# 15.5 网络请求封装

本节我们会基于前面介绍过的dio网络库封装APP中用到的网络请求接口，并同时应用一个简单的缓存策略。下面我们先介绍一下网络接口缓存原理，然后再封装APP的业务请求接口。

## 15.5.1 网络接口缓存

由于在国内访问Github服务器速度较慢，所以我们应用一些简单的缓存策略：将请求的url作为key，对请求的返回值在一个指定时间段类进行缓存，另外设置一个最大缓存数，当超过最大缓存数后移除最早的一条缓存。但是也得提供一种针对特定接口或请求决定是否启用缓存的机制，这种机制可以指定哪些接口或那次请求不应用缓存，这种机制是很有必要的，比如登录接口就不应该缓存，又比如用户在下拉刷新时就不应该再应用缓存。在实现缓存之前我们先定义保存缓存信息的CacheObject类：

class CacheObject {  
 CacheObject(this.response)  
 : timeStamp = DateTime.now().millisecondsSinceEpoch;  
 Response response;  
 int timeStamp; // 缓存创建时间  
  
 @override  
 bool operator ==(other) {  
 return response.hashCode == other.hashCode;  
 }  
  
 //将请求uri作为缓存的key  
 @override  
 int get hashCode => response.realUri.hashCode;  
}

接下来我们需要实现具体的缓存策略，由于我们使用的是dio package，所以我们可以直接通过拦截器来实现缓存策略：

import 'dart:collection';  
import 'package:dio/dio.dart';  
import '../index.dart';  
  
class CacheObject {  
 CacheObject(this.response)  
 : timeStamp = DateTime.now().millisecondsSinceEpoch;  
 Response response;  
 int timeStamp;  
  
 @override  
 bool operator ==(other) {  
 return response.hashCode == other.hashCode;  
 }  
  
 @override  
 int get hashCode => response.realUri.hashCode;  
}  
  
class NetCache extends Interceptor {  
 // 为确保迭代器顺序和对象插入时间一致顺序一致，我们使用LinkedHashMap  
 var cache = LinkedHashMap<String, CacheObject>();  
  
 @override  
 onRequest(RequestOptions options) async {  
 if (!Global.profile.cache.enable) return options;  
 // refresh标记是否是"下拉刷新"  
 bool refresh = options.extra["refresh"] == true;  
 //如果是下拉刷新，先删除相关缓存  
 if (refresh) {  
 if (options.extra["list"] == true) {  
 //若是列表，则只要url中包含当前path的缓存全部删除（简单实现，并不精准）  
 cache.removeWhere((key, v) => key.contains(options.path));  
 } else {  
 // 如果不是列表，则只删除uri相同的缓存  
 delete(options.uri.toString());  
 }  
 return options;  
 }  
 if (options.extra["noCache"] != true &&  
 options.method.toLowerCase() == 'get') {  
 String key = options.extra["cacheKey"] ?? options.uri.toString();  
 var ob = cache[key];  
 if (ob != null) {  
 //若缓存未过期，则返回缓存内容  
 if ((DateTime.now().millisecondsSinceEpoch - ob.timeStamp) / 1000 <  
 Global.profile.cache.maxAge) {  
 return cache[key].response;  
 } else {  
 //若已过期则删除缓存，继续向服务器请求  
 cache.remove(key);  
 }  
 }  
 }  
 }  
  
 @override  
 onError(DioError err) async {  
 // 错误状态不缓存  
 }  
  
 @override  
 onResponse(Response response) async {  
 // 如果启用缓存，将返回结果保存到缓存  
 if (Global.profile.cache.enable) {  
 \_saveCache(response);  
 }  
 }  
  
 \_saveCache(Response object) {  
 RequestOptions options = object.request;  
 if (options.extra["noCache"] != true &&  
 options.method.toLowerCase() == "get") {  
 // 如果缓存数量超过最大数量限制，则先移除最早的一条记录  
 if (cache.length == Global.profile.cache.maxCount) {  
 cache.remove(cache[cache.keys.first]);  
 }  
 String key = options.extra["cacheKey"] ?? options.uri.toString();  
 cache[key] = CacheObject(object);  
 }  
 }  
  
 void delete(String key) {  
 cache.remove(key);  
 }  
}

关于代码的解释都在注释中了，在此需要说明的是dio包的option.extra是专门用于扩展请求参数的，我们通过定义了“refresh”和“noCache”两个参数实现了“针对特定接口或请求决定是否启用缓存的机制”，这两个参数含义如下：

|  |  |  |
| --- | --- | --- |
| 参数名 | 类型 | 解释 |
| refresh | bool | 如果为true，则本次请求不使用缓存，但新的请求结果依然会被缓存 |
| noCache | bool | 本次请求禁用缓存，请求结果也不会被缓存。 |

