JS引擎解析过程

### JS的解释阶段

首先得明确： **JS是解释型语言，所以它无需提前编译，而是由解释器实时运行**

引擎对JS的处理过程可以简述如下：

1. 读取代码，进行词法分析（Lexical analysis），然后将代码分解成词元（token）

2. 对词元进行语法分析（parsing），然后将代码整理成语法树（syntax tree）

3. 使用翻译器（translator），将代码转为字节码（bytecode）

4. 使用字节码解释器（bytecode interpreter），将字节码转为机器码

最终计算机执行的就是机器码。

为了提高运行速度，现代浏览器一般采用即时编译（JIT-Just In Time compiler）

即字节码只在运行时编译，用到哪一行就编译哪一行，并且把编译结果缓存（inline cache）

这样整个程序的运行速度能得到显著提升。

而且，不同浏览器策略可能还不同，有的浏览器就省略了字节码的翻译步骤，直接转为机器码（如chrome的v8）

总结起来可以认为是： **核心的JIT编译器将源码编译成机器码运行**

### JS的预处理阶段

在正式执行JS前，还会有一个预处理阶段（譬如变量提升，分号补全等）

预处理阶段会做一些事情，确保JS可以正确执行，这里仅提部分：

**分号补全**

JS执行是需要分号的，但为什么以下语句却可以正常运行呢？

console.log('a')

console.log('b')

原因就是JS解释器有一个[Semicolon Insertion](http://inimino.org/~inimino/blog/javascript_semicolons)规则，它会按照一定规则，在适当的位置补充分号

譬如列举几条自动加分号的规则：

* 当有换行符（包括含有换行符的多行注释），并且下一个token没法跟前面的语法匹配时，会自动补分号。
* 当有}时，如果缺少分号，会补分号。
* 程序源代码结束时，如果缺少分号，会补分号。

于是，上述的代码就变成了

console.log('a');

console.log('b');

所以可以正常运行

当然了，这里有一个经典的例子：

**function** **b**() {

**return**

{

a: 'a'

};

}

由于分号补全机制，所以它变成了：

**function** **b**() {

**return**;

{

a: 'a'

};

}

所以运行后是undefined

**变量提升**

一般包括函数提升和变量提升

譬如：

a = 1;

b();

**function** **b**() {

console.log('b');

}

**var** a;

经过变量提升后，就变成：

**function** **b**() {

console.log('b');

}

**var** a;

a = 1;

b();

这里没有展开，其实展开也可以牵涉到很多内容的

譬如可以提下变量声明，函数声明，形参，实参的优先级顺序，以及es6中let有关的临时死区等