Large Clusters

燃烧吧安菲拉尔 2017-01-08 原文

MapReduce: Simplified Data Processing on Large Clusters

MapReduce：面向大型集群的简化数据处理

摘要

MapReduce既是一种编程模型，也是一种与之关联的、用于处理和产生大数据集的实现。用户要特化一个map程序去处理key/value对，并产生中间key/value对的集合，以及一个reduce程序去合并有着相同key的所有中间key/value对。本文指出，许多实际的任务都可以用这种模型来表示。

用这种函数式风格写出的程序自动就拥有了在一个大的PC机集群上并行执行的能力。运行时系统会负责细节：切分输入数据，在一组机器上调度执行程序，处理机器错误，以及管理所需的机器间通信。这允许不具备任何并行和分布式系统经验的程序员也能轻松地利用一个大型分布式系统的资源。

我们的MapReduce实现运行在一个大型PC机集群上，且具有很好的扩展性：一个典型的MapReduce计算要在数千台机器上处理若干TB的数据。程序员可以很轻松的使用这一系统：目前已经实现的MapReduce程序数以百计，每天有上千个MapReduce作业运行在Google的集群上。

1 介绍

在过去的5年中，作者以及许多其他Google员工实现了数百个特定用途的计算过程，其中包括了处理大量的原始数据（抓取文档、网络请求日志等等），计算许多种类的衍生数据（倒排索引、网络文档图结构的多种表示、单台主机抓取页面数量的概要、指定日期频次最高的请求集合等等）。大多数这样的计算过程在概念上都很直接。但输入数据量通常都很大，因此计算过程需要分布到数百或数千台机器上进行，才能保证过程在一个合理时间内结束。而为了处理计算并行化、数据分发和错误处理等问题而引入大量复杂的代码则令原本简单的计算过程变的晦涩难懂。

作为对这种复杂性的回应，我们设计了一种新的抽象，允许我们表达出原本简单的计算过程，且将涉及并行、容错性、数据分发和负载均衡的凌乱细节隐藏在一个函数库中。我们的抽象受到了Lisp等函数式语言中的map和reduce原语的启发。我们意识到我们大多数的计算都包含了在每个输入的逻辑“记录”上应用map操作，从而计算出一组中间key/value对的集合，然后再向共享同一个key的所有中间value应用reduce操作，从而适当地合并衍生数据。我们对用户定义的map和reduce操作的使用允许我们轻松地将大型计算并行化，以及将再执行作为容错性的主要机制。

本项工作的主要贡献是一个简单但功能强大的接口，允许自动实现并行化和大范围的分布计算，和一个与此接口结合的实现，能在普通PC机集群上达到很高的性能。

第2部分描述了基本的编程模型并给出了几个例子。第3部分描述了针对我们基于集群的计算环境而裁剪的MapReduce接口的一个实现。第4部分描述了我们觉得非常有用的一些编程模型的技巧。第5部分针对多种不同的任务，对我们的实现进行了性能测量。第6部分探索了在Google中MapReduce的应用，包括了我们在将其作为生产索引系统的重写基础的经验。第7部分讨论了相关的和未来要做的工作。

2 编程模型

计算过程就是输入一组key/value对，再生成输出一组key/value对。MapReduce库的使用者用两个函数来表示这个过程：map和reduce。

map由使用者编写，使用一个输入key/value对，生成一组中间key/value对。MapReduce库将有着相同中间key I的中间value都组合在一起，再传给reduce函数。

reduce也由使用者编写，它接受一个中间key I和一组与I对应的value。它将这些value合并为一个可能更小的value集合。通常每个reduce调用只产生0或1个输出value。中间value是通过一个迭代器提供给reduce函数的。这允许我们操作那些因为大到找不到连续存放的内存而使用链表的value集合。

