算されていることも分かります。

もちろん複雑な帳票であれば、Excelベースではない方が良いケースも多くありますが、単純な単票や、ややページングするだけの帳票であれば、もうすべてこれだけで良いのではないか？ そんな風に思えてきます。簡単なのもありますが、何より誰もが使い慣れたExcelをそのまま利用できて、学習コストが低く抑えられる点も見逃せません。

DioDocs for PDF

さて、DioDocs for Excelがあまりにも個人的に衝撃的すぎた結果、ここまでExcel側の言及が多かったかもしれませんが、DioDocs for PDFも素性の良い製品です。

まずここまでお話ししてきた、次のような「バランスの良さ」はDioDocs for PDFも兼ね備えています。

- .NET Standard 2.0準拠
- ランタイムフリー
- 開発元は日本が本社の企業なので直接サポートが受けられる
- 軽快な動作速度

PDF操作ライブラリの選定を考えた場合、.NET Standard 2.0準拠であるというのは非常に大きな加点要素となります。PDFを扱うとなると、画像を扱う頻度が高くなりますが、先にも説明したように.NET Framework標準の画像ライブラリはサーバープロセスでの利用に耐え得るものではありません。.NET Standardはクロスプラットフォーム対応が前提のため、必然的にWindowsの固有機能であるGDI+へ依存してないことが期待できます。

先にお話ししたPDFSharpがまさに画像処理にGDI+に依存しており、サーバープロセスでの利用ははばかられます。

有償製品を考えた場合、サーバープロセスでの動作を保証している製品はそれなりにありますが、ランタイムライセンスを必要とする製品の割合が高いように思います。先にも記載しましたが、クラウドでスケールアウトすることを考えると、コア数ベースなどのランタイムライセンスは躊躇してしまいます。そもそもFaaSなどでコア数ライセンスを適切に運用できるのでしょうか？

また有償製品を購入するとした場合に、国内で購入するにあたって販売代理店ではなく、開発会社の日本法人が存在するのは、有償製品を利用するケースでは比較的重要な評価指標だと私は考えているので、その点でも優位にあります。

これらの点を考慮するとDioDocs for PDFが非常に良いバランスの上に成り立っているのが理解いただけると思います。

ただしDioDocs for PDFについては、個人的には最も利用したい機能の1つが日本版にはまだリリースされていませんでした。具体的には複数のPDFを結合する機能です。

※複数のPDFを結合する機能は、2019年2月27日にリリースされた最新バージョンのDioDocs for PDFより利用できます。

そこでDioDocsの英語版である「GrapeCity Documents for PDF（以下、GCPDF）」の機能を紹介したいと思います。GCPDFでは先行して機能がリリースされています。DioDocsの次のバージョンには現在のGCPDFの機能が含まれてリリースされるそうで、そうなればPDFも結合や分割の機能も提供されるようになります。日程は未確定ですが、そう遠くない未来にリリースされるようです。

では実際にPDF操作の例を見ていただきましょう。ここでは次の2つの例を紹介いたします。

- 複数のPDFを連結して、1つのPDFにまとめる
- PDFに電子署名を付与する

例としてこの2つを選択したのには明確な理由があります。

業務システムでデータを帳票として出力したいという場合、次のいずれかであるケースが多いでしょう。

- 紙に印刷して顧客などに配送する
- 電子帳票として出力してメールなどで送信する

前者の場合、1枚（もしくは1セット）ずつ印刷するのは非効率なので、単一のPDFファイルにまとめて一括印刷することが必要となるでしょう。

後者の場合、電子帳票は紙と違って改ざんが容易であるというデメリットがありますが、署名することによって発行者を明確にし、発行後に改ざんされていないことを保証することが可能になります。

