

60 | 架构分解：边界，不断重新审视边界

2019-11-26 许式伟

许式伟的架构课

[进入课程 >](#)



讲述：姚迪迈

时长 11:13 大小 10.28M



你好，我是七牛云许式伟。

在上一讲 “[59 | 少谈点框架，多谈点业务](#)” 中，我们强调：

架构就是业务的正交分解。每个模块都有它自己的业务。

这里我们说的模块是一种泛指，它包括：函数、类、接口、包、子系统、网络服务程序、桌面程序等等。

接口是业务的抽象，同时也是它与使用方的耦合方式。在业务分解的过程中，我们需要认真审视模块的接口，发现其中 “过度的（或多余的）” 约束条件，把它提高到足够通用的、

普适的场景来看。

IO 子系统的需求与初始架构

这样说太抽象了，今天我们拿一个实际的例子来说明我们在审视模块的业务边界时，需要用什么样的思维方式来思考。

我们选的例子，是办公软件的 IO 子系统。从需求来说，我们首先考虑支持的是：

读盘、存盘；

剪贴板的拷贝（存盘）、粘贴（读盘）。

读盘功能不只是一是要能够加载自定义格式的文件，也要支持业界主流的文件格式，如：

Word 文档、RTF 文档；

HTML 文档、纯文本文档。


存盘功能更复杂一些，它不只是一是要支持保存为以上基于文本逻辑的流式文档，还要支持基于分页显示的文档格式，如：

PDF 文档；

PS 文档。

对于这样的业务需求，我们应该怎么做架构设计？

我第一次看到的设计大概是这样的：

 复制代码

```
1 type Span struct {
2     ...
3
4     SaveWord(ctx *SaveWordContext) error
5     SaveRTF(ctx *SaveRTFContext) error
6
7     LoadWord(ctx *LoadWordContext) error
8     LoadRTF(ctx *LoadRTFContext) error
9 }
```

```

10
11 type Paragraph struct {
12     ...
13     SpanCount() int
14     GetSpan(i int) *Span
15
16     SaveWord(ctx *SaveWordContext) error
17     SaveRTF(ctx *SaveRTFContext) error
18
19     LoadWord(ctx *LoadWordContext) error
20     LoadRTF(ctx *LoadRTFContext) error
21 }
22
23 type TextPool struct {
24     ...
25     ParagraphCount() int
26     GetParagraph(i int) *Paragraph
27
28     SaveWord(ctx *SaveWordContext) error
29     SaveRTF(ctx *SaveRTFContext) error
30
31     LoadWord(ctx *LoadWordContext) error
32     LoadRTF(ctx *LoadRTFContext) error
33 }
34
35 type Document struct {
36     ...
37     TextPool() *TextPool
38
39     SaveWord(stg IStorage) error
40     SaveRTF(f *os.File) error
41     SaveFile(file string, format string) error
42
43     LoadWord(stg IStorage) error
44     LoadRTF(f *os.File) error
45     LoadFile(file string) error
46 }

```

从上面的设计可以看出，读盘存盘的代码散落在核心系统的各处，几乎每个类都需要进行相关的修改。这类功能我们把它叫做“全局性功能”。我们下一讲将专门讨论全局性功能怎么做设计。

全局性功能的架构设计要非常小心。如果按上面这种设计，我们无法称之为一个独立的子系统，它完完全全是核心系统的一部分。

某种程度上来说，这个架构是受了 OOP 思想的毒害，以为一切都应该以对象为中心，况且在微软的 MFC 框架里面有 Serialization 机制支持，进一步加剧了写这类存盘读盘代码的倾向。


