Zero

a JavaScript object oriented framework.

features:

1. keyword method

   zero提供一系列的以$为前缀的类关键字方法,如$class,$module,$interface…

2. global

   global是Zero的变量管理机制,zero的目标并不包含实现commonjs的规范,$global及其相关关键字方法只是提供一个包管理之外的轻量的对象引用机制,它也不妨碍使用的人使用包管理机制. zero包含三个全局变量,z,$global,$run

```
//在闭包内运行一个方法,避免全局变量的问题
$run(function(){
    eval($global.all); //声明引用$global所管理的对象的变量,以方便在后面的代码中使用

    //some code …

    $global("obj1", obj1); //声明一个变量交给$global管理
})

/*
执行一段代码,run方法会传递包含该段代码方法的签名中引用的$global管理的变量,
但注意在此段代码中不能引用$global.run之外的非全局变量
*/
$global.run(function(obj1, obj2, obj3){
    var prop = obj1.prop;
    obj2.xx();
    //...
})
```

3. module

```
//此关键字并不对传入的模块声明作任何处理,因此符合模块接口的普通对象也可认作是模块
function $module(o){
    return o;
}

//使用$module声明一个模块,
var MAmodule = $module({
    onIncluded: function(){
        this.prop1 = {}; //this为包含此模块的对象
```

```
    },
    prop: {},
    method: function(){}
});

$(obj).include(MAmodule);
```

4. class

```
//此构造函数内调用的方法都是继承自z.Base
var Class = function(){
    this.callBase(); //调用父类的constructor

    this.property("prop"); //声明一个属性prop

    this.property({
        "prop1": {},
        "prop2": {
            get: function(){},
            set: function(){}
        }
    });

    this.include(module); //包含一个模块
}

Class.prototye = {
    sayHello: function(){
        this.callBase(); //调用父类的sayHello
    }
}

/*
 继承z.Base,添加已实现接口名称到类的已实现接口数组
 z.Base是Zero的基础类,建议所有类都继承它
 $class(Class)新建一个包含Class引用并拥有extend,implement,include方法的对象,
 但不会扩展Class本身
 */
$class(Class).extend(z.Base).implement(AInterface);
```

5. wrapper

wrapper是一种对象的扩展机制,可使用$(obj)建立一个引用了obj并包含扩展模块的新对象,它不直接扩展对象本身,$$(obj)则是直接使用模块扩展对象,$和$$是根据所管理的扩展模块的登记信息和对象本身的类型和接口,来决定如何扩展对象的.

```
var StringWrapper = $module({
```

```
    capitalize: function(){
        var o = this.target;
        return o.charAt(0).toUpperCase() + o.slice(1);
    }
});

var NumberWrapper = $module({
    next: function(){ return this.target.valueOf() + 1; }
});

$.regist(StringWrapper, String, "stringWrapper");

$.regist(NumberWrapper, Number, "numberWrapper");

$.setDefault(String, "stringWrapper");

$("string").capitalize();

$(8).wrap("numberWrapper").next();

var str = $$("string");
str.capitalize();
```

6. is

   zero尝试使用$is()统一的类型判断的方式

```
//判断对象typeof
$is('string', 'abc');

//判断对象是否为另一对象的实例
$is(Function, function() {});

//与特殊对象或值比较
$is(null, obj);
$is(undefined, obj);

var spec = $spec({
    "instanceOf": B,
    "prototypeOf": Object.prototype,
    "typeof": "type"
});

$is(spec, obj); //复合判断
```

7. interface

   由于动态语言的对象的成员是可变化的,并不能从其类或者构造函数或者原型链来确定它的成员是否存在和是否可用,因此通过显式的接口和对象直接进行比较,更能准确的确

定对象是属于某种'类型'.

另外动态语言对象成员变更的途径多样,不像强类型语言那样容易从代码看出对象的接口,因此显式接口还有一定的文档功能.

