Mini-Redis Presentation

By 张瑞(Moratoryvan), 秦嘉俊(HobbitQia), 陆晶宇(MYJOKERML)

Part 1: AOF 实现持久化

设计思路

- 考虑到只有SET和DEL需要被记录，所以设计一个enum:

```
pub enum Command {
    Set { key: String, value: String },
    Del { key: String },
}
```

- 每次收到一次指令就创建一个Command并格式化为字符串 SET {} {} 或 DEL {} 后追加到 "log.aof" 文件中。"log.aof" 的文件示例如下:

```
SET ZJU ZhengjiangUniversity
SET ZJU ZhejiangUniversity
DEL ZJU
DEL ZJU
```

- 每次启动redis时读取"log.aof"文件读取所有指令，重复执行这些指令恢复redis中的数据。

AOF 测试

脚本如下：

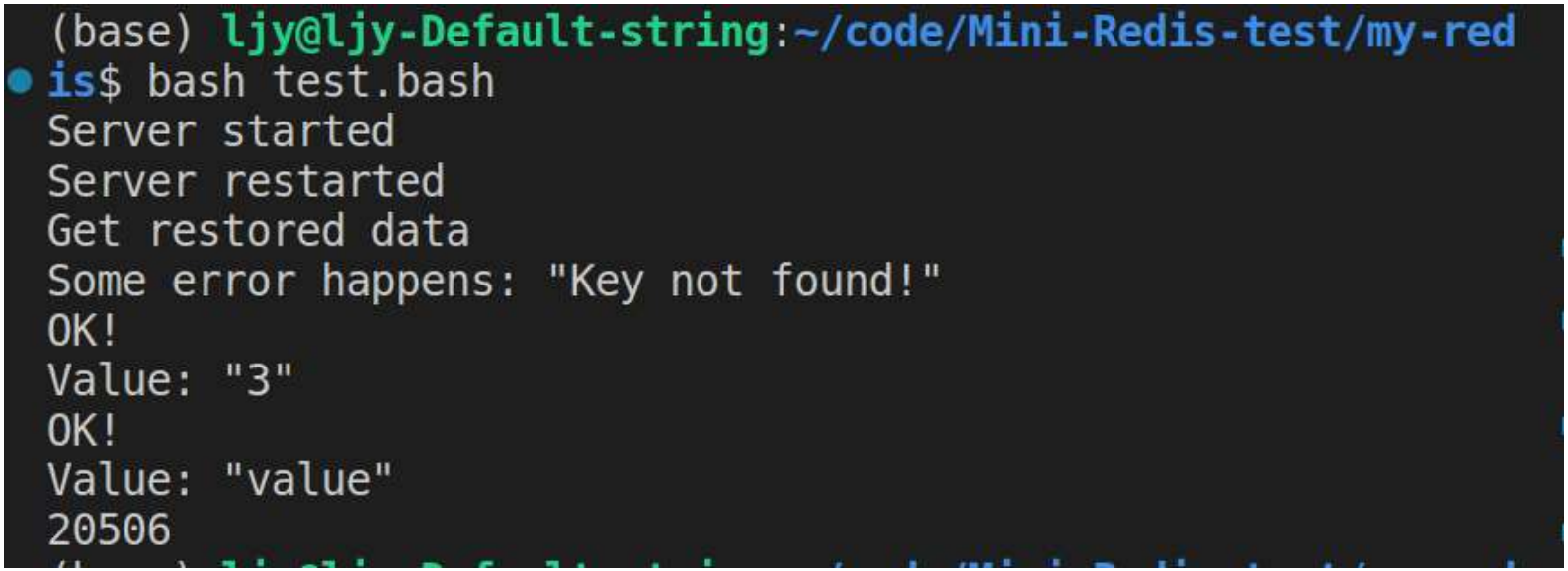
```
#!/bin/bash
chmod +x ./client && chmod +x ./serv

# 启动 server
./server &
echo "Server started"

# 获取 server 进程的 PID (进程 ID)
server_pid=$!

# 定义一个函数，用于重启 server
restart_server() {
 # 杀死当前 server 进程
 kill $server_pid
 # 等待一段时间确保 server 进程已经终止
 sleep 2
 # 启动新的 server
 ./server &
 echo "Server restarted"
 server_pid=$!
}
```

```
# 启动 client
echo "fisrt set data" > out
./client SET key value >> out # {"key": "value"}
./client SET 1 2 >> out # {"1": "2"}
./client SET 2 3 >> out # {"2": "3"}
./client GET key >> out
./client GET 1 >> out
./client GET 2 >> out
./client DEL 1 >> out #delete {"1": "2"}

# 当 client 执行完毕后，重启 server
restart_server

echo "Get restored data"
./client GET 1
./client GET 2
./client GET key

# 输出 server 进程 pid，用于手动杀死进程，防止进程一直运行占用
echo $server_pid
```

