**浅谈nodejs中的Crypto模块**

 发布于 4 年前  作者 [DoubleSpout](https://cnodejs.org/user/DoubleSpout)  52396 次浏览

node.js的crypto在0.8版本并没有改版多少，这个模块的主要功能是加密解密。 node利用 OpenSSL库来实现它的加密技术，这是因为OpenSSL已经是一个广泛被采用的加密算法。它包括了类似MD5 or SHA-1 算法，这些算法你可以利用在你的应用中。

1、我们先来看hash算法： 我们可以通过 crypto.createHash() 来创建一个Hash实例。 我们可以利用以下算法来创建hash实例

1. md5
2. sha1
3. sha256
4. sha512
5. ripemd160 MD5是最常用的，但是他有一定的碰撞的问题，你可以使用更新的sha1算法。 我们看hash的生成方法，代码如下：

var md5 = crypto.createHash(‘md5’); md5.update(‘foo’); {} md5.digest(); ’??\u0018?L??\í?eO??¤?' md5.digest(‘hex’); Error: Not initialized at [object Context]:1:5 at Interface.<anonymous> (repl.js:147:22) at Interface.emit (events.js:42:17) at Interface.\_onLine (readline.js:132:10) at Interface.\_line (readline.js:387:8) at Interface.\_ttyWrite (readline.js:564:14) at ReadStream.<anonymous> (readline.js:52:12) at ReadStream.emit (events.js:59:20) at ReadStream.\_emitKey (tty\_posix.js:280:10) at ReadStream.onData (tty\_posix.js:43:12) var md5 = crypto.createHash(‘md5’); md5.update(‘foo’); {} md5.digest(‘hex’); ‘acbd18db4cc2f85cedef654fccc4a4d8’

这里我们生成了一个md5的hash实例，然后直接运行 md5.digest(); 出现了乱码，因为它默认返回的是2进制的数据，然后我们接着 md5.digest(‘hex’); 期望以16进制的形式打印md5值，但是抛出异常了。 也就是说一旦md5.digest();这个方法被调用了，hash 对象就被清空了是不能被重用的。当然我们可以利用base64格式打印出md5字符串。

对于hash.update()方法是有记忆功能的，我们看如下代码：

> var sha1 = crypto.createHash('sha1');

> sha1.update('foo');

{}

> sha1.update('bar');

{}

> sha1.digest('hex');

'8843d7f92416211de9ebb963ff4ce28125932878'

> var sha1 = crypto.createHash('sha1');

> sha1.update('foobar');

{}

> sha1.digest('hex');

'8843d7f92416211de9ebb963ff4ce28125932878'

>

这2次sha1加密结果是一样的，也就是说hash.update()方法就是将字符串相加，然后在hash.digest()将字符串加密返回

2、HMAC HMAC全名是 keyed-Hash Message Authentication Code，中文直译就是密钥相关的哈希运算消息认证码，HMAC运算利用哈希算法，以一个密钥和一个消息为输入，生成一个加密串作为输出。HMAC可以有效防止一些类似md5的彩虹表等攻击，比如一些常见的密码直接MD5存入数据库的，可能被反向破解。 crypto.createHmac(algorithm, key) 这个方法返回和createHash一样，返回一个HMAC的实例，有update和digest方法。我们来看下实际的应用：

> var crypto = require('crypto');

> var fs = require('fs');

>

> var pem = fs.readFileSync('key.pem');

> var key = pem.toString('ascii');

>

> var hmac = crypto.createHmac('sha1', key);

>

> hmac.update('foo');

{}

> hmac.digest('hex');

'7b058f2f33ca28da3ff3c6506c978825718c7d42'

>

我们先通过 fs.readFileSync 方法读取了key.pem密钥，然后将它转为ascii码，最后通过 createHmac(‘sha1’, key) 方法获得HMAC实例，然后执行update和digest，生成一串密钥字符串。 由于key的不同，所以同样的字符串’foo’经过hmac加密后生成的16进制字符串也是不同的，从而更加保障了数据的安全性。

