# 文件系统测试报告

## 测试结构

我们针对文件系统部分进行了单元测试以及不同程度的集成测试。其中，单元测试主要用于测试文件系统的内部逻辑；由于我们的文件系统使用Zookeeper和Kafka两种外部系统，相关的逻辑难以进行单元测试，因此我们通过集成测试来测试这一部分的逻辑以及可靠性。根据Go的测试规范，测试用例均包含在以`_test.go`结尾的文件中。包含单元测试的测试文件可以直接运行以获得测试结果，另有一部分文件包含集成测试用例，这些用例需要首先启动相应的测试环境（在同目录下的`docker-compose.yaml`文件提供了docker-compose测试环境部署），然后才能执行测试。

## 整体方法

在文件系统的不同部分，我们采用了不同的测试方法，如下表所示：

| 包名 | 功能 | 测试方法 |
| --- | --- | --- |
| `fsclient` | 文件系统客户端库 | 集成测试 |
| `master` | 实现Master节点进程 | 单元测试、集成测试 |
| `datanode` | 实现DataNode节点进程 | 单元测试、集成测试 |
| `protocol` | 定义gRPC协议 | 大部分为生成代码，无需单独测试 |
| `election` | 封装Zookeeper选举算法 | 集成测试 |
| `common_journal` | 封装基于Kafka的通用日志逻辑 | 集成测试 |

# 单元测试

## 测试设计

在单元测试部分，我们主要针对Master节点与DataNode节点这两种节点的内部逻辑进行测试。在Master部分，进行了由低到高三个层次的测试：

- `master/sheetfile`：该包实现了我们的文件系统提供的抽象SheetFile所需的内存数据结构。一个SheetFile对象在内存中组织了系统中的一个文件的元数据。我们在该包中对该对象的操作的和方法的逻辑进行了测试。
- `master/filemgr`：该包组织多个SheetFile对象，提供了顶级目录功能，实现了Master节点的文件系统逻辑。我们在该包中主要测试Master所提供的文件系统操作（`create`,`open`,`read`,`write`,`delete`等）的逻辑是否符合要求。
- `master/server`：该包实现了Master节点的RPC服务器功能。我们使用gRPC实现RPC功能。在该部分测试中，我们通过手动实例化gRPC请求对象并传入RPC服务器对象，并检查该对象方法返回的gRPC响应结构体，从而模拟服务器接收并处理gRPC请求的过程，以测试RPC方法的逻辑是否符合要求。

在DataNode部分，由于我们的设计中DataNode的逻辑相对简单，因此我们只对DataNode的RPC服务器层进行了单元测试，测试方法和对Master的RPC服务器层的测试思路类似。

## 测试结果

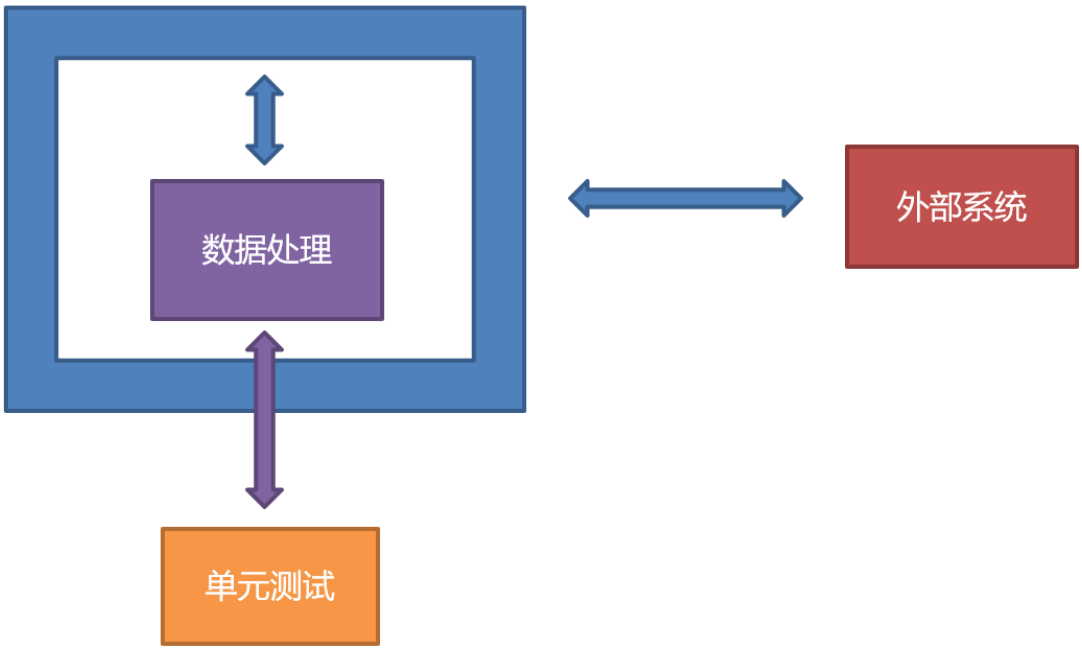
在单元测试部分，我们只针对Master和DataNode的实现进行了单元测试，覆盖率基于这两个包进行计算。

