**正向代理的概念**

正向代理,也就是传说中的代理,他的工作原理就像一个跳板,  
简单的说,  
我是一个用户,我访问不了某网站,但是我能访问一个代理服务器  
这个代理服务器呢,他能访问那个我不能访问的网站  
于是我先连上代理服务器,告诉他我需要那个无法访问网站的内容  
代理服务器去取回来,然后返回给我

从网站的角度,只在代理服务器来取内容的时候有一次记录  
有时候并不知道是用户的请求,也隐藏了用户的资料,这取决于代理告不告诉网站

结论就是 正向代理 是一个位于客户端和原始服务器(origin server)之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标(原始服务器)，然后代理向原始服务器转交请求并将获得的内容返回给客户端。客户端必须要进行一些特别的设置才能使用正向代理。

**反向代理的概念**

继续举例:  
例用户访问 http://ooxx.me/readme  
但ooxx.me上并不存在readme页面  
他是偷偷从另外一台服务器上取回来,然后作为自己的内容吐给用户

但用户并不知情  
这很正常,用户一般都很笨

这里所提到的 ooxx.me 这个域名对应的服务器就设置了反向代理功能

结论就是 反向代理正好相反，对于客户端而言它就像是原始服务器，并且客户端不需要进行任何特别的设置。客户端向反向代理 的命名空间(name-space)中的内容发送普通请求，接着反向代理将判断向何处(原始服务器)转交请求，并将获得的内容返回给客户端，就像这些内容 原本就是它自己的一样。

**两者区别**

从**用途** 上来讲：

正向代理的典型用途是为在防火墙内的局域网客户端提供访问Internet的途径。正向代理还可以使用缓冲特性减少网络使用率。反向代理的典型用途是将 防火墙后面的服务器提供给Internet用户访问。反向代理还可以为后端的多台服务器提供负载平衡，或为后端较慢的服务器提供缓冲服务。

另外，反向代理还可以启用高级URL策略和管理技术，从而使处于不同web服务器系统的web页面同时存在于同一个URL空间下。

从**安全性** 来讲：

正向代理允许客户端通过它访问任意网站并且隐藏客户端自身，因此你必须采取安全措施以确保仅为经过授权的客户端提供服务。

反向代理对外都是透明的，访问者并不知道自己访问的是一个代理。

**Web 容器**

WEB 容器更多的是跟基于 HTTP 的请求打交道。而 EJB 容器不是。它是更多的跟[**数据库**](http://lib.csdn.net/base/mysql)、其它服务打交道。

      容器的行为是 将其内部的应用程序组件与外界的通信协议交互进行了隔离，从而减轻内部应用程序组件的负担（实现方面的负担？）。  
例如，SERVLET 不用关心 HTTP 的细节，而是直接引用环境变量 session、request、response 就行、EJB 不用关心数据库连接速度、各种事务控制，直接由容器来完成。

Web服务器】

      Web 服务器（Web Server）可以处理 HTTP 协议。当 Web 服务器接收到一个 HTTP 请求，会返回一个 HTTP 响应，例如送回一个 HTML 页面。

      Web 服务器可以响应针对静态页面或图片的请求， 进行页面跳转（redirect），或者把动态响应（dynamic response）的产生委托（delegate）给一些其它的程序，例如 CGI 脚本，JSP（JavaServer Pages）脚本，servlets，ASP（Active Server Pages）脚本，服务器端 [**JavaScript**](http://lib.csdn.net/base/javascript)，或者一些其它的服务器端技术。

Web 服务器仅仅提供一个可以执行服务器端程序和返回(程序所产生的)响应的环境，而不会超出职能范围。  
  
Web 服务器主要是处理需要向浏览器发送 HTML 的请求以供浏览。

【应用程序服务器（The Application Server）】

      根据定义，作为应用程序服务器，要求可以通过各种协议（包括 HTTP 协议）把商业逻辑暴露给（expose）客户端应用程序。应用程序使用此商业逻辑就像你调用对象的一个方法或过程（语言中的一个函数）一样。

【serverlet】

      Servlet（Server Applet），全称 [**Java**](http://lib.csdn.net/base/java)Servlet，未有中文译文。是用 [**Java**](http://lib.csdn.net/base/java) 编写的服务器端程序。其主要功能在于交互式地浏览和修改数据，生成动态 Web 内容。狭义的 Servlet 是指 Java 语言实现的一个接口，广义的 Servlet 是指任何实现了这个 Servlet 接口的类，一般情况下，人们将 Servlet 理解为后者。

      Servlet 运行于支持 Java 的应用服务器中。从实现上讲，Servlet 可以响应任何类型的请求，但绝大多数情况下 Servlet 只用来扩展基于 HTTP 协议的 Web 服务器。

【Tomcat】

      Tomcat 服务器是一个免费的开放源代码的 Web 应用服务器，属于轻量级应用服务器，在中小型系统和并发访问用户不是很多的场合下被普遍使用，是开发和调试 JSP 程序的首选。对于一个初学者来说，可以这样认为，当在一台机器上配置好 Apache 服务器，可利用它响应对 HTML 页面的访问请求。实际上 Tomcat 部分是Apache 服务器的扩展，但它是独立运行的，所以当你运行 tomcat 时，它实际上作为一个与 Apache 独立的进程单独运行的。

Apache Tomcat is an open source software implementation of the Java Servlet and JavaServer Pages technologies.

【[Tomcat与Web服务器、应用服务器的关系](http://developer.51cto.com/art/200901/107007.htm)】

      Tomcat 服务器是一个免费的开放源代码的 Web 应用服务器。因为 Tomcat 技术先进、性能稳定且免费，所以深受 Java 爱好者的喜爱并得到了部分软件开发商的认可，成为目前比较流行的 Web 应用服务器。

一、Tomcat 与应用服务器

      到目前为止，Tomcat 一直被认为是 Servlet/JSP API 的执行器，也就所谓的 Servlet 容器。然而，Tomcat并不仅仅如此，它还提供了 JNDI 和 JMX API 的实现机制。尽管如此，Tomcat 仍然还不能算是应用服务器，因为它不提供大多数 J2EE API 的支持。

      很有意思的是，目前许多的应用服务器通常把 Tomcat 作为它们 Servlet 和 JSP API 的容器。由于 Tomcat允许开发者只需通过加入一行致谢，就可以把 Tomcat 嵌入到它们的应用中。遗憾的是，许多商业应用服务器并没有遵守此规则。

      对于开发者来说，如果是为了寻找利用 Servlet、JSP、JNDI 和 JMX 技术来生成 Java Web 应用的话，选择Tomcat 是一个优秀的解决方案；但是为了寻找支持其他的 J2EE API，那么寻找一个应用服务器或者把 Tomcat作为应用服务器的辅助，将是一个不错的解决方案；第三种方式是找到独立的 J2EE API 实现，然后把它们跟Tomcat 结合起来使用。虽然整合会带来相关的问题，但是这种方式是最为有效的。

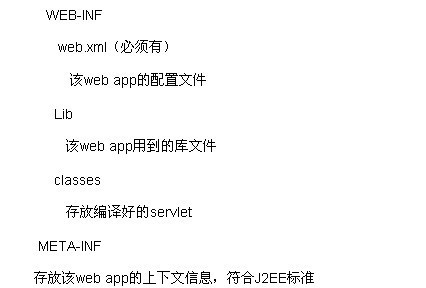
二、Tomcat 与 Web 服务器

      Tomcat 是提供一个支持 Servlet 和 JSP 运行的容器。Servlet 和 JSP 能根据实时需要，产生动态网页内容。而对于 Web 服务器来说， Apache 仅仅支持静态网页，对于支持动态网页就会显得无能为力；Tomcat 则既能为动态网页服务，同时也能为静态网页提供支持。尽管它没有通常的 Web 服务器快、功能也不如 Web 服务器丰富，但是 Tomcat 逐渐为支持静态内容不断扩充。大多数的 Web 服务器都是用底层语言编写如 C，利用了相应平台的特征，因此用纯 Java 编写的 Tomcat 执行速度不可能与它们相提并论。

      一般来说，大的站点都是将 Tomcat 与 Apache 的结合，Apache 负责接受所有来自客户端的 HTTP 请求，然后将 Servlets 和 JSP 的请求转发给 Tomcat 来处理。Tomcat 完成处理后，将响应传回给 Apache，最后 Apache 将响应返回给客户端。

**WEB application**

这里所指的web application是指符合j2ee标准，他就是一个目录，这个目录的结构比较固定，



**JMS:**

**持久化消息**

　　这是ActiveMQ的默认传送模式,此模式保证这些消息只被传送一次和成功使用一次。对于这些消息，可靠性是优先考虑的因素。可靠性的另一个重要方面是确保持久性消息传送至目标后，消息服务在向消费者传送它们之前不会丢失这些消息。

这意味着在持久性消息传送至目标时，消息服务将其放入持久性数据存储。如果消息服务由于某种原因导致失败，它可以恢复此消息并将此消息传送至相应的消费者。虽然这样增加了消息传送的开销，但是却增加了可靠性。

**非持久化消息**

　　保证这些消息最多被传送一次。对于这些消息，可靠性并非主要的考虑因素。此模式并不要求持久性的数据存储，也不保证消息服务由于某种原因导致失败后消息不会丢失。

**JMS消息的顺序**  
   乱序接收，顺序处理。也就是说，消息在发送，传输和接收过程中可能是乱序的，但消费者在接收到消息之后，并不立即处理，而是先将消息排序，然后在处理。 JMS消息头部的 JMSCorrelationID可以帮助我们完成这个工作。JMSCorrelationID存放了另一个消息的id。消息的发送者，如果要保证消息的顺序性，要将后发送的消息的JMSCorrelationID设置成前一个消息的id。消费者接收消息后，如果发现其头部有 JMSCorrelationID，则查看该消息是否已被处理过，如果没有，则等待该消息，至到该消息被处理后，才处理这个消息。这一工作需要发送者和接收者都记住已经发送和接收过的消息，以便于给后来的消息参考。由此可见，在JMS中设置JMS的head还是能起到不少作用的。

**一致性hash**

一致性哈希算法在1997年由麻省理工学院提出的一种分布式哈希（DHT）实现算法，设计目标是为了解决因特网中的热点(Hot spot)问题，初衷和CARP十分类似。一致性哈希修正了CARP使用的简 单哈希算法带来的问题，使得分布式哈希（DHT）可以在P2P环境中真正得到应用。

    一致性hash算法提出了在动态变化的Cache环境中，判定哈希算法好坏的四个定义：

1、平衡性(Balance)：平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用。很多哈希算法都能够满足这一条件。

2、单调性(Monotonicity)：单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到原有的或者新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。

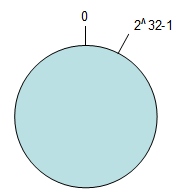
3、分散性(Spread)：在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生，也就是尽量降低分散性。

4、负载(Load)：负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同 的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

    在分布式集群中，对机器的添加删除，或者机器故障后自动脱离集群这些操作是分布式集群管理最基本的功能。如果采用常用的hash(object)%N算法，那么在有机器添加或者删除后，很多原有的数据就无法找到了，这样严重的违反了单调性原则。接下来主要讲解一下一致性哈希算法是如何设计的：

**环形Hash空间**

按照常用的hash算法来将对应的key哈希到一个具有2^32次方个桶的空间中，即0~(2^32)-1的数字空间中。现在我们可以将这些数字头尾相连，想象成一个闭合的环形。如下图



**把数据通过一定的hash算法处理后映射到环上**

现在我们将object1、object2、object3、object4四个对象通过特定的Hash函数计算出对应的key值，然后散列到Hash环上。如下图：

    Hash(object1) = key1；

    Hash(object2) = key2；

    Hash(object3) = key3；

    Hash(object4) = key4；



**将机器通过hash算法映射到环上**

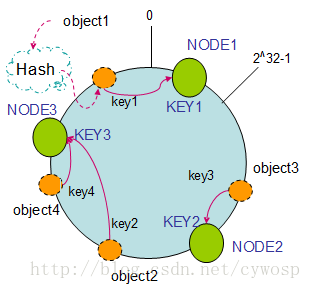
在采用一致性哈希算法的分布式集群中将新的机器加入，其原理是通过使用与对象存储一样的Hash算法将机器也映射到环中（一般情况下对机器的hash计算是采用机器的IP或者机器唯一的别名作为输入值），然后以顺时针的方向计算，将所有对象存储到离自己最近的机器中。

假设现在有NODE1，NODE2，NODE3三台机器，通过Hash算法得到对应的KEY值，映射到环中，其示意图如下：

Hash(NODE1) = KEY1;

Hash(NODE2) = KEY2;

Hash(NODE3) = KEY3;



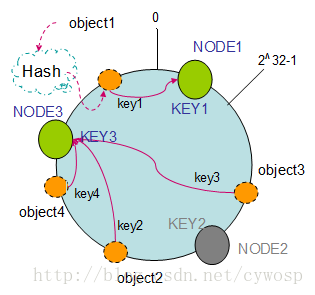
通过上图可以看出对象与机器处于同一哈希空间中，这样按顺时针转动object1存储到了NODE1中，object3存储到了NODE2中，object2、object4存储到了NODE3中。在这样的部署环境中，hash环是不会变更的，因此，通过算出对象的hash值就能快速的定位到对应的机器中，这样就能找到对象真正的存储位置了。

**机器的删除与添加**

普通hash求余算法最为不妥的地方就是在有机器的添加或者删除之后会照成大量的对象存储位置失效，这样就大大的不满足单调性了。下面来分析一下一致性哈希算法是如何处理的。

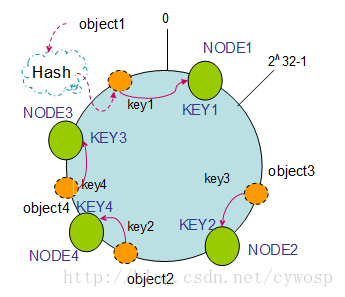
1. 节点（机器）的删除

    以上面的分布为例，如果NODE2出现故障被删除了，那么按照顺时针迁移的方法，object3将会被迁移到NODE3中，这样仅仅是object3的映射位置发生了变化，其它的对象没有任何的改动。如下图：



2. 节点（机器）的添加

    如果往集群中添加一个新的节点NODE4，通过对应的哈希算法得到KEY4，并映射到环中，如下图：



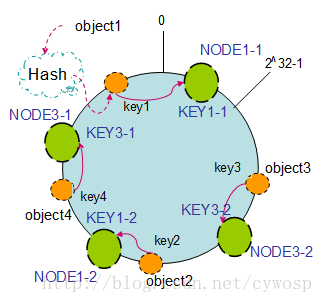
    通过按顺时针迁移的规则，那么object2被迁移到了NODE4中，其它对象还保持这原有的存储位置。通过对节点的添加和删除的分析，一致性哈希算法在保持了单调性的同时，还是数据的迁移达到了最小，这样的算法对分布式集群来说是非常合适的，避免了大量数据迁移，减小了服务器的的压力。

**平衡性**

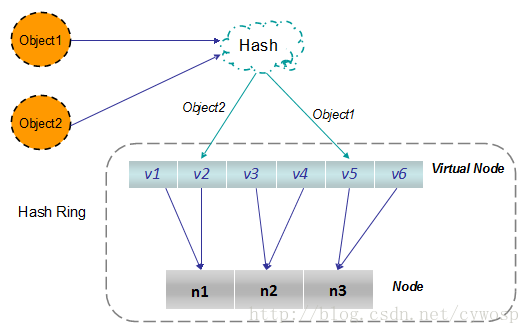
根据上面的图解分析，一致性哈希算法满足了单调性和负载均衡的特性以及一般hash算法的分散性，但这还并不能当做其被广泛应用的原由，因为还缺少了平衡性。下面将分析一致性哈希算法是如何满足平衡性的。hash算法是不保证平衡的，如上面只部署了NODE1和NODE3的情况（NODE2被删除的图），object1存储到了NODE1中，而object2、object3、object4都存储到了NODE3中，这样就照成了非常不平衡的状态。在一致性哈希算法中，为了尽可能的满足平衡性，其引入了虚拟节点。

    ——“虚拟节点”（ virtual node ）是实际节点（机器）在 hash 空间的复制品（ replica ），一实际个节点（机器）对应了若干个“虚拟节点”，这个对应个数也成为“复制个数”，“虚拟节点”在 hash 空间中以hash值排列。

以上面只部署了NODE1和NODE3的情况（NODE2被删除的图）为例，之前的对象在机器上的分布很不均衡，现在我们以2个副本（复制个数）为例，这样整个hash环中就存在了4个虚拟节点，最后对象映射的关系图如下：



根据上图可知对象的映射关系：object1->NODE1-1，object2->NODE1-2，object3->NODE3-2，object4->NODE3-1。通过虚拟节点的引入，对象的分布就比较均衡了。那么在实际操作中，正真的对象查询是如何工作的呢？对象从hash到虚拟节点到实际节点的转换如下图：



“虚拟节点”的hash计算可以采用对应节点的IP地址加数字后缀的方式。例如假设NODE1的IP地址为192.168.1.100。引入“虚拟节点”前，计算 cache A 的 hash 值：

Hash(“192.168.1.100”);

引入“虚拟节点”后，计算“虚拟节”点NODE1-1和NODE1-2的hash值：

Hash(“192.168.1.100#1”); // NODE1-1

Hash(“192.168.1.100#2”); // NODE1-2

**Mencache和redis比较**

没有必要过多的关注性能。由于Redis只使用单核，而Memcached可以使用多核，所以在比较上，平均每一个核上Redis在存储小数据时比Memcached性能更高。而在100k以上的数据中，Memcached性能要高于Redis，虽然Redis最近也在存储[**大数据**](http://lib.csdn.net/base/hadoop)的性能上进行优化，但是比起Memcached，还是稍有逊色。说了这么多，结论是，无论你使用哪一个，每秒处理请求的次数都不会成为瓶颈。

    你需要关注内存使用率。对于key-value这样简单的数据储存，memcache的内存使用率更高。如果采用hash结构，redis的内存使用率会更高。当然，这些都依赖于具体的应用场景。

    你需要关注关注数据持久化和主从复制时，只有redis拥有这两个特性。如果你的目标是构建一个缓存在升级或者重启后之前的数据不会丢失的话，那也只能选择redis。

    你应该关心你需要的操作。redis支持很多复杂的操作，甚至只考虑内存的使用情况，在一个单一操作里你常常可以做很多，而不需要将数据读取到客户端中（这样会需要很多的IO操作）。这些复杂的操作基本上和纯GET和POST操作一样快，所以你不只是需要GET/SET而是更多的操作时，redis会起很大的作用。

    对于两者的选择还是要看具体的应用场景，如果需要缓存的数据只是key-value这样简单的结构时，我在项目里还是采用memcache，它也足够的稳定可靠。如果涉及到存储，排序等一系列复杂的操作时，毫无疑问选择redis。

    关于redis和memcache的不同，下面罗列了一些相关说法，供记录：

    redis和memecache的不同在于[2]：

    1、存储方式：  
    memecache 把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小  
    redis有部份存在硬盘上，这样能保证数据的持久性，支持数据的持久化（笔者注：有快照和AOF日志两种持久化方式，在实际应用的时候，要特别注意配置文件快照参数，要不就很有可能服务器频繁满载做dump）。  
    2、数据支持类型：  
    redis在数据支持上要比memecache多的多。  
    3、使用底层模型不同：  
    新版本的redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。  
    4、运行环境不同：  
    redis目前官方只支持[**Linux**](http://lib.csdn.net/base/linux) 上去行，从而省去了对于其它系统的支持，这样的话可以更好的把精力用于本系统 环境上的优化，虽然后来微软有一个小组为其写了补丁。但是没有放到主干上

个人总结一下，有持久化需求或者对[**数据结构**](http://lib.csdn.net/base/datastructure)和处理有高级要求的应用，选择redis，其他简单的key/value存储，选择memcache。

**3.1.1 Memcached的内存管理机制**

Slab Allocation的原理相当简单。如图3所示，它首先从操作系统申请一大块内存，并将其分割成各种尺寸的块Chunk，并把尺寸相同的块分成组 Slab Class。其中，Chunk就是用来存储key-value数据的最小单位。每个Slab Class的大小，可以在Memcached启动的时候通过制定Growth Factor来控制。假定Figure 1中Growth Factor的取值为1.25，所以如果第一组Chunk的大小为88个字节，第二组Chunk的大小就为112个字节，依此类推。

  
 图3 Memcached内存管理[**架构**](http://lib.csdn.net/base/architecture)

当Memcached接收到客户端发送过来的数据时首先会根据收到数据的大小选择一个最合适的Slab Class，然后通过查询Memcached保存着的该Slab Class内空闲Chunk的列表就可以找到一个可用于存储数据的Chunk。当一条数据库过期或者丢弃时，该记录所占用的Chunk就可以回收，重新添 加到空闲列表中。从以上过程我们可以看出Memcached的内存管理制效率高，而且不会造成内存碎片，但是它最大的缺点就是会导致空间浪费。因为每个 Chunk都分配了特定长度的内存空间，所以变长数据无法充分利用这些空间。如图 4所示，将100个字节的数据缓存到128个字节的Chunk中，剩余的28个字节就浪费掉了。  
 图4 Memcached的存储空间浪费

**redis内存管理**

**3.1.2 Redis的内存管理机制**

Redis的内存管理主要通过源码中zmalloc.h和zmalloc.c两个文件来实现的。Redis为了方便内存的管理，在分配一块内存之 后，会将这块内存的大小存入内存块的头部。如图 5所示，real\_ptr是redis调用malloc后返回的指针。redis将内存块的大小size存入头部，size所占据的内存大小是已知的，为 size\_t类型的长度，然后返回ret\_ptr。当需要释放内存的时候，ret\_ptr被传给内存管理程序。通过ret\_ptr，程序可以很容易的算出 real\_ptr的值，然后将real\_ptr传给free释放内存。

为了保证单点故障下的数据可用性，Redis Cluster引入了Master节点和Slave节点。如图4所示，在Redis Cluster中，每个Master节点都会有对应的两个用于冗余的Slave节点。这样在整个集群中，任意两个节点的宕机都不会导致数据的不可用。当 Master节点退出后，集群会自动选择一个Slave节点成为新的Master节点。

发布订阅(pub/sub)是一种消息通信模式，主要的目的是解耦消息发布者和消息订阅者之间的耦合，这点和设计模式中的观察者模式比较相似。pub /sub不仅仅解决发布者和订阅者直接代码级别耦合也解决两者在物理部署上的耦合。[**Redis**](http://lib.csdn.net/base/redis)作为一个pub/sub server，在订阅者和发布者之间起到了消息路由的功能。订阅者可以通过subscribe和psubscribe命令向[**redis**](http://lib.csdn.net/base/redis) server订阅自己感兴趣的消息类型，redis将消息类型称为通道(channel)。当发布者通过publish命令向redis server发送特定类型的消息时。订阅该消息类型的全部client都会收到此消息。这里消息的传递是多对多的。一个client可以订阅多个 channel,也可以向多个channel发送消息。

HTTP幂等性：

# [Programming.log - a place to keep my thoughts on programming](http://www.cnblogs.com/weidagang2046/)

## [理解HTTP幂等性](http://www.cnblogs.com/weidagang2046/archive/2011/06/04/idempotence.html)

基于HTTP协议的Web API是时下最为流行的一种分布式服务提供方式。无论是在大型互联网应用还是企业级架构中，我们都见到了越来越多的SOA或RESTful的Web API。为什么Web API如此流行呢？我认为很大程度上应归功于简单有效的HTTP协议。HTTP协议是一种分布式的面向资源的网络应用层协议，无论是服务器端提供Web服务，还是客户端消费Web服务都非常简单。再加上浏览器、Javascript、AJAX、JSON以及HTML5等技术和工具的发展，互联网应用架构设计表现出了从传统的PHP、JSP、ASP.NET等服务器端动态网页向Web API + RIA（富互联网应用）过渡的趋势。Web API专注于提供业务服务，RIA专注于用户界面和交互设计，从此两个领域的分工更加明晰。在这种趋势下，Web API设计将成为服务器端程序员的必修课。然而，正如简单的Java语言并不意味着高质量的Java程序，简单的HTTP协议也不意味着高质量的Web API。要想设计出高质量的Web API，还需要深入理解分布式系统及HTTP协议的特性。

## 幂等性定义

本文所要探讨的正是HTTP协议涉及到的一种重要性质：幂等性(Idempotence)。在HTTP/1.1规范中幂等性的定义是：

Methods can also have the property of "idempotence" in that (aside from error or expiration issues) the side-effects of N > 0 identical requests is the same as for a single request.

