《模型-视图-视图框架模型基于多任务的改进方案》

绪论

目录:

1.JavaScript简介 2.JavaScript单线程异步 3.JavaScript异步产生的问题,以及现有解决方案

**1 JavaScript**的简介

JavaScript是一个轻量级解释型向对象但却将函数视为语言中一级公民的编程语言。JavaScript同时支持基于原型的面向对象编 程、命令式编程风格和函数式编程风格,是一种基于原型的、多范式的动态脚本语言。

JavaScript由网景公司雇佣的Brendan Eich实现。它的名字借鉴了当时大行其道的Java并且命名为JavaScript,基本语法借鉴C语言 与Java,数据结构借鉴了Java,函数借鉴了Scheme和Awk,将函数视为一等公民,并且引入了闭包概念。原型链的继承模式借鉴 了Self语言。正则表达式则借鉴了Perl。一些函数和方法借鉴了Python。但是为了保持简单,语言本身缺少了一些关键功能,比如 命名空间、模块等。但是因为语言的动态特性,这些语言都是可以利用现有的功能实现的。所以也造就了学习别的语言通常是学习 如何使用语言的功能,而学习JavaScript通常是学习各种解决问题的模式。

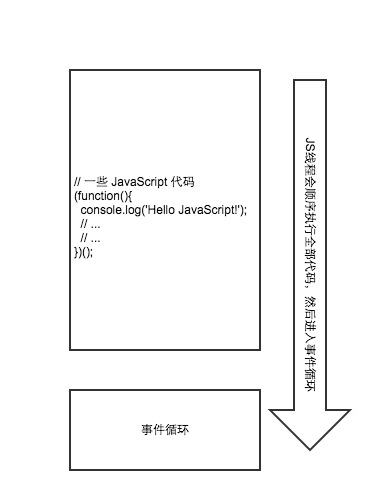
JavaScript最常被用在编写网页脚本上,但是随着Google的V8引擎的发布,JavaScript的效率被瞬间提高了降级20倍,并且接近原 生代码的执行速度。现在JavaScirpt已经别运用在了各个领域当中,比如用于开发移动智能的应用、桌面计算机上的应用、服务器 上的应用、游戏引擎中的脚本,跟甚至跟随这Intel进入了嵌入式应用的领域。比起Java,JavaScript最接近当年Java对世人许下的 承诺:“一处编写,到处运行”。

**2 JavaScript**的单线程与异步事件驱动 **2.1 JavaScript**单线程

JavaScript的单线程是指JavaScript运行时有且只有一个线程在执行JavaScript代码。正因为JavaScript的运行时是单线程的,一旦 被阻塞将会阻塞整个JavaScript引擎。所以JavaScript非常多的API被设计成了非阻塞(no-blocked)的模式,例如AJAX等。当 JavaScript引擎执行到异步非阻塞的AJAX时并不会阻塞等待AJAX请求的结果,而是继续执行接下来的代码。当AJAX请求返回结果 时,会产生一个事件压入JavaScript引擎的消息队列中,最终在事件循环中执行事件的处理的操作。

所以在代码中马上获取异步非阻塞的AJAX是不可能的。(注:虽然大多数浏览器都支持同步的AJAX,但是因为同步的AJAX会因为 等待请求返回而使得整个web页面失去响应,所以绝大多数规范中都命令禁止使用同步的AJAX)。

JavaScript线程会逐行执行所有的JavaScript代码后进入事件循环,通过轮询的方式等待以及处理事件。



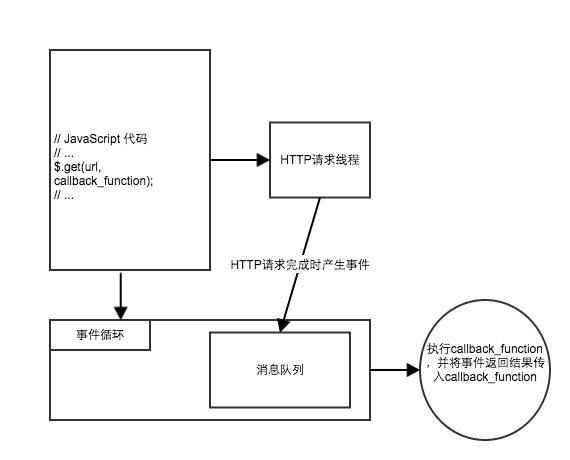
**2.2** 异步事件驱动

正如前面所说,JavaScript的单线程是指JavaScript的运行时有且只有一条线程在执行JavaScript代码。但是我们的浏览器并不是单 线程的,也不可能是单线程的。以Webkit为例,一个页面有JavaScript引擎线程、界面渲染线程、浏览器事件触发线程、Http请求 线程等线程。浏览器很多行为被设计成了异步的,例如,点击鼠标事件,定时器事件,以及XMLHttpRequest的回调等。当一个产 生一个异步事件了以后,它就被会压入一个消息队列。然后由事件循环(Event Loop)中对消息队列进行轮询并进行事件处理事 件。

以AJAX为例,当通过AJAX发出了一个请求,浏览器会开一个线程去请求相应的数据,当请求完成了以后会产生一个事件压入消息 队列。在事件循环中处理AJAX,执行回调并且将AJAX请求返回的结果作为参数传入回调函数。

**3 JavaScript**异步产生的问题,以及现有解决方案

**3.1** 回调地狱(**Callback Hell**) 回调地狱是指在编写JavaScript异步代码时产生的一个导致可维护差的问题。其中一个典型的例子是多重嵌套回调。





这种嵌套的回调函数难以理解,开发人员需要仔细分析哪些代码用于应用的业务逻辑,而哪些代码处理异步函数调用的,代码结构

支离破碎。错误处理也分解了,我们需要在各个地方检测错误的发生并作出相应的处理。

**3.2** 现有解决方案 由于回调地狱产生的非常多的现有解决方案:

**3.2.1 Promise**

Promise最初在E语言中被提出,它是基于并列/并行处理设计的一种编程语言。Promise是一种抽象异步处理对象以及对其进行各种 操作的组件。Promise也是一种同步函数与异步函数联系和通讯的方式。可以让异步函数返回值,并且能够抛出异常。

