Tusd Server总结

一、Tusd基本概念

Tus是一个断点传输协议，Tusd是一个官方对该协议的一个实现。这是采用Go语言所写的一个Server，如果需要使用，自然得去要有对应的Client，Client的包或者库请参考 https://tus.io/implementations.html 。这里我们只讨论分析Tusd这个server。

在这个官方的server里，其实已经为我们提供好了一个实例。Tusd本质来说是Golang的包，而不是一个可以直接使用的程序，但是这个包里给我们提供了一个现成的例子，就在根目录下的 `cmd/tusd`文件夹里，我们执行：

```
git clone git@github.com:tus/tusd.git
cd tusd
go get -u github.com/aws/aws-sdk-go/...
go get -u github.com/prometheus/client_golang/prometheus
go build -o tusd cmd/tusd/main.go
```

这些命令在根目录下生成了一个叫tusd的可执行文件，因为Go语言本身是编译型的。其实这个就已经可以作为一个独立的server来运行了，而且同时兼顾储存在aws s3和本地的支持。官方的文档里给了一些例子。

文件保存在本地

```
$tusd -dir ./data
```

文件保存在aws s3上

```
$ export AWS_ACCESS_KEY_ID=xxxxx
$ export AWS_SECRET_ACCESS_KEY=xxxxx
$ export AWS_REGION=eu-west-1
$ tusd -s3-bucket my-test-bucket.com
```

这里的AWS的配置其实可以放在对应系统的环境变量中，这样就不需要每次启用都需要配置一次了。最后一行命令其实就是对应的aws s3中你需要存放文件的储存桶(bucket)。

当然，有时这个server不完全作为一个独立的应用，而是把功能嵌入到其他的server中去，那么自然官方提供给我们的这个现成的程序就不实用了，所以下面我们还是重点分析下这个包里的主要功能和提供给我们的接口。

二、Tusd的基本嵌入使用

很多时候，我们并不需要把Tus协议的上传服务当做一个单独的程序，我们需要让它成为我们某个server的一部分，那么自然，我们就需要把这些程序嵌入到某些代码中中去，这里我们就给一个实例来说明。此处我们没有采用其他的golang的网络编程框架，都是golang自带的包。代码如下：

```
package main

import (
        "fmt"
        "net/http"

        "github.com/tus/tusd"
        "github.com/tus/tusd/filestore"
)

func main() {
        store := filestore.FileStore{
                Path: "./uploads",
        }

        composer := tusd.NewStoreComposer()
        store.UseIn(composer)

        handler, err := tusd.NewHandler(tusd.Config{
                BasePath:      "/files/",
                StoreComposer: composer,
        })
        if err != nil {
                panic(fmt.Errorf("Unable to create handler: %s", err))
        }

        http.Handle("/files/", http.StripPrefix("/files/", handler))
        err = http.ListenAndServe(":8080", nil)
        if err != nil {
                panic(fmt.Errorf("Unable to listen: %s", err))
        }
}
```

我们来逐行分析代码。　　首先，store这个变量的定义是定义了储存的位置，这个样例给的是储存于本地，那么我们所需要引入的包里的结构体类型就是filestore.FileStore，Path后面填写的就是我们定义的储存目录。然后composer其实就是一个很多方法和函数的组织，用来处理上传、锁定、终止等等功能的一个量。所以新生成了一个composer之后，我们需要让我们的store使用这些方法或者函数。

`tusd.NewHandler(...)`这个函数其实是返回一个http.Handler处理函数，里面填入的是`tusd.Config{...}`这个结构体。这个结构体的配置了很多东西，基本的也就是`BasePath`和`StoreComposer`，这两个尽量按照默认的来填写。

如果要把文件储存到aws s3上面，以上的代码是需要修改一下的，其实就是修改store这个量的定义，代码如下：

```
bucket := "mx-search-corpus"
subPath := "uploads"

sess := session.Must(session.NewSession())
s3Config := aws.NewConfig()
```

```
store := s3store.New(bucket, s3.New(sess, s3Config))
store.ObjectPrefix = subPath
```

这里我定义了一个subPath，是给store.ObjectPrefix的，这个默认是空，也就是直接储存于bucket中，如果想要在bucket中建立一个目录，那么就给这个量赋值，比如我这里给的是 uploads，那么储存的路径就是 mx-search-corpus/uploads/ 。

