MapReduce。简化大型集群上的数据处理

Jeffrey Dean和Sanjay Ghemawat

jeff@google.com。 sanjay@google.com

Google, Inc.

摘要

MapReduce是一个编程模型和一个相关的实现，用于处理和生成大型数据集。用户指定一个*map*函数来处理一个键/值对，生成一组中间键/值对，并指定一个*reduce*函数来合并与同一中间键相关的所有中间值。许多现实世界的任务都可以用这个模型来表达，如论文中所示。

以这种函数式风格编写的程序被自动并行化，并在一个大型的组合机集群上执行。运行时系统负责分割输入数据的细节，在一组机器上调度程序的执行，处理机器故障，以及管理所需的机器间通信。这使得没有任何并行和分布式系统经验的程序员可以轻松地利用大型分布式系统的资源。

我们对MapReduce的实现是在一个大型的商品机集群上运行的，并且具有很强的可扩展性：一个典型的MapReduce计算会在数千台机器上处理许多太字节的数据。程序员们发现这个系统很容易使用：数以百计的MapReduce项目已经被实施，每天有超过1000个MapReduce作业在谷歌的集群上执行。

1 简介

在过去的五年里，作者和谷歌的许多其他人已经实现了数百种特殊用途的计算，这些计算处理大量的原始数据，如抓取的文件、网络请求日志等，以计算各种衍生数据，如倒置指数、网络文件图形结构的各种表示、每个主机抓取的网页数量的总结、一个网站中最频繁查询的集合等。

既定日期，等等。大多数这样的计算在概念上是直截了当的。然而，输入的数据通常很大，计算必须分布在数百或数千台机器上，以便在合理的时间内完成。如何对计算进行分工，分配数据，以及处理故障，这些问题合谋着用大量复杂的代码来处理这些问题，从而掩盖了原来的简单计算。

作为对这种复杂性的反应，我们设计了一个新的抽象，它允许我们表达我们试图执行的简单计算，但将并行化、容错、数据分配和负载平衡等混乱的尾巴隐藏在一个库中。我们的抽象是由Lisp和许多其他函数式语言中的*map*和*reduce*原语来实现的。我们意识到，我们的大多数计算涉及到对我们输入的每个逻辑"记录"应用*地图*操作，以计算一组中间的键/值对，然后对所有共享相同键的值应用*还原*操作，以适当地组合衍生的数据。我们使用用户指定的映射和还原操作的功能模型，使我们能够轻松地解析大型计算，并将重新执行作为容错的主要机制。

这项工作的主要贡献是一个简单而强大的接口，能够实现大规模计算的自动并行化和分布，再加上这个接口的实现，在大型商品PC集群上实现了高性能。

第2节描述了基本的编程模型并给出了几个例子。第3节描述了针对我们基于集群的计算环境的MapReduce接口的实现。第4节描述了我们发现的对编程模型的一些细化。第5节介绍了我们对各种任务实现的性能测量。第6节探讨了MapReduce在Google中的应用，包括我们将其作为基础的经验。

我们的生产索引系统的重写。第7节讨论了相关的和未来的工作。

2 编程模式

该计算接受一组*输入*键/值对，并产生一组*输出*键/值对。
MapReduce库的用户将计算表达为两个函数。*Map*和*Reduce*。

由用户编写的*Map*，接受一个输入对，并产生一组*中间*键/值对。MapReduce库将所有与同一中间键*I*相关的中间值分组，并将它们传递给*Reduce*函数。

*Reduce*函数也是由用户编写的，接受一个中间键*I*和该键的一组值。它将这些值合并起来，形成一个可能更小的值集。一般来说，每次*Reduce*调用只产生0或1个输出值。中间值通过一个迭代器提供给用户的Reduce函数。这使得我们能够处理那些大到无法在内存中容纳的数值列表。

2.1 例子

考虑到在一大批文件中计算每个词的出现次数的问题。用户将编写类似于以下伪代码的代码。

```
map(String key, String value)。
  // 关键：文件名称
  // value：value中每个单词w的
  文件内容。
    EmitIntermediate(w, "1")。

reduce(String key, Iterator values)。
  // 关键：一个词
  // 数值：一个计数列表 int
  result = 0;
  for each v in values:
    result +=
    ParseInt(v);
  Emit(AsString(result))。
```

map函数发射每个单词和相关的出现次数（在这个简单的例子中只有'1'）。reduce函数将一个特定单词的所有计数相加。

此外，用户编写代码，将输入和输出文件的名称以及可选的调整参数填入*mapreduce规范*对象。然后，用户调用*MapReduce*函数，把规范对象传给它。用户的代码与MapReduce库（用C++实现）连接在一起。附录A包含这个例子的完整程序文本。

2.2 类型

尽管前面的伪代码是用字符串输入和输出来写的，但从概念上讲，用户提供的map和reduce函数有相关类型。

```
map    (k1,v1)          →list(k2,v2)
reduce (k2,list(v2)) → list(v2)
```

也就是说，输入的键和值与输出的键和值来自不同的领域。此外。
中间的键和值与输出的键和值来自相同的工作。

我们的C++实现将字符串传递给用户定义的函数，并让用户代码在字符串和适当类型之间进行转换。

2.3 更多例子

这里有几个有趣的程序的简单例子，可以很容易地表达为MapReduce计算。

分布式格雷普。map函数如果与提供的模式相匹配，就会发出一行。reduce函数是一个身份函数，只是将提供的中间数据复制到输出。

URL访问频率的计数。映射函数处理网页请求的日志并输出URL，
(1)。还原函数将同一URL的所有数值相加，并发出一个
URL，总计数对。

反向网络链接图。map函数为在名为source的页面中发现的每个目标URL的链接输出目标、源对。还原函数将与给定的目标URL相关的所有源URL的列表连接起来，并发出一对：（目标，*列表*（源））。

每个主机的术语向量。术语向量将一个文件或一组文件中出现的最重要的词总结为一个*词，频率*对的列表。map函数为每个输入的文档发出一个主机名，术语向量对（其中主机名是从
文档的URL中提取的）。reduce函数被传递给一个给定的主机的所有每个文档术语向量。它把这些术语向量加在一起，扔掉不常见的术语，然后发出一个最终的（主机名，术语向