**3.2.2 Async**

Async主要解决异步的迭代难题以及异步函数工作次序的的难题,主要在Node.js上,最常见的是做IO操作上。

**3.2.3 Step**

Step是极简主义的工作流库。相比Async,Step的实现非常简单,功能也十分单一却十分实用。

**2** 需求分析

上一章已经介绍过JavaScript回调在工程应用产生的问题,同时也介绍了几种解决问题的方 案。但是各种各种方案都有其不足的地方,所以我们尝试设计一个更为优秀的解决方案来解决 不仅仅是几个异步操作顺序工作流这么简单的问题了,而是更高层次的抽象,将整个 JavaScript工程的控制流全部统一抽象出来,通过管道相连接,所有的操作都封装成一个任 务。

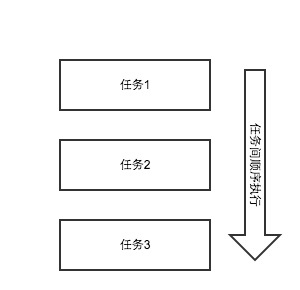
现有解决方案的不足

现有解决方案虽然虽然能够简单解决基本需求的问题，但是有一个非常关键的缺陷，所有解决的问题都是工作流而不是控制流。一个任务没有办法回退到上一步任务，一个任务没有办法循环自己的任务。总而言之就是任务里面没有办法对整个序列做出改变，不能在任务执行中对整个任务序列做动态的调整。

**1** 功能模型

**1.1** 任务间顺序执行

任务间严格按照顺序串行执行。



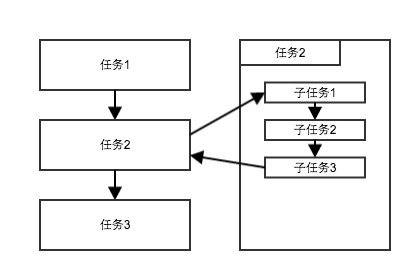
# 1.2 管道

因为管道，才使一个序列能够按顺序得处理一个任务产生意义。上一个任务产生的输出成为下一个任务的输入。在这种情况下可以非常简单的将任务直接组合复用，每一个任务也可以看成管道的一个过滤器。



**1.2** 子任务

一个任务可以拥有子任务,住任务工作流会等待子任务全部按顺序执行完成后才继续执行主工

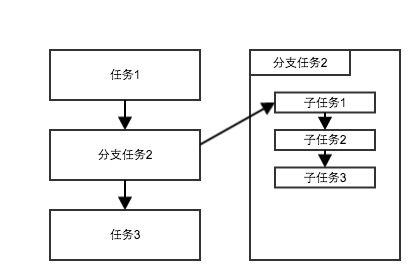


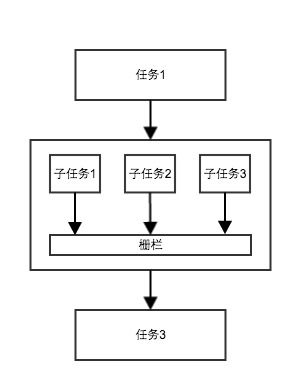
作流。

**1.3** 分支任务 分支类似于子任务,但是主任务工作流并不会等待分支任务执行完成,而是开启新的分支任务

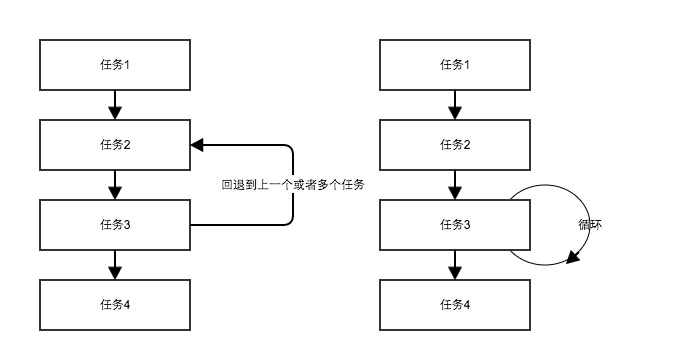
后立即执行下一个任务。

**1.4** 栅栏 等待当多个任务都执行完成后(无顺序先后),才继续执行下一个任务。





# 1.5 回退/循环任务



1.6 fork

通过一个任务可以fork出一个拥有相同基本任务的任务对象。

# 3架构设计

模型 - 视图 - 控制器（MVC）  
  
模型 - 视图 - 控制器（MVC）是一种软件架构模式实现用户界面。它把一个给定的软件应用程序分为三个相互关联的部分，以便分离从该信息提供给或从用户接受的方式信息的内部表示。[1] [2]  
  
内容  
  
    1概述  
        1.1组件  
        1.2相互作用  
    2使用Web应用程序  
    3历史  
    4参见  
    5参考  
    6外部链接  
  
概观  
  
与其他软件模式，MVC的表达到的问题在“解决方案的核心”，同时允许它适于每个系统。[3]的特殊的MVC架构可以与传统的描述在这里显著变化。[4]  
组件  
MVC组件的典型合作  
  
MVC中，模型，所述的中心部件捕获该应用程序的行为在其问题域，独立于用户界面的方面。[5]该模型直接管理的数据，逻辑和应用的规则。视图可以是信息，诸如图表或图中的任何输出表示;同一信息多种视图是可能的，如条形图管理和会计表格视图。的第三部分，所述控制器，接受输入，并将其转换为模型或视图的命令。[6]  
互动  
  
除了将应用程序划分成3种成分中，模型 - 视图 - 控制器设计定义它们之间的相互作用。[7]  
  