从定义上看，HTTP方法的幂等性是指一次和多次请求某一个资源应该具有同样的副作用。幂等性属于语义范畴，正如编译器只能帮助检查语法错误一样，HTTP规范也没有办法通过消息格式等语法手段来定义它，这可能是它不太受到重视的原因之一。但实际上，幂等性是分布式系统设计中十分重要的概念，而HTTP的分布式本质也决定了它在HTTP中具有重要地位。

## 分布式事务 vs 幂等设计

为什么需要幂等性呢？我们先从一个例子说起，假设有一个从账户取钱的远程API（可以是HTTP的，也可以不是），我们暂时用类函数的方式记为：

bool withdraw(account\_id, amount)

withdraw的语义是从account\_id对应的账户中扣除amount数额的钱；如果扣除成功则返回true，账户余额减少amount；如果扣除失败则返回false，账户余额不变。值得注意的是：和本地环境相比，我们不能轻易假设分布式环境的可靠性。一种典型的情况是withdraw请求已经被服务器端正确处理，但服务器端的返回结果由于网络等原因被掉丢了，导致客户端无法得知处理结果。如果是在网页上，一些不恰当的设计可能会使用户认为上一次操作失败了，然后刷新页面，这就导致了withdraw被调用两次，账户也被多扣了一次钱。如图1所示：

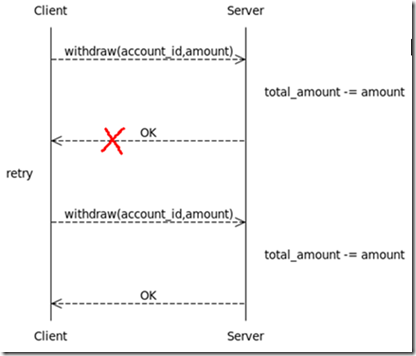


图1

这个问题的解决方案一是采用分布式事务，通过引入支持分布式事务的中间件来保证withdraw功能的事务性。分布式事务的优点是对于调用者很简单，复杂性都交给了中间件来管理。缺点则是一方面架构太重量级，容易被绑在特定的中间件上，不利于异构系统的集成；另一方面分布式事务虽然能保证事务的ACID性质，而但却无法提供性能和可用性的保证。

另一种更轻量级的解决方案是幂等设计。我们可以通过一些技巧把withdraw变成幂等的，比如：

int create\_ticket()

bool idempotent\_withdraw(ticket\_id, account\_id, amount)

create\_ticket的语义是获取一个服务器端生成的唯一的处理号ticket\_id，它将用于标识后续的操作。idempotent\_withdraw和withdraw的区别在于关联了一个ticket\_id，一个ticket\_id表示的操作至多只会被处理一次，每次调用都将返回第一次调用时的处理结果。这样，idempotent\_withdraw就符合幂等性了，客户端就可以放心地多次调用。

基于幂等性的解决方案中一个完整的取钱流程被分解成了两个步骤：1.调用create\_ticket()获取ticket\_id；2.调用idempotent\_withdraw(ticket\_id, account\_id, amount)。虽然create\_ticket不是幂等的，但在这种设计下，它对系统状态的影响可以忽略，加上idempotent\_withdraw是幂等的，所以任何一步由于网络等原因失败或超时，客户端都可以重试，直到获得结果。如图2所示：

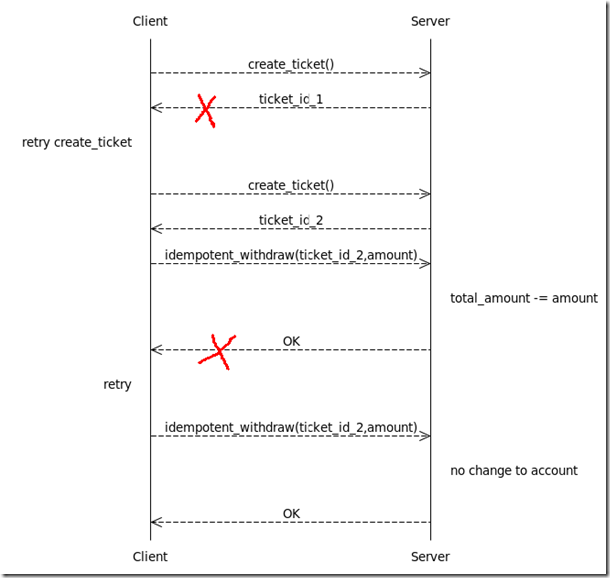


图2

和分布式事务相比，幂等设计的优势在于它的轻量级，容易适应异构环境，以及性能和可用性方面。在某些性能要求比较高的应用，幂等设计往往是唯一的选择。

## HTTP的幂等性

HTTP GET方法用于获取资源，不应有副作用，所以是幂等的。比如：GET http://www.bank.com/account/123456，不会改变资源的状态，不论调用一次还是N次都没有副作用。

HTTP DELETE方法用于删除资源，有副作用，但它应该满足幂等性。比如：DELETE http://www.forum.com/article/4231，调用一次和N次对系统产生的副作用是相同的，即删掉id为4231的帖子；因此，调用者可以多次调用或刷新页面而不必担心引起错误。

比较容易混淆的是HTTP POST和PUT。POST和PUT的区别容易被简单地误认为“POST表示创建资源，PUT表示更新资源”；而实际上，二者均可用于创建资源，更为本质的差别是在幂等性方面。在HTTP规范中对POST和PUT是这样定义的：

POST所对应的URI并非创建的资源本身，而是资源的接收者。比如：POST http://www.forum.com/articles的语义是在http://www.forum.com/articles下创建一篇帖子，HTTP响应中应包含帖子的创建状态以及帖子的URI。两次相同的POST请求会在服务器端创建两份资源，它们具有不同的URI；所以，**POST方法不具备幂等性**。而PUT所对应的URI是要创建或更新的资源本身。比如：PUT http://www.forum/articles/4231的语义是创建或更新ID为4231的帖子。对同一URI进行多次PUT的副作用和一次PUT是相同的；因此，PUT方法具有幂等性。

**反向代理的指令**

这几个都是APACHE的代理指令：

1、ProxyPass：

语法：ProxyPass [path] !|url

它主要是用作URL前缀匹配，**不能有正则表达式**，它里面配置的Path实际上是一个虚拟的路径，在反向代理到后端的url后，**path是不会带过去的**，使用示例：

1）、ProxyPass /images/ !

 这个示例表示，/images/的请求不被转发。

2）、ProxyPass /mirror/foo/ http://backend.example.com/

我们假设当前的服务地址是http://example.com/，如果我们做下面这样的请求：

http://example.com/mirror/foo/bar

那将被转成内部请求：

http://backend.example.com/bar

注：配置的时候，不需要被转发的请求，要配置在需要被转发的请求前面。

3、ProxyPassReverse

语法：ProxyPassReverse [路径] url

它一般和ProxyPass指令配合使用，此指令使Apache调整HTTP重定向应答中Location, Content-Location, URI头里的URL，这样可以避免在Apache作为反向代理使用时，。后端服务器的HTTP重定向造成的绕过反向代理的问题。参看下面的示例：

ProxyPass /example http://www.example.com/  
ProxyPassReverse /example http://www.example.com/

ProxyPassReverse的作用就是反向代理，如果没有加这样的反向代理设置的情况下，访问http://www.test.com/example/a，如果www.example.com对请求进行了redirect至http://www.test.com/b，那么，客户端就会绕过反向代理，进而访问http://www.test.com/example/b。如果设置了反向代理，则会在转交HTTP重定向应答到客户端之前调整它为http://www.test.com/example/a/b，即是在原请求之后追加上了redirect的路径。

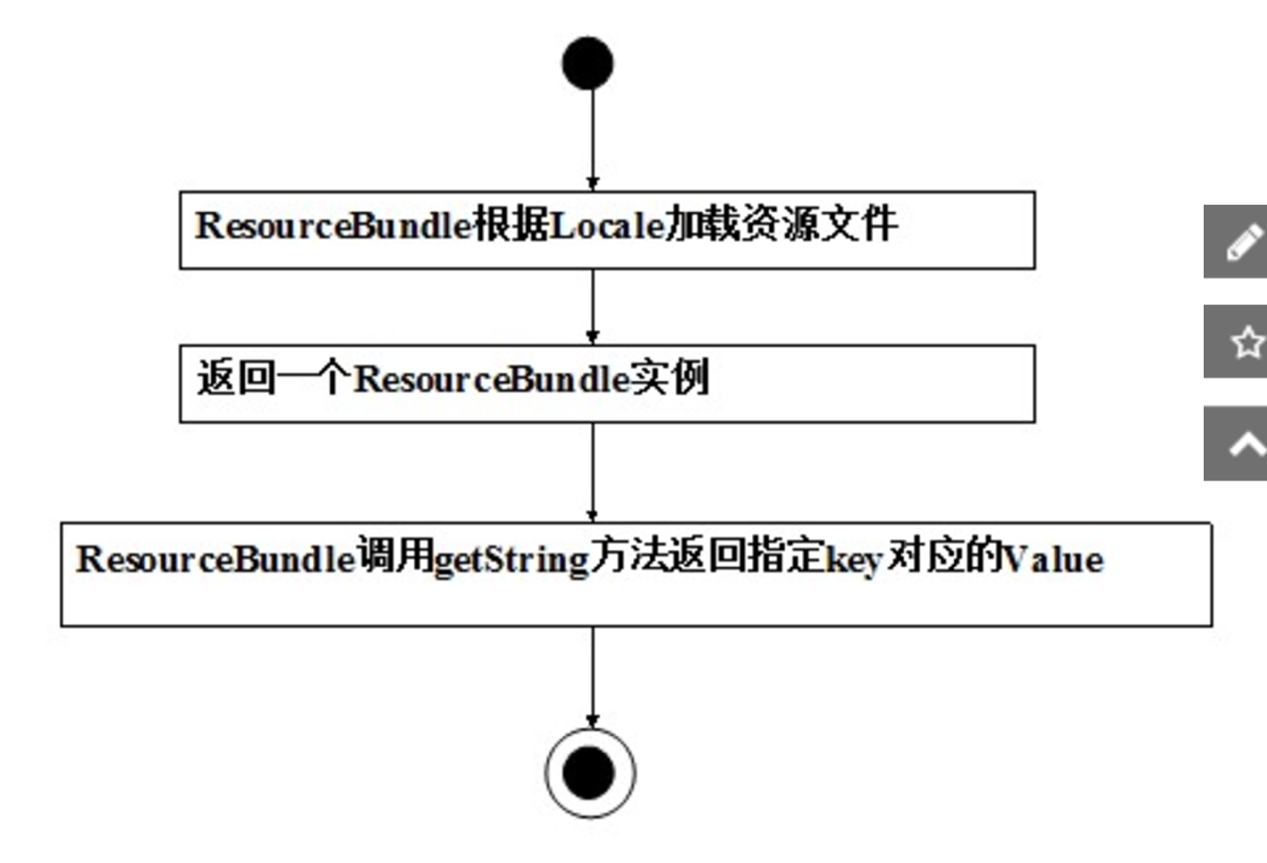
**国际化实现代码：**

### 第一步，实现选择语言页：

这里我们覆盖struts的execute方法，设置语言，讲语言设置到session中，以便后来界面使用。

**[html]** [view plain](http://blog.csdn.net/laner0515/article/details/11831815) [copy](http://blog.csdn.net/laner0515/article/details/11831815)

1. @Override
2. public ActionForward execute(ActionMapping mapping, ActionForm form,
3. HttpServletRequest request, HttpServletResponse response)
4. throws Exception {
5. String lang = request.getParameter("lang");
6. Locale locale= Locale.getDefault();
7. //确定初始化的语言
8. if("zh".equals(lang)){
9. locale = new Locale("zh","CN");
10. }else if("en".equals(lang)){
11. locale=new Locale("en","US");
12. }
14. //将Locale设置到session中
15. //request.getSession().setAttribute(Globals.LOCALE\_KEY,locale);
16. this.setLocale(request, locale);
17. return mapping.findForward("index");
18. }

****

**AOP**

 Join point：一个要被拦截的方法（在spring里面我们主要做的就是拦截方法，所以对于属性的AOP就先不理会了）

      Pointcut：在execution表达式里面声明的一个或多个方法被拦截方法的集合

      advice    ：应用在被拦截方法中的代码或者说函数，如，切面类中被@before(....)等注释了的成员函数

      advisor：负责声明“建议”在什么时候被执行，比如说，在被拦截方法执行前@before（...） ,@afterReturnning（...）等

      Aspect   ：一个封装了切入面和切入点的模块

[spring](http://lib.csdn.net/base/javaee) 的环绕通知和前置通知，后置通知有着很大的区别，主要有两个重要的区别：

1） 目标方法的调用由环绕通知决定，即你可以决定是否调用目标方法，而前置和后置通知   是不能决定的，他们只是在方法的调用前后执行通知而已，即目标方法肯定是要执行的。

2）  环绕通知可以控制返回对象，即你可以返回一个与目标对象完全不同的返回值，虽然这很危险，但是你却可以办到。而后置方法是无法办到的，因为他是在目标方法返回值后调用

@Aspect

@Component

public class LogIntercept {

@Pointcut("execution(public \* com.itsoft.action..\*.\*(..))")

public void recordLog(){}

@Before("recordLog()")

public void before() {

this.printLog("已经记录下操作日志@Before 方法执行前");

}

@Around("recordLog()")

public void around(ProceedingJoinPoint pjp) throws Throwable{

this.printLog("已经记录下操作日志@Around 方法执行前");

pjp.proceed();

this.printLog("已经记录下操作日志@Around 方法执行后");

}

@After("recordLog()")

public void after() {

this.printLog("已经记录下操作日志@After 方法执行后");

}

private void printLog(String str){

System.out.println(str);

}

}

WSDL:

  WSDL 文档将Web服务定义为服务访问点或端口的集合。在 WSDL 中，由于服务访问点和消息的抽象定义已从具体的服务部署或数据格式绑定中分离出来，因此可以对抽象定义进行再次使用：消息，指对交换数据的抽象描述；而端口类型，指操作的抽象集合。用于特定端口类型的具体协议和数据格式规范构成了可以再次使用的绑定。将Web访问地址与可再次使用的绑定相关联，可以定义一个端口，而端口的集合则定义为服务。因此，WSDL 文档在Web服务的定义中使用下列元素：

* **Types** - 数据类型定义的容器，它使用某种类型系统(一般地使用XML Schema中的类型系统)。
* **Message** - 通信消息的数据结构的抽象类型化定义。使用Types所定义的类型来定义整个消息的数据结构。
* **Operation** - 对服务中所支持的操作的抽象描述，一般单个Operation描述了一个访问入口的请求/响应消息对。
* **PortType** - 对于某个访问入口点类型所支持的操作的抽象集合，这些操作可以由一个或多个服务访问点来支持。
* **Binding** - 特定端口类型的具体协议和数据格式规范的绑定。
* **Port** - 定义为协议/数据格式绑定与具体Web访问地址组合的单个服务访问点。
* **Service** - 相关服务访问点的集合。

UDDI

**UDDI 是一种目录服务，企业可以使用它对 Web services 进行注册和搜索。**

**UDDI，英文为 "Universal Description, Discovery and Integration"，可译为“通用描述、发现与集成服务”。**

**什么是 UDDI？**

UDDI 是一个独立于平台的框架，用于通过使用 Internet 来描述服务，发现企业，并对企业服务进行集成。

* UDDI 指的是通用描述、发现与集成服务
* UDDI 是一种用于存储有关 web services 的信息的目录。
* UDDI 是一种由 WSDL 描述的 web services 界面的目录。
* UDDI 经由 SOAP 进行通信
* UDDI 被构建入了微软的 .NET 平台

**UDDI 基于什么？**

UDDI 使用 W3C 和 IETF\* 的因特网标准，比如 XML、HTTP 和 DNS 协议。

UDDI 使用 WSDL 来描述到达 web services 的界面

此外，通过采用 SOAP，还可以实现跨平台的编程特性，大家知道，SOAP 是 XML 的协议通信规范，可在 W3C 的网站找到相关的信息。

**\*注释：**IETF - Internet Engineering Task Force

**UDDI 的好处**

任何规模的行业或企业都能得益于 UDDI。

在 UDDI 之前，还不存在一种 Internet 标准，可以供企业为它们的企业和伙伴提供有关其产品和服务的信息。也不存在一种方法，来集成到彼此的系统和进程中。

UDDI 规范帮助我们解决的问题：

* 使得在成百万当前在线的企业中发现正确的企业成为可能
* 定义一旦首选的企业被发现后如何启动商业
* 扩展新客户并增加对目前客户的访问
* 扩展销售并延伸市场范围
* 满足用户驱动的需要，为在全球 Internet 经济中快速合作的促进来清除障碍

**UDDI 如何被使用**

假如行业发布了一个用于航班比率检测和预订的 UDDI 标准，航空公司就可以把它们的服务注册到一个 UDDI 目录中。然后旅行社就能够搜索这个 UDDI 目录以找到航空公司预订界面。当此界面被找到后，旅行社就能够立即与此服务进行通信，这样由于它使用了一套定义良好的预订界面。

**谁在支持 UDDI？**

UDDI 是一个跨行业的研究项目，由所有主要的平台和软件提供商驱动，比如：Dell, Fujitsu, HP, Hitachi, IBM, Intel, Microsoft, Oracle, SAP, 以及 Sun, 它既是一个市场经营者的团体，也是一个电子商务的领导者。

已有数百家公司参与了这个 UDDI 团体。

JAX-RPC与JAX-WS

Web 服务已经出现很久了。首先是 SOAP，但 SOAP 仅描述消息的情况，然后是 WSDL，WSDL 并不会告诉您如何使用 [**Java**](http://lib.csdn.net/base/java)™ 编写 Web 服务。在这种情况下，JAX-RPC 1.0 应运而生。经过数月使用之后，编写此规范的 [**Java**](http://lib.csdn.net/base/java)Community Process (JCP) 人员认识到需要对其进行一些调整，调整的结果就是 JAX-RPC 1.1。该规范使用大约一年之后，JCP 人员希望构建一个更好的版本：JAX-RPC 2.0。其主要目标是与行业方向保持一致，但行业中不仅只使用 RPC Web 服务，还使用面向消息的 Web 服务。因此从名称中去掉了“RPC”，取而代之的是“WS”（当然表示的是 Web 服务）。因此 JAX-RPC 1.1 的后续版本是 JAX-WS 2.0【[**java**](http://lib.csdn.net/base/java)API for XML-based Web Services】。

相同点？

在列出 JAX-RPC 1.1 和 JAX-WS 2.0 的差异前，我们应该首先讨论一下二者的相同之处：

* JAX-WS 仍然支持 SOAP 1.1 over HTTP 1.1，因此互操作性将不会受到影响，仍然可以在网上传递相同的消息。
* JAX-WS 仍然支持 WSDL 1.1，因此您所学到的有关该规范的知识仍然有用。WSDL 2.0 规范已经接近完成，但在 JAX-WS 2.0 相关工作结束时其工作仍在进行中。

