概念 • 2026年5月23日

← 上一页 • 下一页 →

24 个单词：什么是加密身份

加密身份不是密码：没有服务器存储它，也无法恢复。对比 BIP39 机制的教学式解释，为什么恰好是 24 个单词，以及拥有它们的人承担着怎样的真实责任。

通俗地说：如果你忘记了 Gmail 密码，Google 会为你重置。如果你丢失了构成加密身份的 24 个单词，没有人可以去索要。这并不是因为程序严苛，而是因为在另一端根本没有任何人存在。这种区别就是本质的区别。

密码与身份的区别

在经典的互联网模型中，密码并不是用户的身份。它是一份凭证。用户拥有一个身份——姓名、电子邮件、客户编号——为了向服务器证明自己就是所声称的那个人，用户提交一份密码，服务器将其与存储的指纹进行比对。如果指纹吻合，服务器就会允许该会话。如果密码丢失，用户仍然是同一个用户；他失去的只是凭证，而存在一套恢复程序——发送邮件到注册地址、安全问题——来重新获得凭证。

加密身份的运作方式则不同。它不是由某人与存储的指纹进行比对的凭据；它*本身*就是一个完整的数学秘密。无论它存在于何处——纸上、设备中，甚至是别人的服务器上——身份是依靠其数学原理存在的，而不是依靠验证它的人。这里出现了一种类似于我们在《SHA-256 究竟是什么》中看到的属性：所有权不是通过展示秘密来证明的，而是通过使用秘密进行签名来证明的。这样产生的签名，任何人都可以通过从秘密本身数学推导出的公钥值进行验证，而无需知道秘密本身，也无需第三方的介入。拥有秘密的人就是该身份；失去秘密的人就不再是该身份。结论是断然的：**没有人可以让你去索回身份。那个人并不存在，因为他从一开始就没有拥有过身份。**

24 个单词代表了什么

加密身份通常由一个 32 字节（256 位）的数学秘密表示。这是一个难以记忆且极难在不发生错误的情况下进行转录的数字。加密行业在 2013 年通过一个名为 BIP39 的精巧标准解决了这个问题：将这 256 位表示为从包含 2048 个单词的官方列表中提取的 24 个单词序列。背后的算术逻辑非常精妙；想要详细了解的人可以在边注中找到相关内容。

计算是从结尾开始的。我们要通过添加 8 位校验和（checksum）来表示秘密的 256 位：总共 264 位。如果我们将它们分配给 24 个单词（这是一个便于记录和口述且不易出错的数量），每个单词必须正好提

供 11 位信息。而 11 位就是 2 的 11 次方种可能性，即 2048。因此，BIP39 的官方词汇表恰好就是这个规模：列表是为了解决问题而存在的，而不是相反。

计算并非装饰。如果有人正确转录了 23 个单词，而在第 24 个单词上出了错，校验和就会检测到：软件会告诉他“此序列无效”。如果有人正确转录了全部 24 个单词，软件将毫无歧义地导出相同的身份。词汇列表的选择也是经过深思熟虑的：BIP39 词汇表中的单词都很短，彼此互不相同，没有变音符号，旨在最大限度地减少语音和拼写上的混淆。这是一个旨在让人们能够无误地记忆、书写和口述的词汇表。

从短语到密钥

这二十四个单词并非签署消息的加密密钥。它们是原始熵的可恢复表示，通过名为 PBKDF2 的确定性过程，转换为六十四字节的种子（seed）。从该种子中，同样以确定性的方式衍生出用户使用的具体加密密钥：用于签署的私钥和用于验证签署而发布的相应公钥。不同系统中存在相同的机制：加密货币使用 secp256k1 曲线；Signal 协议和许多现代系统在 Curve25519 曲线上使用 Ed25519。对于像 Ed25519 这样的特定曲线，BIP32 和 SLIP-0010 标准采用该六十四字节种子，并确定性地衍生出构成有效签署密钥的三十二字节——这正是下一节代码示例开头的相同三十二字节。

