gDocs测试报告

SSFD小组

- kafka test
- load balance test
- master log test
- master namespace test

后端测试

正确性

缓存池单元测试

Memsheet

1. 测试了*NewMemSheetFromStringSlice*和*NewMemSheet*经过几轮Set后各个单元格内容是否一致。
2. 测试了写入大量随机字符后，*GetSize*返回的大小是否和字符串大小基本相等。
3. 测试了1000并发同时写同一个单元格表格字符数统计是否正确，内容修改是否正确。

SheetCache

1. 测试写入三倍于缓存容量时是否按照LRU顺序驱逐缓存
2. 测试Get一个MemSheet后，改变大小为原来4倍，重新Put后是否会驱逐出3个缓存
3. 测试被Get但没有被Put的MemSheet是否没有被驱逐
4. 测试大到缓存无法驱逐来获得足够内存的MemSheet是否会被直接驱逐
5. 测试原先驱逐的MemSheet剩下的空间是否能够继续容纳新的缓存

结果

通过率：100%

覆盖率：93.5%

有部分分支是为了健壮性添加，理论上不会到达，所以无法全部覆盖。

LRU单元测试

1. 测试Add方法 - 将新加入或者已有的Key设为最近使用
2. 测试Delete方法 - 删除Key
3. 测试AddToLeastRecent - 将新加入或者已有的Key设为最不常使用，主要是用来恢复DoEvict出的Key
4. 测试DoEvict - 获得最不常使用的Key，并删除
5. 测试Len - 是否和LRU中剩余的Key数量相等

结果

通过率：100%

覆盖率：100%

DFS封装测试

1. 在原本不存在20个目录下每个目录都新建20个文件，测试是否报错和create返回的fd是否正常
2. 使用Scan测试原本不存在中间目录是否被正确创建，且名字正确
3. 使用Scan测试每个目录下文件是否被正确创建，且名字正确
4. 对前10个文件夹里的所有文件调用Write随机写入并用ReadAll读取，比较是否相同
5. 对后10个文件夹里的所有文件调用Append随机写入并用ReadAll读取，比较是否相同
6. Delete测试根目录，测试所有子目录和文件是否都被删除
7. Close所有fd，测试是否报错

结果

通过率：100%

覆盖率：77.3%

没有覆盖到的基本都是对网络错误的处理，只是简单地返回错误，不需要覆盖。

系统测试

User

1. 测试了Login
2. 测试了Register
3. 测试了GetUser
4. 测试了ModifyUser
5. 测试了ModifyUserAuth

Sheet

1. 测试了NewSheet
2. 测试了GetSheet
3. 测试了ModifySheet

结果

注：该测试是针对单机版的测试，后改为分布式部署后只能进行端到端测试

通过率：100%

覆盖率：96.7%

端到端测试

注：以下测试由docker-compose分布式部署后，使用go编写脚本进行端到端测试，故没有覆盖率指标。

单元格顺序一致性

1. 测试单个文件实时编辑：启动多个用户，使用websocket对同一个文件进行无限制的随机写，部分用户会从中间加入，最后比较各个用户通过websocket信息还原的表格内容是否一致
2. 测试多个文件实时编辑：启动多个用户，一边新建房间一边在里面进行限制速率的随机写，用户会从中间加入，最后比较各个用户通过websocket信息还原的表格内容是否一致

内存Evict和文件系统加载

1. 启动多文件实时编辑，调大单个表格的行列，并调大测试新建的文件数量，确保超过服务器的缓存大小，测试各个用户还原的表格应该是否一致
2. 把服务器的最大缓存调整为0，这样每次修改都会触发存盘和加载，然后测试单文件实时编辑，测试各个用户还原的表格应该是否一致

新增和删除节点

1. 删除节点：在Chrome中用前端新建一个表格并进入，随便填写几个单元格，保存版本，在NetWork中查看被分配的后端，手动在Docker关闭该后端，等待ZooKeeper心跳时间(7s)后刷新，看是否被分配到新的后端且不同后端重定向到的位置是否一致。然后看新加载的表格内容是否一致，版本回滚是否一致。
2. 新增节点：新建一个表格，在Chrome中找到对应的后端A，在docker将其关闭，重新打开会重定向到新的后端B，此时使用两个用户登陆B并进入实时编辑状态。重新启动被关闭的后端A，此时该资源在任何后端都会被重定向到原先的后端A，在两个用户未断开连接的情况下，ZooKeeper的分布式锁仍然在后端B，此时新的用户进入实时编辑，会被重定向到后端A，且A会阻塞因为无法获取分布式锁。之后退出B上实时编辑两个用户，A会迅速得到锁并恢复实时编辑的服务。
3. FSCheck：新建表格后多次(>10)修改单元格并创建回滚点，然后手动删除所有checkpoint文件，只保留log文件，将原先负责该资源的服务器在Docker中关闭，重新加载时会触发FSCheck，如果正确的话所有丢失的checkpoint文件都会被恢复，且表格内容和原来一致。

结果

除了缓存为0的无限制单文件测试和多文件测试外，其他的测试都通过，但这两个测试没通过的原因不是因为实现的错误。

在缓存为0的无限制单文件测试中，因为文件写非常慢，而请求不受限制的话会有很多任务堆积在协程池中等待处理，这样导致在测试过程中只处理了几百个请求，而在协程池中还有上万个请求等待处理，这样用户接收到的内容很有可能是服务器还没有返回，而不是返回错误。在调整了协程池大小，减缓了了请求速度，加大了测试等待时间之后，测试是可以通过的。

