Going Asynchronous using AsyncHttpClient: The Complex

January 4, 2011 Jeanfrancois Arcand Leave a comment Go to comments

The Async Http Client library purpose is to allow Java applications to easily execute HTTP requests and asynchronously process the HTTP responses. In this second part on the topic, I will describe more complex operations that can be done with the AsyncHttpClient like resumable download, zero in memory bytes copy, oAuth calculation, optimal transfer listener and performance tricks.

Resumable Donwload

The AsyncHttpClient supports resumable download in two differents scenarios:

- IOException: If an IOException occurs (for whatever reason), you can configure the library to restart the download automatically without having to restart the download from the beginning.
- JVM crashes: If your application or the JVM goes down during a file download, the library can also restart the download automatically when the same download is requested.

You can configure the AsyncHttpClient Library to survive IOException using the IOException Filter:

```
 AsyncHttpClient c = new AsyncHttpClient(
         new AsyncHttpClientConfig.Builder()
           .addIOExceptionFilter(
              new ResumableIOExceptionFilter()).build());

 ResumableAsyncHandler a =
     new ResumableAsyncHandler(
        new ResumableRandomAccessFileListener());

 a.setResumableListener(
      new ResumableRandomAccessFileListener(
          new RandomAccessFile( "file.avi", "rw" ) ) );

Response r = c.prepareGet("http://host:port/file.avi")
      .execute(a).get();
```

If you need something more high level and configurable, you can use a ResumableAsyncHandler, and or implement a ResumableProcessor:

```
AsyncHttpClient c = new AsyncHttpClient();
ResumableAsyncHandler a =
    new ResumableAsyncHandler( new PropertiesBasedResumableProcessor() );
a.setResumableListener(
    new ResumableRandomAccessFileListener(
       new RandomAccessFile( "file.avi", "rw" ) ) );

Response r = c.prepareGet( "http://localhost:8081/file.AVI" )
      .execute( a ).get();
```

You can also simply use a ResumableListener (or use the ResumableRandomAccessFileListener, which does what’ s described below):

```
public interface ResumableListener {

    public void onBytesReceived(ByteBuffer byteBuffer) throws IOException;

    public void onAllBytesReceived();

    public long length();
}
```

As simple as:

```
 AsyncHttpClient c = new AsyncHttpClient();
 final RandomAccessFile file = new RandomAccessFile( "file.avi", "rw" );
 ResumableAsyncHandler a = new ResumableAsyncHandler();
 a.setResumableListener( new ResumableListener() {

    public void onBytesReceived(ByteBuffer byteBuffer) throws IOException {
       file.seek( file.length() );
       file.write( byteBuffer.array() );
    }

    public void onAllBytesReceived() {
       file.close();
    }

    public long length() {
       return file.length();
    }
} );
Response r = c.prepareGet( "http://localhost:8081/file.AVI" )
        .execute( a ).get();
```

Make it simple: TransferListener

In some scenario an application may need to manipulate the received bytes in more than one place, e.g. saves the bytes on disk but also accumulate it for checksum checking later. In that case, instead of using an AsyncHandler and mixes logic inside an AsyncHandler.onBodyPartReceived, it is recommended to use the TransferListener simple API:

```
public interface TransferListener {

    public void onRequestHeadersSent
       (FluentCaseInsensitiveStringsMap headers);

    public void onResponseHeadersReceived
        (FluentCaseInsensitiveStringsMap headers);

    public void onBytesReceived(ByteBuffer buffer)
         throws IOException;

    public void onBytesSent(ByteBuffer buffer);

    public void onRequestResponseCompleted();

    public void onThrowable(Throwable t);
}
```

All you need to do in that case is to create a TransferCompletionHandler and add as many TransferListener as you need:

```
AsyncHttpClient client = new AsyncHttpClient();
TransferCompletionHandler tl = new TransferCompletionHandler();
tl.addTransferListener(new TransferListener(){...});

Response response = httpClient.prepareGet("http://...").execute(tl).get();
```

Zero Bytes Copy

When uploading or downloading bytes, it is important to try to avoid buffering bytes in memory.

Upload

On the upload side, the mechanism is enabled by default when setting the Request’ s body to a File:

```
AsyncHttpClient client = new AsyncHttpClient();

File file = new File("file.avi");
Future f = client.preparePut("http://localhost").setBody(file).execute();
```

If you can’ t use a File, the recommended way (as described in part I) is to use a BodyGenerator. It is strongly recommended to avoid using InputStream as the library will unfortunately buffer the entire content in memory in order to set the content-lenght, which can cause out of memory error.

Download

On the download side, you can use the HttpResponseBodyPart.writeTo to avoid loading bytes in memory and unnecessary copy:

```
 AsyncHttpClient client = new AsyncHttpClient();

 File tmp = new File("zeroCopy.txt");
 final FileOutputStream stream = new FileOutputStream(tmp);
 Future f = client.prepareGet("http://localhost/largefile.avi")
       .execute(new AsyncHandler() {
     public void onThrowable(Throwable t) {
     }

     public STATE onBodyPartReceived(HttpResponseBodyPart bodyPart)
          throws Exception {
        bodyPart.writeTo(stream);
        return STATE.CONTINUE;
     }

     { .... }
});
Response resp = f.get();
```

Limiting the number of connections to improve raw performance

By default the library uses a connection pool and re-use connections as needed. It is important to not let the connection pool grow too large as it takes resources in memory. One way consist of setting the maximum number of connection per host or in total:

```
AsyncHttpClientConfig config = new AsyncHttpClientConfig.Builder()
    .setMaximumConnectionsPerHost(10)
    .setMaximumConnectionsTotal(100)
    .build();
AsyncHttpClient c = new AsyncHttpClient(config);
```

There is no magic number, so you will need to try it and decide which one gives the best result.

Using OAuth

You can use the library to pull data from any OAuth site (like Twitter). This is as simple as:

```
private static final String CONSUMER_KEY = "dpf43f3p214k3103";
private static final String CONSUMER_SECRET = "kd94hf93k423kf44";
public static final String TOKEN_KEY = "nnch734d00sl2jdk";
public static final String TOKEN_SECRET = "pfkkdhi9sl3r4s00";
public static final String NONCE = "kllo9940pd9333jh";
final static long TIMESTAMP = 1191242096;

public void oAuth() {
  ConsumerKey consumer =
    new ConsumerKey(CONSUMER_KEY, CONSUMER_SECRET);
  RequestToken user =
    new RequestToken(TOKEN_KEY, TOKEN_SECRET);
  OAuthSignatureCalculator calc =
     new OAuthSignatureCalculator(consumer, user);
  AsyncHttpClient client = new AsyncHttpClient();

  Response response = client.prepareGet("http://...")
     .setSignatureCalculator(calc).execute().get();
```

What’s Next

In the next part of this series I will explain more complex operations that can be done with the AsyncHttpClient library like:

- Supporting the WebSocket protocol.
- Concurrent use of AsyncHandler

For any questions you can use our Google Group, irc.freenode.net #asynchttpclient or use Twitter to reach me! You can checkout the code on Github, download the jars from Maven Central or use Maven:

```
<dependency>
    <groupId>com.ning</groupId>
    <artifactId>async-http-client</artifactId>
    <version>1.4.1</version>
</dependency>
```

Related

Going Asynchronous using AsyncHttpClient: For DummiesIn "Async Http client"

Going Asynchronous using AsyncHttpClient: The BasicIn "Async Http client"

New Open Source Project Alert! A New Asynchronous Http Client library!In "Ning"