三、 对于Tusd源码的一些理解

如果需要详细的了解这个包，自然是参考官方的doc：https://godoc.org/github.com/tus/tusd 。这里涉及到的type和对应的函数总数也并不是很多，可以好好看一番。这里我只针对一些核心的东西来谈一下理解。

1、S3Store

这里我们就不讲解FileStore也就是本地储存了，因为在我们使用时更多是储存在aws s3上面，而且本地储存的结构相对也简单得多。官方doc是 https://godoc.org/github.com/tus/tusd/s3store#S3Store 。

先看下定义：

```
type S3Store struct {
        // Bucket used to store the data in, e.g. "tusdstore.example.com"
        Bucket string
        // ObjectPrefix is prepended to the name of each S3 object that is
created.
        // It can be used to create a pseudo-directory structure in the
bucket,
        // e.g. "path/to/my/uploads".
        ObjectPrefix string
        // Service specifies an interface used to communicate with the S3
backend.
        // Usually, this is an instance of github.com/aws/aws-sdk-
go/service/s3.S3
        // (http://docs.aws.amazon.com/sdk-for-go/api/service/s3/S3.html).
        Service S3API
        // MaxPartSize specifies the maximum size of a single part
uploaded to S3
        // in bytes. This value must be bigger than MinPartSize! In order
to
        // choose the correct number, two things have to be kept in mind:
        //
        // If this value is too big and uploading the part to S3 is
interrupted
        // expectedly, the entire part is discarded and the end user is
required
        // to resume the upload and re-upload the entire big part. In
addition, the
        // entire part must be written to disk before submitting to S3.
        //
        // If this value is too low, a lot of requests to S3 may be made,
depending
        // on how fast data is coming in. This may result in an eventual
```

```
overhead.
        MaxPartSize int64
        // MinPartSize specifies the minimum size of a single part
uploaded to S3
        // in bytes. This number needs to match with the underlying S3
backend or else
        // uploaded parts will be reject. AWS S3, for example, uses 5MB
for this value.
        MinPartSize int64
        // MaxMultipartParts is the maximum number of parts an S3
multipart upload is
        // allowed to have according to AWS S3 API specifications.
        // See: http://docs.aws.amazon.com/AmazonS3/latest/dev/qfacts.html
        MaxMultipartParts int64
        // MaxObjectSize is the maximum size an S3 Object can have
according to S3
        // API specifications. See link above.
        MaxObjectSize int64
}
```

这里bucket就是我们需要储存的aws s3储存桶的名字，ObjectPrefix可以理解为就是bucket下面的路径名称。其实这里还定义了 S3API这一组接口，Service其实就是接收实现这一组接口的一个类。我们上面给的例子里这个Service是通过 New()函数提供的，我们让Service接收的是s3.New(session.Must(session.NewSession()), s3Config)。

由于Tusd到AWS S3的上传都是分块（part）上传的，所以MaxPartSize和MinPartSize就是对于每个part大小的限制，这里其实我们可以设置一下，也可以不设置采用默认的。MaxMultipartParts就是允许单个文件最多分成多少个part，MaxObjectSize就是单个文件允许上传的最大值。

S3Store的成员函数中，我们直接调用的就是

```
func (store S3Store) UseIn(composer *tusd.StoreComposer)
```

其实在之前我们必须生成一个composer，后面介绍。其实我们可以这样理解，我们新建了一个composer的类，里面实现了让S3Store实现了DataStore（后面说）里几个接口的方法，通过这个composer我们可以知道使用了哪些接口方法，没有使用哪些接口方法。因为有S3Store、FileStore等不同的Store，但是composer就是一个。

实际上传到AWS S3的所有实现都在这个类的下面，对于每一个part的上传，我们其实只用到了四个函数。

- 这个是在第一次上传时通过POST请求才会调用，这个函数建立了Tusd到AWS S3的上传连接，并且在里面生成了一个id，这个id后面会在POST请求函数里拼接成一个url，通过Response Headers返回给客户端

```
func (store S3Store) NewUpload(info tusd.FileInfo) (id string, err error)
```