2.1 示例

考虑一个问题：统计一个很大的文档集合中每个单词出现的次数。使用者能写出与下面的伪代码相似的代码：

```
map(String key,String value):
    // key: 文档名
    // value: 文档内容
    for each word w in value:
        EmitIntermediate(w,"1");

reduce(Stringkey, Iterator values):
    // key: 一个单词
    // value: 计数值列表
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

map函数将每个单词与出现次数一同输出（本例中简单的输出“1”）。reduce函数将针对某个特定词输出的次数都合并相加。

另外，使用者要写代码填充一个符合MapReduce规格的对象，内容包括输入和输出文件的名字，以及可选的调节参数。之后使用者调用MapReduce函数，将指定的对象传进去。用户代码会与MapReduce库（C++实现）链接到一起。附录A包括了这个例子的全部程序文本。

2.2 类型

尽管前面的伪代码写成了用字符串进行输入输出，但从概念上讲用户提供的map和reduce函数是关联着类型的：

map (k1, v1) → list(k2, v2)

reduce(k2, list(v2))→ list(v2)

也就是说，输入的key和value与输出的key和value的域不同。进一步说，中间的key和value与输出的key和value的域相同。

我们的C++实现使用字符串与用户定义的函数交互，而将字符串与相应类型的转换留给用户代码完成。

2.3 更多例子

这里例举了一些有趣的程序，它们都可以很轻松的用MapReduce模型表达。

分布式Grep：map函数在匹配到给定的pattern时输出一行。reduce函数只是将给定的中间数据复制到输出上。

URL访问频次统计：map函数处理网页请求的日志，对每个URL输出〈URL, 1〉。reduce函数将相同URL的所有值相加并输出〈URL, 总次数〉对。

倒转Web链接图：map函数在source页面中针对每个指向target的链接都输出一个〈target, source〉对。reduce函数将与某个给定的target相关联的所有source链接合并为一个列表，并输出〈target, list(source)〉对。

每个主机的关键词向量：关键词向量是对出现在一个文档或一组文档中的最重要的单词的概要，其形式为〈单词, 频率〉对。map函数针对每个输入文档（其主机名可从文档URL中提取到）输出一个〈主机名, 关键词向量〉对。给定主机的所有文档的关键词向量都被传递给reduce函数。reduce函数将这些关键词向量相加，去掉其中频率最低的关键词，然后输出最终的〈主机名, 关键词向量〉对。

倒排索引：map函数解析每个文档，并输出一系列〈单词, 文档ID〉对。reduce函数接受给定单词的所有中间对，将它们按文档ID排序，再输出〈单词, list(文档ID)〉对。所有输出对的集合组成了一个简单的倒排索引。用户可以很轻松的扩展这个过程来跟踪单词的位置。

分布式排序：map函数从每条记录中提取出key，并输出〈key, 记录〉对。reduce函数不改变这些中间对，直接输出。这个过程依赖于4.1节介绍的划分机制和4.2节介绍的排序性质。

3 实现

许多不同的MapReduce的实现都是可行的。选择哪一个要取决于环境。例如，一种实现可能适合于小型的共享内存机器，一种实现可能适合于大型的NUMA多处理器机器，而另一种则适合于更大型的联网机器集。

本部分描述的实现主要面向Google内部广泛使用的计算环境：大型的商用PC机集群，互相之间用交换式以太网连接。我们的环境是：

1. 主要使用的机器为双核X86处理器，运行Linux系统，每台机器的内存从2GB到4GB不等。

2. 使用的都是商用网络硬件设备——在机器层面上通常从100Mbps到1Gbps不等，但平均起来要比总带宽的一半少很多。

3. 集群中拥有数百或数千台机器，因此机器错误经常出现。

4. 每台机器都使用廉价的IDE硬盘来提供存储功能。我们使用一种内部开发的分布式文件系统来管理这些磁盘上的数据。这个文件系统通过复制的方法在不可靠的硬件之上提供了实用性与可靠性。

5. 用户向一个调度系统提交作业。每个作业包括了一个任务集，会由调度器调度到集群内可用的一组机器上。

3.1 执行过程概述

通过自动将输入数据切分为M块，map调用分布在多台机器上进行。输入划分可以在不同的机器上并行执行。reduce调用是通过一个划分函数（例如hash(key) mod R)将中间key空间划分为R块来分布运行。划分的块数R和划分函数都由用户指定。

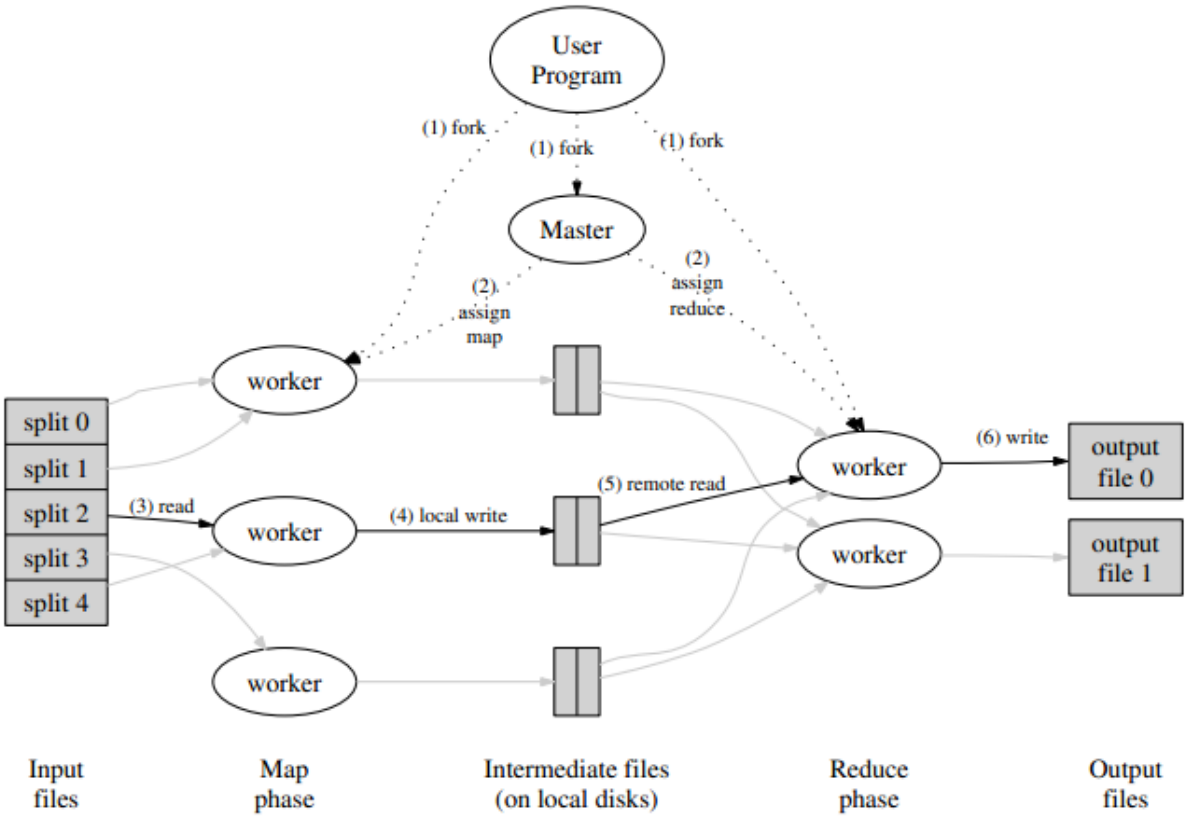

Figure 1: Execution overview

图1展示了我们的实现中MapReduce操作的整体流程。当用户程序调用MapReduce函数时，会发生下面一系列动作（图1中的标号与下面列表顺序相同）：

1. 用户程序中的MapReduce库首先将输入文件切分为M块，每块的大小从16MB到64MB（用户可通过一个可选参数控制此大小）。然后MapReduce库会在一个集群的若干台机器上启动程序的多个副本。

2. 程序的各个副本中有一个是特殊的——主节点，其它的则是工作节点。主节点将M个map任务和R个reduce任务分配给空闲的工作节点，每个节点一项任务。

3. 被分配map任务的工作节点读取对应的输入区块内容。它从输入数据中解析出key/value对，然后将每个对传递给用户定义的map函数。由map函数产生的中间key/value对都缓存在内存中。

4. 缓存的数据对会被周期性的由划分函数分成R块，并写入本地磁盘中。这些缓存对在本地磁盘中的位置会被传回给主节点，主节点负责将这些位置再传给reduce工作节点。

5. 当一个reduce工作节点得到了主节点的这些位置通知后，它使用RPC调用去读map工作节点的本地磁盘中的缓存数据。当reduce工作节点读取完了所有的中间数据，

它会将这些数据按中间key排序，这样相同key的数据就被排列在一起了。同一个reduce任务经常会分到有着不同key的数据，因此这个排序很有必要。如果中间数据数量过多，不能全部载入内存，则会使用外部排序。