## 15.5.2 封装网络请求

一个完整的APP，可能会涉及很多网络请求，为了便于管理、收敛请求入口，工程上最好的作法就是将所有网络请求放到同一个源码文件中。由于我们的接口都是请求的Github 开发平台提供的API，所以我们定义一个Git类，专门用于Github API接口调用。另外，在调试过程中，我们通常需要一些工具来查看网络请求、响应报文，使用网络代理工具来调试网络数据问题是主流方式。配置代理需要在应用中指定代理服务器的地址和端口，另外Github API是HTTPS协议，所以在配置完代理后还应该禁用证书校验，这些配置我们在Git类初始化时执行（init()方法）。下面是Git类的源码：

import 'dart:async';  
import 'dart:convert';  
import 'dart:io';  
import 'package:dio/dio.dart';  
import 'package:dio/adapter.dart';  
import 'package:flutter/material.dart';  
import '../index.dart';  
  
class Git {  
 // 在网络请求过程中可能会需要使用当前的context信息，比如在请求失败时  
 // 打开一个新路由，而打开新路由需要context信息。  
 Git([this.context]) {  
 \_options = Options(extra: {"context": context});  
 }  
  
 BuildContext context;  
 Options \_options;  
 static Dio dio = new Dio(BaseOptions(  
 baseUrl: 'https://api.github.com/',  
 headers: {  
 HttpHeaders.acceptHeader: "application/vnd.github.squirrel-girl-preview,"  
 "application/vnd.github.symmetra-preview+json",  
 },  
 ));  
  
 static void init() {  
 // 添加缓存插件  
 dio.interceptors.add(Global.netCache);  
 // 设置用户token（可能为null，代表未登录）  
 dio.options.headers[HttpHeaders.authorizationHeader] = Global.profile.token;  
  
 // 在调试模式下需要抓包调试，所以我们使用代理，并禁用HTTPS证书校验  
 if (!Global.isRelease) {  
 (dio.httpClientAdapter as DefaultHttpClientAdapter).onHttpClientCreate =  
 (client) {  
 client.findProxy = (uri) {  
 return "PROXY 10.1.10.250:8888";  
 };  
 //代理工具会提供一个抓包的自签名证书，会通不过证书校验，所以我们禁用证书校验  
 client.badCertificateCallback =  
 (X509Certificate cert, String host, int port) => true;  
 };  
 }  
 }  
  
 // 登录接口，登录成功后返回用户信息  
 Future<User> login(String login, String pwd) async {  
 String basic = 'Basic ' + base64.encode(utf8.encode('$login:$pwd'));  
 var r = await dio.get(  
 "/users/$login",  
 options: \_options.merge(headers: {  
 HttpHeaders.authorizationHeader: basic  
 }, extra: {  
 "noCache": true, //本接口禁用缓存  
 }),  
 );  
 //登录成功后更新公共头（authorization），此后的所有请求都会带上用户身份信息  
 dio.options.headers[HttpHeaders.authorizationHeader] = basic;  
 //清空所有缓存  
 Global.netCache.cache.clear();  
 //更新profile中的token信息  
 Global.profile.token = basic;  
 return User.fromJson(r.data);  
 }  
  
 //获取用户项目列表  
 Future<List<Repo>> getRepos(  
 {Map<String, dynamic> queryParameters, //query参数，用于接收分页信息  
 refresh = false}) async {  
 if (refresh) {  
 // 列表下拉刷新，需要删除缓存（拦截器中会读取这些信息）  
 \_options.extra.addAll({"refresh": true, "list": true});  
 }  
 var r = await dio.get<List>(  
 "user/repos",  
 queryParameters: queryParameters,  
 options: \_options,  
 );  
 return r.data.map((e) => Repo.fromJson(e)).toList();  
 }  
}

可以看到我们在init()方法中，我们判断了是否是调试环境，然后做了一些针对调试环境的网络配置（设置代理和禁用证书校验）。而Git.init()方法是应用启动时被调用的（Global.init()方法中会调用Git.init()）。

另外需要注意，我们所有的网络请求是通过同一个dio实例（静态变量）发出的，在创建该dio实例时我们将Github API的基地址和API支持的Header进行了全局配置，这样所有通过该dio实例发出的请求都会默认使用者些配置。

在本实例中，我们只用到了登录接口和获取用户项目的接口，所以在Git类中只定义了login(…)和getRepos(…)方法，如果读者要在本实例的基础上扩充功能，读者可以将其它的接口请求方法添加到Git类中，这样便实现了网络请求接口在代码层面的集中管理和维护。