```
Function.prototype.callCustom = function (thisArg) {
    /**
     * 默认绑定（普通函数内的 this 默认绑定 window，注意在严格模式下 this 无任何指向，即 undefined）
     * 隐式绑定（函数作为方法，this 会绑定该方法所属的对象）
     * 显示绑定（call，apply，bind，Reflect.apply，以及带有修改 this 指向的其他函数或者方法）
     * new 绑定（new 可以使构造函数中的 this 指向 new 关键字创建的对象）
     * 箭头函数的绑定（箭头函数的 this 指向外部的执行上下文）
     */

    /**
     * 在上面的五种方式中，除却箭头函数暂且不谈，我们只能使用隐士绑定的方式改变函数内部的指向，从而实现 call 的核心功能
     */

    let context = thisArg;

    context === undefined || context === null && (context = window);

    typeof context !== 'object' && typeof context !== 'function' && (context = new Object(context));

    /**
     * 这里定义一个属性标识符 _fn_
     * 只需要将函数内部的 this 赋值给 context[_fn_]，那么 this 自然就指向了 context 对象，也就完成了更换 this 指向的核心功能
     * 但是需要考虑一个问题，原本的 context 对象是否原本就存在 _fn_ 属性，一旦有就会被覆盖，所有这里要做 Object.getOwnPropertyNames(context).indexOf(_fn_) 判断处理
     * 一旦原来就存在这个属性，那么我们把这个属性保存在 fn.value 中，并把 fn.exist 标识设置为 true
     * 等到后续操作完毕以后，再把 fn.value 重新赋值给 context 对象
     * 其实属性冲突的问题，我们可以使用 Symbol 来完成，但是为了考虑兼容性，便没有使用 Symbol
     * 后面会有不考虑兼容性的其他实现版本
     */
    let _fn_ = '_fn_';

    let fn = {
        value: undefined,
        exist: false
    };

    if (Object.getOwnPropertyNames(context).indexOf(_fn_) !== -1) {
        fn.value = context[_fn_];
        fn.exist = true;
    }

    context[_fn_] = this;

    let args = [];

    /**
     * 这里将除了 thisArg 所有的参数推到 args 中
     * 为什么使用 args.push("arguments[" + i + "]")
     * 而不是 args.push(arguments[i])
     * 比如 call({}, "真", "不", "错")
     * 通过 eval 转换后，字符串中的引号会被打掉：context[_fn_](真, 不, 错)
     * 很容易看出问题，原本的字符串参数丢失了引号，失去了类型标记，将会导致报错
```

```
     * 所以使用 args.push("arguments[" + i + "]") 来避免这个问题，context[_fn_]
(arguments[1], arguments[2], arguments[3])
     * 掌握 eval 函数的使用方式就可以理解为什么这么处理了
     */
    for (let i = 1; i < arguments.length; i++) {
        args.push("arguments[" + i + "]");
    }

    eval("context[_fn_](" + args.toString() + ")");

    if (fn.exist) {
        context[_fn_] = fn.value;
    } else {
        delete context[_fn_];
    }
}

Function.prototype.callCustom = function (thisArg, ...args) {
    /**
     * 本实现方式使用 es6+ 规范
     */

    let context = thisArg;

    context === undefined || context === null && (context = window);

    typeof context !== 'object' && typeof context !== 'function' && (context =
new Object(context));

    let _fn_ = Symbol('_fn_');

    context[_fn_] = this;

    context[_fn_](...args);

    delete context[_fn_];
}

/**
 * 本部分内容十分的重要，请认真阅读
 *
 * 以上是 apply,call,bind 的实现原理
 * 但是在 'use strict'; 模式中，这三个方法拥有不同的表现
 * 在 sloppy mode （非严格模式）中，当 thisArg 不是一个对象时，内部逻辑会先将其转换为对
象，如果是 undefined 或者 null，那么 this 指向将会指向 window
 * 在 strict mode （严格模式）中，thisArg 传进去什么，那么 this 对象就是什么，不会发生任
何转换或者重新赋值
 *
 * 因为在语言层面上的一些原因，我们无法使用 javascript 语言来模拟严格模式下这三个方法的表
现，但是要谨记三个方法在不同模式下的行为
 *
 * 一般面试时，手写代码就是实现 sloppy mode 模式下的行为
 */

function foo () {
    console.log(this);
}

foo.apply(99);
// 在 sloppy mode 中，打印 Number { 99 }，将 99 转换成了 Number 包装对象
// 在 strict mode 中，打印 99，无任何转换
```

```
foo.apply(null);
// 在 sloppy mode 中，打印 window 对象，将 this 重新指向了 window
// 在 strict mode 中，打印 null，无任何改变
```