```
alert(a === b);   // false
alert(pz === nz); // true

m.set(a, "foo");
m.set(pz, "bar");

alert(m.get(b));  // foo
alert(m.get(nz)); // bar
```

> **注意**　SameValueZero 是 ECMAScript 规范新增的相等性比较算法。关于 ECMAScript 的相等性比较，可以参考 MDN 文档中的文章“Equality Comparisons and Sameness”。

6.4.2　顺序与迭代

与 `Object` 类型的一个主要差异是，`Map` 实例会维护键值对的插入顺序，因此可以根据插入顺序执行迭代操作。

映射实例可以提供一个迭代器（`Iterator`），能以插入顺序生成`[key, value]`形式的数组。可以通过 `entries()`方法（或者 `Symbol.iterator` 属性，它引用 `entries()`）取得这个迭代器：

```
const m = new Map([
  ["key1", "val1"],
  ["key2", "val2"],
  ["key3", "val3"]
]);

alert(m.entries === m[Symbol.iterator]); // true

for (let pair of m.entries()) {
  alert(pair);
}
// [key1,val1]
// [key2,val2]
// [key3,val3]

for (let pair of m[Symbol.iterator]()) {
  alert(pair);
}
// [key1,val1]
// [key2,val2]
// [key3,val3]
```

因为 `entries()`是默认迭代器，所以可以直接对映射实例使用扩展操作，把映射转换为数组：

```
const m = new Map([
  ["key1", "val1"],
  ["key2", "val2"],
  ["key3", "val3"]
]);

console.log([...m]); // [[key1,val1],[key2,val2],[key3,val3]]
```

如果不使用迭代器，而是使用回调方式，则可以调用映射的 `forEach(callback, opt_thisArg)` 方法并传入回调，依次迭代每个键/值对。传入的回调接收可选的第二个参数，这个参数用于重写回调内部 `this` 的值：

```
const m = new Map([
  ["key1", "val1"],
  ["key2", "val2"],
  ["key3", "val3"]
]);

m.forEach((val, key) => alert(`${key} -> ${val}`));
// key1 -> val1
// key2 -> val2
// key3 -> val3
```

keys()和values()分别返回以插入顺序生成键和值的迭代器：

```
const m = new Map([
  ["key1", "val1"],
  ["key2", "val2"],
  ["key3", "val3"]
]);

for (let key of m.keys()) {
  alert(key);
}
// key1
// key2
// key3

for (let key of m.values()) {
  alert(key);
}
// value1
// value2
// value3
```

键和值在迭代器遍历时是可以修改的，但映射内部的引用则无法修改。当然，这并不妨碍修改作为键或值的对象内部的属性，因为这样并不影响它们在映射实例中的身份：

```
const m1 = new Map([
  ["key1", "val1"]
]);

// 作为键的字符串原始值是不能修改的
for (let key of m1.keys()) {
  key = "newKey";
  alert(key);              // newKey
  alert(m1.get("key1"));   // val1
}

const keyObj = {id: 1};

const m = new Map([
  [keyObj, "val1"]
]);

// 修改了作为键的对象的属性，但对象在映射内部仍然引用相同的值
for (let key of m.keys()) {
  key.id = "newKey";
  alert(key);             // {id: "newKey"}
  alert(m.get(keyObj));   // val1
}
alert(keyObj);            // {id: "newKey"}
```

6.4.3　选择 Object 还是 Map

对于多数 Web 开发任务来说，选择 Object 还是 Map 只是个人偏好问题，影响不大。不过，对于在乎内存和性能的开发者来说，对象和映射之间确实存在显著的差别。

1. 内存占用

Object 和 Map 的工程级实现在不同浏览器间存在明显差异，但存储单个键/值对所占用的内存数量都会随键的数量线性增加。批量添加或删除键/值对则取决于各浏览器对该类型内存分配的工程实现。不同浏览器的情况不同，但给定固定大小的内存，Map 大约可以比 Object 多存储 50%的键/值对。

2. 插入性能

向 Object 和 Map 中插入新键/值对的消耗大致相当，不过插入 Map 在所有浏览器中一般会稍微快一点儿。对这两个类型来说，插入速度并不会随着键/值对数量而线性增加。如果代码涉及大量插入操作，那么显然 Map 的性能更佳。

3. 查找速度

与插入不同，从大型 Object 和 Map 中查找键/值对的性能差异极小，但如果只包含少量键/值对，则 Object 有时候速度更快。在把 Object 当成数组使用的情况下（比如使用连续整数作为属性），浏览器引擎可以进行优化，在内存中使用更高效的布局。这对 Map 来说是不可能的。对这两个类型而言，查找速度不会随着键/值对数量增加而线性增加。如果代码涉及大量查找操作，那么某些情况下可能选择 Object 更好一些。

