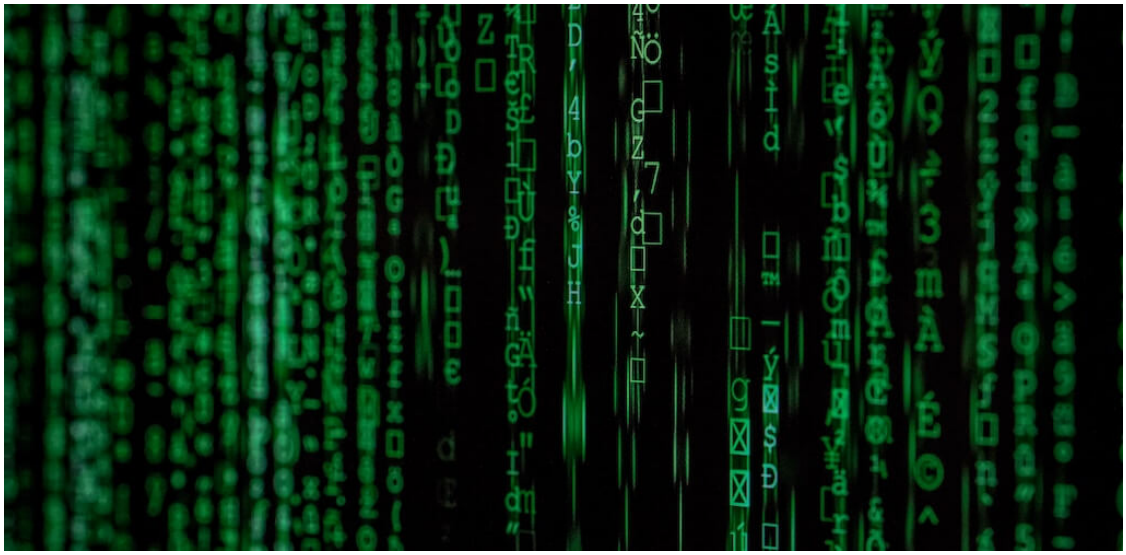


概述

这是关于 JOSE 和密码学的三篇系列文章中的第二篇，你可以在下面的链接中找到其他部分：

1. 基础 - 什么是 JWT 以及 JOSE
2. 理论 - JOSE 中的签名和验证流程 (本文)
3. 实践 - 如何使用 Security.framework 处理 JOSE 中的验证

这一篇中，主要介绍网络传输的密钥的编码和处理方法，以及进行数字签名和验证的基本流程。我们在之后实践一篇里，会使用到这些知识。

密钥的表现形式

显然 JWK 是一种密钥的表现形式，它使用 JSON 的方式，遵守 JWA 的参数，来定义密钥。不过这种表现形式在日常里使用得并不是那么普遍，我们在平时看到得更多的也许是这样的密钥：

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAryQICCl6NZ5gDKrnSztO
3Hy8PEUcuyvg/ikC+VcIo2SFFSf18a3IMYldIugqqqZCs4/4uVW3sbdLs/6PfgdX
7O9D22ZiFWHPYA2k2N744MNiCD1UE+tJyllUhSblK48bn+v1oZHCM0nYQ2NqUkvS
j+hwUU3RiWl7x3D2s9wSdNt7XUtW05a/FXehsPSiJfKvHJJnGOX0BgTvkLnkAOTd
OrUZ/wK69Dzu4IvrN4vs9Nes8vbwPa/ddZEzGR0cQMt0JBkhk9kU/qwqUseP1QRJ
5I1jR4g8aYPL/ke9K35PxZWuDp3U0UPAZ3PjFAh+5T+fc7gzCs9dPzSHloruU+gl
FQIDAQAB
-----END PUBLIC KEY-----
```

这是一个 RSA 公钥的 PEM (Privacy-Enhanced Mail) 表示方式。类似地，对于 ECDSA 密钥，也可以类似表示：

