```
  console.log('Checking calculated digest against published digests...');
  for (const [sha, filename] of shaPairs) {
    if (filename === firefoxBinaryFilename) {
      if (sha === firefoxHexDigest) {
        verified = true;
        break;
      }
    }
  }

  console.log('Verified:', verified);
})();

// Fetching Firefox binary...
// Calculating Firefox digest...
// Fetching published binary digests...
// Checking calculated digest against published digests...
// Verified: true
```

2. CryptoKey 与算法

如果没了密钥，那密码学也就没什么意义了。SubtleCrypto 对象使用 CryptoKey 类的实例来生成密钥。CryptoKey 类支持多种加密算法，允许控制密钥抽取和使用。

CryptoKey 类支持以下算法，按各自的父密码系统归类。

- RSA（Rivest-Shamir-Adleman）：公钥密码系统，使用两个大素数获得一对公钥和私钥，可用于签名/验证或加密/解密消息。RSA 的陷门函数被称为**分解难题**（factoring problem）。
- RSASSA-PKCS1-v1_5：RSA 的一个应用，用于使用私钥给消息签名，允许使用公钥验证签名。
  - SSA（Signature Schemes with Appendix），表示算法支持签名生成和验证操作。
  - PKCS1（Public-Key Cryptography Standards #1），表示算法展示出的 RSA 密钥必需的数学特性。
  - RSASSA-PKCS1-v1_5 是确定性的，意味着同样的消息和密钥每次都会生成相同的签名。
- RSA-PSS：RSA 的另一个应用，用于签名和验证消息。
  - PSS（Probabilistic Signature Scheme），表示生成签名时会加盐以得到随机签名。
  - 与 RSASSA-PKCS1-v1_5 不同，同样的消息和密钥每次都会生成不同的签名。
  - 与 RSASSA-PKCS1-v1_5 不同，RSA-PSS 有可能约简到 RSA 分解难题的难度。
  - 通常，虽然 RSASSA-PKCS1-v1_5 仍被认为是安全的，但 RSA-PSS 应该用于代替 RSASSA-PKCS1-v1_5。
- RSA-OAEP：RSA 的一个应用，用于使用公钥加密消息，用私钥来解密。
  - OAEP（Optimal Asymmetric Encryption Padding），表示算法利用了 Feistel 网络在加密前处理未加密的消息。
  - OAEP 主要将确定性 RSA 加密模式转换为概率性加密模式。
- ECC（Elliptic-Curve Cryptography）：公钥密码系统，使用一个素数和一个椭圆曲线获得一对公钥和私钥，可用于签名/验证消息。ECC 的陷门函数被称为**椭圆曲线离散对数问题**（elliptic curve discrete logarithm problem）。ECC 被认为优于 RSA。虽然 RSA 和 ECC 在密码学意义上都很强，但 ECC 密钥比 RSA 密钥短，而且 ECC 密码学操作比 RSA 操作快。
- ECDSA（Elliptic Curve Digital Signature Algorithm）：ECC 的一个应用，用于签名和验证消息。这个算法是**数字签名算法**（DSA，Digital Signature Algorithm）的一个椭圆曲线风格的变体。

- ECDH（Elliptic Curve Diffie-Hellman）：ECC 的密钥生成和密钥协商应用，允许两方通过公开通信渠道建立共享的机密。这个算法是 Diffie-Hellman 密钥交换（DH，Diffie-Hellman key exchange）协议的一个椭圆曲线风格的变体。
- AES（Advanced Encryption Standard）：对称密钥密码系统，使用派生自置换组合网络的分组密码加密和解密数据。AES 在不同**模式**下使用，不同模式算法的特性也不同。
- AES-CTR：AES 的**计数器模式**（counter mode）。这个模式使用递增计数器生成其密钥流，其行为类似密文流。使用时必须为其提供一个随机数，用作初始化向量。AES-CTR 加密/解密可以并行。
- AES-CBC：AES 的**密码分组链模式**（cipher block chaining mode）。在加密纯文本的每个分组之前，先使用之前密文分组求 XOR，也就是名字中的“链”。使用一个初始化向量作为第一个分组的 XOR 输入。
- AES-GCM：AES 的**伽罗瓦/计数器模式**（Galois/Counter mode）。这个模式使用计数器和初始化向量生成一个值，这个值会与每个分组的纯文本计算 XOR。与 CBC 不同，这个模式的 XOR 输入不依赖之前分组密文。因此 GCM 模式可以并行。由于其卓越的性能，AES-GCM 在很多网络安全协议中得到了应用。
- AES-KW：AES 的**密钥包装模式**（key wrapping mode）。这个算法将加密密钥包装为一个可移植且加密的格式，可以在不信任的渠道中传输。传输之后，接收方可以解包密钥。与其他 AES 模式不同，AES-KW 不需要初始化向量。
- HMAC（Hash-Based Message Authentication Code）：用于生成**消息认证码**的算法，用于验证通过不可信网络接收的消息没有被修改过。两方使用散列函数和共享私钥来签名和验证消息。
- KDF（Key Derivation Functions）：可以使用散列函数从主密钥获得一个或多个密钥的算法。KDF 能够生成不同长度的密钥，也能把密钥转换为不同格式。
- HKDF（HMAC-Based Key Derivation Function）：密钥推导函数，与高熵输入（如已有密钥）一起使用。
- PBKDF2（Password-Based Key Derivation Function 2）：密钥推导函数，与低熵输入（如密钥字符串）一起使用。