- 因为每一次上传，不管是否第一次，都需要HEAD请求一下，获取offset等信息，这个函数的目的主要就是通过info来获取实际上已经上传的offset。

```
func (store S3Store) GetInfo(id string) (info tusd.FileInfo, err error)
```

这个函数中

```
func (store S3Store) GetInfo(id string) (info tusd.FileInfo, err error) {

        ...

        offset := int64(0)
        i := 0
        for _, part := range parts {
                i += 1
                offset += *part.Size
        }
        info.Offset = offset

        return
}
```

这段代码就是获取offset的部分，可以看出，parts其实是通过aws s3 api获取到的已经存在于s3上的part的所有，然后把每个part的大小加起来，赋值给offset并且返回。所以通过这个地方可以看出，offset对每一个url的上传来说，都是从0开始一直到文件大小结束的。所以，每一个url的上传请求，肯定是顺序的。

- 这个就是根据http的请求来上传的函数，里面实现了把文件的chunk从客户端读取到Tusd然后再上传到AWS S3的逻辑。

```
func (store S3Store) WriteChunk(id string, offset int64, src io.Reader)
(int64, error)
```

我们看下函数内部的一些代码：

```
for {
                // Create a temporary file to store the part in it
                file, err := ioutil.TempFile("", "tusd-s3-tmp-")
                if err != nil {
                        return bytesUploaded, err
                }
                defer os.Remove(file.Name())
                defer file.Close()

                limitedReader := io.LimitReader(src, optimalPartSize)
                n, err := io.Copy(file, limitedReader)
                // io.Copy does not return io.EOF, so we not have to
handle it differently.
                if err != nil {
                        return bytesUploaded, err
```

```
                }
                // If io.Copy is finished reading, it will always return
(0, nil).
                if n == 0 {
                        return bytesUploaded, nil
                }
                ...
}
```

这部分代码在一个for循环中，可以看到一开始就在Tusd Server的机器上创立了一个临时的文件（根据其他地方代码判断，这个文件也就是一个chunk最大不会超过int64的上限大小，代码里没有限制临时文件的个数，所以应该是根据硬盘大小来设定），这就是上传到Tusd Server的文件，然后Tusd Server把这个文件上传到S3，成功后删除。 注意，这里上传是针对每一个chunk，这是客户端划分出来的。我们上传到Tusd Server之后，Tusd执行UploadPart把这个chunk传到AWS S3中去。可以看到这里用了 defer os.Remove(file.Name())表明每个file的在for循环内的上传执行完毕之后，我们就删除了这个文件，当然如果没有上传完，这个文件也会删除。

- 这个就是每次执行完一个upload请求后，来判断一下是否完成了上传。判断的其实是这一部分是否传到了AWS S3上，而不仅仅是在Tusd Server上。

```
func (store S3Store) FinishUpload(id string) error
```

四、StoreComposer和DataStore

前面提到的composer其实在包里对应指的就是StoreComposer这个结构。起定义如下：

```
type StoreComposer struct {
    Core DataStore

    UsesTerminater     bool
    Terminater         TerminaterDataStore
    UsesFinisher       bool
    Finisher           FinisherDataStore
    UsesLocker         bool
    Locker             LockerDataStore
    UsesGetReader      bool
    GetReader          GetReaderDataStore
    UsesConcater       bool
    Concater           ConcaterDataStore
    UsesLengthDeferrer bool
    LengthDeferrer     LengthDeferrerDataStore
}
```

这里的 DataStore 其实是一组接口：

```
type DataStore interface {
        NewUpload(info FileInfo) (id string, err error)
```

```
        WriteChunk(id string, offset int64, src io.Reader) (int64, error)
        GetInfo(id string) (FileInfo, error)
}
```

接口定义了建立上传(NewUpload)，写入chunk(WriteChunk)和获取已上传信息(GetInfo)三个函数，在S3Store里其实已经对这个三个接口进行了函数定义，然后通过composer里的UseCore来进行了接口的实现。其实S3Store里的UseIn函数，使得S3Store既实现了DataStore接口，也实现了DataStore.go文件里其他几个接口：

