

65 | 架构范式：文本处理

2019-12-17 许式伟

许式伟的架构课

[进入课程 >](#)



讲述：姚迪迈

时长 17:55 大小 16.42M



你好，我是七牛云许式伟。

上一讲 “[64 | 不断完善的架构范式](#)” 我们提到架构师的武器库是不断完善的架构范式。今天我们围绕一个具体的问题域，看看我们日常能够积累什么样的经验和成果，来完善作为一个架构师的知识体系。

我们选择的问题是 “文本处理”。

计算机之所以叫计算机，是因为计算机的能力基本上就是 “计算 + I/O” 两部分。I/O 只是为了让计算机与物理世界打交道，它也是为计算服务的。所以数据是软件的灵魂，数据处理是软件的能力。

今天我们聊的文本处理，不是通用的数据处理能力，而是收敛在数据的 I/O 上。这里说的文本，是指写入到磁盘的非结构化数据。它可能真的是文本内容，比如 HTML 文档、CSS 文档；也可能是二进制内容，比如 Word 文档、Excel 文档。文本处理则是指对这类非结构化数据的处理过程，常见文本处理的需求场景有：

数据验证 (Data Validation) 。比如判断用户输入的文本是否合法，值的范围是否符合期望。

数据抽取 (Data Extraction) 。比如从某 HTML 页面中抽取出结构化的机票信息（什么时间，从哪里出发，到哪里去，价格几何等等）。

编译器 (Compiler) 。特殊地，在文本格式是某种语言的代码时，我们可以将文本编译成可执行的机器码，或虚拟机解释执行的字节码。当然我们也可以边解释文本的语义边执行。

.....

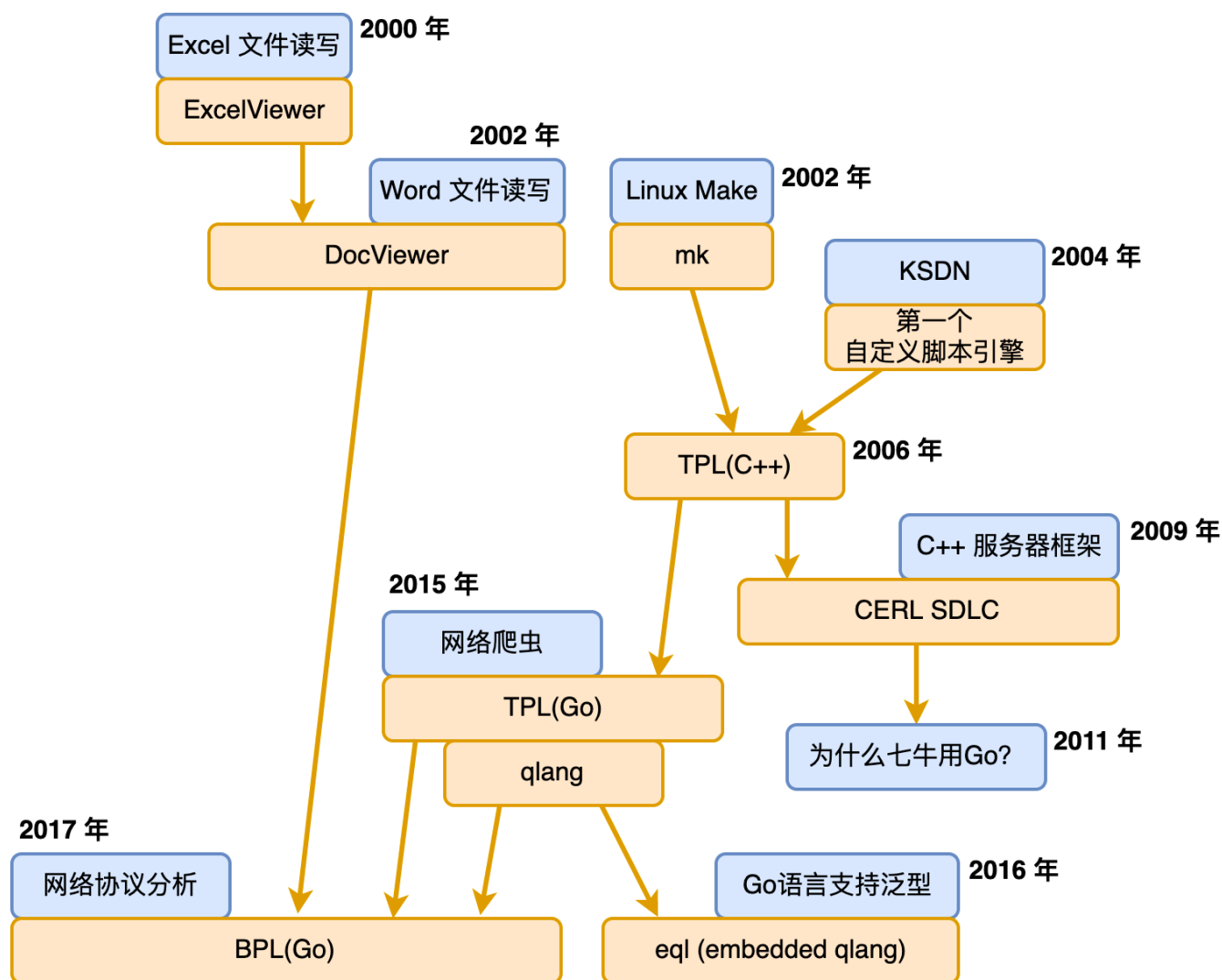
从用户需求的角度来说，文本处理的需求场景是不可穷尽的。网络爬虫与搜索引擎需要文本处理，Office 软件需要文本处理，编程语言的编译器需要文本处理，网络协议解析需要文本处理，等等。