区别何在？

* SOAP 1.2
  + JAX-RPC 和 JAX-WS 都支持 SOAP 1.1。JAX-WS 还支持 SOAP 1.2。
* XML/HTTP
  + WSDL 1.1 规范在 HTTP 绑定中定义，这意味着利用此规范可以在不使用 SOAP 的情况下通过 HTTP 发送 XML 消息。JAX-RPC 忽略了 HTTP 绑定。而 JAX-WS 添加了对其的支持。
* WS-I Basic Profile
  + JAX-RPC 支持 WS-I Basic Profile (BP) V1.0。JAX-WS 支持 BP 1.1。（WS-I 即 Web 服务互操作性组织。）
* 新 Java 功能
  + JAX-RPC 映射到 Java 1.4。JAX-WS 映射到 Java 5.0。JAX-WS 依赖于 Java 5.0 中的很多新功能。
  + Java EE 5 是 J2EE 1.4 的后续版本，添加了对 JAX-WS 的支持，但仍然支持 JAX-RPC，这可能会对 Web 服务新手造成混淆。
* 数据映射模型
  + JAX-RPC 具有自己的映射模型，此模型大约涵盖了所有模式类型中的 90%。它没有涵盖的部分映射到了 javax.xml.soap.SOAPElement。
  + JAX-WS 的数据映射模型是 JAXB。JAXB 可保证所有 XML 模式的映射。
* 接口映射模型
  + JAX-WS 的基本接口映射模型与 JAX-RPC 的区别并不大，不过二者之间存在以下差异：
    - JAX-WS 的模型使用新的 Java 5.0 功能。
    - JAX-WS 的模型引入了异步功能。
* 动态编程模型
  + JAX-WS 的动态客户机模型与 JAX-RPC 的对应模型差别很大。很多更改都是为了认可行业需求：
    - 引入了面向消息的功能。
    - 引入了动态异步功能。
  + JAX-WS 还添加了动态服务器模型，而 JAX-RPC 则没有此模型。
* 消息传输优化机制（Message Transmission Optimization Mechanism，MTOM】
  + JAX-WS 通过 JAXB 添加了对新附件规范 MTOM 的支持。Microsoft 从来没有给 SOAP 添加附件规范；但似乎大家都支持 MTOM，因此应该能够实现附件互操作性。
* 处理程序模型
  + 从 JAX-RPC 到 JAX-WS 的过程中，处理程序模型发生了很大的变化。
  + JAX-RPC 处理程序依赖于 SAAJ 1.2。JAX-WS 处理程序依赖于新的 SAAJ 1.3 规范。

**SOAP调用原理**

1，用工具通过WSDL文件产生一个SEI(service endpoint interface),一个java的interface，能够对应该web service提供的功能。这个interface的类名会对应到WSDL的portType名称，方法会和operation对应，方法的参数会和message以及types对应。CXF和Axis都提供这样的工具：wsdl2java。【上面的WSCXFProvider就是SEI】  
2，初始化Service，指定wsdL URL和service的QName，service的QName的前一部分是命名空间，后一部分是名称，与WSDL文件中的<wsdl:service name="WSCXFProviderService">部分对应。然后通过getPort得到一个实现了SEI的实例，这个实例被叫做Proxy。它的QName与 wsdl文件中的<wsdl:port name="WSCXFProviderPort"部分对应。  
3，通过SEI调用web service，传给SEI的参数是加上JAXB annotation的java类（简单类型和String不用标记）。  
4，传入的参数被序列化为SOAP消息的payload（body部分的xml），这是因为service.getPort是通过proxy机制创建的，调用这个proxy的方法时，和它关联的InvocationHandler的invoke方法也会执行，invoke方法会通过JAXB把java参数序列化为XML。然后会把soap request发到服务器端。  
5，返回response是后与4类似，InvocationHandler负责把XML利用JAXB反序列化为java对象。

**SOAP与REST对比：**

相对REST而言，SOAP 1.2多出一些开销，不过这些开销也带来了一些好处。首先，SOAP在三个方面离不开XML（Extensible Markup Language，可扩展标记语言）：SOAP信封（envelope）是基于XML的，它定义了消息里有什么以及如何处理它；一套用于数据类型的编码规则；过程调用和响应的规划。SOAP信封由传输协议（HTTP/HTTPS）发出，RPC（Remote Procedure Call，远程过程调用）得到执行，然后一个XML文档随SOAP信封返回。

需要注意的是，基于“通用”传输协议是SOAP的一个优点。REST目前基于HTTP/HTTPS；而SOAP可支持任何传输协议，从HTTP/HTTPS到SMTP（Simple Mail Transfer Protocol，简单邮件传送协议），甚至JMS（Java Messaging Service，Java消息传递服务）。不过，由于XML较为冗长且解析费时，因此采用XML也成为一个弊端

REST的适用场合包括：

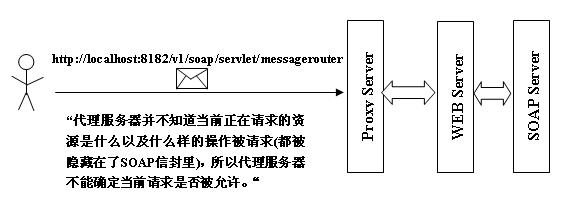
* **有限的带宽和资源** 别忘了返回的结构可以采用（由开发者定义的）任何格式。另外，由于REST采用标准的GET、PUT、POST和DELETE动词，因此可被任何浏览器所支持。除此以外，REST还可以使用为目前大多数浏览器支持的XMLHttpRequest对象，这为AJAX增色不少。
* **完全无状态的操作** 对于那些需多步执行的操作，REST并非最佳选择，采用SOAP更合适。但是，如果你需要无状态的CRUD（Create/Read/Update/Delete，创建/读取/更新/删除）操作，那么应采用REST。
* **缓存考虑** 若要利用无状态操作的特性，使得信息可被缓存，那么REST是很好的选择。

SOAP是最佳选择：

* **异步处理与调用** 如果你的应用需要确保可靠性与安全性，那么请采用SOAP。SOAP 1.2为确保这种操作补充定义了WSRM（WS-Reliable Messaging）等标准。
* **形式化契约** 若提供者/消费者双方必须就交换格式取得一致，那么采用SOAP更合适。SOAP 1.2为这种交互提供了严格的规范。
* **有状态的操作** 如果应用需要上下文信息与对话状态管理，那么应采用SOAP。SOAP 1.2为此补充定义了WS-Security、WS-Transactions和WS-Coordination等标准。相比之下，REST方法要求开发者自己来实现这些框架性工作。

SOAP与代理服务器耦合

##### 图 5. SOAP 与代理服务器 (Proxy Servers)



所有的 SOAP 消息经过代理服务器，只能看到（http://localhost:8182/v1/soap/servlet/messagerouter, HTTP POST）这样的信息，如果代理服务器想知道当前的 HTTP 请求具体做的是什么，必须对 SOAP 的消息体解码，这样的话，意味着要求第三方的代理服务器需要理解当前的 SOAP 消息语义，而这种 SOAP 应用与代理服务器之间的紧耦合关系是不合理的。

要深入理解REST，需要理解REST的五个关键词：

1. 资源（Resource）
2. 资源的表述（Representation）
3. 状态转移（State Transfer）
4. 统一接口（Uniform Interface）
5. 超文本驱动（Hypertext Driven）

什么是资源？

资源是一种看待服务器的方式，即，将服务器看作是由很多离散的资源组成。每个资源是服务器上一个可命名的抽象概念。因为资源是一个抽象的概念，所以它不仅仅能代表服务器文件系统中的一个文件、数据库中的一张表等等具体的东西，可以将资源设计的要多抽象有多抽象，只要想象力允许而且客户端应用开发者能够理解。与面向对象设计类似，资源是以名词为核心来组织的，首先关注的是名词。一个资源可以由一个或多个URI来标识。URI既是资源的名称，也是资源在Web上的地址。对某个资源感兴趣的客户端应用，可以通过资源的URI与其进行交互。

什么是资源的表述？

**资源的表述是一段对于资源在某个特定时刻的状态的描述。**可以在客户端-服务器端之间转移（交换）。资源的表述可以有多种格式，例如HTML/XML/JSON/纯文本/图片/视频/音频等等。资源的表述格式可以通过协商机制来确定。请求-响应方向的表述通常使用不同的格式。

什么是状态转移？

状态转移（state transfer）与状态机中的状态迁移（state transition）的含义是不同的。状态转移说的是：**在客户端和服务器端之间转移（transfer）代表资源状态的表述。**通过转移和操作资源的表述，来间接实现操作资源的目的。

什么是统一接口？

REST要求，必须通过统一的接口来对资源执行各种操作。对于每个资源只能执行一组有限的操作。以HTTP/1.1协议为例，HTTP/1.1协议定义了一个操作资源的统一接口，主要包括以下内容：

* 7个HTTP方法：GET/POST/PUT/DELETE/PATCH/HEAD/OPTIONS
* HTTP头信息（可自定义）
* HTTP响应状态代码（可自定义）
* 一套标准的内容协商机制
* 一套标准的缓存机制
* 一套标准的客户端身份认证机制

REST还要求，对于资源执行的操作，**其操作语义必须由HTTP消息体之前的部分完全表达，不能将操作语义封装在HTTP消息体内部。这样做是为了提高交互的可见性，以便于通信链的中间组件实现缓存、安全审计等等功能。**

什么是超文本驱动？

“超文本驱动”又名“将超媒体作为应用状态的引擎”（Hypermedia As The Engine Of Application State，来自Fielding博士论文中的一句话，缩写为HATEOAS）。将Web应用看作是一个由很多状态（应用状态）组成的有限状态机。资源之间通过超链接相互关联，超链接既代表资源之间的关系，也代表可执行的状态迁移。在超媒体之中不仅仅包含数据，还包含了状态迁移的语义。以超媒体作为引擎，驱动Web应用的状态迁移。通过超媒体暴露出服务器所提供的资源，服务器提供了哪些资源是在运行时通过解析超媒体发现的，而不是事先定义的。从面向服务的角度看，超媒体定义了服务器所提供服务的协议。客户端应该依赖的是超媒体的状态迁移语义，而不应该对于是否存在某个URI或URI的某种特殊构造方式作出假设。一切都有可能变化，只有超媒体的状态迁移语义能够长期保持稳定。

**@Provider**注解用在任何对JAX-RS运行时（如MessageBodyReader和MessageBodyWriter）有意义的事物上。对HTTP请求，MessageBodyReader用来将HTTP请求实体段映射为方法参数。在响应的时候，返回的值使用MessageBodyWriter来映射成HTTP响应实体段。如果应用程序需要提供其他的元数据，如HTTP头或不同的状态代码，方法可以返回一个打包了实体的Response，该Response可以使用Response.ResponseBuilder创建。

SOA与component

Component 的重用，一般是通过在应用开发过程过程中Plug和Assemble实现，与应用具有相同的生命周期。**不同的应用需要部署相同的Component，**因此在应用上线和运行周期内Unplug和Refactoring变得不是那么容易。而Service的出现，则代表了一种更高层次的抽象和演进。它从使用者的视角，更直接地描述到底能提供什么，而不关注它是怎么提供的。只要它提供了业务需要的行为和质量，谁又关注它到底是通过Component实现，还是Legacy系统开放出来的适配接口？较Component而言，它提供了独立于应用部署的可能性，可以与应用具有不同的生命周期，可以独立的重构和演进，使得大规模的重用变得更加的容易。

Service在问题域和实现域之间增加了一个中间层，让业务和IT可以更好的对话。反映到系统设计开发上，SOA并不是彻底颠覆传统面向对象和组件的软件工程方法，而是在分析问题和解决问题方法上的进一步演进，并因此带来新的关于服务工程学的内容，包括服务的识别发现，服务设计实现，服务管控治理等一系列的过程和活动。

## SOA是英文词语"Service Oriented Architecture"的缩写，中文有多种翻译，如"面向服务的体系结构"、"以服务为中心的体系结构"和"面向服务的架构"，其中"面向服务的架构"比较常见。SOA有很多定义，但基本上可以分为两类：一类认为SOA主要是一种架构风格；另一类认为SOA是包含运行环境、编程模型、架构风格和相关方法论等在内的一整套新的分布式软件系统构造方法和环境，涵盖服务的整个生命周期：建模-开发-整合-部署-运行-管理。后者概括的范围更大，着眼于未来的发展，我们更倾向于后者，认为SOA是分布式软件系统构造方法和环境的新发展阶段。

在SOA架构风格中，服务是最核心的抽象手段，业务被划分（组件化）为一系列粗粒度的业务服务和业务流程。业务服务相对独立、自包含、可重用，由一个或者多个分布的系统所实现，而业务流程由服务组装而来。一个"服务"定义了一个与业务功能或业务数据相关的接口，以及约束这个接口的契约，如服务质量要求、业务规则、安全性要求、法律法规的遵循、关键业绩指标（Key Performance Indicator，KPI）等。接口和契约采用中立、基于标准的方式进行定义，它独立于实现服务的硬件平台、操作系统和编程语言。这使得构建在不同系统中的服务可以以一种统一的和通用的方式进行交互、相互理解。除了这种不依赖于特定技术的中立特性，通过服务注册库（Service Registry）加上企业服务总线（Enterprise Service Bus）来支持动态查询、定位、路由和中介（Mediation）的能力，使得服务之间的交互是动态的，位置是透明的。技术和位置的透明性，使得服务的请求者和提供者之间高度解耦。这种松耦合系统的好处有两点：一点是它适应变化的灵活性；另一点是当某个服务的内部结构和实现逐渐发生改变时，不影响其他服务。而紧耦合则是指应用程序的不同组件之间的接口与其功能和结构是紧密相连的，因而当发生变化时，某一部分的调整会随着各种紧耦合的关系引起其他部分甚至整个应用程序的更改，这样的系统架构就很脆弱了。

SOA架构带来的另一个重要观点是业务驱动IT，即IT和业务更加紧密地对齐。以粗粒度的业务服务为基础来对业务建模，会产生更加简洁的业务和系统视图；以服务为基础来实现的IT系统更灵活、更易于重用、更好（也更快）地应对变化；以服务为基础，通过显式地定义、描述、实现和管理业务层次的粗粒度服务（包括业务流程），提供了业务模型和相关IT实现之间更好的"可追溯性"，减小了它们之间的差距，使得业务的变化更容易传递到IT。

因此，可以将SOA的主要优点概括为：IT能够更好更快地提供业务价值（Business Centric）、快速应变能力（Flexibility）、重用（Reusability）。

从演变的历程来看，SOA在很多年前就被提出来了，现在SOA的再现和流行是若干因素的结合。一方面是多年的软件工程发展和实践所积累的经验、方法和各种设计/架构模式，包括OO/CBD/MDD/MDA、EAI和中间件；另一方面是互联网的多年发展带来前所未有的分布式系统的交互能力和标准化基础。与此同时，企业越来越重视业务模型本身的组件化，以支持高度灵活的业务战略。但是现有的企业软件架构不够灵活，难以适应日益复杂的企业整合，难以满足随需应变商务的需要，因此与业务对齐、以业务的敏捷应变能力为首要目标、松散耦合、支持重用的SOA架构方法得到青睐。更多的细节请参见"[以服务为中心的企业整合](http://www.ibm.com/developerworks/cn/webservices/ws-soi1/)"。

基于我们同客户打交道的经验，有必要在这里澄清大家经常混淆的几个基本问题。

第一，SOA是架构风格，是方法；而不是具体架构具体实现技术（如Web Service）、具体架构元素（如企业服务总线，Enterprise Service Bus，ESB）。

经常有人认为只要用了Web Service，就是SOA了。这是不对的，Web Service只是实现服务的一种具体技术表现形式。同样，认为搞SOA，就是买点软件，建个ESB，这也是不对的，ESB只是SOA架构风格中的一部分。首先，ESB是一种从实践中总结出来的架构风格元素，即BUS（总线模式）；其次，ESB的主要功能是负责连通性和服务中介（Service Mediation），解耦服务的请求者和服务的提供者。

第二，SOA的首要目标是IT与业务对齐，支持业务的快速变化；其次是IT架构的灵活性和IT资产的重用。

业务对敏捷性的需要，是SOA最大的驱动力。一方面是业务在这方面的要求越来越高；另一方面是今天的IT很不灵活，很难适应业务快速变化的需求，不仅仅是因为IT架构不灵活，更重要的是业务模型中的元素和IT系统的元素之间存在很大的差距。这种不对齐，导致业务人员和IT人员之间的沟通不够有效，业务的变化需要花费很大的代价传递到IT系统。很难想像，业务人员对一个Java对象，一个EJB或者一个JSP页面感兴趣，这离业务世界太远了。这种业务和IT的对齐，需要在IT系统中实现更高阶的抽象元素，就是业务模型中的元素（服务、流程、业绩管理），并且满足业务需要的水平整合（将人、信息、应用和流程端到端地动态整合起来）。这样一个以服务为中心的、端到端整合的环境，首先使得业务变化可以在业务元素这个层面上沟通，更容易、更准确地从业务传递到IT。其次，这种变化被隔离在需要变化的局部，而不扩散到系统的其他部分。这就需要整个IT架构本身是松散耦合的，一个服务的变化（功能、数据、过程、技术环境等）不影响其他服务。最后，我们希望这些反映业务元素的服务，是相对稳定、可以重用的，这对快速适应变化、减少成本是非常重要的。

第三，在工程上，SOA的重点是服务建模和基于SOA的设计原则进行架构决策和设计。

经常碰到客户提出这样的问题：SOA挺好，为什么好？怎么做才是SOA的方法？跟过去的方法，比如OO/CBD有什么不同？有时候一个J2EE服务器就好了，为什么那么复杂？

从建模和设计的角度来说，SOA更多地侧重在业务层次上，也就是通过服务建模将业务组件化为服务模型，它是业务架构的底层，是技术架构的顶层，承上启下，是灵活的业务模型和IT之间的桥梁，保证二者之间的"可追溯性"。从这里往下，是基于已有的方法，比如OO/CBD来进行的。从架构的层次上，SOA更多地侧重于如何将企业范围内多个分布的系统（包括已有系统/遗留系统）连接起来（ESB，Adapter/Connector），如何将它们的功能、数据转化为服务，如何通过服务中介机制（ESB，Service Registry）保证服务之间以松散耦合的方式交互，如何组装（集成）服务为流程，如何管理服务和流程等。从这往下，对于实现服务的一个具体应用，它的架构、设计和实现是可以基于已有的实践和方法的，比如J2EE或.NET。

有些时候，由于业务需求比较简单，所有这些东西都在一个J2EE的应用服务器上，有些要素不是那么突出，不过随着系统规模的扩大，要解决的业务问题更复杂、范围更大的时候，SOA的各种架构要素就会变得越来越重要。

在下面的小节里就概括地讨论一下SOA的若干个重要方面，包括：面向服务的计算环境，编程模型，架构风格，工程方法，以及相关的技术。

。而之所以能实现这一点，全依赖于组件的自描述功能，也就是说组件中不仅仅包含一个个类，还额外包含了描述这些类、方法、参数等等的类型信息。

对比我们之前提到的组件技术，它也提供了服务对象的元数据，但是这些元数据的描述是与具体的组件技术相关的，而调用组件中对象的技术也是依赖于某一特定平台的。为了获得松耦合的特性，我们采用了Web Services技术作为SOA的具体实现。其中WSDL用于描述服务的元数据，而SOAP则规定了如何调用服务并返回结果。

**LUCENE**

**倒排序表**

|  |  |  |
| --- | --- | --- |
| 关键词 \_ | 文章号[出现频率] | 出现位置 \_ |
| guangzhou | 1[2] | 3，6 |
| he | 2[1] | 1 |
| i | 1[1] | 4 |
| live | 1[2],2[1] | 2，5，2 |
| shanghai | 2[1] | 3 |
| tom | 1[1] | 1 |

为了对文档进行索引，Lucene 提供了五个基础的类，他们分别是 Document, Field, IndexWriter, Analyzer, Directory。下面我们分别介绍一下这五个类的用途：

**Document**

Document 是用来描述文档的，这里的文档可以指一个 HTML 页面，一封电子邮件，或者是一个文本文件。一个 Document 对象由多个 Field 对象组成的。可以把一个 Document 对象想象成数据库中的一个记录，而每个 Field 对象就是记录的一个字段。

**Field**

Field 对象是用来描述一个文档的某个属性的，比如一封电子邮件的标题和内容可以用两个 Field 对象分别描述。

**Analyzer**

在一个文档被索引之前，首先需要对文档内容进行分词处理，这部分工作就是由 Analyzer 来做的。Analyzer 类是一个抽象类，它有多个实现。针对不同的语言和应用需要选择适合的 Analyzer。Analyzer 把分词后的内容交给 IndexWriter 来建立索引。

**IndexWriter**

IndexWriter 是 Lucene 用来创建索引的一个核心的类，他的作用是把一个个的 Document 对象加到索引中来。

**Directory**

这个类代表了 Lucene 的索引的存储的位置，这是一个抽象类，它目前有两个实现，第一个是 FSDirectory，它表示一个存储在文件系统中的索引的位置。第二个是 RAMDirectory，它表示一个存储在内存当中的索引的位置。

**Query**

这是一个抽象类，他有多个实现，比如 TermQuery, BooleanQuery, PrefixQuery. 这个类的目的是把用户输入的查询字符串封装成 Lucene 能够识别的 Query。

**Term**

Term 是搜索的基本单位，一个 Term 对象有两个 String 类型的域组成。生成一个 Term 对象可以有如下一条语句来完成：Term term = new Term(“fieldName”,”queryWord”); 其中第一个参数代表了要在文档的哪一个 Field 上进行查找，第二个参数代表了要查询的关键词。

**TermQuery**

TermQuery 是抽象类 Query 的一个子类，它同时也是 Lucene 支持的最为基本的一个查询类。生成一个 TermQuery 对象由如下语句完成： TermQuery termQuery = new TermQuery(new Term(“fieldName”,”queryWord”)); 它的构造函数只接受一个参数，那就是一个 Term 对象。

**IndexSearcher**

IndexSearcher 是用来在建立好的索引上进行搜索的。它只能以只读的方式打开一个索引，所以可以有多个 IndexSearcher 的实例在一个索引上进行操作。

**Hits**

Hits 是用来保存搜索的结果的。

**Lucene打分**

1. 通过Searcher.explain(Query query, int doc)方法可以查看某个文档的得分的具体构成。   
     
   在Lucene中score简单说是由 tf \* idf \* boost \* lengthNorm计算得出的。   
   **tf(t in d)** 关联到项频率，项频率是指 项 t 在 文档 d 中出现的次数 frequency。默认的实现是：

|  |  |
| --- | --- |
| tf(t in d) = | sqrt(freq) |

1. **idf(t)** 关联到反转文档频率，文档频率指出现 项 t 的文档数 docFreq。docFreq 越少 idf 就越高（物以稀为贵），但在同一个查询下些值是相同的。默认实现：

|  |  |  |
| --- | --- | --- |
| idf(t) = | ln(maxDoc/(docFreq + 1) )+ 1.0 | =ln(maxDoc) -ln(docFreq+1) + 1.0 |

1. **coord(q,d)**评分因子，是基于文档中出现查询项的个数。越多的查询项在一个文档中，说明些文档的匹配程序越高。默认是出现查询项的百分比。
2. **queryNorm(q)**查询的标准查询，使不同查询之间可以比较。此因子不影响文档的排序，因为所有有文档都会使用此因子。默认值：

|  |
| --- |
|  |

还有个问题，为什么一次查询后最大的分值总是1.0呢？   
因为Lucene会把计算后，最大分值超过1.0的分值作为分母，其他的文档的分值都除以这个最大值，计算出最终的得分。