量）对。

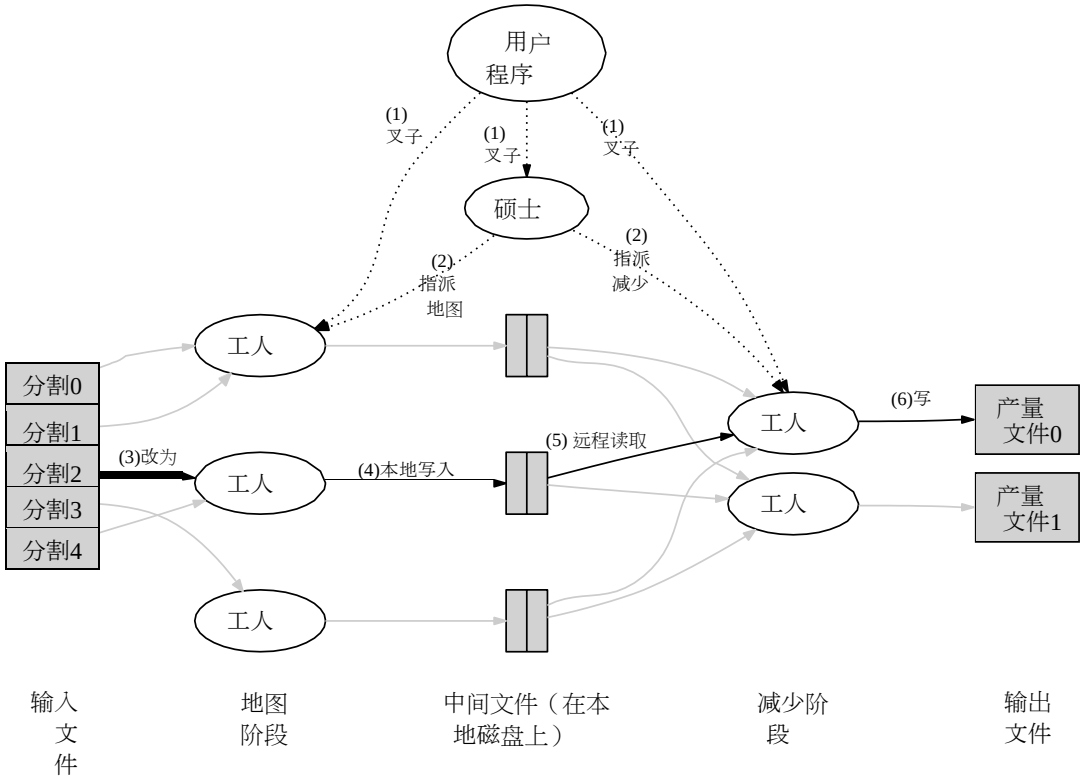


图1：执行情况概述

倒置索引。map函数解析每个文档，并发出一个单词(，文档ID)对的序列。reduce函数接受一个给定的词的所有(对，对相应的文档ID进行排)序，并发出一个词，*list*(文档ID)对。所有输出对的集合形成一个简单的倒置索引。很容易对这个计算进行扩充，以保持对单词位置的跟踪。

分布式排序。map函数从每条记录中提取键，并释放出一个键，(记)录对。还原函数发出所有的配对，不作任何改变。这个编译依赖于第4.1节中描述的分区设施和第4.2节中描述的排序属性。

3 实施

MapReduce界面有许多不同的实现方式。正确的选择取决于环境。例如，一种实现可能适合小型共享内存机器，另一种适合大型NUMA多处理器，还有一种适合更大的网络机器集合。

本节描述了一个针对谷歌广泛使用的计算环境的实现。

用交换式以太网连接在一起的大型商品PC集群[4]。在我们的环境中。

(1) 机器通常是运行Linux的双处理器X86处理器，每台机器有2-4GB的内存。

(2) 使用商品网络硬件--
通常在机器层面是100兆比特/秒或1千兆比特/秒，但平均而言，所有分节带宽要小得多。

(3) 一个集群由数百或数千台机器组成，因此，机器故障是很常见的。

(4) 存储由廉价的IDE磁盘提供，直接连接到各个机器上。一个内部开发的分布式文件系统[8]被用来管理存储在这些磁盘上的数据。该文件系统使用复制技术，在不可靠的硬件上提供可用性和可靠性。

(5) 用户向调度系统提交作业。每个作业由一组任务组成，并由调度器映射到集群中的一组可用机器上。

3.1 执行概述

通过自动划分输入数据，将*地图*调用分布在多个机器上

成一组*M*个*分片*。输入的分片可以由不同的机器并行处理。*Reduce*调用是通过使用分区函数（例如，*hash*(*key*)mod *R*）将中间的密钥空间划分为*R*块来进行分配。分区的数量（*R*）和分区函数由用户指定。

图1显示了我们实现中的MapReduce操作的整体流程。当用户程序调用MapReduce函数时，会发生以下一系列的动作（图1中的数字标签与下面列表中的数字相对应）。

1. 用户程序中的MapReduce库首先将输入文件分割成*M*块，每块通常为16兆到64兆（MB）（用户可通过一个可选参数进行控制）。然后，它在一组机器上启动许多程序的副本。

2. 其中一份程序是特殊的-主人。其余的是由主程序分配工作的工作者。有*M*个地图任务和*R*个还原任务需要分配。主程序挑选空闲的工人，给每个人分配一个地图任务或一个还原任务。

3. 被分配到一个地图任务的工作者会读取相应的输入分片的内容。它从输入数据中解析出键/值对，并将每个键/值对传递给用户定义的*Map*函数。由*Map*函数产生的interme-diate key/value对被缓冲在内存中。

4. 周期性地，缓冲对被写入本地磁盘，通过分区函数被分割成*R*区域。这些缓冲对在本地磁盘上的位置被传回给主站，主站负责将这些位置转发给还原工作者。

5. 当reduce worker收到master关于这些位置的通知时，它使用远程过程调用从map worker的本地磁盘上读取缓冲数据。当一个Reduce Worker读取了所有的中间数据后，它会按照中间键进行排序，以便将所有相同键的出现归为一组。之所以需要排序，是因为通常许多不同的键会映射到同一个还原任务。如果中间数据量太大，无法在内存中容纳，就会使用外部排序。

