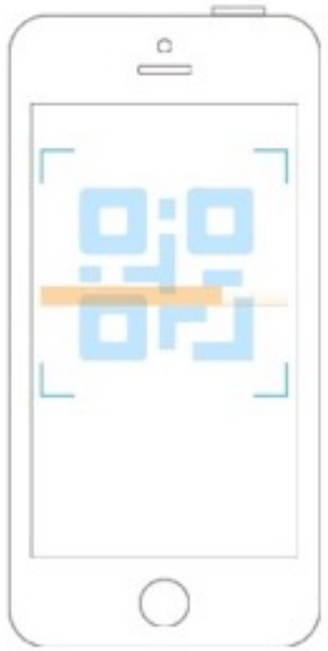
签到

- 拿出您的手机扫一扫右侧二维码进行签到



通用的准则

- 复用性
- 深拷贝

1. 複用性

- 不进行任何不必要的计算
- 缓存任何可以缓存的结果
- 只在使用到时才实例化类对象
- 尽量对Class的实例进行复用

```
Object.defineProperty(app.context, property, {
      get() {
          if (!this[CLASSLOADER]) {
            this[CLASSLOADER] = new Map();
          }
          const classLoader = this[CLASSLOADER];
          let instance = classLoader.get(property);
          if (!instance) {
            instance = getInstance(target, this);
            classLoader.set(property, instance);
          }
          return instance;
    },
});
```

2. 深拷贝

使用Loadsh等第三方模块进行深拷贝

构造原型链?

两种构造原型链的方式

- Object.create
- Object.setPrototypeOf

BenchMark性能比较

```
'use strict'
const lodash = require('lodash');
const BigObject = {};
function deep(obj, i, inner) {
    inner = inner || 0;
    if (inner > 100) {
        return obj
    } else {
        obj[`name-${i}-${inner}`] = {};
        obj[`name-${i}-${inner}`][`inner-${inner}`] = inner++;
        return deep(obj, i, inner);
    }
}
for (let i = 0; i < 100; i++) {
    deep(BigObject, i);
}
module.exports = {
    deepClone(){
        let deep = lodash.merge({}, BigObject);
        let data = deep['name-50-50']['inner-50'];},
    objectCreate(){
        let prot = Object.create(BigObject);
        let data = prot['name-50-50']['inner-50'];},
    objectSet(){
        let prot = {};
        Object.setPrototypeOf(prot, BigObject);
        let data = prot['name-50-50']['inner-50'];}
};
```

```
'use strict'
const Benchmark = require('benchmark');
const suite = new Benchmark.Suite();
const testClone = require('./clone');
suite.add('clone#深拷贝', function () {
        testClone.deepClone();
    })
    .add('clone#原型链构造-Object.create', function () {
        testClone.objectCreate();
    })
    .add('clone#原型链构造-Object.setPrototypeOf', function () {
        testClone.objectSet();
})
.on('cycle', function (event) {
    console.log(String(event.target));
})
.on('complete', function () {
    console.log('Fastest is ' + this.filter('fastest').map('name'));
})
.run({'async': true});
```

BenchMark测试结果

clone#深拷贝 x 17.61 ops/sec ±9.81% (31 runs sampled)

clone#原型链构造—Object.create x 4,381,477 ops/sec ±9.36% (77 runs sampled)

clone#原型链构造—Object.setPrototypeOf x 3,519,379 ops/sec ±1.22% (83 runs sampled)

Fastest is clone#原型链构造—Object.create

在深拷贝和原型链构造都适用的情况下两者性能差距达到了二十万倍！

JS特有的优化点

1. 减少原型链遍历

EventEmitter类的实践—V4版本

- 用来存储事件名——事件回调函数的内部变量方式：`this._events = {};`

EventEmitter类的实践——V6版本

- 用来存储事件名——事件回调函数的内部变量方式：`this._events = new EventHandlers();`

- 空原型：`function EventHandlers() {}`
  `EventHandlers.prototype = Object.create(null);`

