# 绪论

1.1 研究背景与意义

随着云计算、物联网、移动互联、社交媒体等新兴信息技术和应用模式的快速发展，推动人类社会不断地向大数据时代迈进。2010年，全球数据量就已经跨入了ZB时代，据IDC预测，至2020年全球将拥有35ZB的数据量，大量数据实时地影响我们的工作、生活，甚至国家经济、社会发展，大数据时代已经到来。大数据具有数据量巨大、数据类型多样、流动速度快和价值密度低的特点，大数据技术为我们分析问题和解决问题提供了新的方法和思路，其研究已经成为热点。一般意义上来讲，对大数据就是利用现有的理论知识以及各种技术手段和工具都很难再可以接受的时间之类完成数据分析计算，而整体都呈现价值巨大的海量复杂的数据集合体。与此同时，这些巨大的价值量往往都是隐藏在海量的数据当中，变现出了价值密度低、分布极其不规律、信息隐藏深度极深、发现价值极其困难等特点，这些特征必然会为大数据的研究和分析带来前所未有的挑战和机遇。

大数据的分析计算模式主要分为批量计算（batch computing）、流式计算（stream computing）、交互式计算（interactive computing）、图形计算（graph computing）等等。其中批量计算和流式计算这两种计算模式不管是在学术界还是在工业界都是主要的研究模式，同时各自都有广泛的大数据应用场景。其中批量计算是一种适用于大估摸并行批量处理作业的分布式计算模式，也就是我们大家都十分熟悉的MapReduce计算模式。MapReduce的简单易用性使其成为目前大数据处理最成功的主流并行计算模式。在开源社区的努力下，开源的Hadoop系统目前已成为较为成熟的大数据处理平台，并已发展成一个包括众多数据处理工具和环境的完整的生态系统。目前几乎国内外的各个著名IT企业都在使用Hadoop平台进行企业内大数据的计算处理。此外，Spark系统也具备批处理计算的能力。而流式计算是一种高实时性的计算模式，需要对一定时间窗口内应用系统产生的新数据完成实时的计算处理，避免造成数据堆积和丢失。很多行业的大数据应用，如电信、电力、道路监控等行业应用以及互联网行业的访问日志处理，都同时具有高流量的流式数据和大量积累的历史数据，因而在提供批处理计算模式的同时，系统还需要能具备高实时性的流式计算能力。流式计算的一个特点是数据运动、运算不动，不同的运算节点常常绑定在不同的服务器上。Facebook的Scribe和Apache的Flume都提供了一定的机制来构建日志数据处理流图。而更为通用的流式计算系统是Twitter公司的Storm、Yahoo公司的S4以及Apache Spark Steaming。

大数据时代下的流式计算呈现出了鲜明的高带宽、低时延的应用需求，传统的流式计算平台的构造往往是建立在传统的数据库的基础之上的，难以满足大数据流式计算的需求。如何构建一个低时延、高带宽、持续可靠、长期运行的大数据流式计算系统成为了当前亟待解决的问题。Redis这种基于内存计算的、可进行数据持久化的Key-Value存储系统的诞生，为大数据流式计算带来了很好的一个解决方案。Redis数据库最初是为了解决像SNS类网站在数据存取过程中的实时性等刚性需求的，而传统的关系型数据库越来越难以胜任了，这也使得redis这种数据库也越来越受到人们的关注。如今redis数据库已经得到了广泛的应用，不论是在高速缓存系统中，还是在海量文件的实时检索中，甚至是在如何如荼的各种推荐系统中，redis都起着中心地位的作用。Redis的基于内存的数据计算和高效的数据存储策略也能够很好的满足实时流计算问题中的低时延的刚性需求。因此，研究redis的内存计算以及存储策略并将其运用到实时流式计算模型中具有重要的意义和实用价值。