6. Reduce工作者在排序的中间数据上进行迭代，对于每一个独特的中间密钥，它将密钥和相应的中间值集传递给用户的*Reduce*函数。*Reduce*函数的输出被附加到这个Reduce分区的最终输出文件中。

7. 当所有的map任务和reduce任务都完成后，主程序会唤醒用户程序。这时，用户程序中的MapReduce调用会返回到用户代码中。

成功完成后，mapre-duce执行的输出可在*R*输出文件中获得（每个还原任务一个，文件名由用户指定）。通常情况下，用户不需要将这些*R*输出文件合并成一个文件--他们经常将这些文件作为输入传给另一个MapReduce调用，或者从另一个能够处理被分割成多个文件的输入的分布式应用程序中使用这些文件。

3.2 主数据结构

主程序保留了几个数据结构。对于每个地图任务和还原任务，它存储状态（*空闲*、*进行中*或*已完成*），以及工作者机器的身份（对于非空闲任务）。

主站是中间文件区域的位置从地图任务传播到还原任务的渠道。因此，对于每个已完成的地图任务，主站存储了由地图任务产生的*R*个中间文件区域的位置和大小。随着地图任务的完成，会收到对这个位置和大小信息的更新。这些信息被越来越多地推送给有*正在进行的*还原任务的工作者。

3.3 容错性

由于MapReduce库被设计为使用成百上千台机器帮助处理非常大量的数据，所以该库必须优雅地容忍机器故障。

工人失败

主站定期对每个工作器进行ping。如果在一定时间内没有收到某个工作器的回复，主站就会将该工作器标记为失败。该工作器完成的任何地图任务都会被重置为最初的*闲置*状态，因此有资格在其他工作器上进行调度。同样，失败的工作器上正在进行的任何地图任务或还原任务也被重置为*空闲状态*，并有资格进行重新调度。

已完成的map任务在故障时要重新执行，因为它们的输出被存储在故障机器的本地磁盘上，因此无法访问。已完成的还原任务不需要重新执行，因为它们的输出被存储在全局文件系统中。

当一个地图任务先由工作者*A*执行，然后再由工作者*B*执行（因为*A*失败了），所有的

执行还原任务的工作者被通知重新执行。任何尚未从工作者*A*读取数据的还原任务将从工作者*B*读取数据。

MapReduce对大规模工作者的故障具有很强的适应性。例如，在一次MapReduce操作中，正在运行的集群上的网络维护导致一次80台机器的群组在七八分钟内无法连接。MapReduce主程序简单地重新执行了无法访问的工人机器所做的工作，并继续向前推进，最终完成了MapReduce操作。

主体失败

要让主任务对上述主数据结构进行定期检查点是很容易的。如果主任务死亡，可以从最后一个检查点的状态开始一个新的副本。然而，考虑到只有一个主站，它的失败是不太可能的；因此，如果主站失败，我们目前的实现就会中止MapReduce的计算。客户端可以检查这种情况，如果他们愿意，可以重试MapReduce操作。

失败情况下的语义学

当用户提供的*map*和*reduce*操作符是其输入值的去终结函数时，我们的分布式实现产生的输出与整个程序的非故障顺序执行产生的输出相同。

我们依靠map和reduce任务输出的原子提交来实现这一特性。每个进行中的任务都将其输出写入私有的临时文件中。一个还原任务产生一个这样的文件，而一个地图任务产生*R*个这样的文件（每个还原任务一个）。当一个map任务完成时，worker会向master发送一个消息，并在消息中包括*R*个临时文件的名称。如果主站收到一个已经完成的地图任务的完成消息，它将忽略该消息。否则，它会在主数据结构中记录*R*文件的名称。

当一个reduce任务完成后，reduce工作者会将其临时输出文件原子化地重命名为最终输出文件。如果同一个还原任务在多台机器上执行，那么多个重命名调用将为同一个最终输出文件执行。我们依靠底层文件系统提供的原子重命名操作来保证最终的文件系统状态只包含减少任务的一次执行所产生的数据。

我们的绝大多数*map*和*reduce*操作符都是确定性的，在这种情况下，我们的语义等同于顺序执行，这使得程序员可以很容易地推理他们的程序的行为。当*map*和/或*reduce*操作符是非确定性的，我们提供了较弱但仍然合理的规则。在有非确定性操作符的情况下。
一个特定的还原任务R_1
的输出等同于非确定性程序的顺序执行所产生的R_1的输出。然而，R的输出是
不同的还原任务R_2
可能对应于由非确定性程序的不同顺序执行产生的R_2的输出。

考虑map任务*M*和reduce任务R_1和R_2。让$e(R_i)$为R_i的执行，该执行承诺（有
正是一个这样的执行）。出现这种较弱的语义是因为$e(R_1)$可能已经读取了*M*的一个执行所产生的输出，而$e(R_2)$可能已经读取了
*观察M*的不同执行所产生的输出。

3.4 地点

在我们的计算环境中，网络带宽是一种相对稀缺的资源。我们通过利用输入数据（由GFS[8]管理）存储在构成我们集群的机器的本地磁盘上这一事实来节约网络带宽。GFS将每个文件分成64MB的块，并将每个块的几个副本（通常是3个副本）存储在不同的机器上。MapReduce主程序考虑到了输入文件的位置信息，并试图在包含相关输入数据副本的机器上安排一个地图任务。如果做不到这一点，它就会尝试将地图任务安排在该任务的输入数据副本附近（例如，安排在与包含数据的机器在同一网络交换机上的工作者机器上）。当在集群中相当一部分工作者上运行大型MapReduce操作时，大多数输入数据都在本地读取，不消耗网络带宽。

3.5 任务颗粒度

如上所述，我们将地图阶段细分为*M*块，将重构阶段细分为*R*块。理想情况下，*M*和*R*应该远远大于工作机的数量。让每个工作者执行许多不同的任务可以改善动态负载平衡，也可以在一个工作者失败时加快恢复速度：它所完成的许多地图任务可以分散到所有其他工作者机器上。