这当然是不太好的。在良好的设计中，一方面核心系统功能要少，少到只有最小子集；另一方面核心功能要能够收敛，不能越加越多。

但读盘存盘的需求是开放的，今天支持 Word 和 RTF 文档，明天支持 HTML，后天微软又出来新的 docx 格式。文件格式总是层出不穷，难以收敛。

Visitor 模式

所以，以上读盘存盘的架构设计不是一个好的架构设计。那么应该怎么办呢？可能有人会想到设计模式中的 Visitor 模式。

什么是 Visitor 模式？简单来说，它的目的是为核心系统的 Model 层提供一套遍历数据的接口，数据最终是通过事件的方式接收。如下：

 复制代码

```
1  type Visitor interface {
2      StartDocument(attrs *DocumentAttrs) error
3      StartParagraph(attrs *ParagraphAttrs) error
4      StartSpan(attrs *SpanAttrs) error
5      Characters(chars []byte) error
6      EndSpan() error
7      EndParagraph() error
8      EndDocument() error
9  }
10
11 type VisitableDoc interface {
12     Visit(visitor Visitor) error
13 }
14
15 type Document struct {
16     ...
17     Visit(visitor Visitor) error
18 }
19
20 func NewDocument() *Document
21 func LoadDocument(doc VisitableDoc) (*Document, error)
22
23 func SaveWord(stg IStorage, doc VisitableDoc) error
24 func SaveRTF(f *os.File, doc VisitableDoc) error
```

```
25 func SaveFile(file string, format string, doc VisitableDoc) error
26
27 func LoadWord(stg IStorage) (VisitableDoc, error)
28 func LoadRTF(f *os.File) (VisitableDoc, error)
29 func LoadFile(file string) (VisitableDoc, error)
```

这样做的好处是显然的。


一方面，核心系统为 IO 系统提供了统一的数据访问接口。这样 IO 子系统就从核心系统中抽离出来了。

另一方面，Word 文档的支持、RTF 文档的支持这些模块在 IO 子系统也彼此完全独立，却又相互可以非常融洽地进行配合。比如我们可以很方便将 RTF 文件转为 Word 文件，代码如下：

 复制代码

```
1 func ConvRTF2Word(rtf *os.File, word IStorage) error {
2     doc, err := LoadRTF(rtf)
3     if err != nil {
4         return err
5     }
6     return SaveWord(word, doc)
7 }
```

类似地，加载一个 Word 文件的代码如下：

 复制代码

```
1 func LoadWordDocument(stg IStorage) (*Document, error) {
2     vdoc, err := LoadWord(stg)
3     if err != nil {
4         return nil, err
5     }
6     return LoadDocument(vdoc)
7 }
```

那么这个设计有什么问题？

如果你对比上一讲 “[🔗 59 | 少谈点框架，多谈点业务](#)” 提到的 SAX 和 DOM 模式，很容易看出这里的 Visitor 模式本质上就是 SAX 模式，只不过数据源不再是磁盘中的文件，而是换成了核心系统的 Model 层而已。

所以我前面讲的 SAX 模式的缺点它一样有。它最大的问题是有预设的数据访问逻辑，其客户未必期望以相同的逻辑访问数据。

基于事件模型是一个非常简陋的编程框架，与大部分 IO 子系统的需求方，比如我们这里的 Word 文档存盘、RTF 文档存盘的诉求并不那么匹配。解决这种不匹配的常规做法是把数据先缓存下来，等到我当前步骤所有需要的数据都已经发送过来了，再进行处理。

这个设计并不是假想的，实际上我当年在做 WPS Office IO 子系统第一版本的架构设计时，就采用了这个架构。但最终实践下来，我自己总结的时候认为它是一个非常失败的设计。

一方面，虽然 Visitor 或者 SAX 模式看起来是 “简洁而高效” 的，但是实际编码中程序员的心智负担比较大，有大量的冗余代码纯粹就是为了缓存数据，等待更多有效的数据。

另一方面，这个接口仍然是抽象而难以理解的。比如，不同事件的次序是什么样的，需要较长的文档说明。

这也是给架构师们提了个醒，我们架构设计的 KISS 原则提倡的简单，并不是接口外观上的简洁，而是业务语义表达上的准确无歧义。

IO DOM 模式

所以第二次的架构迭代，我们调整为基于 DOM 模式，如下：

 复制代码

```
1 type IoSpan interface {
2     Text() []byte
3     Attributes() IoSpanAttrs
4 }
5
6 type IoSpans interface {
7     Len() int
8     Elem(i int) IoSpan
9 }
```

```

10
11 type IoParagraph interface {
12     Spans() IoSpans
13     Attributes() IoParagraphAttrs
14 }
15
16 type IoParagraphs interface {
17     Len() int
18     Elem(i int) IoParagraph
19 }
20
21 type IoDocument interface {
22     Paragraphs() IoParagraphs
23     Attributes() IoDocumentAttrs
24 }
25
26 func NewIoDocument() IoDocument
27
28 type Document struct {
29     ...
30     Io() IoDocument
31 }
32
33 func NewDocument() *Document
34
35 func SaveWord(stg IStorage, doc IoDocument) error
36 func SaveRTF(f *os.File, doc IoDocument) error
37 func SaveFile(file string, format string, doc IoDocument) error
38
39 func LoadWord(stg IStorage, doc IoDocument) error
40 func LoadRTF(f *os.File, doc IoDocument) error
41 func LoadFile(file string, doc IoDocument) error