在流式数据处理中，它不像传统的批量数据处理，它无需事先存储数据，无法确定数据是什么时候到来以及按什么顺序到来，因此，不需要事先对流式数据进行存储，而是当流动的数据到来后在内存中直接进行数据的实时计算和分析。就像我们熟悉的Twitter的Storm、Yahoo的S4就是典型的流式数据处理框架，数据在任务拓扑中被计算，最后输出有价值的信息。目前这些流行的流式处理框架都有一个共同的缺点就是，没有一个方便的能够快速根据业务构建数据任务的拓扑计算流程，也就是我们所说的计算流（flow），同时也缺乏数据的流化功能。Node-red是基于Node.js的，可视化流程编辑框架，它允许开发人员仅仅使用一个基于浏览器的可视化界面流程编辑器来完成设备、服务器以及API应用的连接。Node-red本身是IBM Emerging Technology团队创建的一个新型开源工具，它允许用户通过组合各种部件来编写应用程序。这些部件可以是硬件设备、Web API或者是在线服务。Node-red被广泛用于物联网领域，实现数据的流式传输。在node-red中从数据的接入，到数据的解析分析，最后到结果的输出都是通过各种各样的节点来完成的，IBM Emerging Technology团队在开发这个工具的时候只引入了少量的大众化的节点，比如常用的http节点、tcp节点、udp节点、debug等数据输入输出节点，还有一些用于数据分析的节点比如sentiment节点，还有一些用于访问存储设备的节点；node-red除了原始已经提供的这些节点外，还运行用户自己按照开发原则开发自己需要的节点。为了能够充分利用node-red的可视化流程编辑的直观性，结合redis数据库的内存计算的特点，探索开发适应于流式数据分析的数据输入输出和数据处理节点，这对流式数据分析有着重要的实际意义。

1.2 国内外研究现状

1.2.1实时流数据处理模型的研究应用现状

大数据时代下的数据处理主要的两种方式就是实时流式处理和批量处理。实时流数据处理主要适合于那些无需事先进行数据存储，可以直接进行数据分析处理，实时性要求比较严格，但数据的准确度要求比较宽松的应用场景。而对于传统的批量数据处理，首先要进行数据的存储，然后再对存储的静态数据进行集中或者分布式计算。目前，对于传统的批量数据处理模型的技术和研究成果已经相对成熟了，最初有Google公司的MapReduce并行编程模型的提出，再有后来在开源社区的努力下开发的Hadoop系统为代表的批处理系统，都已经是稳定而高效的批处理系统。而对于流式数据处理模型的研究仅仅处于一个初级阶段，在早期关于流式数据的研究也主要集中在以数据库为中心而开展的，主要是研究了数据计算的流式化，数据规模也比较小，数据对象也比较单一，很难适应在大数据时代下流式数据处理所呈现出来的新特性。因为，在新时期的流式数据主要呈现出实时性、突发性、无序性等特点，对新的流式计算系统就有了更高更严格的要求。

在国外，Yahoo推出了S4流式数据处理系统，随后在2011年，Twitter也推出了自己的流式数据处理系统Storm,还有就是近年来开源社区新兴的MOA（Massive Online Analysis）、Spark Stream都是流式处理系统，这在一定程度上推动了流式数据处理的发展和应用。但是像S4、Strom这样的流式数据处理系统在可伸缩性、容错性、数据吞吐量等方面存在着明显的不足，而对于MOA，Spark Stream这样的系统，虽然功能和API十分丰富，但是在稳定性和易用性上不尽如人意。所以，如果构建一个低延迟、高吞吐、易用且能持续可靠地运行的流式数据处理系统，是一个亟待解决的问题。

在国内，目前关于流式数据处理模型的研究还比较少，但目前国内主要有百度公司自主研发的Dstream和TM实时计算平台，在学术界主要是有一下关于流式数据挖掘算法的研究。但是，流式数据的可视化分析已经在很多场景得到了应用，比如各大银行都陆续建立的大屏监控系统，就是实时地监控银行的业务状况、系统运行状况、用户行为分析等，又比如政府网站群的监控，也是通过实时监控网站的访问数据，分析用户的行为。在这些应用的背后，如何建立一个高效、稳定、易于维护的实时处理模型显得尤为重要。

文献[1]，

文献[2]，

1.2.2 Node-red的研究应用现状

Node-red作为一种在物联网时代的新型产物，是一种用来快速搭建物联网应用程序的流式处理框架，在信息无处不在的时代，Node-red也越来越受到业界的关注和研究。

它是由IBM Emerging Technologies团队发起的一个开源项目，其中Nick O’Leary 和Dave Conway-Jones工程师为Node-red的设计和开发做出了巨大的贡献。2013年，Node-red以开源项目的形式被发布，经过一年的发展，Node-red已经拥有了一大批活跃的用户和开发人员。Node-red依然是一个新型科技，时至今日，但凡用过Node-red的制造商、实验人员和一大批大大小小的公司，都已经见证了Node-red极具价值的应用之处。

