# 接口鉴权

## 1. 简介

### 1.1 API认证简介

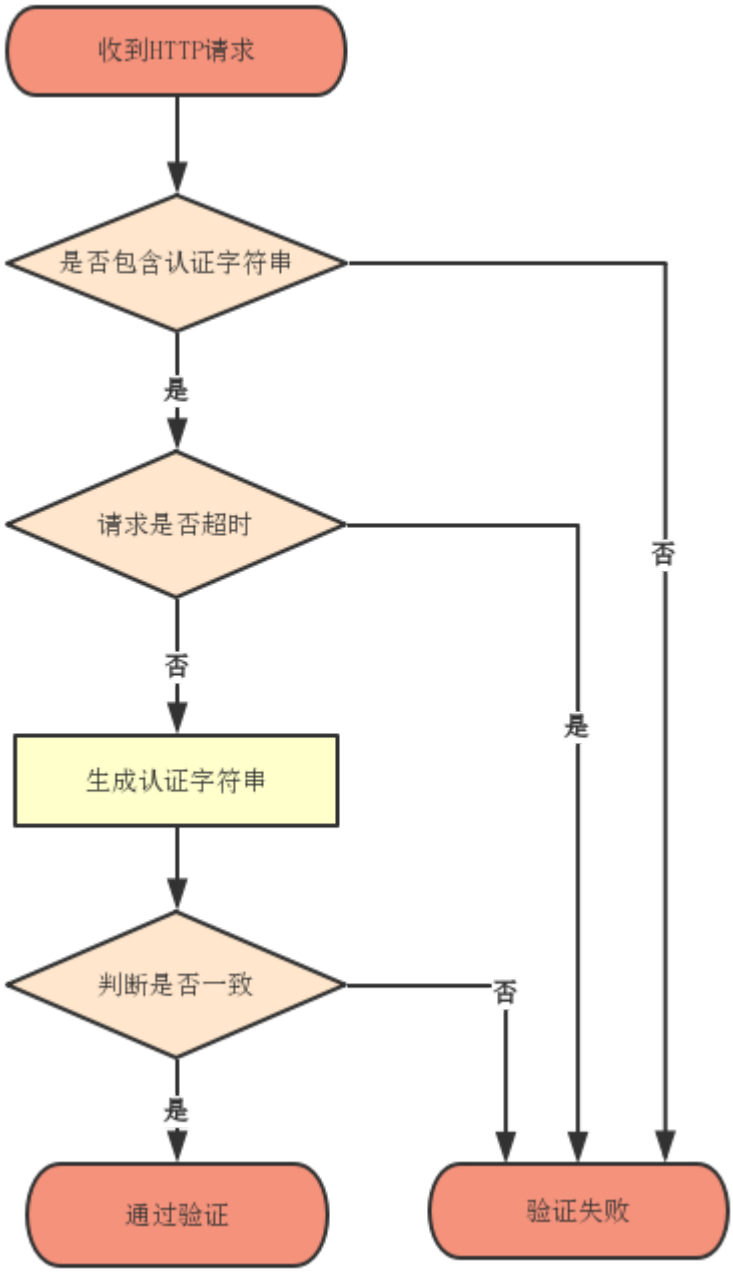
> 用户在授权使用接口前会收到一对Access Key Id和Secret Access Key, Access Key Id (AKI) 用于标记用户及其有权使用的API应用, Secret Access Key (SAK) 用于加密认证字符串, 也是系统验证认证字符串的秘钥, SAK必须保密, 只有用户和系统知道。

> 用户在以POST方式请求API接口时需要验证请求的合法性, 系统将使用相同的SAK和同样的认证机制生成认证字符串, 并与用户请求中包含的认证字符串进行比对。如果认证字符串相同，系统认为用户拥有指定的操作权限，并执行相关操作; 如果认证字符串不同，系统将忽略该操作并返回错误码。

### 1.2 API认证方式

用户首先需要将HTTP请求的指定内容连接成字符串，结合SAK，通过HMAC算法计算密文摘要，这个过程也就是对HTTP请求进行签名过程。系统通过基于认证字符串的HTTP请求签名机制来验证用户身份。对于每个HTTP请求, 都需要在HTTP Header Authorization头域中包含该认证字符串。