6. reduce工作节点遍历排序好的中间数据，并将遇到的每个中间key和与它关联的一组中间value传递给用户的reduce函数。reduce函数的输出会写到由reduce划分过程划分出来的最终输出文件的末尾。

7. 当所有的map和reduce任务都完成后，主节点唤醒用户程序。此时，用户程序中的MapReduce调用返回到用户代码中。

成功完成后，MapReduce执行的输出都在R个输出文件中（每个reduce任务产生一个，文件名由用户指定）。通常用户不需要合并这R个输出文件——他们经常会把这些文件当作另一个MapReduce调用的输入，或是用于另一个可以处理分成多个文件输入的分布式应用。

3.2 主节点数据结构

主节点维持多种数据结构。它会存储每个map和reduce任务的状态（空闲、处理中、完成），和每台工作机器的ID（对应非空闲的任务）。

主节点是将map任务产生的中间文件的位置传递给reduce任务的通道。因此，主节点要存储每个已完成的map任务产生的R个中间文件的位置和大小。位置和大小信息的更新情况会在map任务完成时接收到。这些信息会被逐步发送到正在处理中的reduce任务节点处。

3.3 容错性

既然MapReduce库是为了帮助使用成百上千台机器处理数量非常大的数据的，它就必须能够优雅地承受机器错误。

工作节点错误

主节点周期性的ping每个工作节点。如果工作节点在一定时间内没有回应，主节点就将它标记为已失败。这个工作节点完成的任何map任务都被重置为空闲状态，并可被调度到其它工作节点上。同样地，失败的工作节点上正在处理的任何map或reduce任务也被重置为空闲状态，允许被调度。

失败节点上已完成的map任务需要重执行的原因是它们的输出存储在失败机器的本地磁盘上，因此无法访问到了。已完成的reduce任务不需要重执行，因为它们的输出存储在了一个全球文件系统上。

当一个map任务先被A节点执行过，随后又被B节点重执行（A节点已失败），所有执行reduce任务的工作节点都能收到重执行的通知。任何没有读取完A节点数据的reduce任务都会从B节点读取数据。

MapReduce可以弹性应对大范围的工作节点失败。例如，在一次MapReduce操作期间，运行系统上的网络维护导致了一组约80台机器在同一时间无法访问，持续了数分钟。MapReduce主节点只是简单的重执行了已由无法访问的机器完成的任务，并继续向前执行，最终完成了这次MapReduce操作。

主节点错误

一种简单的方法是令主节点定期将上面描述的数据结构保存为恢复点。如果主节点任务失败，就可以从上一个恢复点状态启动一个新的程序副本。但是给定的条件是只有一个主节点，它也不太可能失败；因此我们当前的实现会在主节点失败时中止MapReduce计算 。客户可以检查到这一情况，并在他们需要时重启MapReduce操作。

出现故障时的语义

当用户提供的map和reduce操作对于它们输入的值都是确定性的，我们的分布式实现产生的输出值就如同将整个程序分成一个不间断的串行执行过程一样。

为了实现这个性质，我们依赖于map和reduce任务输出结果的提交是原子的。每个处理中的任务都会将它的输出写入私有的临时文件中。一个reduce任务产生一个这样的文件，而一个map任务则产生R个这样的文件（每个reduce任务一个）。当map任务完成时，工作节点发送给主节点的消息中带有R个临时文件的名字。如果主节点收到了一个来自已完成节点的完成消息，它就会忽略这个消息。否则，主节点会将R个文件的名字记录在相应的数据结构中。

当reduce任务完成时，工作节点会执行原子性的更名操作，将临时输出文件更名为最终输出文件。如果相同的reduce任务在多个机器上执行，就会有多个更名调用应用在相同的最终输出文件上。我们依赖于由底层文件系统提供的原子更名操作，才能保证最终的文件系统中只包含由其中一个reduce执行产生的数据。

我们的绝大多数map和reduce操作都是确定性的，这种情况下我们的语义和一个串行执行过程是等同的，这也使程序员很容易推出他们程序的行为。当map和reduce操作有不确定性时，我们提供较弱但仍然合理的语义。当存在不确定的操作时，某个reduce任务R_1的输出等价于一个不确定程序的串行执行输出。但某个reduce任务R_2的输出可能符合这个不确定程序的另一个串行执行输出。

考虑map任务M和reduce任务R_1、R_2。令$e(R_i)$为R_i的已提交的执行结果（只执行一次）。此时弱语义生效，因为$e(R_1)$可能读取了M的一次输出，而$e(R_2)$则可能读取了M的另一次输出。

3.4 局部性

在我们的计算环境中，网络带宽是一种比较稀缺的资源。我们利用下面的事实来节省带宽：输入数据（由GFS管理）就存储在组成集群的机器的本地磁盘上。GFS将每个文件分成64MB大小的区块，每块复制若干份（通常为3份）存储到不同的机器上。MapReduce主节点会把输入文件的位置信息考虑进去，并尝试将map任务分配到保存有相应输入数据的机器上。如果失败的话，它会试图将map任务调度到临近这些数据的机器上（例如与保存输入数据的机器处于同一网关的工作节点）。当在一个集群的相当一部分节点上运行MapReduce操作时，大多数输入数据都是本地读取，并不消耗网络带宽。

3.5 任务粒度

如上所述，我们将map阶段分成M份，将reduce阶段分成R份。理想情况下，M和R应该比工作节点机器的数量大很多。每个工作节点处理很多不同的任务，可以增强动态负责均衡能力，也能加速有工作节点失败时的恢复情况：失败节点已经完成的map任务有很多的时候也能传递给其它所有工作节点来完成。

在我们的实现中M和R的数量有一个实际的上限：如上所述，主节点必须做O(M+R)的调度决定以及在内存中保持O(M*R)个状态。（但是内存使用的常数项很小：O(M*R)个状态中每个map/reduce任务对只需要差不多1字节数据。）

进一步分析，R通常由用户指定，因为每个reduce任务都会产生一个独立的输出文件。在实践中我们倾向于这样选择M，即可以将每个单独的任务分成16-64MB大的输入数据（此时上面所说的局部性优化效果最好），同时我们令R为待使用的工作节点数量较小的整数倍。我们经常使用M=200000，R=5000，使用2000台机器来运行MapReduce计算。

3.6 备用任务