**Lucene 查询过程**

**1. 索引过程：**

**1) 有一系列被索引文件**

**2) 被索引文件经过语法分析和语言处理形成一系列词(Term)** **。**

**3) 经过索引创建形成词典和反向索引表。**

**4) 通过索引存储将索引写入硬盘。**

**2. 搜索过程：**

**a) 用户输入查询语句。**

**b) 对查询语句经过语法分析和语言分析得到一系列词(Term)** **。**

**c) 通过语法分析得到一个查询树。**

**d) 通过索引存储将索引读入到内存。**

**e) 利用查询树搜索索引，从而得到每个词(Term)** **的文档链表，对文档链表进行交，差，并得到结果文档。**

**f) 将搜索到的结果文档对查询的相关性进行排序。**

**g) 返回查询结果给用户。**

我们生活中的数据总体分为两种：**结构化数据** 和**非结构化数据** 。

* **结构化数据：** 指具有固定格式或有限长度的数据，如数据库，元数据等。
* **非结构化数据：** 指不定长或无固定格式的数据，如邮件，word文档等。

当然有的地方还会提到第三种，半结构化数据，如XML，HTML等，当根据需要可按结构化数据来处理，也可抽取出纯文本按非结构化数据来处理。

**非结构化数据又一种叫法叫全文数据。**

按照数据的分类，搜索也分为两种：

* **对结构化数据的搜索** ：如对数据库的搜索，用SQL语句。再如对元数据的搜索，如利用windows搜索对文件名，类型，修改时间进行搜索等。
* **对非结构化数据的搜索** ：如利用windows的搜索也可以搜索文件内容，Linux下的grep命令，再如用Google和百度可以搜索大量内容数据。

对非结构化数据也即对全文数据的搜索主要有两种方法：

一种是**顺序扫描法** **(Serial Scanning)：** 所谓顺序扫描，比如要找内容包含某一个字符串的文件，就是一个文档一个文档的看，对于每一个文档，从头看到尾，如果此文档包含此字符串，则此文档为我们要找的文件，接着看下一个文件，直到扫描完所有的文件。如利用windows的搜索也可以搜索文件内容，只是相当的慢。如果你有一个80G硬盘，如果想在上面找到一个内容包含某字符串的文件，不花他几个小时，怕是做不到。[**Linux**](http://lib.csdn.net/base/linux)下的grep命令也是这一种方式。大家可能觉得这种方法比较原始，但对于小数据量的文件，这种方法还是最直接，最方便的。但是对于大量的文件，这种方法就很慢了。

有人可能会说，对非结构化数据顺序扫描很慢，对结构化数据的搜索却相对较快（由于结构化数据有一定的结构可以采取一定的搜索[**算法**](http://lib.csdn.net/base/datastructure)加快速度），那么把我们的非结构化数据想办法弄得有一定结构不就行了吗？

这种想法很天然，却构成了全文检索的基本思路，也即将非结构化数据中的一部分信息提取出来，重新组织，使其变得有一定结构，然后对此有一定结构的数据进行搜索，从而达到搜索相对较快的目的。

这部分从非结构化数据中提取出的然后重新组织的信息，我们称之**索引** 。

这种说法比较抽象，举几个例子就很容易明白，比如字典，字典的拼音表和部首检字表就相当于字典的索引，对每一个字的解释是非结构化的，如果字典没有音节表和部首检字表，在茫茫辞海中找一个字只能顺序扫描。然而字的某些信息可以提取出来进行结构化处理，比如读音，就比较结构化，分声母和韵母，分别只有几种可以一一列举，于是将读音拿出来按一定的顺序排列，每一项读音都指向此字的详细解释的页数。我们搜索时按结构化的拼音搜到读音，然后按其指向的页数，便可找到我们的非结构化数据——也即对字的解释。

**这种先建立索引，再对索引进行搜索的过程就叫全文检索(Full-text Search)** **。**

下面这幅图来自《Lucene in action》，但却不仅仅描述了Lucene的检索过程，而是描述了全文检索的一般过程。



全文检索大体分两个过程，**索引创建** **(Indexing)** 和**搜索索引** **(Search)** 。

* 索引创建：将现实世界中所有的结构化和非结构化数据提取信息，创建索引的过程。
* 搜索索引：就是得到用户的查询请求，搜索创建的索引，然后返回结果的过程。

于是全文检索就存在三个重要问题：

**1. 索引里面究竟存些什么？(Index)**

**2. 如何创建索引？(Indexing)**

**3. 如何对索引进行搜索？(Search)**

下面我们顺序对每个个问题进行研究。

## 二、索引里面究竟存些什么

索引里面究竟需要存些什么呢？

首先我们来看为什么顺序扫描的速度慢：

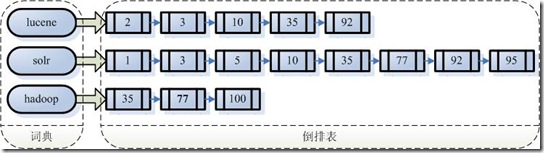
其实是由于我们想要搜索的信息和非结构化数据中所存储的信息不一致造成的。

非结构化数据中所存储的信息是每个文件包含哪些字符串，也即已知文件，欲求字符串相对容易，也即是从文件到字符串的映射。而我们想搜索的信息是哪些文件包含此字符串，也即已知字符串，欲求文件，也即从字符串到文件的映射。两者恰恰相反。于是如果索引总能够保存从字符串到文件的映射，则会大大提高搜索速度。

由于从字符串到文件的映射是文件到字符串映射的反向过程，于是保存这种信息的索引称为**反向索引** 。

反向索引的所保存的信息一般如下：

假设我的文档集合里面有100篇文档，为了方便表示，我们为文档编号从1到100，得到下面的结构



左边保存的是一系列字符串，称为**词典** 。

每个字符串都指向包含此字符串的文档(Document)链表，此文档链表称为**倒排表** (Posting List)。

有了索引，便使保存的信息和要搜索的信息一致，可以大大加快搜索的速度。

比如说，我们要寻找既包含字符串“lucene”又包含字符串“solr”的文档，我们只需要以下几步：

1. 取出包含字符串“lucene”的文档链表。

2. 取出包含字符串“solr”的文档链表。

3. 通过合并链表，找出既包含“lucene”又包含“solr”的文件。

图]倒排表合并过程

看到这个地方，有人可能会说，全文检索的确加快了搜索的速度，但是多了索引的过程，两者加起来不一定比顺序扫描快多少。的确，加上索引的过程，全文检索不一定比顺序扫描快，尤其是在数据量小的时候更是如此。而对一个很大量的数据创建索引也是一个很慢的过程。

然而两者还是有区别的，顺序扫描是每次都要扫描，而创建索引的过程仅仅需要一次，以后便是一劳永逸的了，每次搜索，创建索引的过程不必经过，仅仅搜索创建好的索引就可以了。

**这也是全文搜索相对于顺序扫描的优势之一：一次索引，多次使用。**

## 三、如何创建索引

全文检索的索引创建过程一般有以下几步：

### ****第一步：一些要索引的原文档(Document)。****

为了方便说明索引创建过程，这里特意用两个文件为例：

文件一：Students should be allowed to [**Go**](http://lib.csdn.net/base/go) out with their friends, but not allowed to drink beer.

文件二：My friend Jerry went to school to see his students but found them drunk which is not allowed.

### ****第二步：将原文档传给分次组件(Tokenizer)。****

**分词组件(Tokenizer)会做以下几件事情(** **此过程称为Tokenize)** **：**

**1. 将文档分成一个一个单独的单词。**

**2. 去除标点符号。**

**3. 去除停词(Stop word)** **。**

所谓停词(Stop word)就是一种语言中最普通的一些单词，由于没有特别的意义，因而大多数情况下不能成为搜索的关键词，因而创建索引时，这种词会被去掉而减少索引的大小。

英语中挺词(Stop word)如：“the”,“a”，“this”等。

对于每一种语言的分词组件(Tokenizer)，都有一个停词(stop word)集合。

**经过分词(Tokenizer)** **后得到的结果称为词元(Token)** **。**

在我们的例子中，便得到以下词元(Token)：

“Students”，“allowed”，“[**go**](http://lib.csdn.net/base/go)”，“their”，“friends”，“allowed”，“drink”，“beer”，“My”，“friend”，“Jerry”，“went”，“school”，“see”，“his”，“students”，“found”，“them”，“drunk”，“allowed”。

### ****第三步：将得到的词元(Token)传给语言处理组件(Linguistic Processor)。****

语言处理组件(linguistic processor)主要是对得到的词元(Token)做一些同语言相关的处理。

**对于英语，语言处理组件(Linguistic Processor)** **一般做以下几点：**

**1. 变为小写(Lowercase)** **。**

**2. 将单词缩减为词根形式，如“cars** **”到“car** **”等。这种操作称为：stemming** **。**

**3. 将单词转变为词根形式，如“drove** **”到“drive** **”等。这种操作称为：lemmatization** **。**

**Stemming 和 lemmatization的异同：**

* 相同之处：Stemming和lemmatization都要使词汇成为词根形式。
* 两者的方式不同：
  + Stemming采用的是“缩减”的方式：“cars”到“car”，“driving”到“drive”。
  + Lemmatization采用的是“转变”的方式：“drove”到“drove”，“driving”到“drive”。
* 两者的算法不同：
  + Stemming主要是采取某种固定的算法来做这种缩减，如去除“s”，去除“ing”加“e”，将“ational”变为“ate”，将“tional”变为“tion”。
  + Lemmatization主要是采用保存某种字典的方式做这种转变。比如字典中有“driving”到“drive”，“drove”到“drive”，“am, is, are”到“be”的映射，做转变时，只要查字典就可以了。
* Stemming和lemmatization不是互斥关系，是有交集的，有的词利用这两种方式都能达到相同的转换。

**语言处理组件(linguistic processor)的结果称为词(Term)** **。**

在我们的例子中，经过语言处理，得到的词(Term)如下：

“student”，“allow”，“go”，“their”，“friend”，“allow”，“drink”，“beer”，“my”，“friend”，“jerry”，“go”，“school”，“see”，“his”，“student”，“find”，“them”，“drink”，“allow”。

也正是因为有语言处理的步骤，才能使搜索drove，而drive也能被搜索出来。

### ****第四步：将得到的词(Term)传给索引组件(Indexer)。****

**索引** **组件(Indexer)主要做以下几件事情：**

**1. 利用得到的词(Term)创建一个字典。**

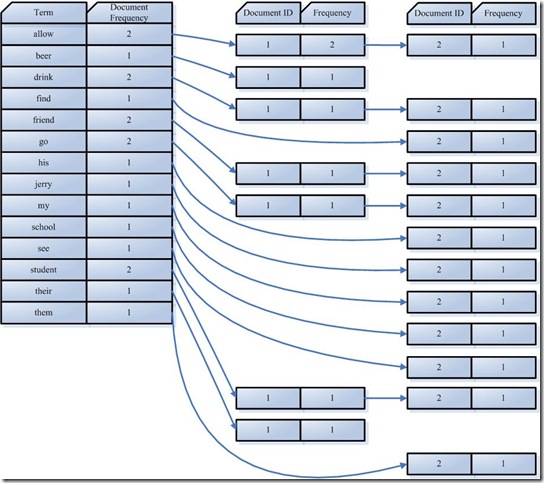
在我们的例子中字典如下：

|  |  |
| --- | --- |
| Term | Document ID |
| student | 1 |
| allow | 1 |
| go | 1 |
| their | 1 |
| friend | 1 |
| allow | 1 |
| drink | 1 |
| beer | 1 |
| my | 2 |
| friend | 2 |
| jerry | 2 |
| go | 2 |
| school | 2 |
| see | 2 |
| his | 2 |
| student | 2 |
| find | 2 |
| them | 2 |
| drink | 2 |
| allow | 2 |

**2. 对字典按字母顺序进行排序。**

|  |  |
| --- | --- |
| Term | Document ID |
| allow | 1 |
| allow | 1 |
| allow | 2 |
| beer | 1 |
| drink | 1 |
| drink | 2 |
| find | 2 |
| friend | 1 |
| friend | 2 |
| go | 1 |
| go | 2 |
| his | 2 |
| jerry | 2 |
| my | 2 |
| school | 2 |
| see | 2 |
| student | 1 |
| student | 2 |
| their | 1 |
| them | 2 |

**3. 合并相同的词(Term)** **成为文档倒排(Posting List)** **链表。**



在此表中，有几个定义：

* Document Frequency 即文档频次，表示总共有多少文件包含此词(Term)。
* Frequency 即词频率，表示此文件中包含了几个此词(Term)。

所以对词(Term) “allow”来讲，总共有两篇文档包含此词(Term)，从而词(Term)后面的文档链表总共有两项，第一项表示包含“allow”的第一篇文档，即1号文档，此文档中，“allow”出现了2次，第二项表示包含“allow”的第二个文档，是2号文档，此文档中，“allow”出现了1次。

到此为止，索引已经创建好了，我们可以通过它很快的找到我们想要的文档。

而且在此过程中，我们惊喜地发现，搜索“drive”，“driving”，“drove”，“driven”也能够被搜到。因为在我们的索引中，“driving”，“drove”，“driven”都会经过语言处理而变成“drive”，在搜索时，如果您输入“driving”，输入的查询语句同样经过我们这里的一到三步，从而变为查询“drive”，从而可以搜索到想要的文档。

## 三、如何对索引进行搜索？

到这里似乎我们可以宣布“我们找到想要的文档了”。

然而事情并没有结束，找到了仅仅是全文检索的一个方面。不是吗？如果仅仅只有一个或十个文档包含我们查询的字符串，我们的确找到了。然而如果结果有一千个，甚至成千上万个呢？那个又是您最想要的文件呢？

打开Google吧，比如说您想在微软找份工作，于是您输入“Microsoft job”，您却发现总共有22600000个结果返回。好大的数字呀，突然发现找不到是一个问题，找到的太多也是一个问题。在如此多的结果中，如何将最相关的放在最前面呢？



当然Google做的很不错，您一下就找到了jobs at Microsoft。想象一下，如果前几个全部是“Microsoft does a good job at software industry…”将是多么可怕的事情呀。

如何像Google一样，在成千上万的搜索结果中，找到和查询语句最相关的呢？

如何判断搜索出的文档和查询语句的相关性呢？

这要回到我们第三个问题：如何对索引进行搜索？

搜索主要分为以下几步：

### ****第一步：用户输入查询语句。****

查询语句同我们普通的语言一样，也是有一定语法的。

不同的查询语句有不同的语法，如SQL语句就有一定的语法。

查询语句的语法根据全文检索系统的实现而不同。最基本的有比如：AND, OR, NOT等。

举个例子，用户输入语句：lucene AND learned NOT [**Hadoop**](http://lib.csdn.net/base/hadoop)。

说明用户想找一个包含lucene和learned然而不包括[**hadoop**](http://lib.csdn.net/base/hadoop)的文档。

### ****第二步：对查询语句进行词法分析，语法分析，及语言处理。****

由于查询语句有语法，因而也要进行语法分析，语法分析及语言处理。

**1. 词法分析主要用来识别单词和关键字。**

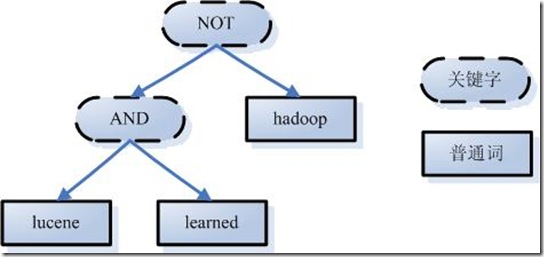
如上述例子中，经过词法分析，得到单词有lucene，learned，hadoop, 关键字有AND, NOT。

如果在词法分析中发现不合法的关键字，则会出现错误。如lucene AMD learned，其中由于AND拼错，导致AMD作为一个普通的单词参与查询。

**2. 语法分析主要是根据查询语句的语法规则来形成一棵语法树。**

如果发现查询语句不满足语法规则，则会报错。如lucene NOT AND learned，则会出错。

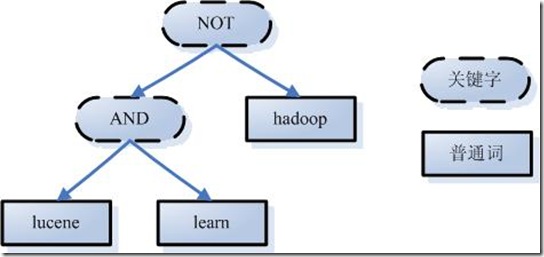
如上述例子，lucene AND learned NOT hadoop形成的语法树如下：



**3. 语言处理同索引过程中的语言处理几乎相同。**

如learned变成learn等。

经过第二步，我们得到一棵经过语言处理的语法树。



### ****第三步：搜索索引，得到符合语法树的文档。****

此步骤有分几小步：

1. 首先，在反向索引表中，分别找出包含lucene，learn，hadoop的文档链表。
2. 其次，对包含lucene，learn的链表进行合并操作，得到既包含lucene又包含learn的文档链表。
3. 然后，将此链表与hadoop的文档链表进行差操作，去除包含hadoop的文档，从而得到既包含lucene又包含learn而且不包含hadoop的文档链表。
4. 此文档链表就是我们要找的文档。

### ****第四步：根据得到的文档和查询语句的相关性，对结果进行排序。****

## MAP容错处理（Fault-Tolerance）

MapReduce 中的容错处理是非常重要的，因为MapReduce 是运行于分布式环境中的，在分布式环境中经常会有机器出现错误，我们不能让个别机器的错误影响到整体。 copyright www.7zhao.net

## Worker 崩溃

Master 通过定期给 Worker 发送心跳（heartbeat）来检测 Worker 是否还在正常工作，如果 Worker 无应答或者是应答有误，我们认定它已经宕机（fail）。如果正在工作的 Worker 宕机了，那么运行在它上面的 map 任务会进行初始化（初始状态为 idle，任务还有其他2种状态，in-progress处理中，completed 已完成），重新被分配到正常的 Worker 上。

如果说 Map Worker 已经完成了一些工作，我们仍然要对运行在它上面的所有任务重新进行分配，这是为什么呢？这里同时可以解决上面的那个问题。因为 Map Worker 处理后的中间结果存在于内存中，或者是 local disk 中，一旦它宕机，这些数据就获取不到了。

但是对于 Reduce Worker，它完成的任务不用重做，因为它处理后的结果是保存在全局存储中的。

如果，在 Map Worker A 宕机之后，它所做的任务被重新分配给了 Map Worker B，后边的 Reduce Worker 会被告知，A 已经宕机，要去 B 去读取数据。

## Master 崩溃

如果说 MapReduce 的 Master 宕机了，又该如何处理呢？ 本文来自去找www.7zhao.net

MapReduce 中的 Master 会定期进行 **checkpoint** 备份，如果 Master 宕机，会根据之前的 checkpoint 进行恢复，但是恢复期间，MapReduce 任务会中断。

## 一些细节问题

#### 1. 考虑用户编写的 reduce 函数是确定的（deterministic，对于同样的输入执行的结果是一样的），如果有多个 Reduce Worker 都执行了一个 Reduce 任务该怎么办？

因为用户的 reduce 函数是 deterministic 的，所以即使有多个 Reduce Worker 都执行了同一个任务，但是它们执行的结果都是一样的，并不影响最后的结果。

#### 2. 如果用户编写的 reduce 函数是不确定（non-deterministic）的呢？

正是因为 reduce 函数是 non-deterministic 的，本来每次执行的结果也不确定，所以更不会产生影响。 www.7zhao.net

#### 3. 我们所需要处理的输入文件是如何保存的？

Input 文件保存于 GFS 中，GFS 会将它们分块保存（每块16MB~64MB），GFS 会对每个文件有3个备份，备份在不同的机器上。

#### 4. Master 是如何分配任务的？

遵循就『近』原则，将任务分配给离任务所保存的位置最『近』的 Worker，这里对『近』的定义是网络层面上的，比如说在同一个交换机下的两个机器就是距离『近』的。

#### 5. MapReduce 是如何做到负载均衡的？

一开始将文件分块时，分为 M 块，远大于 Map Worker 的数量就有助于负载均衡。同时，这样做还有一个好处，就是当一个 Worker 宕机的时候，可以将任务迅速分配开来，分到多个 Worker 上去。如果 M 比较小，有可能当一个 Worker 宕机时，它的任务不够分配到剩下的 Worker 中，会有 Worker 闲置。

#### 6. 如何解决 straggler 问题（其他 Worker 都已经完成了自己的任务，但是有一个异常慢的机器，它还有任务没完成，拖慢了整体的速度）？

MapReduce 有一种机制应对这种情况：MapReduce 会对未完成的任务（in-progress） 定时执行备份执行操作（即，把这些正在某些 Worker 上执行但未完成的任务再次分配给其他 Worker 去执行），不论这个任务被哪个 Worker 完成都会被标记为已完成。

#### 7. 如果在 Map 任务中有一个 key 特别多，可能会拖慢整个网络的速度，该怎么办？（例如，在字数统计的例子中，the 这个词的数量特别多）

MapReduce 给用户提供了一个 Combiner 函数，这个函数可以将结果在发送到网络之前进行合并，例如发送键值对<”by”, 3>。

COMBINER

目标：

Mapreduce中的Combiner就是为了避免map任务和reduce任务之间的数据传输而设置的，[**hadoop**](http://lib.csdn.net/base/hadoop)允许用户针对map task的输出指定一个合并函数。即为了减少传输到Reduce中的数据量。它主要是为了削减Mapper的输出从而减少网络带宽和Reducer之上的负载。

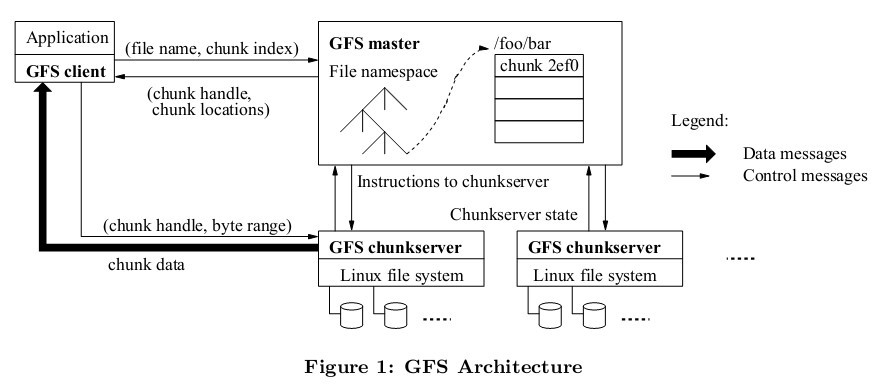
BIGTABLE

一个Bigtable 集群是一系列运行Bigtable软件的进程。每一个集群都有一组tables。Bigtable中的表是稀疏的、分布式、持久的多维有序map。其数据有三个维度：行、列、时间戳。

