一、Tus协议和C/S端实现

Tus是一个基于HTTP协议(HTTP-based)的开源的断点续传协议（源码地址：https://github.com/tus/tusd; 官网地址：https://tus.io），主要目的当然是保证文件上传中断后依然可以续传。Tus可以很容易地嵌入使用库、代理和防火墙的程序，也可以直接在任何网站使用。Tus已经可以用于生产实践中，它经过了很多轮的改进，也接受了来自Vimeo，Google以及其他知名企业的使用者的有益的反馈而进行优化。

基于Tus的协议自然是要求不管是server还是client，都需要满足这个协议的要求，才可以确保文件的断点传输。

1. Tus协议的基本上传流程

首先，Tus是基于HTTP的一个协议，自然所有的请求都是基于url来进行的，总共需要两个url。第一个url的作用是生成一个对应匹配上传源的独有的url，第二个url就是第一个url生成的url（其实就是第一个url加上生成了一个独有的id），用作获取已上传量(`upload-offset`)和上传作用的。续传就是从已上传量的下一个字节开始。自然，最开始的已上传量就是0，也就是从头开始。

对于服务端，需要实现的接口下一节具体介绍；针对于客户端，Tus的客户端（我这里参考的是python写的client的库，地址：https://github.com/tus/tus-py-client） 样例其实提供了两种选择：

- 1. 不采取断点续传
- 2. 采取断点续传

其实两者的差异就在于初始化时，两个参数`store_url`和`url_storage`的填充。如果两个参数都填写，并且正确，那么续传功能才可以实现，我们这里只讨论采取续传的情况。

其实每一次执行上传最多就是三个步骤：

- 1. 最开始是进行判断，也就是该path下的该上传源是否被上传过，或者至少说是否向服务器提供过上传信息。通过上传源的path其实会生成一个独立唯一的md5，也就是任何时候执行生成函数，这个md5的key都是唯一不变的。然后在本地的db（url_storage接收的是一个类，这个类里定义了url的储存地方，可以是本地db，也可以远程数据库，满足关系只要是md5:url这种键值对的json结构即可，具体可以参考client里的FileStore的默认样例）中通过这个key查询是否存在对应的url，如果key不存在，则POST请求服务器，通过服务器那边生成一个url（本质是生存了一个id拼接之前的url，只不过server那边给我们拼好了）并返回，然后把这个url储存到这个db中去。只要续传，每一次都可以在这个数据中查询到对应的url，然后进行续传。

  IMPORTANT：在Production环境中，储存url的数据库不能轻易删除，否则远程服务器上已经有上传数据，但是client会判断从未上传过，请求生成新的url有时会冲突。

- 2. 然后就是HEAD请求，该请求是每一次必执行的，对于client来说，主要是获取`offset`用的，也就是已上传量，因为下一次上传是从`offset+1`开始的。
- 3. 再然后就是最后一步上传，这是一个PATCH请求，在client端可以设置chunk_size，也就是每一次上传的量是多少，默认的chunk_size不同语言写的client不一定一致，一般都是int型类型的最大值。每个chunk只有两种结果，成功或者不成功，也就是除了最后一个chunk的大小可能小于chunk_size之外，之前的offset的值一定是chunk_size的整数倍。python的client库里的上传并不是并发执行的，是按照先后顺序一个chunk一个chunk顺序上传的。

在pyhthon写的client端的流程大致如此。默认的client库里传输出现错误或者异常什么的，是有重连次数设定的（默认0次，失败就抛出异常），所以如果需要不断尝试重连，可以重写一些函数或者修改源码。其他语言的

client端大致思路也是如此，具体实现或许会有差异。

另外，默认的python语言实现的client库中，每个chunk是顺序上传的，这个根Tusd这个server的实现应该有关。我看了Tusd的源码，每个offset都是计算的已上传的字节总和，因此只能顺序传输。但是从实现角度来说，我觉得可以改成并发实现增加上传速度。不过这个需要修改C/S两端的源码，其他语言的server和client对于chunk的实现我并不是很清楚。Tus协议并没有阻止并发的实现，可以参考：https://tus.io/blog/2018/09/25/adoption.html 。

