

目录

前言	1.1
HTTP简介	1.2
HTTP学习目的	1.2.1
HTTP简介	1.2.2
HTTP内部流程	1.2.3
HTTP基本逻辑总结	1.2.4
HTTP详解	1.3
HTTP的典型结构	1.3.1
HTTP的Header头	1.3.2
HTTP的请求参数和编码	1.3.3
HTTP的响应的状态码	1.3.4
HTTP的响应数据格式JSON	1.3.5
HTTP相关	1.4
HTTP的工具和库	1.4.1
HTTP工具	1.4.1.1
HTTP库	1.4.1.2
HTTP的后台API设计	1.4.2
HTTP相关心得	1.4.3
HTTP的文件上传	1.4.4
混合内容	1.4.5
附录	1.5
优质教程	1.5.1
参考资料	1.5.2

HTTP知识总结

- 最新版本: v3.3
- 更新时间: 20210717

简介

总结HTTP技术相关内容。先进行了HTTP的简介，介绍了学习HTTP的目的，以及典型的HTTP的内部流程，以及总结了基本的逻辑。再进行HTTP的详细介绍，包括字段和结构，Header头，请求参数和编码，响应的状态码，响应数据的JSON格式等。再整理HTTP相关的一些内容，比如工具和不同语言的HTTP的网络库，以及如何进行后台HTTP相关的api接口设计，总结了一些相关心得，文件上传、混合内容等。最后附上一些HTTP相关的教程资料。

源码+浏览+下载

本书的各种源码、在线浏览地址、多种格式文件下载如下：

Gitbook源码

- [crifan/http_summary: HTTP知识总结](#)

如何使用此Gitbook源码去生成发布为电子书

详见：[crifan/gitbook_template: demo how to use crifan gitbook template and demo](#)

在线浏览

- [HTTP知识总结 book.crifan.com](#)
- [HTTP知识总结 crifan.github.io](#)

离线下载阅读

- [HTTP知识总结 PDF](#)
- [HTTP知识总结 ePUB](#)
- [HTTP知识总结 Mobi](#)

版权说明

此电子书教程的全部内容，如无特别说明，均为本人原创和整理。其中部分内容参考自网络，均已备注了出处。如有发现侵犯您版权，请通过邮箱联系我 `admin` 艾特 `crifan.com`，我会尽快删除。谢谢合作。

鸣谢

感谢我的老婆陈雪的包容理解和悉心照料，才使得我 crifan 有更多精力去专注技术专研和整理归纳出这些电子书和技术教程，特此鸣谢。

更多其他电子书

本人 crifan 还写了其他 100+ 本电子书教程，感兴趣可移步至：

[crifan/crifan_ebook_readme: Crifan的电子书的使用说明](#)

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新：2021-07-17 21:26:19

HTTP简介

接下来先简要介绍一下HTTP的基础知识。

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新: 2021-07-17 17:44:48

HTTP学习目的

写此HTTP相关内容的教程，期望能对不同类型的人员达到不同的目标：

- 普通用户用浏览器上网
 - 了解打开网址到页面显示背后的故事
- (iOS/Android)移动端开发
 - 了解用代码调用后台API接口时
 - 知道HTTP的有哪些方法，GET/POST等，知道其大概用途
 - 如何传递GET和POST时候的参数
 - GET时的 query string
 - POST时的body的json
 - 学习用 Postman 等工具去测试后台提供的接口
- 后台人员设计API接口
 - 如何设计一个 RESTful 的API
 - 减少不良风格的接口
 - 比如：/getUser, /updateUser
 - 学习用Postman等工具去测试自己写的API接口
 - 学习如何用Postman等工具去生成API文档
- 测试人员
 - 充分利用Postman等工具去实现接口的自动化测试

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新：2021-07-17 17:44:48

HTTP简介

HTTP = Hyper Text Transfer Protocol = 超文本传输协议

HTTP的总体思路是：

- 客户端：向服务器发送一个 请求 = Request， 请求头包含请求的方法、URI、协议版本、以及包含请求修饰符、客户信息和内容的类似于MIME的消息结构。
- 服务器：以一个状态行作出 响应 = Response， 相应的内容包括消息协议的版本，成功或者错误编码加上包含服务器信息、实体元信息以及可能的实体内容。

简而言之就是一个你问我答的协议：

- 客户端Client 问= 请求 = Request
- 服务器Server 答= 响应 = Response

关于HTTP的相关的详细知识，或许很多人不是很熟悉。但是：

- 作为普通电脑用户的你，经常用 浏览器（IE / Chrome / Firefox 等）去浏览网页时
- 作为程序开发，用网络库去调用服务器后台提供的接口时 其实内部都用到了 HTTP的技术。

比如一些典型情况：

普通用户用浏览器去查看网页

比如用户在用浏览器去打开网址：

<http://www.baidu.com>

去查看网页内容：



程序员写代码去调用后台接口

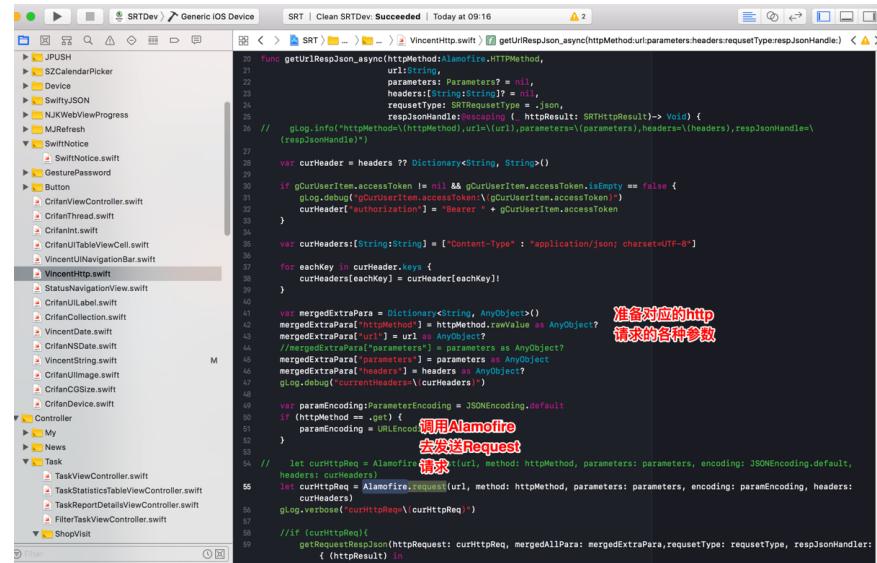
比如（卓越一线的iOS的）swift代码中利用 Alamofire 的HTTP网络库，去实现：

传递用户名ID+密码的参数和POST类型，调用Alamofire发送请求，完成用户的登录

```
// 登录
func requestLogin(userId: String, password: String) {
    let parameter: [String : Any] = [
        "userId": userId,
        "password": password
    ]
    gCurUserItem.password = password
    gCurUserItem.userId = userId

    if autoLogin == false {
        weakSelf?.pleaseWait()
    }
    getUrlResJson_sync(method: .post, url: ServerApi.getLoginUrl(), parameters: parameter as [String : AnyObject]?, responseHandler: { [weak weakSelf!] response in
        if let strongSelf = weakSelf {
            // 解析返回的数据
            if response.isSuccess {
                // 解析数据
                let dictionaryValue = response.successValue.dictionaryValue
                if let accessToken = dictionaryValue["token"]?.stringValue, let userId =
                    dictionaryValue["userId"]?.stringValue {
                    gCurUserItem.accessToken = accessToken
                    gCurUserItem.userId = userId
                    weakSelf?.log.debug("Token:\(gCurUserItem.accessToken)")
                    strongSelf.requestUserInfo()
                }
            }
        }
    })
}
```

HTTP工具



The screenshot shows a Xcode project window with the file `VincentHttp.swift` open. The code is written in Swift and uses the `Alamofire` library for making HTTP requests. The code is annotated with red text at the bottom right:

- 准备对的http请求的各种参数 (Preparing various parameters for the HTTP request)
- 调用Alamofire去发送Request (Calling Alamofire to send the Request)

```
func getUrlResponJson_async(httpMethod: Alamofire.HTTPMethod,  
                           url:String,  
                           parameters:Parameters? = nil,  
                           headers:[String:String]? = nil,  
                           requestType:SRTRequestType = .json,  
                           respJsonHandle:escaping ( HTTPResult<SRTHttpResult>) -> Void) {  
    gLog.info("httpMethod:(httpMethod),url:(url),parameters:(parameters),headers:(headers),respJsonHandle:(respJsonHandle)")  
  
    var curHeader = headers ?? Dictionary<String, String>()  
  
    if gCurUserItem.accessToken != nil && gCurUserItem.accessToken.isEmpty == false {  
        gLog.debug("gCurUserItem.accessToken:(gCurUserItem.accessToken)")  
        curHeader["authorization"] = "Bearer " + gCurUserItem.accessToken  
    }  
  
    var curHeaders:[String:String] = ["Content-Type": "application/json; charset=UTF-8"]  
  
    for eachKey in curHeader.keys {  
        curHeaders[eachKey] = curHeader[eachKey]  
    }  
  
    var mergedExtraPara = Dictionary<String, AnyObject>()  
    mergedExtraPara["httpMethod"] = httpMethod.rawValue as AnyObject?  
    mergedExtraPara["url"] = url as AnyObject?  
    //mergedExtraPara["parameters"] = parameters as AnyObject?  
    //mergedExtraPara["encoding"] = parameters.encoding as AnyObject?  
    mergedExtraPara["headers"] = headers as AnyObject?  
    gLog.debug("curHeaders:(curHeaders)")  
  
    var paramEncoding:ParameterEncoding = JSONEncoding.default  
    if (httpMethod == .get) {  
        paramEncoding = URL Encoding  
    }  
  
    // let curHttpReq = Alamofire.request(url, method: httpMethod, parameters: parameters, encoding: JSONEncoding.default,  
    // headers: curHeaders)  
    let curHttpReq = Alamofire.request(url, method: httpMethod, parameters: parameters, encoding: paramEncoding, headers:  
        curHeaders)  
    gLog.warning("curHttpReq:(curHttpReq)")  
  
    //if (curHttpReq){  
    //    getResponJson((httpRequest: curHttpReq, mergedAllPara: mergedExtraPara, requestType: requestType, respJsonHandle:  
    //        { (httpResult) in
```

crifan.com, 使用署名4.0国际(CC BY 4.0)协议发布 all right reserved, powered by
Gitbook最后更新: 2021-07-17 18:22:44