(row:string, column:string, time:int64) → string

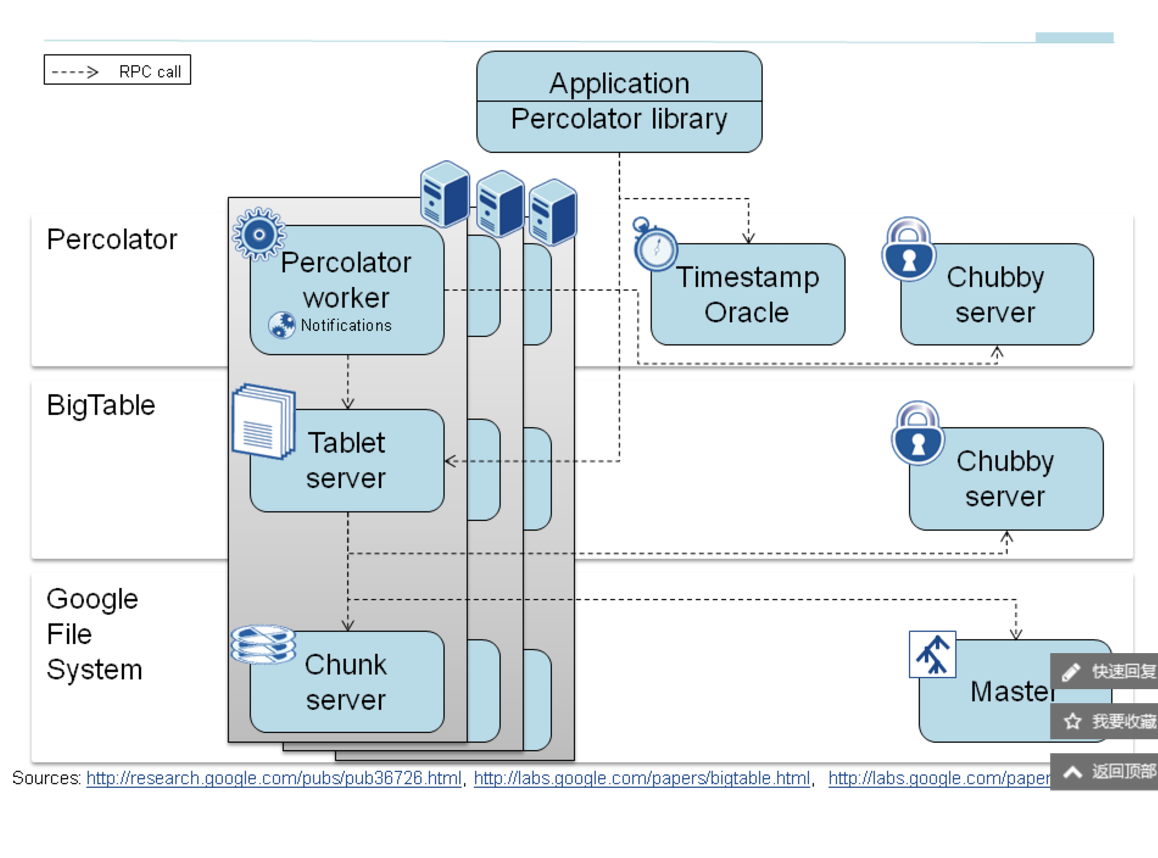
    我们称由一个特定行键、列键、时间戳指定的部分为一个单元(cell)。多行组合起来形成负载平衡的基本单元，多列组合起来形成访问控制和资源分配的基本单元。

GFS包括一个master结点（元数据服务器），多个chunkserver（数据服务器）和多个client（运行各种应用的客户端）。在可靠性要求不高的场景，client和chunkserver可以位于一个结点。图1是GFS的体系结构示意图，每一结点都是普通的[**Linux**](http://lib.csdn.net/base/linux)服务器，GFS的工作就是协调成百上千的服务器为各种应用提供服务。



* chunkserver提供存储。GFS会将文件划分为定长数据块，每个数据块都有一个全局唯一不可变的id（chunk\_handle），数据块以普通Linux文件的形式存储在chunkserver上，出于可靠性考虑，每个数据块会存储多个副本，分布在不同chunkserver。
* GFS master就是GFS的元数据服务器，负责维护文件系统的元数据，包括命名空间、访问控制、文件-块映射、块地址等，以及控制系统级活动，如垃圾回收、负载均衡等。
* 应用需要链接client的代码，然后client作为代理与master和chunkserver交互。master会定期与chunkserver交流（心跳），以获取chunkserver的状态并发送指令。

图1还描述了应用读取数据的流程。1.应用指定读取某个文件的某段数据，因为数据块是定长的，client可以计算出这段数据跨越了几个数据块，client将文件名和需要的数据块索引发送给master；2.master根据文件名查找命名空间和文件-块映射表，得到需要的数据块副本所在的地址，将数据块的id和其所有副本的地址反馈给client；3.client选择一个副本，联系chunkserver索取需要的数据；4.chunkserver返回数据给client。



 Web 5月27

讨论课：1，2，4.3

Meta data data+location

sample：error+bias

Big data 的问题：

1 storage

2 I/O bw

3 processing->spark

Big data- >关联关系 不是因果关系

google大数据TensorFlow jupyter

Outofmem->heap

Stackoverflow->function

Source code:执行js解析返回值

对象存储不可靠 都有副本

负载均衡 将压力很大的数据库一分为二 一半为M 一半为S

Mongodb平衡文件的字节量 应该负载均衡请求量 当数据变热时 切+迁移 变冷时 合并

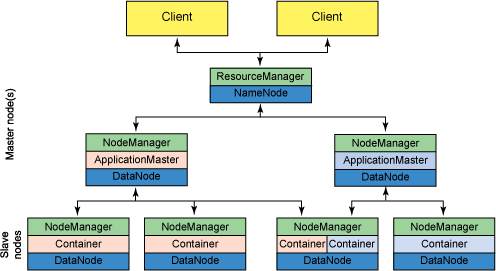
不适用的场景：bigtable 因果关系

MAPREDUCE data locality

如果map task运行的节点上刚好有输入数据的冗余备份（replica），此时hadoop可以发挥出它的最大性能（因为不需要额外的数据传输从而节省了时间），这叫做data locality optimization（数据本地最优化）。

MAPREDUCE和YARN对比

mapreduce

1. 首先用户程序 (JobClient) 提交了一个 job，job 的信息会发送到 Job Tracker 中，Job Tracker 是 Map-reduce 框架的中心，他需要与集群中的机器定时通信 (heartbeat), 需要管理哪些程序应该跑在哪些机器上，需要管理所有 job 失败、重启等操作。
2. TaskTracker 是 Map-reduce 集群中每台机器都有的一个部分，他做的事情主要是监视自己所在机器的资源情况。
3. TaskTracker 同时监视当前机器的 tasks 运行状况。TaskTracker 需要把这些信息通过 heartbeat 发送给 JobTracker，JobTracker 会搜集这些信息以给新提交的 job 分配运行在哪些机器上。上图虚线箭头就是表示消息的发送 - 接收的过程。
4. **主要的问题集中如下：**
5. JobTracker 是 Map-reduce 的集中处理点，存在单点故障。
6. JobTracker 完成了太多的任务，造成了过多的资源消耗，当 map-reduce job 非常多的时候，会造成很大的内存开销，潜在来说，也增加了 JobTracker fail 的风险，这也是业界普遍总结出老 Hadoop 的 Map-Reduce 只能支持 4000 节点主机的上限。
7. 在 TaskTracker 端，以 map/reduce task 的数目作为资源的表示过于简单，没有考虑到 cpu/ 内存的占用情况，如果两个大内存消耗的 task 被调度到了一块，很容易出现 OOM。
8. 在 TaskTracker 端，把资源强制划分为 map task slot 和 reduce task slot, 如果当系统中只有 map task 或者只有 reduce task 的时候，会造成资源的浪费，也就是前面提过的集群资源利用的问题。
9. 回想一下，由于限制了扩展以及网络开销所导致的某些故障模式，MRv1 JobTracker 和 TaskTracker 方法曾是一个重要的缺陷。这些守护程序也是 MapReduce 处理模型所独有的。为了消除这一限制，JobTracker 和 TaskTracker 已从 YARN 中删除，取而代之的是一组对应用程序不可知的新守护程序。
10. [[](http://cms.csdnimg.cn/article/201312/18/52b1541a825ca.jpg)](http://cms.csdnimg.cn/article/201312/18/52b1541a825ca.jpg)
11. **图 2. YARN 的新架构**
12. YARN 分层结构的本质是 ResourceManager。这个实体控制整个集群并管理应用程序向基础计算资源的分配。ResourceManager 将各个资源部分（计算、内存、带宽等）精心安排给基础 NodeManager（YARN 的每节点代理）。ResourceManager 还与 ApplicationMaster 一起分配资源，与 NodeManager 一起启动和监视它们的基础应用程序。在此上下文中，ApplicationMaster 承担了以前的 TaskTracker 的一些角色，ResourceManager 承担了 JobTracker 的角色。
13. ApplicationMaster 管理一个在 YARN 内运行的应用程序的每个实例。ApplicationMaster 负责协调来自 ResourceManager 的资源，并通过 NodeManager 监视容器的执行和资源使用（CPU、内存等的资源分配）。请注意，尽管目前的资源更加传统（CPU 核心、内存），但未来会带来基于手头任务的新资源类型（比如图形处理单元或专用处理设备）。从 YARN 角度讲，ApplicationMaster 是用户代码，因此存在潜在的安全问题。YARN 假设 ApplicationMaster 存在错误或者甚至是恶意的，因此将它们当作无特权的代码对待。
14. NodeManager 管理一个 YARN 集群中的每个节点。NodeManager 提供针对集群中每个节点的服务，从监督对一个容器的终生管理到监视资源和跟踪节点健康。MRv1 通过插槽管理 Map 和 Reduce 任务的执行，而 NodeManager 管理抽象容器，这些容器代表着可供一个特定应用程序使用的针对每个节点的资源。YARN 继续使用 HDFS 层。它的主要 NameNode 用于元数据服务，而 DataNode 用于分散在一个集群中的复制存储服务。
15. 要使用一个 YARN 集群，首先需要来自包含一个应用程序的客户的请求。ResourceManager 协商一个容器的必要资源，启动一个 ApplicationMaster 来表示已提交的应用程序。通过使用一个资源请求协议，ApplicationMaster 协商每个节点上供应用程序使用的资源容器。执行应用程序时，ApplicationMaster 监视容器直到完成。当应用程序完成时，ApplicationMaster 从 ResourceManager 注销其容器，执行周期就完成了。
16. 通过这些讨论，应该明确的一点是，旧的 Hadoop 架构受到了 JobTracker 的高度约束，JobTracker 负责整个集群的资源管理和作业调度。新的 YARN 架构打破了这种模型，允许一个新 ResourceManager 管理跨应用程序的资源使用，ApplicationMaster 负责管理作业的执行。这一更改消除了一处瓶颈，还改善了将 Hadoop 集群扩展到比以前大得多的配置的能力。此外，不同于传统的 MapReduce，YARN 允许使用 Message Passing Interface 等标准通信模式，同时执行各种不同的编程模型，包括图形处理、迭代式处理、机器学习和一般集群计算。

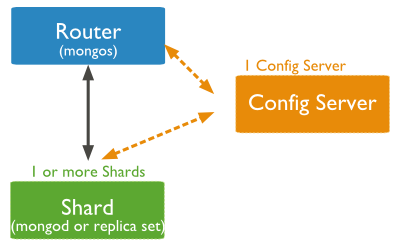
Mongodb

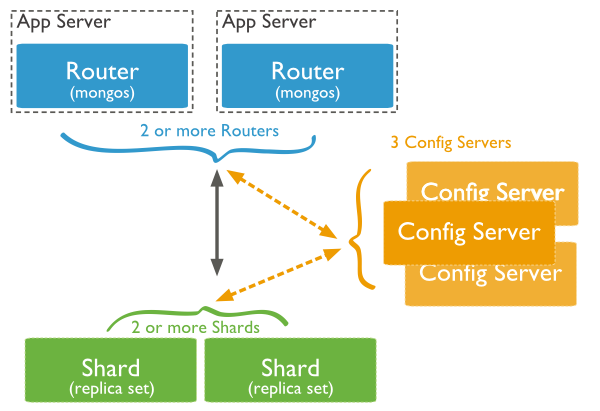
harded cluster分片集群有以下几个组件：shards，query routers,config servers.

shards: 用来存储数据，为这个分片集群提供高可用和数据一致性。在一个生产环境中，每个shard都是一个replica set。

query routers： 或者是mongos实例，用于与应用程序交互，将请求转发到后端的shards，然后将请求结果返回给客户端。一个分片集群可以有多个query router即mongos实例用于分摊客户端的请求压力。如果使用多个mongos实例，可以使用HAProxy或者LVS等代理来转发客户端请求到后端的mongos，必须要配置成client affinity模式保证来自同一个客户端的请求转发到后端相同的mongos.通常会将mongos实例部署到应用服务器上。

config servers： 用于存储分片集群的元数据。这些元数据包含整个集群的数据集合data sets与后端shards的对应关系。query router使用这些元数据来将客户端请求定位到后端相应的shards。生产环境的分片集群正好有3个config servers。config servers里的数据非常重要，如果config servers全部挂掉，整个分片集群将不可用。在生产环境中，每个config server都必须放置到不同的服务器上，并且每个分片集群的config server不能共用，必须分开部署。

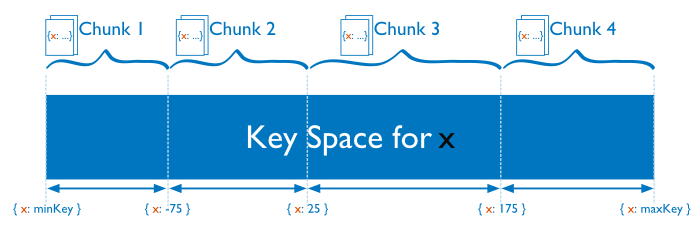




MongoDB Sharding是在Collection即集合层面来分布存储数据的，Sharding依据shard key来讲一个集合的数据来分布存储。

为了将一个集合的数据进行分片，首先需要选择一个shard key。一个shard key可以是存在于一个集合中每个文档的索引字段或者符合索引字段。MongoDB将这个shard key的值切分成多个数据块，然后将这些数据块均匀分布到后端的shard上。MongoDB使用range based partitioning 或者 hash based partitionning来讲一个shard key的值进行切分。shard key一旦选择好是不能变更的。

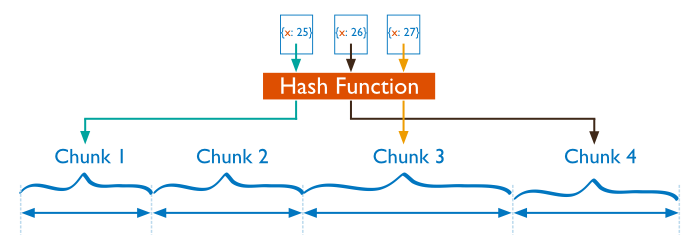
Range Based Sharding



Given a range based partitioning system, documents with “close” shard key values e likely to be in the same chunk, and therefore on the same shard.

Hash Based Sharding

对于hash based partitionning，MongoDB会先计算一个字段值得哈希值，然后使用这些哈希值来创建数据块。



With hash based partitioning, two documents with “close” shard key values are unlikely to be part of t same chunk. This ensures a more random distribution of a collection in the cluster.

Performance Distinctions between Range and Hash Based Partitioning

Range based partitioning支持更有效的范围查询。对于一个shard key给定一个范围查询，query router可以更容易地判断将请求只路由到包含相应数据库的shard上。

Range based partitioning可能会导致数据分布不均，这样会对sharding产生负面作用，比如会出现大部分请求被分发到同一个shard的情况发生。

Hash based partitioning可以确保数据平均分布，但是这样会导致经过哈希处理的值在各个数据块和shard上随机分布，进而使制定的范围查询range query不能定位到某些shard而是在每个shard上进行查询。

Customized Data Distribution with Tag Aware Sharding

MongoDB允许使用tag aware sharding来根据shard key的范围创建并关联一些tag到后端的shards。主要用于同一个分片集群数据分布到多个数据中心的情况。

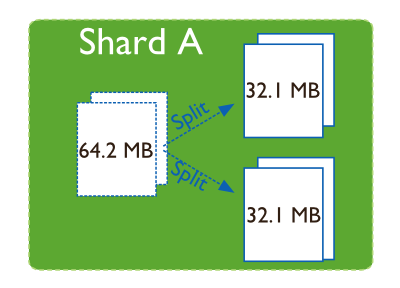
Maintaining a Balanced Data Distribution

随着数据的增加或者是服务器的增加都会导致整个分片集群的数据分布不均衡，比如一个shard比其他shard上的数据库chunk明显多了很多，或者一个数据块chunk的大小明显比其他chunk大很多。

MongoDB使用两个后台进程来确保一个均衡的分片集群，它们分别是splitting和balancer.

Splitting

Splitting是一个防止chunk变得太大的后台进程，当一个chunk大小超过了指定的大小，MongoDB将会把这个chunk分成两半。插入和更新操作都会触发split。



Balancing

balancer是一个用于管理chunk迁移的后台进程。

当一个分片集群中一个分片集合的数据分布不均衡时，balancer进程会把拥有最多chunk的shard上的chunk迁移到拥有最少chunk的shard上，直到这个集合的数据分布均衡为止。例如，集合user在shard 1上拥有100个chunk，在shard2上拥有50个chunk，balancer进程会将shard1上的chunk迁移到shard2，直到两个shard上的chunk数量保持均衡位置。

向一个分片集群中添加或删除shard都会影响整个集群的均衡性。

Primary shard

每个数据库都会有一个primary shard存储这个数据库中所有未被分片的集合数据。



MongoDB Sharding技术的应用场景：

A.如果数据集data set大小将要或者已经超过了单个MongoDB实例的容量大小。

B.活动的工作集working set大小将要超过最大物理内存大小

C.单个MongoDB实例无法满足频繁的写操作。

如果以上三种情况没有满足，不需要部署sharding，只会增加复制度，同时在设计数据模型时，也要考虑到以后作分片的情况。

课堂反转1

bigtable数据结构

bigtable是稀疏的，分布式的，持久化的，多维的排序映射。

行键 列键 时间戳 **(row:string, column:string, time:int64)→string**来表示一条键值对记录。

Bigtable的表会根据行键自动划分为片（tablet），片是负载均衡的单元。

行是表的第一级索引，

列是第二级索引为了方便管理，列被分为列族，一个列族里的列一般存储相同类型的数据。一行的列族很少变化，但是列族里的列可以随意添加删除。

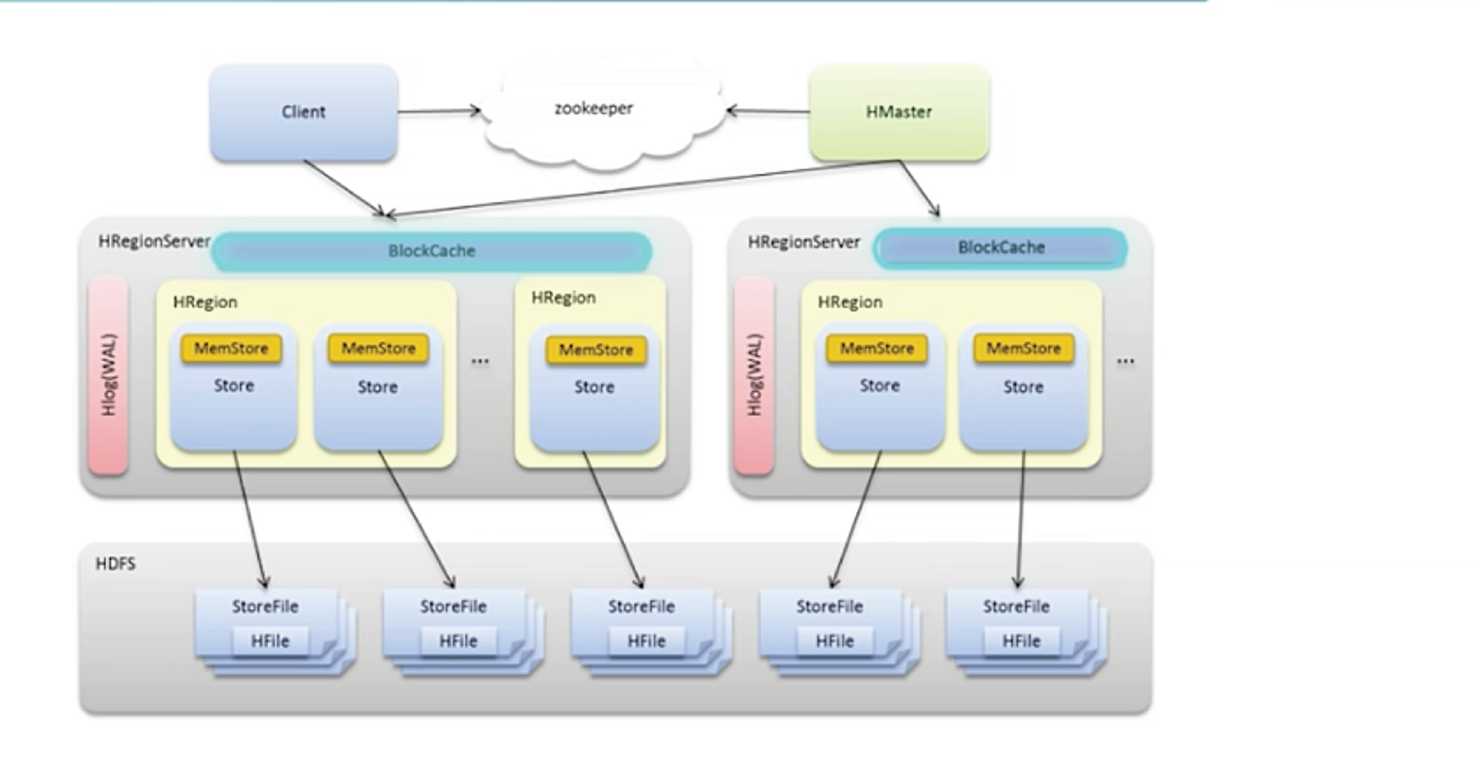
时间戳是第三级索引。Bigtable允许保存数据的多个版本，版本区分的依据就是时间戳。