在国外，IBM公司率先将Node-red应用起来，Node-red被集成到IBM公司的最新云产品Bluemix上。通过Bluemix提供的云服务，用Node-red来建立和管理一个实例（也就是一个应用流程），就可以实现消息的推送服务。Node-RED 的使用，与 Bluemix 中简单的 Push 服务相结合，使整个流程变得非常简单，需要调整的部分也少得多。

在国内，目前也有很多智能设备制造公司在使用Node-red，可以很方便地通过Node-red节点来控制硬件设备的状态，比如拿Node-RED搭配Arduino，是一个快速原型化的好用工具.例如控制RPI的某根脚位去点亮LED。只要简单的拉四个node，串一串再写一点程序代码即可做到。因为Node-red还在进一步完善当中，原始开发的节点可能很难满足实际的需求，所以，我们在运用Node-red来管理数据流程的时候，还需要自己开发需要的功能节点。在这一点上，目前在不少银行的业务监控系统中引入了redis的访问节点。

1.2.3 Redis的研究应用现状

Redis作为存储系统之中的后起之秀，由于其数据结构丰富、基于内存计算、支持网络又可进行数据持久化等特点，迅速为许多企业和开发者所爱戴。不论是在学术界还是在工业界，对Redis的研究都从未停止过。

Redis是由Salvatore Sanfilippo为实时统计系统LLOOGG量身定制的一个数据库，在2009年的时候将Redis开源发布，并开始于另外一位Redis代码贡献者Pieter Noordhuis一起继续Redis的开发，知道现在。随着Redis内存数据库的发布，短短几年内的时间就拥有了一个庞大的用户群体。在国外，像GitHub、Viacom、Pinterest等都是Redis的用户，Github利用Redis集群，来统计用户项目跟进状况。而在国内，新浪微博研究了Redis数据库的源码，搭建了有号称史上最大的Redis集群，实现了传统的SQL数据库难以实现的计数分析（counting）、反向缓存（reverse cache）、top 10 list等功能。近年来，也有不少银行，在自己的实时数据监控平台引入了Redis数据库，实现了数据的实时处理和分析，还有就是随着国家电子政务系统的逐渐推行，不少的地方政府也在自己的数据中心监控系统中引入了Redis数据库，来实现数据实时计算和处理。

文献[3]，

1.3 主要工作和研究内容

本文对大数据背景下流式数据处理过程中所遇到的挑战和难题进行了研究分析，详细研究了Node-red流式处理框架的编程模型和消息推送机制，Redis数据库的存储原理及其基于内存计算的原理。设计了一种新的基于Node-red的流式管理和Redis的内存计算的流式数据处理模型，并通过实现网站访问实时监控系统来验证了该模型的可行性。主要工作内容如下：

1. 本文对当下流行的几款流式处理框架Storm、Spark stream与Node-red流式管理框架进行了对比分析，详细阐述了各自的编程模型，同时对结合MQTT详细阐述了消息推送机制为什么适合流式数据处理。
2. 对Redis数据库做了深入研究，尤其是Redis的存储原理以及内存计算原理，还对Redis的Pub/Sub机制做了深入研究，阐述了在流式数据处理中为什么选择Redis作为数据计算和中间数据缓存的工具。
3. 在研究分析了流式数据的特点和流式数据处理的基本原理后，结合Node-red的编程模型和消息推送机制，设计了一种新的基于Node-red的流式管理和Redis的内存计算的流式数据处理模型。由于原始的Node-red缺乏对redis数据库的访问节点以及redis的pub/sub节点，重新设计了新的数据输入、输出节点以及数据处理节点，并成功发布到Node-red框架当中，实现数据的流式处理，和数据流的管理。
4. 本文最后还对设计好的流式处理模型，加以了应用验证。使用该模型对某政府网站的访问数据进行实时监控分析，设计了一套数据监控系统，包括了数据的采集，数据分析和处理，以及最后的数据可视化展示，并对结果进行了有效性分析。实现了从模型设计到模型应用的全过程。

