# 请求缓存

## 🔹 请求缓存的基本原理

其实可以拆成三个关键问题：

### 1. 什么时候需要缓存？

- 如果一个请求的结果 **在一段时间内不会变化**，我们可以复用它，避免反复请求服务器。
  👉 比如：用户信息、配置表、热门商品列表。
- 如果结果经常变化（比如实时股票价格），缓存就要么不适合，要么 TTL 设置得很短。

### 2. 缓存怎么存？

- 用一个 **Map / 对象** 来保存请求结果。
- key：请求的唯一标识（通常是 `method + url + params + data`）。
- value：请求结果 + 过期时间。

👉 举例：

text

```
key:  GET /api/user?id=1
value: {
  data: { name: "小明", age: 18 },
  expireAt: 1695028800000  // 过期时间戳
}
```

### 3. 缓存的取用逻辑

流程就是三步走：

1. **查缓存**
   - 有缓存 & 未过期 → 直接返回结果（不发请求）。
   - 没缓存 / 已过期 → 进入第 2 步。
2. **发请求**
   - 向服务器发起网络请求，拿到结果。
3. **存缓存**
   - 把结果写入缓存（加上过期时间）。

下次再遇到同样的请求，就可以命中缓存，直接返回。

# 简单的请求缓存

```
//这是一个基于Promise的缓存类
//实现了一个简单的请求缓存的功能

//导入生成hash值的库
import md5 from 'md5'

//这里导入我们已经封装过的一个基础的axios实例
import API from './baseCancel-cancelToekn'

//创建一个缓存类
class CachePromise{

  constructor(){
    //缓存使用这个map来存储
    this.cache=new Map()
  }

  //生成唯一的key
  generateKey(config){
    const {method,url,params,data}=config
    return md5(method+url+JSON.stringify(params)+JSON.stringify(data))
  }

  //发起请求(优先命中缓存)
  request(config,ttl=5000){

    //生成这次请求的唯一key
    const key=this.generateKey(config)

    //查找缓存
    const cached=this.cache.get(key)

    //如果有缓存
    if(cached){

      //检查缓存是否过期
      const isExpired=(Date.now()-cached.timestamp)>ttl

      if(!isExpired){
        return cached.promise //直接返回promise
      }else{
        this.cache.delete(key) //删除过期缓存
      }

    }

    //如果没有缓存，发起请求
    const promise=API.request(config).then(res=>{
      //API.request返回的是promise
      return res
    }).catch(err=>{

      //请求失败删除缓存
```

```
      //请求失败删除缓存
      this.cache.delete(key)
      return Promise.reject(err)
    })

    //存储到缓存
    this.cache.set(key,{
      promise:promise,
      timestamp:Date.now() //存储时间戳，记录当前时间
    })

    //返回这个promise给外部
    return promise

  }


  //请求指定key的缓存
  clear(config){
    const key=this.generateKey(config)
    this.cache.delete(key)
  }

  //清除全部缓存
  clearAll(){
    this.cache.clear()
  }
}

export default new CachePromise()
```

**为什么一定要用MD5生成key**

**如果直接把config当成key存入的map的话，会出问题**

**因为map的key是只判断 你的key的引用地址是不是同一个，而不会理会你的属性变化**

**比如你的config的请求的params发生数值变化了，但是map会认为这还是旧的请求，然后命中缓存**

**然后这里导入的API是二次封装之后的axios**

```
import axios from "axios";

export const API=axios.create({


    baseURL:'http://localhost:5000/api',
    timeout:10000,
    cors:true,
    withCredentials:true,

    validateStatus:(status)=>{
      if(status>400){
        console.error('请求失败')
      }

      return status>=200 && status<500
    }
})

//用map来存储取消取消函数 ， key是请求的url+method
const pendingRequests=new Map()

//生成唯一的请求key
const getPendingKey=(config)=>{

  return `${config.url}-${config.method}`
}

//请求拦截器
API.interceptors.request.use(

  (config)=>{
    //生成唯一的取消请求key
    const key=getPendingKey(config)

    config.cancelToken=new axios.CancelToken((c)=>{
      if(pendingReuqests.has(key)){
        //如果已经有这个请求了，重复请求取消掉
        const cancel=pendingRequests.get(key)
        cancel('取消重复请求')
      }else{
        //没有就存储这个请求的取消函数
        pendingRequests.set(key,c)
      }
    })
    return config
```

```
  },

  //错误处理
  (error)=>{
    return Promise.reject(error)
  }

)

//响应拦截器
API.interceptors.response.use(
  response=>{
    //响应回来后删除这个请求
    const key=getPendingKey(response.config)
    pendingRequests.delete(key)

    //正常返回数据
    return response
  },

  //响应失败
  error=>{
  return Promise.reject(error)
  }

)


//取消对应请求
export const cancelRequest=(url,method)=>{
  const key=`${url}-${method}`
  if(pendingRequests.has(key)){
    const cancel=pendingRequests.get(key)
    cancel('手动取消请求')
    pendingRequests.delete(key)
  }
}

//取消全部请求
export const cancelAllRequests=()=>{
    for(const [key,cancel] of pendingRequests){
      cancel('手动取消全部请求')
    }
    //清空队列map
    pendingRequests.clear()

}
```

```
}

//默认导出实例
export default API
```