那么，怎么才能从这些多变的需求场景中，抽出正交分解后可复用的架构范式？

我们今天聊聊文本处理的通用思路。

我的文本处理技术栈演进

文本处理，很多人都会遇到，只不过大家各自遇到的场景不同。我这里先回顾下我个人遇到的文本处理场景。我总结了一个图，如下：



在 2000 年初，我作为实习生拿到的第一个任务，是金山电子表格自身的文件格式设计和 Excel 文件的读写。此后，我参与了多个版本的 Word 文件读写工作。为了便于分析 Excel 和 Word 文件的格式，我实现了 ExcelViewer 和 DocViewer 这两个文件格式查看器。

实际上这两个 Viewer 非常重要，因为它第一次让文件格式的理解过程用程序固化了下来。这非常利于知识传承。大家可以设想一下，假如没有 Viewer，那么后面接手的人基本上只能靠阅读 ExcelReader 和 DocReader 模块的代码来理解文件格式。

但是这有两个问题。其一，Reader 模块有大量的业务逻辑，对我们理解 Excel 和 Word 文件格式本身会造成干扰。其二，Reader 模块增加功能会比较慢，对于那些我们本身不支持的功能，或者我们还暂时来不及兼容的功能，是没有对应的解析代码的。

但是 Viewer 就不一样。我们会尽可能地把我们对 Excel 和 Word 的理解记录下来，成为稳定可传承的知识，而无需关心是否已经支持该功能。另外，从时间的维度来说，应该先有 Viewer，在理解了文件格式之后，再设计出 Reader 才比较合理。

这个时期的 ExcelViewer 和 DocViewer，它主要抽象出来的是界面呈现部分。具体 ExcelViewer 和 DocViewer 的代码不需要有一行涉及到界面。这有诸多好处。实际上可视化界面只是 ExcelViewer 和 DocViewer 的一种输出终端，它们同时也生成了一个纯文本结果到磁盘文件中。这有助于我们用常规的 diff 工具来对比两个文件的差异，从而加速我们对未知数据格式的了解。

但，此时的 ExcelViewer 和 DocViewer 并没有将文件格式的处理过程抽象出通用的模块。也可以说，还没有抽象出文本处理范式。

这个时期同期还有一个探索性的 WPS for Linux 项目。为了支持跨平台编译，我实现了一个简单的 mk 程序。这个程序区别于 Linux 标准化的 make 程序，没有那么复杂的逻辑需要理解。它的输入是一个类 Windows 平台的 ini 文件，里面只需要指定选择的编译器、相关的编译选项、源代码文件列表等，就可以进行编译。甚至源代码列表可以直接指定为从 Visual C++ 的项目配置 dsp 文件中抽取，极易使用。

这个 mk 程序除了要解析一个类 ini 的配置文件外，也会解析 C/C++ 源代码文件形成源代码文件的依赖表，以更好地支持增量编译。不只是源代码文件本身的修改会触发重新编译，任何依赖文件的修改也会触发重新编译。