HTTP内部流程

其实内部都是通用的HTTP的逻辑：

- 发送HTTP的请求
- 接受到响应后
 - 解析并显示出对应的内容
 - 解析得到所需要的数据

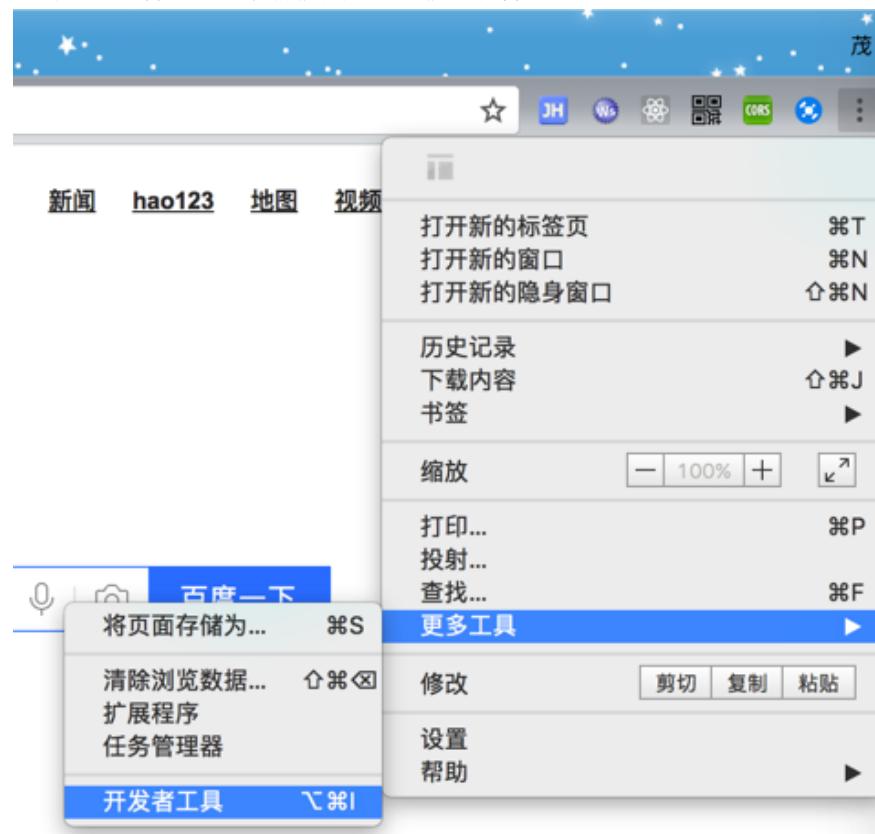
那如何才能看到内部的HTTP的请求和响应到底是什么样的呢？有很多方式可以实现查看内部到底发送了什么样的HTTP请求，和接收到了什么样的响应：

- 对于浏览器访问网页
 - 可以用 Chrome / Safari / Firefox 等浏览器自带的调试工具：[开发者工具](#)
 - 可以用其他工具去模拟
 - 比如：命令行工具 curl
- 对于写代码调用后台接口
 - 可以用postman去模拟和查看

下面详细介绍如何查看内部的具体流程：

用Chrome的开发者工具去查看

先去打开：右上角 三个点-》更多工具-》开发者工具



点击到Network列，然后再去访问地址 www.baidu.com

就可以看到会有一堆的内容列出来，点击第一个 www.baidu.com 的Header

即可看到，对应的Request和Response的信息：

The screenshot shows the Network tab in the Chrome DevTools. A request for `www.baidu.com` is selected. The Headers section is expanded, showing:

- Request URL: `https://www.baidu.com/`
- Request Method: GET
- Status Code: 200 OK
- Remote Address: 188.97.33.187:443
- Referrer Policy: no-referrer-when-downgrade

The Response Headers section is also visible, showing:

- Bdpagegetid: 1
- Bdpageid:47c000000909
- Bdbrand: 8
- Cache-Control: private
- Connection: Keep-Alive
- Content-Encoding: gzip
- Content-Type: text/html; charset=utf-8
- Ctx_all_baidu=1e6b44211f802386f9e10945507eba

通过点击Request Headers和Response Headers的view source可以看到未被解析之前的原始内容：

The screenshot shows the Network tab in the Chrome DevTools with the same request for `www.baidu.com`. The Request Headers section is expanded, showing:

```

X-Powered-By: PHP
X-UA-Compatible: IE=Edge, chrome=1

```

The Response Headers section is expanded, showing:

```

GET / HTTP/1.1
Host: www.baidu.com
Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6
Cookie: BAIDUID=6720BF4EAED8C8448EB6540DAF83BE56;FG=1;BIDUPSID=6720BF4EAED8C8448EB6540DAF83BE56;PSTM=1449798993;MCITY=-224%3A;BD_CK_SAM=1;PSINO=3;BD_HOME=0;H_PS_PSSID=1430_24568_21116_24880_20929;BD_UPN=123253

```

HTTP工具

The screenshot shows the Network tab in the Chrome DevTools. It lists 19 requests made to `www.baidu.com`. The first request, for `/`, has its response headers highlighted with a red box. The headers are as follows:

```
HTTP/1.1 200 OK
Bdpagetype: 1
Bdqid: 0xd4bdc47c00000909
Bduserid: 0
Cache-Control: private
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Cxy_all: baidu+1ea6be4d211f802306
f9e10940507eba
Date: Mon, 13 Nov 2017 01:32:41 GMT
Expires: Mon, 13 Nov 2017 01:32:06 GMT
Server: BWS/1.1
Set-Cookie: BDSVRTM=0; path=/
Set-Cookie: BD_HOME=0; path=/
Set-Cookie: H_PS_PSSID=1430_24568_2116_24880_20929; path=/; domain=.baidu.com
Strict-Transport-Security: max-age=172800
Vary: Accept-Encoding
X-Powered-By: PHP
X-UA-Compatible: IE=Edge,chrome=1
Transfer-Encoding: chunked
```

对应的原始内容为：

Request请求：

```
GET / HTTP/1.1
Host: www.baidu.com
Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6
Cookie: BAIDUID=6720BF4EAED8C8448EB6540DAF83BE56;FG=1; BIDUPSID=6720BF4EAED8C8
```

Response响应：

```
HTTP/1.1 200 OK
Bdpagetype: 1
Bdqid: 0xd4bdc47c00000909
Bduserid: 0
Cache-Control: private
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Cxy_all: baidu+1ea6be4d211f802306f9e10940507eba
Date: Mon, 13 Nov 2017 01:32:41 GMT
Expires: Mon, 13 Nov 2017 01:32:06 GMT
Server: BWS/1.1
Set-Cookie: BDSVRTM=0; path=/
Set-Cookie: BD_HOME=0; path=/
Set-Cookie: H_PS_PSSID=1430_24568_21116_24880_20929; path=/; domain=.baidu.com
Strict-Transport-Security: max-age=172800
Vary: Accept-Encoding
X-Powered-By: PHP
X-UA-Compatible: IE=Edge,chrome=1
Transfer-Encoding: chunked
```

