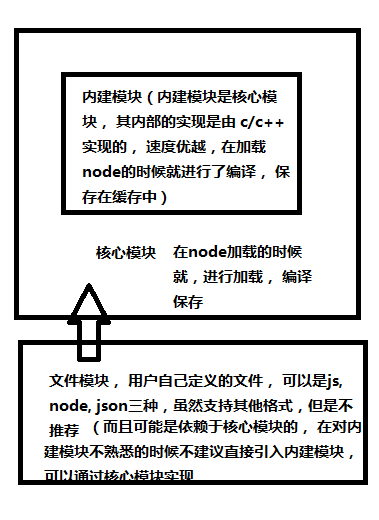
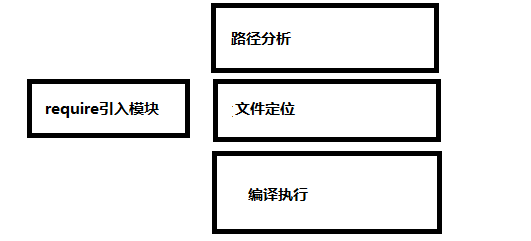
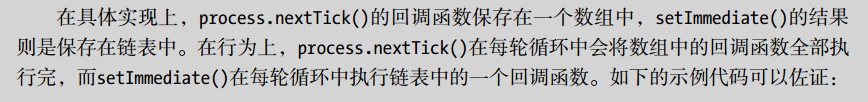


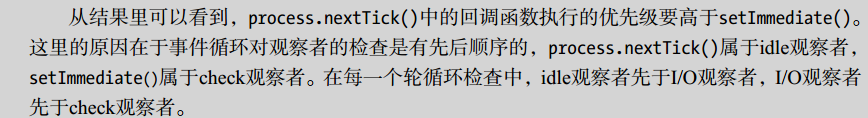
Node 在表面上是和js一样是 单线程的, 这是由于在运行的时候, 都是在主线程中运行的, 但是node内部是实现了线程池的, 通过webwork 通过线程池可以增加cpu的使用率, node 内部的异步IO 都是通过事件驱动的, 在最开始的时候 是通过 轮询来 对每一个异步的IO进行询问, 花费的时间使得cpu在轮询时间上其实并没有做其他的事情, 这样就将异步IO的优势大大降低了, 在不断的优化过程中, 出现了消息通知机制的, 最后都是通过事件驱动, 同时利用线程池完成 每一个事件完成的函数回调, 每一个事件都至少有一个观察者, 每一个观察者 可能有多个的 事件等待他的观察, windows中是利用了IOCP(Windows自己的线程池)来实现线程池的, 其他平台 应该是通过libuv这个 node的底层跨平台机制自己实现的线程池, 异步io是整个node的核心不管是 网络IO ， 文件IO， 等， 异步IO 实现了 速度上的飞速提升， 也使得它能够完成服务器的工作， 将整个js可以贯穿于 前后端

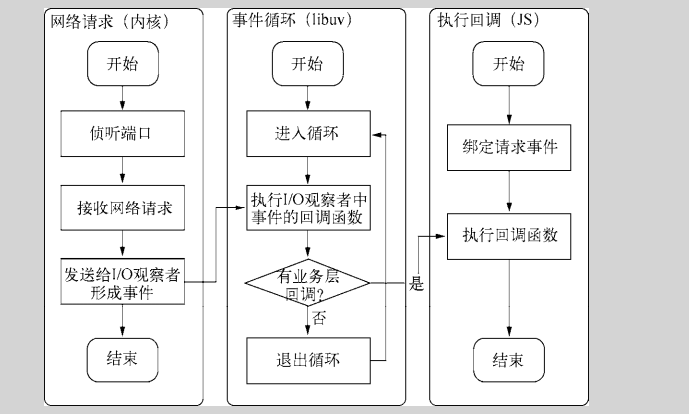
Node 的导出 和引入， 在node中分为 内建模块， 核心模块 ， 文件模块， 他们之间的关系

