代码优化基本原则

1. 易读性优先
2. 如果不是性能瓶颈，就不要为了性能而改写代码
3. 复杂性守恒原则：无论你怎么写代码，复杂性都是不会消失的
   推论：如果逻辑很复杂，那么代码看起来就应该是复杂的。如果逻辑很简单，代码看起来就应该是简单的。

命名

程序员三大难题

1. 变量命名
2. 缓存失效
3. 循环边界

可见变量命名的重要性。

网上有很多命名规范，大家可以参考。本节课只讲基本原则。

1. 注意词性
   - 普通变量/属性用「名词」

     ```
     var person = {
         name: 'Frank'
     }
     var student = {
         grade: 3,
         class: 2
     }
     ```

   - bool变量/属性用「形容词」或者「be动词」或者「情态动词」或者「hasX」

     ```
     var person = {
         dead: false, // 如果是形容词，前面就没必要加 is，比如isDead 就很废话
         canSpeak: true, //情态动词有 can、should、will、need 等，情态动词后面接动词
         isVip: true, // be 动词有 is、was 等，后面一般接名词
         hasChildren: true, // has 加名词
     }
     ```

   - 普通函数/方法用「动词」开头

     ```
     var person = {
         run(){}, // 不及物动词
         drinkWater(){}, // 及物动词
         eat(foo){}, // 及物动词加参数（参数是名词）
     }
     ```

   - 回调、钩子函数用「介词」开头，或用「动词的现在完成时态」

     ```
     var person = {
         beforeDie(){},
         afterDie(){},
         // 或者
         willDie(){}
         dead(){} // 这里跟 bool 冲突，你只要不同时暴露 bool dead 和函数 dead 就行，怕冲突就用上面的 afterDie
     }
     button.addEventListener('click', onButtonClick)
     var component = {
         beforeCreate(){},
         created(){},
         beforeMount(){},
         mounted(){},
         beforeUpdate(){},
         updated(){},
         activated(){},
         deactivated(){},
         beforeDestroy(){},
         destroyed(){},
         errorCaptured(){}
     }
     ```

   - 容易混淆的地方加前缀

     ```
     div1.classList.add('active') // DOM 对象
     div2.addClass('active') // jQuery 对象
     不如改成
     domDiv1 或 elDiv1.classList.add('active')
     $div2.addClass('active')
     ```

   - 属性访问器函数可以用名词

     ```
     $div.text() // 其实是 $div.getText()
     $div.text('hi') // 其实是 $div.setText('hi')
     ```

2. 注意一致性

   - 介词一致性
     如果你使用了 before + after，那么就在代码的所有地方都坚持使用
     如果你使用了 before + 完成时，那么就坚持使用
     如果你改来改去，就「不一致」了，不一致将导致「不可预测」
   - 顺序一致性
     比如 updateContainerWidth 和 updateHeightOfContainer 的顺序就令人很别扭，同样会引发「不可预测」
   - 表里一致性
     函数名必须完美体现函数的功能，既不能多也不能少。
     比如

     ```
     function getSongs(){
         return $.get('/songs).then((response){
             div.innerText = response.songs
         })
     }
     ```

     就违背了表里一致性，getSongs 表示获取歌曲，并没有暗示这个函数会更新页面，但是实际上函数更新了 div，这就是表里不一，正确的写法是

     - 要么纠正函数名

       ```
       function getSongsAndUpdateDiv(){
           return $.get('/songs).then((response){
               div.innerText = response.songs
           })
       }
       ```

     - 要么写成两个函数

       ```
       function getSongs(){
           return $.get('/songs)
       }
       function updateDiv(songs){
           div.innerText = response.songs
       }
       getSongs().then((response)=>{
           updateDiv(response.songs)
       })
       ```

   - 时间一致性
     有可能随着代码的变迁，一个变量的含义已经不同于它一开始的含义了，这个时候你需要及时改掉这个变量的名字。
     这一条是最难做到的，因为写代码容易，改代码难。如果这个代码组织得不好，很可能会出现牵一发而动全身的情况（如全局变量就很难改）

改代码

如果你的代码有单元测试，那么改起来就很放心。如果没有单元测试，就需要用「小步快跑」的策略来修改。

小步快跑的意思是说，每次只修改一点点，测试通过后，再修改一点点，再测试，再修改一点点……如此反复。

那么如何修改一点点呢？《重构》这本书介绍了很多方法，但是讲得挺啰嗦的，如果你有时间可以看看。

我这里只说两个经久不衰的方法。

一、使用函数来改代码

步骤：

1. 将一坨代码放到一个函数里
2. 将代码依赖的外部变量作为参数
3. 将代码的输出作为函数的返回值
4. 给函数取一个合适的名字
5. 调用这个函数并传入参数
6. 这个函数里的代码如果超过 5 行，则依然有优化的空间，请回到第 1 步

二、使用对象来改代码

如果使用了函数改造法改造后，发现有太多的小函数，则可以使用对象讲这个函数串起来。

记得我们讲过「this 是函数和对象的桥梁」吗，我们会用 this 来串联这个对象和所有函数。

最终代码：http://js.jirengu.com/mimazaboke/1/edit?html,js,output

一些固定的套路

1. 表驱动编程（《代码大全》里说的）
   所有一一对应的关系都可以用表来做
2. 自说明代码（以 API 参数为例）
   把别人关心的东西放在显眼的位置

bad smell（坏味道）

有些代码可以用，但是很「臭」。

哪些代码是有坏味道的

1. 表里不一的代码
2. 过时的注释
3. 逻辑很简单，但是看起来很复杂的代码
4. 重复的代码
5. 相似的代码
6. 总是一起出现的代码

破窗效应

> 此理论认为环境中的不良现象如果被放任存在，会诱使人们仿效，甚至变本加厉。一幢有少许破窗的建筑为例，如果那些窗不被修理好，可能将会有破坏者破坏更多的窗户。最终他们甚至会闯入建筑内，如果发现无人居住，也许就在那里定居或者纵火。一面墙，如果出现一些涂鸦没有被清洗掉，很快的，墙上就布满了乱七八糟、不堪入目的东西；一条人行道有些许纸屑，不久后就会有更多垃圾，最终人们会视若理所当然地将垃圾顺手丢弃在地上。这个现象，就是犯罪心理学中的破窗效应。

程序员要做到：只要是经过你手的代码，都会比之前好一点。