测试思路，往数据库中依次执行多次操作后再重启server获取数据。

测试结果

执行效果如下：



执行的最初的命令结果重定向了到当前文件夹的 "out" 文件中。

Part 2: Redis 主从架构

配置文件

```
pattern: [master-slave/cluster]

name: [name]
type: [master/slave/proxy]
host: [host]
port: [port]
master_host(opt): [master_host]
master_port(opt): [master_port]
```

- pattern 决定构建服务器时是以那种架构构建的，本项目有两个选项，
  - master-slave：以主从架构构建
  - cluster：以集群架构构建
- type 有三种类型：
  - master：表示主服务端
  - slave：表示从服务端
  - proxy：表示代理服务端
- 当且仅当 type = "slave" 的时候，master_host 和 master_port 的值才有效

设计思路

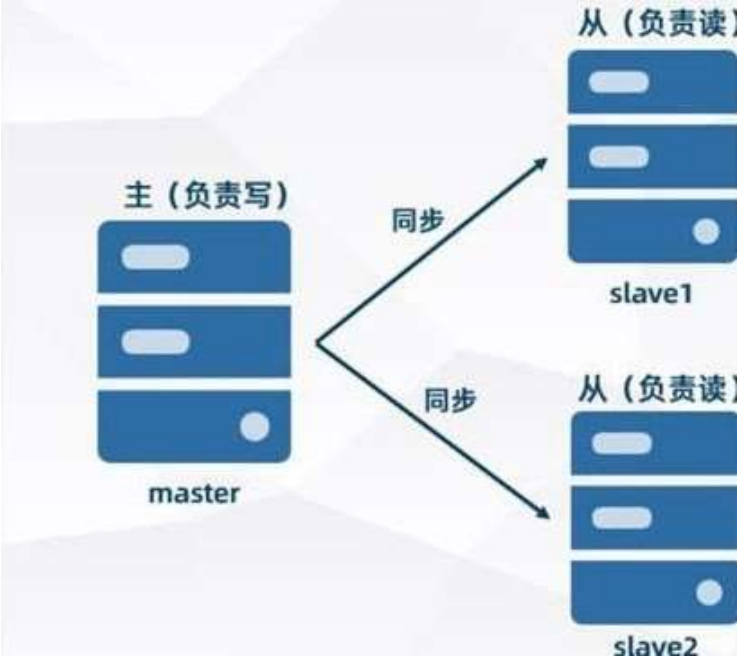
- 一个主服务端，一个从属服务端
  - 主服务端负责处理 SET，DEL，GET 服务请求
  - 从服务端负责处理 GET 服务请求，对于 SET，DEL 请求返回错误
- 服务器运行时，采用增量复制的方法同步数据，每当主服务端接收到写、删除操作请求时，在处理完请求后，将同样的请求信息发送给从服务端，从服务端执行对应的操作，从而达到数据同步的效果。
- 服务器重启时，采用重演历史的方法恢复数据

优点：

- 实现简单
- 数据稳定

缺点：

- 运行效率低



主从架构测试

启动脚本：

先读取配置文件生成启动服务端的bash脚本，再调用生成的脚本启动主从服务器。

```
#!/bin/bash

# 获取当前工作目录，并进入工作目录
workdir=$(cd $(dirname $0); pwd)

echo $workdir

if [ ! -d $workdir/log ]; then
    mkdir $workdir/log
fi

# 读取config文件并将启动主从服务器的命令重定向到m-s.bash中
cargo run --bin read_file > $workdir/m-s.bash

# 启动主从服务器
cd $workdir/../ && bash $workdir/m-s.bash

echo "Server start!"
```

主从架构测试(Cont.)