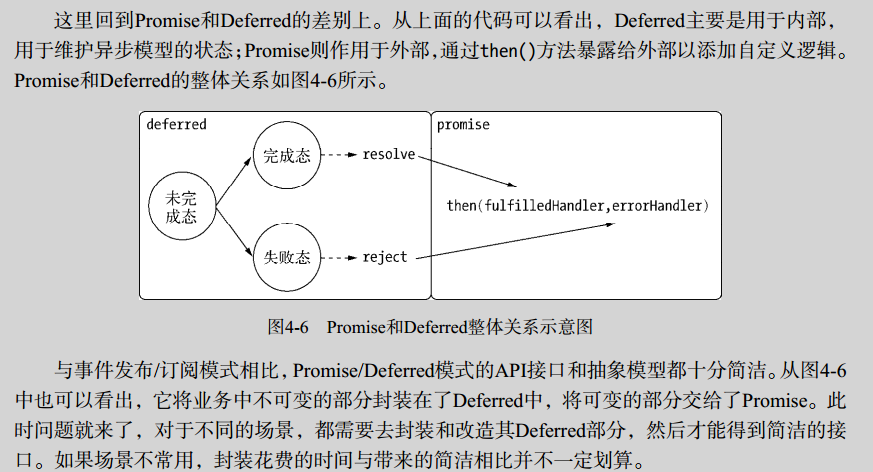


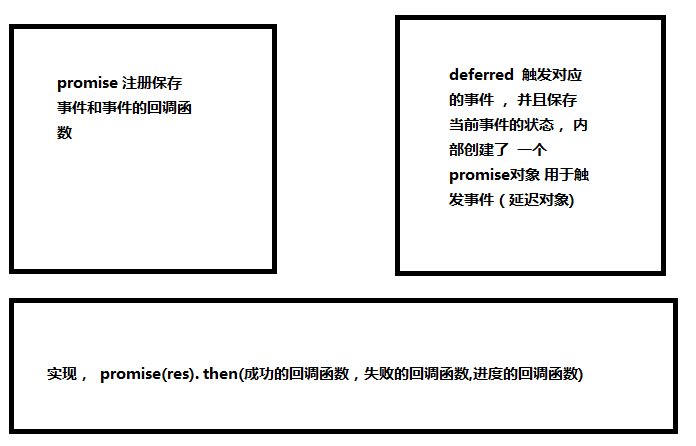


Node中的事件循环 ：  


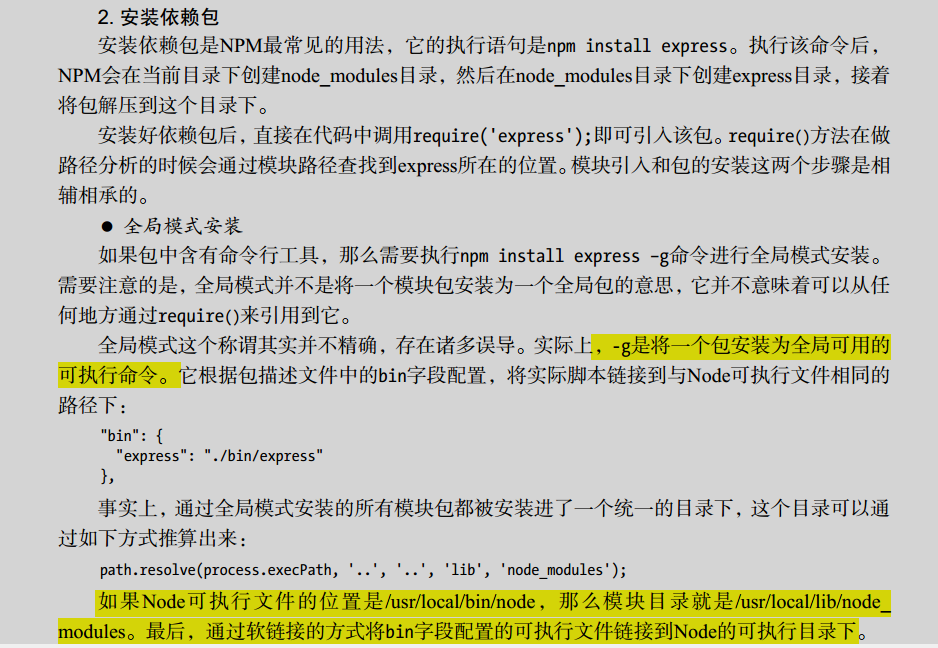




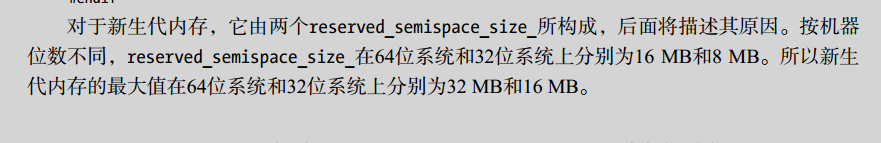
事件的嵌套回调， 在之前 node由于是事件驱动为核心的， 所以大多数的函数都是事件， 加上回调函数， 函数内部嵌套严重， 通过promise/deferred， 可以实现先回调，后执行， 成功解决嵌套严重的问题



