# 集合引用类型

## Set

ES6新增
类似数组，但是不能重复

### 基本API

构造函数接收数组完成初始化实例

```
// 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
```

属性：

- Set.prototype.constructor：构造函数，默认就是Set函数
- Set.prototype.size：返回Set实例的成员总数

方法：

- Set.prototype.add(value)：添加某个值，返回 Set 结构本身
- Set.prototype.delete(value)：删除某个值，返回一个布尔值，表示删除是否成功
- Set.prototype.has(value)：返回一个布尔值，表示该值是否为Set的成员
- Set.prototype.clear()：清除所有成员，没有返回值

```
// 去除数组的重复成员
[...new Set(array)]
//上面的方法也可以用于，去除字符串里面的重复字符。
[...new Set('ababbc')].join('')
```

### 迭代

- Set.prototype.keys()：返回键名的遍历器
- Set.prototype.values()：返回键值的遍历器

- Set.prototype.entries()：返回键值对的遍历器
- Set.prototype.forEach()：使用回调函数遍历每个成员

Set的key=value

```
let set = new Set(['red', 'green', 'blue']);

for (let item of set.keys()) {
  console.log(item);
}
// red
// green
// blue

for (let item of set.values()) {
  console.log(item);
}
// red
// green
// blue

for (let item of set.entries()) {
  console.log(item);
}
// ["red", "red"]
// ["green", "green"]
// ["blue", "blue"]
```

## 定义正式集合操作

通过代码实现 数学中的集合操作

- 某些 Set 操作是有关联性的，因此最好让实现的方法能支持处理任意多个集合实例。
- Set 保留插入顺序，所有方法返回的集合必须保证顺序。
- 尽可能高效地使用内存。扩展操作符的语法很简洁，但尽可能避免集合和数组间的相互转换能够节省对象初始化成本。

```
class XSet extends Set {
 union(...sets) {
 return XSet.union(this, ...sets)
 }
 intersection(...sets) {
 return XSet.intersection(this, ...sets);
 }
 difference(set) {
 return XSet.difference(this, set);
 }
 symmetricDifference(set) {
 return XSet.symmetricDifference(this, set);
 }
 cartesianProduct(set) {
 return XSet.cartesianProduct(this, set);
 }
 powerSet() {
 return XSet.powerSet(this);
```

```
  }
  // 返回两个或更多集合的并集
  static union(a, ...bSets) {
  const unionSet = new XSet(a);
  for (const b of bSets) {
  for (const bValue of b) {
  unionSet.add(bValue);
  }
  }
  return unionSet;
  }
  // 返回两个或更多集合的交集
  static intersection(a, ...bSets) {
  const intersectionSet = new XSet(a);
  for (const aValue of intersectionSet) {
  for (const b of bSets) {
  if (!b.has(aValue)) {
  intersectionSet.delete(aValue);
  }
  }
  }
  return intersectionSet;
  }
  // 返回两个集合的差集
  static difference(a, b) {
  const differenceSet = new XSet(a);
  for (const bValue of b) {
  if (a.has(bValue)) {
  differenceSet.delete(bValue);
  }
  }
  return differenceSet;
  }
  // 返回两个集合的对称差集
  static symmetricDifference(a, b) {
  // 按照定义，对称差集可以表达为
  return a.union(b).difference(a.intersection(b));
  }
  // 返回两个集合（数组对形式）的笛卡儿积
  // 必须返回数组集合，因为笛卡儿积可能包含相同值的对
  static cartesianProduct(a, b) {
  const cartesianProductSet = new XSet();
  for (const aValue of a) {
  for (const bValue of b) {
  cartesianProductSet.add([aValue, bValue]);
  }
  }
  return cartesianProductSet;
  }
  // 返回一个集合的幂集
  static powerSet(a) {
  const powerSet = new XSet().add(new XSet());
  for (const aValue of a) {
  for (const set of new XSet(powerSet)) {
  powerSet.add(new XSet(set).add(aValue));
  }
  }
  return powerSet;
  }
}
```

# WeakSet

类似WeakMap与Map之间的关系

因为Set的本质就是 Key=Value 的 Map，所以WeakSet的成员只能是对象

- 对象被回收，WeakMap自动清理
- 不可迭代

## 使用若集合

给对象打标签

```
const disabledElements = new WeakSet();
const loginButton = document.querySelector('#login');
// 通过加入对应集合，给这个节点打上“禁用”标签
disabledElements.add(loginButton);
```