### 1.3 认证处理流程




### 1.4 认证字符串生成简介

```
CanonicalRequest =
    HTTP Method \n
    CanonicalURI \n
    CanonicalQueryString \n
    CanonicalHeaders

SigningKey = HMAC-SHA256-HEX(SAK, authStringPrefix)

Signature = HMAC-SHA256-HEX(SigningKey, CanonicalRequest)

# 最终的认证字符串
Authorization = yq-api-v1.0/{AKI}/{Timestamp}/{ExpirationTime}/{SignedHeaders}/{Signature}
```

# 2. 认证字符串生成

> 要得到认证字符串, 得知道Signature, 得到Signature得先构造CanonicalRequest并计算SigningKey

## 2.1生成CanonicalRequest

CanonicalRequest = HTTP Method + '\n' + CanonicalURI + '\n' + CanonicalQuery + '\n' + CanonicalHeaders, 其中

| 函数 | 描述 |
| --- | --- |
| hmac.new(key, info, hashlib.sha256).hexdigest() | 调用HMAC SHA256算法, 根据秘钥key和密文info输出密文摘要, 并把结果转换为小写的十六进制字符串 |
| normStrings(strs, encoding_slash=True) | 对给定字符串(先转换为UTF8编码)中除了URI非保留字符的其他字符进行十六进制百分号编码, 字母采用大写. URI非保留字符包括: 大小写英文字母/数字/连接符/点/下划线/波浪线. encoding_slash默认对斜杠也进行编码, 可以指定对斜杠不做此编码 |

***HTTP Method:***

> 请求的方法, 全大写, 只能是POST

***CanonicalURI:***

> 是对URL中的绝对路径进行16进制百分号编码后normStrings的结果

比如api url = `http://127.0.0.1:80/blackcheck`, 则uri = `/blackcheck`, CanonicalURI = normStrings(uri)=`/blackcheck`

***CanonicalQueryString:***

> 对URL中'?'后的键值对字符串(Query String)进行编码后的结果.

1. 将Query String 根据`&`拆分成每项是key=value或只有key的形式
2. 如果该项只有key, 转换成normStrings(key)+'='的形式, 如果是key=value的项, 转换为normStrings(key)+'='+normStrings(value)的形式, value可以为空字符串
3. 将转换后的每项按照字典顺序排列, 最后用`&`连接

***CanonicalHeaders:***

> 对HTTP请求的Headers头域参数进行选择性编码的结果

用户可以自己指定哪些参数需要编码, 但是至少需要包含以下参数:

1. Host
2. Content-Length
3. Content-Type
4. Content-MD5
5. Query-Date
6. 所有以yq-api-开头的Headers参数

如果按照上面至少包含的参数进行编码, 那么认证字符串中的{signedHeaders}可以为空, 如果在此基础上还加入了其他的字段, 那么{signedHeaders}为参与编码的Headers名字转换为小写后按照字典排序, 然后用分号连接的长字符串

对每个参与编码的Header进行如下处理:

1. 将Header的名字变成全小写
2. 将Header的值去掉开头和结尾的空白字符
3. 经过上一步之后值为空字符串的Header忽略，其余的转换为 normStrings(name) + ":" + normStrings(value) 的形式
4. 把上面转换后的所有字符串按照字典序进行排序
5. 将排序后的字符串按顺序用\n符号连接起来得到最终的CanonicalQueryHeaders

举例, 要编码的Header如下:

```
{'Host': 'http://127.0.0.1',
 'Query-Date': '2018-12-27T07:58:19Z',
 'Content-Type': 'application/json',
 'Content-Length': 70,
 'Content-MD5': 'e31bf1b5eaf1b1f113c1af0550090b3d',}
```

经上述处理步骤后的结果CanonicalHeaders为

'content-length:70\ncontent-md5:e31bf1b5eaf1b1f113c1af0550090b3d\ncontent-type:application%2Fjson\nhost:http%3A%2F%2F127.0.0.1\nquery-date:2018-12-27T07%3A58%3A19Z'

展开为:

```
content-length:70
content-md5:e31bf1b5eaf1b1f113c1af0550090b3d
content-type:application%2Fjson
host:http%3A%2F%2F127.0.0.1
query-date:2018-12-27T07%3A58%3A19Z
```

此时, 认证字符串中的signedHeaders内容为`content-length;content-md5;content-type;host;query-date`

### 2.2 生成SigningKey

SigningKey = hmac.new(SAK, authStringPrefix, hashlib.sha256).hexdigest()

其中,
`SAK`是用户的Secret Access Key, 与AKI(Access Key Id)是一对
`authStringPrefix`是认证字符串的前缀部分, `yq-api-v1.0/{AKI}/{timestamp}/{expirationTime}`