というわけで、実際の例を見ていただきましょう。非常に簡単に実現できることが見て取れることと思います。

複数のPDFを連結して、1つのPDFにまとめる

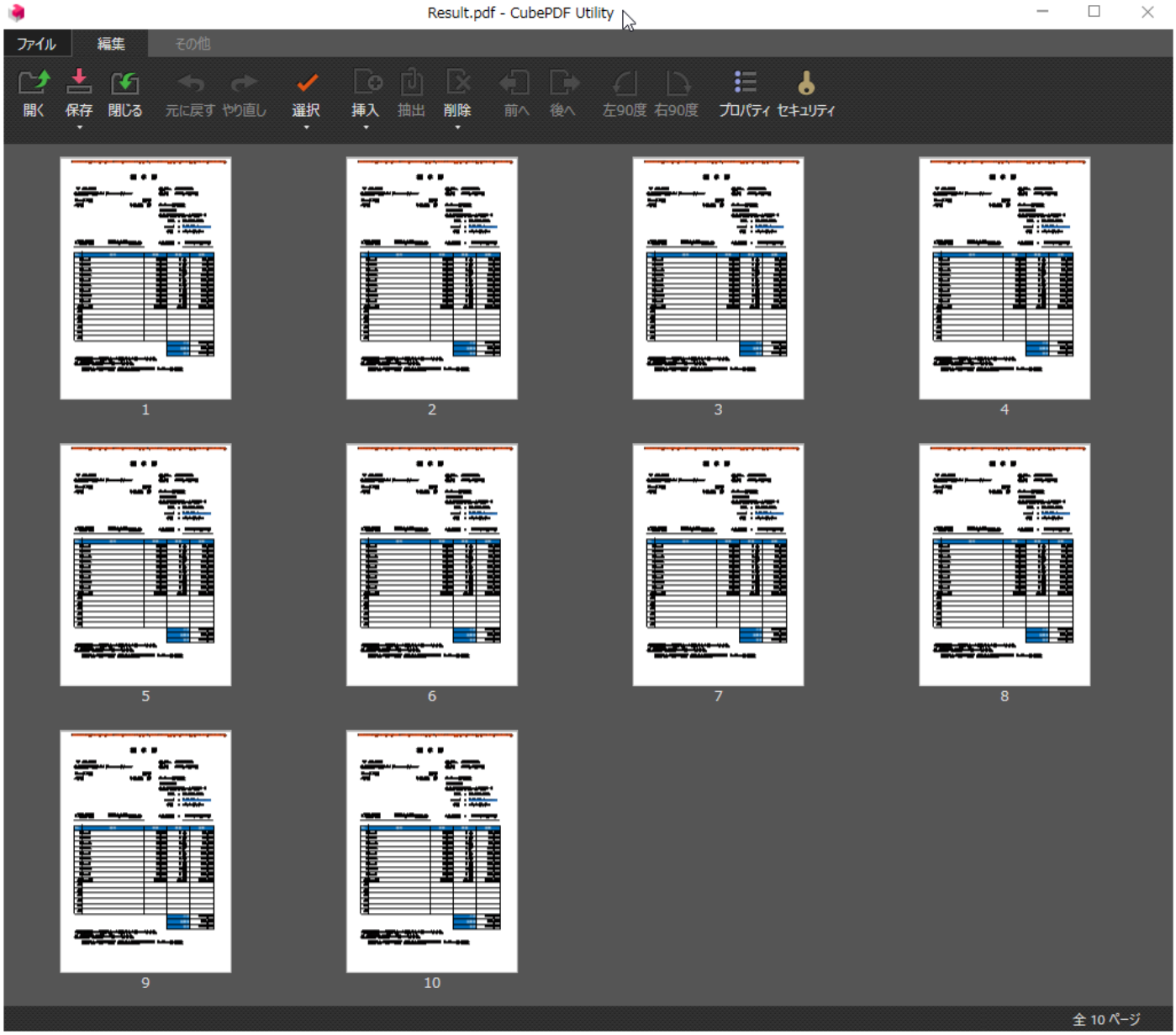
それでは実際のコードを紹介していきましょう。まずはライセンスを有効化します。

```
GcPdfDocument.SetLicenseKey("YOUR_KEY");
```

続いて、必要なPDFを都度読み込んで結合していきます。ここでは同じファイルになっていますが、10回読み込んで結合しています。

```
var newDoc = new GcPdfDocument();

for (var i = 0; i < 10; i++)
{
    using (var inputStream = new FileStream("Invoice.pdf", FileMode.Open))
    {
        var invoice = new GcPdfDocument();
        invoice.Load(inputStream);
        newDoc.MergeWithDocument(invoice, new MergeDocumentOptions());
    }
}

using (var outputStream = new FileStream("Result.pdf", FileMode.Create))
{
    newDoc.Save(outputStream);
    outputStream.Flush();
}
```

結合後に結果PDFとして出力しています。実際にPDFを開いてみてみましょう。



PDFを開いた様子

Result.pdfが10ページの同じ帳票で構成されているのが見て取れます。

さて、複数ページのPDFを作成するにあたって、実のところもう1つの方法があります。それはExcel上で全ページ分のシートを作成して、そこから一括で1枚のPDFを出力する方法です。