4. 删除性能

使用 delete 删除 Object 属性的性能一直以来饱受诟病，目前在很多浏览器中仍然如此。为此，出现了一些伪删除对象属性的操作，包括把属性值设置为 undefined 或 null。但很多时候，这都是一种讨厌的或不适宜的折中。而对大多数浏览器引擎来说，Map 的 delete()操作都比插入和查找更快。如果代码涉及大量删除操作，那么毫无疑问应该选择 Map。

6.5　WeakMap

ECMAScript 6 新增的“弱映射”（WeakMap）是一种新的集合类型，为这门语言带来了增强的键/值对存储机制。WeakMap 是 Map 的“兄弟”类型，其 API 也是 Map 的子集。WeakMap 中的“weak”（弱），描述的是 JavaScript 垃圾回收程序对待“弱映射”中键的方式。

6.5.1　基本 API

可以使用 new 关键字实例化一个空的 WeakMap：

```
const wm = new WeakMap();
```

弱映射中的键只能是 Object 或者继承自 Object 的类型，尝试使用非对象设置键会抛出 TypeError。值的类型没有限制。

如果想在初始化时填充弱映射，则构造函数可以接收一个可迭代对象，其中需要包含键/值对数组。可迭代对象中的每个键/值都会按照迭代顺序插入新实例中：

```
const key1 = {id: 1},
      key2 = {id: 2},
```

```
      key3 = {id: 3};
// 使用嵌套数组初始化弱映射
const wm1 = new WeakMap([
  [key1, "val1"],
  [key2, "val2"],
  [key3, "val3"]
]);
alert(wm1.get(key1)); // val1
alert(wm1.get(key2)); // val2
alert(wm1.get(key3)); // val3

// 初始化是全有或全无的操作
// 只要有一个键无效就会抛出错误，导致整个初始化失败
const wm2 = new WeakMap([
  [key1, "val1"],
  ["BADKEY", "val2"],
  [key3, "val3"]
]);
// TypeError: Invalid value used as WeakMap key
typeof wm2;
// ReferenceError: wm2 is not defined

// 原始值可以先包装成对象再用作键
const stringKey = new String("key1");
const wm3 = new WeakMap([
  stringKey, "val1"
]);
alert(wm3.get(stringKey)); // "val1"
```

初始化之后可以使用 `set()`再添加键/值对，可以使用 `get()`和 `has()`查询，还可以使用 `delete()`删除：

```
const wm = new WeakMap();

const key1 = {id: 1},
      key2 = {id: 2};

alert(wm.has(key1)); // false
alert(wm.get(key1)); // undefined

wm.set(key1, "Matt")
  .set(key2, "Frisbie");

alert(wm.has(key1)); // true
alert(wm.get(key1)); // Matt

wm.delete(key1);     // 只删除这一个键/值对

alert(wm.has(key1)); // false
alert(wm.has(key2)); // true
```

`set()`方法返回弱映射实例，因此可以把多个操作连缀起来，包括初始化声明：

```
const key1 = {id: 1},
      key2 = {id: 2},
      key3 = {id: 3};

const wm = new WeakMap().set(key1, "val1");
```

```
wm.set(key2, "val2")
  .set(key3, "val3");

alert(wm.get(key1)); // val1
alert(wm.get(key2)); // val2
alert(wm.get(key3)); // val3
```

6.5.2　弱键

WeakMap 中“weak”表示弱映射的键是“弱弱地拿着”的。意思就是，这些键不属于正式的引用，不会阻止垃圾回收。但要注意的是，弱映射中值的引用可**不是**“弱弱地拿着”的。只要键存在，键/值对就会存在于映射中，并被当作对值的引用，因此就不会被当作垃圾回收。

来看下面的例子：

```
const wm = new WeakMap();

wm.set({}, "val");
```

set()方法初始化了一个新对象并将它用作一个字符串的键。因为没有指向这个对象的其他引用，所以当这行代码执行完成后，这个对象键就会被当作垃圾回收。然后，这个键/值对就从弱映射中消失了，使其成为一个空映射。在这个例子中，因为值也没有被引用，所以这对键/值被破坏以后，值本身也会成为垃圾回收的目标。

再看一个稍微不同的例子：

```
const wm = new WeakMap();

const container = {
  key: {}
};

wm.set(container.key, "val");

function removeReference() {
  container.key = null;
}
```