    控制器可以发送命令到模型以更新模型的状态（例如，编辑文档）。它也可以发送命令到它的相关联的视图改变模型的视图的呈现（例如，通过一个文档滚动）。  
    所检索由控制器，并在视图中显示的模型存储数据。每当有一个变化的数据是由控制器进行更新。  
    视图从它使用，以产生输出的表示给用户的模型请求信息。  
  
在使用Web应用程序  
  
虽然最初开发用于桌面计算，模型 - 视图 - 控制器已被广泛采用，作为在主要的编程语言万维网应用的架构。几个商业和非商业的Web应用程序框架已创建的执行模式。这些框架各有不同，解释，主要的方式，MVC的职责是在客户端和服务器之间进行分配。[8]  
  
早期的Web MVC框架，采取了瘦客户机的办法，放在了几乎整个模型，视图和控制器逻辑在服务器上。在这种方法中，客户端发送任何超链接的请求或形式输入到控制器，并然后接收从视图的完整的和更新的网页（或其他文件）;该模型完全存在于服务器上。[8]作为客户端技术已经成熟，框架，如AngularJS，Ember.js，JavaScriptMVC和骨干已创建允许MVC组件部分在客户机上执行（另见阿贾克斯）。  
历史  
  
MVC是一家在图形用户界面的早期发展的开创性的见解，并描述和实施其职责方面的软件构造的第一途径之一。[9]  
  
特里夫Reenskaug引入到MVC的Smalltalk-76，而来访的Xerox PARC的[10] [11] 20世纪70年代。在20世纪80年代，吉姆阿尔索夫和其他实施了Smalltalk的-80类库版本的MVC。只是到了后来，在对象技术杂志1988年的一篇文章，说的MVC表示为一个总的概念。[12]  
  
MVC模式已经随之进化，[13]从而产生变种，如HMVC，MVA，MVP，MVVM，和其他人适合模型视图控制器不同的语境。

# Model View ViewModel

模型视图模型（MVVM）是一种架构模式进行软件开发。  
  
MVVM是Martin Fowler的演示模型设计模式的变体。[1] [2]象福勒的演示模式，MVVM抽象视图的状态和行为。[1]然而，尽管呈现模型抽象的视图（即创建一个视图模型）在不依赖于特定的用户界面平台的方式，MVVM开发在微软专门通过接口 - 利用Windows演示基础（WPF）（微软的.NET图形系统）和功能，以简化用户的事件驱动编程的Silverlight（WPF的互联网应用导数）。[1]  
  
约翰Gossman，微软的WPF和Silverlight的建筑师之一，MVVM宣布他在2005年的博客。  
  
MVVM和演示模型无论是从模型 - 视图 - 控制器模式（MVC）获得。 MVVM方便的图形用户界面的发展的一个分离（无论是作为标记语言或GUI代码）从业务逻辑或后端逻辑（数据模型）的发展。 MVVM的视图模型是一个值转换器; [3]，这意味着该视图模型负责暴露在这样​​一种方式，对象被容易管理和消耗从模型的数据对象。在这方面，认为模型比视图多个模型，并处理最如果不是全部的视图的显示逻辑。[3]视图模型还可以实现调解模式，组织访问后端逻辑周围的用例由视图支持。  
  
模型 - 视图 - 视图模型也被称为模型 - 视图 - 粘合剂，特别是在实现，不涉及.NET平台。 ZK（用Java编写的Web应用框架）和KnockoutJS（JavaScript库）使用模型 - 视图 - 粘合剂。[1] [4] [5]  
  
内容  
  
    1组件的MVVM模式  
    2基本原理  
    3批评  
    4参考  
    5外部链接  
  
的MVVM模式组件  
  
模型  
    模型是指无论是一个域模型，它代表了真实情况的内容（一个面向对象的方法），或者，表示的内容（数据为中心的方法）的数据访问层。[来源请求]  
  
视图  
    如在MVC和MVP模式，视图是用户界面（UI）。[进一步解释需要]  
  
视图模型  
    视图模型是公开的公共属性和命令视图的抽象。取而代之的是MVC模式的控制器，或MVP模式的主持人，MVVM有粘合剂。在视图模型，这个粘合剂介导的视图和数据粘合剂之间的通信。[澄清需要]视图模型已被描述为在模型中的数据的状态。[6]  
  
粘合剂  
    声明数据和命令绑定是隐含在MVVM模式。在微软溶液堆栈，粘合剂是一种标记语言，称为XAML。[7]的粘合剂释放被被迫写锅炉板逻辑同步视图模型和视图显影剂。当微软之外实现堆叠的声明式数据绑定技术的存在是一个关键推动的格局。[4] [8] [澄清需要]  
  
合理  
  
MVVM旨在利用数据绑定功能在WPF更好的方便视图层的发展，从图案的其余部分分离，通过从视图层移除几乎所有的GUI代码（“代码隐藏”）。[9]相反需要的用户体验（UX）开发人员编写GUI代码，他们可以使用该框架标记语言（如XAML），并创建数据绑定到视图模型，这是书面和应用程序开发人员维护。这种角色的分离使得互动设计师专注于UX需求，而不是业务逻辑编程。应用程序的层因此可开发在多个工作流为提高生产率。即使当一个开发人员的工作对整个代码库从模型视图的适当分离，更高效的用户界面通常是经常变化和根据最终用户的反馈开发周期的后期。  
  
MVVM模式尝试获得既由MVC提供的功能开发的分离的优点，同时利用数据绑定的优点和框架通过结合数据作为接近纯应用模型越好。[9] [10] [11] [需要澄清]它使用的粘合剂，视图模型，以及任何业务层“数据检查功能来验证输入的数据。其结果是，该模型和框架驱动尽可能多的操作的可能，消除或减少其直接操纵的视图应用逻辑（例如，代码隐藏）。  
批评  
  
的模式的批评来自MVVM创始人John Gossman自己，[12]谁指出，在实施MVVM的开销是“杀鸡用牛刀”的简单的UI操作。他还指出，对于较大的应用，推广视图模型变得更加困难。此外，他说明了数据非常大的应用程序绑定可以造成相当大的内存消耗。