多文件测试也是同样的原因，服务器负载过高，测试等待结束后还没有处理完，所以需要减慢消息速率，延长等待时间，这样也能通过测试。

性能

单文件实时编辑

在缓存充足的情况下，我们测试了十个用户对单文件无限制随机写5分钟的性能，结果如下：

> concurrency_test.go:37:
> Duration: 300s
> Users: 10
> Total Send: 74250003 op
> Total Recv: 72389638 op
> Send Rate: 24750 op/(s * user)
> Recv Rate: 24129 op/(s * user)
> --- PASS: TestBenchmarkSingleFile (320.26s)

其中Recv Rate是最重要的参数，它是在后端处理完消息之后返回的，代表着后端处理消息的速率；而因为后端会丢弃超过协程池容量的消息，Send Rate和网络关系比较大，而和后端处理能力关系不大。所以我们重点关注Recv Rate。

可以看到消息处理和转发的吞吐非常高，这是因为WebSocket的无锁生产者消费者架构的良好设计带来的，同时也是缓存的作用。我将后端的最大缓存设置为0，这样每次修改都需要读写文件系统，在这样的条件下测试结果如下：

> concurrency_test.go:37:
> Duration: 300s
> Users: 10
> Total Send: 165382496 op
> Total Recv: 40944 op
> Send Rate: 55127 op/(s * user)
> Recv Rate: 13 op/(s * user)
> --- PASS: TestBenchmarkSingleFile (320.27s)

可以看到Recv吞吐非常低，说明后端每秒钟只能处理十几个请求，只有缓存的两千分之一，缓存的重要性得以体现。

后端个数对性能的影响

后端越多，文件的缓存越大，对实时编辑计算资源的也可以分摊更多。如果我们无限制地模拟用户创建新文件并进行实时编辑，后端越多，性能曲线应该越好。但是由于设备的限制，我们很难进行效果能够

明显到体现出计算和缓存分摊的大规模测试，所以只是进行了方案设想，并未进行实际测试。

我们设想的方案如下：

1. 采用二八定律，先创建大量文件，然后将80%的访问概率集中在20%的文件上，记录Cache Miss的概率，测试LRU缓存驱逐策略的影响
2. 在不同后端数量下，分别测试 10^1~10^6 个文件实时编辑的各个文件的吞吐
3. 在不同后端数量下，对测试 10^1~10^6 个文件使用GetSheet，使其加载到缓存中，测试Cache Miss的概率

Zookeeper测试

心跳 & 服务发现

1. 为四个服务器创建回调函数，使用四个Map记录收到的对应节点事件的数量
2. 四个服务器依次注册，每次注册时调用*GetOriginMates*得到的伙伴数量应该和之前注册的服务器数量相同
3. 检查每个服务器收到的*onConn*事件数量是否正确
4. 每个服务器主动断开连接，检查每个服务器收到的*onDisConn*事件数量是否正确

消息队列

1. 创建10个Log Room
2. 为每个Room创建20/5个初始/追加的Channel，测试初始Channel和追加事件是否正确
3. 为每个channel创建5/20个初始/追加的Log，测试初始Log和追加事件是否正确
4. 使用WaitGroup等待预计的事件结束，并设置超时时间，超时则测试失败

选举

1. 生成10个候选者
2. 每个候选者成为Leader后立刻退位
3. 测试是否同时只有一个Leader，且退位后状态是否改变
4. 使用WaitGroup等待预计的事件结束，并设置超时时间，超时则测试失败

互斥锁

1. 生成10个锁的竞争者
2. 每个竞争者竞争到锁后睡眠一段时间再放锁
3. 测试释放锁的一定是得到锁的
4. 拿到锁时给计数加1，放锁时减1

5. 测试最后计数是否为0

测试结果

通过率：100%

覆盖率：72.6%

未覆盖的大多数都是为了健壮性添加的错误处理，只是简单地返回错误，不需要覆盖。

DFS测试

在编写DFS的过程中，我们同时编写了详尽的测试脚本，从不同层面测试了DFS各个功能的正确性。以下是每个测试脚本的简介（仅列出 unit test，实际上还有做集成测试与端到端测试）：

chunkServer test

对Chunk的并发读、写、追加操作进行正确性测试,测试全部通过。

client API test

以使用者的角度，对client提供的接口进行测试，包括了read,write,list,open,close等，同时分为single client和Multi client不同的做不同并发度测试，测试全部通过。

heartbeat election test

测试zookeeper支撑下的多Master DFS是否正确实现，测试了心跳，选举等机制，并检查各个回调函数是否正确调用，测试全部通过。

kafka test

对kafka的生产消费正确性测试。

load balance test

测试负载均衡的两个方面是否正确实现（allocate,reallocate），集群的负载是否均衡，测试全部通过。

master log test

分为两个方面，一个是测试log的持久化与崩溃后从Log进行恢复的功能，另一个是测试多master环境下使用kafka进行元数据同步是否成功，测试全部通过。

master namespace test

专门测试master元数据中文件系统树是否正确实现，不同并发度下测试全部通过。