> **注意** `CryptoKey` 支持很多算法，但其中只有部分算法能够用于 `SubtleCrypto` 的方法。要了解哪个方法支持什么算法，可以参考 W3C 网站上 Web Cryptography API 规范的“Algorithm Overview”。

3. 生成 `CryptoKey`

使用 `SubtleCrypto.generateKey()`方法可以生成随机 `CryptoKey`，这个方法返回一个期约，解决为一个或多个 `CryptoKey` 实例。使用时需要给这个方法传入一个指定目标算法的参数对象、一个表示密钥是否可以从 `CryptoKey` 对象中提取出来的布尔值，以及一个表示这个密钥可以与哪个 `SubtleCrypto` 方法一起使用的字符串数组（`keyUsages`）。

由于不同的密码系统需要不同的输入来生成密钥，上述参数对象为每种密码系统都规定了必需的输入：

- RSA 密码系统使用 `RsaHashedKeyGenParams` 对象；

- ECC 密码系统使用 EcKeyGenParams 对象；
- HMAC 密码系统使用 HmacKeyGenParams 对象；
- AES 密码系统使用 AesKeyGenParams 对象。

keyUsages 对象用于说明密钥可以与哪个算法一起使用。至少要包含下列中的一个字符串：

- encrypt
- decrypt
- sign
- verify
- deriveKey
- deriveBits
- wrapKey
- unwrapKey

假设要生成一个满足如下条件的对称密钥：

- 支持 AES-CTR 算法；
- 密钥长度 128 位；
- 不能从 CryptoKey 对象中提取；
- 可以跟 encrypt()和 decrypt()方法一起使用。

那么可以参考如下代码：

```
(async function() {
  const params = {
    name: 'AES-CTR',
    length: 128
  };

  const keyUsages = ['encrypt', 'decrypt'];

  const key = await crypto.subtle.generateKey(params, false, keyUsages);

  console.log(key);
  // CryptoKey {type: "secret", extractable: true, algorithm: {...}, usages: Array(2)}
})();
```

假设要生成一个满足如下条件的非对称密钥：

- 支持 ECDSA 算法；
- 使用 P-256 椭圆曲线；
- 可以从 CryptoKey 中提取；
- 可以跟 sign()和 verify()方法一起使用。

那么可以参考如下代码：

```
(async function() {
  const params = {
    name: 'ECDSA',
    namedCurve: 'P-256'
  };

  const keyUsages = ['sign', 'verify'];

  const {publicKey, privateKey} = await crypto.subtle.generateKey(params, true,
      keyUsages);
```

```
  console.log(publicKey);
  // CryptoKey {type: "public", extractable: true, algorithm: {...}, usages: Array(1)}

  console.log(privateKey);
  // CryptoKey {type: "private", extractable: true, algorithm: {...}, usages: Array(1)}
})();
```

4. 导出和导入密钥

如果密钥是可提取的，那么就可以在 CryptoKey 对象内部暴露密钥原始的二进制内容。使用 exportKey()方法并指定目标格式（"raw"、"pkcs8"、"spki"或"jwk"）就可以取得密钥。这个方法返回一个期约，解决后的 ArrayBuffer 中包含密钥：

```
(async function() {
  const params = {
    name: 'AES-CTR',
    length: 128
  };
  const keyUsages = ['encrypt', 'decrypt'];

  const key = await crypto.subtle.generateKey(params, true, keyUsages);

  const rawKey = await crypto.subtle.exportKey('raw', key);

  console.log(new Uint8Array(rawKey));
  // Uint8Array[93, 122, 66, 135, 144, 182, 119, 196, 234, 73, 84, 7, 139, 43, 238,
  // 110]
})();
```