导致MapReduce操作用时延长的一个常见原因是出现了“落后者”：某台机器在完成最后的一个map或reduce任务时花费了反常的漫长时间。很多原因都会导致落后者的产生。例如，一台磁盘损坏的机器可能会遭遇频繁的可校正错误，导致它的读取性能从30MB/s降至1MB/s。集群调度系统可能已经调度了其它任务到这台机器，导致它在执行MapReduce代码时因为CPU、内存、本地磁盘或网络带宽的竞争而更加缓慢。最近我们遇到的一个问题是机器的初始化代码有一个bug，导致处理器缓存被禁用：受影响的机器上的计算速度下降了超过100倍。

我们有一个通用的机制来减轻落后者问题。当MapReduce操作接近完成时，主节点会将仍在处理中的剩余任务调度给其它机器作备用执行。原本的执行和备用执行中的任一个完成时都会将对应任务标记为已完成。我们已经调整过这个机制，使它因这个机制而增加的计算资源消耗通常只有一点点。我们已经观察到这一机制有效地减少了大型MapReduce操作花费的时间。例如，5.3节中的排序程序在禁用这一机制时要多花费44%的时间。

4 技巧

尽管仅仅用map和reduce函数提供的基本功能就足够解决大多数需求了，我们还是发现了一些很有用的扩展。这些扩展将在本节进行描述。

4.1 划分函数

MapReduce的用户指定他们想要的reduce任务/输出文件的数量。通过划分函数可以将数据按中间key划分给各个reduce任务。我们默认提供了散列函数当作默认的划分函数（例如，hash(key) mod R)。通常这就能得出很平衡的划分结果了。但在有些情况下，用key的其它信息来划分数据也很有帮助。例如有时输出的key都是URL，而我们想让所有来自同一主机的项最后都在同一个输出文件中。为了支持类似这样的情况，MapReduce的用户可以提供一个特殊的划分函数。例如，使用“hash(主机名(urlkey)) mod R”来解决上面的问题。

4.2 顺序保证

我们保证在给定的划分中，中间key/value对是按增序排列的。这个顺序保证使每个划分产生一个有序的输出文件变得很容易，当输出文件的格式需要支持高效的按key随机访问，或用户需要输出数据有序时，这一性质会非常有用。

4.3 合并函数

某些情况中，不同的map任务产生的中间key重复率非常高，而且用户指定的reduce函数可进行交换组合。一个典型的例子就是2.1节中的单词统计。单词频率符合齐夫分布（百度百科），因此每个map任务都会产生非常多的<the, 1>这样的记录。所有这些记录都会通过网络被发送给一个reduce任务，然后再通过reduce函数将它们相加，产生结果。我们允许用户指定一个可选的合并函数，在数据被发送之前进行局部合并。

合并函数由每个执行map任务的机器调用。通常合并函数与reduce函数的实现代码是相同的。它们唯一的区别就是MapReduce库处理函数输出的方式。reduce函数的输出会写到最终的输出文件中，而合并函数的输出会写到中间文件中，并在随后发送给一个reduce任务。

4.4 输入和输出类型

MapReduce库支持多种不同格式输入数据的读取。例如，“text”模式将每行输入当作一个key/value对：key是文件中的偏移，而value则是该行的内容。另一种支持的常见格式将key/value对按key排序后连续存储在一起。每种输入类型的实现都知道如何将它本身分成有意义的区间从而能分成独立的map任务进行处理（例如text模式的区间划分保证了划分点只出现在行边界处）。用户也可以通过实现一个简单的reader接口来提供对新的输入类型的支持，尽管大多数用户只是使用为数不多的几种预定义输入类型中的一种。

reader的实现不一定要提供从文件中读数据的功能。例如，可以很容易的定义一个从数据库读记录的reader，或是从映射在内存中的某种数据结构中。

类似的，我们也支持一组输出类型来产生不同格式的数据，而用户提供代码去支持新的输出类型也不难。

4.5 边界效应

有些情况下，MapReduce的用户发现从他们的map或reduce操作中产生一些额外的辅助文件很有帮助。我们依赖于应用作者来确保这样的边界效应是原子且幂等的。通常应用会写一个临时文件，并在文件生成完毕时将其原子的更名。

我们不提供在单一任务产生的多个输出文件中原子的两段提交。因此，如果一个任务产生多个输出文件，且要求有跨文件的一致性，它必须是确定性的。这个限制在实践中还没有引起过问题。

4.6 略过坏记录

有时用户代码中的bug会导致map或reduce函数一遇到特定的记录就崩溃。这些bug会导致MapReduce操作无法完成。通常的应用措施是修复这些bug，但有时难以实现：也许bug存在于得不到源代码的第三方库中。同样地，有时忽略一些记录是可以接受的，例如在一个大数据集上做统计分析时。我们提供了一个可选的执行模式，在MapReduce库检测到确定会导致崩溃的记录时路过它们从而继续进度。

每个工作进程都要安装一个信号处理程序来捕捉违规操作和总线错误。在调用用户的map或reduce操作前，MapReduce库将用于验证的序列号保存在全局变量中。如果用户代码产生了信号，信号处理程序就将一个包含这个序列号的“最后一步”UDP包发送给MapReduce主节点。当主节点在某个记录上发现了超过一个错误时，就表明下一次重执行相应的map或reduce任务时要跳过这个记录。

4.7 本地执行

map或reduce函数的调试问题常常令人难以捉摸，因为实际的计算过程都发生在分布式系统上，经常包含数千台机器，而工作分配决策都由主节点动态产生。为了帮助调试、性能分析、小范围测试，我们开发了MapReduce库的一个替代实现，可以将MapReduce操作的全部工作在一台本地机器上顺序执行。用户拥有控制权，因此计算可以被限制在特定的map任务中。用户调用他们的程序时加上一个特殊标志，就可以方便的使用任何有用的调试或测试工具。

4.8 状态信息

主节点内置了一个HTTP服务器，可以将当前状态输出为一组网页供用户使用。状态网页能显示计算的进度，例如有多少任务被完成，多少正在处理，输入数据大小，中间数据大小，输出数据大小，处理速度，等等。这些网页还包含指向每个任务的stdout和stderr输出文件的链接。用户可以用这些数据来预测计算要花费多长时间，以及是否应该增加计算使用的资源。这些网页也能用于在计算速率比预期慢很多时发现这一情况。

另外，顶层的状态网页还能显示哪些工作节点失败了，它们失败时正在处理哪些map和reduce任务。当要在用户代码中确定bug时这些时间非常有用。

4.9 计数器

MapReduce库提供了计数器机制，可以统计多种事件的发生次数。例如，用户代码可能想统计已处理的单词总数，或被索引的德语文献的数量等。

为了使用这一机制，用户代码需要创建一个有名的计数器对象，并在map和reduce函数的适当位置增加它的值。例如：

