2. 拦截的操作

- `proxy.property = value`
- `proxy[property] = value`
- `Object.create(proxy)[property] = value`
- `Reflect.set(proxy, property, value, receiver)`

3. 捕获器处理程序参数

- `target`：目标对象。
- `property`：引用的目标对象上的字符串键属性。
- `value`：要赋给属性的值。
- `receiver`：接收最初赋值的对象。

4. 捕获器不变式

如果 `target.property` 不可写且不可配置，则不能修改目标属性的值。

如果 `target.property` 不可配置且`[[Set]]`特性为 `undefined`，则不能修改目标属性的值。

在严格模式下，处理程序中返回 `false` 会抛出 `TypeError`。

9.2.3 has()

`has()`捕获器会在 `in` 操作符中被调用。对应的反射 API 方法为 `Reflect.has()`。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  has(target, property) {
    console.log('has()');
    return Reflect.has(...arguments)
  }
});

'foo' in proxy;
// has()
```

1. 返回值

`has()`必须返回布尔值，表示属性是否存在。返回非布尔值会被转型为布尔值。

2. 拦截的操作

- `property in proxy`
- `property in Object.create(proxy)`
- `with(proxy) {(property);}`
- `Reflect.has(proxy, property)`

3. 捕获器处理程序参数

- `target`：目标对象。
- `property`：引用的目标对象上的字符串键属性。

4. 捕获器不变式

如果 `target.property` 存在且不可配置，则处理程序必须返回 `true`。

如果 `target.property` 存在且目标对象不可扩展，则处理程序必须返回 `true`。

9.2.4 defineProperty()

defineProperty()捕获器会在 Object.defineProperty()中被调用。对应的反射 API 方法为 Reflect.defineProperty()。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  defineProperty(target, property, descriptor) {
    console.log('defineProperty()');
    return Reflect.defineProperty(...arguments)
  }
});

Object.defineProperty(proxy, 'foo', { value: 'bar' });
// defineProperty()
```

1. 返回值

defineProperty()必须返回布尔值，表示属性是否成功定义。返回非布尔值会被转型为布尔值。

2. 拦截的操作

- Object.defineProperty(proxy, property, descriptor)
- Reflect.defineProperty(proxy, property, descriptor)

3. 捕获器处理程序参数

- target：目标对象。
- property：引用的目标对象上的字符串键属性。
- descriptor：包含可选的 enumerable、configurable、writable、value、get 和 set 定义的对象。

4. 捕获器不变式

如果目标对象不可扩展，则无法定义属性。

如果目标对象有一个可配置的属性，则不能添加同名的不可配置属性。

如果目标对象有一个不可配置的属性，则不能添加同名的可配置属性。

9.2.5 getOwnPropertyDescriptor()

getOwnPropertyDescriptor()捕获器会在 Object.getOwnPropertyDescriptor()中被调用。对应的反射 API 方法为 Reflect.getOwnPropertyDescriptor()。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  getOwnPropertyDescriptor(target, property) {
    console.log('getOwnPropertyDescriptor()');
    return Reflect.getOwnPropertyDescriptor(...arguments)
  }
});

Object.getOwnPropertyDescriptor(proxy, 'foo');
// getOwnPropertyDescriptor()
```

1. 返回值

getOwnPropertyDescriptor()必须返回对象，或者在属性不存在时返回 undefined。

2. 拦截的操作

- Object.getOwnPropertyDescriptor(proxy, property)
- Reflect.getOwnPropertyDescriptor(proxy, property)

3. 捕获器处理程序参数

- target：目标对象。
- property：引用的目标对象上的字符串键属性。

4. 捕获器不变式

如果自有的 target.property 存在且不可配置，则处理程序必须返回一个表示该属性存在的对象。

如果自有的 target.property 存在且可配置，则处理程序必须返回表示该属性可配置的对象。

如果自有的 target.property 存在且 target 不可扩展，则处理程序必须返回一个表示该属性存在的对象。

如果 target.property 不存在且 target 不可扩展，则处理程序必须返回 undefined 表示该属性不存在。

如果 target.property 不存在，则处理程序不能返回表示该属性可配置的对象。

9.2.6 deleteProperty()

deleteProperty()捕获器会在 delete 操作符中被调用。对应的反射 API 方法为 Reflect.deleteProperty()。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  deleteProperty(target, property) {
    console.log('deleteProperty()');
    return Reflect.deleteProperty(...arguments)
  }
});

delete proxy.foo
// deleteProperty()
```

1. 返回值

deleteProperty()必须返回布尔值，表示删除属性是否成功。返回非布尔值会被转型为布尔值。

2. 拦截的操作

- delete proxy.property
- delete proxy[property]
- Reflect.deleteProperty(proxy, property)

3. 捕获器处理程序参数

- target：目标对象。
- property：引用的目标对象上的字符串键属性。

4. 捕获器不变式

如果自有的 target.property 存在且不可配置，则处理程序不能删除这个属性。

9.2.7 ownKeys()

ownKeys()捕获器会在 Object.keys()及类似方法中被调用。对应的反射 API 方法为 Reflect.ownKeys()。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  ownKeys(target) {
    console.log('ownKeys()');
    return Reflect.ownKeys(...arguments)
  }
});