管道（软件）  
这篇文章是关于软件管道的一般介绍。对于原执行贝壳，看到管道（UNIX）。  
  
在软件工程中，管道由一连串的处理元件（进程，线程，协程，函数等），设置成，使得每个元件的输出是下一个的输入;该名称是由类似于物理管道。通常缓冲一定量的连续元件之间提供的。在这些管道中流动的信息是通常的记录，字节或比特，和一个管道可以被称为滤波器的元流;这也被称为管道和过滤器的设计模式。连接元件进入管道类似于函数组合。  
  
狭义地讲，一个管道是线性的，单向的，尽管有时将术语适用于更一般的流动。例如，一个主要的单向管线可以具有在其他方向一些通信，被称为一个返回信道或反向信道，如在词法分析器劈或管道可以是完全双向的。与一个方向的树和向无环图的拓扑流动行为类似于（线性）管道 - 缺乏循环使得它们简单 - 并且因此可以被松散地称为“管道”。  
  
内容  
  
    1日实施  
    2 VM / CMS和MVS  
    3管线对象  
    4管线的图形用户界面  
    5其他注意事项  
    6参见  
    7笔记  
    8外部链接  
  
履行  
  
管线通常在一个多任务操作系统实现，通过启动所有的元素同时随着流程，并自动维修数据读取由每个过程与写的上游过程中的数据的请求 - 这可以被称为一个多道管道。在这种方式中，CPU将自然的处理由调度器之间切换，以便最大限度地减少它的空闲时间。在其他常见车型，元素被实现为轻量级线程或协程，以减少操作系统开销往往涉及流程。根据不同的操作系统时，线程可以直接由OS或由线程管理器调度。协程总是定于某种形式的协同程序经理。  
  
通常，读取和写入请求被阻塞操作，这意味着在源处理的执行，书写时，暂停，直到所有的数据可以被写入到目的过程中，以及，同样地，在目标处理的执行，通过阅读，被暂停，直到至少一些所请求的数据的可从源方法得到。这不能导​​致死锁，其中两个进程将无限期地等待对方响应，因为这两个过程中的至少一个将在此后不久有其请求的服务由操作系统，并继续运行。  
  
出于性能，实现管大多数操作系统都使用管道缓冲区，它允许源过程中提供更多的数据比目标的过程是目前能够或愿意接受。在大多数的Unices和类Unix操作系统，一个特殊的命令也可实现其潜在的更大，配置大小的管道缓冲区，通常被称为“缓冲”。这个命令可以是有用的，如果在目标过程比源过程显著慢，但是它无论如何需要使源进程可以尽快完成其任务。例如，如果源过程包括其中从CD读取的音频轨道和目的地过程包括其中压缩波形音频数据，如MP3格式的命令的命令。在这种情况下，缓冲整个轨道的管道缓冲区将允许CD驱动器更快地降速，并且使用户能够从驱动器中取出光盘之前，编码过程已经完成。  
  
可使用系统来实现，例如一个缓冲器命令调用用于读取和写入数据。浪费忙等待，可避免使用的设施，如投票或选择或多线程。  
VM / CMS和MVS  
  
CMS管道是管道想法VM / CMS和MVS系统的端口。它支持更复杂的管道结构比Unix shell中，与步骤服用多种输入流，并产生多个输出流。 （这种功能是由Unix内核的支持，但很少使用的程序，因为它使得复杂的语法和阻塞模式，虽然有些炮弹通过做任意的文件描述符分配支持它）。由于IBM的大型机操作系统的不同性质，它实现了内部CMS管道而在Unix中是独立的外部程序很多步骤，也可以要求其功能独立的外部程序。此外，由于文件在IBM大型机上的面向记录的性质，管道在面向记录的，而不是面向流的方式。[编辑]操作  
管道对象  
  
旁字节流基管线，也有对象管道。在一个对象的管道，处理元件的输出对象，而不是文字。 Windows PowerShell中包含了PowerShell运行时内部功能之间传输的.NET对象的内部对象的管道。频道，在凌波编程语言，IPython的ipipe扩展发现是这个比喻的其他例子。  
管线的GUI  
  
图形化的环境中，如RISC OS和ROX桌面还利用管道。而不是提供一个包含文件管理器让用户指定的程序应该写数据保存​​对话框中，RISC OS和ROX提供了保存包含的图标（和一个字段指定的名称）对话框。目的是通过拖放图标指定。用户可以在任何地方放下图标已经保存的文件可能被丢弃，包括到其他程序的图标。如果图标拖放到一个程序的图标，它的加载和，否则将被保存在内容上的新程序的标准输入流传递。  
  
例如，用户浏览万维网可能会遇到他们想要编辑和重新上传一个.gz的压缩图像。使用GUI管道，他们可以拖动链接到他们去归档程序，拖动代表所提取的内容，以他们的图像编辑器的图标，编辑，打开另存为对话框，然后拖动图标到他们上传软件。  
  
从概念上讲，这种方法可与常规的保存对话框使用，但是这将需要用户的方案，以具有在可导航到文件系统明显和易于访问的位置。在实践中，这通常是不是这种情况，那么GUI管道是罕见的。  
其他注意事项  
  
名称'管道'来自粗略类比与该管道的物理管道通常[1]允许信息流只在一个方向，像水经常流动的管道。  
  
管道和过滤器可以被看作是功能的编程的一种形式，使用字节流数据对象;更具体地，它们可以被看作是单子用于I / O的一种特殊形式。[2]  
  