しかし、個人的にはPDFを1枚ずつ出力して最後に統合する方が好みです。

と言うのは、途中で何らかのエラーになった場合、エラーが発生する前までのPDFは残しておくことができるからです。エラーの直前までのファイルがあるなら、業務的に先にそれらを処理することも可能になります。

1つの大きな処理で完結するより、小さな処理を組み合わせた方が柔軟性が増すことが多いと思うからです。

PDFに電子署名を付与する

続いてデジタル証明書を利用してPDFに署名する例を紹介しましょう。デジタル証明書による署名を行うことで改ざんの検知などが可能となり、不正の抑止にもつながります。

では具体的な実装例を紹介しましょう。ライセンスはすでに有効化されているものとします。

まず署名対象のStreamと、署名後のファイル保存用のStreamを開き、GcPdfDocumentにPDFをロードします。

```
using (var inputStream = new FileStream("Invoice.pdf", FileMode.Open))
using (var outputStream = new FileStream("Signed.pdf", FileMode.Create))
{
    var doc = new GcPdfDocument();
    doc.Load(inputStream);
```

続いて署名をPDF上に表示する署名フィールドを作成し、ドキュメントに追加します。

```
// 署名を保持する署名フィールドを初期化
var signatureField = new SignatureField();
signatureField.Widget.Rect = new RectangleF(400, 750, 140, 36);
signatureField.Widget.Page = doc.Pages.Single();
signatureField.Widget.BackColor = Color.LightSeaGreen;
signatureField.Widget.TextFormat.FontName = "游ゴシック";
// ドキュメントに署名フィールドを追加
doc.AcroForm.Fields.Add(signatureField);
```

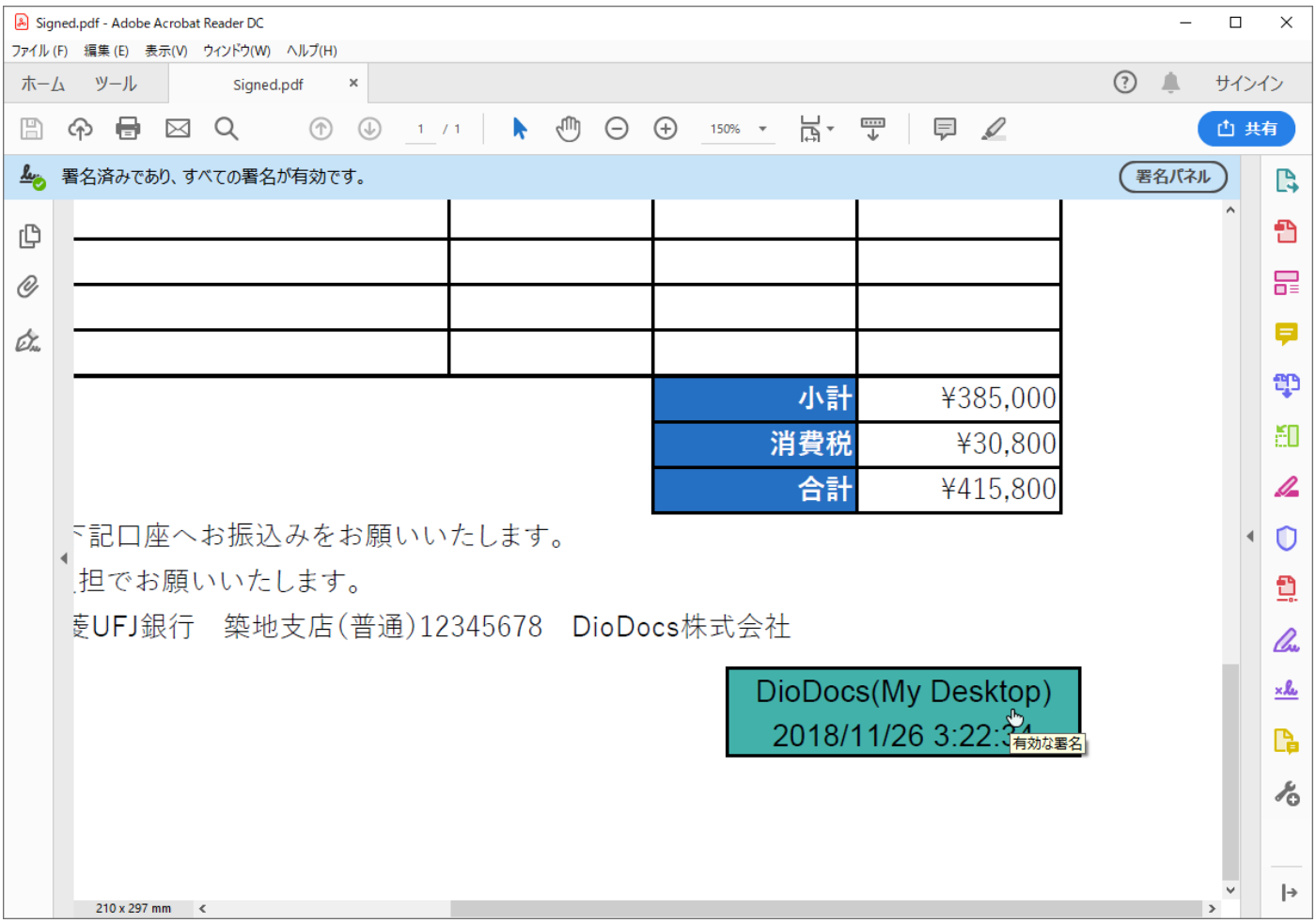
そして署名フィールドとひも付ける形で署名を作成します。

```
// 署名フィールドと署名を結びつけ
var signatureProperties = new SignatureProperties
{
    Certificate = new X509Certificate2(File.ReadAllBytes("diodocs.pfx"), "diodocs"),
    Location = "My Desktop",
    SignerName = "DioDocs",
    SignatureField = signatureField
};
```

