Article
Iryna Mykhailova · Juil 1, 2022 10m de lecture

Open Exchange

Transmission de fichiers via REST pour les stocker en Property, partie 3

Dans le premier article de cette série, nous avons vu comment lire un "gros" volume de données dans le corps brut d'une méthode HTTP POST et l'enregistrer dans une base de données en tant que propriété de flux d'une classe. Le deuxième article explique comment enregistrer des fichiers et leurs noms dans un format JSON.

Examinons maintenant de plus près l'idée d'envoyer des fichiers volumineux par parties au niveau du serveur. Il existe plusieurs approches que nous pouvons utiliser pour y parvenir. Cet article traite de l'utilisation de l'en-tête Transfer-Encoding pour indiquer un transfert par blocs. La spécification HTTP/1.1 a introduit l'en-tête Transfer-Encoding, et RFC 7230, section 4.1 l'a décrit, mais il n'est pas mentionné dans la spécification HTTP/2.

Transfer-Encoding

L'objectif de l'en-tête Transfer-Encoding est de spécifier la forme d'encodage utilisée pour transférer le corps de la charge utile à l'utilisateur en toute sécurité. Vous utilisez cet en-tête principalement pour délimiter avec précision une charge utile générée dynamiquement et pour distinguer les codages de charge utile pour l'efficacité du transport ou la sécurité des caractéristiques de la ressource sélectionnée.

Vous pouvez utiliser les valeurs suivantes dans cet en-tête :

- Chunked (en blocs)
- Compress (compresser)
- Deflate (dégonfler)
- gzip

Le codage de transfert est égal à l'encodage par blocs

Lorsque vous définissez le codage de transfert par blocs, le corps du message est constitué d'un nombre non spécifié de blocs réguliers, d'un bloc de fin, d'un trailer et du caractère retour chariot (CRLF).

Chaque partie commence par une dimension de bloc représentée par un nombre hexadécimal, suivie d'une extension facultative et d'un CRLF. Ensuite vient le corps du bloc avec CRLF à la fin. Les extensions contiennent les métadonnées du bloc. Par exemple, les métadonnées peuvent inclure une signature, un hachage, des informations de contrôle à mi-message, etc. Le bloc de fin de message est un bloc régulier de longueur nulle. Un trailer, qui consiste en des champs d'en-tête (éventuellement vides), suit le block de fin.

Pour rendre tout cela plus facile à imaginer, voici la structure d'un message avec Transfer-Encoding = par blocs :

chunked_body	*chunk last_chunk trailer_part CRLF
chunk	chunk_size [chunk_ext] CRLF chunk_data CRLF
chunk_size	*size-of-current-chunk-in-HEX*
chunk_ext	*(";" chunk_ext_name ["=" chunk_ext_val])
chunk_ext_name	*token*
chunk_ext_val	*token / quoted-string*
chunk_data	*contents-of-current-chunk*
last_chunk	1*("0") [chunk_ext] CRLF
trailer_part	*(header_field CRLF)

Voici un exemple de message court et découpé en blocs :

```
13\r\n
Transferring Files \r\n
4\r\n
on\r\n
1A\r\n
community.intersystems.com
0\r\n
\r\n
```

Ce corps de message se compose de trois blocs significatifs. Le premier bloc a une longueur de dix-neuf octets, le deuxième en a quatre et le troisième en a vingt-six. Vous pouvez constater que les CRLF de fin qui marquent la fin des blocs ne sont pas pris en compte dans la taille du bloc. Mais si vous utilisez le CRLF comme marqueur de fin de ligne (EOL), alors le CRLF est compté comme une partie du message et prend deux octets. Le message décodé ressemble à ceci :

```
Transferring Files on
community.intersystems.com
```

Formation de messages groupés dans IRIS

Pour ce tutoriel, nous allons utiliser la méthode sur le serveur créé dans le premier article. Cela signifie que nous allons envoyer le contenu du fichier directement dans le corps de la méthode POST. Comme nous envoyons le contenu du fichier dans le corps, nous envoyons le POST à http://webserver/RestTransfer/file.

Maintenant, voyons comment nous pouvons former un message en bloc dans IRIS. Comme indiqué dans Envoi de requêtes HTTP, à la section Envoi d'une requête par blocs, vous pouvez envoyer une requête HTTP par blocs si vous utilisez HTTP/1.1. La meilleure partie de ce processus est que %Net.HttpRequest calcule automatiquement la longueur du contenu de l'ensemble du corps du message côté serveur, de sorte qu'il n'est pas nécessaire de modifier le côté serveur du tout. Par conséquent, pour envoyer une requête en bloc, vous devez suivre ces étapes dans le client uniquement.

La première étape consiste à créer une sous-classe de %Net.ChunkedWriter et à implémenter la méthode

OutputStream. Cette méthode doit recevoir un flux de données, l'examiner, décider s'il faut le diviser en parties ou non, et comment le diviser, et invoquer les méthodes héritées de la classe pour écrire la sortie. Dans notre cas, nous appellerons la classe RestTransfer.ChunkedWriter.

