# 11.4 实例：Http分块下载

本节将通过一个“Http分块下载”的示例演示一下dio的具体用法。

### 原理

Http协议定义了分块传输的响应header字段，但具体是否支持取决于Server的实现，我们可以指定请求头的“range”字段来验证服务器是否支持分块传输。例如，我们可以利用curl命令来验证：

bogon:~ duwen$ curl -H "Range: bytes=0-10" http://download.dcloud.net.cn/HBuilder.9.0.2.macosx\_64.dmg -v  
# 请求头  
> GET /HBuilder.9.0.2.macosx\_64.dmg HTTP/1.1  
> Host: download.dcloud.net.cn  
> User-Agent: curl/7.54.0  
> Accept: \*/\*  
> Range: bytes=0-10  
# 响应头  
< HTTP/1.1 206 Partial Content  
< Content-Type: application/octet-stream  
< Content-Length: 11  
< Connection: keep-alive  
< Date: Thu, 21 Feb 2019 06:25:15 GMT  
< Content-Range: bytes 0-10/233295878

我们在请求头中添加“Range: bytes=0-10”的作用是，告诉服务器本次请求我们只想获取文件0-10(包括10，共11字节)这块内容。如果服务器支持分块传输，则响应状态码为206，表示“部分内容”，并且同时响应头中包含“Content-Range”字段，如果不支持则不会包含。我们看看上面“Content-Range”的内容：

Content-Range: bytes 0-10/233295878

0-10表示本次返回的区块，233295878代表文件的总长度，单位都是byte, 也就是该文件大概233M多一点。

基于此，我们可以设计一个简单的多线程的文件分块下载器，实现的思路是：

1. 先检测是否支持分块传输，如果不支持，则直接下载；若支持，则将剩余内容分块下载。
2. 各个分块下载时保存到各自临时文件，等到所有分块下载完后合并临时文件。
3. 删除临时文件。

### 实现

下面是整体的流程：

// 通过第一个分块请求检测服务器是否支持分块传输   
Response response = await downloadChunk(url, 0, firstChunkSize, 0);  
if (response.statusCode == 206) { //如果支持  
 //解析文件总长度，进而算出剩余长度  
 total = int.parse(  
 response.headers.value(HttpHeaders.contentRangeHeader).split("/").last);  
 int reserved = total -  
 int.parse(response.headers.value(HttpHeaders.contentLengthHeader));  
 //文件的总块数(包括第一块)  
 int chunk = (reserved / firstChunkSize).ceil() + 1;  
 if (chunk > 1) {  
 int chunkSize = firstChunkSize;  
 if (chunk > maxChunk + 1) {  
 chunk = maxChunk + 1;  
 chunkSize = (reserved / maxChunk).ceil();  
 }  
 var futures = <Future>[];  
 for (int i = 0; i < maxChunk; ++i) {  
 int start = firstChunkSize + i \* chunkSize;  
 //分块下载剩余文件   
 futures.add(downloadChunk(url, start, start + chunkSize, i + 1));  
 }  
 //等待所有分块全部下载完成  
 await Future.wait(futures);  
 }  
 //合并文件文件   
 await mergeTempFiles(chunk);  
}

下面我们使用dio的download API 实现downloadChunk：

//start 代表当前块的起始位置，end代表结束位置  
//no 代表当前是第几块  
Future<Response> downloadChunk(url, start, end, no) async {  
 progress.add(0); //progress记录每一块已接收数据的长度  
 --end;  
 return dio.download(  
 url,  
 savePath + "temp$no", //临时文件按照块的序号命名，方便最后合并  
 onReceiveProgress: createCallback(no), // 创建进度回调，后面实现  
 options: Options(  
 headers: {"range": "bytes=$start-$end"}, //指定请求的内容区间  
 ),  
 );  
}

接下来实现mergeTempFiles:

Future mergeTempFiles(chunk) async {  
 File f = File(savePath + "temp0");  
 IOSink ioSink= f.openWrite(mode: FileMode.writeOnlyAppend);  
 //合并临时文件   
 for (int i = 1; i < chunk; ++i) {  
 File \_f = File(savePath + "temp$i");  
 await ioSink.addStream(\_f.openRead());  
 await \_f.delete(); //删除临时文件  
 }  
 await ioSink.close();  
 await f.rename(savePath); //合并后的文件重命名为真正的名称  
}

