**更好的分离**

那如果要加载多个就得放置多个 script 标签，如果是加载模块的话，拿 easy.js 来说，页面中始终只要引用 easy.js 即可，这样对于 HTML 和 JavaScript 分离很有好处，在某些场景下这个分离度很重要。

就拿我司的情况来说吧。后端的 view 层并不是由我们前端来开发的，项目上线的时候，前端只能更新 CSS 和 JS 文件，HTML 文件动不了。如果前端要在该页面新增一个 JavaScript 文件的引用是很麻烦的，因为后端程序的更新都要严格按照流程来并有固定的更新时间，如果是用模块加载的方式不用再理会后端的更新。当然，你也许会说，如果原页面中已经有 JavaScript 文件，我直接在原文件中加代码不就行了，那么接下来说说直接在原文件中新增代码会碰到什么样的问题。

**更好的代码组织方式**

如果单个文件越来越大，维护起来出问题的几率也会越来越大，一个人开发还好，如果是多人开发，不同的代码风格，超多的业务逻辑混杂在一起，不要说维护了，光想想都蛋痛。模块式的开发，一个文件就是一个模块，控制了文件的粒度，每个模块可以专注于一个功能。正所谓一个萝卜一个坑，多人开发时，各自管好自己坑里的萝卜就行了，这也正是 OOP 的思想。

**按需加载**

还是围绕单个文件来说事，当文件大到一定的程度，性能问题也随之而来了。合并文件是能减少请求，这是会带来性能的提升，但是当文件大到一定的体积时，此时的下载时间可能并不会比多个小文件的下载时间更短。此时就需要权衡请求数和文件体积的关系了。

单文件还有一个问题，那就是缓存是否能充分的利用好。如果一个大体积的文件内包含了超多的业务逻辑和复杂的功能，而这个文件同时又被很多页面引用到。比如在某页面，实际只用到了该文件一个很少的功能，那么其他的代码对于该页面来说就是多余的，浪费了加载流量。你或许会说，这个文件虽然大，但是它第一次加载的时候就被缓存过了，尽管在某页面只用到了极少部分的功能，但只要缓存过，加载还是挺快的。是的，如果情况有这么理想肯定是好事。但是，往往理想和现实都会有差距，如果产品的迭代更新太快，业务需求一天一个样，那么该文件就会三天两头的更新，更新可能是一个很小的功能，但是这样的更新对于刷新缓存的代价可是很大的。如果确实有这种情况，就说明这种设计确实是有问题。如果能合理的对文件进行模块化的管理，那么可以尽量减少不必要的加载，尽量减少刷新大文件的缓存带来的损失，这也需要权衡好，比如将很少更新的合并成一个文件，常更新的独立成模块。

**避免命名冲突**

JavaScript 本身是没有命名空间的，为了避免命名冲突，经常会使用对象和闭包的办法来避免。用对象仅仅是降低了冲突的概率而已，拿经常使用 jQuery 的开发来说，无论是往 $ 上扩展还是在 $.fn 上扩展，人多了难免会起冲突。或者用自定义的对象，搞个好几层，不光是写起来难记，这样的调用也会在性能上打折扣的。模块化就很好的解决了这个问题，在该模块内的任何形式的命名都不会再和其他模块有冲突，你想起啥名都行了。当然，你硬要在模块内部给 window 上挂一堆东西，我也没办法。

既然每个模块都是封闭式的，那么模块之间如何通信呢？接口的设计那是必须的。很简单，对于像 seajs 这种 CMD 规范的，需要使用自定义的关键字来向外部暴露一个接口。

define(function( require, exports ){  
    var hello = 'hello world';  
    // 向外部暴露该模块的接口  
    exports.hello = hello;  
});

easy.js 遵循 AMD 规范，暴露接口更简单，直接用 return 关键词即可。

define( 'hello', function(){  
    var hello = 'hello world';  
    // 向外部暴露该模块的接口  
    return hello;  
});

**更好的依赖处理**

传统的开发模式，如果 B 文件要依赖 A 文件，那么必须在 B 文件前面用 script 的形式先加载好 A 文件。如果有一天，B 文件不再需要依赖 A 文件，或者要增加依赖文件 C，那么又回到了我说的第一个问题上。如果这个 B 文件被 N 个页面在调用，而且页面还跨业务站点，那改起来简直就是噩梦啊。如果是用模块化，只需要在模块内部声明好依赖就行了，增加删除都直接修改模块即可。调用的时候也不用管该模块依赖了哪些其他模块，放心的用就是了。

好了，讲了这么多，如果你还是觉得无动于衷，要么是你接触到的项目较小，还没复杂到这个程度