Bigtable存储稀疏数据方便，键值对的存储方式，没有值的数据不会被存储。

键-》行键 列族 列 时间戳

tag是额外信息，如访问控制。

**存储方式**

****

HRegionServer的主要组成部分是RegionHRegionServer还包括了HLog和BlockCache

一个Region由多个Store组成 每一个列族对应一个Store HBase的的文件又称为store file

它是HFile的格式 被直接存放在HDFS中 HLog又称为write-ahead log也就是WAL

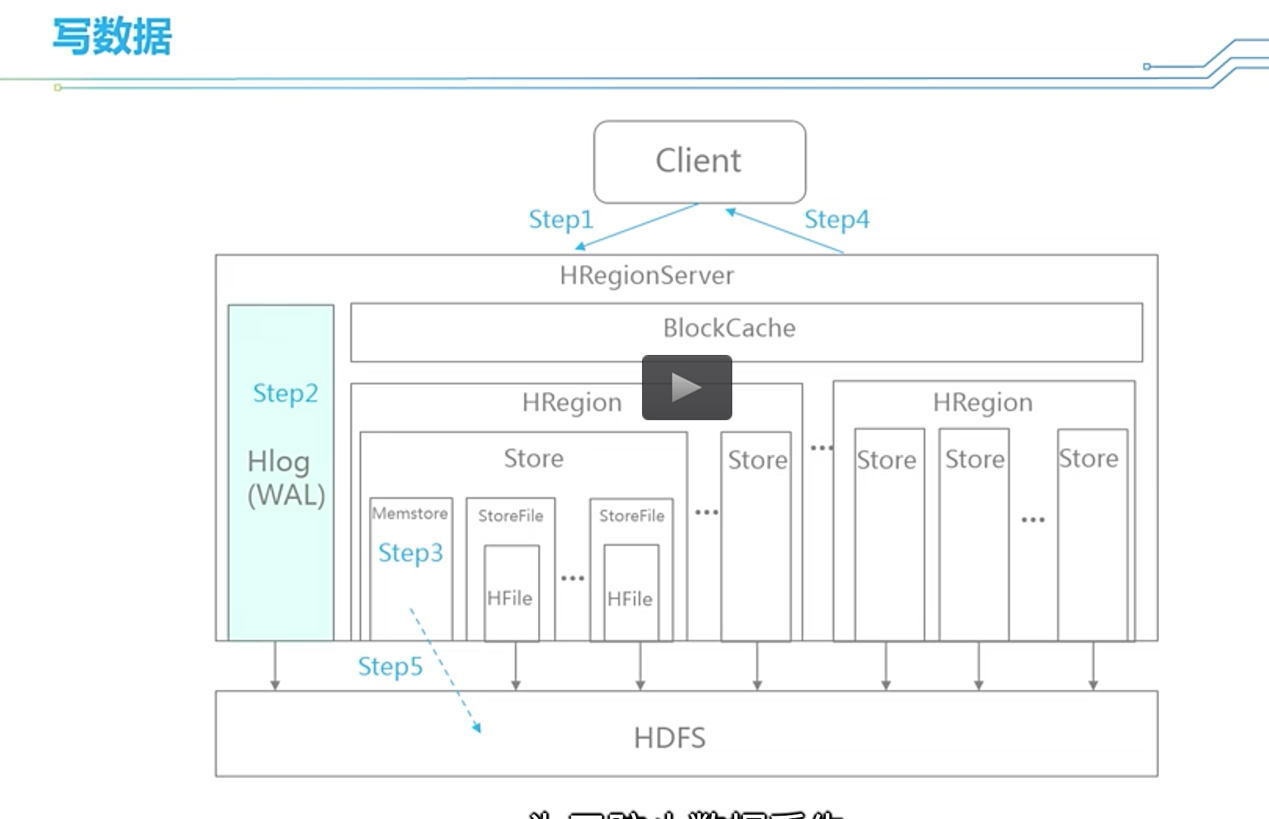
Hbase写入

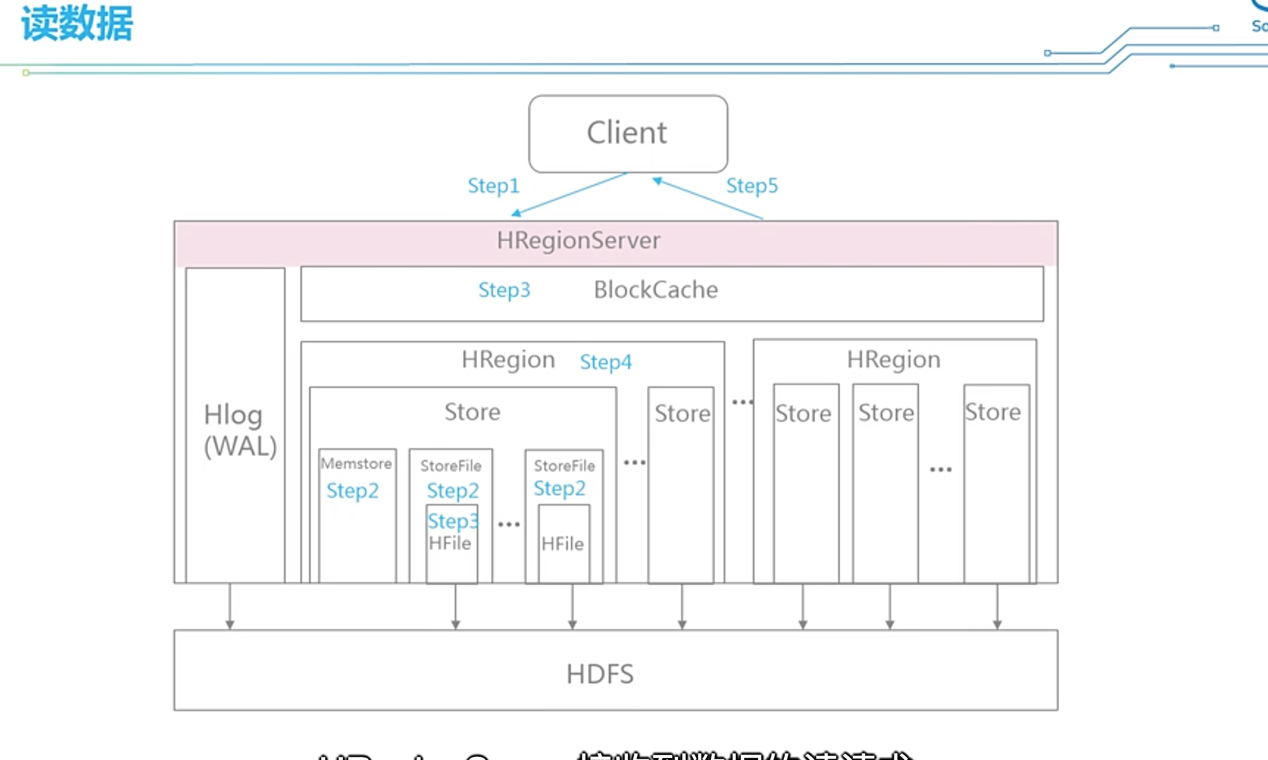
因为HBase新写入的数据 会被保存在内存中 也就是这个memstore中 如果集群发生故障数据就会丢失 所以为了防止数据丢失 所有被写入memstore中的数据 都会被先写入到这些HLog中 这样遇到集群重启 HRegionServer会重新运行 保存在HLog中的记录 恢复集群数据 接着数据被写入memstore 随后写入请求被返回到客户端 随着memstore被写满

memstore中的数据 会被异步的写入磁盘中 生成一个storefile PUT实际上是一个upsert

即update加insert 从用户的角度看 就是如果一条记录不存在 就直接写入该记录若记录存在就更新该记录DELETE是HBase中的删除操作添加一条DELETE的记录即delete marker

在读取的时候hbase的scanner会将删除的数据 从结果中过滤掉 懒删除





查询数据的过程HRegionServer接收到数据的读请求 通过Region将读请求 发送到相关的Store StoreScanner打开memstore和相应的StoreFile 在读取StoreFile的数据块时

Scanner会尝试从BlockCache中读取数据 若无法查到 则从StoreFile中读取数据

并存入blockcache中从memstore和storefiles中查询得到的数据会被进行merge-sort

并加入到最终的结果中返回到客户端GET是读取某条记录Scan是扫描一段连续的记录

之前我们有提到列族最好不要太多

但是在读取时需要访问很多的磁盘文件所以hbase会对这些磁盘上的store file进行合并

生成一个新的更大的文件当一个store中的storefile达到某个阀值的时候默认是3

hbase便会选择部分 或者全部文件进行合并每个Store都拥有自己的memstore

当一个region拥有的所有memstore的数据量之和达到一个阀值的时候默认是128MB

就会触发memstore的flush一个列族对应一个store一个store对应一个storefile

如果一个Region拥有的Store过多会造成每个写入磁盘的store file过小随着小文件的增多

也会更加频繁的触发compaction造成更多额外的I/O负担所以不建议用户设置过多的列族

compaction：

compaction会优先选择 老的和小的文件来合并compaction又分为两种一种叫minor compaction就是合并部分文件 一种叫major compaction就是合并所有文件 在major compaction中delete marker会被删除

Hbase不适用的场景

HBase并不擅长处理表之间的关系 因为没有外键关联无法处理如Join之类的操作

但是HBase提供了查询接口 用户如果有需要 可以自己实现相应的join功能

HBase虽然可以保证同一region数据的读写的事务但并不能处理传统意义上的事务（分布式事务）一系列写操作中访问的数据可能位于不同的分区服务器，这样的事务就变成[分布式事务](https://www.baidu.com/s?wd=%E5%88%86%E5%B8%83%E5%BC%8F%E4%BA%8B%E5%8A%A1&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkPW0vujfdP17bm1KBPH0Y0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH6vPHnzPHckPj64P1ckrjfd)，实现原子性需要彼此协调，而协调是耗费时间的，每台机器在一个大事务过程中必须依次确认，这就需要一种协议确保一个事务中没有任何一台机器写操  
作失败。这种协调是昂贵的，会增加[延迟时间](https://www.baidu.com/s?wd=%E5%BB%B6%E8%BF%9F%E6%97%B6%E9%97%B4&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkPW0vujfdP17bm1KBPH0Y0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH6vPHnzPHckPj64P1ckrjfd)，关键问题是，当协调没有完成时，其他操作是不能读取事务中写操作结果的，这是因为事务的all-or-nothing原理导致，万一协调过程发现某个写操作不能完成，那么需要将其他写操作成功的进行回滚。针对[分布式事务](https://www.baidu.com/s?wd=%E5%88%86%E5%B8%83%E5%BC%8F%E4%BA%8B%E5%8A%A1&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkPW0vujfdP17bm1KBPH0Y0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH6vPHnzPHckPj64P1ckrjfd)的分布式协调对整体数据库性能有严重影响。

另外HBase中只有主键索引 不像传统关系型数据那样 拥有二级索引和全文索引

对于非主键的查询并不高效 不得不进行全表扫描 当然现在有很多种途径去实现二级索引和全文索引但是在HBase的代码中并没有实现

对大对象的支持 传统关系型数据库中有BLOB用来存储大对象HBase对于存储大于50MB的大对象并不擅长往往存储几个对象就需要把内存中的数据写入磁盘过多的文件会频繁的触发compaction并会产生大量额外的I/O影响集群的性能

主键递增的数据 这样会造成一段时间内 某个region成为写入热点 影响数据写入的吞吐量

HDFS主要是适用于以下三种场景 首先是我们需要存储TB或者PB级别的大数据集

并且要支持单个的大文件 其次HDFS假设我们的使用模式是一次写入、多次读取

而且是顺序地读取大量数据的情况HDFS的随机访问性能并不高最后HDFS对数据的可靠性提供保障就不要求我们使用高可靠的服务器在部分的节点出现故障的时候可以继续使用

下面这几个场景就不适合使用HDFS

比如HDFS注重的是高吞吐量 而非低延迟 因此要求低延迟的数据访问 就不应该使用HDFS

另外HDFS也不能很好地 应对大量小文件的情况因为所有的文件的元数据 都需要保存在内存中 因此HDFS可以支持的文件总数量 是受内存限制的

最后HDFS不支持单个文件的并发写入而且当文件生成以后 对文件的修改也只支持Append操作

Block概念

Block概念其实不是HDFS所特有的 对于磁盘而言也有Block的概念 那么对磁盘来说一个Block代表了最小的读写单元 通常为512个字节 我们常见的文件系统也有Block的概念

大小通常在KB级别并且是磁盘Block的整数倍HDFS也有自己的Block在HDFS当中

每个文件会被切分为多个Block每个Block是一个独立的存储单元 这样做的好处是单个文件的大小 可以超出任意一台节点的磁盘容量另外这样做也便于HDFS对文件 进行管理和复制

与常见的文件系统不同的是HDFS的Block的大小要大的多 默认为128兆

较大的Block可以带来一些好处

首先是在读取数据的时候 较大的Block减少了磁盘寻道的时间开销 这样读取一个大文件的时候 读取的速率就能近似于磁盘的传输速率 另外较大的Block可以减少Block的总数 就像我们之前所说的Block信息也要存储在内存中 因此较少的Block个数 可以减轻内存的压力

如果Block的大小设置得太大

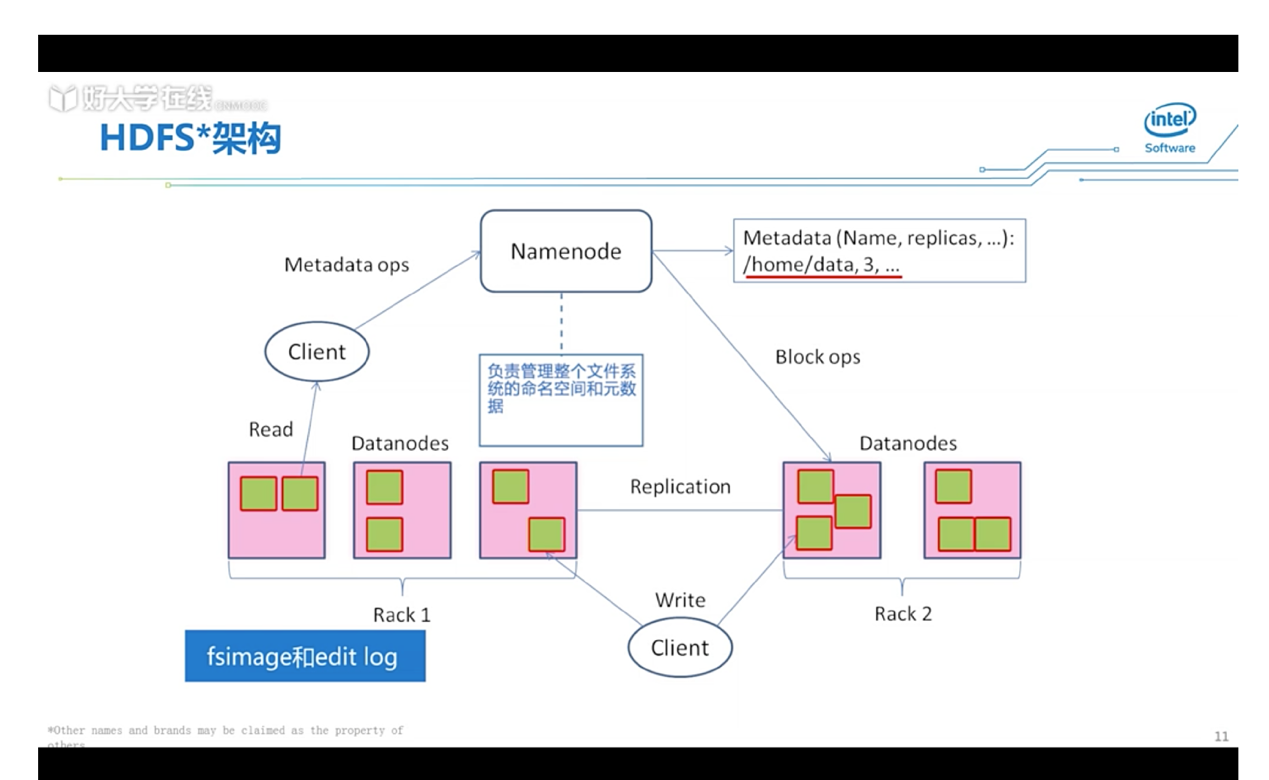
也会有一些缺点 比如在执行Map Reduce任务的时候 通常一个Map Task 会处理一个Block的数据 这就意味着较大的Block 会导致较少的Map Task 这并不利于充分利用集群的计算资源 因此我们在设置HDFS Block大小的时候 需要做一定的取舍

Block的size不能太大，如果只读一个小文件，需要把整个block都load进来，I/O的效率低

太小的话大文件寻址负担重，并且namenode种需要存大量的元数据。

Block对于HDFS而言是一个逻辑的概念 如果一个文件的大小小于的Block那么HDFS还是会为这个文件分配一个Block但是并不代表这个文件会 实际占用一个Block的存储空间

下面我们来介绍一下HDFS的架构



HDFS采用了master work的模式 一个HDFS集群 由一个NameNode和多个DataNode组成 NameNode负责管理 整个文件系统的命名空间和元数据 是HDFS的master节点

比如这个图中展示的NameNode记录了home下面data这个文件的复制份数为3

元数据文件的型式保存在NameNode的本地磁盘上主要包含了两个文件fsimage和edit log

fsimage保存了整个文件系统的命名空间而edit log记录了在集群运行期间对文件系统的所有修改操作比如创建了新的文件重命名了已有的文件等另外NameNode还需要管理

所有的DataNode以及一个文件所有Block的存储位置最后NameNode还要协调

客户端对数据的访问

我们再来看一下DataNode

Datanode属于HDFS的work节点负责将HDFS当中的数据存储在本地的文件系统里

图中绿色的部分就代表了Datanode所存储的各个Block

Datanode**要定时向Namenode汇报状态** 自己存了哪些文件的哪些block，同步备份的状态如何。另外Datanode还要响应用户的读写请求 客户端就是用户访问HDFS的接口

例如客户端可以向Namenode请求读取某个文件或者写入新的文件等

HDFS是如何保证数据的可靠性的

这里所讲的可靠性是指 当集群中部分的节点出现故障的时候HDFS要保证用户的数据不会丢失 并且可以继续被访问 可以想到 最简单的保证可靠性的方法就是对数据进行复制

HDFS当中每一个Block会有多个副本默认为三份 这些副本分布的存储在 不同的Datanode上 这样部分Datanode出现故障 就不会导致数据丢失

但是因为每一份数据都要被复制多份 因此会有较大的存储开销为了解决这一缺陷

HDFS又引入了Erasure Coding的机制

Erasure Coding通过 对数据Block进行编码 可以得到校验的Parity Block Parity Block和数据Block分布的存储在集群中 当部分数据丢失的时候 我们可以通过剩余的Block来恢复丢失的数据 采用Erasure Coding的好处就是 可以在保证数据可靠性的同时 大幅的降低存储开销 当然另一方面 这需要一些额外的编解码的计算

假设一个文件由ABC3个Block组成 每一个Block会有三个副本HDFS会将这些副本分布的存储在 不同的Datanode上 如果HDFS拥有集群的网络拓扑信息 那么还会将这些Block尽量的分布在 不同的机柜上 这些措施都是为了保证 部分的Datanode出现故障时用户的数据不会丢失

在这种机制下 假如我们需要存储的数据为1T那么实际我们需要3T的存储空间额外的存储空间开销为200%

那么我们再来看一下Erasure Coding最常用的是Reed-Solomon算法

其他的算法还包括像异或 以及HitchHiker等

我们就以Reed-Solomon算法为例假设我们有K个数据Block经过编码后可以得到M个校验Block我们把这些校验Block称为parity Block我们将K加M个Block分布的存储在集群中

在节点出现故障的时候如果丢失的Block数量小于等于M我们就可以通过剩余的Block来做解码并还原出所有的数据这里需要说明的是丢失的Block既可以是数据Block也可以是parity BlockK和M的值可以比较灵活的进行配置通常情况下我们选K等于6、M等于3

这就意味着Erasure Coding与复制的机制一样可以承受最多3台Datanode出现故障

但额外的存储开销只有50%远远低于我们之前讲的复制

HDFS的存储布局 在引入Erasure Coding之前HDFS的Block都是连续存储的

如这张图中上半部门展示的每一个Block都连续的存储在一台Datanode上

为了适应Erasure Coding的机制 我们引入了分条的存储布局 目前所有使用

Erasure Coding存储的文件 都是要按照分条来存储的 在分条的布局下数据会被切分为较小的单元格 我们可以看这个图下半部分每个单元格的大小是1M6个单元格就组成了一个条带

并且存储在6台Datanode上相对于连续存储来说分条存储的一个优势是说我们可以进行并发的数据读写但是它也有一些缺点

比如像MapReduce这样的计算 会失去数据本地化

fsimage文件其实是[Hadoop](https://www.iteblog.com/archives/tag/hadoop/)文件系统元数据的一个永久性的检查点，其中包含[Hadoop](https://www.iteblog.com/archives/tag/hadoop/)文件系统中的所有目录和文件idnode的序列化信息；

、edits文件存放的是[Hadoop](https://www.iteblog.com/archives/tag/hadoop/)文件系统的所有更新操作的路径，文件系统客户端执行的所以写操作首先会被记录到edits文件中。