这是整个行业向用户展示该机制的标准方式——加密货币钱包、去中心化身份管理器、Signal 的持久身份部分，以及 Solo2——：在实践中，用户永远看不到种子或衍生密钥。他在创建身份时看到这二十四个单词，并可以选择将它们记在纸上。当他想要迁移身份时，这些单词随后会在他的设备之间传输：他将它们输入到新应用程序中，应用程序衍生出相同的种子、相同的密钥和相同的身份。这是一种便携的、加密强度高且在合理范围内可记忆的机制。

如何使用密钥签署（Zig 笔触）

在 Zig 中，一旦拥有从二十四个单词衍生的三十二字节种子，使用 Ed25519 签署消息仅需几行代码：

```
const std = @import("std");
const Ed25519 = std.crypto.sign.Ed25519;

// 'semilla' son los 32 bytes derivados de las 24 palabras.
const par = Ed25519.KeyPair.create(semilla);

// Firmar un mensaje con la clave privada:
const mensaje = "Este mensaje lo escribí yo.";
const firma = try par.sign(mensaje, null);

// Cualquiera con la clave pública del par puede verificar:
try Ed25519.Signature.verify(firma, mensaje, par.public_key);
```

签署操作生成六十四个字节——称为签名——这只能由相应的私钥生成。验证是公开的：任何拥有公钥的人都可以检查签名是否与消息对应。没有私钥，没有人可以为该消息生成有效的签名；拥有公钥，所有人都可以检测签名是否有效。这种不对称性使得签署者能够在不共享秘密的情况下证明其作者身份。

前面的例子是手册中的最小版本。在 Solo2 的真实代码中，这个链条穿过两个文件：一个是用 JavaScript 编写的，运行在用户的浏览器中，负责从 24 个单词中重建熵；另一个是用 Zig 编写的，位于 *zcatcrypto* 库中，它接收该熵并导出具体的加密密钥。从浏览器端开始：

```
// solo2/web-app/js/lib/bip39.js
async function mnemonicToEntropy(mnemonic, lang) {
    const validation = await validateMnemonic(mnemonic, lang);
    if (!validation.valid) {
        return { entropy: null, valid: false, error: validation.error };
    }
    const wordlist = WORDLISTS[lang || 'en'];
    const words = mnemonic.trim().split(/\s+/);

    // Cada palabra aporta 11 bits (su índice en la lista de 2048).
    let bits = '';
    for (let i = 0; i < words.length; i++) {
        bits += wordlist.indexOf(words[i]).toString(2).padStart(11, '0');
    }

    // 24 palabras = 264 bits. Los primeros 256 son la entropía.
    const entropyBytes = new Uint8Array(32);
    for (let j = 0; j < 32; j++) {
        entropyBytes[j] = parseInt(bits.slice(j * 8, (j + 1) * 8), 2);
    }
    return { entropy: entropyBytes, valid: true };
}
```

那 32 字节的熵，连同在同一步骤中导出的另外 32 字节，一起传输到 Zig 的 WebAssembly 模块，该模块生成真正的 Ed25519 密钥。完整的函数及其最后的内存清理只需一个屏幕就能显示：

```
// zcatcrypto/wasm/bindings/identity.zig
const Ed25519 = std.crypto.sign.Ed25519;
const X25519  = std.crypto.dh.X25519;

export fn identity_generate() ?*IdentityHandle {
    var seed: [64]u8 = undefined;
    if (!common.getRandomBytes(&seed)) return null;

    const handle = common.wasm_allocator.create(IdentityHandle) catch return null;

    // Bytes 0..31: semilla determinista del par Ed25519 (firma).
    const sign_kp = Ed25519.KeyPair.generateDeterministic(seed[0..32].*) catch {
        common.wasm_allocator.destroy(handle);
        return null;
    };
    handle.sign_secret = sign_kp.secret_key.toBytes();
    handle.sign_public = sign_kp.public_key.toBytes();

    // Bytes 32..63: secreto X25519 (para acordar claves de cifrado con el otro).
    handle.exchange_secret = seed[32..64].*;
    handle.exchange_public = X25519.recoverPublicKey(handle.exchange_secret) catch {
        common.wasm_allocator.destroy(handle);
        return null;
```

```
    };

    @memset(&seed, 0);  // Borra la semilla de la memoria.
    return handle;
}
```