在我们的实现中，对*M*和*R*的大小是有实际限制的，因为主程序必须做出$O(M+R)$的调度决定，并在内存中保留$O(M * R)$的状态，如上所述。(然而，内存使用的恒定因素很小：状态的$O(M * R)$部分包括每个地图任务/还原任务对的大约一个字节的数据）。

此外，*R*经常受到用户的限制，因为每个reduce任务的输出最终都在一个单独的输出文件中。在实践中，我们倾向于选择*M*，使每个任务的输入数据大致在16MB到64MB之间（这样上述的定位优化才是最有效的），并使*R*成为我们预期使用的工人机器数量的一个小倍数。我们经常以*M*=200,000和*R*=5,000的形式进行MapReduce计算，使用2,000台工作机。

3.6 备份任务

延长MapReduce操作总时间的常见原因之一是"散兵游勇"：一个机器需要花费异常长的时间来完成计算中最后几个映射或还原任务之一。滞留者的出现有很多原因。例如，一台有坏磁盘的机器可能会遇到频繁的可纠正的错误，使其读取性能从30MB/s降到1MB/s。集群调度系统可能在机器上安排了其他任务，由于对CPU、内存、本地磁盘或网络带宽的竞争，导致它执行MapReduce代码的速度变慢。我们最近遇到的一个问题是机器初始化代码中的一个错误，导致进程缓存被禁用：受影响的机器上的计算速度降低了100倍以上。

我们有一个通用的机制来缓解落伍者的问题。当一个MapReduce操作接近完成时，主程序会安排其余*正在进行的*任务的备份执行。只要主任务或备份任务执行完毕，该任务就被标记为完成。我们对这一机制进行了调整，使其通常会使操作所使用的计算资源增加不超过百分之几。我们发现，这大大减少了完成大型MapReduce操作的时间。举个例子，第5.3节中描述的排序程序，当备份任务机制被禁用时，完成的时间要长44%。

4 完善

尽管简单地编写*Map*和*Reduce*函数所提供的基本功能足以满足大多数需求，但我们发现有一些扩展很有用。本节将介绍这些扩展。

4.1 分割功能

MapReduce的用户指定他们想要的还原任务/输出文件的数量（*R*）。在这些任务中，数据被分割，使用的是一个分割函数。

的中间密钥。提供了一个使用散列法的默认分区函数（例如，"*hash*(*key*) mod *R*"）。这往往会导致相当均衡的分区。然而，在某些情况下，通过密钥的一些其他函数来划分数据是很有用的。例如，有时输出键是URL，而我们希望一个主机的所有条目最终都在同一个输出文件中。为了支持这样的情况，MapReduce库的用户可以提供一个特殊的分区函数。例如，使用"*hash*(*Hostname*(*urlkey*)) mod *R*"作为分区函数，可以使同一主机的所有URL最终出现在同一输出文件中。

4.2 订购保证

我们保证在一个给定的分区内，互为因果的键/值对是按键的递增来处理的。这种排序保证使得每个分区很容易产生一个排序的输出文件，这在输出文件格式需要支持有效的随机访问键的查找，或者输出的用户发现数据排序很方便的时候是很有用的。

4.3 组合器功能

在某些情况下，每个地图任务产生的中间键有很大的重复性，而用户指定的*Reduce*函数是换元的和同元的。这方面的一个很好的例子是第2.1节中的单词计数的例子。由于单词频率倾向于遵循Zipf分布，每个地图任务将产生成百上千条<the, 1>形式的记录。所有这些计数将通过网络被发送到一个单独的reduce任务，然后由*Reduce*函数相加，产生一个数字。我们允许用户指定一个可选的*Combiner*函数，在通过网络发送之前对这些数据进行部分合并。

*Combiner*函数在每台执行map任务的机器上执行。通常情况下，使用相同的代码来实现组合器和还原函数。减少函数和组合函数的唯一区别是MapReduce库如何处理函数的输出。还原函数的输出被写到最终的输出文件中。组合器函数的输出被写到一个中间文件中，该文件将被发送到还原任务中。

部分组合器大大加快了某些类别的MapReduce操作。附录A包含一个使用组合器的例子。

4.4 输入和输出类型

MapReduce库提供了对读取几种不同格式的输入数据的支持。例如，"文本"

模式的输入将每一行视为一个键/值对：键是文件中的偏移，值是该行的内容。另一种常见的支持格式是存储一个按键排序的键/值对的序列。每个输入类型的实现都知道如何将自己分割成平均的范围，以便作为单独的地图任务进行处理（例如，文本模式的范围分割确保范围分割只在行的边界出现）。用户可以通过提供一个简单的*阅读器*接口的实现来增加对新的输入类型的支持，尽管大多数用户只是使用少量预定义的输入类型中的一种。

一个*阅读器*不一定需要提供从文件中读取的数据。例如，很容易定义一个从数据库或从内存中映射的数据结构读取记录的*阅读器*。

以类似的方式，我们支持一组输出类型，用于产生不同格式的数据，而且用户代码很容易增加对新输出类型的支持。

4.5 副作用

在某些情况下，MapReduce的用户发现产生辅助文件作为他们的map和/或reduce操作的额外输出是很方便的。我们依靠应用程序的编写者来使这种副作用成为原子性的和空闲的。通常情况下，应用程序会写入一个临时文件，一旦该文件完全生成，就会以原子方式重命名该文件。

我们不提供对一个任务产生的多个输出文件的原子性两阶段commits的支持。因此，产生多个具有跨文件一致性要求的输出文件的任务应该是确定的。这一限制在实践中从未成为一个问题。

4.6 跳过不良记录

有时，用户代码中会出现一些bug，导致*Map*或*Reduce*函数在某些记录上确定性地崩溃。这样的bug会阻碍MapReduce操作的完成。通常的做法是修复该错误，但有时这并不可行；也许该错误是在第三方库中，而该库的源代码是不可用的。另外，有时忽略一些记录也是可以接受的，例如在对一个大的数据集进行统计分析时。我们提供了一种可选的执行模式，在这种模式下，MapReduce库会检测哪些记录会导致确定性崩溃，并跳过这些记录，以便取得进展。