用curl工具去模拟浏览器访问百度的过程

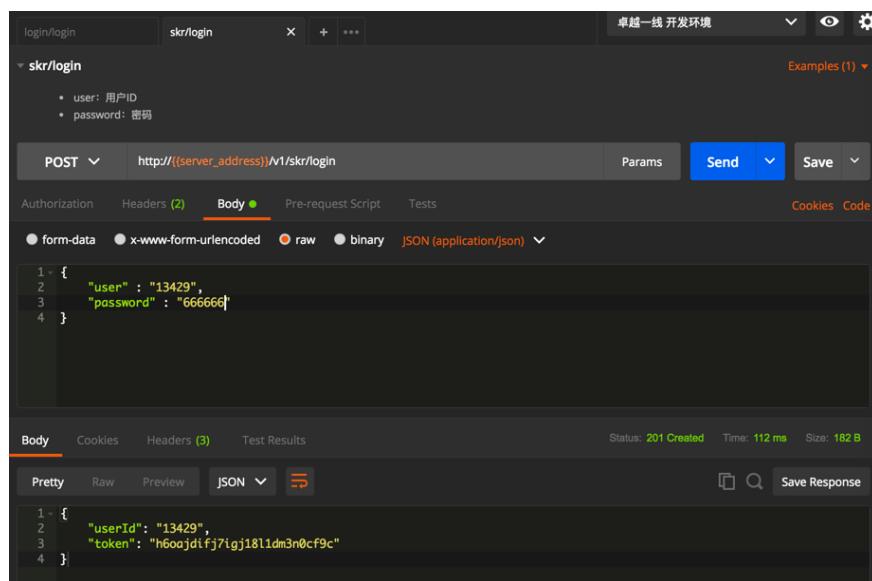
在命令行中输入：

```
curl -v www.baidu.com
```

返回的结果是：

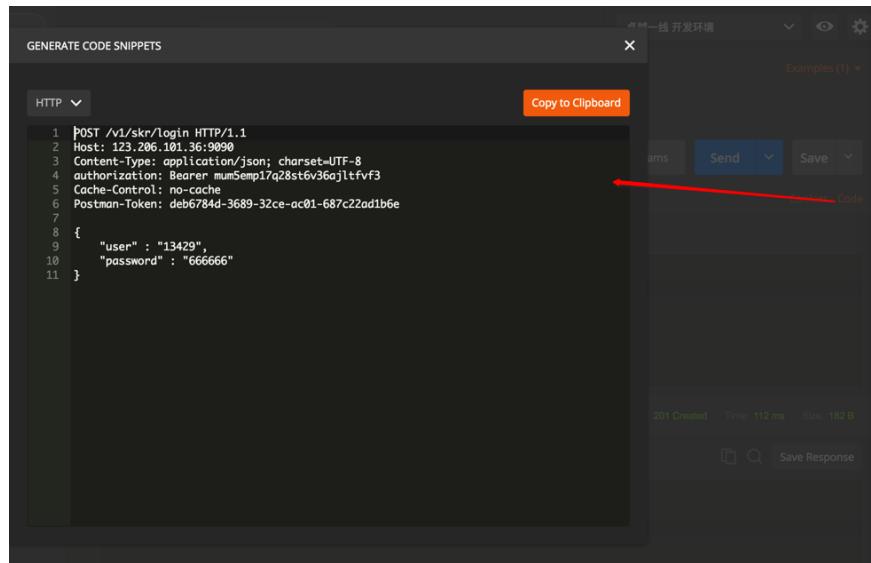
```
→ ~ curl -v www.baidu.com
* Rebuilt URL to: www.baidu.com/
*   Trying 180.97.33.107...
* TCP_NODELAY set
* Connected to www.baidu.com (180.97.33.107) port 80 (#0)
> GET / HTTP/1.1
> Host: www.baidu.com
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: bfe/1.0.8.18
< Date: Mon, 13 Nov 2017 01:35:55 GMT
< Content-Type: text/html
< Content-Length: 2381
< Last-Modified: Mon, 23 Jan 2017 13:28:24 GMT
< Connection: Keep-Alive
< ETag: "588604f8-94d"
< Cache-Control: private, no-cache, no-store, proxy-revalidate, no-transform
< Pragma: no-cache
< Set-Cookie: BDORZ=27315; max-age=86400; domain=.baidu.com; path=/
< Accept-Ranges: bytes
<
<!DOCTYPE html>
<!STATUS OK--><html> <head><meta http-equiv=content-type content=text/html>
* Connection #0 to host www.baidu.com left intact
```

用 postman 去模拟前面的调用后台接口的登录过程



HTTP工具

其中内部发送的HTTP请求，可以通过Code查看：



```
POST /v1/skr/login HTTP/1.1
Host: 123.206.101.36:9090
Content-Type: application/json; charset=UTF-8
authorization: Bearer mum5emp17q28st6v36ajltfvf3
Cache-Control: no-cache
Postman-Token: deb6784d-3689-32ce-ac01-687c22ad1b6e

{
  "user" : "13429",
  "password" : "666666"
}
```

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新：2021-07-17 17:46:47

HTTP基本逻辑总结

上述的所有用浏览器访问网页，用代码调用后台接口去登录等等，其内部的HTTP的基本逻辑和核心流程都是一样的：

(客户端) 发送请求Request

HTTP协议规定request的格式：



此处：

```
GET / HTTP/1.1
...
User-Agent: curl/7.54.0
Accept: */*
```

其中：

- GET : 方法类型，HTTP/1.1表示HTTP协议的版本是1.1
 - 使用第三方库时，也需要制定对应的GET / POST / ...。
 - 现在多数的HTTP协议都是使用的1.1
- User-Agent : 用户代理，表示自己的客户端的类型是curl
 - 使用第三方库时，一般不需要设置此参数
 - 第三方库会自动帮你设置
 - 如果是浏览器的话，不同浏览器对应不同的预设值
 - 比如上面的Chrome此处是：
 - User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/61.0.3163.100 Safari/537.36
- Accept : 声明自己接收的信息的类型为/，表示所有类型内容都运行
 - 如果调用服务器接口时，往往设置为：application/json

获得 (服务器返回的) 响应Response

HTTP协议规定的响应的格式：

HTTP-Version	Status-Code (状态码)	Reason-Phrase	CRLF(换行)
消息报头			
空行（只有CRLF的行）			
响应正文			

举例：

HTTP/1.1 200 OK
Date: Sat, 31 Dec 2005 23:59:59 GMT
Content-Type: text/html; charset=ISO-8859-1
Content-Length: 122

<html>
<head>
<title>Wrox Homepage</title>
</head>
<body>
<!-- body goes here -->
</body>
</html>

状态行
消息报头
空行
下面的就是响应正文了

此处的：

```
< HTTP/1.1 200 OK
...
< Content-Type: text/html
< Content-Length: 2381
...
<
<!DOCTYPE html>
<!--STATUS OK--><html> <head><meta http-equiv=content-type content=text/html;c
* Connection #0 to host www.baidu.com left intact
```

其中：

- HTTP/1.1 200 OK：表示1.1版本的HTTP协议，返回状态为200，表示正常
 - 其他典型的，不正常的，有问题的状态有很多，比如404找不到，500服务器内部错误等等
- Content-Type: text/html：内容的类型为html
- Content-Length: 2381：内容的长度为2381字节
- <!DOCTYPE html>...：html的内容，浏览器加载此内容后，即可显示出你所看到的网页

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新：2021-07-17 17:44:48

HTTP详解

接下来详细解释HTTP的结构和组成。

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新: 2021-07-17 17:44:48

HTTP的典型结构

Request

- URL
- Method
 - GET
 - 获取信息
 - 不会更改服务器内部的数据
 - POST
 - 向服务器发送数据:
 - 创建xxx
 - 更新xxx
 - 准确的说，其实应该用PUT或UPDATE
 - 会更改服务器内部的数据
 - PUT
 - PATCH
 - DELETE
 - HEAD: 获取资源的元数据
 - OPTIONS: 获取信息，关于资源的哪些属性是客户端可以改变的
- Query Parameters
 - for GET
 - query string = qs = query parameters = params = query component
 - /xxx/xxx?key1=value1&key2=value2
 - for POST
 - body
 - JSON
 - url encoded
 - key1=value1&key2=value2
- Headers
 - Content-Type
 - application/json
 - application/json; charset=UTF-8
 - Accept
 - application/json
 - Authorization
 - Bearer 725a7c44b76c45ab95152bcee014ae6e_1
- Other
 - Cookie
 - LocalStorage
 - SessionStorage

响应Response

- Status

- Informational - 1xx
 - 100 CONTINUE
- Successful - 2xx
 - 200 OK
- Redirection - 3xx
 - 301 MOVED_PERMANENTLY
- Client Error - 4xx
 - 400 BAD_REQUEST
 - 401 UNAUTHORIZED
 - 403 FORBIDDEN
 - 404 NOT_FOUND
- Server Error - 5xx
 - 500 INTERNAL_SERVER_ERROR
- Body
 - raw
 - to json
- Other
 - Cookie

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新: 2021-07-17 17:48:34

HTTP的Header头

HTTP的Header包括两类：

- 请求头=Request Header
- 响应头=Response Header

HTTP的Header方面的逻辑总结

既然是 Client问，Sever答 的过程，那么基本逻辑就是：

- Client告诉服务器端，自己的Request请求，能够接受的各种信息是什么类型的
 - 所以Request中有很多Accept方面的请求头
 - Accept: 能接受（返回）哪些类型
 - 格式：type/sub-type
 - /* 表示任何类型
 - 举例：
 - Accept: application/json
 - Accept: text/plain
 - Accept: text/html
 - Accept: image/jpeg
 - Accept: application/msword
 - Accept: image/png
 - Accept: application/pdf
 - Accept-Charset
 - 能接受的字符集
 - Accept-Encoding
 - 能接受的（编码）压缩类型
 - 比如：gzip, deflate
 - Accept-Language
 - 能接受的语言类型
 - 以及其他一些额外的请求和希望
 - User-Agent:告诉你我是哪种浏览器（所处的操作系统是什么类型）等信息
 - 举例：
 - User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN;rv:1.8.1.14) Gecko/20080404 Firefox/2.0.0.14
 - User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36
 - User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_1) AppleWebKit/604.3.5 (KHTML, like Gecko) Version/11.0.1 Safari/604.3.5
 - Referer: 告诉服务器是从（参考）哪个链接去访问的
 - 举例：Referer: <https://www.google.co.uk/>
 - Keep-Alive: 希望服务器保持此次连接（多长时间）
 - Cache-Control: 表示是否使用缓存

- Connection: 完成本次请求的响应后, 是否断开连接, 是否要等待本次连接的后续请求了
 - Connection: close
 - Connection: keepalive
- Sever返回响应, 告诉客户端, 自己的响应中数据都是什么类型的
 - 所以Response中有很多Content方面的字段, 表示服务器返回的内容的各种类型说明
 - Content-Encoding: 此响应中使用了什么压缩方法 (gzip, deflate) 压缩响应中的对象
 - 表示客户端的你用什么对应的方法去解压缩, 才能得到数据的原文
 - 举例: Content-Encoding: gzip
 - Content-Language: 用了什么语言
 - Content-Length: 数据内容的长度, 单位: 字节
 - 举例: Content-Length: 2684
 - Content-Type: 自己返回的数据是什么格式的
 - 和Request中的Accept对应
 - 举例: Content-Type: application/json
 - Connection: 和Request对应
 - Connection: close
 - Connection: keepalive
 - 以及其他一些表示当前信息的情况的
 - Location: 一个url地址, 表示你想要的资源被转移到别处了,
 - 典型的是发生了302表示自动跳转, 告诉你要的东西, 换了地址了

常见Header举例

- Request Header请求头
 - Accept
 - 代码调用API时:
 - Accept: application/json
 - Accept: application/json; charset=UTF-8
 - 浏览器访问网页时:
 - Accept:

text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
 - Content-Type
 - 代码调用API时:
 - Content-Type:application/json
 - Content-Type:application/json; charset=UTF-8
 - Content-Type:application/x-www-form-urlencoded
 - Authorization
 - 代码调用API时:
 - Authorization: Bearer 6k8p8ucshobav531lftkb2f1bv
 - User-Agent
 - 浏览器访问网页时:
 - User-Agent: Mozilla/5.0(Linux;X11)