1.4 结构安排

本论文共分为六章，其章节结构安排如下：

1. ，绪论，首先介绍了本论文的研究背景和意义，通过阅读大量相关文献和论文资料，总结了国内外流式数据处理模型的研究现状，以及Node-red、Redis的研究应用现状。然后简单的介绍了本论文的主要研究内容和全文的章节结构安排。
2. ，实时流数据处理的理论基础和技术，本章详细介绍了目前几个主流的流式处理框架的编程模型，有Storm的流式编程模型、Spark stream的流式编程模型、Node-red的编程模型以及他们各自的特点和不足之处。还介绍了MQTT协议以及在消息推送服务中的应用。最后还介绍了Node-red的异步非阻塞模式与事件驱动机制，以及Redis的存储技术和Pub/Sub机制。
3. ，基于Node-red和Redis的实时流数据处理模型的设计，本章首先对在实际场景中的流式数据处理应用做了详尽的需求分析，然后对模型的总体架构做了详细设计，最后对于Node-red中原本缺少的用于流式数据的输入和输出节点以及数据处理节点做了重新设计，并将设计的各个节点重新部署到Node-red框架当中，使其成为一个能够胜任流式数据处理的完整框架。
4. ，基于node-red与redis的实时流数据处理模型在网站访问监控方面的应用，本章主要是对设计的新模型加以应用，以此来验证模型的可行性。为此设计了一个实时网站访问的监控系统，数据处理就用到了我们设计好的流式数据处理模型，将数据处理的结果输出到前端页面Board做可视化展示。本章详细阐述了系统的功能，各个模块的设计与实现以及系统的展示。
5. ，系统测试与性能分析，这一章是整个模型的测试环节，主要是分析了模型对流式数据的处理能力，并与Storm和Spark stream流式处理系统在吞吐量和时延上做了对比分析。
6. ，全文总结与展望，是对本论文的主要工作进行最后总结，并对后续工作做了一些说明。

# 实时流数据处理的基础理论和技术

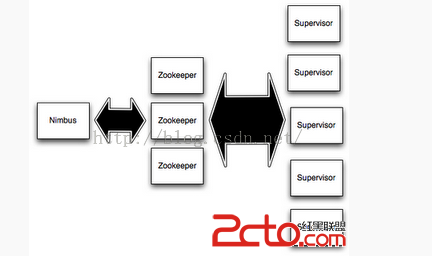
2.1 实时流数据处理的编程模型

2.1.1 Storm的编程模型

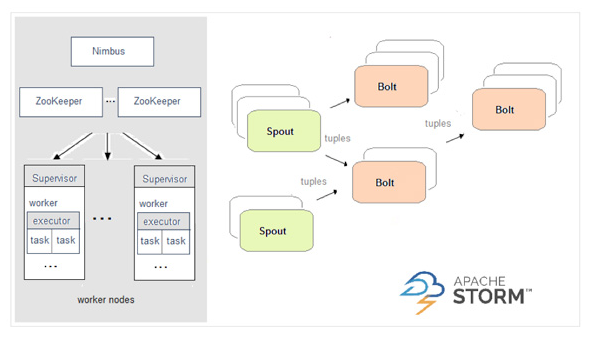
Storm是一个分布式的，可靠的，容错的数据流处理系统。它会把工作任务委托给不同类型的组件，每个组件负责处理一项简单特定的任务。Storm集群的输入流由一个被称作spout的组件管理，spout把数据传递给bolt， bolt要么把数据保存到某种存储器，要么把数据传递给其它的bolt。你可以想象一下，一个Storm集群就是在一连串的bolt之间转换spout传过来的数据。

在Storm编程框架中，只要不是人为干预，Storm就一直实时不断地进行数据处理。值得注意的是：并不是Storm去处理，而是它可以将我们程序的很多jar包，业务程序，同时放到不同的服务器中并发的运行，最终得到的结果就是不同系统的海量数据就会分散到不同的服务器中并发的进行处理，负载能力很强。所以真正进行数据处理的是我们写好的数据处理程序，Storm的强大作用之一就是它为这些程序提供了运行温床，将应用程序上传到Storm集群中，在多台机器上并发运行，这样就可以扩展程序的负载处理能力，实现流式计算。