在Deep-Into-Node中，经过作者在jsperf 上的比较，两者大概有2倍左右的性能差距

2. Hidden Class

JS对象存储

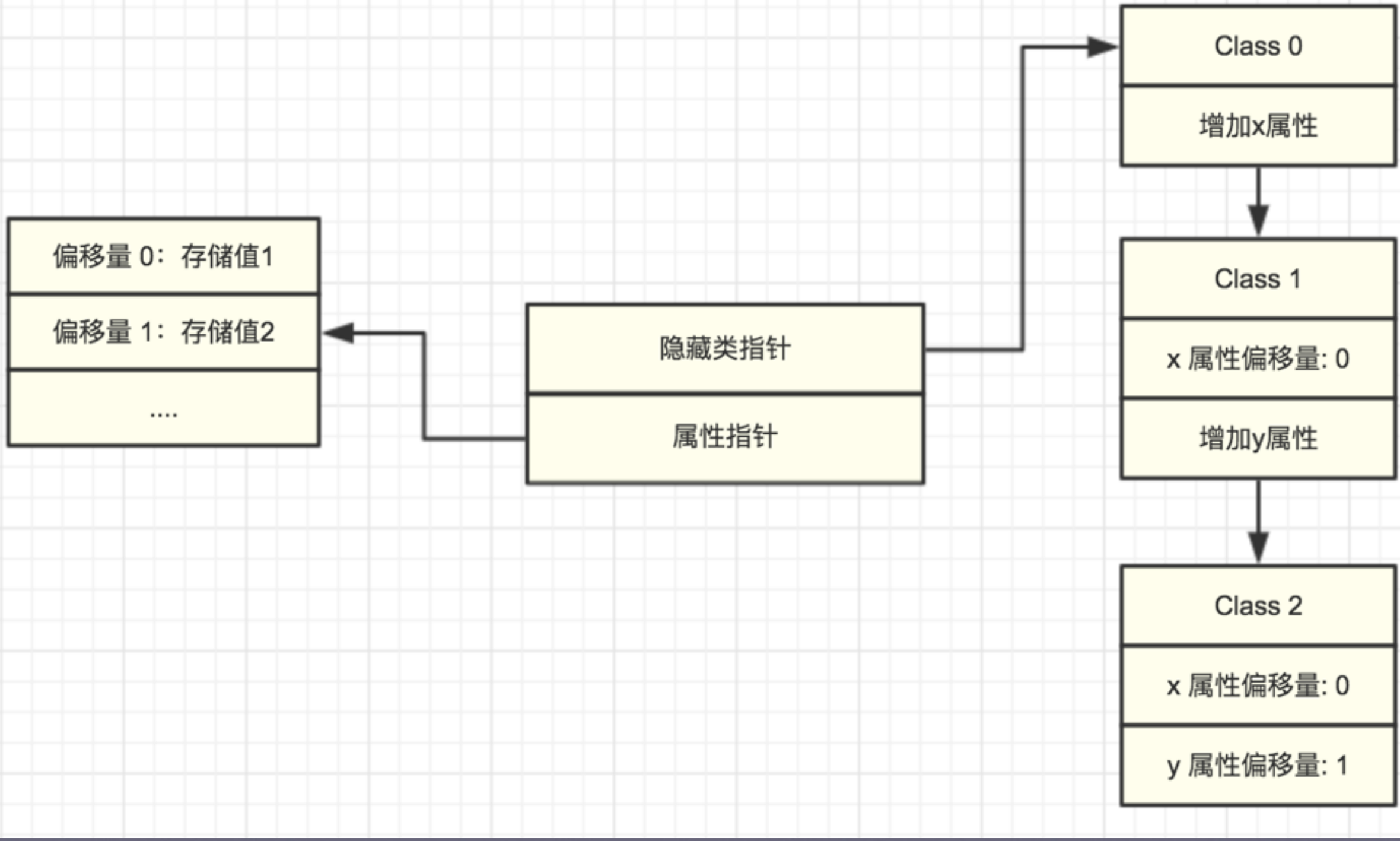
- 我们在JS的对象中存储Key时，实质上Key首先被Hash，所以相对的，从对象中获取某一个Key对应的Value时，相当于从Hash字典中进行查找。