这一次，container 对象维护着一个对弱映射键的引用，因此这个对象键不会成为垃圾回收的目标。不过，如果调用了 removeReference()，就会摧毁键对象的最后一个引用，垃圾回收程序就可以把这个键/值对清理掉。

6.5.3　不可迭代键

因为 WeakMap 中的键/值对任何时候都可能被销毁，所以没必要提供迭代其键/值对的能力。当然，也用不着像 clear()这样一次性销毁所有键/值的方法。WeakMap 确实没有这个方法。因为不可能迭代，所以也不可能在不知道对象引用的情况下从弱映射中取得值。即便代码可以访问 WeakMap 实例，也没办法看到其中的内容。

WeakMap 实例之所以限制只能用对象作为键，是为了保证只有通过键对象的引用才能取得值。如果允许原始值，那就没办法区分初始化时使用的字符串字面量和初始化之后使用的一个相等的字符串了。

6.5.4 使用弱映射

WeakMap 实例与现有 JavaScript 对象有着很大不同，可能一时不容易说清楚应该怎么使用它。这个问题没有唯一的答案，但已经出现了很多相关策略。

1. 私有变量

弱映射造就了在 JavaScript 中实现真正私有变量的一种新方式。前提很明确：私有变量会存储在弱映射中，以对象实例为键，以私有成员的字典为值。

下面是一个示例实现：

```
const wm = new WeakMap();

class User {
  constructor(id) {
    this.idProperty = Symbol('id');
    this.setId(id);
  }

  setPrivate(property, value) {
    const privateMembers = wm.get(this) || {};
    privateMembers[property] = value;
    wm.set(this, privateMembers);
  }

  getPrivate(property) {
    return wm.get(this)[property];
  }

  setId(id) {
    this.setPrivate(this.idProperty, id);
  }

  getId() {
    return this.getPrivate(this.idProperty);
  }
}

const user = new User(123);
alert(user.getId()); // 123
user.setId(456);
alert(user.getId()); // 456

// 并不是真正私有的
alert(wm.get(user)[user.idProperty]); // 456
```

慧眼独具的读者会发现，对于上面的实现，外部代码只需要拿到对象实例的引用和弱映射，就可以取得“私有”变量了。为了避免这种访问，可以用一个闭包把 WeakMap 包装起来，这样就可以把弱映射与外界完全隔离开了：

```
const User = (() => {
  const wm = new WeakMap();

  class User {
    constructor(id) {
      this.idProperty = Symbol('id');
```

```
      this.setId(id);
    }

    setPrivate(property, value) {
      const privateMembers = wm.get(this) || {};
      privateMembers[property] = value;
      wm.set(this, privateMembers);
    }

    getPrivate(property) {
      return wm.get(this)[property];
    }

    setId(id) {
      this.setPrivate(this.idProperty, id);
    }

    getId(id) {
      return this.getPrivate(this.idProperty);
    }
  }
  return User;
})();

const user = new User(123);
alert(user.getId()); // 123
user.setId(456);
alert(user.getId()); // 456
```

这样，拿不到弱映射中的健，也就无法取得弱映射中对应的值。虽然这防止了前面提到的访问，但整个代码也完全陷入了 ES6 之前的闭包私有变量模式。

2. DOM 节点元数据

因为 `WeakMap` 实例不会妨碍垃圾回收，所以非常适合保存关联元数据。来看下面这个例子，其中使用了常规的 `Map`：

```
const m = new Map();

const loginButton = document.querySelector('#login');

// 给这个节点关联一些元数据
m.set(loginButton, {disabled: true});
```

假设在上面的代码执行后，页面被 JavaScript 改变了，原来的登录按钮从 DOM 树中被删掉了。但由于映射中还保存着按钮的引用，所以对应的 DOM 节点仍然会逗留在内存中，除非明确将其从映射中删除或者等到映射本身被销毁。

如果这里使用的是弱映射，如以下代码所示，那么当节点从 DOM 树中被删除后，垃圾回收程序就可以立即释放其内存（假设没有其他地方引用这个对象）：

```
const wm = new WeakMap();

const loginButton = document.querySelector('#login');

// 给这个节点关联一些元数据
wm.set(loginButton, {disabled: true});
```

6.6 Set

ECMAScript 6 新增的 Set 是一种新集合类型，为这门语言带来集合数据结构。Set 在很多方面都像是加强的 Map，这是因为它们的大多数 API 和行为都是共有的。