- User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_1)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/61.0.3163.100 Safari/537.36
- Accept-Language
 - 浏览器访问网页时:
 - Accept-Language: zh-CN,zh;q=0.8,en;q=0.6
- Accept-Encoding
 - 浏览器访问网页时:
 - Accept-Encoding: gzip, deflate, br
- Response Header响应头
 - Content-Type
 - 代码调用API时:
 - Content-Type:application/json
 - 浏览器访问网页时:
 - Content-Type: text/html; charset=utf-8
 - Connection
 - 浏览器访问网页时:
 - Connection: Keep-Alive
 - Content-Encoding
 - 浏览器访问网页时:
 - Content-Encoding: gzip

更全更完整的Header

Request的Header

Header	解释	示例
Accept	指定客户端能够接收的内容类型	Accept:text/plain,text/html
Accept-Charset	浏览器可以接受的字符编码集。	Accept-Charset:iso-8859-5
Accept-Encoding	指定浏览器可以支持的web服务器返回内容压缩编码类型。	Accept-Encoding:compress,gzip
Accept-Language	浏览器可接受的语言	Accept-Language:en,zh
Accept-Ranges	可以请求网页实体的一个或者多个子范围字段	Accept-Ranges:bytes
Authorization	HTTP授权的授权证书	Authorization:Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
Cache-Control	指定请求和响应遵循的缓存机制	Cache-Control:no-cache
Connection	表示是否需要持久连接。 (HTTP 1.1默认进行持久连接)	Connection:close

Header	解释	示例
Cookie	HTTP请求发送时，会把保存在该请求域名下的所有cookie值一起发送给web服务器。	Cookie:\$Version=1;Skin=new;
Content-Length	请求的内容长度	Content-Length:348
Content-Type	请求的与实体对应的MIME信息	Content-Type:application/x-www-form-urlencoded
Date	请求发送的日期和时间	Date:Tue, 15 Nov 2010 08:12:31 GMT
Expect	请求的特定的服务器行为	Expect:100-continue
From	发出请求的用户的Email	From: user@email.com
Host	指定请求的服务器的域名和端口号	Host: www.zcmhi.com
If-Match	只有请求内容与实体相匹配才有效	If-Match: "737060cd8c284d8af7ad3082f209582d"

Header	解释	示例
If-Modified-Since	如果请求的部分在指定时间之后被修改则请求成功, 未被修改则返回304代码	If-Modified-Since:Sat,29 Oct 2010 19:43:31 GMT
If-None-Match	如果内容未改变返回304代码, 参数为服务器先前发送的Etag, 与服务器回应的Etag比较判断是否改变	If-None-Match: "737060cd8c284d8af7ad3082f209582d"
If-Range	如果实体未改变, 服务器发送客户端丢失的部分, 否则发送整个实体。参数也为Etag	If-Range: "737060cd8c284d8af7ad3082f209582d"
If-Unmodified-Since	只在实体在指定时间之后未被修改才请求成功	If-Unmodified-Since:Sat,29 Oct 2010 19:43:31 GMT

Header	解释	示例
Max-Forwards	限制信息通过代理和网关传送的时间	Max-Forwards:10
Pragma	用来包含实现特定的指令	Pragma:no-cache
Proxy-Authorization	连接到代理的授权证书	Proxy-Authorization:Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
Range	只请求实体的一部分，指定范围	Range:bytes=500-999
Referer	先前网页的地址，当前请求网页紧随其后，即来路	Referer:http://
TE	客户端愿意接受的传输编码，并通知服务器接受尾加头信息	TE:trailers,deflate;q=0.5
Upgrade	向服务器指定某种传输协议以便服务器进行转换（如果支持）	Upgrade:HTTP/2.0,SHTTP/1.3,IRC/6.9,RTA/x11

Header	解释	示例
User-Agent	User-Agent的内容包含发出请求的用户信息	User-Agent:Mozilla/5.0(Linux;X11)
Via	通知中间网关或代理服务器地址，通信协议	Via:1.0 fred,1.1 nowhere.com (Apache/1.1)
Warning	关于消息实体的警告信息	Warn:199 Miscellaneous warning

Response的Header

Header	解释	示例
Accept-Ranges	表明服务器是否支持指定范围请求及哪种类型的分段请求	Accept-Ranges: bytes
Age	从原始服务器到代理缓存形成的估算时间(以秒计, 非负)	Age: 12
Allow	对某网络资源的有效请求行为,不允许则返回405	Allow: GET, HEAD
Cache-Control	告诉所有的缓存机制是否可以缓存及哪种类型	Cache-Control: no-cache
Content-Encoding	web服务器支持的返回内容压缩编码类型。	Content-Encoding: gzip
Content-Language	响应体的语言	Content-Language: en,zh
Content-Length	响应体的长度	Content-Length: 348

Header	解释	示例
Content-Location	请求资源可替代的备用的另一地址	Content-Location: /index.htm
Content-MD5	返回资源的MD5校验值	Content-MD5: Q2hIY2sgSW50ZWdyaXR5IQ==
Content-Range	在整个返回体中本部分的字节位置	Content-Range: bytes 21010-47021/47022
Content-Type	返回内容的MIME类型	Content-Type: text/html; charset=utf-8
Date	原始服务器消息发出的时间	Date: Tue, 15 Nov 2010 08:12:31 GMT
ETag	请求变量的实体标签的当前值	ETag: "737060cd8c284d8af7ad3082f209582d"
Expires	响应过期的日期和时间	Expires: Thu, 01 Dec 2010 16:00:00 GMT
Last-Modified	请求资源的最后修改时间	Last-Modified: Tue, 15 Nov 2010 12:45:26 GMT
Location	用来重定向接收方到非请求URL的位置来完成请求或标识新的资源	Location: http://www.zcmhi.com/archives/94.html

Header	解释	示例
Pragma	包括实现特定的指令，它可应用到响应链上的任何接收方	Pragma: no-cache
Proxy-Authenticate	它指出认证方案和可应用到代理的该URL上的参数	Proxy-Authenticate: Basic
refresh	应用于重定向或一个新的资源被创造，在5秒之后重定向（由网景提出，被大部分浏览器支持）	Refresh: 5; url= http://www.zcmhi.com/archives/94.html
Retry-After	如果实体暂时不可取，通知客户端在指定时间之后再次尝试	Retry-After: 120
Server	web服务器软件名称	Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
Set-Cookie	设置HttpCookie	Set-Cookie: UserID=JohnDoe; Max-Age=3600; Version=1

Header	解释	示例
Trailer	指出头域在分块传输编码的尾部存在	Trailer: Max-Forwards
Transfer-Encoding	文件传输编码	Transfer-Encoding:chunked
Vary	告诉下游代理是使用缓存响应还是从原始服务器请求	Vary: *
Via	告知代理客户端响应是通过哪里发送的	Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1)
Warning	警告实体可能存在的问题	Warning: 199 Miscellaneous warning
WWW-Authenticate	表明客户端请求实体应该使用的授权方案	WWW-Authenticate: Basic

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新: 2021-07-17 17:44:48

HTTP的请求参数和参数编码

下面介绍HTTP的请求参数 Request Parameters 和请求参数的编码 Request Parameters Encoding

GET的请求的参数： query string

关于Http的GET的请求参数，也有别的叫法：

- 请求参数 = request parameter
- = 查询 = query
- = 查询字符串 = query string
- = 请求参数 = request parameters
- = 参数序列化 = param serialization = parameter serialization

转换后放到Url中的，是如下形式：

```
?key1=value1  
?key1=value1&key2=value2
```

典型的是：

GET：参数想要放在url中以 ?key1=value1&key2=value2 的形式

则有两种做法：

- 自己把参数组合成对应的格式，放到url中
- 把参数（字典，对象等）调用url encode函数去编码，生成对应的格式（再放到url中）

GET的一般特点

- GET 请求有长度限制
 - 所以query string一般被限制在1024个字节
 - 超过限制则后台无法解析参数
- GET 请求可被缓存
- GET 请求保留在浏览器历史记录中
- GET 请求可被收藏为书签
- GET 请求不应用于处理敏感数据时使用
- GET 请求只应当用于取回数据

处理查询参数的库

JS的qs

- js中http网络库：axios
 - 其中有，处理查询参数的： qs
 - <https://www.npmjs.com/package/qs>
 - 当需要处理get的query string时，就可以利用其中的Qs子库去处理

```
qs.stringify(data)
```

- 实现：把（一般是字典类型==JSON类型的）变量，转换，字符串换，序列化，为 `&key1=value1` 之类的形式

关于别名：`参数序列化`

其中官网demo中：

```
// `paramsSerializer` is an optional function in charge of serializing `params`  
// (e.g. https://www.npmjs.com/package/qs, http://api.jquery.com/jquery.param/  
paramsSerializer: function(params) {  
    return qs.stringify(params, {arrayFormat: 'brackets'})  
},
```

也说明了：

此处的 `stringify` = 字符串化，也叫做：`paramsSerializer` = `参数序列化`

举例：

某js项目中用到了 `stringify`：

```
import { stringify } from 'qs';  
  
export async function queryRule(params) {  
    return request(`/api/rule?${stringify(params)}`);  
}
```

POST的请求的参数：`post body`

典型的是：把一堆的参数，放到 `post` 的 `body` 中，格式一般都是 `json` 格式

则一般也有两种做法：

- 自己把参数（对象，字典等），转换为对应的json字符串
- 调用库提供的方法去encode你的参数对象为json

其他一些解释：

POST的一般特点

- POST 请求不会被缓存
- POST 请求不会保留在浏览器历史记录中
- POST 不能被收藏为书签
- POST 请求对数据长度没有要求

GET和POST的请求的参数的编码

举例：

**http库axios中create时支持的config中的
paramsSerializer**

就支持利用其他序列化的库，比如：

- Javascript中的：
 - qs库
 - <https://www.npmjs.com/package/qs>
 - 举例：
 - `Qs.stringify(params, {arrayFormat: 'brackets'})`

Alamofire中，对于get的参数支持url encoding，对于post支持json encoding

所以代码可以写成：

```

var paramEncoding:ParameterEncoding = JSONEncoding.default
if (httpMethod == .get) {
    paramEncoding = URLEncoding.default
}

let curHttpReq = Alamofire.request(url, method: httpMethod, parameters: parameters, encoding: paramEncoding, headers: curHeaders)
  