同样地，这个时期的 mk 程序同样没有引入任何通用的文本处理范式。

此后大约在 2004 年，我开始在金山办公软件内部推 KSDN。KSDN 这个名字承自 MSDN，我们希望打造一个全局的文档系统，它自动从项目的源代码中提取并生成。每天日构建完毕后得到最新版本的 KSDN。

KSDN 处理的输入主要是 C++ 和 Delphi 源代码文件（当时的界面是 Delphi 写的），是纯文本的。这和 ExcelViewer、DocViewer 不同，他们的输入是二进制文件。

KSDN 第一次引入了一个通用的脚本，来表达我们想从源代码中抽取什么内容。整个 KSDN 处理单个源代码文件的工作原理可以描述为：

通过文件后缀选择源代码文件的解析脚本，通过该脚本解析 C++ 或 Dephi 的源代码，并输出 XML 格式的文件；

通过 XSLT 脚本，将 XML 文件渲染为一个或多个 HTML 文件。XSLT 全称是 Extensible Stylesheet Language Transformations（可扩展样式表转换），是 XML 生态中的一项技术。

在 2006 年的时候，我决定实现 KSDN 2.0 版本。这个版本主要想解决第一个版本的脚本语法表达能力比较局限的问题。

于是 C++ 版本的 TPL（Text Processing Language）诞生了。它非常类似于 Boost Spirit，但功能要强大很多。它的项目主页为：

🔗 <https://github.com/xushiwei/tpl>

它依赖基础库 stdext，项目主页为：

🔗 <https://github.com/xushiwei/stdext>

C++ 版本的 TPL 支持的表达能力，已经完全不弱于 UNIX 经典的 LEX + YACC 组合，使用上却轻量很多。KSDN 2.0 的工作原理变成了：

基于 TPL 将 C++ 或 Delphi 文件转为 json 格式；

与 XSLT 类似地，我们引入了 JSPT，即以 json 为输入，PHP 为 formatter，将内容转为一个或多个 HTML 文件。

这个过程非常通用，可以用于实现任意文件格式之间的变换。包括我们前面的 mk 程序，它本质上也是类 ini 文件格式变换到 Makefile 的过程，我们基于 TPL 很轻松就改造了一个 mk 2.0 版本。

2009 年的时候，我们基于 C++ 实现一个名为 CERL 的网络库，它和 Go 语言的 goroutine 类似，也是基于协程来实现高并发。在这个网络库中，我们定义了一个名为 SDL（Server Description Language）的语言来描述服务器的网络协议。很自然地，我们基于 TPL + JSPT 来实现了 SDL 文件的解析过程。

2011 年，七牛云成立，我们选择了 Go 语言作为技术栈的核心。在转 Go 语言后，除了 TPL，我个人沉淀的大部分 C++ 基础库都不再需要，因为它们往往已经被 Go 语言的标准

库解决得很好。

在 2015 年的时候，出于某种原因我实现了一个网络爬虫，这个爬虫会在收到网页内容后，抽取网页中的结构化信息并存储起来。这个抽取信息的过程，最终导致 Go 语言版本 TPL 的诞生。它的项目主页为：

🔗 <https://github.com/qiniu/text>

为了验证 Go 语言版本 TPL 的有效性，我在实现了经典的 “计算器 (Calculator) ” 之余，顺手实现了一门语言，这就是 qlang。它的项目主页为：

🔗 <https://github.com/qiniu/qlang>

由于 Go 语言中实现泛型数据结构的需要，我给 qlang 实现了一个 embedded 版本，简称 eql。它是类似 erubis/erb 的东西。结合 go generate，它可以很方便地让 Go 支持模板（不是 html template，是指泛型）。

在 2017 年，出于 rtmp 网络协议理解的需要，我创建了 BPL (Binary Processing Language)，它的项目主页为：

🔗 <https://github.com/qiniu/bpl>

区别于 TPL 的是，BPL 主要用于处理二进制文档。前面我们谈到 ExcelViewer 和 DocViewer 时说过，我们并没有建立任何通用的架构范式。这一直是我引以为憾的事情，所以 2006 年 C++ 版本的 TPL 诞生后就有过 BPL 相关的尝试。这里是尝试残留的痕迹：

🔗 [tpl/binary/*](https://github.com/qiniu/tpl/blob/master/binary/*)