每个工作进程都会安装一个信号处理程序，用于捕捉分段违规和总线错误。在调用用户的*Map*或*Reduce*操作之前，MapReduce库将参数的序列号存储在一个全局变量中。如果用户代码产生了一个信号。信号处理程序会向MapReduce主机发送一个包含序列号的"最后喘息"UDP数据包。当主站在某一特定记录上看到一个以上的故障时，它表示当它发出相关的Map或Reduce任务的下一次重新执行时，该记录应该被跳过。

4.7 本地执行

调试*Map*或*Reduce*函数中的问题可能很棘手，因为实际计算是在一个分布式系统中进行的，通常是在几千台机器上进行，工作分配的决定是由主站动态做出的。为了方便调试、分析和小规模测试，我们开发了一种MapReduce库的替代实施方案，在本地机器上按顺序执行MapReduce操作的所有工作。我们为用户提供了控制措施，以便将计算限制在特定的地图任务上。用户用一个特殊的标志来调用他们的程序，然后可以很容易地使用他们认为有用的任何调试或测试工具（如`gdb`）。

4.8 状态信息

主程序运行一个内部的HTTP服务器，并输出一组状态页供人使用。状态页显示了计算的进展，例如有多少任务已经完成，有多少任务正在进行，输入的字节数，中间数据的字节数，输出的字节数，处理率，等等。这些页面还包括每个任务生成的标准错误和标准输出文件的链接。用户可以使用这些数据来预先判断计算需要多长时间，以及是否应该在计算中添加更多的资源。这些页面也可以用来计算出什么时候计算比预期的慢得多。

此外，顶层的状态页面显示了哪些工作者失败了，以及他们失败时正在处理哪些映射和还原任务。这些信息在试图诊断用户代码中的错误时非常有用。

4.9 计数器

MapReduce库提供了一个计数器，用来计算各种事件的发生次数。例如，用户代码可能想计算处理的总字数或索引的德语文档数量等。

为了使用这一设施，用户代码创建一个命名的计数器对象，然后在*Map*和/或*Reduce*函数中适当地增加该计数器。比如说。

```
计数器*大写。
uppercase = GetCounter("uppercase")。

map(String name, String
  contents): 对于内容中的每个单词w。
    如果(IsCapitalized(w)):
      uppercase->Increment()。
    EmitIntermediate(w, "1")。
```

来自个别工人机器的计数器值
定期传播到主控中心（捎带Ping响应）。主站从成功的map和reduce任务中汇总计数器值，并在MapReduce操作完成后将其返回给用户代码。当前的计数器值也会在主站的状态页面上显示出来，这样人类就可以看到实时计算的进展。当聚集计数器的值时，主站消除了同一地图或还原任务的重复执行的影响，以避免重复计算。(重复执行可能来自于我们对备份任务的使用，以及因故障而重新执行的任务)。

有些计数器值是由MapReduce库自动维护的，比如处理的输入键/值对的数量和产生的输出键/值对的数量。

用户发现计数器对于检查MapReduce操作的行为非常有用。例如，在某些MapReduce操作中，用户代码可能希望确保产生的输出对的数量与输入对的数量完全相等，或者确保所处理的德国文件的比例不超过所处理的文件总数的某个可容忍的比例。

5 业绩

在本节中，我们测量了MapReduce在一个大型机器集群上运行的两个计算的性能。其中一个计算在大约一兆字节的数据中搜索，寻找一个特定的路径。另一个计算则对大约一兆字节的数据进行排序。

这两个程序代表了MapReduce用户编写的大量真实程序的子集--
一类程序将数据从一个代表区洗到另一个代表区，另一类程序从一个大数据集中提取少量有趣的数据。

5.1 集群配置

所有的程序都在一个由大约1800台机器组成的集群上执行。每台机器有两个2GHz的英特尔至强处理器，启用了超线程，4GB内存，两个160GB的IDE磁盘，和一个千兆以太网链接。这些机器被安排在一个两层的树形交换网络中，在根部有大约100-200Gbps的总带宽。所有的机器都在同一个主机设施中，因此任何一对机器之间的往返时间都小于一毫秒。

在4GB的内存中，大约1-1.5GB被集群上运行的其他任务所保留。这些程序是在一个周末的下午执行的，当时CPU、磁盘和网络大多处于空闲状态。

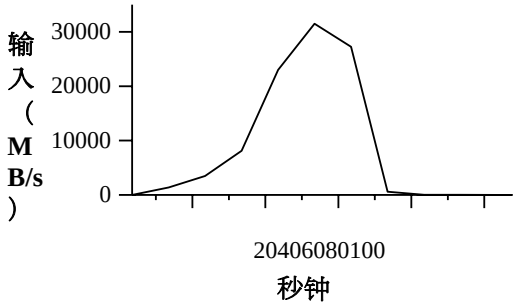


图2：数据传输率随时间变化

5.2 筛选

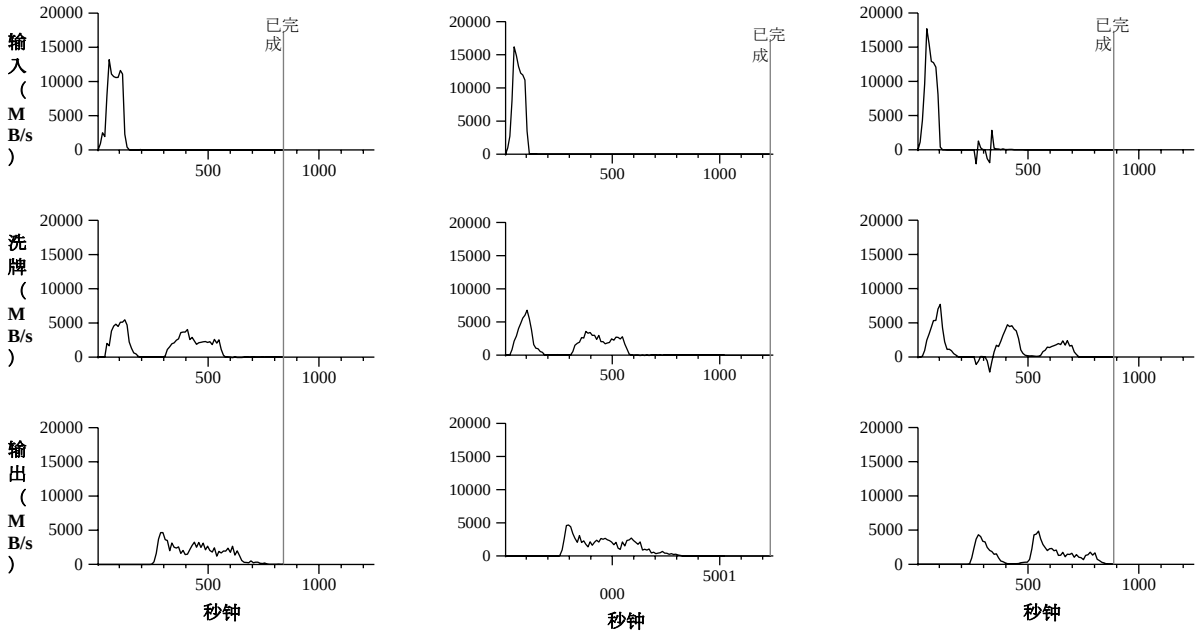
*grep*程序扫描了10^{10}条100字节的记录，寻找一个相对罕见的三字符模式（该模式出现在92,337条记录中）。输入被分割成大约64MB的片段（M=15000），轮胎的输出被放在一个文件中（R=1）。