此外接口还可用于方法的参数验证,方法的重载等其他地方.

接口并不需要一定使用$interface来声明,符合IInteface接口的普通对象也可

```
var IBase = $interface({
    member: {
        prop: String, //声明此成员的类型
        method: "function(p1, p2)", //声明此成员的类型为function,其后签名只起文档的作用
        prop3: {
            type: { //复合的形式描述对象的类型
                instanceOf: "prop3",
                prototypeOf: [Object]
            },
            value: {}, //默认值
            check: function(v){}, //对象值的验证器
            required: true,  //此成员是否必须存在
            ownProperty: true //此成员是否为对象自己拥有还是原型链上的
        },
        prop4: {type: [String, Number]} 对象的类型可以为多种
    },
    type: Object, //对象的类型
    freeze: true, //是否冻结对象,即不让对象拥有member以外的成员
    base: IObject //父接口
})

//如果只定义member和type,可以用更简单的写法
var IBase = $interface({
    prop: String,
    method: "function(p1,p2)"
}, Object)

$support(Interface, aObj); //判断一个对象是否支持某接口
```

8. function option

zero约定方法的option属性表示方法参数的interface,有了这个接口,可以方便的实现参数验证,同时支持普通参数传递形式和key/value对象传递参数,还有方法重载,参数组默认值和当前设定的合并.

zero建议给超过2个参数的方法设定参数接口

```
fn = function(p1, p2) {
    var option = $option(); //将参数转换成key/value形式,并和参数接口的默认值合并
```

```
    $support(fn.option, option); //判断参数是否符合接口

    //option.key1
    //option.key2
    //option.key3
    //...
}

fn.option = { key1: {value: {}}, key2: {value: {}}, key3: {}}

//也可一次性定义方法与其option
var fn = $fn(function(){//...}, {key1: {}, key2: {}});

fn("p1", "p2");

fn({
    key3: {},
    key2: {}
});
```

9. overwrite

zero可以通过$overwrite()来生成能够分派重载方法的主方法.

也可以自定义主方法,使用$dispatch()来手动获取根据当前参数和重载方法的接口选择应该执行的方法,然后手动调用该方法.

```
    var fnWithStringAndNumber = function(){}
    fnWithStringAndNumber.option = {
        name: 'string',
        age: 'number'
    }

    function fnWithTwoString(){
        //do something...
    }

    fnWithTwoString.option = {
        name: 'string',
        interest: 'string'
    }

    var fn = $overwrite(fnWithStringAndNumber, fnWithTwoString);

    fn("jim", 8); //call fnWithStringAndNumber

    //使用$dispatch选择要调用的方法
    var fn2 = function(){
        var args = ["jim", 8];
        var fn = $dispatch([fnWithStringAndNumber, fnWithTwoString],
```

```
args);
          fn.apply(this, args);
     }
```

10. inspect

```
Object.prototype.m = {};

var obj = {p: {}, __p: {}, fn: function(){}, __fn: function(){}};

var o = $inspect(obj);

o.methods();
//--> ["fn", "__fn"]

o.publicMethods();
//--> ["fn"]

o.privateMethods();
//--> ["__fn"]

o.fields();
//--> ["p", "__p", "m"]

o.publicFields();
//--> ["p", "m"]

o.privateFields();
//--> ["__p"]

o.keys();
//--> ["p", "__p", "fn", "__fn"]

o.allKeys();
//--> ["p", "__p", "fn", "__fn", "m"]

o.proto();
//--> Object {}

o.protoLink();
//--> [Object {}]

o.creator();
//--> Object()
```

11. utilies and more
    - 遍历 $every, $everyKey, $trace

- 对象操作 $copy, $merge, $slice, $property
- 对象工厂 $enum, $array, $fn
- 对象反射 $traceProto, $callBase
- 实用方法 $isPrivate, $keys, $thisFn, $containsAll