```

在这个架构，我们认为有两套 DOM，一套是 IO DOM，即 IoDocument 接口及其相关的接口。一套是核心系统自己的 DOM，也就是 Document 类及其相关的接口。这两套接口几乎是雷同的，理论上 Document 只是 IoDocument 这个 DOM 的超集。

那么为什么不是直接在接口上体现出超集关系？从语法表达上很难，毕竟这是一个接口族，而不是一个接口。这里我们通过 Document 类引入 Io() 函数来将其转为 IoDocument 接口，以体现双方的超集关系。

在这个方案下，将 RTF 文件转为 Word 文件的代码如下：

```

1 func ConvRTF2Word(rtf *os.File, word IStorage) error {

```

 复制代码

```
2  doc := NewIoDocument()
3  err := LoadRTF(rtf, doc)
4  if err != nil {
5      return err
6  }
7  return SaveWord(word, doc)
8 }
```

类似地，加载一个 Word 文件的代码如下：

 复制代码

```
1 func LoadWordDocument(stg IStorage) (*Document, error) {
2     doc := NewDocument()
3     err := LoadWord(stg, doc.Io())
4     if err != nil {
5         return nil, err
6     }
7     return doc, nil
8 }
```

相比前面的 Visitor 模式，采用 IO DOM 除了让所有存盘读盘的模块代码工程量变低，接口的理解一致性更好外，还有一个额外的好处，是 IO DOM 更自然，避免了惊异。因为核心系统的 Model 层通常就是通过 DOM 接口暴露的，而 IO DOM 从概念上只是一个子集关系，显然对客户理解成本来说是最低的。而 Visitor 模式你可以理解为它是核心系统 Model 层为 IO 子系统提供的专用插件机制，它对核心系统来说是额外的成本。

事实上，在 DOM 模式基础上提供 Visitor 模式是有点多余的。DOM 模式通常提供了极度灵活的数据访问接口，可以适应几乎所有的数据读取场景。

回到最初的需求

我们是否解决了最初 IO 子系统的所有需求？

我们简单分析下各类用户故事（User Story）就能够发现其实并没有。我们解决了所有流式文档的存盘读盘，但是没有解决基于分页显示的文档格式支持，如：

PDF 文档；

PS 文档。

因为从核心系统 DOM 得到的文档，或者我们抽象的 IO DOM，都是流式文档，并没有分页信息。如果我们 PDF、PS 文档的存盘接口是这样的：

 复制代码

```
1 func SavePDF(f *os.File, doc IoDocument) error
2 func SavePS(f *os.File, doc IoDocument) error
```

那么意味着这些存盘模块的实现者需要对 IO DOM 进行排版（Render），得到具备分页信息的数据结构，然后以此进行存盘。


这意味着 IO 子系统在特定的场景下，其实与排版与绘制子系统相关，包括：

屏幕绘制（onPaint）；

打印（onPrint）。

可能有些人能够回忆起来，前面在 “[🔗22 | 桌面程序的架构建议](#)” 一讲介绍 Model 和 ViewModel 之间的关系时，我也是拿 Office 文档举例。核心系统的 DOM，或者 IO 子系统的 IO DOM，通过排版（Render）功能，可以渲染出 View 层所需的显示数据，我们不妨称之为 View DOM。

而有了 View DOM，我们就不只是可以进行屏幕绘制和打印，也可以支持 PDF/PS 文档的存盘了。代码如下：

 复制代码

```
1 func Render(doc IoDocument) (ViewDocument, error)
2
3 func SavePDF(f *os.File, doc ViewDocument) error
4 func SavePS(f *os.File, doc ViewDocument) error
```