**他是基于自己内部实现了一个请求取消机制**

**(取消重复的请求)**

**但是这个判断比较基础，只把method和url作为key**

**我们这里使用这个封装好的缓存类进行三次封装**

```
import cacheAPI from './CacheAPI'

// 获取用户信息
export const getUser = (id, ttl = 5000) => {
  return cacheAPI.request({
    url: '/user',
    method: 'get',
    params: { id }
  }, ttl)
}

// 更新用户信息（不缓存）
export const updateUser = (id, data) => {
  return cacheAPI.request({
    url: `/user/${id}`,
    method: 'post',
    data
  }, 0)
}

// 删除用户（不缓存）
export const deleteUser = (id) => {
  return cacheAPI.request({
    url: `/user/${id}`,
    method: 'delete'
  }, 0)
}
```

然后我们知道 **缓存的优先级是高于发请求的**

**就是如果缓存没有命中，才会去发起真正的具体请求**

**（同时会执行删除，过期的缓存）**

然后我们在基础文件封装的**取消请求**（取消重复请求），其实不会被触发，

**因为重复的请求都优先命中缓存了**

**所以说取消重复请求不会被触发，因为重复的请求都命中缓存了**


**但是这里衍生出来一个问题，那如果我真的需要取消某个请求怎么办？**

## 解决方法

**方案 A：在缓存里也存取消函数**

- 每次发请求时生成 `{ promise, cancel }` 并存入缓存
- 缓存命中时，返回的也是 `{ promise, cancel }`
- 外部可以随时调用 `cancel()` 取消这个请求，即使是缓存命中

```
js

const { promise, cancel } = cacheAPI.request(config)
```

- 这样 **缓存命中 + 可取消** 都能满足

**方案 B：外部不取消缓存命中请求**

- 如果请求已经完成或正在使用缓存，直接返回 promise
- **不支持取消缓存命中请求**
- 优点：逻辑简单
- 缺点：无法"撤回"已经缓存的请求

推荐方案A

**最好的方法就是改变一下，map的存储结构**

**用一个result对象，来存储{ promise，signal}**

```
// 新建 cancelToken
let cancel
config.cancelToken = new axios.CancelToken(c => {
  cancel = c
})

// 发请求
const promise = API.request(config)
  .then(res => res)
  .catch(err => {
    this.cache.delete(key)
    throw err
  })

// 封装结果
const result = { promise, cancel }
```

用请求的config生成一个

存储函数和promise

# 最终版本

```
//导入已经封装好的一个API （基于Axios）
import API from './advancedCancel-cancelToken'

//导入计算hash的库
import md5 from 'md5'

//生成取消请求函数需要用到
import axios from 'axios'

// 创建类
class CachePromise{

  constructor(){
    //使用map来存储缓存
    this.cache=new Map()
  }


  //一个函数，用于生成唯一的key
  generateKey(config){
    const {method,url,params,data}=config
    return md5(method+url+JSON.stringify(params)+JSON.stringify(data))
  }

  //发起请求(优先命中缓存)
  requestCache (config,ttl=50000){

    //生成这次请求唯一的KEY
    const key=this.generateKey(config)

    //查找缓存
    const cached=this.cache.get(key)

    //检查缓存
    if(cached){

      //检查缓存是否过期
      const isExpired=(Date.now()-cached.timestamp)>ttl

      if(!isExpired){
        return cached.result //返回缓存 包含promise和cancel取消请求函数
      }

    }

    //如果没有缓存，发起请求

    const Promise=API.request(config).then(res=>{
      //返回成功的promise
      return res
    }).catch(err=>{
      //请求失败删除缓存
      this.cache.delete(key)
      return Promise.reject(err)
    })
```

```
    //生成config请求的取消函数
    let cancel
    config.cancelToken=new axios.CancelToken((c)=>{
      cancel=c
    })

    const result={Promise,cancel}

    //存储到缓存
    this.cache.set(key,{
      result:result,
      timestamp:Date.now() //存储时间戳，记录当前时间
    })
  }

  //清除缓存
  clear(){
    this.cache.clear()
  }

  //清除指定key的缓存
  clearKey(config){
    const key=this.generateKey(config)
    this.cache.delete(key)
  }

  //清除过期缓存
  clearExpired(ttl=5000){
    const now=Date.now()
    for(const [key,value] of this.cache){
      if((now-value.timestamp)>ttl){
        this.cache.delete(key)
      }
    }
  }

  //取消单个请求
  cancelRequest(config){
    const key=this.generateKey(config)
    const cached=this.cache.get(key)
    if(cached){

      //调用取消请求函数
      cached.result.cancel('手动取消请求')

      //删除缓存
      this.cache.delete(key)
    }
  }

  //取消全部请求
  cancelAllRequests(){
    for(const [key,value] of this.cache){
```

```
      //调用取消请求函数
      value.result.cancel('手动取消全部请求')
    }

    //清空缓存
    this.cache.clear()
  }

}
,,

//生成config请求的取消函数
let cancel

//这里是使用旧版的cancelToken来生成取消函数
config.cancelToken=new axios.CancelToken((c)=>{
  cancel=c
})

//如果是使用新版的AbortController来生成取消函数

const controller=new AbortController()

//生成到config中
config.signal=controller.signal

//生成取消函数
cancel=controller.abort
```