```
Counter *uppercase = GetCounter("uppercase");

map(String name, String contents):
    for each word w in contents:
        if (IsCapitalized(w)):
            uppercase->Increment();
        EmitIntermediate(w, "1");
```

工作节点上的计数器值会定期发送给主节点（附在ping的回应里）。当MapReduce操作完成时，主节点会将运行成功的map和reduce任务发来的计数器值合并后返回给用户代码。当前的计数器值也会显示在主节点的状态网页上，其它人可以看到实时的计算进度。在合并计数器值时，主节点会忽略同一个map或reduce任务的重复的结果，从而避免多次叠加。（重复执行可能发生在我们的备用任务和失败节点重执行中。）

有些计数值是由MapReduce库自动维护的，例如输入key/value对已处理的数量和输出key/value对产生的数量等。

用户们观察到这一机制在对MapReduce操作的智能检查上很有帮助。例如，在一些MapReduce操作中，用户代码可能想要确认产生的输出对的数量恰好与输入对的数量相等，或是已处理的德语文献占处理文献总数的比例是在接受范围内的。

5 性能

本节中我们会在一个大型集群上测量MapReduce在两个计算上的性能。一个计算是在差不多1TB的数据中查找指定的模式。另一个计算则是排序1TB的数据。

这两个程序能代表大量真实的MapReduce程序——一类程序是将数据从一种表示形式转换为另一种，另一类程序则是从很大数量的数据集中提取出一小部分感兴趣的数据。

5.1 集群配置

所有的程序都运行在由1800台机器组成的集群上。每台机器配有：2GHz的Intel Xeon处理器且支持超线程，4GB内存，2块160GB的IDE硬盘，和1GB以太网连接。所有机器都安排在2层树结构的网关内，根节点的可用带宽加起来有100-200Gbps。所有的机器都处于相同的托管设施下，因此任意两台机器间的往返通信时间都少于1ms。

除了4GB的内存，还有1-1.5GB的内存保留给了集群上运行的其它任务。程序运行的时间是一个周末的下午，此时大多数CPU、磁盘和网络资源都空闲中。

5.2 Grep

grep程序要扫描10^{10}个长度为100字节的记录来寻找一个相当罕见的三字符模式（这个模式出现于92337个记录中）。输入被切分成64MB大小的部分（M = 15000），整个输出为一个文件（R = 1）。

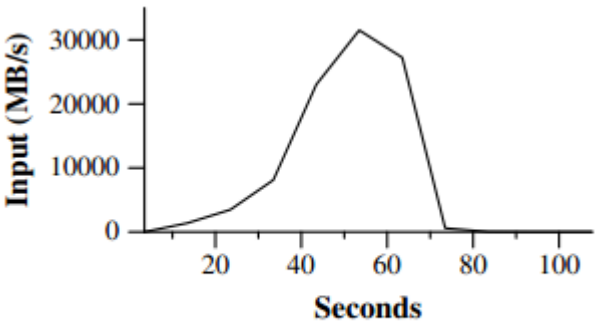

Figure 2: Data transfer rate over time

图2显示了随时间变化的计算进度。Y轴是输入数据扫描的速率。这个速率逐渐提升代表更多的机器被分配给MapReduce计算，并在分配的机器数达到1764台时超过了30GB/s。随着map任务的结束，这个速率开始下降，并在计算开始后80s时降至0。整个计算过程共花费约150s。这包括了大约1分钟的启动开销。启动开销是由于要把程序传播到所有工作机器上，还包括GFS要打开1000个输入文件和获取局部优化所需信息而导致的延迟。

5.3 排序

排序程序要对10^{100}个100字节的记录进行排序（差不多1TB数据）。这个程序模仿了TeraSort测试程序。

排序程序只包括不到50行的用户代码。map函数一共3行，它从一个文本行中提取出10字节的排序key，再将key和原始文本行输出为中间key/value对。我们使用了一个内置的Identity函数作为reduce函数。这个函数会将未更改的中间key/value输出为结果。最终的排序后输出被出到一组2路复制的GFS文件（例如，这个程序的输出要写2TB的数据）。

如前所述，输入数据被分成若干个64MB大小的部分（M = 15000）。我们将已排序的输出分成4000个文件（R = 4000）。划分函数使用key的首字节来将它分到R个文件中的一个。

此次测试中我们的划分函数使用了key分布的内建知识。在一个一般的排序程序中，我们会预先加一轮MapReduce操作，收集key的样本，并使用key抽样的分布来计算最终输出文件的划分点。

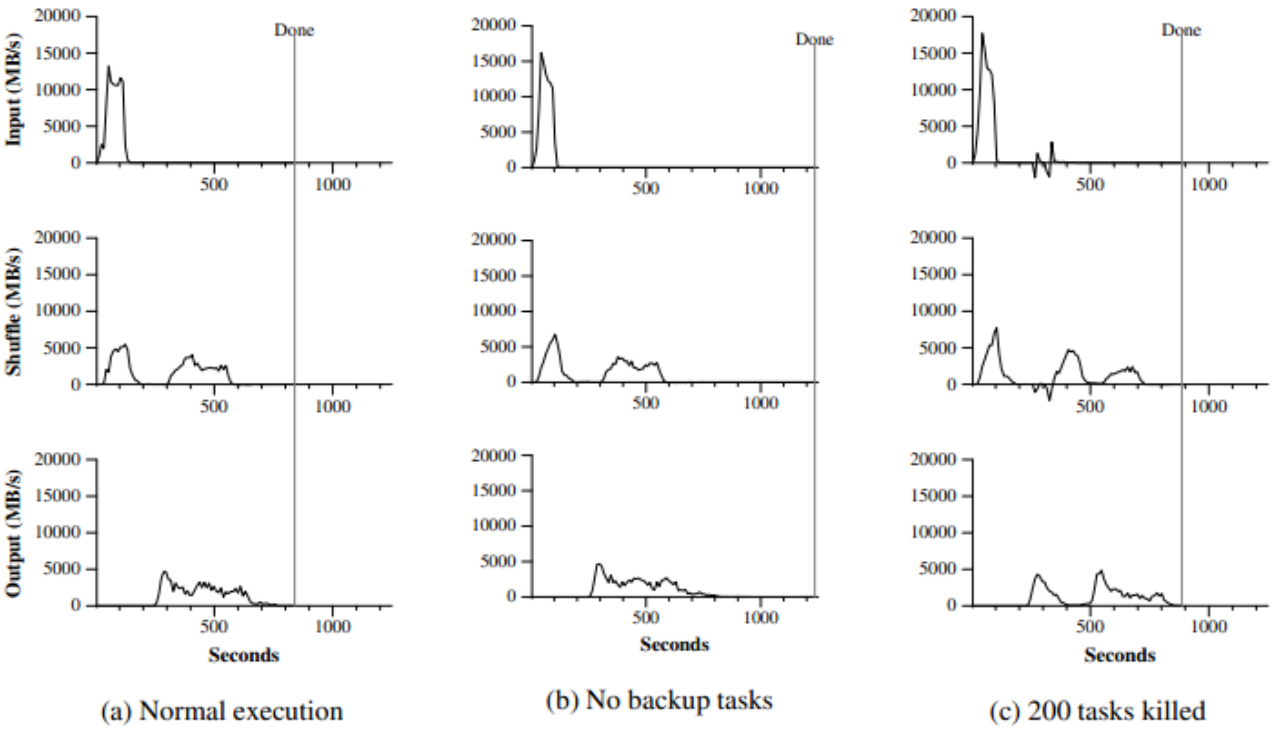

Figure 3: Data transfer rates over time for different executions of the sort program

图3(a)是排序程序的正常执行过程。上边的图显示了输入读取的速度。输入速度的峰值为13GB/s，并在200秒后所有的map任务都完成时迅速下降至0。可以注意到这个速度要比grep的速度小。这是因为排序的map任务要在向本地磁盘写中间文件上花费大约一半的时间和I/O带宽。而grep中相应的中间输出则可以忽略不计。

中间的图显示了数据从map节点通过网络向reduce节点发送的速度。这个移动开始于第一个map任务完成。图中的第一个驼峰出现在reduce任务第一次达到1700个时（整个MapReduce共分配到1700台机器，每台机器同时最多只运行一个reduce任务）。计算开始差不多300秒时，第一批reduce任务已经有部分完成，我们开始向剩余的reduce任务传送数据。所有传送过程在计算开始600秒后结束。

下面的图是reduce任务向最终的输出文件写入已排序数据的速度。在第一个传输周期的结束与写入周期的开始之间有一个延迟，因为此时机器正在排序中间数据。写入速度在2-4GB/s下持续了一段时间。所有的写操作在计算开始后850秒左右结束。包括启动开销的话，整个计算花费了891秒。这与TeraSort上目前公布的最佳记录1057秒很接近。

一些要注意的事：因为我们的局部性优化，输入速度要比传播速度和输出速度都快——大多数数据都读自本地磁盘，绕开了我们带宽相当有限的网络。传播速度要比输出速度高，因为

输出阶段要写两份已排序的数据（因为可靠性和实用性的考虑）。我们写两份输出是因为这是我们的底层文件系统针对可靠性和实用性提供的机制。如果底层文件系统使用擦除代码而不是复制，那么写数据需要的网络带宽就会减少。

5.4 备用任务的影响

在图3(b)中显示的是禁用了备用任务情况下排序程序的执行过程。这种情况下的执行过程与图3(a)中的很相似，但在末尾处有很长一段时间几乎没有任何的写操作发生。在960秒后，只有5个reduce任务还没有完成。但这最后的几个任务直到300秒后才结束。整个计算花费了1283秒，比正常情况多花费44%的时间。

5.5 机器失败

在图3(c)中显示了有机器失败情况下的排序程序执行过程。在计算开始几分钟后，我们有意的杀掉了1746个工作节点中的200个。底层集群调度器立即在这些机器上重启了新的工作进程（只有进程被杀掉了，机器还在正常运行中）。

工作节点的死亡显示为一个负的输入速度，因为一些已经完成的map任务消失了（因为对应的map节点被杀掉了）需要重新完成。这些map任务的重执行很快就会发生。整个计算过程在933秒后结果，包括了启动开销（相比正常执行时间只增加了5%）。

