STEP3

STEP2中我们完成了架构的分层，可以把一个对象随机分配到一个储存节点进行存储。但问题来了，如果我们重复上传10遍同一个对象，那我们就可能把这一个东西重复存在10个不同的节点里，有必要吗?

为了在上传的时候判断这个东西我存没存过，我们需要对其哈希值进行判断。

同时我们可以给对象添加一个版本号，不但可以用来标记这是上传的第几个版本，还可以用来设置为“已删除”标志位，这样如果用户不小心删除掉了这个东西我们也可以帮用户找回来。

有了版本号，删除就变成了假删除，名义上GET不到了，但节点里还有。

那我们需要一个东西存哈希值和具体内容的一一对应关系，就是ES

我们在ES中创建一个名叫metadata的索引，里面有一个名为object的类型

最终ES会向get函数返回一组数据如下

```
{
    Name:string
    Version:int,
    Size:int,
    Hash:String
}
```

包括了这个对象的名字，版本，大小和他的哈希值

```
func Handler(w http.ResponseWriter, r *http.Request) {
    m := r.Method
    if m != http.MethodGet {
        w.WriteHeader(http.StatusMethodNotAllowed)
        return
    }
    from := 0
    size := 1000
    name := strings.Split(r.URL.EscapedPath(), sep: "/")[2]
    for {
        metas, e := es.SearchAllVersions(name, from, size)
        if e != nil {
            log.Println(e)
            w.WriteHeader(http.StatusInternalServerError)
            return
        }
        for i := range metas {
            b, _ := json.Marshal(metas[i])
            w.Write(b)
            w.Write([]byte("\n"))
        }
        if len(metas) != size : ⤴
        from += size
    }
}
```

version包里的handler函数很简单，主要功能由SearchAllVersion完成

```
func SearchAllVersions(name string, from, size int) ([]Metadata, error) {
    url := fmt.Sprintf("http://#{os.Getenv("ES_SERVER")}/metadata/_search?sort=name,version&from=#{from}&size=#{size}")
    if name != "" {
        url += "&q=name:" + name
    }
    r, e := http.Get(url)
    if e != nil : nil, e ⤴
    metas := make([]Metadata, 0)
    result, _ := ioutil.ReadAll(r.Body)
    var sr searchResult
    json.Unmarshal(result, &sr)
    for i := range sr.Hits.Hits {
        metas = append(metas, sr.Hits.Hits[i].Source)
    }
    return metas, nil
}
```

在增加了ES以后，对象的删除方法有一些不同了

```go
func del(w http.ResponseWriter, r *http.Request) {
    name := strings.Split(r.URL.EscapedPath(), sep: "/")[2]
    version, e := es.SearchLatestVersion(name)
    if e != nil {
        log.Println(e)
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
    e = es.PutMetadata(name, version.Version+1, 0, "")
    if e != nil {
        log.Println(e)
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
}
```

可以看出主要是由两个函数进行具体操作

首先是在ES中找出最新的对象版本，然后将这个对象的其版本号加一，对象大小size设置为0，哈希值改为空字符串以表示这个对象被删除了

object包里的put函数也有变化但同样很好理解

```go
func put(w http.ResponseWriter, r *http.Request) {
    hash := utils.GetHashFromHeader(r.Header)
    if hash == "" {
        log.Println( v...: "missing object hash in digest header")
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    size := utils.GetSizeFromHeader(r.Header)
    c, e := storeObject(r.Body, hash, size)
    if e != nil {
        log.Println(e)
        w.WriteHeader(c)
        return
    }
    if c != http.StatusOK {
        w.WriteHeader(c)
        return
    }

    name := strings.Split(r.URL.EscapedPath(), sep: "/")[2]
    e = es.AddVersion(name, hash, size)
    if e != nil {
        log.Println(e)
        w.WriteHeader(http.StatusInternalServerError)
    }
}
```

就是分别拿出哈希值，对象的大小，和名字，然后

首先判断哈希值是不是空的，要是空的那报错，这对象有问题，

不然用storeObject函数将对象内容、哈希值和size写入。

get这边逻辑上也很清晰，就是按照name和版本号去ES里找

```go
func get(w http.ResponseWriter, r *http.Request) {
    name := strings.Split(r.URL.EscapedPath(), sep: "/")[2]
    versionId := r.URL.Query()["version"]
    version := 0
    var e error
    if len(versionId) != 0 {
        version, e = strconv.Atoi(versionId[0])
        if e != nil {
            log.Println(e)
            w.WriteHeader(http.StatusBadRequest)
            return
        }
    }
    meta, e := es.GetMetadata(name, version)
    if e != nil {
        log.Println(e)
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
    if meta.Hash == "" {
        w.WriteHeader(http.StatusNotFound)
        return
    }
    object := url.PathEscape(meta.Hash)
    stream, e := getStream(object)
    if e != nil {
        log.Println(e)
        w.WriteHeader(http.StatusNotFound)
        return
    }
    io.Copy(w, stream)
}
```

我们刚才说到删除对象的时候哈希值要设置成空的，所以如果找出的哈希值是空的，即使能在ES里查到，那也是被删除过了，逻辑上不应该给用户再查出来，所以返回“没有找到”

STEP4

上一步我们实现了数据的版本记录功能，并没有完全避免重复数据的上传所以这一步我们实现数据的校验功能

要实现的逻辑是：

当put一个对象时，我们在接口层首先进行一次定位，如果对象已经存在，那么直接跳过数据层的操作，在ES里存进去一个新版本的对象，数据层不变。并且要在上传后判断数据是否出现了错误，是否完整。

可以看出这一步的变化发生在objects接口中，删除了原来的PUT方法，新增了temp接口，其中有PUT、POST、PATCH、DELETE等操作，为的是增加一层缓存。

这层缓存解决的问题是，我们为了检验这个对象是否已经存在，需要把它放在内存里，但是内存的大小也许不够用，所以我们需要暂时将其存在硬盘上，然后检验其哈希值判断对象是否已经存在，如果存在那么立刻将其删除，如果不存在说明是新的对象，将其转正保留下来。

整个逻辑过程如下：客户端在PUT的时候需要提供其哈希值和size。接口层首先通过哈希值判断是否存在，如果存在，直接把版本加一存进ES，如果不存在那么用POST方法访问temp接口，提前告知服务器其哈希值和size。然后通过PATCH方法，将数据上传到数据接口。在传输完成后计算对象的哈希值，看是否与之前POST方法告知的哈希值和size一致，如果一致说明其上传是完整无误。这个时候说明这个对象是新的，并且没错的，调用PUT方法将其转正存储在节点中。否则有一点错误都要用DELETE方法删除掉。

STEP5

剩下的利用RS纠删码实现数据冗余去除和断点续传功能我们在面试问答里面细说，因为这两个属于业务功能的实现，并不涉及到架构的设计，架构上没有变化，只是上传下载的时候多加几个步骤。