**目标：**

实现一个分布式MapReduce，由master和worker组成，一个master进程和一个或多个worker进程并行执行。

真实系统中，worker将运行在一堆不同的机器上，但此实验中，在一台机器上运行即可。workers通过RPC与master对话。

每个worker向master请求一个任务，从一个或多个文件中读取任务的输入，执行任务，并将任务的输出写入一个或多个文件。如果一个worker没有在合理的时间内完成它的任务(如10秒)，master应该注意到，并将相同的任务交给不同的worker。

**一些规则：**

1、map阶段应该把中间键给nReduce个redeuce任务分到buckets中，nReduce是main/mrmaster.go 传给MakeMaster()的参数。

2、worker实现应该把第X个reduce任务的输出放到mr-out-X文件中。

3、mr-out-X文件内容为“%v %v”格式，例如统计单词出现次数，则每行形如“test 1”。

4、可暂时的修改mr/worker.go，mr/master.go，and mr/rpc.go以供测试，最终测试需要原始版本。

5、worker应该将中间映射输出放入当前目录的文件中，从而worker可以读取它们作为reduce任务。

6、main/mrmaster.go 期待 mr/master.go 实现一个Done() 方法，它mapreduce工作全部完成时返回true，然后mrmaster.go 会退出。

7、当任务完全完成时，worker应该退出。一个简单方法是使用call()的返回值：如果worker未联系到master，它就假设master已经退出，因为任务已经完成，因此worker也可以终止。也可以让master将“请退出”伪任务分配给worker。

**一些提示：**

1、一种方法是修改mr/worker.go的worker()，让其发送一个RPC到master请求一个任务。然后修改master以使用尚未启用的map任务的文件名来响应。然后修改worker以读取该文件并调用应用程序的map函数，如mrsequence .go中所示。

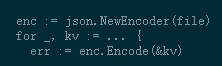
2、应用程序的map和reduce函数在运行时通过go plugin从.so文件中加载。

3、如果改变了mr/目录中的任何内容，可能需要重新构建使用的所有MapReduce插件，使用类似go build -buildmode=plugin ./mrapps/wc.go这样的方法

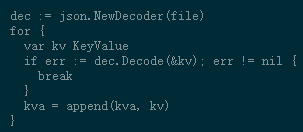
4、实验依赖于worker共享一个文件系统。当worker程序都运行在同一台机器上，这很简单，但如果在不同的机器上，则需要一个像GFS这样的全局文件系统。

5、中间文件的合理命名约定是mr-X-Y，其中X是map任务号，Y是reduce任务号。

6、worker的map任务代码将需要一种方法来在文件中存储中间kv对，从而可在reduce任务期间正确地读取。可使用Go的encoding/json包。将kv对写入JSON文件:



读取形如：



7、worker的map部分可以使用ihash(key)函数(在worker.go中)为给定的键选择reduce任务。

8、可以从mrsequential.go中借鉴一些代码用于读取map输入文件，对map和Reduce之间的kv对进行排序，以及存储Reduce的输出到文件。

9、master作为RPC服务器，会产生并发；不要忘记锁定共享数据。

10、使用Go的race 检测器，Go build -race和Go run -race。test-mr.sh有一个说明如何为测试启用race 检测器的注释。

11、worker有时需要等待，例如直到最后一个map完成后reduce才能开始。一种方法是worder定期向master请求工作，每个请求之间通过time.Sleep()睡觉。另一种方法是，master中的相关RPC处理程序有一个等待的循环，使用time.Sleep()或sync.Cond。Go在它自己的线程中为每个RPC运行处理程序，因此一个处理程序正在等待不会阻止master处理其他RPC。