测试脚本：

测试思路：先往master写入数据 {"THU": "TsinghuaUniversity"} ，再测试三个slave是否能正确读取数据。

然后测试往slave1里写入数据和删除数据，均被拒绝，只能通过master进行写操作

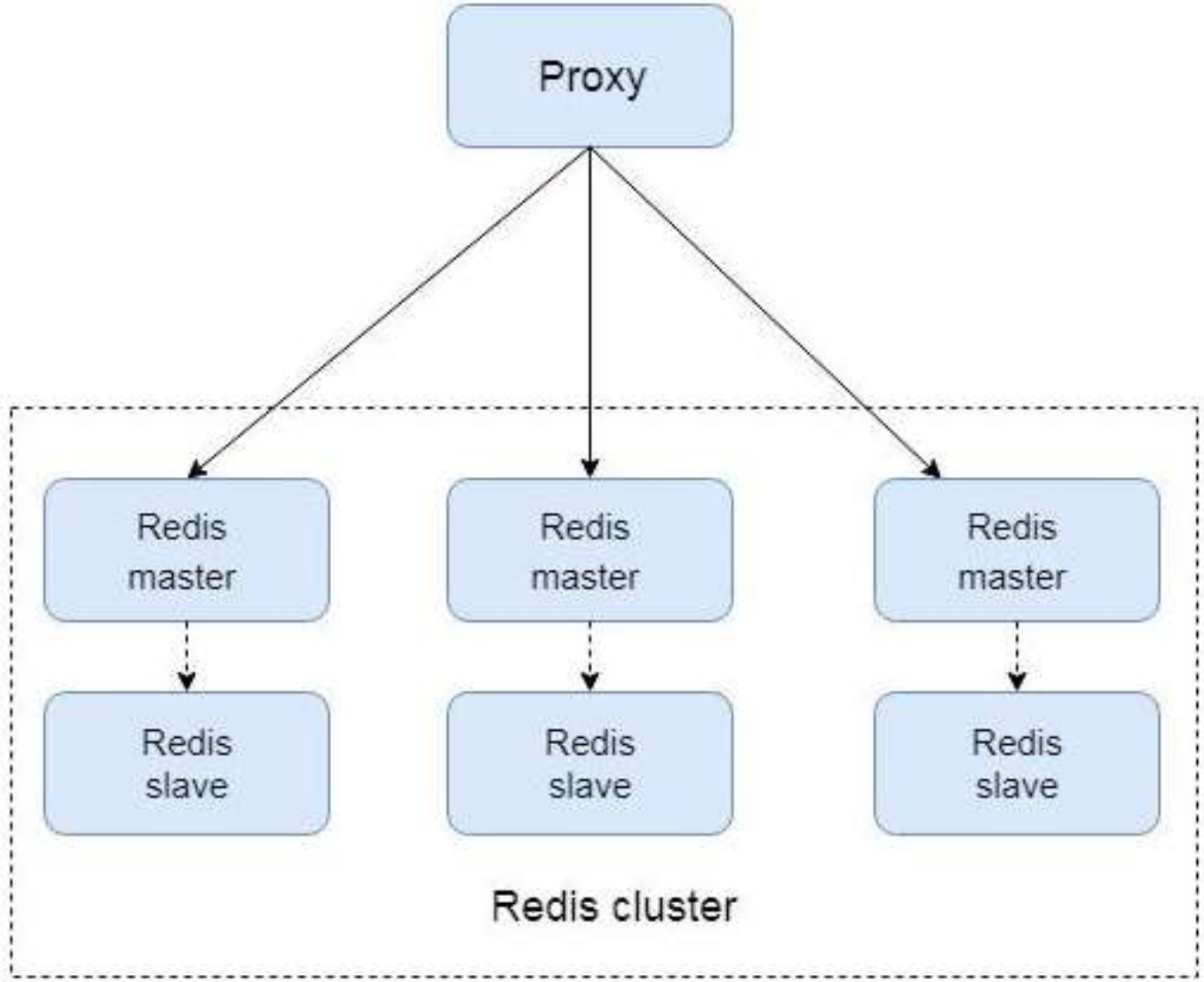
```
# 获取当前工作目录，并进入工作目录
workdir=$(cd $(dirname $0); pwd)
echo $workdir
echo "redis-master-1 set THU TsinghuaUniversity: " && cd $workdir/../target/debug && ./c
echo "redis-slave-1 get THU: " && cd $workdir/../target/debug && ./client redis-slave-1 g
echo "redis-slave-2 get THU: " && cd $workdir/../target/debug && ./client redis-slave-2 g
echo "redis-slave-3 get THU: " && cd $workdir/../target/debug && ./client redis-slave-3 g
echo "redis-slave-1 set ZJU ZhengjiangUniversity" && cd $workdir/../target/debug && ./cli
echo "redis-slave-1 get ZJU" && cd $workdir/../target/debug && ./client redis-slave-1 get
echo "redis-master-1 get ZJU" && cd $workdir/../target/debug && ./client redis-master-1 g
echo "redis-master-1 set ZJU ZhejiangUniversity" && cd $workdir/../target/debug && ./clie
echo "redis-slave-1 get ZJU" && cd $workdir/../target/debug && ./client redis-slave-1 get
echo "redis-slave-1 del ZJU" && cd $workdir/../target/debug && ./client redis-slave-1 del
echo "redis-master-1 del ZJU" && cd $workdir/../target/debug && ./client redis-master-1 d
echo "redis-master-1 get ZJU" && cd $workdir/../target/debug && ./client redis-master-1 g
```