但是二进制文档的确很难，它的格式描述中通常有一定的条件判断逻辑，所以 BPL 背后需要依赖一门语言。在 qlang 诞生后，这个条件就得到了满足，这是最终 BPL 得以能够诞生的原因。

BPL 非常强大，它可以处理任意的二进制文件，也可以用于处理任意的 TCP 网络协议数据流。有了 BPL，我们最初的 ExcelViewer 和 DocViewer 可以轻松得以实现。关于 BPL 更

详细的介绍，请参阅 [@https://github.com/qiniu/bpl](https://github.com/qiniu/bpl) 中的文档说明。

文本内容的处理范式


介绍了我个人文本处理的技术栈演进过程后，我们把话题重新回到架构范式。

首先，让我们把焦点放在文本内容的处理上。

文本内容的处理，有非常标准的方式。它通常分词法分析（Lex）和语法分析（Parser）两个阶段。UNIX 系的操作系统还提供了 lex 和 yacc 两个经典的程序来协助我们做文本文件的分析处理。


词法分析（Lex）通常由一个 Scanner 来完成，它负责将文本内容从字节流（Byte Stream）转为 Token 流（Token Stream）。我们以解析 Go 源代码的 Scanner 为例（参见 [@https://godoc.org/go/scanner](https://godoc.org/go/scanner)），其 Scan 函数的原型如下：

```
1 type Scanner struct {
2     Scan() (pos token.Pos, tok token.Token, lit string)
3     ...
4 }
```

 复制代码

其使用范式如下：

```
1 import (
2     "go/scanner"
3     "go/token"
4 )
5
6 func doScan(s *scanner.Scanner) {
7     for {
8         pos, tok, lit := s.Scan()
9         if tok == token.EOF {
10             break
11         }
12         ...
13         // pos 是这个 token 的位置
14         // tok 是这个 token 的类型，见 https://godoc.org/go/token
15         // lit 是这个 token 的文本内容
```

 复制代码

```
16 }
17 }
```

Scanner 有时候也叫 Tokenizer。例如 Go 语言中 HTML 的 Tokenizer 类（参阅 <https://golang.org/x/net/html>）的原型如下：

 复制代码

```
1 type Token struct {
2     Type      TokenType
3     DataAtom  atom.Atom
4     Data      string
5     Attr      []Attribute
6 }
7
8 type Tokenizer struct {
9     Next() TokenType
10    Err() error
11    Token() Token
12    ...
13 }
```

其使用范式如下：

 复制代码


```
1 import (
2     "golang.org/x/net/html"
3 )
4
5 func doScan(z *html.Tokenizer) error {
6     for {
7         if z.Next() == html.ErrorToken {
8             // Returning io.EOF indicates success.
9             return z.Err()
10        }
11        token := z.Token()
12        ...
13    }
14 }
```

词法分析 (Lex) 过程非常基础，大部分情况下我们不会直接和它打交道。我们打交道的基本都是语法分析器，通常叫 Parser。而从 Parser 的使用方式来说，分为 SAX 和 DOM 两

种模型。SAX 模型基于事件机制，DOM 模型则基于结构化的数据访问接口。

前面我们已经多次分析过 SAX 与 DOM 的优劣，这里不再展开。通常来说，我们会倾向于采用 DOM 模型。这里我们还是以 Go 文法和 HTML 文法的解析为例。

先看 Go 文法的 Parser（参阅 <https://godoc.org/go/parser>），它的原型如下：

 复制代码

```
1 func ParseExpr(x string) (ast.Expr, error)
2
3 func ParseFile(
4     fset *token.FileSet,
5     filename string, src interface{},
6     mode Mode) (f *ast.File, err error)
```

这里看起来有点复杂的是 ParseFile，它输入的字节流（Byte Stream）可以是：


scr != nil，且为 io.Reader 类型；

src != nil，且为 string 或 []byte 类型；

src == nil，filename 非空，字节流从 filename 对应的文件中读取。

而 Parser 的输出则统一是一个抽象语法树（Abstract Syntax Tree，AST）。显然，它基于的是 DOM 模型。

我们再看 HTML 文法的 Parser（参阅 <https://godoc.org/golang.org/x/net/html>），它的原型如下：

 复制代码

```
1 func Parse(r io.Reader) (*Node, error)
```


超级简单的基于 DOM 模型的使用接口，任何解释都是多余的。

那么，我前面提的 TPL（Text Processing Language）是做什么的呢？它实现了一套通用的 Scanner + Parser 的机制。首先是词法分析，也就是 Scanner，它负责将文本流转换为