- 在静态语言中，一般来类信息可以在各个实例中共享——属性值地址相对实例首地址的偏移量。

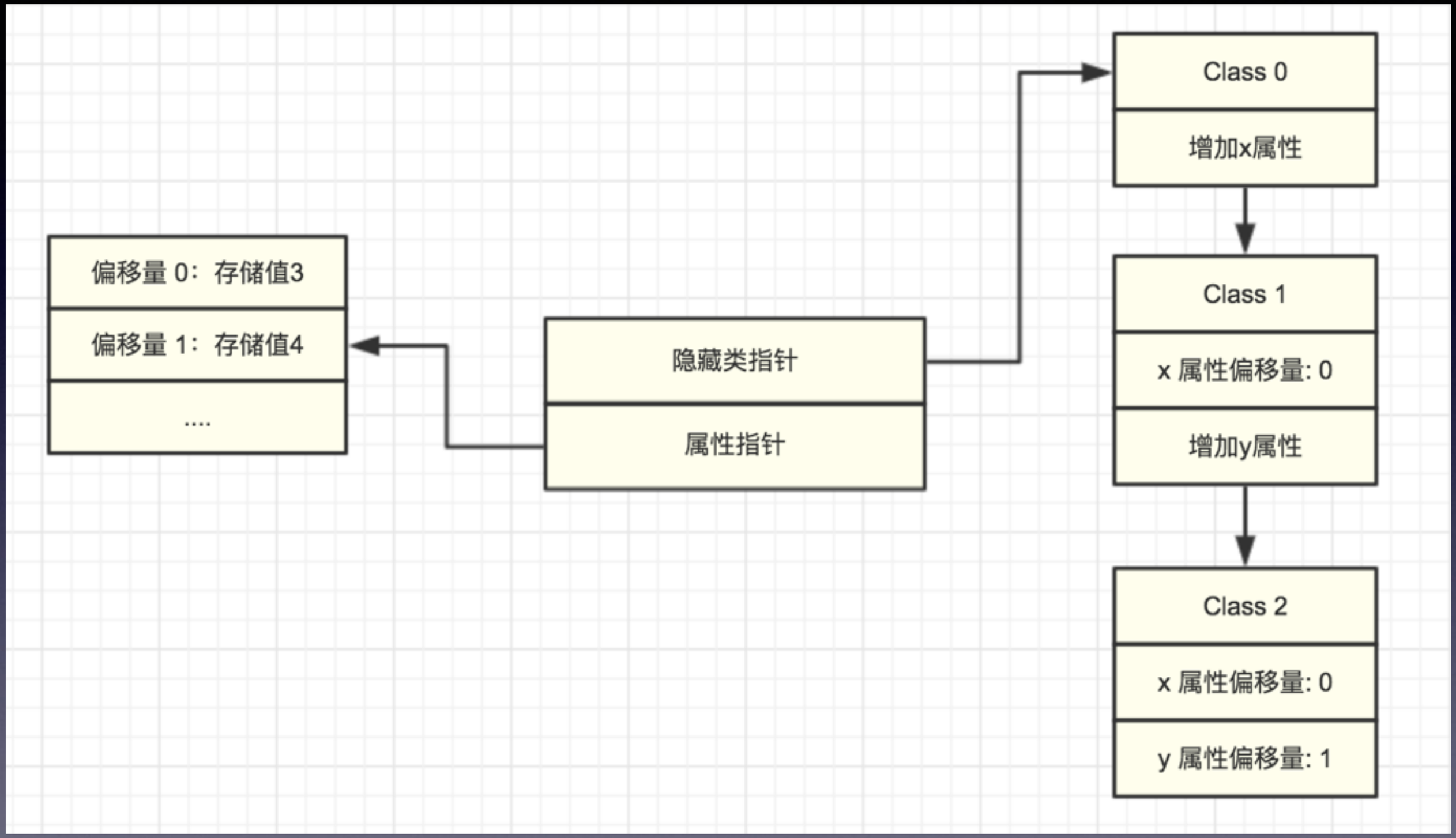
- V8对此作了一些优化，即Hidden Class，从对象中获取属性时，根据隐藏类的某一个属性地址偏移，直接读取对应属性值，而不用经过Hash字典计算。

一个隐藏类生成的例子

```
function Point(x, y) {
        this.x = x;
        this.y = y;
    }
var p1 = new Point(1, 2);
var p2 = new Point(3, 4);
```


偏移量 0：存储值1
偏移量 1：存储值2
....
隐藏类指针
属性指针
Class 0
增加x属性
Class 1
x 属性偏移量: 0
增加y属性
Class 2
x 属性偏移量: 0
y 属性偏移量: 1

当p1实例生成后，实际上隐藏类Clss 0, Class1, Class 2均已生成完毕，此时实例化p2，不需要重新生成隐藏类。


Class 0
增加x属性
Class 1
x 属性偏移量: 0
增加y属性
Class 2
x 属性偏移量: 0
y 属性偏移量: 1
隐藏类指针
属性指针
偏移量 0：存储值3
偏移量 1：存储值4
....