关于key.pem是什么，我们可以利用opensll命令来创建一个key.pem，简单命令如下：

[root[@localhost](/user/localhost) test]# openssl genrsa -out server.pem 1024

Generating RSA private key, 1024 bit long modulus

...............................++++++

....................++++++

e is 65537 (0x10001)

[root[@localhost](/user/localhost) test]# vi server.pem

-----BEGIN RSA PRIVATE KEY-----

MIICXQIBAAKBgQCoWUIOtZAA4EB/bBpKBiHzDRdPCsSqTkR0Bva5kUUjVQ/2y6mi

eZl244GGTwfiFtbHdu0eum4MkIes6IL/kJ8IDeyg4za26fEcO2mfty6BQbSo2SRr

ZdIGVCZ2atvuc2ObzgVpJ+p1GGoFeNS1HOhw0Hq2GoeKt2rMdXceklXaLQIDAQAB

AoGASC/t+Wy1UQrX3Uu3giJbEGN+qTAO4JArHi2WQkAei2YAMG1oUfkeazJm6fad

hL8SXfmK9/AtHEolQ2l4MqoK5rD5kvoLNOWs+sIOd1Zi9jLow8F8x+ctphn5FdZz

J444Mk3M3ua5ZS1YCy12pEbdO6urNMgUpqYcEla0BHIJ8MECQQDaxG9UyId5CgOH

PCXy2oT3dSAa3XJeQ7fC5Tl4Tb4lKezho0Lk/g369cU3O6LWM9g65QQk/Vewtbpe

fKoOtQcFAkEAxQAciOPbASojxZDiyy0CTbgj4PGaPBFwJis/wetAyBQcOKOM7l1h

wHg8fUSXNYLHi6paDqtjaOvVfDOnRdMfCQJBAJg/4WNV88JvD6mMWLS9E5DMeL15

pGCqLDd9JBPvtwdSOEOIRcPsc3pWlRwtatQ8XJ4QSGQd1Gts7flYNVnq7qkCQQC0

3txUBrfNwu3i95pyppll1/oBDnHoUq5gLPc1yRPKX0Rl8Ct1soEMYJhQ/wfBlpg1

MCvNKih0bYqtpRMgNa1ZAkAT5kLpwW5Rb0OxVkHLwtOvixgKABPeGg8JuQS/POHS

lEWvqaaUKdcVcje5YgMg6hDjWHTFJRQ1UEUubYTLFU/D

-----END RSA PRIVATE KEY-----

这样我们就生成了一个密钥，具体openssl命令可以参考相关文案，openssl的命令很多，多的如天上的繁星！这里给大家简单介绍一下ssl的知识。 其实整个ssl知识很多，我查阅了一些资料，目前 SSL(SecureSocketLayer)是netscape公司提出的主要用于web的安全通信标准，.TLS(TransportLayerSecurity)是IETF的TLS工作组在SSL3.0基础之上提出的安全通信标准，SSL/TLS提供的安全机制可以保证应用层数据在互联网络传输不被监听,伪造和窜改。