管线的概念也是中央对茧的Web开发框架或任何Xproc的（W3C的标准）的实现，它允许一个源流最终显示之前进行修改。  
  
这种模式鼓励使用文本流作为节目的输入和输出。这依赖于文字具有创建图形壳时文本程序进行核算。

我的架构设计

需求属性：

JavaScript多任务模型的：易用性，可扩展性。

WEB框架：易用性，可扩展性。

选取架构：

JavaScript单线程多任务的设计非常适合选用管道/过滤器设计。但是不遵守传统的管道/过滤器架构的体系。但是不遵循传统的管道过滤器架构模型。因为传统的管道过滤器模型，过滤器只是过滤数据而不会修改处理数据，也不会附加数据。这在控制流中是不可能实现的，因为控制流中需要严格的工序对数据进行处理加工以及附加。但是在整个数据流上来看，这个数据的流动也和管道非常类似。

WEB框架则套用现在最为主流的MVVM层次架构。

# 概要设计

多任务系统的设计

1. 单线程下的多任务

如上面所降到的一样，JavaScript Runtime始终只有一条线程在执行JavaScript代码，那是不是意味着任务间只能串行而无法并行完成了呢？这显然不是，如同计算机发展史上的多任务一样，一开始只有单CPU，单线程。但是仍然能够完成多任务的并行操作，这是如何完成的呢？

多任务处理

现代桌面操作系统能够处理大量不同的过程的同时。该截图显示了Linux Mint的同时运行Xfce桌面环境，火狐，一个计算器程序，内置万年历，Vim的，GIMP，以及VLC媒体播放器。  
  
在计算中，多任务是通过同时执行它们执行多个任务（也称为工艺）在一定的时间周期的概念。新任务开始和中断已经开始的人，他们已经达到了，而不是在执行任务之前完成，所以按顺序启动每个任务需要一个新的开始之前达到其结束。其结果是，使计算机执行以交织方式的多个任务分段，而任务共享公共处理资源，如中央处理单元（CPU）和主存储器。  
  
多任务并不一定意味着多个任务正在执行在完全相同的时间。换言之，多任务并不意味着并行执行，但它意味着一个以上的任务可以在同一时间，和一个以上的任务是在给定时间周期前进超过通过执行是部分路。  
  
在一台计算机具有单个CPU的情况下，只有一个任务被认为是运行在任何时间点上，这意味着在CPU正在积极执行指令该任务。多任务解决由调度问题哪个任务可能是一个在任何给定时间运行，而当另一等待任务得到一个转弯。重新分配的CPU从一个任务到另一个的动作被称为上下文切换。当上下文切换发生的足够频繁，并行的假象得以实现。  
  
即使在多处理器或多核计算机有多个CPU /核心让多个任务可以立即执行（物理，每个CPU或核心之一），多任务允许运行更多任务以外还有的CPU。一词多任务处理已成为国际来看，由于同一个词被用在许多其他语言，如德语，意大利语，荷兰语，丹麦和挪威。  
  
操作系统可以采取许多不同的调度策略，一般分为以下几个类别：[来源请求]  
  
    在多道系统，正在运行的任务保持运行，直到执行需要等待外部事件（如从磁带读取）或到计算机的调度交换强行将正在运行的任务出CPU的操作。多道系统的设计最大限度地提高CPU使用率。  
    在分时系统中，正在运行的任务，需要放弃CPU，自愿或由外部事件诸如硬件中断。分时系统被设计为允许多个程序同时明显地执行。  
    在实时系统中，一些等待任务，保证给予CPU当外部事件发生。实时系统被设计来控制的机械装置，例如工业机器人，需要及时处理。  
  
内容  
  
    1多道  
    2协同多任务处理  
    3抢占式多任务  
    4实时  
    5多线程  
    6内存保护  
    7内存交换  
    8编程  
    9参见  
    10参考  
  
多道  
  
在计算的初期，CPU时间是昂贵的，和外设都非常缓慢。当计算机运行，需要访问外围设备的程序时，中央处理单元（CPU）将必须停止执行，而外设处理的数据的程序指令。这被认为是非常低效的。[来源请求]  
  
使用多道系统中的第一台计算机是英国利奥三在计算机存储器被装在分批J.里昂和有限的几个不同的程序所拥有，和第一个开始运行。当第一程序到达的指令等待一个外围设备，该程序的上下文被存储起来，并且在存储器中的第二方案是给运行的机会。这个过程一直持续到所有程序运行完毕。[来源请求]  
  
使用多道是由虚拟存储器和虚拟机技术的到来，这使单个程序来利用存储器和操作系统资源仿佛其它同时运行的方案是，增强了对所有的实际目的，不存在的和不可见的，以它们[编辑]  
  
多道不作出任何保证一个程序将适时地运行。实际上，第一个程序可能非常好小时运行，而不需要访问外围。由于没有用户以交互终端等，这是没有问题：用户交寄的穿孔卡片甲板，以操作员，回来后几个小时后的打印结果。多道时，正在处理多批大大缩短等待时间。[来源请求]  
合作多任务  
另请参见：非抢占多任务处理  
  
表达“时间共享”，通常由指定交互用户终端处，如IBM的TSO共享计算机，和VM / CMS。不再普遍使用的，术语“时间共享”已替换为“多任务”，以下的个人计算机和工作站，而不是共享的交互系统的出现。  
  
早期的多任务系统中使用的应用程序，自愿割让时间彼此。这种做法，这最终被许多计算机操作系统的支持，今天被称为协作式多任务。虽然现在很少用在大型系统中，除了特定的应用，如CICS或JES2子系统，协同多任务曾经是由Microsoft Windows（之前的Windows 95和Windows NT）和Mac OS采用的调度方案（之前的OS X）为了使多个应用程序同时运行。 Windows 9x中也使用合作多任务，但仅限于16位旧应用程序，几乎相同的方式作为预豹的PowerPC版本的Mac OS X中使用了它的经典应用。网络操作系统NetWare上使用协同多任务最多的NetWare 6.5。合作多任务今天仍然使用在RISC OS系统。[1]  
  
作为协作多任务系统依赖于每个进程定期向其他进程放弃系统上的时间，人们设计不当程序可以消耗所有的CPU时间本身，或者通过执行大量计算或由忙等待;既会导致整个系统挂起。在服务器环境中，这是一个危险，使整个环境不可接受脆弱。  
抢占式多任务  
主要文章：抢占（计算）  
  