与 exportKey()相反的操作要使用 importKey()方法实现。importKey()方法的签名实际上是 generateKey()和 exportKey()的组合。下面的方法会生成密钥、导出密钥，然后再导入密钥：

```
(async function() {
  const params = {
    name: 'AES-CTR',
    length: 128
  };
  const keyUsages = ['encrypt', 'decrypt'];
  const keyFormat = 'raw';
  const isExtractable = true;

  const key = await crypto.subtle.generateKey(params, isExtractable, keyUsages);

  const rawKey = await crypto.subtle.exportKey(keyFormat, key);

  const importedKey = await crypto.subtle.importKey(keyFormat, rawKey, params.name,
      isExtractable, keyUsages);

  console.log(importedKey);
  // CryptoKey {type: "secret", extractable: true, algorithm: {...}, usages: Array(2)}
})();
```

5. 从主密钥派生密钥

使用 SubtleCrypto 对象可以通过可配置的属性从已有密钥获得新密钥。SubtleCrypto 支持一个 deriveKey()方法和一个 deriveBits()方法，前者返回一个解决为 CryptoKey 的期约，后者返回一个解决为 ArrayBuffer 的期约。

> **注意**　deriveKey()与 deriveBits()的区别很微妙，因为调用 deriveKey()实际上与调用 deriveBits()之后再把结果传给 importKey()相同。

deriveBits()方法接收一个算法参数对象、主密钥和输出的位长作为参数。当两个人分别拥有自己的密钥对，但希望获得共享的加密密钥时可以使用这个方法。下面的例子使用 ECDH 算法基于两个密钥对生成了对等密钥，并确保它们派生相同的密钥位：

```
(async function() {
  const ellipticCurve = 'P-256';
  const algoIdentifier = 'ECDH';
  const derivedKeySize = 128;

  const params = {
    name: algoIdentifier,
    namedCurve: ellipticCurve
  };

  const keyUsages = ['deriveBits'];

  const keyPairA = await crypto.subtle.generateKey(params, true, keyUsages);
  const keyPairB = await crypto.subtle.generateKey(params, true, keyUsages);

  // 从 A 的公钥和 B 的私钥派生密钥位
  const derivedBitsAB = await crypto.subtle.deriveBits(
      Object.assign({ public: keyPairA.publicKey }, params),
      keyPairB.privateKey,
      derivedKeySize);

  // 从 B 的公钥和 A 的私钥派生密钥位
  const derivedBitsBA = await crypto.subtle.deriveBits(
      Object.assign({ public: keyPairB.publicKey }, params),
      keyPairA.privateKey,
      derivedKeySize);

  const arrayAB = new Uint32Array(derivedBitsAB);
  const arrayBA = new Uint32Array(derivedBitsBA);

  // 确保密钥数组相等
  console.log(
      arrayAB.length === arrayBA.length &&
      arrayAB.every((val, i) => val === arrayBA[i])); // true
})();
```

deriveKey()方法是类似的，只不过返回的是 CryptoKey 的实例而不是 ArrayBuffer。下面的例子基于一个原始字符串，应用 PBKDF2 算法将其导入一个原始主密钥，然后派生了一个 AES-GCM 格式的新密钥：

```
(async function() {
  const password = 'foobar';
  const salt = crypto.getRandomValues(new Uint8Array(16));
  const algoIdentifier = 'PBKDF2';
  const keyFormat = 'raw';
  const isExtractable = false;

  const params = {
```

```
    name: algoIdentifier
  };

  const masterKey = await window.crypto.subtle.importKey(
    keyFormat,
    (new TextEncoder()).encode(password),
    params,
    isExtractable,
    ['deriveKey']
  );

  const deriveParams = {
    name: 'AES-GCM',
    length: 128
  };

  const derivedKey = await window.crypto.subtle.deriveKey(
    Object.assign({salt, iterations: 1E5, hash: 'SHA-256'}, params),
    masterKey,
    deriveParams,
    isExtractable,
    ['encrypt']
  );

  console.log(derivedKey);
  // CryptoKey {type: "secret", extractable: false, algorithm: {...}, usages: Array(1)}
})();
```

