今天我发现了一个关于请求加密的有效写法，特此分享给大家。如果你的加密需求是将请求参数也包含在内，通常情况下，我们需要先将请求体转换成 JSON 格式或其他对象类型，再使用字符串的形式进行加密操作。以下是伪代码示例，展示了这一过程的实现方法：

```
String payloadString = ModelOptionsUtils.toJsonString(payload);
String hashedRequestPayload = sha256Hex(payloadString);
//将hashedRequestPayload封装到jsonContentHeaders中，并添加到请求头中
ResponseEntity<String> retrieve = this.restClient.post().uri("/").headers(headers -> {
    headers.addAll(jsonContentHeaders);
}).body(chatRequest).retrieve().toEntity(String.class);
```

这种方法看起来有些繁琐，而且如果在转换过程中与实际请求的结构不一致，可能会导致加密失败。关键问题在于 `ModelOptionsUtils.toJsonString(payload);` 这一过程，它与 `restClient` 中对象转化的方式并不完全一致。如果在转化时出现任何问题，我们不仅难以复现错误，还可能会面临很难排查的问题。理想的情况是，如果我们能够准确了解请求体在加密前最终的转换结果，那将大大简化排查过程并提高加密的可靠性。

所以今天我们就以拦截器的形式加密一下，保证与真实上传的请求体保持一致。

拦截器

今天简单介绍一下请求类 `RestClient` 。其实，它和我们之前使用的 `HttpUtils` 功能上是类似的，但相较于 `HttpUtils` ， `RestClient` 在可操作性和灵活性方面做了很多优化，能够提供更加丰富的功能和更高效的操作体验。特别是今天我们要重点介绍的拦截器功能，它可以让我们更加便捷地处理请求和响应的相关逻辑。接下来，我们就通过一个示例来详细了解这个过滤器的使用。

```
ApiAuthHttpRequestInterceptor apiAuthHttpRequestInterceptor = new
ApiAuthHttpRequestInterceptor();
this.restClient = RestClient.Builder.baseUrl(baseUrl)
        .defaultHeaders(jsonContentHeaders)
        .defaultStatusHandler(responseErrorHandler)
        .requestInterceptor(apiAuthHttpRequestInterceptor)
        .build();
```

我们在具体看下ApiAuthHttpRequestInterceptor类是如何实现的。

```
public class ApiAuthHttpRequestInterceptor implements ClientHttpRequestInterceptor {
    @Override
    public ClientHttpResponse intercept(HttpRequest request, byte[] body,
ClientHttpRequestExecution execution) throws IOException {
        String hashedRequestPayload = sha256Hex(payloadString);
        //将hashedRequestPayload封装到jsonContentHeaders中，并添加到请求头中
        request.getHeaders().putAll(httpHeadersConsumer);
      ClientHttpResponse response = execution.execute(request, body);
```

```
        return response;
    }
}
```

这一步的伪代码非常简洁明了，主要是因为我们能够直接获取到需要发送的请求体，因此无需再进行复杂的对象转换或序列化处理，避免了中间环节可能带来的不必要错误或数据变动。

内部原理

这里面的原理也很简单，核心思想就是对我们所注入的每个拦截器进行逐一遍历，并按照预定的逻辑依次执行我们重写的相关方法。通过这样的方式，可以在不改变原有逻辑的基础上，实现灵活的扩展与控制。具体的流程和实现细节如下图所示：

```
private class InterceptingRequestExecution implements ClientHttpRequestExecution {

    private final Iterator<ClientHttpRequestInterceptor> iterator;

    public InterceptingRequestExecution() { this.iterator = interceptors.iterator(); }

    @Override
    public ClientHttpResponse execute(HttpRequest request, byte[] body) throws IOException {
        if (this.iterator.hasNext()) {
            ClientHttpRequestInterceptor nextInterceptor = this.iterator.next();
            return nextInterceptor.intercept(request, body, execution: this);
        }
        else {
            HttpMethod method = request.getMethod();
            ClientHttpRequest delegate = requestFactory.createRequest(request.getURI(), method);
            request.getHeaders().forEach((key, value) -> delegate.getHeaders().addAll(key, value));
            if (body.length > 0) {
                if (delegate instanceof StreamingHttpOutputMessage streamingOutputMessage) {
                    streamingOutputMessage.setBody(new StreamingHttpOutputMessage.Body() {
                        @Override
                        public void writeTo(OutputStream outputStream) throws IOException {
                            StreamUtils.copy(body, outputStream);
                        }
```

这里就是一个递归的过程，全部完成之后就可以正常去请求了。

总结

通过今天的分享，我们探讨了如何在请求中实现加密操作，并通过拦截器优化了加密过程。在传统方法中，依赖对象转换和序列化处理，容易导致加密不一致或难以调试的问题。通过引入拦截器，我们能够直接操作请求体，避免了不必要的转换步骤，确保加密过程与请求体完全一致，从而提高了加密的可靠性和调试的便捷性。希望这种方法对大家在加密需求的实现上有所帮助！