- TerminaterDataStore，接口里的Terminate方法就是中止文件上传的，会删除已经上传到服务器也就是aws s3里的缓存（parts），这个接口也是HTTP Delete方法所需要的接口。

```
type TerminaterDataStore interface {
        Terminate(id string) error
}
```

- FinisherDataStore，接口里的FinishUpload函数就是正常结束上传所调用的函数，自然是上传量offset等于文件大小之后执行该函数。执行后，aws s3会把各个上传好的part合并为一个文件储存在之前设定好的bucket/prefix这样的路径下面。

```
type FinisherDataStore interface {
        FinishUpload(id string) error
}
```

- GetReaderDataStore，这个接口里的GetReader函数并不是官方tusd的要求，算第三方的。如果下载文件需要，这个接口还是需要实现的。

```
type GetReaderDataStore interface {
        GetReader(id string) (io.Reader, error)
}
```

- ConcaterDataStore，这个接口就是处理各个上传完的part，当最后一个part上传结束，需要执行以便连接各个part所用。

```
type ConcaterDataStore interface {
        ConcatUploads(destination string, partialUploads []string) error
}
```

其实还有其他几种DataStore，这些可以根据需要使用，tusd里关于AWS S3储存的实现只调用了这几个。StoreComposer和DataStore更多来说只是告诉我们用了哪些接口以及哪些接口可以使用，真正的实现其实是在S3Store的定义文件中。

五、Config配置文件

在介绍handler处理之前，先得介绍一下Config这个结构。定义如下：

```
type Config struct {
        // 就是之前介绍的DataS
        DataStore DataStore
        // 也是之前介绍的StoreComposer
        StoreComposer *StoreComposer
        // 单个upload允许的最大上传量，不设定的话，默认 math.MaxInt64 大小
        MaxSize int64
        // Base的url，基本路径
        BasePath string
   // 是否绝对路径
        isAbs    bool
        // 是否使用 完成上传 chan (golang里的channel)
        NotifyCompleteUploads bool
        // 是否使用 结束上传 chan
   NotifyTerminatedUploads bool
        // 是否使用 上传进行 chan
        NotifyUploadProgress bool
        // 是否使用 创建上传 chan
        NotifyCreatedUploads bool
        // Log的定义
        Logger *log.Logger
        RespectForwardedHeaders bool
}
```

其实这里我们一般的定义也就是BasePath和StoreComposer，这两个就已经足够了，DataStore其实在StoreComposer里已经包含了，除非StoreComposer我们没有定义，但是推荐还是按照BasePath和StoreComposer来定义。

       文件里还有一个validate()函数，就是来判断必要的参数是否已经传入的，已经对于BasePath进行一下格式化。

六、Handler文件

Tusd里用了两个文件实现了Handler，一个handler.go，一个unrouted_handler.go。首先UnroutedHandler定义如下：

```
 type UnroutedHandler struct {
        config        Config
        composer      *StoreComposer
        isBasePathAbs bool
        basePath      string
        logger        *log.Logger
        extensions    string
```

```
        CompleteUploads chan FileInfo
        TerminatedUploads chan FileInfo
        UploadProgress chan FileInfo
        CreatedUploads chan FileInfo
        Metrics Metrics
 }
```

其实这里的定义并不复杂，Handler这个结构继承了UnroutedHandler，并且传入了HTTP的方法。

```
type Handler struct {
        *UnroutedHandler
        http.Handler
}
```

其实是在UnroutedHandler里定义了函数，Handler里进行了HTTP接口的定义。UnroutedHandler里主要的方法：

- 创建一个新的upload请求，并且返回一个url。

```
 func (handler *UnroutedHandler) PostFile(w http.ResponseWriter, r
*http.Request)
```

- 获得已经上传完的offset偏移量，每一次upload都得执行的一个请求。

```
func (handler *UnroutedHandler) HeadFile(w http.ResponseWriter, r
*http.Request)
```

- 上传每一个chunk，每次upload必调用的。

```
func (handler *UnroutedHandler) PatchFile(w http.ResponseWriter, r
*http.Request)
```

其余删除下载等就不赘述了。其实两个Handler文件里的内容主要就是面对客户端调用了，也只是一个写了具体的函数，一个是通过http接口调用了这些函数。这些函数里调用的方法，其实就是S3Store里定义的那些。