你的工作（中等/困难）

你的任务是实现一个分布式MapReduce程序，包括两个程序：协调者和工作者。只有一个协调者进程和一个或多个并行执行的工作者进程。在真实系统中，工作者将运行在多台不同的机器上，但在这个实验中，你将在同一台机器上运行它们。工作者将通过RPC与协调者通信。每个工作者进程将循环执行以下操作：向协调者请求任务，从一个或多个文件读取任务输入，执行任务，将任务的输出写入一个或多个文件，然后再次向协调者请求新任务。如果协调者发现某个工作者在合理的时间内（在这个实验中，设置为10秒）没有完成任务，则应将相同的任务分配给另一个工作者。

我们已经为你提供了一些启动代码。main程序中的协调者和工作者代码位于 `main/mrcoordinator.go` 和 `main/mrworker.go` 中；不要修改这些文件。你应该将你的实现放在 `mr/coordinator.go`、`mr/worker.go` 和 `mr/rpc.go` 中。

以下是如何在word-count MapReduce应用程序上运行你的代码。首先，确保word-count插件已经重新构建：

```
bash
复制代码
$ go build --buildmode=plugin ../mrapps/wc.go
```

在 `main` 目录下，运行协调者。

```
bash复制代码$ rm mr-out*
$ go run mrcoordinator.go pg-*.txt
```

`mrcoordinator.go` 的 `pg-*.txt` 参数是输入文件；每个文件对应一个"分片"，并作为一个Map任务的输入。

在一个或多个其他窗口中，运行一些工作者程序：

```
bash
复制代码
$ go run mrworker.go wc.so
```

当工作者和协调者完成工作后，查看 `mr-out-*` 中的输出文件。当你完成实验时，输出文件的排序结果应与顺序执行的输出一致，如下所示：

```
bash复制代码$ cat mr-out-* | sort | more
A 509
ABOUT 2
ACT 8
...
```

我们为你提供了 `main/test-mr.sh` 中的测试脚本。测试将检查 `wc` 和 `indexer` MapReduce应用程序在给定 `pg-xxx.txt` 文件作为输入时，是否能生成正确的输出。测试还会检查你的实现是否能并行运行Map和Reduce任务，并且在工作者运行任务时崩溃后是否能够恢复。

规则

Map 阶段应将中间键划分为 `nReduce` 个 Reduce 任务，其中 `nReduce` 是 Reduce 任务的数量——这个参数由 `main/mrcoordinator.go` 传递给 `MakeCoordinator()`。每个 mapper 应创建 `nReduce` 个中间文件供 Reduce 任务使用。

Worker 实现应将第 X 个 Reduce 任务的输出放入文件 `mr-out-X` 中。

一个 `mr-out-X` 文件应包含每个 Reduce 函数输出的一行。该行应使用 Go 的 `"%v %v"` 格式生成，带有键和值。请查看 `main/mrsequential.go` 中标注 "this is the correct format" 的行注释。如果你的实现偏离此格式太多，测试脚本将失败。

你可以修改 `mr/worker.go`、`mr/coordinator.go` 和 `mr/rpc.go`。你可以临时修改其他文件以进行测试，但请确保你的代码与原始版本配合工作；我们将使用原始版本进行测试。

Worker 应将中间 Map 输出放在当前目录下的文件中，你的 worker 随后可以将它们作为输入读取到 Reduce 任务中。

`main/mrcoordinator.go` 期望 `mr/coordinator.go` 实现一个 `Done()` 方法，当 MapReduce 任务完全完成时返回 `true`；此时，`mrcoordinator.go` 将退出。

当任务完全完成时，worker 进程应退出。实现这一点的一个简单方法是使用 `call()` 的返回值：如果 worker 无法联系到 coordinator，它可以假设 coordinator 已退出，因为任务已经完成，所以 worker 也可以终止。根据你的设计，你可能还需要添加一个“请退出”的伪任务，coordinator 可以将其发送给 workers。