```

```

35     var curHeaders:[String:String] = ["Content-Type" : "application/json; charset=UTF-8"]
36
37     for eachKey in curHeader.keys {
38         curHeaders[eachKey] = curHeader[eachKey]!
39     }
40
41     var mergedExtraPara = Dictionary<String, AnyObject>()
42     mergedExtraPara["httpMethod"] = httpMethod.rawValue as AnyObject?
43     mergedExtraPara["url"] = url as AnyObject?
44     mergedExtraPara["parameters"] = parameters as AnyObject?
45     mergedExtraPara["headers"] = headers as AnyObject?
46     gLog.debug("currentHeaders=\\" + curHeaders + "\\")
47
48
49     if paramEncoding == ParameterEncoding.JSONEncoding.default {
50         if (httpMethod == .get) {
51             paramEncoding = URLEncoding.default
52         }
53
54     //     let curHttpReq = Alamofire.request(url, method: httpMethod, parameters: parameters, encoding: JSONEncoding.default, headers: curHeaders)
55     let curHttpReq = Alamofire.request(url, method: httpMethod, parameters: parameters, encoding: paramEncoding, headers: curHeaders)
56     gLog.verbose("curHttpReq=\\" + curHttpReq + "\\")
57
58     //if (curHttpReq){
59     //    getHttpRequestJson(httpRequest: curHttpReq, mergedAllPara: mergedExtraPara, requestType: requestType, respJsonHandler: {
60     //        (httpResult) in
61     //        gLog.verbose("respResult.debugDescription=\\" + respResult.debugDescription + "\\, mergedAllPara=\\" + mergedAllPara + "\\")
62     //        respJsonHandler!(respJsonHandle!)
63     //        //com.apple.root.background-qos [CrifanLibHttp.swift:145]
64     //    })
65   
```

然后从外部调用时，对于get的url参数，可以直接传递：

get中的url的参数的对象，让Alamofire内部去利用URLEncoding转换为对应的?
key1=value1&key2=value2的格式：

```

var parameters = [String : AnyObject]()
parameters = [
    "planId":self.visitId as AnyObject,
    "name":self.missionInfo.missionInfo.name as AnyObject,
    "pics":pics as AnyObject,
    "publisher":gCurUserItem.userInfo.pkEmpno as AnyObject,
    "resultDescription":self.missionInfo.missionInfo.resultDescription as AnyObject,
    "finish":finishInt as AnyObject,
    "planProblem":planProblem as AnyObject
]

getUrlRespJson_async(
    httpMethod: HTTPMethod,
    url: url,
    parameters: parameters,
    respJsonHandle: { [weak self] (response) in
  
```

```

556
557     if checkKey() {
558         self.pleaseWait()
559         let parameters = self.getParameters()
560         gLog.debug("\(self.getParameters())")
561         getUrlRespJson_async(
562             httpMethod: HTTPMethod,
563             url: url,
564             parameters: parameters,
565             respJsonHandle: { [weak self] (response) in
566                 self?.clearAllNotice()
567                 if response.isSuccess {
568                     gLog.debug(response.successValue)
569

```

另外，对应的对于POST来说，把json对象转换为json字符串的例子：

```

let parameters = [
    "id":self.missionInfo.missionProblem.id as AnyObject,
    "checkPlanId":self.checkPlanId as AnyObject,
    "checkResult":self.resultCell.textView.text as AnyObject,
    "pass":self.pass as AnyObject ,
]

] as [String : AnyObject]?

getUrlRespJson_async(
    httpMethod: .post,
    url: ServerApi.getProblemIdResult(visitId: self.visitId,problemId: self.m
parameters: parameters ,
respJsonHandle: { [weak self] (response) in

565         self.pleaseWait()
566         let parameters = [
567             "id":self.missionInfo.missionProblem.id as AnyObject,
568             "checkPlanId":self.checkPlanId as AnyObject,
569             "checkResult":self.resultCell.textView.text as AnyObject,
570             "pass":self.pass as AnyObject ,
571
572         ] as [String : AnyObject]?
573         //gLog.debug("\(self.getParameters())")
574         getUrlRespJson_async(
575             httpMethod: .post,
576             url: ServerApi.getProblemIdResult(visitId: self.visitId,problemId: self.missionInfo.missionProblem.id),
577             parameters: parameters ,
578             respJsonHandle: { [weak self] (response) in
579                 self?.clearAllNotice()

```

则内部的Alamofire就会把该参数对象，通过JSONEncoding转换为json字符串了。

举例：

python中，get中url参数可以利用 urllib.urlencode（或 urllib.quote_plus）去把dict字典转换为key=value的形式：

```

import urllib
params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
url = "http://www.musi-cal.com/cgi-bin/query?%s"; % params

```

关于HTTP参数编码的常见场景和问题

空格应该被encode编码为 + 还是 %20 ?

如果你接触Python的url的encode比较多，可能会注意到一个现象：

好像空格有时候被编码为+，有时候被编码为%20，到底哪个才对？

Percent Encode也被叫做URL Encode

Percent Encode指的是，一些字符，在被(url) encode后，往往都是变成%xx
比如：

内部对应着HTTP请求时，类型被设置为：application/x-www-form-urlencoded

表格太长，分两个

!	#	\$	&	'	()	*	-
%21	%23	%24	%26	%27	%28	%29	%2A	%

和：

,	/	:	;	=	?	@	[]
%2C	%2F	%3A	%3B	%3D	%3F	%40	%5B	

而其他常见的字符被编码后的效果是：

newline	space	"	%	-	.	<	>
%0A or %0D or %0D%0A	%20	%22	%25	%2D	%2E	%3C	%3

和

\	^	-	'	{		}	~
%5C	%5E	%5F	%60	%7B	%7C	%7D	%7E

即：

空格正常情况下被url encode=percent encode，的结果是%20

但是之所以有时候会看到+是因为：

历史上，最早的网页技术中，在表单form被（通过HTTP的GET / POST等请求，或者是邮件发送）提交时，键key和值value，都是被percent encode=url encode的

对应着类型是：application/x-www-form-urlencoded

但是后来有些变种的处理，其中就包括把空格space编码为+（而不是%20）

空格被编码的逻辑的历史

- 空格被url encode=percent encode，应该是：%20
- 而之前历史上有些变种的处理，会编码为：+

而Python中对于url encode相关的函数有3种，对应的效果分别如下：

```

#!/usr/bin/python
# -*- coding: utf-8 -*-

import urllib

paraValue = "Crifan Li"
queryPara = { "name" : paraValue }
urlEncodedQueryPara = urllib.urlencode(queryPara)
print "urlEncodedQueryPara=%s"%(urlEncodedQueryPara) # urlEncodedQueryPara=name=Crifan%20Li
quotedValue = urllib.quote(paraValue) # quotedValue=Crifan%20Li
quotedPlusValue = urllib.quote_plus(paraValue) # quotedPlusValue=Crifan+Li
print "quotedValue=%s,quotedPlusValue=%s"%(quotedValue, quotedPlusValue)

```

空格被编码

- `urllib.urlencode` 编码（字典中的）value
 - `urllib.quote_plus` 编码字符串：空格编码为`+`
- `urllib.quote` 编码字符串：空格编码为`%20`

在url地址里包含中文时的编码显示和内部逻辑

背景：

如果url地址中有非普通的ASCII字符串，理论上都是会被编码后，所以你看到的如果url地址中有中文，实际上打开都是`%xx`之类的地址
中文字符串和其他字符串编码逻辑是一样的。

对于中文类字符被url encode，有很多在线网站可以帮你实现，比如：

- <http://tool.chinaz.com/tools/urlencode.aspx>
- <http://tool.oschina.net/encode?type=4>

输入中文李茂就可以被（UTF-8）编码为：`%e6%9d%8e%e8%8c%82`

The screenshot shows a web-based URL encoder. At the top, there are tabs for 'Native/Unicode', 'Native/UTF-8', 'Native/ASCII', and 'URL转码'. The 'URL转码' tab is selected. Below it, there's a 'Url:' input field containing '李茂'. To the right, there are two radio button options: 'encodeURI' (selected) and 'encodeURIComponent'. A large blue button labeled 'URL编码 >' is highlighted with a red box. Below the input fields, there's a note: '当前位置: 站长工具 > UrlEncode编码/解码'. At the bottom, there are tabs for 'Unicode编码', 'UTF-8编码' (which is selected), 'URL编码/解码', 'Unis时间戳', and 'Ascii/Native编码互转'. The output area shows the encoded result: '%e6%9d%8e%e8%8c%82'. At the very bottom, there are buttons for 'utf-8', 'UrlEncode编码', 'UrlDecode解码', and '清空结果'.