Npm安装包



V8限制 内存 在64位电脑上 限制是1.4G 32位约为 0.7G



堆内存中 分为 老生代 和 新生代 在新生代中 主要通过scavenge算法 来完成新生代的内存管理 ， 将新生代分为两个semispace 空间 ， 一个叫From 一个叫To,在单纯的scavenge过程中， From的存活对象会 复制到 To空间中去， 然后反转两个空间的角色对象， 进行下一次的转换 复制，



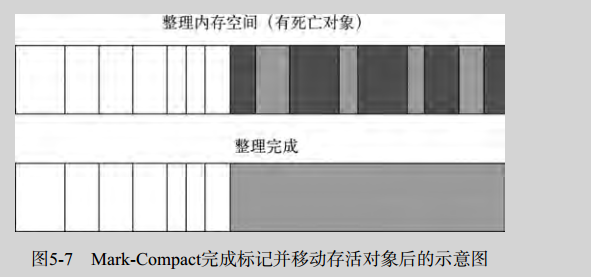
对象晋升 ： 将新生代的对象 移动到老生代中， To空间的内存占用比 为 25% 由于 scavenge算法后需要转换空间 所以To空间转换后 会用来分配新的对象的堆内存地址

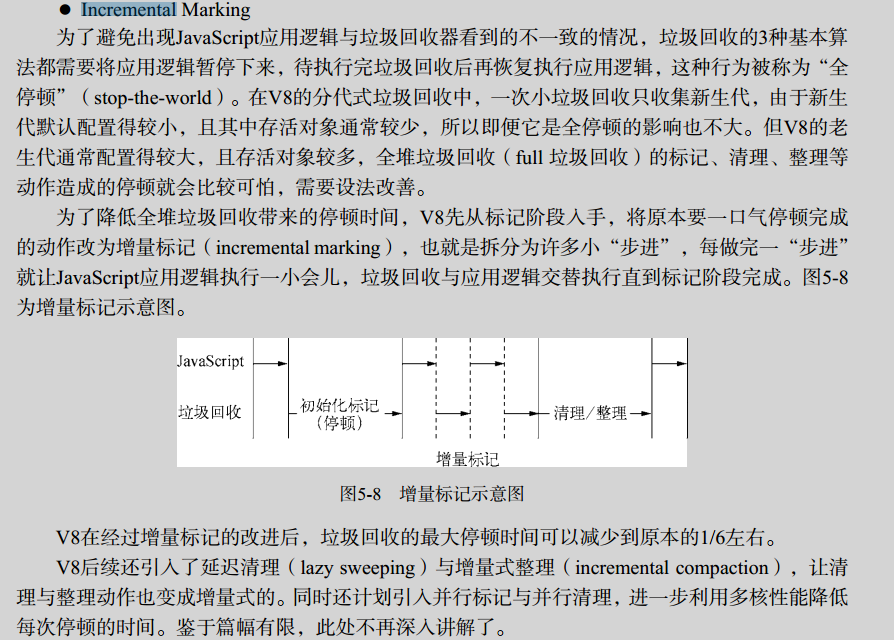
Scavenge算法是使用空间换时间的算法， 大大的提高了速度， 但是将整个的新生代才分成了两个空间， 浪费掉了空间

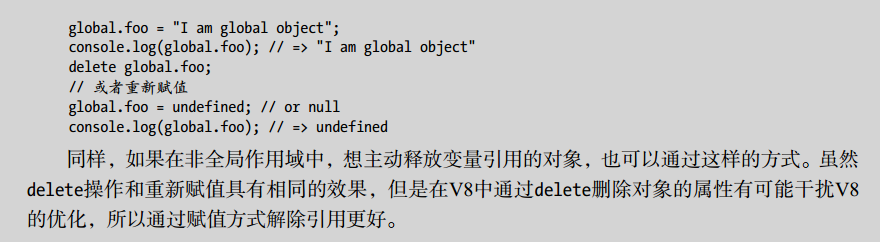
在老年代 ， 中由于大量的存在 长时间存活的对象， 所以这种方式不太实用， 于是切换了算法 使用的是Mark\_Sweep 和 Mark\_Compact相结合的方式进行垃圾回收

Mark\_Sweep 标记清除 ： 在整个老生代中 ， 大量的对象存活的时间较长， 所以这种方式， 在标记阶段， 会遍历所有的对象， 然后将活着的对象标记， 在随后的清除阶段，将没有标记的对象清除 Mark\_Sweep只清理没有标记的对象 大大的提高了效率



由于这种标记清理的内存是不连续的 就存在着清理的空间是碎片化的， 可能在存储一个大的对象的时候， 没有一个连续的内存能存储下， 就会提前进行垃圾回收了，所以提出了 Mark\_Compact

这种方式是将活着的对象全部移动到另一端， 然后清空剩余的部分， 使得清理出来的空间是连续的



堆外内存 Buffer