抢先多任务允许计算机系统，以保证更可靠的每个进程的操作时间定期“片”。它也允许系统具有重要的外部事件，如输入数据，这可能需要一个或另一个过程的立即注意迅速处理。操作系统被开发以利用这些硬件功能和抢先运行多个进程。抢占式多任务负载在DEC公司的PDP-8的电脑，并实现在OS / 360 MFT于1967年，在MULTICS（1964），和Unix（1969）;它是所有类Unix的操作系统，如Linux，Solaris和BSD与及其衍生物的核心功能。[2]  
  
在任何特定的时间，过程可分为两类：一类是那些正在等待输入或输出（称为“I / O绑定”），而那些被充分利用的CPU（“CPU限制”）。在原始的系统中，软件经常会出现“轮询”或“busywait”在等待请求输入（如磁盘，键盘或网络输入）。在此期间，该系统不执行有用的工作。随着中断和抢占式多任务的到来，I / O密集​​型进程可能会被“封杀”，或暂时搁置，待必要的数据的到来，让其他进程能够利用CPU。由于所请求的数据的到来会产生中断，阻塞的进程能够保证及时归还到执行。[来源请求]  
  
提供给家庭用户最早抢占式多任务操作系统是辛克莱QDOS的辛克莱QL，发布于1984年，但很少人买机器。 Commodore的强大的Amiga，发布次年，是第一个商业成功的家用电脑使用的技术，它的多媒体能力使其成为现代的多任务处理的个人电脑一个明确的祖先。微软开发Windows NT 3.1，然后点击Windows 95.后来通过了关于苹果的Macintosh由Mac OS X时，作为一个类Unix操作系统，采用抢占做出抢占式多任务的旗舰操作系统在90年代初的一个核心特征多任务处理所有的本地应用程序。  
  
类似的模型被用来在Windows 9x和Windows NT的家庭，其中原生32位应用程序抢先式多任务，和传统的16位Windows 3.x的程序都在一个协同处理多任务，虽然在NT系列有可能给力的16位应用程序作为一个单独的抢先式多任务进程中运行。[3]的Windows 64位版本，既为X86-64和Itanium架构，不再提供支持传统的16位应用程序，从而提供抢占式多任务为所有支持的应用程序。  
即时的  
  
另一个原因多任务是实时计算系统，其中，有许多的需要由一个单一处理器系统来控制可能无关的外部活动的设计。在这样的系统中的分层中断系统加上进程优先级，以确保关键活动给予更大份额的可处理时间。[来源请求]  
多线程  
  
由于多任务大大提高计算机的吞吐量，程序员开始实施应用，集协作进程（例如，一个过程收集输入数据，一个进程处理输入数据，一个进程写在磁盘上的结果）的。然而，这需要一些工具来允许进程有效地交换数据。[来源请求]  
  
线程是从，最有效的方式为合作进程交换的数据将是分享他们的整个存储空间的想法诞生了。因此，线程是有效地运行在相同的内存上下文并分享他们的父进程等资源的过程，如打开的文件。线程被描述为轻量级进程，因为线程之间的切换不涉及改变存储器内容。[4] [5] [6]  
  
虽然线程抢先调度，某些操作系统提供一个变种线程，命名纤维，被定合作。操作上不提供光纤系统，应用程序可以使用重复调用工人功能实现自己的纤维。纤维甚至比线程更加轻便，比较容易与编程，虽然他们往往会失去一些线程的好处或全部在具有多个处理器的计算机。[来源请求]  
  
有些系统直接支持多线程硬件。  
内存保护  
主要文章：内存保护  
  
必不可少的任何多任务系统是安全和有效地共享访问系统资源。访问存储器必须严格管理，以确保没有进程可以无意或故意读或写存储单元的进程的地址空间之外。这样做是为了一般系统的稳定性和数据完整性，以及数据的安全性的目的。  
  
在一般情况下，存储器访问管理是操作系统内核的责任，在与硬件机制（如存储器管理单元（MMU）），提供支持功能性的组合。如果一个进程试图访问其内存空间之外的存储位置，MMU拒绝请求，并通知内核采取适当的行动;这通常会导致强行终止有问题的进程。根据软件和内核设计和问题的具体错误，用户可能会收到访问冲突错误消息，如“分段错误”。  
  
在一个良好的设计和正确地实施多任务系统中，一个给定的过程不能直接访问存储器属于另一个进程。一个例外是共享内存的情况下;例如，在系统V进程间通信机制的内核内存分配将由双方通过多个进程共享。这些特征经常被用来通过数据库管理软件，如PostgreSQL的。  
  
不足内存保护机制，要么是由于其设计或实现差的缺陷，允许那些可能被恶意软件利用的潜在安全漏洞。  
内存交换  
  
使用交换文件或交换分区是一种为操作系统提供更多的内存比实际可用以保持主存储器的部分在二级存储。而多任务和内存交换是两个完全不相关的技术，它们经常一起使用时，作为交换存储器允许多个任务到同时被装载。通常情况下，一个多任务系统允许其他进程运行时，运行过程中碰到一个点，它必须等待一些内存部分从二级存储重装。[来源请求]  
程序设计  
  
流程是完全独立的并没有太大麻烦的程序在多任务环境。大多数的多任务系统的复杂性来自于需要任务之间进行同步的协同操作任务操作共享计算机资源。[来源请求]  
  
各种并发计算技术被用于避免由多个任务试图访问相同的资源的潜在问题。[引证需要]  
  
更大的系统有时建有一个中央处理器（S）和I / O处理器的一些数字，是一种非对称多处理的。[来源请求]  
  
多年来，多任务系统进行了改进。现代操作系统一般包括详细的机制，优先处理，而对称多引入了新的复杂性和能力。[来源请求]