```
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEEVs/o5+uQbTjL3chynL4wXgUg2R9
q9UU8I5mEovUf86QZ7kOBIjJwqnzD1omageEHWwHdBO6B+dFabmdT9POxg==
-----END PUBLIC KEY-----
```

在处理 JOSE 相关的验证时，我们其实是不会涉及这种格式的密钥的。但是我们会用到里面的相关的一些编码方式来处理 JWK 密钥和 Security 框架中 `SecKey` 的转换。所以这里把它作为一节单独介绍。

PEM，ASN.1，X.509 和 DER 编码

上面的 PEM 格式的密钥可以用任意的文本编辑器打开，它就是一个简单的纯 ASCII 字符的文件。由于容易读写复制，所以在交换密钥时这种格式非常流行。每个 PEM 密钥都由 “—–BEGIN #{label})—–” 标签开头，以 “—–END #{label}—–” 标签结尾。注意，PEM 并非专门为了传递 Key 而生，BEGIN 和 END 之后的 label 并不一定就是例子中的 “PUBLIC KEY”，它只是一个让人能读懂的描述，来表示通过这个 PEM 传递的数据到底是什么。

在两个标签之间，就是密钥本身。PEM 中的换行字符需要被忽略掉，可以很清楚地看到，这其实就是一个 **Base64 编码**的字符串。用上面的 ECDSA 密钥为例，将这个 Base64 还原为字节数据的话，结果是：

```
30 59 30 13 06 07 2a 86 48 ce 3d 02 01 06 08 2a 86 48 ce 3d 03 01
07
03 42 00 04 11 5b 3f a3 9f ae 41 b4 e3 2f 77 21 ca 72 f8 c1 78 14
83
64 7d ab d5 14 f0 8e 66 12 8b d4 7f ce 90 67 b9 0e 04 88 c9 c2 a9
f3
0f 5a 26 6a 07 84 1d 6c 07 74 13 ba 07 e7 45 69 b9 9d 4f d3 ce c6
```

很多同学到这里就退缩了，觉得这种二进制没有实际意义。但其实这一串字节是通过 ASN.1 (Abstract Syntax Notation One) 定义的数据。ASN.1 定义了一些表示信息的标准句法，用来对字符串，整数等等进行无歧义和精确地传输。ASN.1 里有很多具体的编码规则，来具体将一些数据按照 ASN.1 的方式进行编码，进行具体表达。在网络传输和密码学中，最简单和最常见的编码方式是 DER (Distinguished Encoding Rules)。

> ASN.1 格式对应的标准是 X.680，DER 被定义在 X.690 中。

有了编码方式以后，为了能表达密钥，我们还需要定义一些元信息，比如一个密钥应该需要声明自己的身份 (在 ASN.1 中称为 “OBJECT IDENTIFIER”)，是一个什么种类的密钥，采用的是什么样的曲线或者 padding，**X.509 标准**就是做这件事的。最后，对于 ECDSA 来说，还有一个 **X9.62** 的标准。它是 X.509 中规定的用来编码 ECDSA 密钥的方式。

光这样说会很抽象，具体来讲，可以简单对这几个概念和各自的作用进行总结：

- ASN.1 - 一种数据或者信息表达时使用的句法，比如“接下来是一串连续内容(SEQUENCE)，长度是...”；“现在开始一个整数”；“从这里开始是位串(BITSTRING)”等这样句法信息。
- DER - 是 ASN.1 的一种具体编码方式，比如使用 `0x30` 表示 SEQUENCE 的开始，然后下一个/若干个字节表示这段内容的长度；使用 `0x02` 表示现在开始是一个整数；使用 `0x03` 表示 BIT STRING 开始等。
- X.509 - 在网络证书和公钥传输时，所应该遵守的 ASN.1 形式。它定义了一个特定证书或者公钥应该由哪些部分构成，比如“一开始应该有一个 SEQUENCE，然后紧接着是两个整数来代表密钥值”等。这些构成的部分由 ASN.1 格式表达，一般由 DER 编码。
- X9.62 - 针对 ECDSA 相关算法的定义。X.509 是一个一般性的密钥编码规定，在 X.509 中指定了 ECDSA 的密钥和签名需要遵守 X9.62。(类似相应地，它也规定了 RSA 的密钥和签名要遵守 PKCS (Public Key Cryptography Standards))。
- PEM - 将证书或者密钥用 DER 编码后，可以得到一组字节数据。把这些数据转换为 Base64 编码的字符串，然后在前后加上 BEGIN 和 END 标签，就得到 PEM 的表现形式。

> 标准有点多？没错，这个世界上有很多标准化制定的组织，在这篇文章中，标准来源也不尽相同。从名字基本可以简单分类：
>
> - RFC (Request For Comments) 是 IETF 这个专门推动互联网标准的组织所发布的
> - “X.” 开头的是 ITU-T 相关的标准，比如 ASN.1 (X.680) 是 ISO 和 ITU-T 的联合标准，X.509 是基于 ASN.1 的补充和扩展。
> - “X9.” 开头的，比如 X9.62，是 ANSI (美国国家标准学会) 的产品
> - PKCS 是 RSA Security 所制定的标准
>
> 我们会看到，在一些 RFC 标准中，会引用和规定需要使用 ANSI 的标准；而本来属于 RSA Security 的一些标准，也出现在了 RFC 中。另外，IETF 也会收录某些其他标准化组织的内容，比如 RFC 3280 其实就是 X.509。和专利市场的相互授权类似，各个标准组织之间也有竞争合作。不过这是另外一个关于爱恨情仇的故事了，其中八卦，我们有机会以后再说。

DER 编码规则

DER 编码的通用规则是，在一个代表类型的字节后面，一般都会接上这个类型的数据所占用的字节长度，然后是实际的数据。

整数

举例说明，在 DER 中，使用 `0x02` 代表整数，所以如果我们想要编码十进制的 100 这个整数时，会得到：