如果你做需求分析的时候，没有把这些需求关联性找到，那就不是一次合格的需求分析过程。


不断重新审视边界

到此为止，我们的分析是否已经足够细致，把所有关键细节都想得足够清楚？

其实并没有，我们在理需求时，我们首先要考虑支持的是：


剪贴板的拷贝（存盘）、粘贴（读盘）。

但是我们在整理用户故事（User Story）的时候仍然把它给漏了。当然，剪贴板带来的影响没有 PDF/PS 文档大，它只是意味着我们的数据流不再是 `*os.File` 可以表达，而是需要用更抽象的 `io.Reader/Writer` 来表示。也就是说，以下接口：

 复制代码

```
1 func SaveRTF(f *os.File, doc IoDocument) error
2 func LoadRTF(f *os.File, doc IoDocument) error
3
4 func SavePDF(f *os.File, doc ViewDocument) error
5 func SavePS(f *os.File, doc ViewDocument) error
```


要改为：

 复制代码

```
1 func SaveRTF(f io.Writer, doc IoDocument) error
2 func LoadRTF(f io.Reader, doc IoDocument) error
3
4 func SavePDF(f io.Writer, doc ViewDocument) error
5 func SavePS(f io.Writer, doc ViewDocument) error
```

这其实就是我前面强调的“发现模块接口中多余的约束”的一种典型表现。在我们模块提高到足够通用的、普适的场景来看时，实际上并不需要剪贴板这样具体的用户场景，也能够及时地发现这种过度约束。


另外，我们的 IO 子系统的入口级的接口：

 复制代码

```
1 func SaveFile(file string, format string, doc IoDocument) error
2 func LoadFile(file string, doc IoDocument) error
```


我们且不说这里面怎么实现插件机制，以便于我们非常方便就能够不修改任何代码，就增加一种新的文件格式的读写支持。我们单就它的边界来看，也需要进一步探讨。

其一，LoadFile 方法我们可能希望知道加载的文件具体是文档格式，所以应该改为：

 复制代码

```
1 func LoadFile(file string, doc IoDocument) (format string, err error)
```


其二，考虑到剪贴板的支持，我们输入的数据源不一定是文件，还可能是 io.Reader、IStorage 等，在 Windows 平台下有 STGMEDIUM 结构体来表达通用的介质类型，可以参考。从跨平台的角度，也可以考虑直接用 Go 语言中的任意类型。如下：

 复制代码

```
1 func Save(src interface{}, format string, doc IoDocument) error
2 func Load(src interface{}, doc IoDocument) (format string, err error)
```

既然用了 interface{} 这样的任意类型，就意味着我们需要在文档层面上补充清楚我们都支持些什么，不支持些什么，避免在团队共识上遇到麻烦。

其三，考虑 PDF/PS 这类非流式文档的支持，我们不能用 IoDocument 作为输入文档的类型。也就是说，以下接口：

 复制代码

```
1 func Save(dest interface{}, format string, doc IoDocument) error
```

需要作出适当的调整。具体应该怎么调？欢迎留言发表你的观点。

结语

这一讲我们通过一个实际的例子，来剖析架构设计过程中我们如何在思考模块边界。

最重要的，当然是职责。不同的业务模块，分别做什么，它们之间通过什么样的方式耦合在一起。这种耦合方式的需求适应性如何，开发人员实现上的心智负担如何，是我们决策的影响因素。

为了避免留下难以调整的架构缺陷，我们强烈建议你认真细致做好需求分析，并且在架构设计时，认真细致地过一遍所有的用户故事（User Story），以确认我们的架构适应性。

最后，我们在具体接口的每个输入输出参数的类型选择上，一样要非常考究，尽可能去发现其中“过度的（或多余的）”约束。

如果你对今天的内容有什么思考与解读，欢迎给我留言，我们一起讨论。下一讲我们的话题按照大纲是“全局性功能的架构设计”，但我计划做一篇加餐，内容是架构思维实战，把前面我们的实战案例“画图程序”和这几讲的理论知识结合起来。

大家可以提前思考以下内容：对画图程序进行子系统的划分，我们的哪些代码是核心系统，哪些是周边系统？从判断架构设计的优劣的角度，我们如何评判它好还是不好？

如果你自己也实现了一个“画图程序”，可以根据这几讲的内容，对比一下我们给出的样例，和自己写的有哪些架构思想上的不同，怎么评价它们的好坏？

如果你觉得有所收获，也欢迎把文章分享给你的朋友。感谢你的收听，我们下期再见。




许式伟的架构课

从源头出发，带你重新理解架构设计

许式伟
七牛云 CEO



新版升级：点击「请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言 (9)