Object.keys(proxy);
// ownKeys()
```

1. 返回值

ownKeys()必须返回包含字符串或符号的可枚举对象。

2. 拦截的操作

- Object.getOwnPropertyNames(proxy)
- Object.getOwnPropertySymbols(proxy)
- Object.keys(proxy)
- Reflect.ownKeys(proxy)

3. 捕获器处理程序参数

- target：目标对象。

4. 捕获器不变式

返回的可枚举对象必须包含 target 的所有不可配置的自有属性。

如果 target 不可扩展，则返回可枚举对象必须准确地包含自有属性键。

9.2.8 getPrototypeOf()

getPrototypeOf()捕获器会在 Object.getPrototypeOf()中被调用。对应的反射 API 方法为 Reflect.getPrototypeOf()。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  getPrototypeOf(target) {
    console.log('getPrototypeOf()');
    return Reflect.getPrototypeOf(...arguments)
  }
});

Object.getPrototypeOf(proxy);
// getPrototypeOf()
```

1. 返回值

getPrototypeOf()必须返回对象或 null。

2. 拦截的操作

- Object.getPrototypeOf(proxy)

- Reflect.getPrototypeOf(proxy)
- proxy.__proto__
- Object.prototype.isPrototypeOf(proxy)
- proxy instanceof Object

3. 捕获器处理程序参数

- target：目标对象。

4. 捕获器不变式

如果 target 不可扩展，则 Object.getPrototypeOf(proxy)唯一有效的返回值就是 Object.getPrototypeOf(target)的返回值。

9.2.9 setPrototypeOf()

setPrototypeOf()捕获器会在 Object.setPrototypeOf()中被调用。对应的反射 API 方法为 Reflect.setPrototypeOf()。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  setPrototypeOf(target, prototype) {
    console.log('setPrototypeOf()');
    return Reflect.setPrototypeOf(...arguments)
  }
});

Object.setPrototypeOf(proxy, Object);
// setPrototypeOf()
```

1. 返回值

setPrototypeOf()必须返回布尔值，表示原型赋值是否成功。返回非布尔值会被转型为布尔值。

2. 拦截的操作

- Object.setPrototypeOf(proxy)
- Reflect.setPrototypeOf(proxy)

3. 捕获器处理程序参数

- target：目标对象。
- prototype：target 的替代原型，如果是顶级原型则为 null。

4. 捕获器不变式

如果 target 不可扩展,则唯一有效的 prototype 参数就是 Object.getPrototypeOf(target)的返回值。

9.2.10 isExtensible()

isExtensible()捕获器会在 Object.isExtensible()中被调用。对应的反射 API 方法为 Reflect.isExtensible()。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  isExtensible(target) {
    console.log('isExtensible()');
```

```
    return Reflect.isExtensible(...arguments)
  }
});

Object.isExtensible(proxy);
// isExtensible()
```

1. 返回值

isExtensible()必须返回布尔值，表示 target 是否可扩展。返回非布尔值会被转型为布尔值。

2. 拦截的操作

- Object.isExtensible(proxy)
- Reflect.isExtensible(proxy)

3. 捕获器处理程序参数

- target：目标对象。

4. 捕获器不变式

如果 target 可扩展，则处理程序必须返回 true。

如果 target 不可扩展，则处理程序必须返回 false。

9.2.11 preventExtensions()

preventExtensions()捕获器会在 Object.preventExtensions()中被调用。对应的反射 API 方法为 Reflect.preventExtensions()。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  preventExtensions(target) {
    console.log('preventExtensions()');
    return Reflect.preventExtensions(...arguments)
  }
});

Object.preventExtensions(proxy);
// preventExtensions()
```

1. 返回值

preventExtensions()必须返回布尔值，表示 target 是否已经不可扩展。返回非布尔值会被转型为布尔值。

2. 拦截的操作

- Object.preventExtensions(proxy)
- Reflect.preventExtensions(proxy)

3. 捕获器处理程序参数

- target：目标对象。

4. 捕获器不变式

如果 Object.isExtensible(proxy)是 false，则处理程序必须返回 true。

9.2.12 apply()

apply()捕获器会在调用函数时中被调用。对应的反射 API 方法为 Reflect.apply()。

```
const myTarget = () => {};

const proxy = new Proxy(myTarget, {
  apply(target, thisArg, ...argumentsList) {
    console.log('apply()');
    return Reflect.apply(...arguments)
  }
});

proxy();
// apply()
```

1. 返回值

返回值无限制。

2. 拦截的操作

- `proxy(...argumentsList)`
- `Function.prototype.apply(thisArg, argumentsList)`
- `Function.prototype.call(thisArg, ...argumentsList)`
- `Reflect.apply(target, thisArgument, argumentsList)`

3. 捕获器处理程序参数

- `target`：目标对象。
- `thisArg`：调用函数时的 `this` 参数。
- `argumentsList`：调用函数时的参数列表

4. 捕获器不变式

`target` 必须是一个函数对象。

9.2.13 `construct()`

`construct()`捕获器会在 `new` 操作符中被调用。对应的反射 API 方法为 `Reflect.construct()`。