我们的实现通过全部单元测试测试用例，其中，源文件覆盖率为84.6%，语句覆盖率为49.1%。

由于Go的测试覆盖率以package为单位指定，并不支持从测试覆盖率结果中去除一部分指定源文件的功能。而`master`和`datanode`包中都分别有大量的逻辑用于处理选举和日志相关的工作，这部分我们无法直接进行单元测试，因此在单元测试中基本没有覆盖，导致语句覆盖率很低。

当然，从另一个角度来看，这也说明我们对程序的抽象工作存在一些不足之处，使得与外部系统（Zookeeper、Kafka等）交互的逻辑和系统内部处理由外部系统返回的数据的逻辑耦合较为紧密。在理想的程序结构中，后者应当被独立抽象封装，而前者通常实现为一些对后者的包装（wrappers），将外界系统返回的数据传入后者或将后者返回的数据传出，如下所示。由于数据处理逻辑属于系统内部逻辑，这使得我们可以通过模拟外界传入的数据来针对这一部分逻辑进行独立的测试，而非因为二者的耦合导致单元测试难以进行，这是我们在今后的项目中需要注意的。

数据交互




# 集成测试

## 测试设计

在集成测试部分，我们针对单元测试中难以测试的逻辑进行测试，主要包括引入多备份之后的选举和日志相关逻辑，以及客户端库与实际运行的系统的交互、错误处理逻辑。主要包括以下层次的测试：

- `common_journal`包中将日志处理封装为`Writer`和`Receiver`两种抽象。该包中通过集成测试测试了两种抽象的交互逻辑。
- `election/example`包中通过一个独立示例程序测试了使用该包进行选举的逻辑。
- `master/node`包中包含完整运行一个Master节点所需的各种准备工作。在该包中通过完整实例化多个Master节点，测试了它们组成一个集群后的选举与日志处理逻辑。
- `datanode/node`包中包含完整运行一个DataNode节点所需的各种准备工作。在该包中通过完整实例化多个DataNode节点，测试了它们组成一个集群后的选举与日志处理逻辑。
- `tests/integration`包中包含对系统整体的集成测试工作。我们在该包通过直接调用客户端库`fsclient`，对完整运行的文件系统发起RPC请求并检查返回结果，来测试系统的整体逻辑与错误一致性。