有两个细节值得注意。第一：同一个种子（seed）总是产生相同的密钥对——正是这一点使得通过在新设备上输入 24 个单词来恢复身份成为可能。第二：种子在最后一行被明确地从内存中擦除。过了那个点，即使是函数本身也无法重建密钥；用户的单词将是唯一的来源。

给那些想用小数字验证的人。 签名方案可以用足够小的数字完整地演示，以便进行手工计算。那些不想涉及算术的人可以跳过这一块而不丢失文章的线索；那些想一步步看机制运行的人可以在这里找到它。**公开规则**，任何人都可以阅读：素数 $p = 23$（在真实的 Ed25519 中，它大约有 77 位数字；我们使用 23 是为了让计算能在一页纸内完成），基数 $g = 2$，其在该群中的阶为 $q = 11$，以及使用 g 的所有算术运算都是 *módulo p* 并且所有指数都 *módulo q* 约简的约定。**私有选择**，仅此一个且绝不分享：秘密 $x = 6$。这就是身份。

第 1 步 — 身份的公开部分。 计算一次并公开。

$$y = g^x \bmod p$$

$$y = 2^6 \bmod 23 = 64 \bmod 23 = 18$$

身份的公开部分是 **18**。任何人都可以获取它并使用它来验证使用该身份生成的签名。仅通过观察 18，没有人能找回秘密 6：这就是我们最后会再谈到的离散对数问题。

第 2 步 — 对消息签名。 身份持有者想要对消息 $m = 7$ 签名。他首先选择一个新的随机值 $k = 4$，该值仅使用一次且绝不分享（在真实的 Ed25519 中，k 是从消息和秘密中确定性地导出的，以避免重复使用的危险，但它起到的作用正是这个）。然后他计算三个数字：

$$r = g^k \bmod p = 2^4 \bmod 23 = 16$$

$$e = H(r, m) \bmod q = (16 + 7) \bmod 11 = 1$$

$$s = (k + x \cdot e) \bmod q = (4 + 6 \cdot 1) \bmod 11 = 10$$

签名是数对 **(r, s) = (16, 10)**。它与消息一起公开传输。任何人都可以阅读它。教学说明：在真实的 Ed25519 中，函数 H 是加密强度极高的 SHA-512；这里我们使用简化的 $e = (r + m) \bmod q$，以便读者无需计算哈希即可遵循这些步骤。算法结构是一样的。

第 3 步 — 验证签名。 验证者拥有公开部分 $y = 18$、消息 $m = 7$ 和签名 $(r, s) = (16, 10)$。他以同样的方式重建 e —— $e = (16 + 7) \bmod 11 = 1$ —— 并检查这个等式是否成立：

$$g^s \bmod p \stackrel{?}{=} r \cdot y^e \bmod p$$

分别计算两边：

Izquierda: $2^{10} \bmod 23 = 1024 \bmod 23 = 12$

Derecha: $16 \cdot 18^{1} \bmod 23 = 288 \bmod 23 = 12$

两边都得到 **12**。签名有效。拥有公开部分 18 的任何人都能得出这个结论，而无需知道秘密是 6。

那么试图伪造的第三方呢？ 艾娃看到了通道中传输的所有公开内容：$p = 23$, $g = 2$, $q = 11$, $y = 18$, $m = 7$, $r = 16$, $s = 10$。为了以该身份的名义签署一条*不同的*消息，她需要知道 x。她唯一的途径是问自己：“对于哪个指数 x，$2^{x} \bmod 23 = 18$ 成立？”。对于 $p = 23$，她可以尝试 0, 1, 2, 3, … 并在几秒钟内找到它。但是，如果用真实 Ed25519 尺寸的素数替换 23，可能指数的空间将超过可观测宇宙中的原子数量。**今天人类已知的任何算法都无法在不到数十亿年的时间内遍历那个空间。** 这就是支持上一篇文章中 Diffie-Hellman 的同一个离散对数问题，这里应用到了签名方案中。