2. Tusd server的基本接口

我们这里介绍的Tusd是Tus协议server的一个实现，它是由go语言实现的。这个server提供以下五个接口：

- `POST "/"`， 这是最开始的接口，在返回的Headers里的`Location`参数有一个后面需要的独有的url。
- `HEAD "/:id"`，其实这里的"/:id"就是获取的url，我们在POST得到的就是整个url，这是HEAD请求获取offset用的。
- `PATCH "/:id"`，这就是传输文件的接口，直到某个具体的chunk上传完毕，patch接口才会关闭。
- `GET "/:id"`，顾名思义，这是读取文件或者就是下载作用。
- `DELETE "/:id"`，这就是删除文件的操作。

对于接口的调用，在client部分已经有提及，具体实现可以参考所用语言对应的client库。

3. Tus里涉及的Headers

其实有许多的headers是必备的，由于官网给的server和client都有demo，而且很多逻辑没有让我们自己实现，所以想清楚的了解请求参数这些的，最好去读源码了解一下。我这边做个简单的介绍，主要的Headers有七个（包括req和res的Header）：

- `Upload-Offset`：Request和Response都存在的Header。已上传量，初始时是0，HEAD请求后的response里会在Header里给出新的upload-offset，然后PATCH请求时需要填入新的值
- `Upload-Length`：Reques和Response里都有的值，这个值必须非负，是判断上传是否完成的不可缺少的值。
- `Tus-Version`：Response的Header里需要的，按照次数以逗号隔开，填写Tus协议的版本，第一个是优先版本，次序按照优先级降低
- `Tus-Resumable`：除了可选择的Request，所有的Reques和Response这个Header是必须的。这个必须保持client和server一致使用的Tus协议版本。
- `Tus-Extension`：这个是Response里可选的Header，如果server里有使用插件什么的，逗号隔开填入插件；如果没有使用，这个Header就必须删掉，不可以为空。
- `Tus-Max-Size`：Response的Header里必须为一个非负整数，这个是告诉client端该server每个upload请求允许单一完整的上传的大小。
- `X-HTTP-Method-Override`：Request里的Header，是可以选择的。也就是HTTP方法重写，对于不支持的浏览器什么的重写方法，这里不赘述。这个得根据实际需要选择。

二、对于保存至AWS S3的一些认识

由golang语言实现的Tusd的server提供了数据保存至aws s3的支持。Server和s3之间的传输我们并不需要了解那么多，具体是采用的aws对于golang s3分段上传的库，这也是支持断点续传的，我们重点需要的是理解client和server的通信。凡是储存到s3的文件，必须得在s3上生成文件成功后才会结束上传。

上传到s3，其实是在Tusd的server里，把上传好的部分储存到了一个系统内的临时文件夹，然后Tusd的server自己在做分块上传，每一块也只是成功或者失败。传完一个part，server这边对应的就是删除掉临时文件。直到整个文件的每一个part都储存到s3后，server这边执行一个结束上传文件，s3那边才会生成文件到s3的桶内让我们看到。

由于储存在s3中的未完成整个上传的文件part是储存在我们看不到的地方的，我们也无法对其进行读取等操作，所以对于那些一直没有能上传完的文件，为了空间和时间的成本考虑，是需要对s3的桶进行一些设置的，也就是设置其生命周期。比如多少天内没有上传完毕的文件，我们需要删除，具体方法请参阅 https://docs.aws.amazon.com/zh_cn/AmazonS3/latest/dev/object-lifecycle-mgmt.html

三、参考文档

- Tus官方协议: https://tus.io/protocols/resumable-upload.html
- Tus官方demo: https://tus.io/demo.html
- Tus官方或社区的C/S端的一些实现: https://tus.io/implementations.html