6. 使用非对称密钥签名和验证消息

通过 SubtleCrypto 对象可以使用公钥算法用私钥生成签名，或者用公钥验证签名。这两种操作分别通过 SubtleCrypto.sign()和 SubtleCrypto.verify()方法完成。

签名消息需要传入参数对象以指定算法和必要的值、CryptoKey 和要签名的 ArrayBuffer 或 ArrayBufferView。下面的例子会生成一个椭圆曲线密钥对，并使用私钥签名消息：

```
(async function() {
  const keyParams = {
    name: 'ECDSA',
    namedCurve: 'P-256'
  };

  const keyUsages = ['sign', 'verify'];

  const {publicKey, privateKey} = await crypto.subtle.generateKey(keyParams, true,
      keyUsages);

  const message = (new TextEncoder()).encode('I am Satoshi Nakamoto');

  const signParams = {
    name: 'ECDSA',
    hash: 'SHA-256'
  };

  const signature = await crypto.subtle.sign(signParams, privateKey, message);

  console.log(new Uint32Array(signature));
  // Uint32Array(16) [2202267297, 698413658, 1501924384, 691450316, 778757775, ... ]
})();
```

希望通过这个签名验证消息的人可以使用公钥和 SubtleCrypto.verify()方法。这个方法的签名几乎与 sign()相同，只是必须提供公钥以及签名。下面的例子通过验证生成的签名扩展了前面的例子：

```
(async function() {
  const keyParams = {
    name: 'ECDSA',
    namedCurve: 'P-256'
  };

  const keyUsages = ['sign', 'verify'];

  const {publicKey, privateKey} = await crypto.subtle.generateKey(keyParams, true,
      keyUsages);

  const message = (new TextEncoder()).encode('I am Satoshi Nakamoto');

  const signParams = {
    name: 'ECDSA',
    hash: 'SHA-256'
  };

  const signature = await crypto.subtle.sign(signParams, privateKey, message);

  const verified = await crypto.subtle.verify(signParams, publicKey, signature,
      message);

  console.log(verified); // true
})();
```

7. 使用对称密钥加密和解密

SubtleCrypto 对象支持使用公钥和对称算法加密和解密消息。这两种操作分别通过 SubtleCrypto.encrypt()和 SubtleCrypto.decrypt()方法完成。

加密消息需要传入参数对象以指定算法和必要的值、加密密钥和要加密的数据。下面的例子会生成对称 AES-CBC 密钥，用它加密消息，最后解密消息：

```
(async function() {
  const algoIdentifier = 'AES-CBC';

  const keyParams = {
    name: algoIdentifier,
    length: 256
  };

  const keyUsages = ['encrypt', 'decrypt'];

  const key = await crypto.subtle.generateKey(keyParams, true,
      keyUsages);

  const originalPlaintext = (new TextEncoder()).encode('I am Satoshi Nakamoto');

  const encryptDecryptParams = {
    name: algoIdentifier,
    iv: crypto.getRandomValues(new Uint8Array(16))
  };

  const ciphertext = await crypto.subtle.encrypt(encryptDecryptParams, key,
      originalPlaintext);
```

```
  console.log(ciphertext);
  // ArrayBuffer(32) {}

  const decryptedPlaintext = await crypto.subtle.decrypt(encryptDecryptParams, key,
      ciphertext);

  console.log((new TextDecoder()).decode(decryptedPlaintext));
  // I am Satoshi Nakamoto
})();
```

8. 包装和解包密钥

SubtleCrypto 对象支持包装和解包密钥，以便在非信任渠道传输。这两种操作分别通过 SubtleCrypto.wrapKey()和 SubtleCrypto.unwrapKey()方法完成。