```
00000010 00000001 01100100
  0x02     0x01     0x64
```

`0x02` 代表之后是一个整数，这个整数占用的字节长度为 1（`0x01`），值为 100（`0x64`）。

我们在系列的上一篇文章中提到过，如果数值的首个字节超过 `0x80` 的话，就说明第一个 bit 是 1，在有符号域上这代表一个负数。这时候如果我们想要编码的是一个正数的话，就需要在前面添加一个 `0x00` 的字节。比如我们如果想要编码 `0xCE 2910` 这个整数的话，就需要添加 `0x00`，因为首位的 `0xCE` 在二进制下为 `0b_1100_1110`：

```
0x02 0x04 0x00 0xCE 0x29 0x10
```

序列

将两个整数前后排列，就可以形成一个序列（SEQUENCE），序列的类型编码为 `0x30`，类似地，在类型编码后面也是字节长度值。比如两个整数 `0x64` 和 `0xCE2910` 编码成一个序列，得到的结果是：

```
30 09 02 01 64 02 04 00 CE 29 10
```

为了看上去舒适一些，可以整理一下：

```
30 09 -> SEQUENCE 9 bytes
   02 01 -> Int 1 byte
      64          -> Value 0x64
   02 04 -> Int 4 byte
      00 CE 29 10 -> Value 0xCE2910
```

其他类型及实例

列举几个我们在本文中用到的 DER 的类型编码：

类型	编码
INTEGER	0x02
SEQUENCE	0x30
BIT STRING	0x03
OBJECT IDENTIFIER	0x06

> 如果你想对 DER 有更深入了解，最好的办法应该是看微软的的这个文档，相比于冰冷的标准定义，这里面用人类能懂的语言详细描述了 DER 编码方法。

有了这些基础知识，我们可以来看看本文一开始例子中的 ECDSA 公钥的二进制里都是些什么内容了：