Token 序列。简单来说，就是一个从 text []byte 到 tokens []Token 的过程。

尽管世上语言多样，但是词法非常接近，所以在词法分析这块，TPL 抽象了一个 Tokenizer 接口，方便用户自定义。TPL 也内置了一个与 Go 语言词法类似的 Scanner，只是做了非常细微的调整，增加了 ?、~、@ 等操作符。

TPL 的 Parser 通过类 EBNF 文法表达。比如一个浮点运算的计算器（Calculator），支持加减乘除、函数调用、常量（如 pi 等）的类 EBNF 文法如下：

 复制代码

```
1 term = factor *('*' factor/mul | '/' factor/quo | '%' factor/mod)
2
3 doc = term *('+ ' term/add | '- ' term/sub)
4
5 factor =
6     FLOAT/push |
7     '- ' factor/neg |
8     '(' doc ')' |
9     (IDENT '(' doc %= ', '/ARITY ')')/call |
10    IDENT/ident |
11    '+ ' factor
```

关于这个类 EBNF 文法，有以下补充说明：

我们用 *G 和 +G 来表示重复，而不是用 {G}。要记住这条规则其实比较简单。在编译原理的图书中，我们看到往往是 G* 和 G+。但语言文法中除了 ++ 和 -- 运算符，很少是后缀形式，所以我们选择改为前缀。

我们用 ?G 来表示可选，而不是用 [G]。同上，只要能够回忆起编译原理中我们用 G? 表示可选，我们就很容易理解这里为什么可选是用 ?G 表示。

我们直接用 G1 G2 来表示串接，而不是 G1, G2。

我们用 G1 % G2 和 G1 %= G2 表示 G1 G2 G1 G2 ... G1 这样的列表。其中 G1 % G2 和 G1 %= G2 的区别是前者不能为空列表，后者可以。在上面的例子中，我们用 doc %= , 表示函数的参数列表。

我们用 G/action 表示在 G 匹配成功后执行 action 这个动作。action 最终是调用到 Go 语言中的回调函数。在上面这个计算器中大量使用了 G/action 文法。

与 UNIX 实用程序 yacc 不同的是，TPL 中文法描述的脚本，与执行代码尽可能分离，以加业务语义的可读性。

从模型的归属来说，TPL 属于 SAX 模型。但 G/action 不一定真的是动作。在 extractor 模式下，G/action 被视为 G/marker，TPL 变成 DOM 模型。也就是说，此时 action 只是一个标记，用于形成输出的 DOM 树。

关于 TPL 更详细的介绍需要很长的篇幅，你可以参考 [🔗 TPL Doc](#)。


在文本内容处理的技术栈中，还有一个分支是正则表达式（Regular Expression）。在简单场景下，正则表达式是比较方便的，但是它的缺点也比较明显，可伸缩性和可读性都不强。

二进制内容的处理范式

接下来我们讨论二进制内容的通用处理范式。

二进制内容的处理过程整体来说，似乎比较“容易”。如果说出一点问题的话，那就是“有点繁琐”。

还记得序列化机制吧？它基本上算得上二进制内容的 I/O 框架了。它看起来是这样的：

 复制代码

```
1 type Foo struct {
2     A uint32
3     B string
4     C float64
5     D Bar
6 }
7
8 func readFoo(foo *Foo, ar *Archive) {
9     readUint32(&foo.A, ar)
10    readString(&foo.B, ar)
11    readFloat64(&foo.C, ar)
12    readBar(&foo.D, ar)
13 }
```

在 C++ 的操作符重载的支持下，这段代码看起来会更简洁一些：

[复制代码](#)

```
1 Archive& operator>>(Archive& ar, Foo& foo) {  
2     ar >> foo.A >> foo.B >> foo.C >> foo.D;  
3     return ar;  
4 }
```

当然，上面只是最基础的情形，所以看起来还比较简洁。但在考虑可选、重复、数组等场景，实际上并不会那么简单。比如对于数组，理想情况下代码是下面这样的：

[复制代码](#)

```
1 type Foo struct {  
2     N uint16  
3     Bars []Bar // [N]Bar  
4 }  
5  
6 func readFoo(foo *Foo, ar *Archive) {  
7     readUint16(&foo.N, ar)  
8     readArray(&foo.Bars, int(foo.N), ar)  
9 }
```