接下来我们看一下Storm的集群组件，Storm集群主要有两个角色扮演者，一个就是Nibus，是集群的主节点，主要负责任务分配、响应客户端提交topology请求以及任务失败的调度。另外一个就是Supervisor，是集群的从节点，主要负责启动、停止业务逻辑组件程序进程。主从节点之间通过zookeeper集群进行连接，主从节点之间是fail-fast（java的一种错误机制）、无状态的，主从节点的状态信息均保存到zookeeper中或者本地硬盘里。这样的好处就在于，哪怕是主节点kill掉了，storm会自动起一个备份主节点，因为无状态的关系，所以任意一个节点都可以充当Nimbus一角。下图展示了Storm集群的架构，这种设计使得Storm十分稳定。



在[Storm](https://storm.apache.org/" \t "http://www.csdn.net/article/2015-03-09/_blank)中，先要设计一个用于实时计算的图状结构，我们称之为拓扑（topology），这也是Storm处理流式数据计算的核心编程模型。正如下图所示，这个拓扑将会被提交给集群，由集群中的主控节点（master node）分发代码，将任务分配给工作节点（worker node）执行。一个拓扑中包括spout和bolt两种角色，其中spout发送消息，负责将数据流以tuple元组的形式发送出去；而bolt则负责转换这些数据流，在bolt中可以完成计算、过滤等操作，bolt自身也可以随机将数据发送给其他bolt。由spout发射出的tuple是不可变数组，对应着固定的键值对。

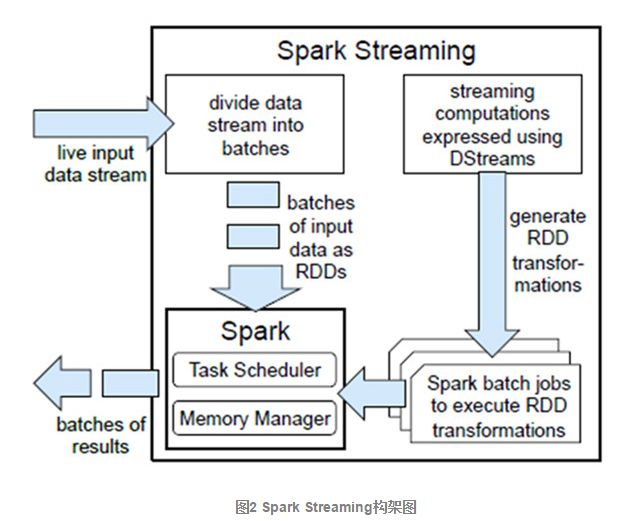


简单理解，Storm中的拓扑结构（topology）就是包含了数据源、逻辑处理组件的一个外在集合框架，使用storm可以定义一个topology里set多少个数据源组件，多少个逻辑处理组件。

2.1.2 Spark streaming的编程模型

Spark是一个类似于MapReduce的分布式计算框架，其核心是弹性分布式数据集，提供了比MapReduce更丰富的模型，可以在快速在内存中对数据集进行多次迭代，以支持复杂的数据挖掘算法和图形计算算法。Spark Streaming是一种构建在Spark上的实时计算框架，它扩展了Spark处理大规模流式数据的能力。

Spark Streaming是将流式计算分解成一系列短小的批处理作业。这里的批处理引擎是Spark，也就是把Spark Streaming的输入数据按照batch size（如1秒）分成一段一段的数据（Discretized Stream），每一段数据都转换成Spark中的RDD（Resilient Distributed Dataset），然后将Spark Streaming中对DStream的Transformation操作变为针对Spark中对RDD的Transformation操作，将RDD经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加，或者存储到外部设备。下图显示了Spark Streaming的整个流程。



使用Spark Streaming编写的程序与编写Spark程序非常相似，在Spark程序中，主要通过操作RDD（Resilient Distributed Datasets弹性分布式数据集）提供的接口，如map、reduce、filter等，实现数据的批处理。而在Spark Streaming中，则通过操作DStream（表示数据流的RDD序列）提供的接口，这些接口和RDD提供的接口类似。图3和图4展示了由Spark Streaming程序到Spark jobs的转换图。





在图3中，Spark Streaming把程序中对DStream的操作转换为DStream Graph，图4中，对于每个时间片，DStream Graph都会产生一个RDD Graph；针对每个输出操作（如print、foreach等），Spark Streaming都会创建一个Spark action；对于每个Spark action，Spark Streaming都会产生一个相应的Spark job，并交给JobManager。JobManager中维护着一个Jobs队列, Spark job存储在这个队列中，JobManager把Spark job提交给Spark Scheduler，Spark Scheduler负责调度Task到相应的Spark Executor上执行。