```
30 59 30 13 06 07 2a 86 48 ce 3d 02 01 06 08 2a 86 48 ce 3d 03 01
07
03 42 00 04 11 5b 3f a3 9f ae 41 b4 e3 2f 77 21 ca 72 f8 c1 78 14
83
64 7d ab d5 14 f0 8e 66 12 8b d4 7f ce 90 67 b9 0e 04 88 c9 c2 a9
f3
0f 5a 26 6a 07 84 1d 6c 07 74 13 ba 07 e7 45 69 b9 9d 4f d3 ce c6
```

整理一下：

```
30 59 -> SEQUENCE 89 (0x59) bytes
   30 13 -> SEQUENCE 19 bytes
      06 07 -> OBJECT IDENTIFIER 7 bytes
         2a 86 48 ce 3d 02 01
      06 08 -> OBJECT IDENTIFIER 8 bytes
         2a 86 48 ce 3d 03 01 07
   03 42 -> BIT STRING 66 bytes
         00 -> BIT STRING 0 byte UNUSED
         04 -> The first byte of the string. uncompressed flag
         11 5b 3f a3 9f ae 41 b4 e3 2f 77 21 ca 72 f8 c1
         78 14 83 64 7d ab d5 14 f0 8e 66 12 8b d4 7f ce -> x
         90 67 b9 0e 04 88 c9 c2 a9 f3 0f 5a 26 6a 07 84
         1d 6c 07 74 13 ba 07 e7 45 69 b9 9d 4f d3 ce c6 -> y
```

需要简单说明的有两点：

1. 在 `30 13` 这个 SEQUENCE 里，我们能找到两个 OBJECT IDENTIFIER 的定义。关于 OBJECT IDENTIFIER 的编解码规则，可以参考这里的说明。这部分不是重点，所以就简单只说结论然后跳过了。这两个值分别代表：
   - `1.2.840.10045.2.1` - (ecPublicKey)
   - `1.2.840.10045.3.1.7` - (P-256)

可以看到，它定义了这个公钥的类型，以及使用的曲线。

> 关于解码后的 OBJECT IDENTIFIER 所代表的涵义，可以在这里进行查询。

1. BIT STRING 定义的是一个一串 BIT 数据 (注意这里的 STRING 并不是字符串的意思)。在一个 bit 为单位的数据里，可能存在想要传输的数据 bit 数不是 8 的倍数的情况。但是在 DER 编码长度时，我们指定的是 byte 数。比如在 `03 42` 这 BIT STRING 中，我们的 STRING 长度是 66 个字节 (528 bit)。但是如果我们想要传输的 bit 数只有 523 bit 时，最后一个 byte 中的后 5 bit 数据其实并不是我们想要的。这时候我们需要一种方式来指定应该“丢弃”掉最后若干 bit。`03 42` 之后的字节 `0x00` 负责指定数据末尾有多少 bit 不应该使用。当然，这里我们想要传输的数据 bit 数恰好是 8 的倍数，所以设为 0，表示所有 bit 我们都要使用。如果我们想要舍弃最后 5 bit 的话，这个 byte 就应该是 `0x05`。

> DER 中还有一个类型叫做 OCTET STRING，它定义的是一个 8 bit (OCTET，或者说字节) 组成的字节串流。而 BIT STRING 传输的单位是一个 bit，要注意区分。(我们在本文中不会用到 OCTET STRING，它通常用来传输一些 ACSII 字符串等)

接下来按照 SEC 和 X9.62 的规定，整个 BIT STRING 就应该是 ECDSA 公钥的 x 和 y 的值了。这里第一位的 `0x04` 表示这个密钥中的整数值是没有被压缩的。这个 byte 其他可选的值有 `0x00` (椭圆曲线取点在无穷)， `0x02` (压缩，even y)， `0x03` (压缩， odd y)。通常我们见到的 (以及 iOS 默认能接受的) 都是无压缩的整数值。之后 BIT STRING 还剩 64 位，它们分别就是 32 位的 x 和 32 位的 y 值了！

如果你回到第一篇文章，我们曾经给过一个示例的 JWK：