图2显示了随时间推移的计算进度。Y轴显示输入数据的扫描速度。随着越来越多的机器被分配到这个MapReduce计算中，速度逐渐加快，当分配到1764个工人时，速度达到峰值，超过30GB/s。随着地图任务的完成，速率开始下降，并在计算的80秒左右达到零。整个计算从开始到结束大约需要150秒。这包括大约1分钟的启动开销。这些开销是由于将程序传播到所有工人机器上，以及与GFS互动以打开1000个输入文件集并获得位置优化所需信息的延迟。

5.3 分类

该排序程序对10^{10}条100字节的记录进行排序（大约1兆字节的数据）。这个程序是以TeraSort基准[10]为模型的。

该排序程序由不到50行的用户代码组成。一个三行的*Map*函数从一个文本行中提取一个10字节的排序键，并将该键和



(a) 正常执行　　(b) 没有备份任务　　(c) 200个任务被杀

图3：分类程序的不同执行的数据传输率随时间变化

原始文本行作为中间的键/值对。我们使用一个内置的*Identity*函数作为*Reduce*操作符。这个函数将未改变的中间键/值对作为输出键/值对。最终的排序输出被写入一组双向复制的GFS文件中（即2TB作为程序的输出被写入）。

和以前一样，输入数据被分割成64MB的碎片（$M = 15000$）。我们将排序后的输出划分为4000个文件（$R = 4000$）。分区功能使用钥匙的初始字节将其隔离到*R*块中的一个。

我们对这个基准的分区功能内置了对键的分布的了解。在一般的排序程序中，我们会添加一个预处理的MapReduce操作，收集一个键的样本，并使用抽样的键的分布来计算最终排序的分割点。

图3（a）显示了排序程序的正常执行进度。左上图显示了读取输入的速度。速率在大约13GB/s时达到峰值，并很快消失，因为所有的地图任务在200秒之前就已经完成。请注意，输入率比*grep*要低。这是因为排序地图任务花了大约一半的时间和I/O带宽将中间输出写到本地磁盘上。grep的相应中间输出的大小可以忽略不计。

左边中间的图显示了数据通过网络从地图任务发送至重构任务的速度。当第一个地图任务完成后，这种洗牌就开始了。图中的第一个驼峰是为第一批大约1700个还原任务（整个MapReduce被分配了大约1700台机器，每台机器每次最多执行一个还原任务）。在计算进行到大约300秒时，这些第一批还原任务中的一些完成了，我们开始为剩余的还原任务洗数据。所有的洗牌工作在计算的600秒左右完成。左下图显示了还原任务将排序后的数据写入最终输出文件的速度。在第一个洗牌期结束和写入期开始之间有一个延迟，因为机器正忙于对中间数据进行分类。写入工作以大约2-4GB/s的速度持续了一段时间。所有的写入工作在计算过程中大约850秒后完成。包括启动开销，整个计算需要891秒。这与目前TeraSort基准的最佳报告结果1057秒相似[18]。有几件事需要注意：输入率高于洗牌率和输出率，因为我们进行了定位优化--大多数数据是从本地磁盘读取的，绕过了我们相对带宽有限的网络。洗牌率高于输出率是因为输出阶段写了两份排序后的数据（出于可靠性和可用性的考虑，我们做了两份输出的副本）。我们写两个副本是因为这是我们的底层文件系统所提供的可靠性和可用性机制。如果底层文件系统使用擦除编码[14]而不是"复制"，那么写入数据的网络带宽需求就会减少。复制。

5.4 备份任务的效果

在图3（b）中，我们展示了在禁用备份任务的情况下执行排序程序。其执行流程与图3（a）所示相似，只是有一个很长的尾巴，几乎没有任何写入活动发生。960秒后，除了5个还原任务外，其他的都完成了。然而，这最后几个落伍者直到300秒后才完成。整个计算过程需要1283秒，耗时增加了44%。

5.5 机器故障

在图3（c）中，我们展示了排序程序的执行情况，在计算的几分钟内，我们故意杀死了1746个工人进程中的200个。底层集群调度器立即在这些机器上重新启动了新的工作进程（因为只有进程被杀死，机器仍然正常运行）。

工人的死亡显示为负输入率，因为一些先前完成的地图工作消失了（因为相应的地图工人被杀），需要重新做。这种地图工作的重新执行发生得相对较快。整个计算在933秒内完成，包括启动开销（只是比正常执行时间增加了5%）。

6 经验

我们在2003年2月编写了MapReduce库的第一个版本，并在2003年8月对其进行了重大改进，包括位置优化、工作机上任务执行的动态负载平衡等。从那时起，我们就对MapReduce库在我们所处理的各种问题中的广泛适用性感到惊喜。它已被用于谷歌内部的各种领域，包括。

- 大规模机器学习问题。
- 谷歌新闻和Froogle产品的聚类问题。
- 提取用于制作流行查询报告的数据（例如，Google Zeitgeist）。
- 为新的实验和产品提取网页的属性（例如，从大量的网页语料库中提取地理位置，用于本地化搜索），以及
- 大规模图计算。