在计算机中单CPU实现多任务是靠硬件中断来产生调度任务的，从这点来看，CPU的设计其实在这一点也是事件驱动的。由中断产生事件，然后进行事件处理。

而JavaScript引起并没有中断机制，也无法产生任务调度，那么如何实现一个单线程的多任务系统呢？其实在JavaScript引擎中还有另一个类似能够产生事件调度的机制——消息队列。我们将任务细分成一个不包含任何子任务的基本任务单元，然后将这些基本任务单元逐步压入消息队列里面，然后再通过事件循环处理消息队列里面的每一个任务。

因为事件队列的是先入先出的，所以任务的调度机制只能遵循最基本的先入先出机制。但是不同的是，只有当异步事件产生的时候，才会将调度事件压入消息队列。最终依靠JavaScript引擎的消息队列进行任务的调度。

模块：

1. Class模块

在JavaScript中由于其基于原型的面向对象机制的特殊性，其面向对象编程并不如其他语言方便。所以一般需要有一个库来处理面向对象的基本操作，比如最基础的继承。这里将作为整个面向对象设计的基础。

1. EventEmiter模块

EventEmiter是一个事件监听模块，可以用于自定义事件、监听事件、触发事件，并且所有的任务都将会继承自EventEmiter。因为任务的调度深度依赖事件，所有任务都是驱动于事件，所以继承至事件对象将会更为方便优雅地将任务之间关联起来，更好地完成整个控制流的设计。

1. Task模块

Task 模块是对任务的封装。一个任务可能是单独的一个任务单元，也可能是一个拥有众多子任务的任务序列，但是他们都会被看成一个整个任务对象。在一个的任务中，任务有权知道自己位于整个任务序列的何处，也有对整个任务序列的走向进行操作，比如循环，回退上一个任务，跳过下一个任务等。任务也有能里对其子任务的执行进行调整，使得子任务能够产生分支任务或者栅栏任务。子任务之间的通讯是依靠管道设计的进行的，上一个任务的输出将成为下一个任务的输入。

1. MVVM前端WEB应用框架的设计

我们这里的MVVM前端WEB应用框架将会使用上面所有的JavaScript单线程下的多任务系统进行开发和设计。在WEB前端的多任务设计上面进行一次尝试和探索。

1. MVVM层次结构的设计

MVVM分成三个层次，Model 层、View层以及View-Model层。整个框架将会按照这个层次进行设计。

1. 模型层（Model）

模型层可能包括两种模型：数据模型和业务模型。但是无论哪一个模型都必须继承自Model对象。由于一般JavaScript很少在浏览器端存取数据，更多可能是通过AJAX来取位于远端服务器的数据。当然也有部分数据会存储在本地，不过这一切都能够通过模型层来解藕合。对于其他层次，这一切都是不可见的。

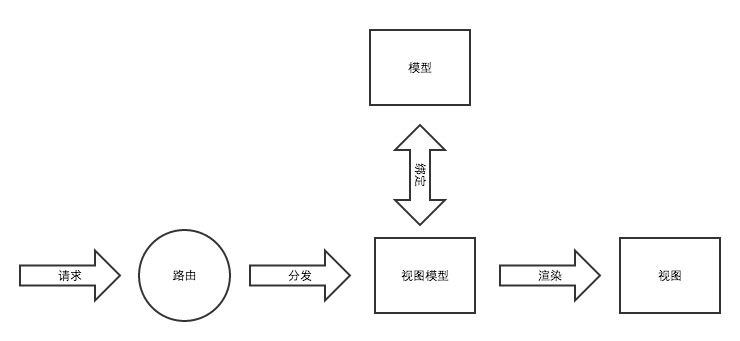
1. 视图层（View）

视图层将模型的数据通过渲染并且表现出来，往往视图层不会参与数据逻辑上的任务。只需要将模型上的数据通过视图模型层来和模型建立对应的关系。当模型改变的时候，相应的数据将会发生更改。通过数据绑定技术，实现视图层的局部更新。

1. 视图模型层（View-Model）

视图模型层可以说是一个经过巧妙设计的控制层，但是不同于控制层的是。往往是控制层是在一次性在一个请求中将从模型中取得数据并且渲染视图。在这个过程完成后，模型的更改将不会动态的映射到视图层，只能通过“刷新”的步骤再次执行动作。而视图模型层而是动过模型的驱动，将模型的数据绑定在视图层上，当模型层的数据改变的时候，视图模型层将会把相应的更改隐射到对应的视图层相应位置而不需要重新渲染模板。

1. WEB框架的流程设计



第一步、监听请求

当接到一个URL（通过hash或者history在HTML5标准中新的API更改URL但不刷新页面）请求后，在处理完之前关闭View层的事件触发。并且等待整个请求执行完成后等待下一次请求。

第二步、通过路由分发请求

当接受到URL请求的时候，这时候通过一个路由来将请求分配给相应的视图模型。

第三部、视图模型从绑定的模型获取数据

第四部、视图模型将渲染视图，并将模型上的数据绑定在视图上。并且监听模型的更改，如果是模型更改，视图模型将会把更改后的数据同步到视图层上。

# 详细设计

1. JavaScript下单线程多任务机制的设计

1. Class模块的实现

Class主要实现JavaScript面向对象的继承，封装从一个对象集成到另一个对象的操作。然后JavaScript的继承有非常多种类，但是大体上分成两个三个类别：原型继承、对象扩展以及混合。

这里我们选择被推荐的，也是比较正统的继承方式：原型链继承。并且为了能够更方便的使用父原型的对像的原型方法，所以我们将会给对象一个父原型的指针。

2.EventEmitter模块实现

EventEmitter主要有三个功能：1，绑定/解绑事件。2，监听事件。3，触发事件。

EventEmitter遵从发布者/订阅者模式。一个EventEmitetter对象会维护一个哈希表（在JavaScript中，一个哈希表就是一个JavaScript原生对象）。每个事件都对应着一个监听队列。当一个EventEmitter对象被绑定上一个为定义的事件的时候，会在哈希表上创建一个以事件命名的键，并且将事件监听者压入监听者列表中。当此事件被触发的时候，则会遍历整个监听者列表。