对于 Go 语言来说，这里我们想要的 `readArray` 并不存在。而在 C++ 则可以通过泛型来做到，我们示意如下：

[复制代码](#)

```
1 template <class T>  
2 void readArray(T[]& v, int n, Archive& ar) {  
3     v = new T[n];  
4     for (int i = 0; i < n; i++) {  
5         ar >> T[i];  
6     }  
7 }
```

呼唤一下 Go 语言的泛型吧。不过泛型大概率需要破坏 Go 的一些基础假设，比如不支持重载。所以 Go 的泛型之路不会那么容易。

回到序列化机制。常规意义的序列化，通常还提供了 Object 动态序列化与反序列化的能力。但是实际上这个机制属于过度设计。

为什么这么说？

因为 Object 动态序列化的确带来了一定的便捷性，但是这个便捷性的背后是让使用者放弃了对磁盘文件格式设计的思考。这是非常不正确的指导思想。


数据是软件的灵魂，文件是软件最重要的资产。

文件 I/O 的序列化机制，最重要的是定义严谨的数据格式，而非提供任何出于便捷性考虑的智能。

所以我们只需要保留序列化的形式就好了，任何额外的“智能”都是多余的。

基于这样的基本原则，稍作探究你就会发现，在数据结构清晰的情况下，其实整个序列化的代码是非常平庸的。假如我们参考 TPL 的类 EBNF 文法，定义以下这样一条规则：

```
1 Foo = {  
2   N uint16  
3   Bars [N]Bar  
4 }
```

 复制代码


这样，我们就可以自动帮这里的 Foo 类型实现它的序列化代码了。

而这正是 BPL 诞生的灵感来源。

BPL 设计的核心思想是，不破坏 TPL 的 EBNF 文法的任何语义，把自己作为 TPL 的扩展。这就好比，如果我们把 TPL 看作 C 的话，BPL 就是 C++。所有 TPL 的功能，BPL 都应该具备而且行为一致。

我们以 MongoDB 的网络协议为例，看看 BPL 文法是什么样的：

```
1 document = bson  
2  
3 MsgHeader = {/C  
4   int32    messageLength; // total message size, including this
```

 复制代码

```

5  int32  requestID;      // identifier for this message
6  int32  responseTo;    // requestID from the original request (used in respo
7  int32  opCode;        // request type - see table below
8  }
9
10 OP_UPDATE = {/C
11  int32  ZERO;          // 0 - reserved for future use
12  cstring fullCollectionName; // "dbname.collectionname"
13  int32  flags;         // bit vector. see below
14  document selector;    // the query to select the document
15  document update;      // specification of the update to perform
16 }
17
18 OP_INSERT = {/C
19  int32  flags;         // bit vector - see below
20  cstring fullCollectionName; // "dbname.collectionname"
21  document* documents;  // one or more documents to insert into the c
22 }
23
24 OP_QUERY = {/C
25  int32  flags;         // bit vector of query options. See below
26  cstring fullCollectionName; // "dbname.collectionname"
27  int32  numberToSkip;  // number of documents to skip
28  int32  numberToReturn; // number of documents to return
29                          // in the first OP_REPLY batch
30  document query;       // query object. See below for details.
31  document? returnFieldsSelector; // Optional. Selector indicating the field
32                          // to return. See below for details.
33 }
34
35 OP_GET_MORE = {/C
36  int32  ZERO;          // 0 - reserved for future use
37  cstring fullCollectionName; // "dbname.collectionname"
38  int32  numberToReturn; // number of documents to return
39  int64  cursorID;      // cursorID from the OP_REPLY
40 }
41
42 OP_DELETE = {/C
43  int32  ZERO;          // 0 - reserved for future use
44  cstring fullCollectionName; // "dbname.collectionname"
45  int32  flags;         // bit vector - see below for details.
46  document selector;    // query object. See below for details.
47 }
48
49 OP_KILL_CURSORS = {/C
50  int32  ZERO;          // 0 - reserved for future use
51  int32  numberOfCursorIDs; // number of cursorIDs in message
52  int64* cursorIDs;      // sequence of cursorIDs to close
53 }
54
55 OP_MSG = {/C
56  cstring message;      // message for the database