- 所有类的使用到的属性在构造函数内初始化，尽量不要做动态的实例属性增加。
- delete方法只在万不得已的情况下使用：动态的delete掉属性元素，会造成隐藏类失效，从而导致所有此对象的属性获取都退化成Hash字典查找。

3. 编写V8可优化代码

- V8引擎有两个编译器，其中有一个是专门用来对运行非常频繁的JS代码进行热优化。
- 如果我们编写的JS代码中尽量避免引擎无法优化的格式，这样子就能发挥引擎最大的能力。

反馈结果

```
Functions V8 Engine Optimization Failed List(Sort By HitTimes DESC):
1. compile (TryCatchStatement) (/Users/node/achilles/node_modules/.2.5.5@ejs/lib/ejs.js 431)
2. anonymous (TryCatchStatement)
3. include (Optimized too many times) (/Users/node/achilles/node_modules/.2.5.5@ejs/lib/
ejs.js 505)
4. onFulfilled (TryCatchStatement) (/Users/node/achilles/node_modules/.4.6.0@co/index.js 62)
5. next (Optimization is disabled) (native generator.js 13)
6. rethrow.forEach (Optimized too many times)
7. handle (TryCatchStatement) (/Users/node/achilles/node_modules/.4.14.1@express/lib/router/
layer.js 86)
8. Promise (TryCatchStatement) (native promise.js 42)
9. PromiseHandle (TryCatchStatement) (native promise.js 87)
10. _genRenderPage (Yield) (/Users/node/achilles/node_modules/.1.0.11@@tuniu/node-pc-sdk/lib/
pcRender.js 181)
11. __invoke (Yield) (/Users/node/achilles/node_modules/.0.1.32@@tuniu/remote-client/lib/
RemoteClient.js 85)
12. _genRender (Yield) (/Users/node/achilles/node_modules/.1.0.6@@tuniu/node-render/lib/
render.js 199)
13. tryOnImmediate (TryFinallyStatement) (timers.js 606)
14. Join (TryFinallyStatement) (native array.js 172)
15. _gen (Yield) (/Users/node/achilles/node_modules/.1.0.3@@tuniu/node-plugin/lib/create.js
52)
```

根据反馈来优化代码

TryCatchStatement

- 产生的原因：控制流不稳定，很难在运行时优化
- 解决的方式：首先肯定是尽量不要再频繁调用的函数中使用try/catch，如果必须使用，可以把该函数重构为：`try{ func }catch(e){ }`的形式。

TryFinallyStatement

- 产生的原因：和TryCatchStatement一样
- 解决的方式：和TryCatchStatement一样

Yield

- 产生的原因：generator 的引擎实现中，其状态保持、恢复通过拷贝函数栈帧实现，但在优化编译器中并不适用。
- 解决的方式：暂时无解，crankshaft无法优化，新一代turbofan优化编译器中可以被优化。等Node v7成为LTS版本后，可以彻底放弃Yield魔法，拥抱Async函数。

Assignment to parameter in arguments object

- 例子：`function test(a) { a = 0 }`
- 产生的原因：给函数中的参数重新赋值
- 解决的方式：采用strict模式，且可以在函数内部新建一个引用指向要使用到的参数，所有对该参数直接操作更改为对此引用的操作。

Rest parameters

- 例子：`function test(…rest) { return rest[0] }`

- 产生的原因：…的形式会操作arguments对象

- 解决的方式：在新一代的turbofan优化编译器能对rest参数进行优化前，请尽量避免使用这种参数写法（实在想写的话也请用Babel编译）

Too many parameters

- 例子：`function test(p1,p2,…p512) {}`
- 产生的原因：参数大于512个
- 解决的方式：应该不会有人传>512个参数吧。参数太长可以考虑用对象的形式传参。

ForInStatement is not fast case

- 例子：`for(){ /**lots of code**/ }`
- 产生的原因：for循环中的代码太长
- 解决的方式：将for循环中的代码抽成一个独立的函数。

Thank You

END