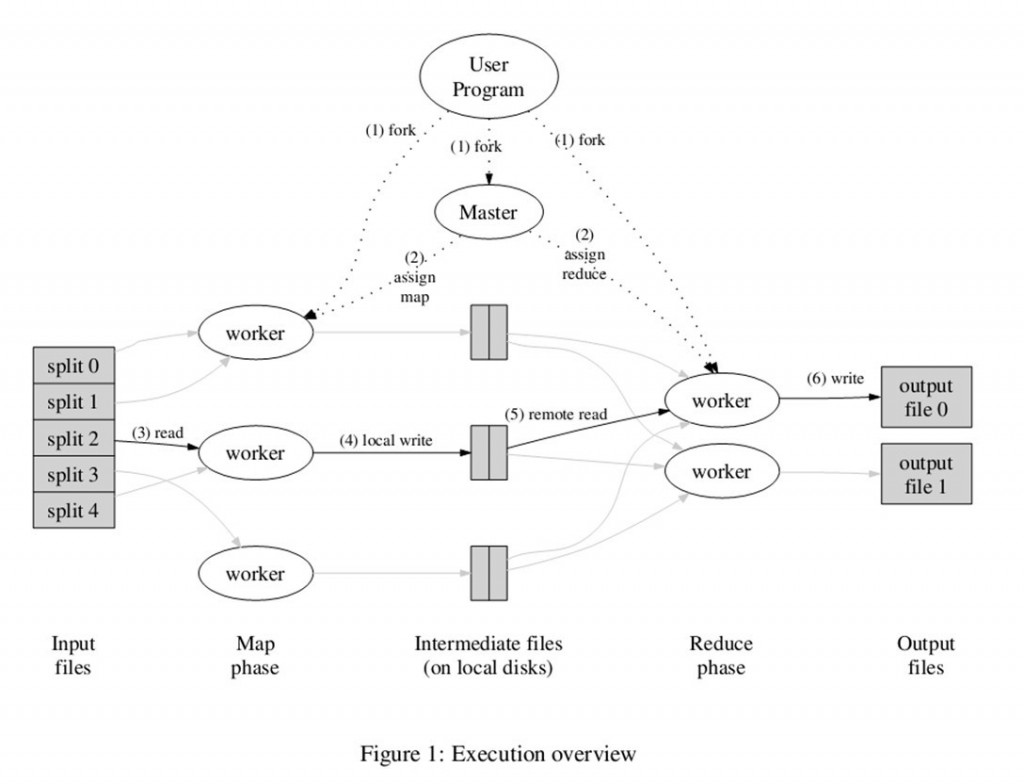
12、master无法有效区分崩溃的worker、活着但由于某种原因停止工作的worker和正在执行但太慢而无法使用的worker。最好的方式是让master等待一段时间，然后放弃并将任务重新发送给另一个worker。此实验中让master等待10秒钟，此后强制认为该worker已经死亡。

13、要测试崩溃恢复，可以使用mrapps/crash.go 应用插件。它在Map和Reduce函数中随机退出。

14、为了确保在出现崩溃时不会出现写到一半的文件，MapReduce论文提到了使用临时文件并在其完全写入后自动重命名的技巧。你可以使用ioutil.TempFile创建临时文件和os.Rename为其原子重命名。

15、test-mr.sh在子目录mr-tmp中运行所有进程，因此如果出现错误而想查看中间文件或输出文件，请查看那里。

**MapReduce工作流程：**



上图是论文里给出的流程图。一切都是从最上方的user program开始的，user program**链接了MapReduce库**，**实现了最基本的Map函数和Reduce函数**。

　　1.MapReduce库先把输入文件划分为M份（用户定义），如图分成了split0~4；

　　2.user program的副本中有一个为master，其余为worker，master负责调度，为worker分配作业（Map/Reduce），worker的数量也可由用户指定。

　　3.分配了Map作业的worker，开始读取对应分片的输入数据。从中抽取出键值对，每一个键值对都作为参数传递给map函数，map函数产生的中间键值对被缓存在内存中。

4.缓存的中间键值对会被定期写入本地磁盘，而且被分为R个区，R的大小是由用户定义的，将来每个区会对应一个Reduce作业；这些中间键值对的位置会被通报给master，master负责将信息转发给Reduce worker。

**R个区是逻辑的，写入本地磁盘是物理的，就是说一个Map作业会映射到多个不同的区，但这个Map作业映射的多个区在物理上是存储在一起的，而后面某个Reduce作业需要处理所有的Map作业映射的相同编号的区。**

5.master通知分配了Reduce作业的worker它负责的分区位置（多个地方），当Reduce worker把所有它负责的中间键值对都读过来后，先进行排序，使得相同键的键值对聚集在一起。

6.reduce worker遍历排序后的中间键值对，对于每个唯一的键，都将键与关联的值传递给reduce函数，reduce函数产生的输出会添加到**这个分区**？的输出文件中。

7.当所有的Map和Reduce作业都完成了，master唤醒正版的user program，MapReduce函数调用返回user program的代码。

所有执行完毕后，MapReduce输出放在了R个分区的输出文件中（分别对应一个Reduce作业）。用户通常并不需要合并这R个文件，而是将其作为输入交给另一个MapReduce程序处理。整个过程中，**输入数据来自底层分布式文件系统，中间数据放在本地文件系统，最终输出数据写入底层分布式文件系统的**。

注意**Map/Reduce作业和map/reduce函数的区别：Map作业处理一个输入数据的分片，可能需要调用多次map函数来处理每个输入键值对；Reduce作业处理一个分区的中间键值对，期间要对每个不同的键调用一次reduce函数，Reduce作业最终也对应一个输出文件。**