HDFS如何保证可用性

Namenode是一个单点故障因此如果Namenode出现错误无法提供服务的时候整个集群都无法访问

Namenode的可用性 主要面临两种威胁一个是Namenode出现故障 导致保存的元数据丢失

这种情况下集群将会完全不可用甚至导致所有的用户数据都会丢失

另一种情况是Namenode运行一段时间以后需要重启重启的原因可能是日常的维护等等

但是这个时候会有比较大的重启开销导致用户无法使用服务

我们可以让Namenode将原数据备份到多个文件系统当中比如同时写入本地和网络存储

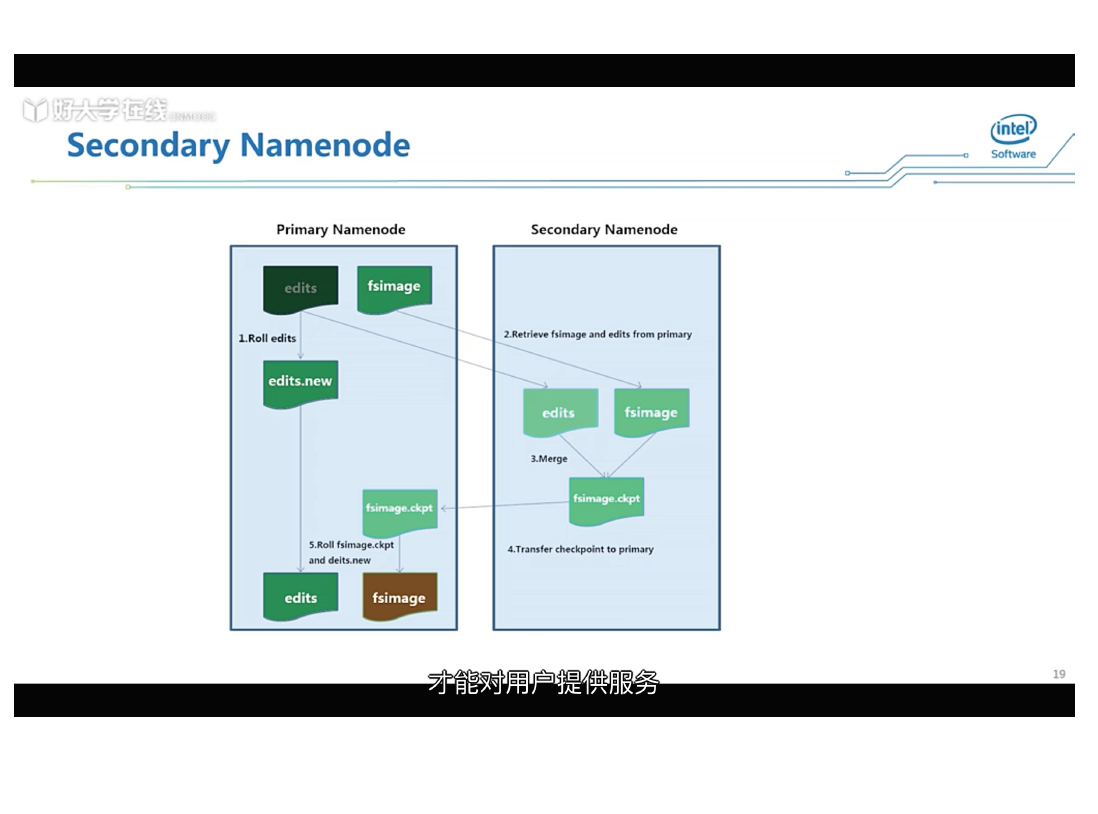
另外HDFS还提供了Secondary Namenode和Standby Namenode

Namenode在运行的过程当中会将用户所有的修改保存在edit log中这就会导致edit log

随着Namenode的运行而不断增长当Namenode需要重启的时候需要读取fsimage

然后将edit log中的修改记录合并进去这个时候如果edit log的内容太多就会导致Namenode有很长的启动时间我们可以用Secondary Namenode来解决这个问题

通常Secondary Namenode运行在跟主Namenode不同的节点上并定期的对主Namenode的fsimage和edit log进行合并保证edit log的内容不会过长



首先Secondary Namenode会通知主Namenode来更换edit log

主Namenode将当前使用的edit log文件关闭并创建新的edit log来记录

后续的修改操作Secondary Namenode把原来的edit log和fsimage拷贝到本地

然后进行合并 得到fsimage checkpoint然后Secondary Namenode把

这份checkpoint拷贝回主Namenode主Namenode就可以用这个checkpoint

来替换原来的fsimage了这样就能保证在主Namenode上edit log的长度不会过长

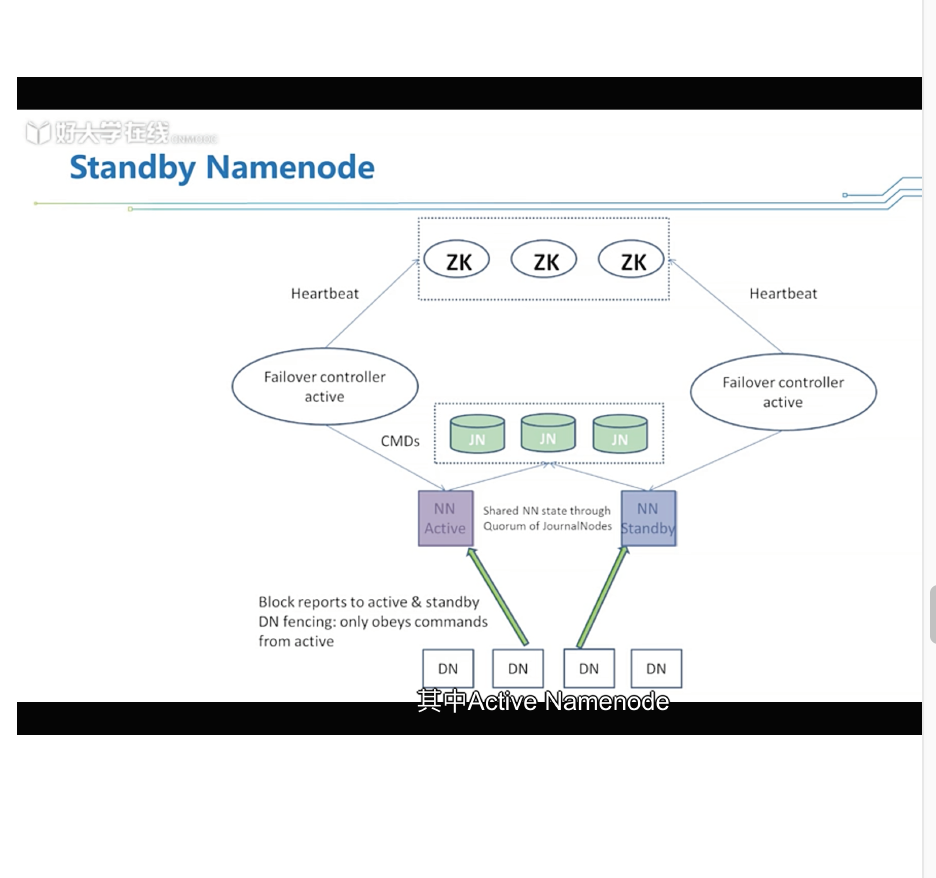
Secondary Namenode虽然可以解决主Namenode的启动开销问题但是并不能真正带来高可用性因为当主Namenode出现故障的时候我们仍然需要重新启动一个Namenode

并把原来备份的fsimage和edit log加载到内存中然后还要等足够多的Datanode

向其汇报状态之后新的Namenode才能对用户提供服务在一个规模比较大的集群上

这个过程通常需要至少三十分钟时间

为了克服这个问题我们在HDFS当中又引入了Standby Namenode



使用Standby Namenode的时候 集群中会同时存在两台Namenode

在任意一个时间点只有一台Namenode可以对外提供服务我们就称之为Active Namenode

另外一台就称之为Standby Namenode这张图展示了Standby Namenode的工作模式

其中Active Namenode和Standby Namenode通过可靠的共享存储来共享edit log

这就保证了Standby Namenode拥有跟Active Namenode一样的最新的状态

而另外所有的Datanode需要同时向Active和Standby Namenode回报状态

这样当Active Namenode出现故障的时候Standby Namenode可以很快的成为新的Active Namenode并对外提供服务一般来说从Active Namenode切换到Standby Namenode

只有分钟级别的延迟另外 Standby Namenode也具备Secondary Namenode的功能

也就是说会定期的对Active Namenode上的fsimage和edit log进行合并

使用Standby Namenode的一个重点呢就是要保证任意一个时间点只有一台Namenode是Active的否则就可能会导致元数据的损坏这意味着当我们决定从Active Namenode切换到

Standby Namenode的时候我们必须保证原来的Active Namenode不会再继续写入元数据

为了实现这一点我们可以要求共享存储只能授予一台Namenode写权限或者我们可以远程终止原来的Active Namenode的进程关闭其网络端口甚至是对原来的Active Namenode的节点进行强行的断电

客户端：比如我们需要将HDFS URL设置为逻辑主机名这样可以映射到两台Namenode上

HDFS的扩展性

所有的文件的元数据都要保存在内存中这就意味着HDFS所能管理的文件总数是受Namenode内存限制的 为了突破这一限制HDFS引入了一种叫做federation的机制

**【Block Storage和namespace高耦合】**

当前namenode中的namespace和block management的结合使得这两层架构耦合在一起，难以让其他可能namenode实现方案直接使用block storage。

为了水平扩展namenode，federation使用了多个独立的namenode/namespace。这些namenode之间是联合的，也就是说，他们之间相互独立且不需要互相协调，各自分工，管理自己的区域。分布式的datanode被用作通用的数据块存储存储设备。每个datanode要向集群中所有的namenode注册，且周期性地向所有namenode发送心跳和块报告，并执行来自所有namenode的命令。

一个block pool由属于同一个namespace的数据块组成，每个datanode可能会存储集群中所有block pool的数据块。

每个block pool内部自治，也就是说各自管理各自的block，不会与其他block pool交流。一个namenode挂掉了，不会影响其他namenode。

某个namenode上的namespace和它对应的block pool一起被称为namespace volume。它是管理的基本单位。当一个namenode/nodespace被删除后，其所有datanode上对应的block pool也会被删除。当集群升级时，每个namespace volume作为一个基本单元进行升级。

就是允许一个集群可以配置多台Namenode大家可以看这张图每个Namenode会管理一个

独立的命名空间每个命名空间会对应一个Block pool每个Datanode要向所有的Namenode注册并且回报状态每一个Datanode可以存储多个Block pool当中的数据

另一方面的扩展性 fileSystem 的扩展性 对象存储

我们先来看读取一个文件的步骤

首先客户端通过文件系统接口 打开要读取的文件 文件系统与Namenode进行通信

然后Namenode会返回 这个文件前几个Block的地址我们 这里所说的Block的地址是指

存储了这个Block的Datanode的列表而且这个列表当中的Datanode**会按照它们与客户端之间的距离从近到远进行排序**举个简单的例子如果客户端所在的节点也是一台Datanode且存储了要读取的Block那么这个Datanode会出现在这个列表当中的第一个 文件打开以后客户端会得到一个inputstream后就可以通过这个inputstream来读取数据了大家可以看这个图中的第三步当客户端调用read时inputstream会根据Block的地址连接到相应的Datanode上读取数据当文件的第一批Block读取完毕以后inputstream会再跟Namenode通信然后来请求后续的Block地址在读取一个Block的时候如果发现地址当中的某个Datanode出现了故障

就会自动的选择列表当中的下一个Datanode来读取数据这一系列的过程对于客户端来说都是透明的从用户的角度来看就是在读取一个连续的数据流从这一个流程中我们能够看出数据流是分布在整个集群中的Namenode只需要提供文件的元数据信息就可以了这样就大大减轻了Namenode的压力

面我们再来看一下写文件的步骤

用户首先通过文件系统接口 创建一个文件 这里会跟Namenode进行通信Namenode会进行一系列的检查比如用户是否有权限创建这个文件等等如果一切顺利的话Namenode会在命名空间当中为这个新的文件创建一条记录 文件创建成功以后用户得到一个outputsteam通过这个outputsteam就可以写入数据了 在底层outputsteam通过data streamer来发送数据

随着用户数据的写入data streamer会跟Namenode进行通信请求Namenode分配新的Block对于每一个Blockdata streamer还会选择一组Datanode来存储这个Block我们将这样一组Datanode称为一个pipelinepipeline当中Datanode的数量就等于文件的复制份数

如这个图中所示的默认的pipeline会由三台Datanode组成

之后data streamer会将用户的数据包发送到pipeline中第一台Datanode上这台Datanode再负责将数据包拷贝到pipeline中下一台Datanode上以此类推直到pipeline中所有的Datanode都 接收到了数据每一台Datanode在收到数据包以后 会发送一个ack确认信息 只有当pipeline中 所有的Datanode都确认以后 客户端才认为这个数据包是发送成功的 在写数据的过程当中如果某一个Datanode出现故障则会将其从当前的pipeline中移除

客户端会重发已经发送但是还没有被确认的数据包这样做的目的是为了保证pipeline中的后续的Datanode能够收到数据 在数据写入完成以后Namenode会将这个文件标记为复制份数不足并且在后台进行一个异步的复制直到文件的副本数达到用户设定的值

SPARK

批处理与流处理：

在典型的批处理系统中数据需要预先按小时或天划分成批次 一次执行只处理一个批次同一批的数据会存储在一个分布式文件系统上通过一个或多个Map-Shuffle-Reduce

过程处理数据每个阶段都需要将中间结果写入磁盘一旦某个阶段出错可以从中间结果中快速恢复而不用从头再来 批处理系统通常只能处理预先划分好的数据 数据是前一个小时或者前一天的 因此不能保证实时性

而在流处理系统中

数据不会预先分成一个个批次 而是实时流入 连续不断 无穷无尽 系统则持续不断地

分布式处理流入的数据 实时返回结果

那它的容错性如何呢

因为流是无穷无尽的 一个流处理系统必须保持在线 出错后能迅速恢复

流处理系统的容错性保证有三个层次

最弱的保证就是允许消息丢失对于近似计算 比如统计每分钟的大致用户访问量来说这并不会影响结果的正确性

再强一点 消息不能丢失但是可以重复此时系统要能检测到消息丢失并且重放丢失的消息

像数据清洗这类应用重复的无用数据还是会被过滤掉但是不允许有用消息的丢失

最强的保证是消息不丢不重比如搜索引擎按广告点击量向广告主收费如果消息重了

广告主会被多收费如果消息丢了搜索引擎就要蒙受损失

如何检测消息丢失呢

第一种是中心节点确认图中起点A先发送一个带有独特ID的消息给中心确认节点 Acker

再把消息本身发送给 BB收到之后会发送确认消息给Acker同时告知它将消息发给了C

C收到B的消息后也会通知Acker消息收到了假如在一定时间内Acker没有收到所有的确认消息就会超时报错（单一节点故障）

第二种办法是上下游确认A给B发消息每条消息会带一个独特IDB收到后会回复一个

带有该ID的确认消息如果A没有收到对应的确认消息则消息丢失 一般通过批量确认可以提高性能比如每一百条消息确认一次

如果消息在丢失之后需要重放

一种是基于可重放队列(环状)

数据源的消息在进入流处理系统前会先被写入一个可重放队列一旦系统在处理过程中出错

可以从队列中重新读取之前的消息

另一种做法是从上游重放各个节点将计算结果写入到可持久化存储如果下游节点出错则上游节点从存储中读取出错消息进行重放

那么如何去除重复消息

如果计算本身是幂等运算即重复运算不影响结果则无需去重

第二种做法是在接收端过滤 每条消息都有一个独特的ID接收端收到消息会查询存储

是否已经收到过这个ID如果没有就接收该消息并将ID保存 如果存在则为重复消息 直接丢弃

这是一个网页搜索异常检测的应用 当用户进行网页搜索的时候，首先应用会对关键词进行窗口聚合运算，统计其在一段时间内的搜索量统计，结果会发送给一个机器学习模块，构建出关键词的历史模型，知道它的正常走势，同时新的统计结果会和这个历史模型进行比对 进行主键连接运算 过高或者过低都会触发异常

窗口聚合运算

一个消息从数据源产生 到进入流处理系统总会存在延迟 消息被处理的时钟 总是晚于消息产生的时钟 并且消息在传输过程中可能发生乱序 比如图中 7点钟的消息 到了7点15分才进入系统 7点08分的消息 会在7点12分的消息之后进入系统 这里我们定义两个时钟

一个是消息时钟 即消息产生的时间 另一个是系统时钟 即消息进入系统被处理的时间

图中横坐标是消息时钟 纵坐标是系统时钟 由于存在延时 一条消息的系统时钟 总是大于消息时钟 因此图上所有的点都落在左上方 如果我们现在要统计 7点到7点10分的消息数量

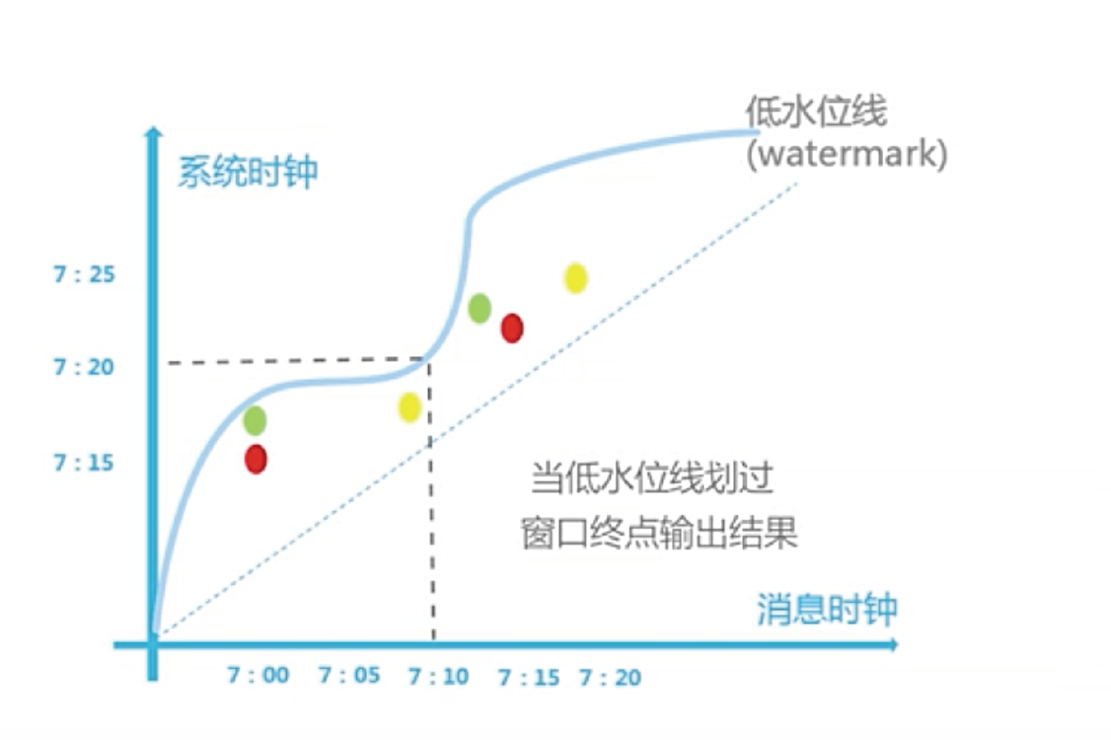
我们不能在系统时间 为7点10分的时候输出结果 因为此时有些消息 还在网络传输的过程中

通常我们会借助低水位线的概念 如图中的蓝色曲线 低水位线代表消息时钟的进度

它表示不会有更小时钟的消息 进入系统 如果有系统通常会直接丢弃

低水位线是单调递增的 它的划定通常基于经验 以及应用的容错性假如应用不能容忍消息丢失

那么低水位线的划定就会保守一些



现在从7点开始 我们要基于消息时钟 统计窗口大小为10分钟的消息数量

系统会在内存中维护多个时间窗口 比如当系统时钟为7点18分时7点到7点10分的窗口里

有两条7点的消息 7点10分到7点20的窗口有一条7点12分的消息

此时低水位线小于7点10分 表示消息时钟小于7点10分的消息

还有可能进入系统 比如7点08分的消息此时不能输出7点

到7点10这个窗口的结果 当系统时钟为7点20时低水位线等于7点10分

认为不会再有小于7点10分的消息 进入系统 可以输出7点到7点10分的统计结果

它们的统计结果 在低水位线划过窗口之前都不能输出

是主键连接 主要有流和表的连接 流和流的连接两种

流和表连接中 内存里已经有一张哈希表 这样当消息到达时 我们就去查询哈希表的对应主键

例如主键是搜索关键词 输入是张三的窗口统计结果 通过在哈希表中查找张三的模型

我们就能知道张三的统计结果 是否有异常 两条流连接又分两种情况

如果一条流比另一条流先到达 我们可以在内存中将先到达的流数 添加到哈希表中

第二条流到达的时候 就和流和表的连接同理 如果两条流到达的顺序不确定

那么我们需要动态去构建两个哈希表 对于同一个主键 如果流1先到了 我们就把对应数据

添加到它的哈希表里 流2到的时候去查询流1的哈希表 如果流2先到了 就刚好相反

因为内存的容量总是有限的 先到的流 不能一直等待另一个流的对应消息 所以我们通常会基于一定时间窗口 进行缓存 过了时间窗口 则对应的数据会从哈希表中删除

SPARK内部结构

它肯定需要有一个分布式的cluster 这个cluster它有很多个node组成

我们把一个node就叫做一个worker 一个Spark的cluster里面 它就会有多个worker node

这时一个node我们可以把它当做一台机器 在一个worker node上面 Spark会可能启动一个或者说多个executor 在同一个worker node上面 在这边我们假设 每一台worker node

都只启一个executor 而一个executor是 在这个node上面的一个进程 单单启动一个进程

是不能够达到很高的一个并行度的 所以在Spark里面 它会在executor内部 再去启很多个task 而这个task是真正去做计算的单元这个task是以线程的形式存在的