在介绍 SSL/TLS 知识的之前，我们有必要了解下加密算法的知识： 1、什么是加密算法 加密算法很容易理解，就是把明文变成人家看不懂的东西，然后送给自己想要的送到的地方，接收方用配套的解密算法又把密文解开成明文，这样就不怕密文给人家截获而泄密。 2、加密算法的种类 大致分为2类，一种是基于key的，一种不是基于key的。 不基于key的算法就是消息双方都通过一定的加密和解密算法来进行通信，这种算法缺点很明显如果加密算法被破解了就泄露了。 3、基于key的加密算法 key是一个什么东西呢？随便你，可以是一个随机产生的数字，或者一个单词，啥都行，只要你用的算法认为你选来做key的东西合法就行。所以基于key的加密算法又分为2类：对称加密和不对称加密。对称加密算法的原理很容易理解，通信一方用KEK加密明文，另一方收到之后用同样的KEY来解密就可以得到明文。 4、不对称加密算法 不对称加密指双方用不同的KEY加密和解密明文，通信双方都要有自己的公共密钥和私有密钥。举个例子比较容易理解，我们们假设通信双方分别是A,B. A,拥有KEY\_A1,KEY\_A2,其中KEY\_A1是A的私有密钥，KEY\_A2是A的公共密钥。 B,拥有KEY\_B1,KEY\_B2,其中KEY\_B1是B的私有密钥，KEY\_B2是B的公共密钥。公共密钥和私有密钥的特点是，经过其中任何一把加密过的明文，只能用另外一把才能够解开。也就是说经过KEY\_A1加密过的明文，只有KEY\_A2才能够解密，反之亦然。 5、不对称加密算法通信过程： A-------->;KEY\_A2------------>B A<--------KEY\_B2<------------A 这个过程叫做公共密钥交换，老外管这叫keyexchange。之后A和B就分别用对方的公共密钥解密，用自己的私有密钥加密。 一般公共密钥是要发布出去的，这就是SSL使用的验证机制（注意不是数据传输机制）。常用的不对称加密一般有RSA,DSA,DH等。我们一般使用RSA。

关于SSL: 一般情况下的网络协议应用中，数据在机器中经过简单的由上到下的几次包装，就进入网络，如果这些包被截获的话，那么可以很容易的根据网络协议得到里面的数据.由网络监听工具可以很容易的做到这一点。

SSL就是为了加密这些数据而产生的协议，可以这么理解，它是位与应用层和 TCP/IP之间的一层，数据经过它流出的时候被加密，再往TCP/IP送，而数据从TCP/IP流入之后先进入它这一层被解密，同时它也能够验证网络连接两端的身份（根据我们之前学习的不对称加密算法只是可知）。

SSL协议包含2个子协议，一个是包协议，一个是握手协议。包协议位于握手协议更下一层，我们暂时对包协议的内容没有兴趣。SSL握手过程说简单点就是：**通信双方通过不对称加密算法来协商好一个对称加密算法以及使用的key，然后用这个算法加密以后所有的数据完成应用层协议的数据交换。**

SSL通信流程： 握手一般都是由client发起的，SSL也不例外。 1、client送给server它自己本身使用的ssl的version(ssl一共有三个version)，加密算法的一些配置，和一些随机产生的数据，以及其他在SSL协议中需要用到的信息。 2、server送给client它自己的SSL的version，加密算法的配置，随机产生的数据，还会用自己的私有密钥加密SERVER-HELLO信息。Server还同时把自己的证书文件给送过去。同时有个可选的项目，就是server可以要求需要客户的certificate。 3、client就用server送过来的certificate来验证server的身份。如果server身份验证没通过，本次通信结束。通过证书验证之后，得到server的公共密钥，解开server送来的被其用私有密钥加密过的SERVER-HELLO信息，看看对头与否。如果不对，说明对方只有该server的公共密钥而没有私有密钥，必是假的。通信告吹。 4、client使用到目前为止所有产生了的随机数据(sharedsecret)，client产生本次握手中的premastersecret(这个步骤是有可能有server的参与的，由他们使用的加密算法决定)，并且把这个用server的公共密钥加密，送回给server.如果server要求需要验证client,那么client也需要自己把自己的证书送过去，同时送一些自己签过名的数据过去。 RSA就是我们上一章说过的一种不对称加密算法。首先server把自己的RSA公共密钥送给client，client于是用这个key加密一个随机产生的值(这个随机产生的值就是sharedsecret)，再把结果送给server. 5、Server验证完client的身份之后，然后用自己的私有密钥解密得到premastersecret然后双方利用这个premastersecret来共同协商，得到mastersecret. 6、双方用master一起产生真正的sessionkey,着就是他们在剩下的过程中的对称加密的key了。这个key还可以用来验证数据完整性。双方再交换结束信息。握手结束。