使用新版本的取消函数

## 进阶封装（请求缓存）

**设置多一个类，这个类用来存放所有的请求（请求队列）**

**我们先写一个基础的请求队列的类**

```
//写一个简单的请求队列类

class RequestQueue{

  //用Map存储请求
  constructor(){
    this.RequestQueue= new Map()
  }

  //加入请求队列
  addRequest(key,cancel){
    if(!this.RequestQueue.has(key)){
      this.RequestQueue.set(key,cancel)
    }
  }

  //移除请求队列
  removeRequest(key){
    if(this.RequestQueue.has(key)){
      this.RequestQueue.delete(key)
    }
  }

  //取出请求
  getRequest(key){
    return this.RequestQueue.get(key)
  }

  //取出全部请求
  getAllRequests(){
    return Array.from(this.RequestQueue.values())
  }

  //清空请求队列
  clear(){
    this.RequestQueue.clear()
  }


}
```

**然后在缓存类里面加入**

```
//导入已经封装好的一个API （基于Axios）
import API from './advancedCancel-cancelToken'

//导入计算hash的库
import md5 from 'md5'

//生成取消请求函数需要用到
import axios from 'axios'

//导入请求队列
import RequestQueue from './requestQueue'


// 创建类
class CachePromise{

  constructor(){
    //使用map来存储缓存
    this.cache=new Map()

    // ✅ 在这里引入请求队列x
    this.requestQueue=new RequestQueue()
  }


  //一个函数，用于生成唯一的key
  generateKey(config){
    const {method,url,params,data}=config
    return md5(method+url+JSON.stringify(params)+JSON.stringify(data))
  }

  //发起请求(优先命中缓存)
  requestCache (config,ttl=50000){

    //生成这次请求唯一的KEY
    const key=this.generateKey(config)

    //查找缓存
    const cached=this.cache.get(key)

    //检查缓存
    if(cached){

      //检查缓存是否过期
      const isExpired=(Date.now()-cached.timestamp)>ttl

      if(!isExpired){
        return cached.result //返回缓存 包含promise和cancel取消请求函数
      }

    }

    //如果没有缓存，发起请求

    const Promise=API.request(config).then(res=>{
      //返回成功的promise
      return res
    }).catch(err=>{
      //请求失败删除缓存
```

```
    //请求失败删除缓存
    this.cache.delete(key)
    return Promise.reject(err)
  }).finally(()=>{

    //请求完成后，从请求队列中移除  (无论成功失败)
    this.requestQueue.removeRequest(key)
  })

  //生成config请求的取消函数
  let cancel

  //这里是使用旧版的cancelToken来生成取消函数
  config.cancelToken=new axios.CancelToken((c)=>{
    cancel=c
  })

  //如果是使用新版的AbortController来生成取消函数

  const controller=new AbortController()

  //生成到config中
  config.signal=controller.signal

  //生成取消函数
  cancel=controller.abort

  //保存结果
  const result={Promise,cancel}

  //存储到缓存
  this.cache.set(key,{
    result:result,
    timestamp:Date.now() //存储时间戳，记录当前时间
  })


  // ✅ 将请求添加到请求队列中
  this.requestQueue.addRequest(key,cancel)

  //返回这个promise给外部
  return result
}

//清除缓存
clear(){
  this.cache.clear()
  this.requestQueue.clear()
}

//清除指定key的缓存
clearKey(config){
  const key=this.generateKey(config)
  this.cache.delete(key)
  this.requestQueue.removeRequest(key)
}

//清除过期缓存
clearExpired(ttl=5000){
```

```
    const now=Date.now()
    for(const [key,value] of this.cache){
      if((now-value.timestamp)>ttl){
        this.cache.delete(key)
        this.requestQueue.removeRequest(key)
      }
    }
  }

  //取消单个请求
  cancelRequest(config){
    const key=this.generateKey(config)
    const cached=this.cache.get(key)
    if(cached){

      //调用取消请求函数
      cached.result.cancel('手动取消请求')

      //删除缓存
      this.cache.delete(key)

      //从请求队列中移除
      this.requestQueue.removeRequest(key)
    }
  }

  //取消全部请求
  cancelAllRequests(){
    for(const [key,value] of this.cache){
      //调用取消请求函数
      value.result.cancel('手动取消全部请求')
    }

    //清空缓存
    this.cache.clear()

    //清空队列
    this.requestQueue.clear()
  }

}
```