最後にPDFを署名して保存します。

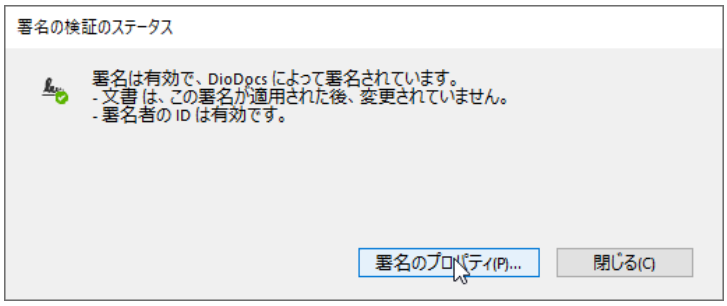
```
// 署名して文書を保存
// 注意
// - 署名と保存は一連の操作で、2つは分離できません
// - Sign()メソッドに渡されたストリームは読み込み可能である必要があります。
doc.Sign(signatureProperties, outputStream);
```

それでは署名されたファイルを開いてみましょう。



署名されたファイル

SignatureFieldで追加した領域が右下の緑の矩形の領域です。ここをクリックすると次のダイアログが表示されます。



「署名の検証のステータス」ダイアログ

さらに「署名のプロパティ」を開いてみましょう。

「署名のプロパティ」ダイアログ

適切に署名され、署名された日時から変更されていないことが確認できます。非常に簡単に署名を追加することができました。

さいごに

さて、ここまでお付き合いいただいた方は、DioDocsに大きな興味を持っていただけたものと思います。

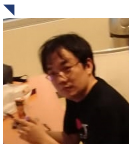
DioDocsは、掛け値なしに大変素晴らしい製品です。私の身近なユースケースで考えた場合、ユーザーメンテナンスが可能な帳票サービスを、クラウド上に作成するケースで非常に有効に働くと思います。

実際、簡便な帳票出力であれば、今後はDioDocsの採用を検討したいと私自身感じています。既存の帳票ライブラリと比較すると、厳密な出力制御や柔軟性は劣る面もあるかと思いますが、Excel＆DioDocsで事足りる範囲では圧倒的な生産性向上が見込める可能性があります。その利用範囲を広げるためにも、PDFの結合などを含んだDioDocs for PDFの次のバージョンの早期リリースを期待します。実際のプロダクトに採用してみて、深く掘り下げて評価・利用してみたいところです。

本稿はここで終了とさせていただきますが、別の機会に、具体的にユーザーメンテナンスが可能な帳票サービスを、DioDocsを利用してどう実現するか？ アーキテクチャ設計を踏まえて紹介したいと思います。アプリケーションレベルで活用する具体例になる予定です。ご期待ください。

バックナンバー WEB用を表示

著者プロフィール



中村 充志（リコージャパン株式会社）（ナカムラ アツシ）

Microsoft MVP for Visual Studio and Development Technologies　リコージャパン株式会社 金融事業部 金融ソリューション開発部所属。　エンタープライズ領域での業務システム開発におけるアプリケーション アーキテクト・プログラマおよび中間...

※プロフィールは、執筆時点、または直近の記事の寄稿時点での内容です

ページトップへ