回过头来我们看openssl，openssl就是实现ssl的一个软件，下面我就讨论刚才的命令，生成一个私有密钥： openssl genrsa -des3 -out server.key 1024 genras表示生成RSA私有密钥文件，-des3表示用DES3加密该文件，1024是我们的key的长度。一般用512就可以了，784可用于商业行为了，1024可以用于军事用途了。生成server.key的时候会要你输入一个密码，这个密钥用来保护你的server.key文件，这样即使人家偷走你的server.key文件，也打不开，拿不到你的私有密钥。

Public Key Cryptography 公开密钥加密包括4个类，Cipher, Decipher,Sign, and Verify，即加密，解密，签名，验证。 我们先看Cipher, Decipher这个加密和解密，这里是使用对称加密算法。 看如下代码：

> var crypto = require('crypto');

> var fs = require('fs');

>

> var pem = fs.readFileSync('key.pem');

> var key = pem.toString('ascii');

>

> var cipher = crypto.createCipher('blowfish', key);

>

> cipher.update(new Buffer(4), 'binary', 'hex');

''

> cipher.update(new Buffer(4), 'binary', 'hex');

'ff57e5f742689c85'

> cipher.update(new Buffer(4), 'binary', 'hex');

''

> cipher.final('hex')

'96576b47fe130547'

我们看下代码，我们读取之前生成的key，然后利用 blowfish 加密算法生成 cipher 实例，接着update内容到cipher实例，最后通过cipher.final()方法输出加密串。 其中有几个方法我们要看下api的解释 crypto.createCipher(algorithm, password) crypto.createCipheriv(algorithm, key, iv) 上面这2个方法都返回cipher实例，第一个参数 algorithm 表示用何种加密算法，可以利用 openssl list-cipher-algorithms 命令来查看你的系统支持哪些加密算法。password和key, iv表示密钥，即利用何种密钥加密，password是用来派生key和iv的，key的话是算法原生的key，iv表示初始化向量。

[root[@localhost](/user/localhost) ~]# openssl list-cipher-algorithms

openssl:Error: 'list-cipher-algorithms' is an invalid command.

Standard commands

asn1parse ca ciphers crl crl2pkcs7

dgst dh dhparam dsa dsaparam

enc engine errstr gendh gendsa

genrsa nseq ocsp passwd pkcs12

pkcs7 pkcs8 prime rand req

rsa rsautl s\_client s\_server s\_time

sess\_id smime speed spkac verify

version x509

Message Digest commands (see the `dgst' command for more details)

md2 md4 md5 rmd160 sha

sha1

Cipher commands (see the `enc' command for more details)

aes-128-cbc aes-128-ecb aes-192-cbc aes-192-ecb aes-256-cbc

aes-256-ecb base64 bf bf-cbc bf-cfb

bf-ecb bf-ofb cast cast-cbc cast5-cbc

cast5-cfb cast5-ecb cast5-ofb des des-cbc

des-cfb des-ecb des-ede des-ede-cbc des-ede-cfb

des-ede-ofb des-ede3 des-ede3-cbc des-ede3-cfb des-ede3-ofb

des-ofb des3 desx rc2 rc2-40-cbc

rc2-64-cbc rc2-cbc rc2-cfb rc2-ecb rc2-ofb

rc4 rc4-40

cipher.update(data, [input\_encoding], [output\_encoding]) 往cipher实例中添加数据，第一个参数是填充的数据，第二个参数表示传入数据的格式，可以是’utf8’, ‘ascii’ 或 ‘binary’，默认是 ‘binary’。第三个参数是返回block的数据格式。 注意这里我们update了 new Buffer(4),表示通过随机内存中的4byte字节的内容填充进去。为什么第一次update没有block返回呢，因为4byte不够生成一个block，所以这点我们要注意下。 最后我们通过final方法和之前digest方法一样，生成加密过后的串。