包装密钥需要传入一个格式字符串、要包装的 CryptoKey 实例、要执行包装的 CryptoKey，以及一个参数对象用于指定算法和必要的值。下面的例子生成了一个对称 AES-GCM 密钥，用 AES-KW 来包装这个密钥，最后又将包装的密钥解包：

```
(async function() {
  const keyFormat = 'raw';
  const extractable = true;

  const wrappingKeyAlgoIdentifier = 'AES-KW';
  const wrappingKeyUsages = ['wrapKey', 'unwrapKey'];
  const wrappingKeyParams = {
    name: wrappingKeyAlgoIdentifier,
    length: 256
  };

  const keyAlgoIdentifier = 'AES-GCM';
  const keyUsages = ['encrypt'];
  const keyParams = {
    name: keyAlgoIdentifier,
    length: 256
  };

  const wrappingKey = await crypto.subtle.generateKey(wrappingKeyParams, extractable,
      wrappingKeyUsages);

  console.log(wrappingKey);
  // CryptoKey {type: "secret", extractable: true, algorithm: {...}, usages: Array(2)}

  const key = await crypto.subtle.generateKey(keyParams, extractable, keyUsages);

  console.log(key);
  // CryptoKey {type: "secret", extractable: true, algorithm: {...}, usages: Array(1)}

  const wrappedKey = await crypto.subtle.wrapKey(keyFormat, key, wrappingKey,
      wrappingKeyAlgoIdentifier);

  console.log(wrappedKey);
  // ArrayBuffer(40) {}

  const unwrappedKey = await crypto.subtle.unwrapKey(keyFormat, wrappedKey,
      wrappingKey, wrappingKeyParams, keyParams, extractable, keyUsages);

  console.log(unwrappedKey);
  // CryptoKey {type: "secret", extractable: true, algorithm: {...}, usages: Array(1)}
})()
```

20.13　小结

除了定义新标签，HTML5 还定义了一些 JavaScript API。这些 API 可以为开发者提供更便捷的 Web 接口，暴露堪比桌面应用的能力。本章主要介绍了以下 API。

- Atomics API 用于保护代码在多线程内存访问模式下不发生资源争用。
- `postMessage()` API 支持从不同源跨文档发送消息，同时保证安全和遵循同源策略。
- Encoding API 用于实现字符串与缓冲区之间的无缝转换（越来越常见的操作）。
- File API 提供了发送、接收和读取大型二进制对象的可靠工具。
- 媒体元素`<audio>`和`<video>`拥有自己的 API，用于操作音频和视频。并不是每个浏览器都会支持所有媒体格式，使用 `canPlayType()`方法可以检测浏览器支持情况。
- 拖放 API 支持方便地将元素标识为可拖动，并在操作系统完成放置时给出回应。可以利用它创建自定义可拖动元素和放置目标。
- Notifications API 提供了一种浏览器中立的方式，以此向用户展示消通知弹层。
- Streams API 支持以全新的方式读取、写入和处理数据。
- Timing API 提供了一组度量数据进出浏览器时间的可靠工具。
- Web Components API 为元素重用和封装技术向前迈进提供了有力支撑。
- Web Cryptography API 让生成随机数、加密和签名消息成为一类特性。

第21章 错误处理与调试

本章内容

- 理解浏览器错误报告
- 处理错误
- 调试 JavaScript 代码

JavaScript 一直以来被认为是最难调试的编程语言之一，因为它是动态的，且多年来没有适当的开发工具。错误经常会以令人迷惑的浏览器消息形式抛出，比如"object expected"。这样的消息没有上下文，因此很难理解。ECMAScript 第 3 版致力于改进这个方面，引入了 try/catch 和 throw 语句，以及一些错误类型，以帮助开发者在出错时正确地处理它们。几年后，JavaScript 调试器和排错工具开始在浏览器中出现。到了 2008 年，大多数浏览器支持一些 JavaScript 调试能力。

有了适当的语言和开发工具，Web 开发者如今已可以实现适当的错误处理并找到问题的原因。

21.1 浏览器错误报告

所有主流桌面浏览器，包括 IE/Edge、Firefox、Safari、Chrome 和 Opera，都提供了向用户报告错误的机制。默认情况下，所有浏览器都会隐藏错误信息。一个原因是除了开发者之外这些信息对别人没什么用，另一个原因是网页在正常操作中报错的固有特性。

21.1.1 桌面控制台

所有现代桌面浏览器都会通过控制台暴露错误。这些错误可以显示在开发者工具内嵌的控制台中。在前面提到的所有浏览器中，访问开发者工具的路径是相似的。可能最简单的查看错误的方式就是在页面上单击鼠标右键，然后在上下文菜单中选择 Inspect（检查）或 Inspect Element（检查元素），然后再单击 Console（控制台）选项卡。

要直接进入控制台，不同操作系统和浏览器支持不同的快捷键，如下表所示。

浏 览 器	Windows/Linux	Mac
Chrome	Ctrl+Shfit+J	Cmd+Opt+J
Firefox	Ctrl+Shfit+K	Cmd+Opt+K
IE/Edge	F12，然后 Ctrl+2	不适用
Opera	Ctrl+Shift+I	Cmd+Opt+I
Safari	不适用	Cmd+Opt+C