我们刚才经历的*正是* Schnorr，Ed25519 是其适应椭圆曲线的一个变体。在真实的 Ed25519 中，所有操作都是在特定曲线（Curve25519）的点上进行的，而不是在模素数的整数上进行的，且函数 H 是 SHA-512，而不是我们上面使用的玩具加法。这两个替换是实现上的调整——获得抗暴力破解的加密强度，为 k 获得额外的安全属性。算法结构、三个操作以及不对称性的原因都是一样的。

这里适合稍作停留，因为整个链条乍一看可能会与三原语中的另一个混淆：哈希。它不是。哈希是一个进行压缩的唯一函数——输入许多字节，输出一个短的足迹，路径到此为止。加密身份是一对数学上的互补：秘密保留并签名；其公开对应物发布并验证。哈希在单一方向上折叠信息，而身份在两半之间建立了一种不对称性。哈希证明说了什么；身份证明是谁说的。

短语不是什么

有三个常见的误区需要澄清。短语并非严格意义上的密码：它不会与存储在服务器上的指纹进行对比；它被输入到用户的设备中，以通过数学方式重建身份。短语无法恢复：如果丢失，没有人可以索取；如果被复制，身份也会被复制。短语不是可以从身份中分离出来的凭证：短语*就是*身份。拥有它的人就可以作为该身份行事，无需额外的许可，无需授权过程，也无法恢复。

正是这第三个属性改变了事情的分量。丢失密码是行政上的麻烦。丢失加密身份就是丢失身份本身。第三方发现带有短语的纸张不是账户被盗的风险：而是移交了整个身份。系统的承诺——没有人可以撤销你的身份或任意屏蔽你——与责任密不可分——即你是没有人可以为你恢复的东西的唯一保管人。

承诺与分量

加密身份模型通常被称为*自主身份*（英语文献中为 self-sovereign）。词语的选择是经过深思熟虑的，并且非常准确地描述了这种状态。用户在几乎是中世纪的意义上对自己的身份拥有主权：没有任何国王、任何发行者、任何中央机构授予它；上述任何一方也无法撤回它。但同时，就像中世纪的君主一样，用户要承担其错误的全部后果：如果丢失印章，没有摄政者可以代替其做出决定。

在由第三方管理的身份和自主身份之间进行选择并没有唯一的全球正确答案。对于一个无关紧要的论坛账户，托管身份可能与风险成正比。但对于签署具有法律约束力文件的职业身份、守护个人储蓄的经济

身份、与托付敏感信息的客户进行专业沟通的身份，情况就不同了。在那里，问题不再是“方便吗？”，而变成了“除了我之外，谁有权代表我行事，以及在什么情况下？”

这种机制在实际系统中出现在哪里

BIP39 诞生于 2013 年的 Bitcoin 世界，并迅速扩展到整个加密货币生态系统：今天，任何严肃的钱包都接受 12 或 24 个词的 BIP39 短语作为其持有人经济身份的备份。在加密货币之外，同样的基础概念——无需中介即可证明作者身份的加密对——也出现在具有不同语法的其他系统中。系统管理员用于访问其服务器的 SSH 密钥就是一个经典案例：管理员在自己的机器上保存私钥，而公钥则被复制到每个服务器上；没有任何类似于中心化服务的实体介入。Signal 协议在设备上使用具有持久密钥材料的 Ed25519；欧洲的 eIDAS 在其合格签名部分也基于相同的加密原理，不同之处在于密钥由合格的信任服务提供商而非用户保管。