测试结果

```
(base) ljy@ljy-Default-string:~/code/Mini-Redis-test/my-redis/src$ bash client_part2_test.bash
/home/ljy/code/Mini-Redis-test/my-redis/src
redis-master-1 set THU TsinghuaUniversity:
OK!
redis-slave-1 get THU:
OK!
Value: "TsinghuaUniversity"
redis-slave-2 get THU:
OK!
Value: "TsinghuaUniversity"
redis-slave-3 get THU:
OK!
Value: "TsinghuaUniversity"
redis-slave-1 set ZJU ZhengjiangUniversity
Some error happens: "You can not set values into slave server."
redis-slave-1 get ZJU
Some error happens: "Key not found!"
redis-slave-1 set ZJU ZhengjiangUniversity
Some error happens: "Key not found!"
redis-master-1 set ZJU ZhejiangUniversity
OK!
redis-slave-1 get ZJU
OK!
Value: "ZhejiangUniversity"
redis-slave-1 del ZJU
Some error happens: "You can not delete values from slave server."
redis-master-1 del ZJU
OK!
redis-master-1 get ZJU
Some error happens: "Key not found!"
```

Part 3: Redis Cluster

Redis Cluster 架构示意图



设计思路

- 本质上是实现多个主从架构一起对外服务，通过哈希算法实现负载均衡
  - 哈希算法：对 key 计算出一个 hash 值，然后用哈希值对 master 数量进行取模，模数为 16383。再由计算得到的 hash 算出 index ，通过 index 进行服务器的索引，从而可以将 key 负载均衡到每一个 Redis 节点上去

```
let hash = State::<ARC>::calculate(req.key.clone().unwrap().as_bytes()) % MOD;
let size = self.proxy_box.len() as u16;
let index = hash / (MOD / size);
```

- 最外层 proxy 服务端接收到 key 后，会根据上述的哈希算法找到对应的主从架构服务器，对其发送服务请求，从而执行对应操作。
- 每一个主从架构都有一个独立的备份日志

Redis Cluster 测试

启动脚本：

```
#!/bin/bash

# 获取当前工作目录，并进入工作目录
workdir=$(cd $(dirname $0); pwd)

echo $workdir

if [ ! -d $workdir/log ]; then
    mkdir $workdir/log
fi

# 读取config文件并将启动主从服务器的命令重定向到m-s.bash中
cargo run --bin read_file > $workdir/m-s.bash

# 启动主从服务器
cd $workdir/../ && bash $workdir/m-s.bash

echo "Server start!"
```

Redis Cluster 测试(Cont.)

