The Journal of Open Source Software

PDFIO: PDF Reader Library for native Julia

Sambit Kumar Dash[1]

1 Director, Lenatics Solutions Pvt. Ltd.

DOI: 10.21105/joss.01453

Software

- Review
- Repository
- Archive

Submitted: 15 March 2019
Published: 14 November 2019

Summary

Portable Document Format (PDF) is the most ubiquitous file format for text, scientific research, legal documentation and many other fields for information presentation and dissemination. Being a final form format of choice, a large body of text is currently archived in this format. Julia is an upcoming programming language in the field of data sciences. Extracting archived content and understanding document metadata is beneficial to the language usage.

`PDFIO` is an API developed purely in Julia. Almost, all the functionalities of PDF understanding is entirely written from scratch in Julia with the only exception of certain (de)compression codecs and cryptography, where standard open source libraries are being used.

The following are some of the benefits of utilizing this approach:

1. PDF files have been in existence for over three decades. Implementations of PDF writers are not always accurate to the specification. They may even vary significantly from vendor to vendor. Every time someone gets a new PDF file, there is a possibility that file may not be interpreted as per the specification. A script-based language makes it easier for the consumers to quickly modify and enhance the code to their specific needs.

2. When a higher-level scripting language implements a C/C++ PDF library API, the scope is limited to achieving certain high- level tasks like graphics or text extraction, annotation or signature content extraction, or page extraction or merging.

   However, `PDFIO` represents the PDF specification as a model in the Model, View and Controller parlance. A PDF file can be represented as a collection of interconnected Julia structures. Those structures can be utilized in granular tasks or simply can be used to understand the structure of the PDF document.

   As per the PDF specification, text can be presented as part of the page content stream or inside PDF page annotations. An API like `PDFIO` can create two categories of object types. One representing the text object inside the content stream and the other for the text inside an annotation object. Thus, it provides flexibility to the API user.

3. Since, the API is written as an object model of PDF documents, it's easier to extend with additional PDF write or update capabilities. Although, the current implementation does not provide the PDF writing capabilities, the foundation has been laid for future extension.

There are also certain downsides to this approach:

1. Any API that represents an object model of a document tends to carry the complexity of introducing abstract objects. They can be opaque objects (handles) that are representational-specific to the API. They may not have any functional meaning. The

methods are granular and may not complete one use-level task. The amount of code needed to complete a user-level task can be substantially higher.

A comparative presentation of such approach can be seen in the illustration given below. A text extraction task that can be one simple method invocation in a competing library like `Taro`, can involve more number of steps in `PDFIO`. For example, in `PDFIO` the following steps have to be carried out:

a. Open the PDF document and obtain the document handle.

b. Query the document handle for all the pages in the document.
c. Iterate the pages and obtain page object handles for each of the pages.

d. Extract the text from the page objects and write to a file IO.

e. Close the document ensuring all the document resources are reclaimed.

2. The API user may need to refer to the PDF specification (PDF-32000-1:2008)(Adobe Systems Inc., 2008) for semantic understanding of PDF files in accomplishing some of the tasks. For example, the workflow of PDF text extraction above is a natural extension from how text is represented in a PDF file as per the specification. A PDF file is composed of pages and text is represented inside each page content object. The object model of `PDFIO` is a Julia language representation of the PDF specification.

Illustration

The popular package `Taro.jl`(Avik Sen, 2013) that utilizes Java based Apache Tika, Apache POI and Apache FOP libraries for reading PDF and other file types may need the following code to extract text and other metadata from the document.

```
using Taro
Taro.init()
meta, txtdata = Taro.extract("sample.pdf");
```

The same functionality with `PDFIO` may look like the code below:

```
function getPDFText(src, out)
    doc = pdDocOpen(src)
    docinfo = pdDocGetInfo(doc)
    open(out, "w") do io
        npage = pdDocGetPageCount(doc)
        for i=1:npage
            page = pdDocGetPage(doc, i)
            pdPageExtractText(io, page)
        end
    end
    pdDocClose(doc)
    return docinfo
end
```

While `PDFIO` requires a larger number of lines of code, it provides a more granular set of APIs for understanding the PDF document structure.

Functionality

PDFIO is implemented in layers enabling following features:

1. Extract and render the Contents in a PDF page. This ensures the contents are organized in a hierarchical grouping that can be used for rendering of the content. Rendering is used here in a generic sense and is not confined to painting on a raster device. For example, extracting document text can also be considered a rendering task. pdPageExtractText is an apt example of the same.
2. Provide functional tasks to PDF document access. A few of such functionalities are:
    - Getting the page count in a document (pdDocGetPageCount)
    - Finding labels in a document page (pdDocGetPageLabel)
    - Extracting outlines defined in the document (pdDocGetOutline)
    - Extracting document metadata information (pdDocGetInfo)
    - Validation of signatures in a PDF document (pdDocValidateSignatures)
    - Extracting fonts and font attributes (pdPageGetFonts, pdFontIsItalic etc.)
3. Access low level PDF objects (CosObject) and obtain information when high-level APIs do not exist. These kinds of functionalities are mostly related to the file structure of the PDF documents and also known as the COS layer APIs.

Acknowledgments

We acknowledge contributions of all the community developers who have contributed to this effort. Their contribution can be viewed at: https://github.com/sambitdash/PDFIO.jl/graphs/contributors

References

Adobe Systems Inc. (2008). *Document Management - Portable Document Format - Part 1: PDF 1.7*. Adobe Systems Inc. Retrieved from https://www.adobe.com/devnet/pdf/pdf_reference.html

Avik Sen. (2013). Taro.jl - Read and write Excel, Word and PDF documents in Julia. Retrieved from https://github.com/aviks/Taro.jl