6 经验

我们在2003年2月写出了MapReduce库的第一个版本，并在2003年8月进行了非常大的改进，包括了局部性优化、各工作机器上任务执行的动态负载平衡等。从那时开始，我们愉快地惊讶于MapReduce被如此广泛地应用在我们工作中遇到问题上。它已被用于Google的很多领域中，包括：

- 大规模机器学习问题；
- 针对Google News和Froogle产品的集群问题；
- 用于产生流行需求报告的数据提取（例如Google Zeitgeist）；
- 针对新的尝试和产品的网页属性提取（例如，从大量的定位搜索中提取出地理位置信息）；

- 大规模的图计算。

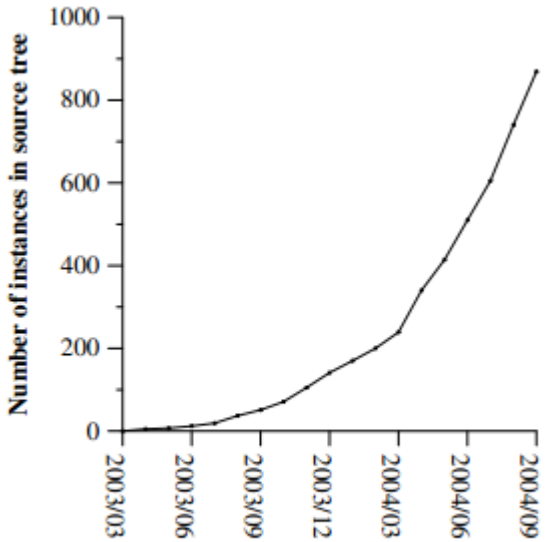

Figure 4: MapReduce instances over time

图4显示了在我们的源代码管理系统中进行了登记的MapReduce程序的数量随时间的显著增长，从2003年初的0到2004年9月底的接近900个不同版本。MapReduce取得如此成功的原因是它令花半小时时间写出一个简单的程序并运行在一千台机器上成为了可能，这极大地加速了开发和原型实现循环。进一步说，它允许没有分布式与并行系统经验的程序员也能轻松地利用大量的资源。

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Table 1: MapReduce jobs run in August 2004

在每项作业的结尾，MapReduce库会将作业花费的计算资源统计写入日志。在表1中，我们能看到Google在2004年8月运行的MapReduce作业的一部分统计情况。

6.1 大规模索引

迄今为止我们的一个最重要的MapReduce应用是完全重写了生产索引系统，它负责产生用于Google网络搜索服务的数据结构。索引系统把由我们的爬取系统检索的一个大的文档集合作为输入，并存储为一组GFS文件。这些文档的原始文本数据超过了20TB。索引进程分成了5-10个MapReduce操作运行。使用MapReduce（而不是之前版本使用的ad-hoc分布系统）提供了许多好处：

- 索引代码更简单，更小，也更易懂，因为处理容错性和分布与并行的代码都被隐藏在MapReduce库中了。例如，其中一个阶段的计算在使用了MapReduce后从差不多3800行C++代码降到了700行代码。

- MapReduce库的性能足够好，因此我们可以令概念上无关的计算过程相互分离，而不需要为了避免额外的数据处理而将它们混合在一起。这减小了修改索引进程的难度。例如，在我们的旧索引系统中进行一个修改要花费几个月时间，而在新系统中只需要几天时间。

- 索引进程更容易去操作，因为大多数由机器失败、部分机器缓慢和网络暂时中断引起的问题都由MapReduce库自动处理了，不需要操作者干预。此外，通过向集群中增加机器，可以很容易地增强索引进程的性能。

7 相关工作

许多系统都提供了约束好的程序模型，并利用这些约束自动的进行并行计算。例如，一个组合函数可以在N个处理器上使用并行前缀计算用log N的时间计算出包含N个元素的数组的所有前缀。MapReduce可以被认为是基于来自我们在真实世界中的大型计算的经验，对这些模型的简单化和提炼。更重要的是，我们提供了可以扩展到上千个处理器上使用的容错机制。作为对比，大多数并行处理系统只能在小规模下实现，并将处理机器错误的细节留给程序员去完成。

Bulk Synchronous Programming和一些MPI原语提供了更高层次的抽象，更容易写出并行程序。这些系统和MapReduce的一个关键区别是MapReduce利用了一个受约束的程序模型去自动并行化用户程序，并提供透明的容错机制。

我们的局部性优化受到了诸如活动磁盘等技术的启发，即将计算推给靠近本地磁盘的处理单元，来减少要跨过I/O子系统或网络发送的数据总量。我们用运行在直接连接了少量磁盘的商用处理器上替代了直接运行在磁盘控制处理器上，但通用的方法是相似的。

我们的备用任务机制与Charlotte系统上的eager调度机制类似。简单的eager调度系统有一个缺点，就是如果某个任务导致了多次失败，整个计算都会失败。我们通过自己的略过坏记

录的机制解决了一些这类问题。

MapReduce的实现依赖于一个内部的集群管理系统，它负责在一组数量很多的共享机器上分布运行用户任务。尽管不是本文的重点，这个集群管理系统与Condor等其它系统在原理上是类似的。

排序机制作为MapReduce库的一部分，与NOW-Sort操作是类似的。源机器（map节点）切分待排序的数据并将其发送给R个reduce节点。每个reduce节点在本地排序它的数据（尽量存放在内存中）。当然NOW-Sort没有用户定义的map和reduce函数，这些使我们的库被广泛应用。

River提供了一个编程模型，通过在分布式队列上发送数据来处理通信。类似于MapReduce，River系统试图提供提供好的平均情况性能，即使存在因不统一的硬件或系统扰动而导致的不一致性。River通过小心的调度磁盘和网络传输来实现平衡的完成时间，从而达到这一目标。MapReduce使用了不同的方法。通过约束编程模型，MapReduce框架能够将问题划分成大量细粒度的任务。这些任务被动态调度给可用的工作节点，因此更快的节点会处理更多的任务。这个约束的编程模型同样允许我们在临近作业结束时将剩余的任务重复调度执行，这显著的减少了存在不一致时（例如有缓冲或停顿的机器）的完成时间。

BAD-FS有着与MapReduce非常不同的编程模型，而且不像MapReduce，它面向跨越广域网的作业执行。但是，它与MapReduce有着两个本质上的相同点：

1. 两个系统都使用了备用执行来恢复因错误而导致的数据丢失。

2. 两个系统都使用了局部感知调度来减少在拥堵的网络连接上发送的数据量。

TACC被设计为一个高度可用的网络服务的简化结构系统。类似于MapReduce，它依赖于重执行机制来实现容错性。

8 结论

MapReduce编程模型已经成功的用于Google的许多不同的用途中。我们将它的成功归功于几个原因。首先，该模型即使是对于没有并行和分布式系统经验的程序员也是很易于使用的，因为它隐藏了并行化、容错机制、局部性优化、以及负载平衡的细节。其次，很多种类的问题都很容易表示成MapReduce计算。例如，MapReduce被用于为Google的网络搜索服务的数据产生、排序、数据挖掘、机器学习、以及许多其它系统。第三，我们已经开发了

一个MapReduce的实现，可以扩展到包含上千台机器的大型集群。该实现可以高效使用这些机器的资源，因此适合于Google遇到的很多大型计算问题。

我们从这项工作中学到了很多东西。首先，约束这个编程模型令并行和分布式计算，以及令这些计算可容错，变得简单了。其次，网络带宽是一种稀缺资源。我们系统中的很多优化都因此针对减少通过网络发送的数据总量：局部性优化允许我们从本地磁盘读，同时将中间文件写入本地磁盘也节省了网络带宽。第三，备用执行可以用于减小缓慢的机器的影响，及应对机器失败和数据丢失。

附录A 词频统计源代码