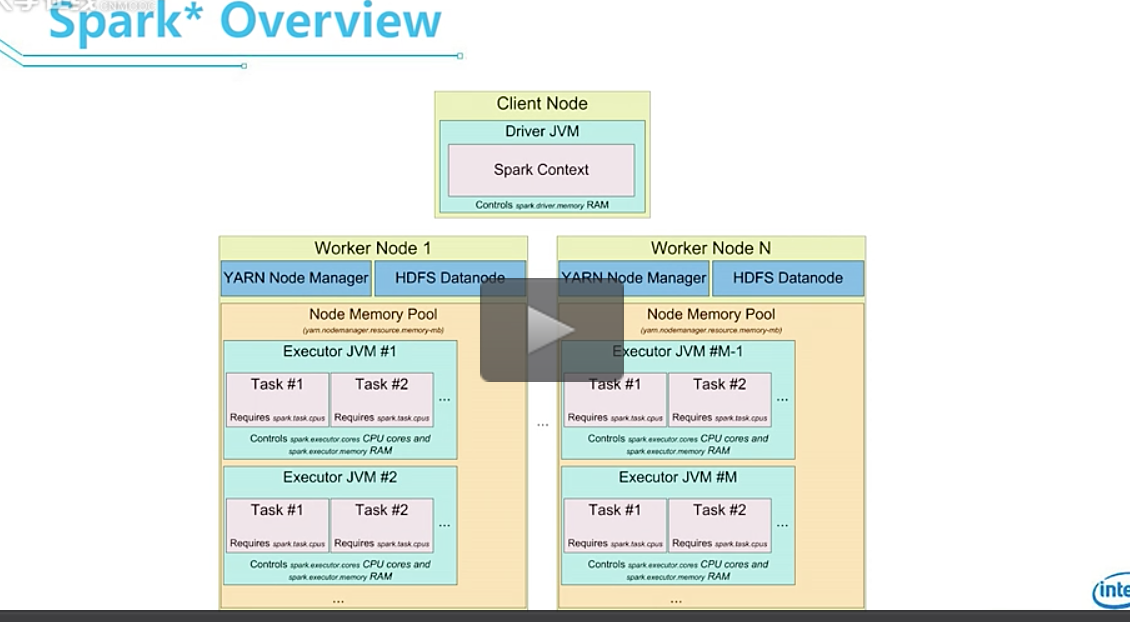
Driver Program其实就是我们 Spark的一些应用 它就是通过Spark Context

去把它传递给Cluster Manager Cluster Manager它是一个 用来管理这么多worker node的一个Manager

由于Spark它有不同的运行模式 现在的话它至少可以支持三种

第一种是standalone模式 standalone模式的话 它就是一个master worker这种架构

还有一种是yarn的这种模式 Cluster Manager就由 yarn会自己去管理这些worker node



我们这里用yarn mode来作为一个例子 我们可以看到在yarn mode上 我们其实也是可以有

多个的worker node 在每一个worker node上面，首先在yarn mode上面 它会有一些yarn相关的一些进程 包括node manager进程 另外一个的话也包括 HDFS Datanode进程

我们可以启多个 从executor 1甚至到executor N都可以 在每个worker node上面

都可以启动多个的executor 在每一个executor内部 我们也可以启动多个task比如说这executor它需要多少个CPU它需要多少个memory都是通过spark调度你可以去指定

shuffle

重新归类 我想要拿到 a到底出现了多少次 a要把它统一到一个地方然后我再来进行计算

我需要让a都放在同一块partation里面 去进行计算 我要把第二个partation里面的a

也要传到这边来 我就要对这个Map Task里面所有的数据 分成三个集合

每一个集合我们交给它一个bucket 每一个bucket就对应一个Reduce从Map Task去生成这些bucket的过程 是一个Shuffle write的过程 而从Reduce Task 去把这些bucket

这些数据拿过来的这个过程 叫做Shuffle fetch

Spark假定在大多数用户的case中，shuffle数据的sort不是必须的，比如word count，强制地进行排序只会使性能变差，因此Spark并不在Reducer端做merge sort。既然没有merge sort那Spark是如何进行reduce的呢？这就要说到aggregator了。

aggregator本质上是一个hashmap，它是以map output的key为key，以任意所要combine的类型为value的hashmap。当我们在做word count reduce计算count值的时候，它会将shuffle fetch到的每一个key-value pair更新或是插入到hashmap中(若在hashmap中没有查找到，则插入其中；若查找到则更新value值)。这样就不需要预先把所有的key-value进行merge sort，而是来一个处理一个，省下了外部排序这一步骤。但同时需要注意的是reducer的内存必须足以存放这个partition的所有key和count值，因此对内存有一定的要求。

内存管理

假设有M个map，C个CPU，R个reduce，则同时有C个map在运行，一共产生M\*R个file handle，memory开销巨大 小文件的处理非常非常耗磁盘的IO

优化1 Consolidate Shuffle file:M>C时有效

第一批生成C\*R个file，第二批的bucket放在第一批的file里

若C=R 性能不会提高

优化2 Sort Based Shuffle对map task里的数据根据partition来sort 一个FileSegment就是一个bucket只不过这些bucket是按顺序 存放在了这个文件里面 同时打开C个文件，一共打开M个文件，reduce去找对应的fileSegment

Legacy

它的这个storage memory 默认情况下它是占60% 那在像execution memory状态

就占20% safe fraction是0.9 最多只能用54 超过相当用满 防止溢出

这种内存分配机制，最大的问题是，谁都不能超过自己的上限，规定了是多少就是多少，虽然另外一片内存闲着呢。这在是StorageMemory 和 ExecutionMemory比较严重，他们都是消耗内存的大户。

这个问题引出了Unified Memory Management模型，重点是打破ExecutionMemory 和 StorageMemory 这种分明的界限。

Unified 执行空间和存储空间加起来分配75% 各自用多少不定。

这片内存区域是为了解决 shuffles,joins, sorts and aggregations 过程中为了避免频繁IO需要的buffer。

StorageMemory。这片内存区域是为了解决 block cache(就是你显示调用dd.cache, rdd.persist等方法), 还有就是broadcasts,以及task results的存储。可以通过参数 spark.storage.memoryFraction(默认0.6)。

SQL：

把一个txt文件把它转换成一个people类型的RDD就是先每一行根据逗号去split开

然后再把它映射成一个person的case class之后我们得到的是一个people类型的RDD

它会隐式的把people类型的RDD转换成一个DataFrameHolderDataFrameHolder就有一个

to DataFrameHolder的一个函数通过这个函数调用我们最后返回的是一个DataFrame

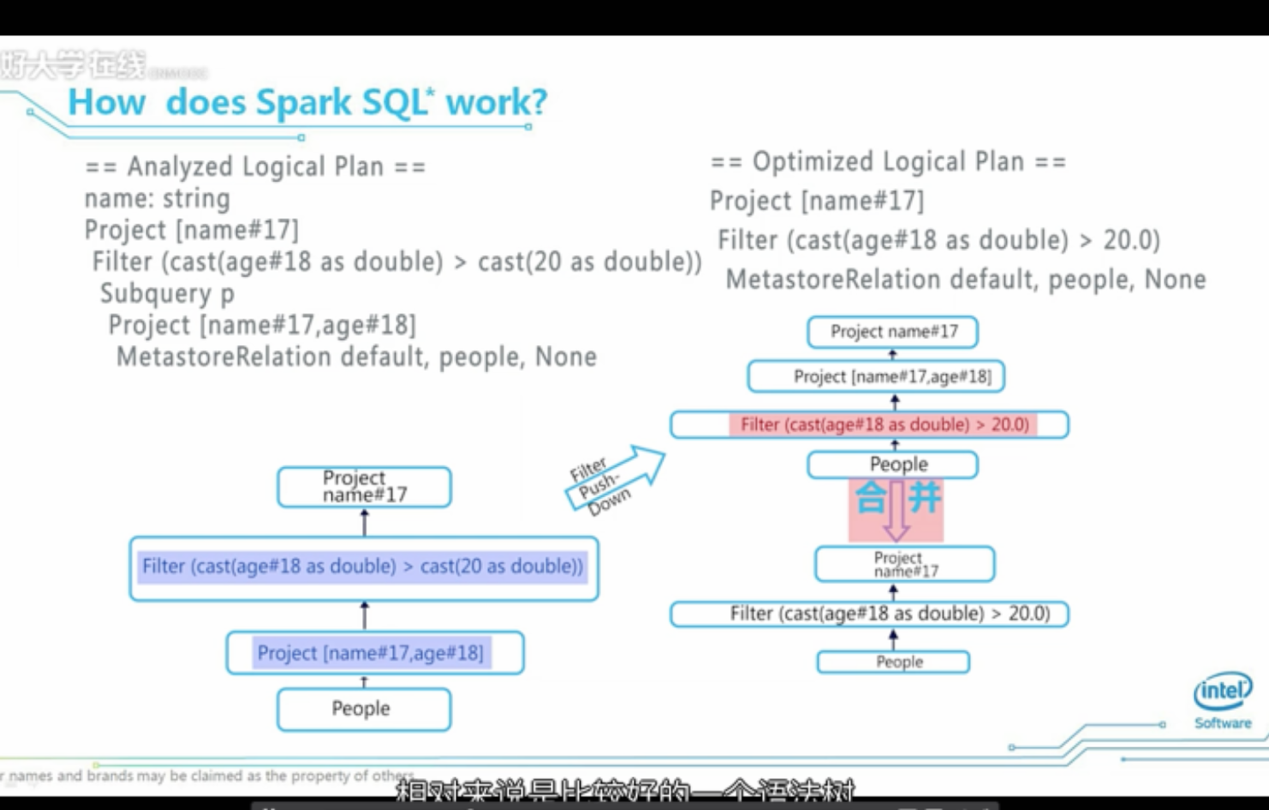
temptable临时表如果测试什么的 你可以直接调用registerTempTable

你可以用saveAsTable这样的方式可以把整张Table把它序列化在磁盘上面

下一次你再重新打开context的时候这张表还是一样存在的

通过field的index或者name访问结果

列存储好处：按列访问性能高 针对相同类型的数据去选择编码和压缩算法



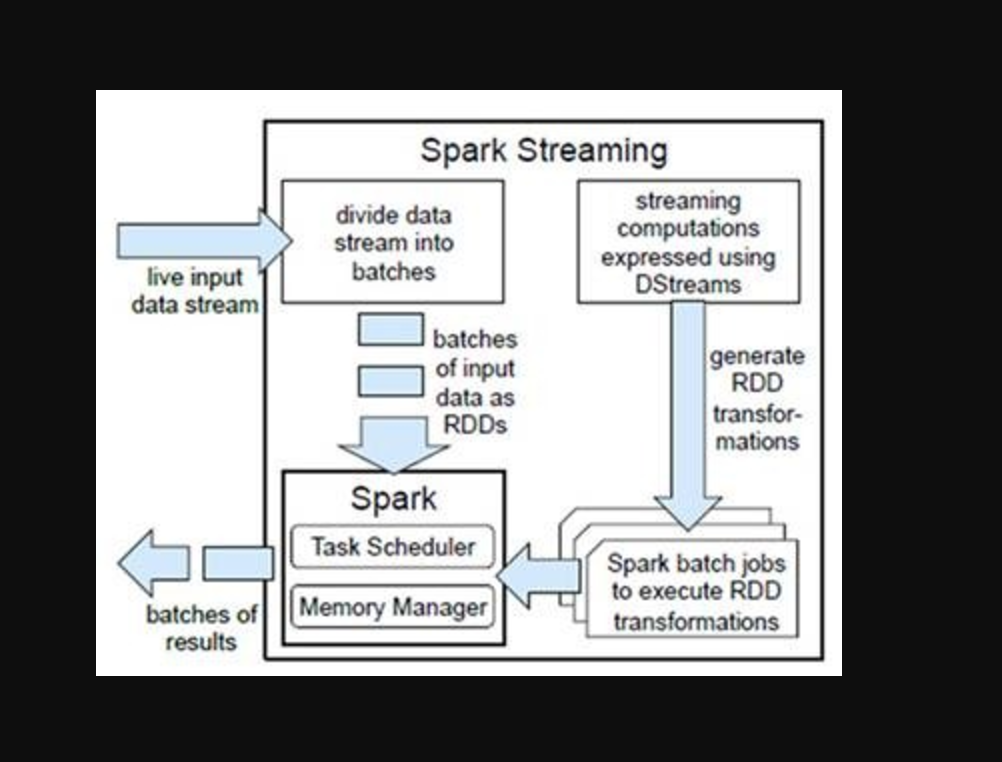
Optimizer的主要职责是将Analyzer给Resolved的Logical Plan根据不同的优化策略Batch，来对语法树进行优化，优化逻辑计划节点(Logical Plan)以及表达式(Expression)，也是转换成物理执行计划的前置。它的工作原理和analyzer一致，也是通过其下的batch里面的Rule[LogicalPlan]来进行处理的。

Spark 运行架构如下图：

各个RDD之间存在着依赖关系，这些依赖关系形成有向无环图DAG，DAGScheduler对这些依赖关系形成的DAG，进行Stage划分，划分的规则很简单，从后往前回溯，遇到窄依赖加入本stage，遇见宽依赖进行Stage切分。完成了Stage的划分,DAGScheduler基于每个Stage生成TaskSet,并将TaskSet提交给TaskScheduler。TaskScheduler 负责具体的task调度,在Worker节点上启动task。

spark streaming

**计算流程**：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 Core，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。

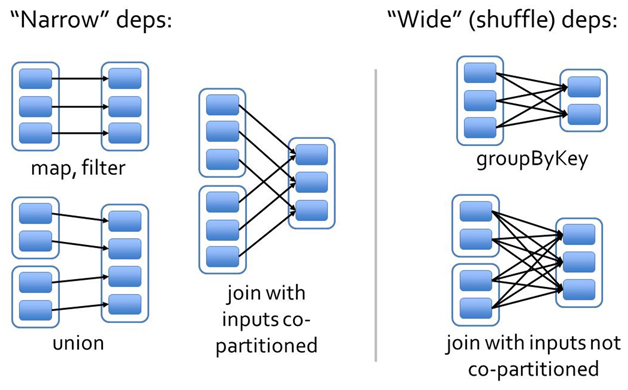


**容错性**：对于流式计算来说，容错性至关重要。首先我们要明确一下Spark中RDD的容错机制。每一个RDD都是一个不可变的分布式可重算的数据集，其记录着确定性的操作继承关系（lineage），所以只要输入数据是可容错的，那么任意一个RDD的分区（Partition）出错或不可用，都是可以利用原始输入数据通过转换操作而重新算出的。

流数据典型应用：

,数据流可被视为一个随时间延续而无限增长的动态数据集合。应用于[网络监控](http://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E7%9B%91%E6%8E%A7)、[传感器网络](http://baike.baidu.com/item/%E4%BC%A0%E6%84%9F%E5%99%A8%E7%BD%91%E7%BB%9C)、航空航天、气象测控和金融服务等领域。

* 窄依赖是指父RDD的每个分区只被子RDD的一个分区所使用，**子RDD分区通常对应常数个父RDD分区**(O(1)，与数据规模无关)
* 相应的，宽依赖是指父RDD的每个分区都可能被多个子RDD分区所使用，**子RDD分区通常对应所有的父RDD分区**(O(n)，与数据规模有关)



相比于宽依赖，窄依赖对优化很有利 ，主要基于以下两点：

1. 宽依赖往往对应着shuffle操作，需要在运行过程中将同一个父RDD的分区传入到不同的子RDD分区中，中间可能涉及多个节点之间的数据传输；而窄依赖的每个父RDD的分区只会传入到一个子RDD分区中，通常可以在一个节点内完成转换。
2. 当RDD分区丢失时（某个节点故障），spark会对数据进行重算。
   1. 对于窄依赖，由于父RDD的一个分区只对应一个子RDD分区，这样只需要重算和子RDD分区对应的父RDD分区即可，所以这个重算对数据的利用率是100%的；
   2. 对于宽依赖，重算的父RDD分区对应多个子RDD分区，这样实际上父RDD 中只有一部分的数据是被用于恢复这个丢失的子RDD分区的，另一部分对应子RDD的其它未丢失分区，这就造成了多余的计算；更一般的，宽依赖中子RDD分区通常来自多个父RDD分区，极端情况下，所有的父RDD分区都要进行重新计算。

Dstream和Dstream Graph

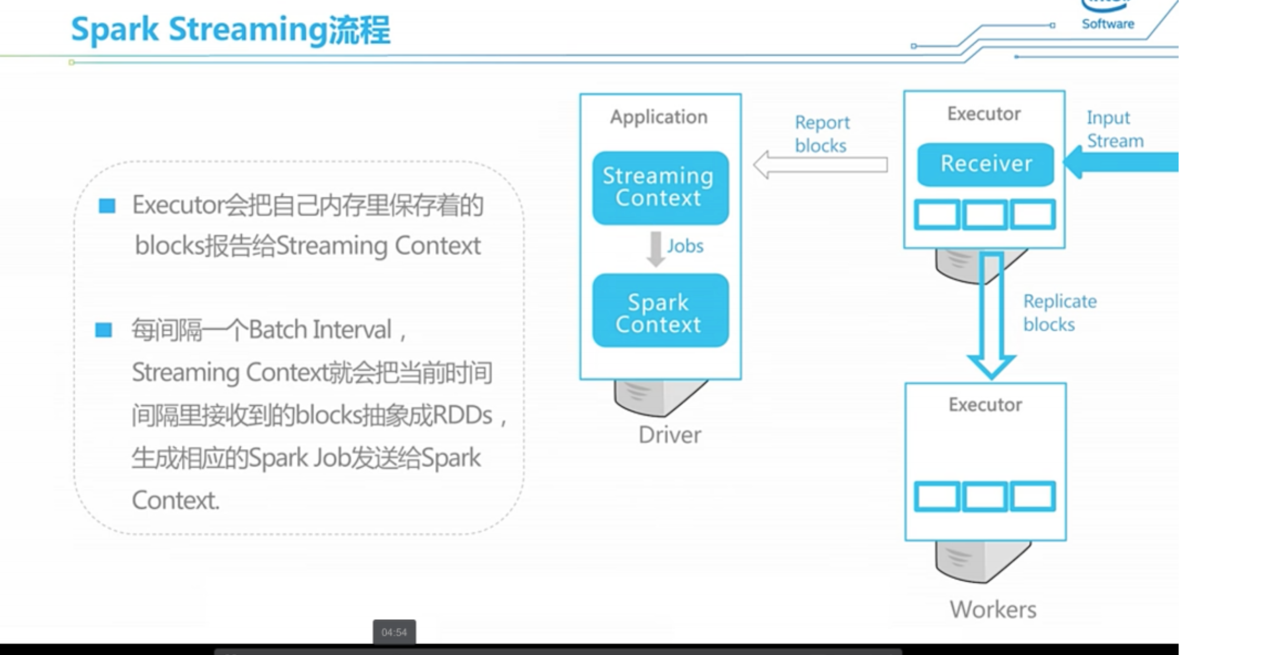
1. 在 Spark Streaming 里，这个 RDD “模板”对应的具体的类是 DStream，RDD DAG “模板”对应的具体类是 DStreamGraph。
2. 回想一下，RDD 的定义是一个只读、分区的数据集（an RDD is a read-only, partitioned collection of records），而 DStream 又是 RDD 的模板，所以我们把 Dstream 也视同数据集

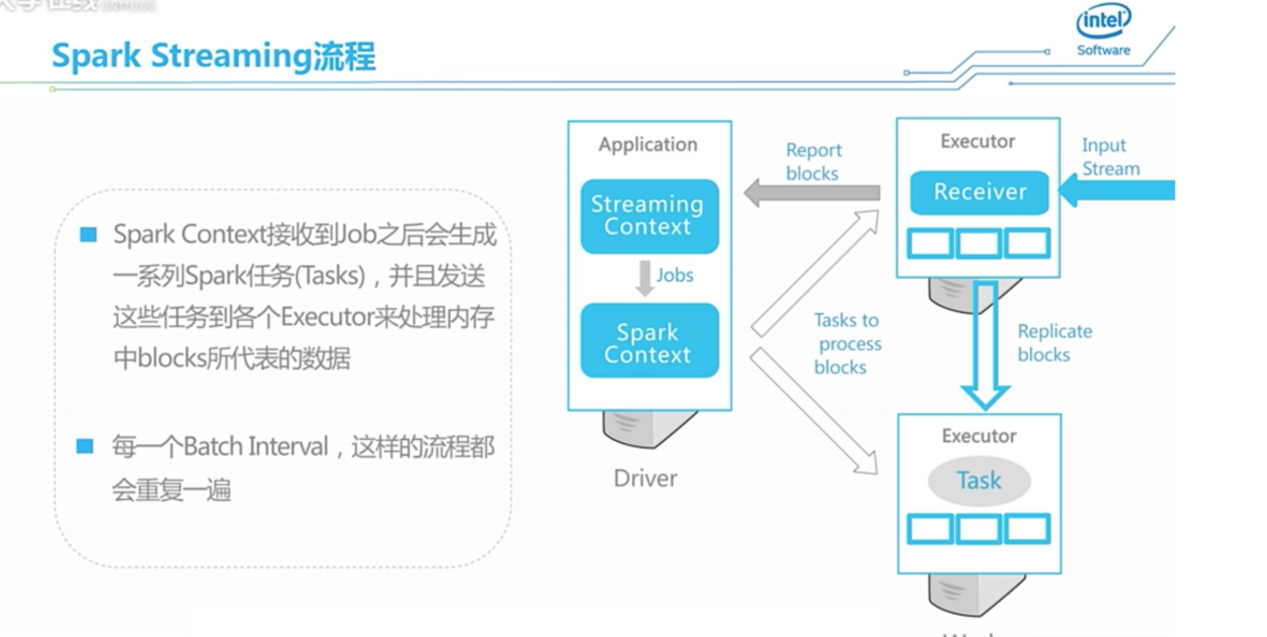
**Spark Streaming的处理过程：**

1. 以时间片为单位划分形成数据流形成RDD（DStream）
2. 对每个划分数来的RDD以批处理的方式处理数据
3. 每个划分出来的RDD地处理都会提交成Job

**比较适用的场景：**

* 历史数据和实时数据结合进行分析
* 对实时性要求不是很苛刻的场景





流式数据动态变化

在数据进入Spark Streaming之后会对数据进行切分把它按照一定的时间间隔划分成一个一的小batch又会被抽象称为一个一个的RDD是一个容错的、并行的数据结构

是对底层实际数据的一个逻辑抽象DStream可以看成是Spark Streaming最核心的部分

DStream是Spark Streaming对于持续流式数据的一种基本抽象在内部实现上

DStream会被表示成一系列连续的RDD每一个RDD都代表一定时间间隔内到达的数据

在应用了某种DSL里的操作之后 就会产生一个新的DStream像这边如果一个input的DStream apply filter之后就会生成一个Filtered DStream把这个Filtered DStream

和另外一个input DStream进行union操作之后就会生成一个union DStream

这些DStream就会生成一个依赖关系的图 我们称它为DStreamGraph