6.6.1 基本 API

使用 new 关键字和 Set 构造函数可以创建一个空集合：

```
const m = new Set();
```

如果想在创建的同时初始化实例，则可以给 Set 构造函数传入一个可迭代对象，其中需要包含插入到新集合实例中的元素：

```
// 使用数组初始化集合
const s1 = new Set(["val1", "val2", "val3"]);

alert(s1.size); // 3

// 使用自定义迭代器初始化集合
const s2 = new Set({
  [Symbol.iterator]: function*() {
    yield "val1";
    yield "val2";
    yield "val3";
  }
});
alert(s2.size); // 3
```

初始化之后，可以使用 add()增加值，使用 has()查询，通过 size 取得元素数量，以及使用 delete()和 clear()删除元素：

```
const s = new Set();

alert(s.has("Matt"));    // false
alert(s.size);           // 0

s.add("Matt")
 .add("Frisbie");

alert(s.has("Matt"));    // true
alert(s.size);           // 2

s.delete("Matt");

alert(s.has("Matt"));    // false
alert(s.has("Frisbie")); // true
alert(s.size);           // 1

s.clear(); // 销毁集合实例中的所有值

alert(s.has("Matt"));    // false
alert(s.has("Frisbie")); // false
alert(s.size);           // 0
```

add()返回集合的实例，所以可以将多个添加操作连缀起来，包括初始化：

```
const s = new Set().add("val1");

s.add("val2")
 .add("val3");

alert(s.size); // 3
```

与 Map 类似，Set 可以包含任何 JavaScript 数据类型作为值。集合也使用 SameValueZero 操作（ECMAScript 内部定义，无法在语言中使用），基本上相当于使用严格对象相等的标准来检查值的匹配性。

```
const s = new Set();

const functionVal = function() {};
const symbolVal = Symbol();
const objectVal = new Object();

s.add(functionVal);
s.add(symbolVal);
s.add(objectVal);

alert(s.has(functionVal));   // true
alert(s.has(symbolVal));     // true
alert(s.has(objectVal));     // true

// SameValueZero 检查意味着独立的实例不会冲突
alert(s.has(function() {})); // false
```

与严格相等一样，用作值的对象和其他“集合”类型在自己的内容或属性被修改时也不会改变：

```
const s = new Set();

const objVal = {},
      arrVal = [];

s.add(objVal);
s.add(arrVal);

objVal.bar = "bar";
arrVal.push("bar");

alert(s.has(objVal)); // true
alert(s.has(arrVal)); // true
```

add()和 delete()操作是幂等的。delete()返回一个布尔值，表示集合中是否存在要删除的值：

```
const s = new Set();

s.add('foo');
alert(s.size); // 1
s.add('foo');
alert(s.size); // 1

// 集合里有这个值
alert(s.delete('foo')); // true

// 集合里没有这个值
alert(s.delete('foo')); // false
```

6.6.2 顺序与迭代

Set 会维护值插入时的顺序，因此支持按顺序迭代。

集合实例可以提供一个迭代器（Iterator），能以插入顺序生成集合内容。可以通过 values()方法及其别名方法 keys()（或者 Symbol.iterator 属性，它引用 values()）取得这个迭代器：

```
const s = new Set(["val1", "val2", "val3"]);

alert(s.values === s[Symbol.iterator]); // true
alert(s.keys === s[Symbol.iterator]);   // true

for (let value of s.values()) {
  alert(value);
}
// val1
// val2
// val3

for (let value of s[Symbol.iterator]()) {
  alert(value);
}
// val1
// val2
// val3
```

因为 values()是默认迭代器，所以可以直接对集合实例使用扩展操作，把集合转换为数组：

```
const s = new Set(["val1", "val2", "val3"]);

console.log([...s]); // ["val1", "val2", "val3"]
```

集合的 entries()方法返回一个迭代器，可以按照插入顺序产生包含两个元素的数组，这两个元素是集合中每个值的重复出现：

```
const s = new Set(["val1", "val2", "val3"]);

for (let pair of s.entries()) {
  console.log(pair);
}
// ["val1", "val1"]
// ["val2", "val2"]
// ["val3", "val3"]
```

如果不使用迭代器，而是使用回调方式，则可以调用集合的 forEach()方法并传入回调，依次迭代每个键/值对。传入的回调接收可选的第二个参数，这个参数用于重写回调内部 this 的值：

```
const s = new Set(["val1", "val2", "val3"]);

s.forEach((val, dupVal) => alert(`${val} -> ${dupVal}`));
// val1 -> val1
// val2 -> val2
// val3 -> val3
```

修改集合中值的属性不会影响其作为集合值的身份：

```
const s1 = new Set(["val1"]);

// 字符串原始值作为值不会被修改
for (let value of s1.values()) {
```