```
{
  "kty":"EC",
  "alg":"ES256",
  "use":"sig",
  "kid":"3829b108279b26bcfcc8971e348d116",
  "crv":"P-256",
  "x":"EVs_o5-uQbTjL3chynL4wXgUg2R9q9UU8I5mEovUf84",
  "y":"kGe5DgSIycKp8w9aJmoHhB1sB3QTugfnRWm5nU_TzsY"
}
```

你可以尝试一下将这里的 x 和 y 的值转换为字节：

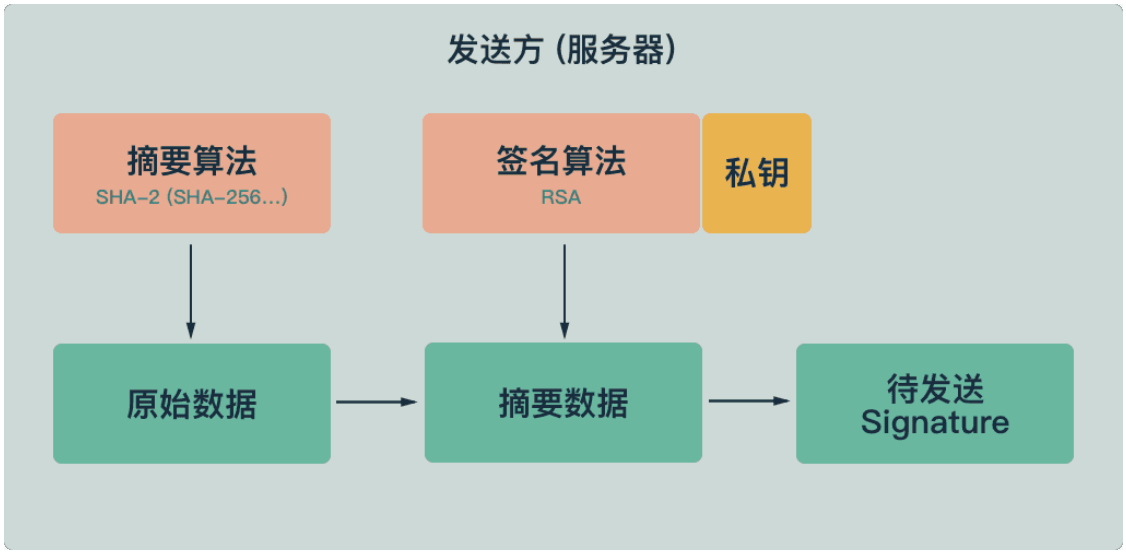
```
x ->
11 5b 3f a3 9f ae 41 b4 e3 2f 77 21 ca 72 f8 c1
78 14 83 64 7d ab d5 14 f0 8e 66 12 8b d4 7f ce