```
#include "mapreduce/mapreduce.h"

// User's map fuction
class WordCounter: public Mapper {
public:
        virtual void Map(const MapInput &input) {
                const string &text = input.value();
                const int n = text.size();
                for (int i = 0; i < n; ) {
                        // Skip past leading whitespace
                        while ((i < n) &&
isspace(text[i]))
                                ++i;

                        // Find word end
                        int start = i;
                        while ((i < n) &&
!isspace(text[i]))
                                ++i;

                        if (start < i)
                                Emit(text.substr(start, i-
start), "1");
                }
        }
};
REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder: public Reducer {
        virtual void Reduce(ReduceInput *input) {
                // Iterate over all entries with the
                // same key and add the values
                int64_t value = 0;
                while (!input->done()) {
```

```
                value += StringToInt(input->value());
                input->NextValue();
            }

            // Emit sum for input->key()
            Emit(IntToString(value));
        }
};
REGISTER_REDUCER(Adder);

int main(int argc, char **argv) {
        ParseCommandLineFlags(argc, argv);

        MapReduceSpecification spec;

        // Store list of input files into "spec"
        for (int i = 1; i < argc; ++i) {
                MapReduceInput *input = spec.add_input();
                input->set_format("text");
                input->set_filepattern(argv[i]);
                input->set_mapper_class("WordCounter");
        }