提示

- **指导页面** 有一些开发和调试的建议。
- 一种开始的方法是修改 `mr/worker.go` 中的 `Worker()`，使其发送一个 RPC 给协调器以请求任务。然后修改协调器以响应一个尚未启动的 map 任务的文件名。接着修改 worker 读取该文件并调用应用程序 Map 函数，如 `mrsequential.go` 中所示。
- 应用程序的 Map 和 Reduce 函数在运行时使用 Go 插件包从以 `.so` 结尾的文件中加载。
- 如果你更改了 `mr/` 目录中的任何内容，可能需要重新构建你使用的任何 MapReduce 插件，类似于 `go build -buildmode=plugin ../mrapps/wc.go`。
- 该实验依赖于 workers 共享文件系统。当所有 workers 在同一台机器上运行时，这很简单，但如果 workers 在不同的机器上运行，则需要类似于 GFS 的全局文件系统。
- 对于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。
- worker 的 Map 任务代码需要一种方法来将中间键/值对存储在文件中，并且在 Reduce 任务期间能够正确读取这些数据。一个可能的办法是使用 Go 的 `encoding/json` 包。如下所示以 JSON 格式写入键/值对到打开的文件中：

```
go复制代码enc := json.NewEncoder(file)
for _, kv := ... {
    err := enc.Encode(&kv)
}
```

并且从这样的文件中读取：

```
go复制代码dec := json.NewDecoder(file)
for {
    var kv KeyValue
    if err := dec.Decode(&kv); err != nil {
        break
    }
    kva = append(kva, kv)
}
```

- worker 的 Map 部分可以使用 `ihash(key)` 函数（在 `worker.go` 中）来为给定键选择 Reduce 任务。
- 你可以从 `mrsequential.go` 中偷点代码，用于读取 Map 输入文件、在 Map 和 Reduce 之间排序中间键/值对，并将 Reduce 输出存储在文件中。
- 作为 RPC 服务器的协调器将是并发的；不要忘记锁定共享数据。
- 使用 Go 的竞争检测器，使用 `go run -race`。`test-mr.sh` 的开头有一个注释告诉你如何使用 `-race` 运行它。我们在评估你的实验时**不会**使用竞争检测器。然而，如果你的代码有竞争条件，即使没有竞争检测器也很可能失败。
- 有时 workers 需要等待，例如 Reduce 不能在最后一个 Map 完成之前开始。一个可能的办法是 workers 定期向协调器请求工作，每次请求之间使用 `time.Sleep()` 休眠。另一个可能的办法是让协调器中的相关 RPC 处理程序有一个等待的循环，使用 `time.Sleep()` 或 `sync.Cond`。Go 会为每个 RPC 处理程序在它自己的线程中生成一个 handler，因此如果其中一个处理程序在等待，它不会阻止协调器处理其他 RPC。
- 协调器无法可靠地区分崩溃的 workers、存活但因某种原因停止的 workers，以及正在执行但过于缓慢而无效的 workers。对于此实验，让协调器等待十秒钟；然后协调器应假设 worker 已死（当然，也可能没有）。
- 如果你选择实现备份任务（3.6 节），请注意我们测试你的代码时不安排多余的任务，除非 workers 在不崩溃的情况下执行任务。备份任务应仅在相对较长的一段时间（例如10秒）之后安排。
- 为测试崩溃恢复，你可以使用 `mrapps/crash.go` 应用程序插件。它会随机在 Map 和 Reduce 函数中退出。
- 为确保没有人在崩溃的情况下观察到部分写入的文件，MapReduce 论文提到使用临时文件并在完全写入后原子重命名它的技巧。你可以使用 `ioutil.TempFile`（或如果你使用 Go 1.17 或更高版本可以使用 `os.CreateTemp`）创建临时文件，并使用 `os.Rename` 原子重命名它。
- `test-mr.sh` 在子目录 `mr-tmp` 中运行其所有进程，因此如果有任何问题并且你想查看中间或输出文件，请查看那里。可以随时临时修改 `test-mr.sh` 以在失败的测试后退出，这样脚本就不会继续测试（并覆盖输出文件）。
- `test-mr-many.sh` 多次运行 `test-mr.sh`，这可能是为了发现低概率的错误。它接受一个参数，即运行测试的次数。你不应该并行运行多个 `test-mr.sh` 实例，因为协调器将重复使用相同的套接字，导致冲突。
- Go RPC 只发送字段名以大写字母开头的结构体字段。子结构体的字段名也必须大写。
- 调用 RPC `call()` 函数时，reply 结构体应包含所有默认值。RPC 调用应如下所示：

```
go复制代码reply := SomeType{}
call(..., &reply)
```

在调用之前不要设置 reply 的任何字段。如果你传递了包含非默认字段的 reply 结构体，RPC 系统可能会静默地返回不正确的值。