下面我们看一下完整实现：

/// Downloading by spiting as file in chunks  
Future downloadWithChunks(  
 url,  
 savePath, {  
 ProgressCallback onReceiveProgress,  
}) async {  
 const firstChunkSize = 102;  
 const maxChunk = 3;  
  
 int total = 0;  
 var dio = Dio();  
 var progress = <int>[];  
  
 createCallback(no) {  
 return (int received, \_) {  
 progress[no] = received;  
 if (onReceiveProgress != null && total != 0) {  
 onReceiveProgress(progress.reduce((a, b) => a + b), total);  
 }  
 };  
 }  
  
 Future<Response> downloadChunk(url, start, end, no) async {  
 progress.add(0);  
 --end;  
 return dio.download(  
 url,  
 savePath + "temp$no",  
 onReceiveProgress: createCallback(no),  
 options: Options(  
 headers: {"range": "bytes=$start-$end"},  
 ),  
 );  
 }  
  
 Future mergeTempFiles(chunk) async {  
 File f = File(savePath + "temp0");  
 IOSink ioSink= f.openWrite(mode: FileMode.writeOnlyAppend);  
 for (int i = 1; i < chunk; ++i) {  
 File \_f = File(savePath + "temp$i");  
 await ioSink.addStream(\_f.openRead());  
 await \_f.delete();  
 }  
 await ioSink.close();  
 await f.rename(savePath);  
 }  
  
 Response response = await downloadChunk(url, 0, firstChunkSize, 0);  
 if (response.statusCode == 206) {  
 total = int.parse(  
 response.headers.value(HttpHeaders.contentRangeHeader).split("/").last);  
 int reserved = total -  
 int.parse(response.headers.value(HttpHeaders.contentLengthHeader));  
 int chunk = (reserved / firstChunkSize).ceil() + 1;  
 if (chunk > 1) {  
 int chunkSize = firstChunkSize;  
 if (chunk > maxChunk + 1) {  
 chunk = maxChunk + 1;  
 chunkSize = (reserved / maxChunk).ceil();  
 }  
 var futures = <Future>[];  
 for (int i = 0; i < maxChunk; ++i) {  
 int start = firstChunkSize + i \* chunkSize;  
 futures.add(downloadChunk(url, start, start + chunkSize, i + 1));  
 }  
 await Future.wait(futures);  
 }  
 await mergeTempFiles(chunk);  
 }  
}

现在可以进行分块下载了：

main() async {  
 var url = "http://download.dcloud.net.cn/HBuilder.9.0.2.macosx\_64.dmg";  
 var savePath = "./example/HBuilder.9.0.2.macosx\_64.dmg";  
 await downloadWithChunks(url, savePath, onReceiveProgress: (received, total) {  
 if (total != -1) {  
 print("${(received / total \* 100).floor()}%");  
 }  
 });  
}

### 思考

1. 分块下载真的能提高下载速度吗？

* 其实下载速度的主要瓶颈是取决于网络速度和服务器的出口速度，如果是同一个数据源，分块下载的意义并不大，因为服务器是同一个，出口速度确定的，主要取决于网速，而上面的例子正式同源分块下载，读者可以自己对比一下分块和不分块的的下载速度。如果有多个下载源，并且每个下载源的出口带宽都是有限制的，这时分块下载可能会更快一下，之所以说“可能”，是由于这并不是一定的，比如有三个源，三个源的出口带宽都为1G/s，而我们设备所连网络的峰值假设只有800M/s，那么瓶颈就在我们的网络。即使我们设备的带宽大于任意一个源，下载速度依然不一定就比单源单线下载快，试想一下，假设有两个源A和B，速度A源是B源的3倍，如果采用分块下载，两个源各下载一半的话，读者可以算一下所需的下载时间，然后再算一下只从A源下载所需的时间，看看哪个更快。
* 分块下载的最终速度受设备所在网络带宽、源出口速度、每个块大小、以及分块的数量等诸多因素影响，实际过程中很难保证速度最优。在实际开发中，读者可可以先测试对比后再决定是否使用。

1. 分块下载有什么实际的用处吗？

* 分块下载还有一个比较使用的场景是断点续传，可以将文件分为若干个块，然后维护一个下载状态文件用以记录每一个块的状态，这样即使在网络中断后，也可以恢复中断前的状态，具体实现读者可以自己尝试一下，还是有一些细节需要特别注意的，比如分块大小多少合适？下载到一半的块如何处理？要不要维护一个任务队列？