y ->
90 67 b9 0e 04 88 c9 c2 a9 f3 0f 5a 26 6a 07 84
1d 6c 07 74 13 ba 07 e7 45 69 b9 9d 4f d3 ce c6
```

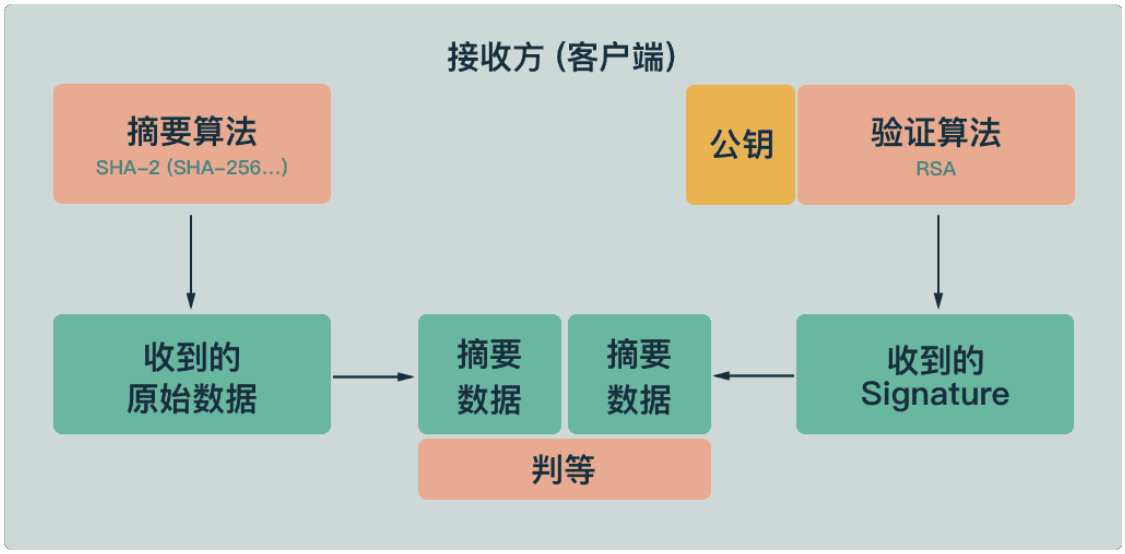
是不是感觉有点眼熟？没错，它们就是上面 DER 编码的 BITSTRING 里 `0x04` 后面的部分。嗯...要不你也可以试试看自己分析一下本文一开始给出的那个 PEM 格式的 RSA 公钥？它和系列第一篇文中的 JWK 格式的 RSA 公钥也是等价的。

> 如果你遇到困难，可以利用搜索引擎。你也可以在 RFC 2315 或者 PKCS#7 中找到 RSA 公钥的编码方式。
>
> 另外，顺带一提，作为 iOS 开发者经常从 Keychain 导出证书时使用的 .p12 文件，其实就是遵守 PKCS#12，来将一个 X.509 证书和私钥打包到一起。

非对称密钥的签名和验证

至此，我们完整了解了密钥，至少是 RSA 和 ECDSA 公钥。使用 RSA 进行数据签名和验证的流程如下：





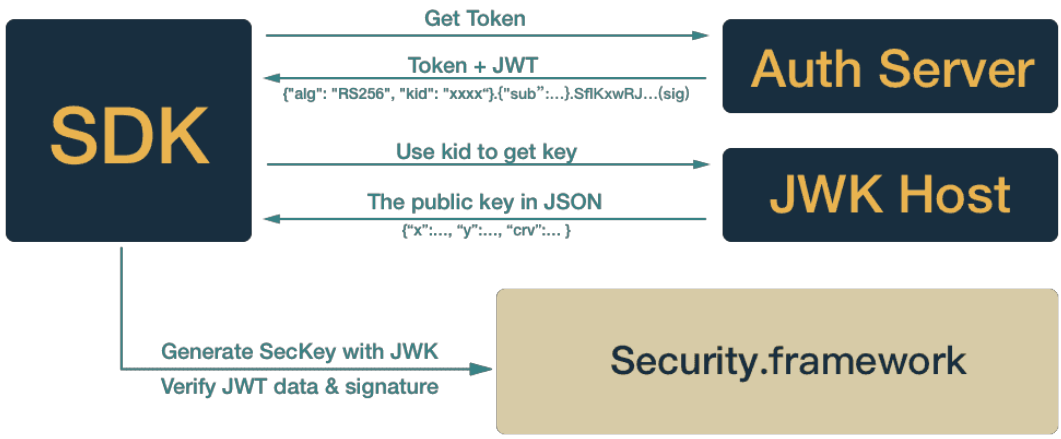
注意，上面的图是 RSA 算法的情况，对于 ECDSA 来说，流程稍有不同。ECDSA 在验证时**并不是**把收到的 Signature 还原成摘要数据，然后进行对比。ECDSA 的 Signature 是由两个大数组成 (通常写作 {r, s})，通过 Signature 中的整数 s，以及收到的原始数据所计算出的摘要，可以计算出另一个整数 v。如 r 和 v 相等的话，就认为验证通过，否则失败。

> RSA 签名时，实际上是对摘要进行稳定的变换运算 (当然通常你也可以把它叫做加密)，所以对于给定的数据，散列算法和加密私钥，得到的签名是一致的。但是对于 ECDSA 来说，签名时需要一个随机选择的 k 值，因此每次进行 ECDSA 所得到的签名内容是不同的。不太了解这一点的话，可能会在看到每次签名变化时感到疑惑。

RSA 或者 ECDSA 具体的算法原理在这里就不展开了，都是一些基本的数学运算。几乎所有的平台对这些算法都会有现成的实现，如果没有特殊必要，一般不会需要自己去进行实现。比如在 iOS 平台上，Security.framework 的 `SecKeyCreateSignature` 和 `SecKeyVerifySignature` 就为我们实现了签名和验证的相关接口，我们只需要传入合适的参数即可。