Spark Streaming的另一大优势在于其容错性，RDD会记住创建自己的操作，每一批输入数据都会在内存中备份，如果由于某个结点故障导致该结点上的数据丢失，这时可以通过备份的数据在其它结点上重算得到最终的结果。

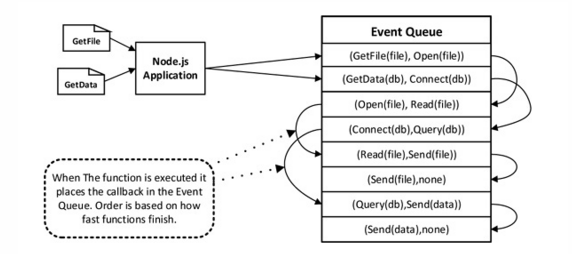
正如Spark Streaming最初的目标一样，它通过丰富的API和基于内存的高速计算引擎让用户可以结合流式处理，批处理和交互查询等应用。因此Spark Streaming适合一些需要历史数据和实时数据结合分析的应用场合。当然，对于实时性要求不是特别高的应用也能完全胜任。另外通过RDD的数据重用机制可以得到更高效的容错处理。

2.1.3 Node-red的编程模型

Node-Red是IBM公司开发的一个可视化的编程工具。它允许程序员通过组合各部件来编写应用程序。这些部件可以是硬件设备(如：Arduino板子)、Web API(如：WebSocket in和WebSocket out)、功能函数(如：range)或者在线服务(如：twitter)。Node-Red提供基于网页的编程环境。通过拖拽已定义node到工作区并用线连接node创建数据流来实现编程。程序员通过点击‘Deploy’按钮实现一键保存并执行。程序以JSON字符串的格式保存，方便用户分享、修改。Node-Red基于Node.js，它的执行模型和Node.js一样，也是事件驱动非阻塞的。理论上，Node.js的所有模块都可以被封装成Node-Red的一个或几个node。

2.2 Node的事件驱动和非阻塞机制

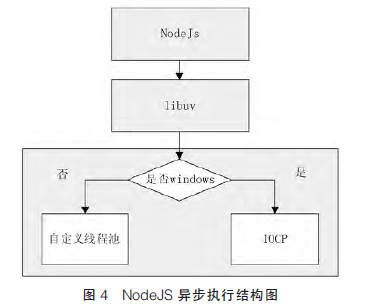
与Nginx的服务原理类似，Node采用事件驱动的运行方式。不过nginx式多进程单线程，而Node通过事件驱动的方式处理请求时无需为每一个请求创建额外的线程。在事件驱动的模型当中，每一个IO工作被添加到事件队列中，线程循环地处理队列上的工作任务，当执行过程中遇到来堵塞(读取文件、查询数据库)时，线程不会停下来等待结果，而是留下一个处理结果的回调函数，转而继续执行队列中的下一个任务。这个传递到队列中的回调函数在堵塞任务运行结束后才被线程调用。



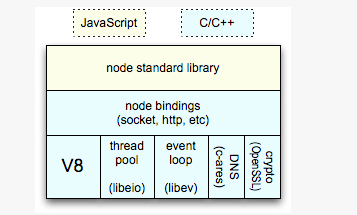
Node Async IO = CPS + Callback，这一套实现开始于Node开始启动的进程，在这个进程中Node会创建一个循环，每次循环运行就是一个Tick周期，每个Tick周期中会从事件队列查看是否有事件需要处理，如果有就取出事件并执行相关的回调函数。事件队列事件全部执行完毕，node应用就会终止。Node对于堵塞IO的处理在幕后使用线程池来确保工作的执行。Node从池中取得一个线程来执行复杂任务，而不占用主循环线程。这样就防止堵塞IO占用空闲资源。当堵塞任务执行完毕通过添加到事件队列中的回调函数来处理接下来的工作。

我们知道，IO操作无疑是耗时的，当服务器端接收到大量请求时，为每一个请求创建进程或线程的同时，也增加了额外的内存开销，也可能浪费更多的时间资源。由于Node.js是事件驱动的，于是它使用了事件循环来解决IO操作带来的瓶颈问题。在Node.js中，一个IO操作通常会带有一个回调函数，当IO操作完成并返回时，就会调用这个回调函数，而主线程则继续执行接下来的代码。下面我们说明一下Node.js异步非阻塞I/O的运行原理。