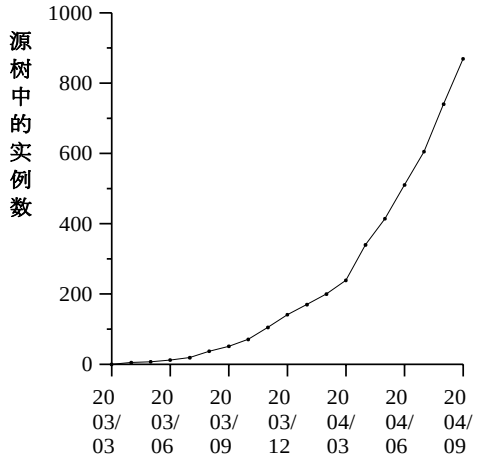


图4：MapReduce实例随时间变化的情况

工作数量	29,423
平均工作完成时间	634秒
机器使用天数	79,186天
读取输入数据	3,288 TB
产生的中间数据	758TB
写出的输出数据	193 TB
每项工作的平均工人机器	157
每项工作的平均工人死亡人数	1.2
每项工作的平均地图任务	3,351
平均减少每项工作的任务	55
独特的*地图*实现方式	395
独特的*还原*实现方式	269
独特的*地图/还原*组合	426

表1：2004年8月运行的MapReduce作业

图4显示了随着时间的推移，在我们的主要源代码管理系统中检查到的独立MapReduce程序数量的显著增长，从2003年初的0到2004年9月底的近900个独立实例。MapReduce之所以如此成功，是因为它可以在半小时内编写一个简单的程序并在一千台机器上高效运行，大大加快了开发和原型设计的周期。此外，它还允许没有分布式和/或并行系统经验的程序员轻松利用大量的资源。

在每个作业结束时，MapReduce库会记录该作业所使用的计算资源的统计数据。在表1中，我们展示了2004年8月在Google运行的MapReduce作业子集的一些统计数据。

6.1 大规模的索引编制

迄今为止，我们对MapReduce最重要的使用之一是完全重写了生产索引----。

索引系统产生用于谷歌网络搜索服务的数据结构。该索引系统将我们的爬行系统检索到的大量文件作为输入，这些文件以一组GFS文件的形式存储。这些文件的原始内容是超过20兆字节的数据。索引过程以五到十次MapReduce操作的顺序运行。使用MapReduce（而不是之前的索引系统版本中的临时分布式传递）提供了几个好处。

- 索引代码更简单、更小、更容易理解，因为处理容错、分布和并行化的代码都隐藏在MapReduce库中。例如，使用MapReduce表达时，一个阶段的计算规模从大约3800行的C++代码下降到大约700行。
- MapReduce库的性能足够好，我们可以把概念上不相关的计算分开，而不是把它们混在一起，以避免对数据的额外传递。这使得我们很容易改变索引过程。例如，在我们旧的索引系统中花了几个月的时间，在新的系统中只用了几天就实现了一个变化。
- 索引过程变得更容易操作，因为大多数由机器故障、慢速机器和网络故障引起的问题都由MapReduce库自动处理，不需要操作员干预。此外，通过在索引集群中增加新的机器，很容易提高索引过程的性能。

7 相关工作

许多系统提供了限制性的编程模型，并利用这些限制来自动并行化计算。例如，一个关联函数可以在*N*个处理器上用对数*N*的时间计算N个元素数组的所有前缀，使用并行前缀计算[6, 9, 13]。MapReduce可以被认为是基于我们在大型现实世界中的编译经验对其中一些模型的简化和提炼。更重要的是，我们提供了一个可扩展到数千个处理器的容错实现。相比之下，大多数并行处理系统只在较小的规模上实现，并将处理机器故障的细节留给程序员。

批量同步编程[17]和一些MPI基元[11]提供了更高层次的抽象。这些系统使程序员更容易编写并行程序。这些系统和MapReduce之间的一个关键区别是，MapReduce利用了一个有限的编程模型来自动并行化用户程序，并提供透明的容错功能。

我们的位置优化从主动磁盘[12,15]等技术中获得灵感，在这些技术中，编译被推到靠近本地磁盘的处理元件中，以减少跨I/O子系统或网络的数据量。我们在商品处理器上运行，少量的磁盘直接连接到这些处理器上，而不是直接在磁盘控制器处理器上运行，但一般的方法是类似的。

我们的备份任务机制类似于Charlotte System[3]中采用的急切调度机制。简单的急切调度的缺点之一是，如果一个给定的任务导致重复失败，整个计算就无法完成。我们用我们的跳过坏记录的机制来解决这个问题的某些方面。

MapReduce的实现依赖于一个内部集群管理系统，该系统负责在大量的共享机器上分配和运行用户任务。虽然不是本文的重点，但该集群管理系统在精神上与其他系统如Condor[16]相似。

作为MapReduce库的一部分，其排序设施在操作上与NOW-Sort[1]相似。源机器（地图工作者）对要排序的数据进行分区，并将其发送给*R*还原工作者之一。每个还原工作者在本地对其数据进行排序（如果可能的话，在内存中）。当然，NOW-Sort没有用户可定义的Map和Reduce功能，而这些功能使我们的库得到广泛的应用。

River[2]提供了一个编程模型，其中程序通过分布式队列发送数据来相互通信。像MapReduce一样，River系统试图提供良好的平均性能，即使在异构硬件或系统扰动所带来的非均匀性的情况下。River通过对磁盘和网络传输的精心调度来实现这一目标，以达到平衡的完成时间。Mapreduce有一个不同的方法。通过限制编程模型，MapReduce框架能够将问题分割成大量细粒度的任务。这些任务被动态地安排在可用的工作器上，以便更快的工作器能够处理更多的任务。
限制性编程模型还允许我们在工作结束前安排多余的任务执行，这在存在非均匀性（如缓慢或卡住的工人）的情况下大大减少了完成时间。

BAD-FS[5]的编程模型与MapReduce非常不同，与MapReduce不同的是，它的目标是

在广域网上执行工作。然而，有两个基本的相似之处。(1)
两个系统都使用冗余执行来恢复由故障引起的数据丢失。(2)
两者都使用局部感知调度，以减少在有争议的网络链接上发送的数据量。