本刊的出版平台 Solo2 使用 24 个词的 BIP39 短语作为每个用户的身份。用户在创建账户时会看到这些词一次。它们不会存储在 Solo2 或任何其他人的任何服务器上：如果用户记录并妥善保管它们，就能永远保持其身份。如果丢失了，就彻底丢失了。这是中间没有运营商的架构的必然结果：如果 Solo2 能够将身份归还给丢失身份的用户，那么它也能将其交给任何向 Solo2 施压要求获取身份的人。

供专业读者参考

为那些评估在专业背景下采用自主 (autosoberana) 加密身份的人提供四点考虑：

1. 短语即身份。物理保管——纸张、在不同地点的多份副本、乃至用于长期使用的刻制金属——比增加攻击面却不降低丢失风险的数字保管提供更多的保障。
2. 没有恢复手段。在设计流程时假设总有一天主副本会丢失，比在丢失那天发现这一事实要明智得多。地理位置分散的第二份副本几乎可以解决所有场景。
3. 这与 eIDAS 合格证书不同。对于欧盟境内的合格签名——公证契据、与行政部门的某些程序——法律要求由保管密钥的合格提供者负责。自主 (autosoberana) 加密身份适用于专业沟通和具有证据价值的文件签署，但在法律要求的特定情况下，它并不能自动替代合格证书。
4. 如果身份需要转交——遗产、专业继承、停止活动——建议在事前而非事后准备好程序。使用火漆 (lacre) 密封的信封、给遗嘱执行人的指示、在公证处寄存，都是与资产的加密性质完全兼容的经典安排。

本文总结了开启本系列的三个概念——hash、加密、身份。这三个理念环环相扣：hash 提供不可篡改的指纹，加密在没有受信任第三方的情况下提供机密性，身份在没有授予第三方的情况下提供作者身份。这三者共同拥有一个同样非意识形态的属性：它们将传统上由运营商掌握的技术能力，从服务管理者手中转移到了使用者手中。同时也随之转移了责任。诚实地探讨其中任何一个，都要求同时探讨另外两个。

来源及延伸阅读

- Palatinus, M.; Rusnak, P.; Voisine, A.; Bowe, S. — *BIP-0039: Mnemonic code for generating deterministic keys*，2013 年的 Bitcoin 改进提案。加密行业恢复短语的行业标准。
- RFC 8032 — Edwards-Curve Digital Signature Algorithm (EdDSA)，包括 Ed25519。IETF，2017 年 1 月。当代工业大部分领域所使用的签名方案的标准规范。
- RFC 2898 — PKCS #5: Password-Based Cryptography Specification，2.0 版本。IETF，2000 年 9 月。定义了从短语到种子 (seed) 的 BIP39 派生中使用的 PBKDF2 算法。
- 法规 (EU) 910/2014 (eIDAS) 及其通过法规 (EU) 2024/1183 (eIDAS 2) 的演进——欧洲电子身份和合格签名框架。与自主模式不同的制度，但在概念上由相同的加密原语支撑。
- Allen, C. — *The Path to Self-Sovereign Identity* (2016)。关于自主模型原则和承诺的经典文献，虽然较早但对于理解当代解决方案系列仍具参考价值。

← 上一页商业模式作为信任的信号下一页 →作为专业实践的自托管 (Self-hosting)

最近阅读

- 反思 · 2026年6月29日 你并不匿名
- 思考 · 2026年5月27日 签名无法解决的问题
- 分析 · 2026年5月26日 真实隐私 vs 表象隐私：你应该问自己的问题

随身携带本文，以备不时之需。

↓ Markdown ↓ 纯文本 ↓ PDF

文件将下载到您的设备中。您可以从那里将其保存、导入 Solo2 或在任何地方共享。Cuadernos 不会为您决定文件的去向。

火漆印章 · SHA-256 2b34b2d3ce27845aed33c6ad9c4f0bcc6158449e769bfede520f1ef6959b33aa

Cuadernos Lacre · Menzuri Gestión S.L. 的出版物 ·
由 R.Eugenio 撰写 · 由 Solo2 团队编辑。