        // Specify the output files:
        //    /gfs/test/freq-00000-of-00100
        //    /gfs/test/freq-00001-of-00100
        //    ...
        MapReduceOutput *out = spec.output();
        out->set_filebase("/gfs/test/freq");
        out->set_num_tasks(100);
        out->set_format("text");
        out->set_reducer_class("Adder");

        // Optional: do partial sums within map
        // tasks to save network bandwidth
        out->set_combine_class("Adder");

        // Tuning parameters: use at most 2000
        // machines and 100MB of memory per task
        spec.set_machines(2000);
        spec.set_map_megabytes(100);
        spec.set_reduce_megabytes(100);

        // Now run it
        MapReduceResult result;
        if (!MapReduce(spec, &result))
                abort();

        // Done: 'result' structure contains info
        // about counters, time taken, number of
        // machines used, etc.
```

```
84.
85.         return 0;
86.     }
```

[翻译]MapReduce: Simplified Data Processing on Large Clusters的更多相关文章

1. 《MapReduce: Simplified Data Processing on Large Clusters》论文研读
   MapReduce 论文研读 说明:本文为论文 <MapReduce: Simplified Data Processing on Large Clusters> 的个人理解,难免有理解不 ...
2. 《MapReduce: Simplified Data Processing on Large Cluster 》翻译
   Abstract MapReduce是一种编程模型和一种用来处理和产生大数据集的相关实现.用户定义map函数来处理key/value键值对来产生一系列的中间的key/value键值对.还要定义一个re ...
3. 翻译-In-Stream Big Data Processing 流式大数据处理
   相当长一段时间以来,大数据社区已经普遍认识到了批量数据处理的不足.很多应用都对实时查询和流式处理产生了迫切需求.最近几年,在这个理念的推动下,催生出了一系列解决方案,Twitter Storm,Yah ...
4. In-Stream Big Data Processing
   http://highlyscalable.wordpress.com/2013/08/20/in-stream-big-data-processing/ Overview In recent y ...
5. Linux command line exercises for NGS data processing
   by Umer Zeeshan Ijaz The purpose of this tutorial is to introduce students to the frequently used to ...
6. 翻译——1_Project Overview, Data Wrangling and Exploratory Analysis-checkpoint
   为提高提高大学能源效率进行建筑能源需求预测 本文翻译哈佛大学的能源分析和预测报告,这是原文 暂无数据源,个人认为学习分析方法就足够 内容: 项目概述 了解数据 探索性分析 使用不同的机器学习方法进行预 ...
7. SQL Server Reporting Services 自定义数据处理扩展DPE(Data Processing Extension)
   最近在做SSRS项目时,遇到这么一个情形:该项目有多个数据库,每个数据库都在不同的服务器,但每个数据库所拥有的数据库对象(table/view/SPs/functions)都是一模一样的,后来结合网络 ...
8. Dubbo Data length too large: 11557050, max payload: 8388608 传输数据超限

com.alibaba.dubbo.remoting.transport.AbstractCodec.checkPayload() ERROR Data length too large: 11557 ...

9. dubbo报错Data length too large: 10710120处理，及服务提供者协议配置详细说明
   工作中遇到以下报错信息
   cause: java.io.IOException: Data length too large: 10710120, max payload: 8388608, chann ...

随机推荐

1. Jquery自定义扩展方法（二）--HTML日历控件
   一.概述 研究了上节的Jquery自定义扩展方法,自己一直想做用jquery写一个小的插件,工作中也用到了用JQuery的日历插件,自己琢磨着去造个轮子--HTML5手机网页日历控件,废话不多说,先看 ...
2. Python基础（10）--数字
   本文的主题是 Python 中的数字.会详细介绍每一种数字类型,它们适用的各种运算符,以及用于处理数字的内建函数.在文章的末尾, 简单介绍了几个标准库中用于处理数字的模块. 本文地址:http:// ...
3. Solr5.3.1 SolrJ查询索引结果
   通过SolrJ获取Solr检索结果 1.通过SolrParams的方式提交查询参数 SolrClient solr = new HttpSolrClient("http://localhos ...
4. [C#] 常用工具类——系统日志类
   using System; using System.Collections.Generic; using System.Text; using System.Diagnostics; namespa ...
5. Spring MVC程序中怎么得到静态资源文件css,js,图片文件的路径问题
   问题描述 在用springmvc开发应用程序的时候.对于像我一样的初学者,而且还是自学的人,有一个很头疼的问题.那就是数据都已经查出来了,但是页面的样式仍然十分简陋,加载不了css.js,图片等资源文 ...
6. 50A
   #include <iostream> using namespace std; int main() { int m, n; cin>>m>>n; cout< ...
7. 连接远程数据库时出现 SSH: expected key exchange group packet from server / 2003 - Can't connect to MySQL server on 'XXX' (10038) / 1130 - Host 'XXX' is not allowed to connect to this MySQL server

昨天在自己的远程服务器上玩,把系统重装了.新装了MySQL,在本地用navicat连接的时候出了几个小问题. 问题一:SSH: expected key exchange group packet f ...

8. jquery 格式化数字字符串(小数位)
   用于页面上格式化数字字符串,此代码为工作时所需,留作笔记,比较常用. /** * author: xg君 * 描述: 格式化数字字符串,格式化小数位 * obj为需要格式的对象(例如:input标签) ...
9. bzoj3223: Tyvj 1729 文艺平衡树 splay裸题
   splay区间翻转即可 /** Problem: 3223 User: walf ...
10. MySQL 使用 SSL 连接(附 Docker 例子)
    查看是否支持 SSL 首先在 MySQL 上执行如下命令, 查询是否 MySQL 支持 SSL: mysql> SHOW VARIABLES LIKE 'have_ssl'; +-------- ...

热门专题

国新测试证书-1.PFX | 带参数执行CMD文件 | PF_RING DNS模式免LISENCE | SYS.DM_TRAN_LOCKS 表中的内容可以全部清除吗 | REACT INPUT 获取光标 | ISE 14.7 补丁 | PYTHON的XLUTILS如何把原有的格式复制过来 | WINNT网络访问权限 | 单表达到多少时候需要拆分 | RPCBIND停止服务 | LOADRUNNER上传图片 | WINDOWSAPI 点击 | DJANGO REST FRAMEWORK PK 传参 | PROMETHEUS FOR 持续时间,变量 | PYTHON中如何实现SUBSTITUTE | SPRINGCLOUD OAUTH2 自定义异常处理 | CAN网络上同时存在扩展帧和标准帧 | PYTHON2怎么连接DB2数据库 | PYCHARM连接WSL下的MYSQL | C# 非广播模式 远程唤醒