最后还有一个api： decipher.setAutoPadding(auto\_padding=true) 如果这些加密块不是使用标准的填充块的话，你可以把自动填充关闭。这么做是为了防止执行 decipher.final()的时候监察和去除标准填充块，从而可能出错，一般这个方法不会去用。必须在执行update之前执行它。

下面我们看2个完整的加密和解密的代码示例，代码很容易理解不解释了。

var crypto = require('crypto');

var cipher = crypto.createCipher('aes-256-cbc','InmbuvP6Z8')

var text = "123|123123123123123";

var crypted = cipher.update(text,'utf8','hex')

crypted += cipher.final('hex')

var decipher = crypto.createDecipher('aes-256-cbc','InmbuvP6Z8')

var dec = decipher.update(crypted,'hex','utf8')

dec += decipher.final('utf8')

第二段代码

> var crypto = require('crypto');

> var fs = require('fs');

>

> var pem = fs.readFileSync('key.pem');

> var key = pem.toString('ascii');

>

> var plaintext = new Buffer('abcdefghijklmnopqrstuv');

> var encrypted = "";

> var cipher = crypto.createCipher('blowfish', key);

> ..

> encrypted += cipher.update(plaintext, 'binary', 'hex');

> encrypted += cipher.final('hex');

>

> var decrypted = "";

> var decipher = crypto.createDecipher('blowfish', key);

> decrypted += decipher.update(encrypted, 'hex', 'binary');

> decrypted += decipher.final('binary');

>

> var output = new Buffer(decrypted);

>

> output

<Buffer 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76>

> plaintext

<Buffer 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76>

>

最后我们看下签名和验证 Class: Signer 和 Class: Verify 还记得我们之前说的不对称加密算法么，这里我们就利用私钥和公钥来做个简单的例子，我先通过openssl命令生成公钥：

[root[@localhost](/user/localhost) test]# openssl req -key server.pem -new -x509 -out cert.pem

You are about to be asked to enter information that will be incorporated

into your certificate request.

What you are about to enter is what is called a Distinguished Name or a DN.

There are quite a few fields but you can leave some blank

For some fields there will be a default value,

If you enter '.', the field will be left blank.

-----

Country Name (2 letter code) [GB]:

State or Province Name (full name) [Berkshire]:

Locality Name (eg, city) [Newbury]:

Organization Name (eg, company) [My Company Ltd]:

Organizational Unit Name (eg, section) []:

Common Name (eg, your name or your server's hostname) []:

Email Address []:

[root[@localhost](/user/localhost) test]#

[root[@localhost](/user/localhost) test]# ls

cert.pem len.js server.key server.pem

[root[@localhost](/user/localhost) test]#

node.js代码：

> var crypto = require('crypto');

> var fs = require('fs');

>

> var privatePem = fs.readFileSync('server.pem');

> var publicPem = fs.readFileSync('cert.pem');

> var key = privatePem.toString();

> var pubkey = publicPem.toString();

>

> var data = "abcdef"

>

> var sign = crypto.createSign('RSA-SHA256');

> sign.update(data);

{}

> var sig = sign.sign(key, 'hex');

>

> var verify = crypto.createVerify('RSA-SHA256');

> verify.update(data);

{}

> verify.update(data);

{}

> verify.verify(pubkey, sig, 'hex');

1

首先通过，crypto.createVerify(algorithm)和crypto.createSign(algorithm)方法生成实例，然后利用update方法更新数据，最后利用key（私钥）生成签名，同样的验证也是如此，最后通过 verify.verify(pubkey, sig, ‘hex’); 函数签名。

另外几个api 像 createDiffieHellman 等下次再研究吧

本文例子参考部分：node up and running 本文示例代码我都跑通过，请放心参考

下一篇地址：[浅谈nodejs中的Crypto模块（补完）](http://cnodejs.org/topic/5043b97ffef591855138b4e0)

[原文地址：http://snoopyxdy.blog.163.com/blog/static/601174402012730105523656/](http://snoopyxdy.blog.163.com/blog/static/601174402012730105523656/)