测试脚本：

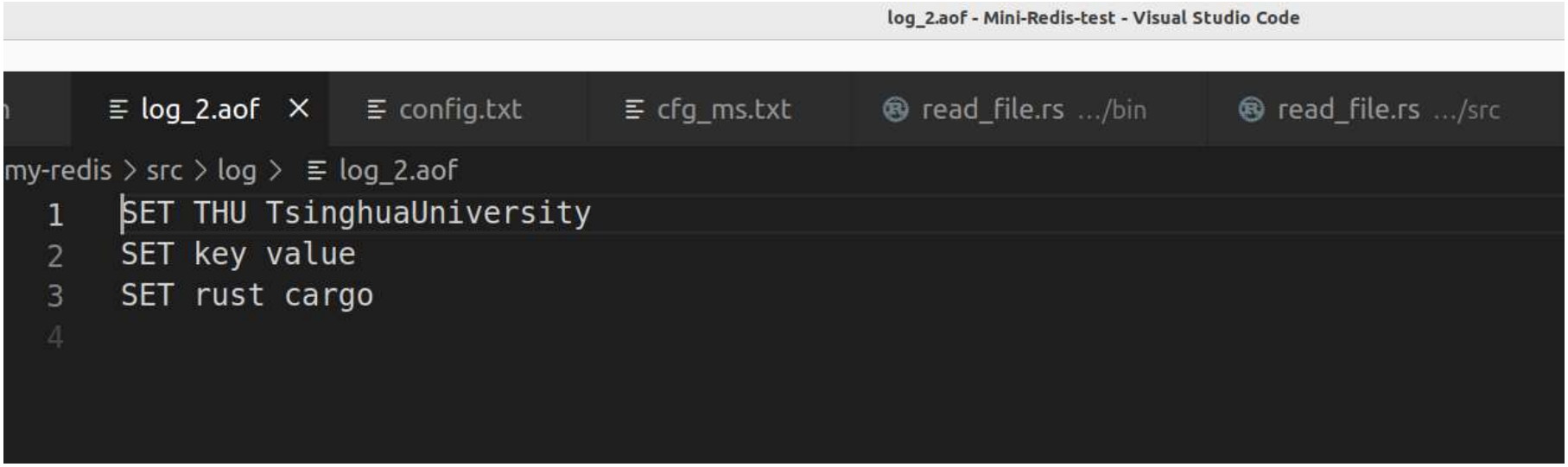
```
# 获取当前工作目录，并进入工作目录
workdir=$(cd $(dirname $0); pwd)

echo $workdir

cd $workdir/../target/debug && ./client set THU TsinghuaUniversity
cd $workdir/../target/debug && ./client get THU
cd $workdir/../target/debug && ./client set ZJU ZhengjiangUniversity
cd $workdir/../target/debug && ./client get ZJU
cd $workdir/../target/debug && ./client set ZJU ZhejiangUniversity
cd $workdir/../target/debug && ./client get ZJU
cd $workdir/../target/debug && ./client del ZJU
cd $workdir/../target/debug && ./client del ZJU
cd $workdir/../target/debug && ./client get ZJU
cd $workdir/../target/debug && ./client set key value
cd $workdir/../target/debug && ./client set rust cargo
```

测试结果

```
(base) ljy@ljy-Default-string:~/code/Mini-Redis-test/my-redis/src$ bash client_part3_test.bash
/home/ljy/code/Mini-Redis-test/my-redis/src
OK!
OK!
Value: "TsinghuaUniversity"
OK!
OK!
Value: "ZhengjiangUniversity"
OK!
OK!
Value: "ZhejiangUniversity"
OK!
Some error happens: "Key not found!"
Some error happens: "Key not found!"
OK!
OK!
(base) ljy@ljy-Default-string:~/code/Mini-Redis-test/my-redis/src$
```

测试结果(Cont.)