Ensuite, dans la méthode côté client responsable de l'envoi des données (appelée ici "SendFileChunked"), vous devez créer une instance de la classe RestTransfer.ChunkedWriter et la remplir avec les données demandées que vous souhaitez envoyer. Comme nous envoyons des fichiers, nous ferons tout le travail dans la classe RestTransfer.ChunkedWriter. Nous ajoutons une propriété nommée Filename As %String et un paramètre nommé " MAXSIZEOFCHUNK = 10000. " Bien sûr, vous pouvez décider de définir une dimension maximale autorisée pour le bloc en tant que propriété et la définir pour chaque fichier ou message.

Enfin, définissez la propriété EntityBody de %Net.HttpRequest comme étant égale à l'instance créée de la classe RestTransfer.ChunkedWriter.

Ces étapes correspondent simplement au nouveau code que vous devez écrire et remplacer dans votre méthode existante qui envoie des fichiers à un serveur.

La méthode ressemble à ceci :

```
ClassMethod SendFileChunked(aFileName) As %Status
{
  Set sc = $$$OK
  Set request = ..GetLink()
  set cw = ##class(RestTransfer.ChunkedWriter).%New()
  set cw.Filename = aFileName
  set request.EntityBody = cw
  set sc = request.Post("/RestTransfer/file")
  Quit:$System.Status.IsError(sc) sc
  Set response=request.HttpResponse
  do response.OutputToDevice()
  Quit sc
}
```

La classe %Net.ChunkedWriter est une classe de flux abstraite qui fournit une interface et possède quelques méthodes et propriétés implémentées. Ici, nous utilisons les propriétés et méthodes suivantes :

- La propriété TranslateTable en tant que %String force la traduction automatique des blocs lors de leur écriture dans le flux de sortie (EntityBody). Nous nous attendons à recevoir des données brutes, nous devons donc définir TranslateTable sur "RAW".
- La méthode OutputStream est une méthode abstraite surchargée par une sous-classe pour faire tout le découpage en blocs.

- La méthode WriteSingleChunk(buffer As %String) écrit l'en-tête HTTP Content-Length suivi du corps de l'entité en un seul bloc. Nous vérifions si la dimension du fichier est inférieure à la méthode MAXSIZEOFCHUNK, auquel cas, nous utilisons cette méthode.
- La méthode WriteFirstChunk(buffer As %String) écrit l'en-tête Transfer-Encoding suivi du premier bloc. Il doit toujours être présent. Il peut être suivi de zéro ou plusieurs appels pour écrire d'autres blocs, puis d'un appel obligatoire pour écrire le dernier bloc avec la chaîne vide. Nous vérifions que la longueur du fichier est supérieure à la méthode MAXSIZEOFCHUNK et appelons cette méthode.
- La méthode WriteChunk(buffer As %String) écrit des blocs conséquents. Vérifiez si le reste du fichier après le premier bloc est toujours supérieur à MAXSIZEOFCHUNK puis utilisez cette méthode pour envoyer les données. Nous continuons à le faire jusqu'à ce que la taille de la dernière partie du fichier soit inférieure à MAXSIZEOFCHUNK.
- La méthode WriteLastChunk(buffer As %String) écrit le dernier bloc suivi d'un bloc de longueur zéro pour marquer la fin des données.

Sur la base de tout ce qui précède, notre classe RestTransfer.ChunkedWriter est comme suit :

```
Class RestTransfer.ChunkedWriter Extends %Net.ChunkedWriter
{
  Parameter MAXSIZEOFCHUNK = 10000;
  Property Filename As %String;
  Method OutputStream()
  {
    set ..TranslateTable = "RAW"
    set cTime = $zdatetime($Now(), 8, 1)
    set fStream = ##class(%Stream.FileBinary).%New()
    set fStream.Filename = ..Filename
    set size = fStream.Size
    if size &lt; ..#MAXSIZEOFCHUNK {
      set buf = fStream.Read(.size, .st)
      if $$$ISERR(st)
      {
        THROW st
      } else {
        set ^log(cTime, ..Filename) = size
        do ..WriteSingleChunk(buf)
      }
    } else {
      set ^log(cTime, ..Filename, 0) = size
      set len = ..#MAXSIZEOFCHUNK
      set buf = fStream.Read(.len, .st)
      if $$$ISERR(st)
      {
        THROW st
      } else {
        set ^log(cTime, ..Filename, 1) = len
        do ..WriteFirstChunk(buf)
      }
      set i = 2
      While 'fStream.AtEnd {
        set len = ..#MAXSIZEOFCHUNK
        set temp = fStream.Read(.len, .sc)
    if len&lt;..#MAXSIZEOFCHUNK
    {
      do ..WriteLastChunk(temp)
    } else {
          do ..WriteChunk(temp)
        }
        set ^log(cTime, ..Filename, i) = len
        set i = $increment(i)
      }
    }
  }
}
```

Pour voir comment ces méthodes divisent le fichier en parties, nous ajoutons une globale ^log avec la structure suivante :

```
//for transfer in a single chunk
^log(time, filename) = size_of_the_file
//pour un transfert en plusieurs blocs
^log(time, filename, 0) = size_of_the_file
^log(time, filename, idx) = size_of_the_idx’s_chunk
```