### 2.3 生成Signature

Signature = hmac.new(SigningKey, CanonicalRequest, hashlib.sha256).hedigest()

### 2.4 生成authString

认证字符串`authString = yq-api-v1.0/{AKI}/{timestamp}/{expirationTime}/{signedHeaders}/{signature}`

`timestamp:` 签名生效UTC+8时间，格式为yyyy-mm-ddThh:mm:ssZ，例如：2018-12-27T07:58:19Z，默认值为当前北京时间
`expirationTime:` 签名有效期限，从timestamp所指定的时间开始计算，时间为秒，默认值为1800秒（30）分钟
`signedHeaders:` 签名算法中涉及到的HTTP头域列表。HTTP头域名字一律要求小写且头域名字之间用分号（;）分隔，如content-length;query-date;host。列表按照字典序排列。当signedHeaders为空时表示取默认值

### 2.5 认证字符串生成举例

假设用户向地址`http://127.0.0.1:80/blackcheck`采用POST的方式传递了`json={'idcard': '320310198211195371', 'phone': '18111112222', 'name':'李四'}`的数据, 即

```
access_key_id = '6jrmeqzg4z5hyu8yz7bi0f4z6bzvkl00'
secret_access_key = 'y97cdobpg6s79nctrxpyeworsnxl8gwn'
host = 'http://127.0.0.1:80/blackcheck'
http_method = 'POST'
path = '/blackcheck'
json = {'idcard': '320310198211195371', 'phone': '18111112222', 'name':'李四'}

headers = {
    'Host': 'http://127.0.0.1:80/blackcheck',
    'Content-Type': 'application/json',
    'Content-MD5' : content_md5,
    'Content-Length': str(content_len),
    'Query-Date': query_date,}

headers_to_sign = None
timestamp = 1545901200.0
expiration_time = 1800
```

其中, headers中的
Content-Type是固定的, 必须是`application/json`,
Content-MD5, Content-Length和Query-Date的值是需要进一步计算或转换才能得到, 先将json的取值转换为unicode编码, 如果再转换为utf8编码后计算的MD5值即为Content-MD5, 直接求得的长度为Content-Length,
Query-Date即timestamp格式化为`%Y-%m-%dT%H:%M:%SZ`的字符串

1. 生成 CanonicalRequest

```
# HTTP Method
POST
# CanonicalURI
/blackcheck
# CanonicalQueryString 此例为空

# CanonicalHeaders
content-length:70
content-md5:4c09808622a1df08e2902e726b44920b
content-type:application%2Fjson
host:http%3A%2F%2F127.0.0.1
query-date:2018-12-27T17%3A00%3A00Z
```

2. 生成SigningKey

```
SAK = 'y97cdobpg6s79nctrxpyeworsnxl8gwn'
auth_string_prefix = yq-api-v1.0/6jrmeqzg4z5hyu8yz7bi0f4z6bzvkl00/2018-12-27T17:00:00Z/1800
SigningKey = hmac.new(SAK, auth_string_prefix, hashlib.sha256).hexdigest()
           = '15d0f8e4c3cc8e810e10e9d37a3a62030573a5807f25b1e664e0851629269faf'
```

3. 生成Signature

```
Signature = hmac.new(SigningKey, CanonicalRequest, hashlib.sha256).hexdigest()
           = '479a53e69d8412dc85c714a1feb31d204937df5bcf165bd431f58cdcc7043fea'
```

4. 生成authString(认证字符串)

```
Authorization = yq-api-v1.0/6jrmeqzg4z5hyu8yz7bi0f4z6bzvkl00/2018-
    12-27T17:00:00Z/1800//479a53e69d8412dc85c714a1feb31d204937df5bcf165bd431f58cdcc7043fea
```

此时signedHeaders = headers_to_sign = None, 采用默认的headers签名所以超时时间1800后面有两个斜杠

# 在headers中包含认证字符串

> **生成认证字符串后, 加入到headers中再向系统请求**

```
url = 'http://127.0.0.1:80/blackcheck'
json = {'idcard': '320310198211195371', 'phone': '18111112222', 'name':'李四'}
headers = {
    'Authorization': 'yq-api-v1.0/6jrmeqzg4z5hyu8yz7bi0f4z6bzvkl00/2018-12-27T17:00:00Z/1800//479a5
    'Host': 'http://127.0.0.1',
    'Content-Type': 'application/json',
    'Content-MD5': '4c09808622a1df08e2902e726b44920b',
    'Content-Length': '70',
    'Query-Date': '2018-12-27T17:00:00Z'}
response = requests.post(url, json=json, headers=headers)
```