提示

- 一般网页地址中的字符编码都用的是UTF-8
- 上面编码也都是采用的UTF-8编码得到的结果

而如果想要换成别的编码，比如另一种常见的中文编码GB2312，则编码出来的是另外的结果：

同样字符串的不同编码的效果

- 李茂 -» UTF-8 编码后 -» %e6%9d%8e%e8%8c%82
- 李茂 -» GB2312 编码后 -» %c0%ee%c3%af

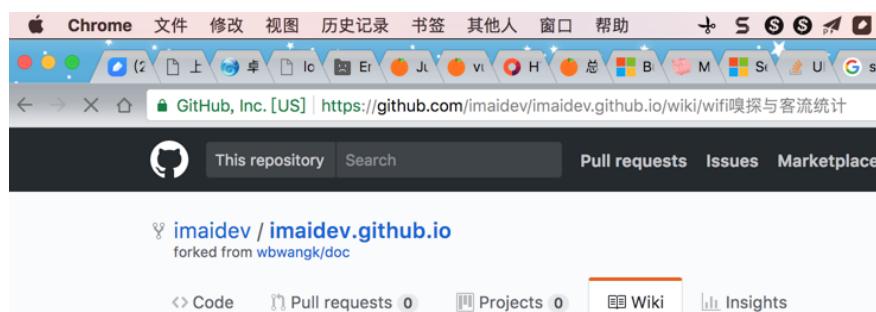
为何浏览器中的地址栏中的url地址可以看到有中文，而不是url encode之后的%xx？

而你在浏览器中看到的中文地址其实是浏览器帮你解码后的中文

真正的url地址是%xx形式的被（UTF-8）编码后的地址

比如你在Chrome（或其他浏览器）中看到的地址中有中文：

<https://github.com/imaidev/imaidev.github.io/wiki/wifi嗅探与客流统计>



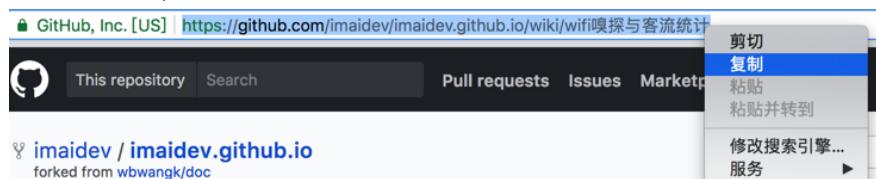
wifi嗅探与客流统计

王伟兵 edited this page on 10 Mar · 20 revisions

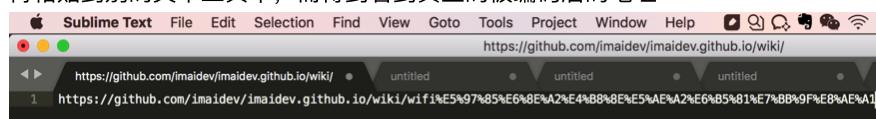
实际上内部真正的地址是：

<https://github.com/imaidev/imaidev.github.io/wiki/wifi%E5%97%85%E6%8E%A2%E4%B8%8E%E5%AE%A2%E6%B5%81%E7%BB%9F%E8%AE%A1>

-» 你可以通过，在地址栏中右键-» 复制



再粘贴到别的文本工具中，而得到看到真正的被编码后的地址：



类似的，把该地址去（利用前面说的在线网站帮忙）解码也可以得到同样的中文：

HTTP工具

http://tool.oschina.net/encode?type=4

在线编码转换 Feedback

Native/Unicode Native/UTF-8 Native/ASCII URL转码

Url:

encodeURI encodeURIComponent

URL编码 > < URL解码

编码结果: `https://github.com/lmaidev/lmaidev.github.io/wiki/wifi%E5%97%85%E6%8E%A2%E4%B8%8E%E5%AE%A2%E8%B5%81%E7%BB%9F%E8%AE%A1`



crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新: 2021-07-17 18:12:58

HTTP的响应的状态码

Http状态码 = HTTP Code

关于状态码的类别的含义的总体解释是：

状态码大类	表示的含义	客户端client要做的事	服务器端server要做的事
1xx	Informational 信息	啥都不用做，知道就好	告诉client，信息收到了，我后续会处理
2xx	Successful 成功	啥都不用做，知道就好	告诉client，请求已正确处理
3xx	Redirection 重定向	重新请求返回的新地址 -> 才能获取真正需要的数据	告诉client，你需要的内容，由于一些原因，比如地址已发生变化了，然后返回该内容的新地址
4xx	Client Error 客户端的错误	确保用正确的参数和信息正确，重新请求	告诉client，请求已正确处理
5xx	Server Error 服务器端的错误	(一般来说) 都无需啥操作 -> 往往需要服务器端改了bug后，重新发送请求	需要服务器Server端自己找到具体出了啥错 -> 往往是服务器端的代码的bug导致了出错

最常用的状态码及含义

- Successful - 2xx：成功类，行为被成功地接受、理解和采纳
 - 200 = OK
 - 服务器成功返回用户请求的数据
 - 注意：（部分人）为了简化处理，`POST` 创建成功后应该返回 201 的但也返回 200
- 400 = BAD REQUEST
 - 错误的请求
 - 注意：（部分人）往往为了简化处理，把属于 401 的 没有权限 和属于 403 的 有权限但是权限不够，往往都返回 404
- 404 = NOT FOUND
 - 找不到资源
 - 注意：有些做法是，把属于 401 或 403 的原因，假装找不到而返回 404
- 500 = INTERNAL SERVER ERROR
 - 服务器内部错误

- 最常见的原因是：服务器内部挂了
- 比如：你传递参数中有些参数是空，而导致后台代码无法解析，出现异常而崩溃

次常用的响应码及含义

- Informational - 1xx: 信息类，请求收到，继续处理
- Successful - 2xx: 成功类，行为被成功地接受、理解和采纳
 - 201 CREATED
 - 通过POST或PUT创建资源成功
 - 通过response header中会返回新创建的资源的url连接
 - response body可能有信息，也可能为空
 - 202 Accepted
 - 服务器接受了此请求，但是待会才会执行
 - 204 NO CONTENT
 - 资源创建成功，但是没有返回内容
 - 常用于DELETE或PUT操作的返回
- Redirection - 3xx: 重定向类，为了完成请求，必须进一步执行的动作
 - 301 Moved
 - 该资源的URI接口已经变了，变成别的接口了
 - response header中应该告诉新接口地址，比如：
 - URI: url地址
- Client Error - 4xx: 客户端错误类，请求包含语法错误或者请求无法实现
 - 401 UNAUTHORIZED
 - 没有权限访问该资源
 - 典型情况：用户没有登录，没有获得对应的access token而直接访问某资源
 - 403 FORBIDDEN
 - 禁止访问
 - 典型情况：虽然用户已登录，但是去更新/删除需要更高权限才能操作的资源
 - 405 METHOD NOT ALLOWED
 - 方法不允许
 - 举例：比如某个资源不允许PATCH操作，但是你调用PATCH操作
 - 406 - Not Acceptable
 - 无效的请求
 - 比如：请求中的部分参数是无效的
 - 409 CONFLICT
 - 有冲突
 - 举例：新建一个用户，但是主键手机号和之前已有用户冲突
 - 415 Unsupported Media Type
 - 典型情况：请求的header有问题
 - 422 Unprocessable Entity
 - Specify values for all required parameter fields
- Server Error - 5xx: 服务器错误类，服务器不能实现一种明显无效的请求
 - 502 BAD GATEWAY
 - 网管错误

- 503 SERVICE UNAVAILABLE
 - 无服务
 - 举例：比如服务器内部正在维护，暂时不提供服务。可以换个时间重试

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新：2021-07-17 18:01:45

HTTP的响应数据格式JSON

调用服务器API接口，常见返回内容格式是： JSON

http的response返回的内容，原始raw格式，都是字符串string，text的类型

如果想要把raw的text/string转换为json，则可以：

利用很多库自带的功能，把返回内容转换为JSON

iOS的Alamofire

```
Alamofire.request("https://httpbin.org/get").responseJSON { response in
    if let json = response.result.value {
        print("JSON: \(json)") // serialized json response
    }
}
```

Python的Requests

```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User", ...}'
>>> r.json()
{u'disk_usage': 368627, u'private_gists': 484, ...}
```

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新：2021-07-17 17:44:48

HTTP相关

接下来介绍HTTP相关的一些心得，工具，后台API设计等内容。

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新：2021-07-17 17:44:48

HTTP的工具和库

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新: 2021-07-17 21:24:29

HTTP工具

此处整理HTTP相关的库和工具：

Postman

可用于服务器端和其他移动端等， 调试接口数据返回是否正常

举例：比如用Postman去调试奶牛云的后台的登录接口：

更多内容详见另外的教程：[API开发利器：Postman](#)

Chrome的开发者工具Developer Tools

调试页面内容是否正常， 包括布局， 参数等等

详见：[【总结】浏览器中的开发人员工具（IE9的F12和Chrome的Ctrl+Shift+I） -网页分析的利器](#)

curl

模拟去请求服务器数据的命令行工具

HTTPie

后来从[jobbole/awesome-python-cn: Python资源大全中文版](#)中知道了还有
个 `HTTPie`

- Github: [jakubroztocil/httpie: Modern command line HTTP client](#)
- 官网: [HTTPie – command line HTTP client](#)
- 简介:
 - `HTTPie` is a command line HTTP client with an intuitive UI, JSON support, syntax highlighting, wget-like downloads, plugins, and more

看了下，比curl更好用，更方便。值得推荐。

Httpbin

Httpbin 是一个网站：

<http://httpbin.org/>

用于测试http各方面的内容。

首页：

The screenshot shows the homepage of httpbin.org. At the top, it says "httpbin.org 0.8.3" and "[Base URL: httpbin.org/]". Below that is a note: "A simple HTTP Request & Response Service." It includes instructions to "Run locally: \$ docker run -p 80:80 kennethreitz/httpbin", a link to "the developer's Website", and a link to "Send email to the developer". A dropdown menu "Schemes" is set to "HTTP". Below this is a navigation bar with links: "HTTP Methods", "Auth", "Status codes", "Request inspection", and "Response inspection", each with a "Test" button. A "显示菜单" (Show menu) link is at the bottom.