OpenID Connect Discovery

作为这部分的结尾，让我们看一点轻松的话题吧。在理论的海洋里“畅游”了一番以后，应该回顾一下我们最初的目的，就是这张流程图：



第一步，从 Auth Server 获取 JWT，并完成解析，是很简单的事情。最后一步，将 JWK 转换成 Security 框架中的密钥，并且对 JWT 的数据进行验证，我们在详细了解了密钥和签名/验证的知识以后，大概也能解决。剩下的问题是中间的框图：我们应该去哪儿寻找 JWT Header 中定义的公钥?

把 JWK Host 的 URL 写死在客户端当然是最简单省事儿的方法，但是其实有业界更通用一些的做法，那就是用 Discovery Document。最初的起源是 OpenID Connect，或者说 OAuth2 中需要一系列 API (发起验证请求，交换 token 等，都是不同的 API entry)。这一套内容都可以通过配置来改变，相比于把每个 API 都写死在客户端，我们可能更愿意选择只写死一个入口，而 Discovery Document 就是这个入口。

> 有时候有人会把 OpenID Connect 和 OAuth2 弄混淆，它们其实是不一样的东西：OpenID Connect 负责的是“验证”，也就是负责“你真的是你吗”的问题；而 OAuth2 负责“授权”，也就是“我可以访问你的数据吗”的问题。不过两者有时候会被一起执行，所以不需要分得那么清楚。比如 Google 的 OAuth 2.0 API 也负责了验证的工作。(LINE 的 Login API 亦是如此，LINE SDK 在授权时同时也完成了验证。)

你可以找到一些 Discovery Document 的例子，比如 Google 的，或者 LINE 的。在里面可以找到 `jwks_uri` 这个 key，它就是我们的 JWK Host 的位置，这个位置会放置了若干个 JWK (它们合在一次称为 JWK Set)。里面应该包括之前在 JWT Header 中所指定的 `kid` 的密钥。

小结

本文主要介绍了如何处理和编码的密钥，以及进行数字签名和验证的基本流程。有一部分内容虽然在处理 JOSE 时用不到 (比如 PEM 格式)，但是作为密码学和网络交换中最常用的知识，了解后相信会在未来的某一天派上用场。

我们在之后实践一篇里，会先解析收到的 JWT。然后依靠本篇文章的这些知识，将 JWK 转换为 `SecKey`，并使用 Security.framework 提供的 API 和算法，来完成 JWT 的验证工作。同时，也会讨论一些工程中的经验和选择。