写留言



Sam

2019-11-27

许大，请教您一个问题。文中提到的如下代码片段：

```
func Save(src interface{}, format string, doc ioDocument) error
```

```
func Load(src interface{}, doc ioDocument) (format string, err error)
```

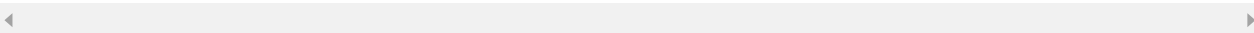
其中format参数有何用意，麻烦指点下。

第二个： `func Save(dest interface{}, format string, doc ioDocument) error...`

展开 ▾

作者回复：1、format就是要保存的文件格式；

2、其实你说的是一个好方法，我也用的是这个方法。



2



丁丁历险记

2019-11-26

这下好了，满脑子架构就是业务的正交分解了。。。。。



1



许式伟-七牛云(已满...)

2019-11-26

其实，这里面有一个隐含的决策没有交代，为什么有引入 IO DOM，直接拿核心系统的 DOM 来作为 IO 系统依赖行不行？欢迎留言探讨。

展开 ▾



1



吴

2019-11-26

越来越有味，这系列文章需要反复研究

展开 ▾



1





丁丁历险记

2019-11-26

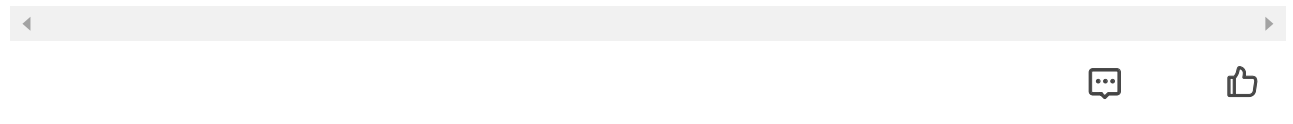
最近买了打印机，pdf 打出来模糊，wps pdf 转word 想的挺好，实施起来各种坑。想来想去，还是github 上找找pdf 转html的代码，毕竟ai 很成熟了。

然后html 打印。

毕竟对开发来说，调html样式 比编写word 容易太多。

展开 ∨

作者回复: pdf 转 word 的确比较复杂，这里没有讨论。



吴

2019-11-26

老大，go语言的入门书介绍一下，go语言擅长做啥了

展开 ∨

作者回复: 网上随意买一本都可以，Go语法确定性较强，掌握门槛比较低，关键在用起来。Go能够用的领域可以很广泛，主要做后端开发，单我甚至也用它做过游戏。



落石

2019-11-26

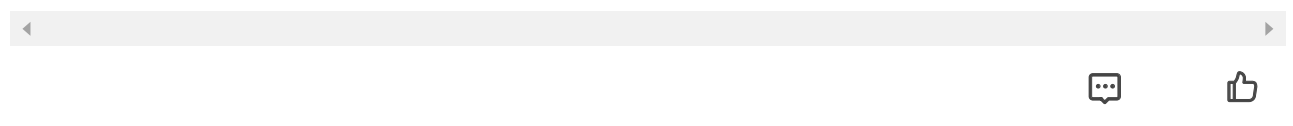
`func Save(dest interface{}, format string, func () interface{} documentLoader) error`

由调用方决定 document 的类型。

1. 将document也调整为父子类的形式。但隐约感觉到老师好像不太赞同继承？
2. 或者在调用时强转为 SaveWord 或 SavePDF 中的 IoDocument 和 ViewDocument

展开 ∨

作者回复: 仔细想想，是否可以解决问题



Aaron Cheung

2019-11-26

模块边界 受教了

期待架构思维实战篇

展开 



leslie

2019-11-26

期待老师的《架构思维设计》：希望老师可以在阳历年之前分享出来，这样可以更好的用在自己的将来新项目上。

一路跟着老师学习还是觉得受益匪浅，许多思路梳理清楚了。看到很多扩展性的坑、边界有时需要自己不断的调整到更高的高度，完成了一个类似看到了、审清还要做清，例如：架构的分级乱、接口乱、代码的效率/规范化乱、执行乱。...

展开 