比如：

get

测试get方法：

<http://httpbin.org/get>

返回各种信息，包括headers

```
{  
  "args": {},  
  "headers": {  
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8  
    "Accept-Encoding": "gzip, deflate",  
    "Accept-Language": "zh-cn",  
    "Cookie": "_gauges_unique=1",  
    "Host": "httpbin.org",  
    "Upgrade-Insecure-Requests": "1",  
    "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_2) AppleWebKit/  
  },  
  "origin": "222.92.130.218, 222.92.130.218",  
  "url": "https://httpbin.org/get"  
}
```

```
{  
  "args": {},  
  "headers": {  
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",  
    "Accept-Encoding": "gzip, deflate",  
    "Accept-Language": "zh-cn",  
    "Cookie": "_gauges_unique=1",  
    "Host": "httpbin.org",  
    "Upgrade-Insecure-Requests": "1",  
    "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_2) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.0.2 Safari/605.1.15"  
  },  
  "origin": "222.92.130.218, 222.92.130.218",  
  "url": "https://httpbin.org/get"  
}
```

目的和作用：可以通过你的代码或工具中，去调用 <http://httpbin.org/get>，从返回的值中，确认你的接口调用GET时的各种参数，是否符合你的预期

ip

去测试当前IP：

HTTP工具

<http://httpbin.org/ip>

返回：

```
{  
    "origin": "222.92.130.218, 222.92.130.218"  
}
```



用途举例：

【已解决】PySpider中使用多贝云IP代理池实现每次请求IP都不同

中就利用了：

<http://httpbin.org/ip>

去获取当前（代理）的IP，来确认每次通过代理访问时，代理IP地址是否正确，代理是否真的生效了。

json

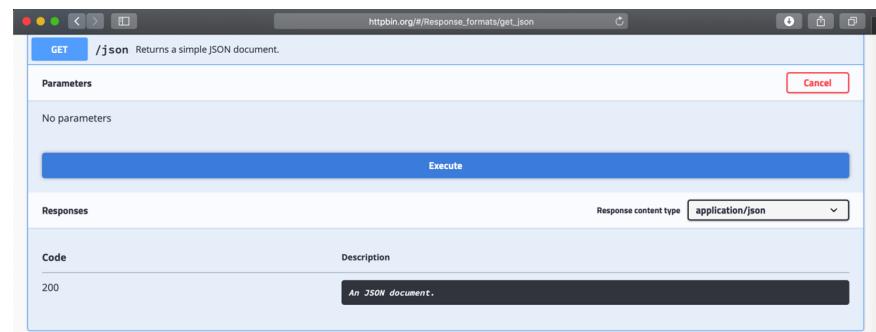
你想要测试，返回json数据，写代码去解析json

可以用：

<http://httpbin.org/json>

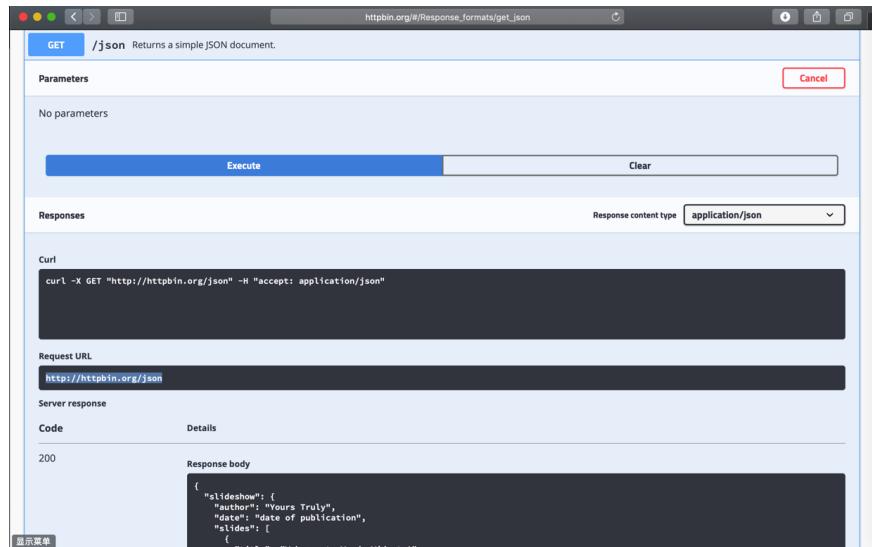
返回json数据供你解析和测试代码

同时，该网页上还有测试界面供你测试数据效果：



点击Execute后效果：

HTTP工具



crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新: 2021-07-17 21:17:43

HTTP库

此处整理不同编程语言的HTTP方面的库：

iOS 的 swift / Obj-C

- 第三方
 - [Alamofire](#)
- swift
 - [SwiftHTTP](#)

Python

- 内置
 - Python 3
 - <https://docs.python.org/3/library/urllib.html>
 - Python 2
 - urllib
 - <https://docs.python.org/2/library/urllib.html>
 - urllib2
 - <https://docs.python.org/2/library/urllib2.html>
- 第三方
 - [Requests](#)
- 详见独立教程
 - [Python/心得：http网络库](#)

JavaScript

- 好用的
 - [axios](#)
 - <https://github.com/axios/axios>
- 其他的
 - [Request](#)
 - <https://github.com/request/request>
 - [Web API的Fetch](#)
 - https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
 - [SuperAgent](#)
 - <https://github.com/visionmedia/superagent>
 - [fetch](#)
 - <https://github.com/github/fetch>
 - 举例

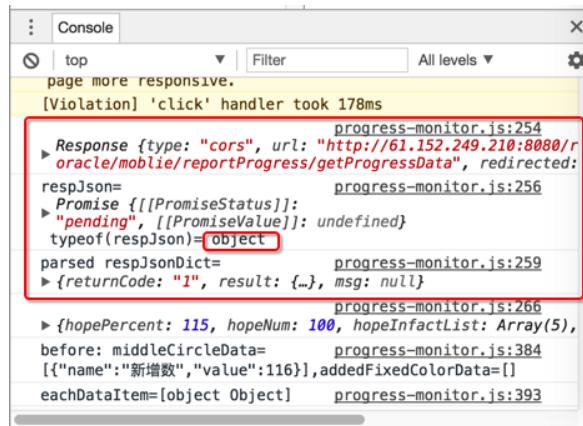
```

import 'whatwg-fetch';
fetch(
  'http://xxxxx/getProgressData',
{
  method : 'POST',
  headers : {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Accept': 'application/json'
  },
  body: 'currTime=2017-08-25&orgCode=SK316005'
}
)
.then((resp) => {
  console.log(resp);
  let respJson = resp.json(); //typeof(respJson)= object
  console.log('respJson=', respJson, 'typeof(respJson)=',
  return respJson;
})
.then((respJsonDict) => {
  console.log('parsed respJsonDict=', respJsonDict);
});

```

■ 效果

- 返回的是object对象，是Response类型，其中json()之后得到的是Promise对象，也就是可以直接拿来使用的json对象了，然后就可以进行后序数据处理了



C#

- [HttpWebRequest](#)
- crifan的lib的C#的http的部分
 - [第9章 crifanLib.cs之Http](#)

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新: 2021-07-17 21:25:55

HTTP的后台API设计

HTTP的后台API设计常用规范：**RESTful**

关于 RESTful，详见另外的教程：[HTTP后台端：RESTful API接口设计](#)

后台设计时记得要支持 **CORS**

设计和开发后台的 REST 的API时，记得要添加 CORS 支持，否则前端调用接口容易出现各种和 CORS 方面的问题，导致无法使用接口。

关于 CORS 的详见，见另外的教程：[CORS开发总结](#)

crifan.com，使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新：2021-07-17 17:44:48

HTTP相关心得

iOS (swift) 用Alamofire时，注意返回response时所处的线程

iOS (swift) 对于http库， Alamofire的响应返回默认处于主的UI线程

- > 而很多耗时的操作，是不建议，不允许，在主线程中操作的
- > 所以我后来自己在Alamofire基础上封装的库，是用 dispatchBackground_async 让 responseJSON 返回后运行在后台线程
- > 避免大量的操作堵塞了UI的响应
- > 又由于iOS中只能在主线程UI线程中操作UI元素
- > 所以此时如果直接在Alamofire返回的地方去操作UI元素则会报错： Terminating app due to uncaught exception NSInternalInconsistencyException reason Only run on the main thread 或 UI API called from background thread xx must be used from main thread only

解决办法是：

对于每个Alamofire的返回的response时，自己根据需要，加上异步主线程，在其中处理UI操作的部分。

示例代码：

```
func getUnreadCount(){
    let url = ServerApi.getUnreadCountUrl()

    getUrlRespJson_async(
        httpMethod: .get,
        url: url,
        parameters: nil,
        respJsonHandle: { (response) in
            if response.isSuccess {
                if let count = response.successValue["count"].int {
                    gLog.debug("count:\\" + (count.description))
                    dispatchMain_async ({
                        if count <= 0{
                            self.tabBarItem.badgeValue = nil
                        } else if count >= 100 {
                            self.tabBarItem.badgeValue = "99+"
                        } else {
                            self.tabBarItem.badgeValue = "\\" + (count.description)
                        }
                    })
                }
            } else if response.isFailure {
                dispatchMain_async ({
                    self.noticeInfo(response.failedMessage)
                })
            }
        })
}
```

在 `respJsonHandle` 中当 `isSuccess` 时，用 `dispatchMain_async` 确保处于主线程，然后才能去操作UI中的元素：`self.tabBarItem.badgeValue`

而对于上述函数详见：

- http函数[CrifanLibHttp.swift](#)中的 `getUrlRespJson_async`
- 线程函数[CrifanThread.swift](#)中的 `dispatchBackground_async`

相关帖子可参考：

[\[已解决\] swift中Alamofire的request的responseJSON不执行不返回了](#)

断点续传就是利用Http的Range实现的

对于断点续传功能的实现，就是利用了HTTP的头Range去实现的：

- 通过设置请求头Range可以指定每次从网路下载数据包的大小
- Range示例
 - bytes=0-499 从0到499的头500个字节
 - bytes=500-999 从500到999的第二个500字节
 - bytes=500- 从500字节以后的所有字节
- bytes=-500 最后500个字节
- bytes=500-599, 800-899 同时指定几个范围
- Range小结
 - - 用于分隔
 - 前面的数字表示起始字节数
 - 后面的数组表示截止字节数，没有表示到末尾
 - , 用于分组，可以一次指定多个Range，不过很少用

详见：

[\[已解决\] swift 下载时支持断点续传](#)

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新：2021-07-17 17:44:48

HTTP的文件上传

在HTTP相关开发期间，往往会涉及到上传文件，其中会涉及 `multipart/form-data`。

而官方的HTTP和 `multipart/form-data` 相关的协议有：

- [Returning Values from Forms: multipart/form-data](#)
- [Format of Internet Message Bodies](#)
- [Form submission](#)

对于用HTTP的 `multipart/form-data` 实现文件上传：

- 既有合适的如Postman等工具方便去调试
- 也有iOS中Alamofire等库很好的支持
- （如果清楚协议详情）也可以自己手动写代码拼接出 `multipart/form-data` 所需要的字段和内容去实现文件上传