```

```

57 }
58
59 OP_REPLY = {/C
60     int32      responseFlags; // bit vector - see details below
61     int64      cursorID;      // cursor id if client needs to do get more's
62     int32      startingFrom;  // where in the cursor this reply is starting
63     int32      numberReturned; // number of documents in the reply
64     document*  documents;     // documents
65 }
66
67 OP_REQ = {/C
68     cstring    dbName;
69     cstring    cmd;
70     document   param;
71 }
72
73 OP_RET = {/C
74     document   ret;
75 }
76
77 Message = {
78     header MsgHeader // standard message header
79     let bodyLen = header.messageLength - sizeof(MsgHeader)
80     read bodyLen do case header.opCode {
81         1:      OP_REPLY      // Reply to a client request. responseTo is set.
82         1000: OP_MSG          // Generic msg command followed by a string.
83         2001: OP_UPDATE
84         2002: OP_INSERT
85         2004: OP_QUERY
86         2005: OP_GET_MORE // Get more data from a query. See Cursors.
87         2006: OP_DELETE
88         2007: OP_KILL_CURSORS // Notify database that the client has finished with
89         2010: OP_REQ
90         2011: OP_RET
91         default: {
92             body [bodyLen]byte
93         }
94     }
95 }
96
97 doc = *Message

```

我们对比 MongoDB 官方的协议文档（参考

<https://docs.mongodb.com/manual/reference/mongodb-wire-protocol/>），你会发现很有趣的一点是，我们 BPL 文法几乎和 MongoDB 官方采用的伪代码完全一致，除了一个小细节：在 BPL 中，我们用 {...} 表示采用 Go 语言结构体的文法，而 {/C ...} 表示采用 C 语言结构体的文法。

当前 BPL 还只支持解释执行，但这只是暂时的。就像在 TPL 中我们除了动态解释执行外，也已经提供 tpl generator 来生成 Go 代码以静态编译执行。

要进一步了解 BPL 的功能，请参阅 [🔗https://github.com/qiniu/bpl](https://github.com/qiniu/bpl)。我们也还提供了不少具体 BPL 的样例，详细可参考：

[🔗https://github.com/qiniu/bpl/tree/master/formats](https://github.com/qiniu/bpl/tree/master/formats)

结语

文本处理是一个非常庞大的课题，本文详细解剖了我个人在这个领域下的经验总结。相信这些经验对你面对相关场景时会有帮助。

但是更重要的一点是，我们平常需要有意识去分析我们工作中遇到的业务场景，从中提炼通用的需求场景形成架构范式的积累。

如此，架构的正交分解思想方能得到贯彻。而我们的业务迭代，也就越来越容易。

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。下一讲我们的话题是“架构老化与重构”。


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。

21 天打卡行动

- 99 元报名参与打卡
- 连续坚持 21 天
- 全额退还报名费

报名即赠 ¥199 奖金 

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。

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

精选留言 (3)

写留言



leslie

2019-12-17

"数据是软件的灵魂，文件是软件最重要的资产。"老师这话很经典：故而之前老师讲数据库时提出的是"中间件存储"时，通过大量的查阅资料和总结研究找到另外一个称呼"数据系统"。

从硬件层考虑"中间件存储"就是资产,软件层考虑“数据系统”就是灵魂。说个真实的案例，前段时间一个DB圈子的朋友数据库出问题，软件层各种排查做尽就是没看到问题...
展开



2



Aaron Cheung

2019-12-17

可运行的系统做重构 需要耗费不少时间 更需要团队支持😓

作者回复: 首先团队要有重构价值的共识，找到重构的步骤，然后才是执行。下一讲我们就是谈重构。



1



沫沫 (美丽人生)

2019-12-17

许老师好，如果有两个服务A和B，A依赖于B，同时B也依赖于A，出现这种情况是属于循环依赖吗？这种情况的出现，是业务的正交分解没有做好吗？如果系统中出现大量的这种依赖，感觉系统的调用就成了复杂的网络结构 这种情况，重构应该怎么做呢？盼复！

展开

作者回复: 循环依赖是从源代码层面的，不是运行时的网络调用。所以你说的不属于循环依赖。但如果大量存在两个服务之间的相互调用，解耦肯定也是没做好。高内聚低耦合，从功能内聚性看边界划分



1