log_1.aof - Mini-Redis-test - Visual Studio Code
config.txt
cfg_ms.txt
read_file.rs .../bin
read_file.rs .../src
log_1.aof
my-redis > src > log > log_1.aof
1 SET ZJU ZhengjiangUniversity
2 SET ZJU ZhejiangUniversity
3 DEL ZJU
4 DEL ZJU
5


log_2.aof - Mini-Redis-test - Visual Studio Code
log_2.aof
config.txt
cfg_ms.txt
read_file.rs .../bin
read_file.rs .../src
my-redis > src > log > log_2.aof
1 SET THU TsinghuaUniversity
2 SET key value
3 SET rust cargo
4

Bonus: Graceful Exit

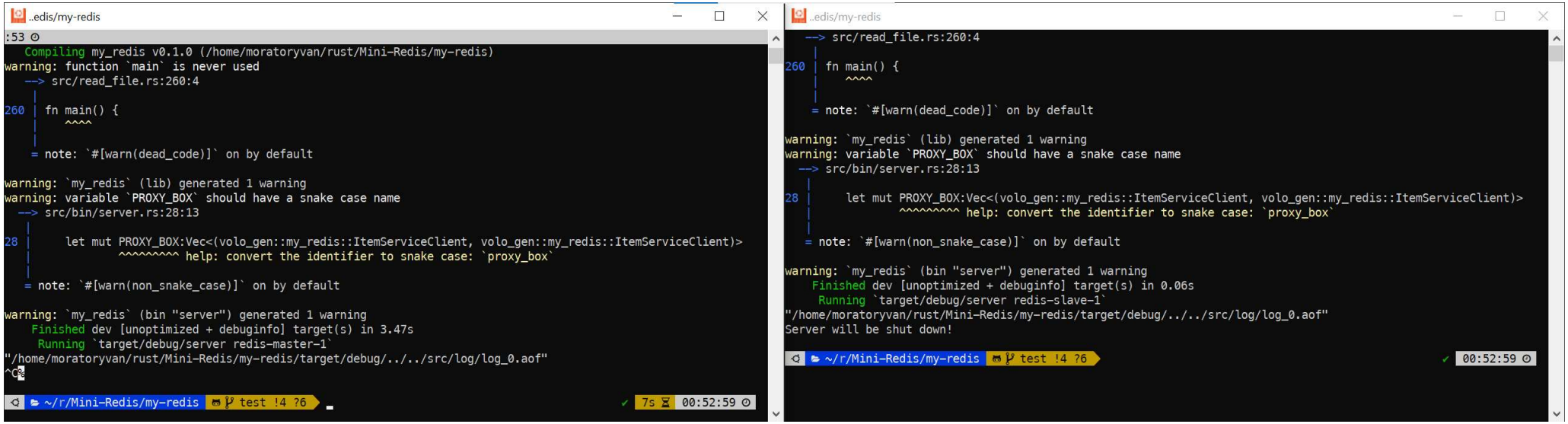
Graceful Exit

在我们向服务器发出退出信号时（如输入 CTRL+C 发送 SIGINT 信号时），我们的 server 主程序中的 run 方法会跳出循环。因此我们选择在 server.rs 中 run 方法返回后的部分处理其他任务。具体处理为：

- 如果当前是主从模式，并且我们是要退出一个主节点，那么我们向从属于这个主节点的从节点发送命令，要求其终止运行。（发送方式与 Part 2，3 中的发送方式相同）而从节点收到命令后，直接使用 exit(0) 进行终止。（此时后台的服务器已经执行完了退出信号之前的所有 Task，因此可以直接终止）
- 如果当前是 cluster 模式，我们要退出 proxy 节点，那么我们向所有节点都发送命令要求其终止运行。