TACC[7]是一个旨在简化高可用网络服务的构建的系统。与MapReduce一样，它依靠重新执行作为实现容错的机制。

8 结论

MapReduce编程模型已经在谷歌成功地应用于许多不同的目的。我们把这种成功归功于几个原因。首先，该模型很容易使用，即使是没有并行和分布式系统经验的程序员也可以使用，因为它隐藏了并行化、容错、位置优化和负载平衡等细节。
其次，大量的问题都可以用MapReduce的组合来表达。例如，MapReduce被用于谷歌生产网络搜索服务的数据生成、排序、数据挖掘、机器学习和许多其他系统。第三，我们开发了一种MapReduce的实现，可以扩展到由成千上万台机器组成的大型机器集群。该实施方案有效地利用了这些机器的再资源，因此适用于在谷歌遇到的许多大型计算问题。

我们从这项工作中学到了几件事。首先，对编程模型的限制使其很容易实现计算的平行化和分布化，并使这种计算具有容错性。第二，网络带宽是一种稀缺资源。因此，我们系统中的一些优化是以减少网络上发送的数据量为目标的：本地性优化使我们可以从本地磁盘读取数据，而将中间数据的单一副本写入本地磁盘则可以节省网络带宽。第三，可以用冗余执行来减少慢速机器的影响，并处理机器故障和数据丢失。

鸣谢

Josh
Levenberg根据他使用MapReduce的经验和其他人提出的改进建议，对用户级MapReduce
API进行了修订和扩展，增加了许多新的功能。MapReduce从谷歌文件系统读取输入，并将输出写入谷歌文件系统[8]。我们要感谢Mohit Aron, Howard Gobioff, Markus Gutschke, David Kramer, Shun-Tak Leung, 和Josh Redstone在开发GFS方面所做的工作。我们还要感谢Percy Liang和Olcan Sercinoglu在开发MapReduce使用的集群管理系统方面的工作。Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, 和 Debby Wallach对本论文的早期草稿提供了有益的意见。OSDI的匿名审稿人和我们的监督人Eric
Brewer提供了许多有用的建议，指出本文可以改进的地方。最后，我们感谢谷歌工程部门的所有MapReduce用户提供的有益反馈、建议和错误报告。

参考文献

[1] Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson.工作站网络上的高性能分拣。在*1997年ACM SIGMOD数据管理国际会议论文集中*，亚利桑那州图森市，1997年5月。

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick.Cluster I/O with River:让快速的情况变得普遍。在*第六届并行和分布式系统中的输入/输出研讨会（IOPADS '99）上*，第10-22页，亚特兰大，乔治亚州，1999年5月。

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff.夏洛特。网络上的元计算.在*第九届平行和分布式计算系统国际会议的论文*中，1996年。

[4] Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. 一个星球的网络搜索。谷歌集群架构。*IEEE Micro*, 23(2):22-28, April 2003.

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny.分批感知的分布式文件系统中的显式控制。In *Pro- ceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.

[6] Guy E. Blelloch.Scans as primitive parallel operations.*IEEE Transactions on Computers*, C-38(11), November 1989.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier.基于集群的可扩展的网络服务。在*第16届ACM操作系统原理研讨会上*，第78-91页，法国圣马洛，1997。

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung.谷歌文件系统.在*第19届操作系统原理研讨会上*，第29-43页，纽约乔治湖，2003。

[9] S.Gorlatch.扫描和其他列表同构的系统性高效并行化。In L. Bouge, P. Fraigni- aud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96.Parallel Processing*, Lecture Notes in Computer Science 1124，第401-408页。Springer-Verlag, 1996.

[10] 吉姆 - 格雷。排序基准首页。http://research.microsoft.com/barc/SortBenchmark/。

[11] William Gropp, Ewing Lusk, and Anthony Skjellum.*Using MPI:用消息传递接口进行可移植的并行编程*。麻省理工学院出版社，剑桥，MA，1999。

[12] L.Huston, R. Sukthankar, R. Wickremesinghe, M. Satya- narayanan, G. R. Ganger, E. Riedel, and A. Ailamaki.Di-amond:主动搜索中早期丢弃的存储架构。在*2004年USENIX文件和存储技术FAST会议论文集中*，2004年4月。

[13] Richard E. Ladner and Michael J. Fischer.并行前缀计算。*ACM杂志*，27（4）：831-838，1980。

[14] Michael O. Rabin.用于安全、负载平衡和容错的信息的有效分散。*ACM杂志*，36（2）：335-348，1989。

[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle.用于大规模数据处理的活动磁盘。*IEEE计算机*，第68-74页，2001年6月。

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny.分布式计算的实践。The Condor experi-ence.*并发与计算。实践与体验》*，2004年。

[17] L.G. Valiant.Abridging model for parallel computation. *Communications of the ACM*, 33(8):103-111, 1997.

[18] Jim Wyllie.Spsort：如何快速对一兆字节进行分类。http://alme1.almaden.ibm.com/cs/spsort.pdf。

A 词频

本节包含一个程序，用于计算在命令行上指定的一组输入文件中每个独特单词的出现次数。

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

// 用户的地图功能
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; ) {
      // 跳过前面的空白处 while ((i < n) &&
      isspace(text[i]))
      i++;

// 查找字尾 int
start = i;
while ((i < n) && !isspace(text[i]))
  i++;
```

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

// 用户的还原函数 class Adder :
public Reducer {
   虚无缥缈的 Reduce(ReduceInput* input) {
          // 遍历所有条目，其中包括
    // 相同的键并添加值 int64 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规范。

  // 将输入文件的列表存入 "spec "中 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")。
  }

  // 指定输出文件。
       ///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")。

  // 可选：在地图内做部分求和
  //任务是为了节省网络带宽 out-
  >set_combiner_class("Adder")。

  // 调谐参数：最多使用2000个
  //机器和每个任务100MB的内存
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100)。

  // 现在运行它
  MapReduceResult result
  。
  if (!MapReduce(spec, &result)) abort()。

  // 完成：'结果'结构包含信息
  //关于计数器、所需时间、数量的
  // 使用的机器，等等。

  返回0。
}
```