下面就来详细介绍这方面的内容。

iOS中Alamofire支持文件上传的 `MultipartFormData`

可以基本实现文件上传的iOS的Alamofire的代码：

```

func uploadImageData(imageData:NSData){
    dispatchBackground_async({
        gLog.verbose("imageData.length=\\" + (imageData.length) + "\\")
        self.waitForCompletion()
    })
}

var curHeader = Dictionary<String, String>()
curHeader["authenticate"] = "token " + gCurUserItem.accessToken

gLog.verbose("curHeader=\\" + (curHeader) + "\\")

let curDate = NSDate()

let name = "file"
let filename = curDate.toString("yyyyMMdd_hhmss") + ".jpg"
gLog.verbose("name=\\" + (name) + ", filename=\\" + (filename) + "\\")
//uploadImageData > name=file, filename=20160625_085958.jpg

Alamofire.upload(
    .POST,
    ServerApi.uploadFileUrl(gCurUserItem.id),
    headers: curHeader,
    multipartFormData: {multipartFormData in
        multipartFormData.appendBodyPart(
            data: imageData,
            name: name,
            fileName: filename,
            mimeType: "image/jpeg"
        )
    },
    encodingCompletion: { encodingResult in
        gLog.verbose("encodingResult=\\" + (encodingResult) + "\\")
    }
)
encodingResult=Success($ curl -i \
-X POST \
-H "User-Agent: Sales App/com.qoro.QorosSalesApp (2016.6.24; OS ban ben 9."
-H "authenticate: token fmc2q2m25vre0htehsijmlur53" \
-H "Content-Type: multipart/form-data; boundary=alamofire.boundary.5fd5efc"
-H "Accept-Encoding: gzip;q=1.0, compress;q=0.5" \
-H "Accept-Language: en-US;q=1.0, zh-Hans-US;q=0.9" \
"http://192.168.1.102:8080/app/user/10000010/upload", false, nil)

encodingResult=Success($ curl -i \
-X POST \
-H "User-Agent: Sales App/com.qoro.QorosSalesApp (2016.6.24; OS ban ben 9."
-H "authenticate: token 4up3enr7nfd6c3rfei1ukcgvv" \
-H "Content-Type: multipart/form-data; boundary=alamofire.boundary.01fc8db"
-H "Accept-Encoding: gzip;q=1.0, compress;q=0.5" \
-H "Accept-Language: en-US;q=1.0, zh-Hans-US;q=0.9" \
"http://qapp.chinacloudapp.cn/app/user/10000010/upload", false, nil)
/*
switch encodingResult {
//case .Success(let uploadRequest, streamingFromDisk, streamFi
//    case .Success(let uploadRequest, _, _):
//        gLog.verbose("uploadRequest=\\" + (uploadRequest) + "\\")
//        gLog.verbose("uploadRequest=\\" + (uploadRequest) + "\\")
//        //upload=POST http://192.168.1.102:8080/app/user/10000010/
//        //uploadRequest=POST http://qapp.chinacloudapp.cn/app/user

        uploadRequest.responseJSON(completionHandler:
            { response in
                gLog.verbose("response=\\" + (response) + "\\")
                /*
                    response=SUCCESS: {
                        code = 200;
                        data = {
                            created = 1466850689624;
                            id = 1000001020160625103129624;
                */
            }
        )
}

```

```

        location = "<null>";
        name = "image.jpg";
        owner = "<null>";
        uploader = 10000010;
    };
    message = ok;
}

response=SUCCESS: {
    errors = (
        "The request body may not be null",
        "The request body may not be null"
    );
}

response=SUCCESS: {
    code = 200;
    data = {
        created = 1466859610413;
        id = 1000001020160625130010413;
        location = "<null>";
        name = "20160625_085958.jpg";
        owner = "<null>";
        uploader = 10000010;
    };
    message = ok;
}
*/
switch response.result {
case .Success(let value):
    let valueJson = JSON(value)
    gLog.verbose("\(valueJson)")
    /*
    {
        "message" : "ok",
        "code" : 200,
        "data" : {
            "name" : "image.jpg",
            "id" : "1000001020160625103129624",
            "uploader" : 10000010,
            "location" : null,
            "created" : 1466850689624,
            "owner" : null
        }
    }
    */
    let statusCode = valueJson["code"].int ?? 0
    if statusCode == 200 {
        let dataObj = valueJson["data"].object
        let dataJson:JSON = JSON(dataObj)
        self.uploadAttachmentsHandler(Alamofire.Re
        gLog.verbose("dataJson=\(dataJson)")

    } else {
        self.noticeError("上传附件出错", autoClear:
            true)
        let message = valueJson["message"].string
        let messageStrArr = message.splitToStrArr(
            "\n")
        var messageStr:String = message
        var subCode:Int = 0

        if messageStrArr.count > 1 {
            messageStr = messageStrArr[0]
            subCode = Int(messageStrArr[1]) ?? 0
        }

        let error:NSError = NSError(domain: HttpEr
            "message" : messageStr,

```

```
        "code"      : statusCode,
        "subCode"   : subCode,
    ])

        self.uploadAttachmentsHandler(Alamofire.Re
    }
    case .Failure(let error):
        gLog.verbose("error=\\" + (error) + \\
        self.uploadAttachmentsHandler(Alamofire.Result
    }

    })
}

case .Failure(let encodingError):
    gLog.error("上传附件失败\\(" + encodingError + \\
}
}

})
}
```

详见：

[【已解决】Alamofire中multipartFormData中的name和filename到底应该是什么值](#)

Postman中可以方便的调试文件上传

当然，如果后台设计的不是很规范，则可能导致，即使传入了标准的参数设置：

- POST xxx/xxx/upload
- Headers
 - Authorization: Bearer
6c9f198878cb42e6aad83fecbb598d10_8a9bfdab5a21ca27015a21d72fb20000
 - Content-Type: multipart/form-data;boundary=any_random_value
 - Accept: application/json

却仍可能会返回错误 HTTP 415 Unsupported Media Type

详见：

[【已解决】Mac中Postman去POST实现文件上传时出错：HTTP 415 Unsupported Media Type – 在路上](#)

但是本身Postman对于文件上传的支持，还是很方便很好用的。

C#中如何手动裸写 multipart/form-data 的文件上传

之前也曾折腾过，自己用C#代码拼凑出 multipart/form-data 所需要的内容的格式。

如此折腾后，算是对于 multipart/form-data 有了更深入的认识。

详见：

[【教程】模拟登陆之如何分析并用代码模拟上传文件 – 在路上](#)

crifan.com, 使用署名4.0国际(CC BY 4.0)协议发布 all right reserved, powered by Gitbook最后更新：2021-07-17 17:44:48

混合内容

当很多网站提高安全性，而把 http 的域名，都改为 https 了。

但是如果在此期间网站内部没有完全处理好，残留部分 http 的链接，则就会出现：

Mixed Content = 混合内容 = http网页中加载了http的内容/资源

不同浏览器对Mixed Content 有不一样的处理规则。

此处暂时不需要深究。知道有这个事情即可。

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新：2021-07-17 17:44:48

附录

整理出一些教程、参考资料等内容供参考。

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新: 2021-07-17 17:44:48

优质教程

- [HTTP 协议入门 - 阮一峰的网络日志](#)
- [curl网站开发指南 - 阮一峰的网络日志](#)

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新: 2021-07-17 17:44:48

参考资料

- 【已解决】PySpider中使用多贝云IP代理池实现每次请求IP都不同
- 【整理Book】Web前端三剑客：HTML+CSS+JS
-
- 【未解决】给Axios的http请求添加公共处理部分
- 【整理】Httpbin免费提供HTTP请求和响应的测试网站
- 【已解决】ReactJS中好用的获取网络数据的http库
- CORS开发总结
- 【已解决】JS中如何实现将字典变量去url的encode
- 【已解决】iOS中Alamofire参数解析JSONEncoding出错 – 在路上
-
- URL encoding the space character: + or %20? - Stack Overflow
- Percent-encoding - Wikipedia
- GitHub - Alamofire/Alamofire: Elegant HTTP Networking in Swift
- GitHub - requests/requests: Python HTTP Requests for Humans™ 🎉🍰🎉
- python中 urllib, urllib2, httplib, httplib2 几个库的区别
- 21.11. http — HTTP modules — Python 3.6.3rc1 documentation
- HTTP请求与响应的知识点总结 - 简书
- 20.5. urllib — Open arbitrary resources by URL — Python 2.7.14 documentation
- HTTP请求响应报文&&相关状态码&&GET_POST请求方法 总结
- url encoding - How to urlencode a querystring in Python? - Stack Overflow
- HTTP协议详细总结 - 周东尧的个人页面
- Get Started - PayPal Developer
- Status Codes - Flask API
- Status codes in HTTP
- HTTP头部详解-zooyo-ChinaUnix博客
- HTTP头信息详解-转 - chuncn - 博客园
- HTTP Header 详解_知识库_博客园
- HTTP 方法：GET 对比 POST
- List of HTTP header fields - Wikipedia, the free encyclopedia
- HTTP 教程 | 菜鸟教程
- HTTP Header 详解_知识库_博客园
- HTTP/1.1: Header Field Definitions
- Requests Header | Http Header
- HTTP content-type | 菜鸟教程
- 403 Forbidden vs 401 Unauthorized HTTP responses - Stack Overflow
- HTTP/1.1: Status Code Definitions
- Fetch Standard
- Mixed Content
- 什么是混合内容? |Web | Google Developers
- 关于启用 HTTPS 的一些经验分享（一） | JerryQu 的小站
- 混合内容 - 安全 | MDN
- XTM Connect SDK – REST API
- Vue乱搞系列之axios发起表单请求 - 简书

- [Vue.js 学习系列三——axios 学习实践 - 前端 - 掘金](#)
- [axios/axios: Promise based HTTP client for the browser and node.js](#)
-

crifan.com, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by
Gitbook最后更新: 2021-07-17 21:15:44