3. Task模块设计

Task模块是整个系统的核心。

首先，Task继承至EventEmitter。所以Task也是一个事件对象。Task所拥有的基本事件是：taskstart 和 taskend。这两个事件分别代表了这个在任务执行开始时，以及事件结束。

Task对象也分成两种，一种是不可再分的基本任务单元对象，基本任务单元对象维护一个具体任务的执行函数。而另一种是拥有多个子任集合的任务对象，它只维护一个任务的序列。

Task任务顺序流程：

首先创建一个任务对象A，然后调用任务对象A的run()方法。若此对象A维护的是一个具体的任务，则执行这个具体的任务，然后触发taskend事件。若此任务A维护的是若干个子任务的序列，则取第一个任务B，调用run()方法，然后绑定这个子任务对象B的 taskend 事件。对象B触发B的taskend事件后，任务对象A取得任务对象B的返回结果，然后将返回结果作为参数传给下一个任务对象C，并执行任务C的run(任务对象B返回的结果) 方法。

Task任务Fork流程：

若Fork一个基本任务单元对象，则直接返回一个拥有相同具体任务（相同函数）的任务对象副本。若Fork一个基本对象序列则会递归fork。但当出现非控制内允许的任务循环是，则会出现问题。

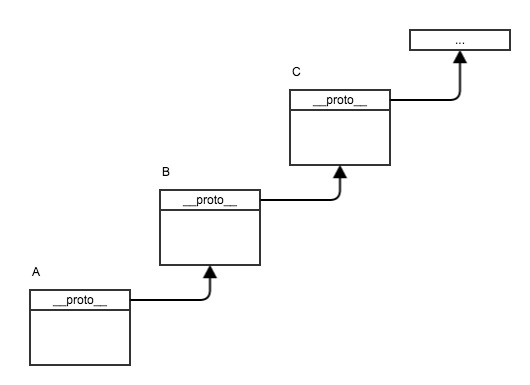
2.WEB框架的实现

流程实现

# 系统实现

1 JavaScript单线程多任务实现。

1. Class模块继承的实现。们选用JavaScript原生的原型链来实现。



最后暴露出来的接口是 BlxClass(); BlxClass()有一个可选参数，若没有参数则继承自Object对象。所有的用BlxClass()继承出来的原型对象全部都以Object为根。

1. EventEmitter实现

EventEmitter原型的属性以及方法：

属性：

events: type Object

{ ‘event1’ : [listener list], ‘event2’: [listener list], … }

evnets维护了一个事件列表，以及每一个事件所有的Listener

方法：

addListener(String event, Function listener)

给事件对象的一个事件添加一个监听器。

removeListener(String event, Function listener)

删除事件对象的一个事件指定的监听器。

on() alias to addListener()

off() alias to removeListener()

one(String event, Function listener)

给事件对象的一个事件添加一个监听器。当事件触发后删除此监听器。

tigger(String event, <arguments…> )

触发一个事件。并且遍历事件的所有监听器。

emit() alias to tigger()

var EventEmitter = function EventEmitter(){};

1. Task实现
2. Task对象

拥有的事件：

start：当任务开始执行是产生start事件。

end: 当任务执行结束并且时产生end事件，需要手动调用end方法。

error: 当任务执行中抛出异常的时候，触发error事件。

timeout: 当一个任务等待超时的时候产生此事件。

属性：

status: type Int

当前任务状态。分别是 正在运行，停止，超时，错误。四个状态。

基本任务单元属性：

task: type Function

具体的执行函数

任务序列属性：

pc: type Int

保存当前执行的位置

task: type Array

保存一个任务序列

私有方法：

\_\_next\_\_()

执行下一个任务。下一个任务可以是当前

共有方法：

run()

fork()

1. 任务实现

函数的具体实现 function(req, next){ …; next(req) }

1. 任务序列实现

status: type Int

当前任务状态。分别是 正在运行，停止，超时，错误。四个状态。

基本任务单元属性：

task: type Function

具体的执行函数

任务序列属性：

pc: type Int

保存当前执行的位置

task: type Array

保存一个任务序列

私有方法：

\_\_next\_\_()

执行下一个任务。下一个任务可以是当前

共有方法：

run()

fork()

subtask()

branch()

barrier()

loop()

mvvm框架实现

框架主流成实现：

Task()

.loop(

Task()

.subtask(watiForHashChanged) // 等待url中hash的变化

.subtask(Router) // 分析url中hash所代表的路由

.subtask(ModelViewTask) // 指定相应的ModelView

.subtask(RenderView) // 渲染View并且绑定Model

)

.run()

waitForHashChanged实现

var waitForHashChanged =Task( function(req, next){

$(window).one(‘hashchange’, function(){

next(location.href);

});

});

Router Task实现：

路由分成三个部分 /<ModelView>/<Action>/<Arguments>

第一个部分是 ModelView

第二个部分是Action

第三个部分是参数Agruments

Agruments是可选的，并且可能由若干个argument组成并且以 ‘ / ‘ 字符作为分隔。

例如一个路由 /post/index/，那么ModelView = post，Action = index，Agruments = null

例如一个路由 /post/show/1/w，那么ModelView = post, Action = show，Agruments = [‘1’, ‘w’]

var Router = Task(function(req, next){

var parsed = \_.words(req, ‘\/’);

var result = {

ModelView: parsed[0] || DefaultModelView,

Action: parsed[1] || ‘index’,

Arguments: parsed.slice(2)

};

next(result);

});

ModelViewTask实现

ModelView = Task(function(req, next){

var modelview = ModeView[req.ModelView];

var action = modelview[req.Action];

// action.apply(modelview, req.Arguments);

Task()

.on(‘end’, function(){next()})

run(req.Arguments);

});

RenderView实现

RenderView = Task(function(req, next){

});

# 系统测试

# 总结