在Windows平台上，Node.js是直接利用Windows下的IOCP（I/O Completion Port）通常称为I/O完成端口来实现的，在IOCP的内部其实是利用了线程池的原理，这些线程是由Windows系统内核自动管理，不需要我们手动加以管理。而在Linux平台上，Node.js从v0.9.3版本以后，都是通过自行实现的线程池来完成异步非阻塞I/O的。由于这种跨平台的差异性，Node.js通过构建一个平台层架构libuv，来屏蔽平台的差异性，所有平台的兼容性问题都是由这一层来完成。下图就是Node.js的异步执行架构图。



以上就异步非塞模型和事件驱动机制进行了总结。而这个事件循环的机制并不仅仅是Node.js所独有的，并且Node.js的代码是单线程执行的，在面对大量并发请求的时候，Node.js是有着自己独特优势的，下图我们给出了Node.js的架构图。



从该架构图中可以看出，Node.js的底层有一个模块负责维护线程池，当一个IO请求发出的时候，Node.js的底层模块将新建一个线程来处理请求，完成后再将结果交还给上层。那么，当有多个请求的时候，Node.js的底层模块将利用尽可能少的线程来完成最多的任务，如果存在空闲的线程，它将继续被利用来做其他的事情，这对于前面说的针对每个请求开一个新的进程或线程而言，无疑“聪明”许多，也更加高效了。

2.3 Node-red可视化流式处理框架

2.3.1 Node-red的概述

Node-Red是IBM公司开发的一个可视化的编程工具。它允许程序员通过组合各部件来编写应用程序。这些部件可以是硬件设备(如：Arduino板子)、Web API(如：WebSocket in和WebSocket out)、功能函数(如：range)或者在线服务(如：twitter)。Node-Red提供基于网页的编程环境。通过拖拽已定义node到工作区并用线连接node创建数据流来实现编程。程序员通过点击‘Deploy’按钮实现一键保存并执行。程序以JSON字符串的格式保存，方便用户分享、修改。Node-Red基于Node.js，它的执行模型和Node.js一样，也是事件驱动非阻塞的。理论上，Node.js的所有模块都可以被封装成Node-Red的一个或几个node。

2.3.2 Node-red的非阻塞模式与事件驱动机制

2.4 基于内存计算的数据库Redis

2.4.1 Redis数据库的概述

redis是一个key-value[存储系统](http://baike.baidu.com/view/51839.htm" \t "http://baike.baidu.com/_blank)。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list([链表](http://baike.baidu.com/view/549479.htm" \t "http://baike.baidu.com/_blank))、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。这些[数据类型](http://baike.baidu.com/view/675645.htm" \t "http://baike.baidu.com/_blank)都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis支持各种不同方式的排序。与memcached一样，为了保证效率，数据都是缓存在内存中。区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。

与其他key-value存储系统相比，Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，因为数据量不能大于硬件内存。在内存数据库方面的另一个优点是，相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

2.4.4 Redis数据库的储存原理

Redis存储机制分成两种Snapshot 和 AOF。无论是那种机制,Redis都是将数据存储在内存中。

Snapshot工作原理: 是将数据先存储在内存，然后当数据累计达到某些设定的伐值的时候，就会触发一次DUMP操作，将变化的数据一次性写入数据文件（RDB文件）。

AOF 工作原理: 是将数据也是先存在内存，但是在存储的时候会使用调用fsync来完成对本次写操作的日志记录，这个日志揭露文件其实是一个基于Redis网络交互协议的文本文件。AOF调用fsync也不是说全部都是无阻塞的，在某些系统上可能出现fsync阻塞进程的情况，对于这种情况可以通过配置修改，但默认情况不要修改。AOF最关键的配置就是关于调用fsync追加日志文件的平率，有两种预设频率，每次记录进来都添加，每秒添加一次。

这两种存储策略各有优点，从性能上讲，Snapshot方式的性能明显要高于AOF方式，因为前者是采用2进制方式存储数据，数据文件比较小，加载快速，存储的时候是按照配置中的save策略来存储，每次都是聚合很多数据批量存储，写入的效率很高，而AOF则一般都是工作在实时存储或者准实时模式下。相对来说存储的频率高，效率却偏低。但是从数据安全性上来讲，AOF数据安全性高于Snapshot存储，因为Snapshot存储是基于累计批量的思想，也就是说在允许的情况下，累计的数据越多那么写入效率也就越高，但数据的累计是靠时间的积累完成的，那么如果在长时间数据不写入RDB，但Redis又遇到了崩溃，那么没有写入的数据就无法恢复了，但是AOF方式偏偏相反，根据AOF配置的存储频率的策略可以做到最少的数据丢失和较高的数据恢复能力。