以下我们对部分内容较为复杂的测试做进一步说明。

## Master

- 测试环境：由3个节点组成的Zookeeper集群与1个Kafka节点，通过docker-compose启动。
- 测试流程：
  - 通过调用`master/node`包中的逻辑实例化3个Master节点。为了测试的便利性，我们使用单独的goroutine运行一个Master节点，从而避免阻塞执行测试逻辑的goroutine，而又避免启动独立Master进程后难以测试的问题（无法直接调用其方法，必须使用客户端库发起RPC请求，从而导致测试的耦合）。
  - 实例化后的3个节点首先进行选举工作，选举出1个Primary与2个Secondary节点。我们通过与Zookeeper服务器建立单独连接并反复轮询用于确认选举结果的Znode，来检查选举结果。
  - 模拟生成gRPC请求对象，并直接调用Primary节点的RPC方法，以模拟客户端实际与Primary节点交互的过程。由于要测试日志逻辑，在Primary节点中，我们主要调用产生元数据写入的方法，包括创建一系列文件并向其中写入数据，随后删除其中的一些文件。
  - 模拟生成gRPC请求对象，并直接调用Secondary节点的RPC方法。在我们的设计中，Secondary节点并不直接提供RPC服务，将它们运行在同一地址空间的goroutine中使得我们可以直接控制这些节点的RPC服务器对象，从而检查测试结果。在Secondary节点上，我们主要调用读取元数据的方法，包括打开文件与读取其中的单元格，以**检查Secondary节点是否正确应用了由Primary节点产生的日志。**
  - 等待一定时间，使得所有节点的定时Checkpoint逻辑完成执行。随后额外启动两个新的节点，其中一个节点不继承任何数据，从日志的开头依次应用日志；另一个节点继承先前的一个Secondary节点的数据库（即其中的Checkpoint数据）。随后，在两个新的节点上再次调用对Secondary节点的测试中使用的读取用例，以此**模拟节点崩溃重启后从Checkpoint中恢复数据的情况，检查相关逻辑是否正确。**

## DataNode

测试思路与Master基本相同，不再赘述。

## fsclient

- 测试环境：由3个节点组成的Zookeeper集群与1个Kafka节点，通过docker-compose启动。
- 测试脚本：在该部分测试中，我们除了测试客户端库是否能正常通过RPC与文件系统集群交互外，还要测试文件系统集群是否能实现崩溃一致性，以及客户端库是否能正常处理文件系统集群节点发生崩溃的问题。这就要求我们在测试中能够随时使得某个被测节点产生崩溃，而这是我们前述的通过goroutine运行测试节点的设计无法实现的。goroutine并不能被直接强制终止，必须花费大量额外逻辑编写异步的`Context`终止逻辑，而这些逻辑在系统实际运行中并没有额外的收益，因此并不可行。为了解决这一问题，我们使用Python编写了独立测试脚本（`tests/integration/drive_tests.py`）。它通过运行独立子进程的方式启动多个Master和DataNode节点，并通过系统调用强制终止子进程从而模拟节点的崩溃情况。实际的测试用例仍然使用Go编写，测试脚本只负责准备所需的测试环境，并调用`go test`执行测试用例。
- 测试流程：
  - 测试脚本通过docker-compose启动Zookeeper集群与Kafka。
  - 测试脚本启动文件系统集群，包含1个Master集群与3个DataNode集群，每个集群包含3个节点。每个节点运行为一个子进程。
  - 测试脚本与Zookeeper集群建立单独连接，确认并记录每个集群的Primary节点所对应的子进程。
  - 测试脚本启动子进程执行`go test`，从而执行集成测试用例。
  - 集成测试用例中，通过产生特殊的输出，指示测试脚本产生相应的崩溃场景，并通过读标准输入的方式暂时阻塞测试执行。

- 测试脚本读取到子进程的输出后，终止Master集群以及DataNode集群的Primary节点所对应的子进程，从而触发集群的重新选举。
- 测试脚本写入子进程的标准输入，从而恢复测试执行。

## 测试结果

我们的实现通过全部集成测试用例，证明我们的实现能较为顺利地与上层的应用后端进行集成。测试中，源文件覆盖率为87.9%，语句覆盖率为80%以上。语句覆盖率为粗略计算结果，这是因为go test的输出会将gRPC生成代码计入覆盖率中，后者对于覆盖率的负面影响较大，我们通过简单粗略计算去除了它们的影响。