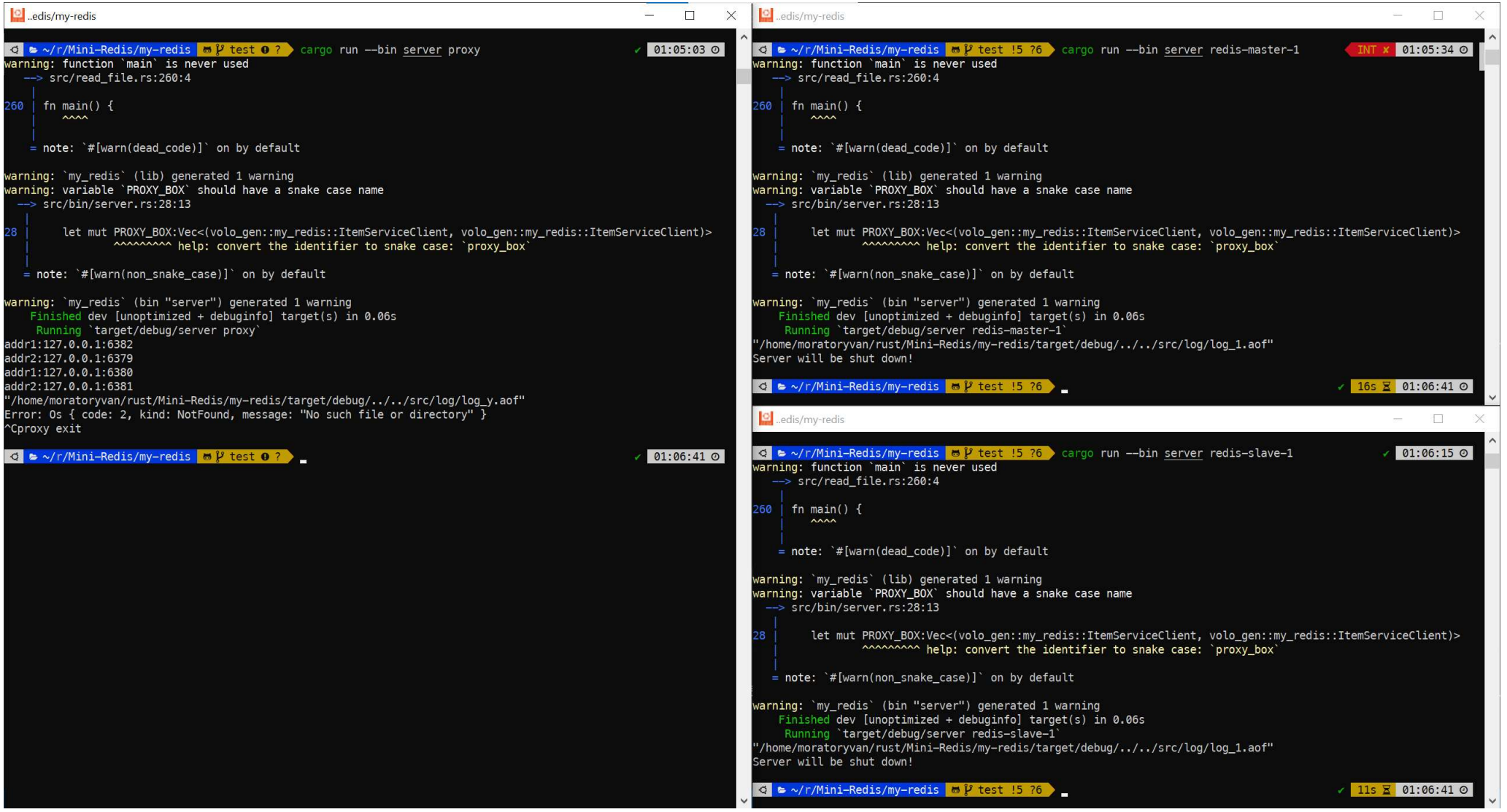
测试结果

主从架构下测试结果：



测试结果(Cont.)

Cluster 下测试结果:



总结

总结

这次我们实现了一个mini-redis，当然比我们想象的要庞大地多（x）。

总体上我们实现了所有要求，并且实现了graceful exit，成功实现了基于AOF机制的持久化策略，保证数据的稳定行；成功实现了主从架构，每次写操作同步主从服务器的数据，进一步保证数据稳定；同时实现了基于主从架构的mini-redis cluster，实现了一个Redis Proxy，通过hash值来分配每一个请求对应的服务器，以提高服务器的性能。实现了graceful exit。

但是因为时间匆促，难免有许多不足，在此，我们也有很多的不足：

THANKS