# 生成认证字符串的Python代码示例

```
#coding:utf-8
import time
import hmac
import hashlib
import string
from urllib import parse

class authBlackCheck:
    """生成认证字符串"""

    def __init__(self,
            access_key_id, secret_access_key, http_method, path, params, data, headers,
            headers_to_sign=None, timestamp=0, expiration_in_seconds=1800):
        """初始化类时传入上述参数, 有两个有默认参数"""

        self.access_key_id = access_key_id
        self.secret_access_key = secret_access_key
        self.http_method = http_method
        self.path = path
        self.params = params
        self.data = data
        self.timestamp = timestamp
        self.headers = headers
        self.headers_to_sign = headers_to_sign
        self.expiration_in_seconds = expiration_in_seconds

        # 不需要做百分号编码的字符集合及十六进制编码列表: 大小写英文字母, 数字, '.~-_'
        set_char = set(string.ascii_letters + string.digits + '.~-_')
        #print(set_char)
        self.list_normalized = [chr(i) if chr(i) in set_char else '%%%02X'%i
                for i in range(256)]
        #print(self.list_normalized)


    def transTs2CanonicalTime(self):
        """将时间戳转换为指定格式时间字符串, 传入的timestamp为UTC时间"""

        if self.timestamp == 0:
            mktime = time.strftime('%Y-%m-%dT%XZ', time.localtime())
        else:
            mktime = time.strftime('%Y-%m-%dT%XZ', time.localtime(self.timestamp))

        return mktime
```

```
def normStrings(self, strs, encoding_slash=True):
    """
        对传入字符串进行编码，默认对字符串内的斜杠也进行编码
        保留所有'URI非保留字符'原样不变. RFC 3986规定，"URI非保留字符"包括以下字符：
        字母（A-Z，a-z）、数字（0-9）、连字号（-）、点号（.）、下划线（_）、波浪线（~）
        对其余字节做一次RFC 3986中规定的百分号编码(Percent-encoding)
        即一个'%'后面跟着两个表示该字节值的十六进制字母，字母一律采用大写形式。
    """

    if strs is None:
        return ''

    # python3中不需要做此转换?
    #strs = strs.encode('utf-8')
    #    if isinstance(strs, str) else bytes(str(strs), 'utf-8')

    if encoding_slash:
        encode_f = lambda x: self.list_normalized[ord(x)]
    else:
        encode_f = lambda x: self.list_normalized[ord(x)] if x != '/' else x

    return ''.join([encode_f(x) for x in str(strs)])


def transCanonicalUri(self):
    """规范化Request URL绝对路径uri, 对除'/'外的所有字符编码"""

    return self.normStrings(self.path, False)


def transCanonicalQueryString(self):
    """将params字段转换为标准字符串并用'&'拼接, 不转换authorization字段"""

    if self.params is None:
        return ''

    #print(self.params.items())
    result = ['%s=%s' % (k, self.normStrings(v))
            for k, v in self.params.items() if k.lower != 'authorization']
    result.sort()

    return '&'.join(result)


def transCanonicalHeaders(self):
    """对参与签名的headers元素进行编码, 并构造成一个长字符串返回"""

    headers = self.headers or {}
    headers_to_sign = self.headers_to_sign
    #print(headers)

    # 如果没有指定headers, 默认一下参数参与签名
    # host/content-md5/content-length/content-type/query-date
    if headers_to_sign is None or len(headers_to_sign) == 0:
        headers_to_sign = {
            'host', 'content-md5', 'content-length', 'content-type', 'query-date',}

    result = []
    for k,v in [(k.strip().lower(), str(v).strip()) for k,v in headers.items()]:
        if k.startswith('yq-api-') or k in headers_to_sign:
            result.append("%s:%s" % (self.normStrings(k), self.normStrings(v)))
    #print(result)
    result.sort()
    #print(result)

    return '\n'.join(result)
```

```
    def sign(self):
        """"""

        #1 生成SigningKey
        #1.1 得到认证字符串前缀, authStringPrefix格式为
        # yq-api-v1.0/{access_key_id}/{query_date}/{expiration_time}
        self.auth_string_prefix = '/'.join(['yq-api-v1.0',
                self.access_key_id,
                self.transTs2CanonicalTime(),
                str(self.expiration_in_seconds),])