Maintenant que la programmation est terminée, voyons comment ces trois approches fonctionnent pour différents fichiers. Nous écrivons une simple méthode de classe pour faire des appels au serveur :

```
ClassMethod Run()
{
  // D'abord, je supprime les globales.
  kill ^RestTransfer.FileDescD
  kill ^RestTransfer.FileDescS
  // Ensuite, je forme une liste des fichiers que je veux envoyer
  for filename = "D:\Downloads\wiresharkOutput.txt", // 856 bytes
     "D:\Downloads\wiresharkOutput.pdf", // 60 134 bytes
     "D:\Downloads\Wireshark-win64-3.4.7.exe", // 71 354 272 bytes
     "D:\Downloads\IRIS_Community-2021.1.0.215.0-win_x64.exe" //542 370 224 bytes
  {
    write !, !, filename, !, !
    // Et je lance les trois méthodes d'envoi de données au serveur.
    set resp1=##class(RestTransfer.Client).SendFileChunked(filename)
    if $$$ISERR(resp1) do $System.OBJ.DisplayError(resp1)
    set resp1=##class(RestTransfer.Client).SendFile(filename)
    if $$$ISERR(resp1) do $System.OBJ.DisplayError(resp1)
    set resp1=##class(RestTransfer.Client).SendFileDirect(filename)
    if $$$ISERR(resp1) do $System.OBJ.DisplayError(resp1)
  }
}
```

Après avoir exécuté la méthode de classe Run, dans la sortie des trois premiers fichiers, l'état était correct. Mais pour le dernier fichier, alors que les premier et dernier appels ont fonctionné, celui du milieu a renvoyé une erreur : 5922, Dépassement de délai en attente de réponse. Si nous examinons notre méthode des globales, nous voyons que le code n'a pas enregistré le onzième fichier. Cela signifie que ##class(RestTransfer.Client).SendFile(filename) a échoué - ou pour être précis, la méthode qui déballe les données de JSON n'a pas réussi.

Maintenant, si nous regardons nos flux, nous voyons que tous les fichiers enregistrés avec succès ont des dimensions correctes.

Si nous regardons la globale ^log, nous voyons combien de blocs le code a créé pour chaque fichier :

Vous aimeriez probablement voir le corps des messages réels. Eduard Lebedyuk a suggéré dans son article Debugging Web qu'il est possible d'utiliser CSP Gateway Logging and Tracing.

Si nous examinons le journal des événements pour le deuxième fichier découpé en blocs, nous constatons que la valeur de l'en-tête Transfer-Encoding est effectivement "découpé en blocs". Malheureusement, le serveur a déjà collé le message ensemble, donc nous ne voyons pas le découpage en blocks réel.

L'utilisation de la fonction Trace ne montre pas beaucoup plus d'informations, mais elle permet de clarifier la présence d'un écart entre l'avant-dernière et la dernière demande.

Pour voir les parties réelles des messages, nous copions le client sur un autre ordinateur pour utiliser un renifleur. Ici, nous avons choisi d'utiliser Wireshark car il est gratuit et possède les fonctions nécessaires. Pour mieux vous montrer comment le code divise le fichier en blocs, nous pouvons changer la valeur de MAXSIZEOFCHUNK à 100 et choisir d'envoyer un petit fichier. Ainsi, nous pouvons maintenant voir le résultat suivant :

Nous constatons que la longueur de tous les blocs, sauf les deux derniers, est égale à 64 en HEX (100 en DEC), que le dernier bloc contenant des données est égal à 21 DEC (15 en HEX) et que la dimension du dernier bloc est égale à zéro. Tout semble correct et conforme à la spécification. La longueur totale du fichier est égale à 421 (4x100+1x21), ce que nous pouvons également voir dans les globales :

Coonclusion

Dans l'ensemble, nous pouvons constater que cette approche fonctionne et permet d'envoyer sans problème de gros fichiers au serveur. En outre, si vous envoyez de grandes quantités de données à un client, vous pouvez vous familiariser avec Fonctionnement et configuration de la passerelle Web, section Paramètres de configuration du chemin d'application, paramètre Notification de la dimension de la réponse. Celui-ci spécifie le comportement de la passerelle Web lors de l'envoi de grandes quantités de données en fonction de la version de HTTP utilisée.

Le code de cette approche est ajouté à la version précédente de cet exemple sur GitHub et InterSystems Open Exchange.

À propos de l'envoi de fichiers en blocs, il est également possible d'utiliser l'en-tête Content-Range avec ou sans l'en-tête Transfer-Encoding pour indiquer quelle partie exacte des données est transférée. En outre, vous pouvez utiliser un tout nouveau concept de flux disponible avec la spécification HTTP/2.

Comme toujours, si vous avez des questions ou des suggestions, n'hésitez pas à les écrire dans la section des commentaires.

#REST API #InterSystems IRIS
Voir l'application sur InterSystems Open Exchange