```
const myTarget = function() {};

const proxy = new Proxy(myTarget, {
  construct(target, argumentsList, newTarget) {
    console.log('construct()');
    return Reflect.construct(...arguments)
  }
});

new proxy;
// construct()
```

1. 返回值

`construct()`必须返回一个对象。

2. 拦截的操作

- `new proxy(...argumentsList)`
- `Reflect.construct(target, argumentsList, newTarget)`

3. 捕获器处理程序参数

- `target`：目标构造函数。

- ❑ argumentsList：传给目标构造函数的参数列表。
- ❑ newTarget：最初被调用的构造函数。

4. 捕获器不变式

target 必须可以用作构造函数。

9.3 代理模式

使用代理可以在代码中实现一些有用的编程模式。

9.3.1 跟踪属性访问

通过捕获 get、set 和 has 等操作，可以知道对象属性什么时候被访问、被查询。把实现相应捕获器的某个对象代理放到应用中，可以监控这个对象何时在何处被访问过：

```
const user = {
  name: 'Jake'
};

const proxy = new Proxy(user, {
  get(target, property, receiver) {
    console.log(`Getting ${property}`);

    return Reflect.get(...arguments);
  },
  set(target, property, value, receiver) {
    console.log(`Setting ${property}=${value}`);

    return Reflect.set(...arguments);
  }
});

proxy.name;     // Getting name
proxy.age = 27; // Setting age=27
```

9.3.2 隐藏属性

代理的内部实现对外部代码是不可见的，因此要隐藏目标对象上的属性也轻而易举。比如：

```
const hiddenProperties = ['foo', 'bar'];
const targetObject = {
  foo: 1,
  bar: 2,
  baz: 3
};
const proxy = new Proxy(targetObject, {
  get(target, property) {
    if (hiddenProperties.includes(property)) {
      return undefined;
    } else {
      return Reflect.get(...arguments);
    }
  },
  has(target, property) {
```

```
    if (hiddenProperties.includes(property)) {
      return false;
    } else {
      return Reflect.has(...arguments);
    }
  }
});

// get()
console.log(proxy.foo);  // undefined
console.log(proxy.bar);  // undefined
console.log(proxy.baz);  // 3

// has()
console.log('foo' in proxy);  // false
console.log('bar' in proxy);  // false
console.log('baz' in proxy);  // true
```

9.3.3　属性验证

因为所有赋值操作都会触发 `set()`捕获器，所以可以根据所赋的值决定是允许还是拒绝赋值：

```
const target = {
  onlyNumbersGoHere: 0
};

const proxy = new Proxy(target, {
  set(target, property, value) {
    if (typeof value !== 'number') {
      return false;
    } else {
      return Reflect.set(...arguments);
    }
  }
});

proxy.onlyNumbersGoHere = 1;
console.log(proxy.onlyNumbersGoHere);  // 1
proxy.onlyNumbersGoHere = '2';
console.log(proxy.onlyNumbersGoHere);  // 1
```

9.3.4　函数与构造函数参数验证

跟保护和验证对象属性类似，也可对函数和构造函数参数进行审查。比如，可以让函数只接收某种类型的值：

```
function median(...nums) {
  return nums.sort()[Math.floor(nums.length / 2)];
}

const proxy = new Proxy(median, {
  apply(target, thisArg, argumentsList) {
    for (const arg of argumentsList) {
      if (typeof arg !== 'number') {
        throw 'Non-number argument provided';
      }
    }
```

```
    return Reflect.apply(...arguments);
  }
});

console.log(proxy(4, 7, 1));  // 4
console.log(proxy(4, '7', 1));
// Error: Non-number argument provided
```

类似地，可以要求实例化时必须给构造函数传参：

```
class User {
  constructor(id) {
    this.id_ = id;
  }
}

const proxy = new Proxy(User, {
  construct(target, argumentsList, newTarget) {
    if (argumentsList[0] === undefined) {
      throw 'User cannot be instantiated without id';
    } else {
      return Reflect.construct(...arguments);
    }
  }
});

new proxy(1);

new proxy();
// Error: User cannot be instantiated without id
```

9.3.5 数据绑定与可观察对象

通过代理可以把运行时中原本不相关的部分联系到一起。这样就可以实现各种模式，从而让不同的代码互操作。

比如，可以将被代理的类绑定到一个全局实例集合，让所有创建的实例都被添加到这个集合中：

```
const userList = [];

class User {
  constructor(name) {
    this.name_ = name;
  }
}

const proxy = new Proxy(User, {
  construct() {
    const newUser = Reflect.construct(...arguments);
    userList.push(newUser);
    return newUser;
  }
});

new proxy('John');
new proxy('Jacob');
new proxy('Jingleheimerschmidt');

console.log(userList); // [User {}, User {}, User{}]
```