        print('auth_string_prefix: ', self.auth_string_prefix)
        #1.2 生成SigningKey, HMAC-SHA256-HEX(sk, authStringPrefix)
        self.sign_key = hmac.new(
                bytes(self.secret_access_key, 'utf8'),
                bytes(self.auth_string_prefix, 'utf8'),
                hashlib.sha256).hexdigest()
        print('SigningKey:', self.sign_key)

        #2 生成CanonicalRequest, 其组成为,
        #2 HTTP Method + '\n' + CanonicalURI + '\n' + \
        #2 CanonicalQueryString + '\n' + CanonicalHeaders
        #2.1 对URL中的绝对路径进行编码 path = '/blackcheck'
        self.canonical_uri = self.transCanonicalUri()
        #2.2 对URL中的请求参数进行编码 URL中?后的键值对
        self.canonical_querystring = self.transCanonicalQueryString()
        #2.3 对HTTP请求中的Header部分进行选择性编码的结果
        self.canonical_headers = self.transCanonicalHeaders()
        #2.4 生成CanonicalRequest, 待签名字符串strins_to_sign
        self.strings_to_sign = '\n'.join([
                self.http_method, self.canonical_uri,
                self.canonical_querystring, self.canonical_headers])
        #print(self.http_method)
        #print(self.canonical_uri)
        #print(self.canonical_querystring)
        #print(self.canonical_headers)
        print('CanonicalRequest:\n', self.strings_to_sign)

        #3 生成签名字符串Signature, 对待签名字符串利用签名关键字进行签名
        self.signature = hmac.new(
                bytes(self.sign_key, 'utf8'),
                bytes(self.strings_to_sign, 'utf8'),
                hashlib.sha256).hexdigest()
        print('\nsignature', self.signature)

        #4 生成认证字符串
        if self.headers_to_sign:
            result = '/'.join([self.auth_string_prefix,
                    ';'.join(sorted(list(self.headers_to_sign))), self.signature])
        else:
            result = '/'.join([self.auth_string_prefix, '', self.signature])

        return result


if __name__ == "__main__":

    #1 设置 应用ID/应用密钥/请求地址/请求方法
    access_key_id = '6jrmeqzg4z5hyu8yz7bi0f4z6bzvkl00'
    secret_access_key = 'y97cdobpg6s79nctrxpyeworsnxl8gwn'

    url = 'http://127.0.0.1:80/blackcheck'
    http_method = 'POST'
    #1 提取 请求主机地址/请求路径, 设置请求参数
    url_split = parse.urlsplit(url)
    host = url_split.scheme + '://' + parse.splitport(url_split.netloc)[0]
    path = url_split.path
    params = {}
```

```
#2 提交数据, 及提交内容的MD5和数据长度
data = {'idcard': '320310198211195371', 'phone': '18111112222', 'name':'李四'}
content_type = 'application/json'
content_md5 = hashlib.md5(str(data).encode('utf8')).hexdigest()
content_len = len(str(data))

#3 请求时刻(北京时间)时间戳
timestamp = time.mktime(time.localtime())
# headers中的查询时间转换为 UTC时间戳
query_date = time.strftime('%Y-%m-%dT%XZ', time.localtime(timestamp))

#4 构造请求头headers, 指定参与签名的headers参数
headers = {
    'host': host,
    'content-type': content_type,
    'content-md5' : content_md5,
    'content-length': str(content_len),
    'query-date': query_date,}
headers_to_sign = None
#headers_to_sign = {'host', 'content-type', 'content-md5', 'content-length', 'query-date'}

#5 传输延迟时间(秒)
expiration_time = 1800

'''
print('host:\t\t', host)
print('path:\t\t', path)
print('data:\t\t', data)
print('content_md5:\t\t', content_md5)
print('content_len:\t\t', content_len)
print('timestamp:\t\t', timestamp)
print('query_date:\t\t', query_date)
'''
print('headers:\t\t', headers)
#print('headers_to_sign:\t\t', headers_to_sign)
#print('expiration_time:\t\t', expiration_time)

# 初始化计算认证字符串的类, 给定 :
# 应用ID, 应用密钥, 请求方法, 请求路径, 请求参数, 提交数据,
# 请求时刻时间戳, 请求头, 延迟时间(秒), 自定义请求头
abc = authBlackCheck(
        access_key_id, secret_access_key, http_method, path, params,
        data, headers, headers_to_sign, timestamp, expiration_time)
# 得到认证字符串

result = abc.sign()
print('\nauth_string:', result)
```

# 其他