2.4.5 Redis数据库的pub与sub机制

pub/sub功能也就是publish/subscribe功能，也就是发布订阅功能。基于事件的[系统](http://www.2cto.com/os/" \t "http://www.2cto.com/database/201502/_blank)中，pub/sub是目前广泛使用的通信模型，它采用事件作为基本的通信机制，提供大规模系统所要求的松散耦合的交互模式：订阅者比如客户端以事件订阅的方式表达出它有兴趣接收的一个事件或一类事件，发布者比如服务器可以将订阅者兴趣的事件随时通知相关订阅者。

Redis数据库也支持pub/sub机制，本论文所设计的流式数据处理模型中，新引入了数据的输入节点也就是redis的subscribe节点，以及数据的输出节点publish节点，将这两个节点在采集数据的时候用。订阅者可以订阅多个Channal，而发布者可以通过Channel，向订阅者发送消息。但是发布者所发的消息是异步的，不需要等待订阅者订阅，也不关心订阅者是否订阅，简单地说就是订阅者只能收到发布者后续所发送到该Channel上的消息，如果之前发送的消息没有接收，那么也再也接收不到了，下面是Redis数据库的发布订阅命令。

IMG_256

PUBLISH：向channel\_1发布消息Hello。

SUBSCRIBE：订阅channel\_1消息，会收到发布者所发送的Hello消息。

2.5 本章总结

本章主要是对本论文的相关技术进行了介绍，首先详细介绍了当前流行的几种流式数据处理框架的编程模型和一些基本概念。然后对Node的事件驱动和非阻塞机制进行了详细阐述，主要是为后面Node-red的节点设计打下理论基础，再是对Node-red进行了详细介绍，最后还详细介绍了Redis数据库的基本概念、存储原理以及发布/订阅消息的模型。

# 基于Node-red与Redis的实时流数据处理模型的设计

3.1 需求分析

本论文主要是研究和设计一种基于Node-red与Redis的实时流数据处理模型，应用场景为实际项目中的网站群的实时访问监控。在本项目中旨在实时了解用户访问网站群的行为。捕捉用户请求并跟踪其通过所以响应，收集、处理并显示用户行为的细节数据，并可视化展示数据和挖掘数据背后的信息。针对该流式计算模型在该实际应用场景下提出如下的需求。

（1）高可扩展性；数据采集点可扩展:数据采集部分能够根据采集数据量的要求，通过添加服务器和配置的方式，无需编程实现水平扩展。数据存储可扩展：数据存储部分能够根据数据量的要求，通过添加服务器和配置的方式，无需编程实现水平扩展。分析/处理部分可扩展：数据分析部分能够根据应用的要求，通过添加服务器和配置的方式，无需编程实现水平扩展。

（2）高性能；支持流数据采集：能够实时采集业务日志、流水和各种软件运行状况的数据；业务数据采集速度不低于50笔/秒，每种软件运行状况数据不低于1笔/秒。细节数据存储：存储每笔业务活动记录（包括查询）。实时动态展示：业务统计数据实时变化延时不超过1分钟，软件系统统计数据变化延时不超过1分钟。

（3）高可用；系统可以通过集群等方式实现分布式部署，避免单点故障。

（4）安全性；数据安全是任何系统的一个首要前提，流式数据处理模型也必须要保证数据的安全性。

本论文要在这些需求的基础之上，要在Node-red上设计出高效的数据接入和输出节点，同时也要有高效的数据处理计算的节点，并充分利用Redis的pub/sub机制来实现数据的流式异步传输。并最终将这套模型应用到实际系统中去加以验证。

3.2 模型的架构设计

3.3 节点处理模块的设计

3.3.1 数据输入节点的设计

3.3.2 数据输出节点的设计

3.3.3 数据处理节点的设计

3.3.4 数据库访问节点的设计

3.3.5 redis数据库的发布订阅节点的设计

3.4 节点的重新部署

3.5 本章总结