replication测试

rep_tests

rep_tests是replication的单元测试，由若干case组成。

case输入与输出

每个case都有一个case文件（.act），提供case运行过程中的指令（action）序列，每个指令独占一行；还有一个配置文件（.ini），提供运行时的rDSN配置。

case运行时，会从第一个指令开始，如果检查类指令得到满足或者动作类指令得到执行，就会前进到下一条指令，直到所有的指令都遍历完成。如果中间没有出现错误，则表明case运行成功。

指令主要分为两大类别：

- 检查类(check)：检查系统状态或者读写结果是否符合期望
  - state检查：当前各replica-server的状态是否符合期望
  - config检查：当前partition_configuration是否符合期望
  - client检查：client的异步读写收到的结果是否符合期望
- 动作类(action)：执行某种特定类型的动作
  - inject动作：在on_rpc_call或者on_aio_call的join point点，如果满足条件，则发起inject动作
  - client动作：client发起异步读写请求
  - set动作：一种特殊的动作，设置一些测试的options，譬如null_loop值
  - skip动作：一种特殊的动作，跳过接下来遇到的指定数量的检查

可以这样理解：检查类指令是被动指令；动作类指令是主动指令。

case运行时，会在stdout打印运行情况，同时还会生成case-{id}.out和case-{id}.log两个文件：

- stdout：case运行情况，每行对应一个action，并在行首标明该action的类型：
  - 数字：表示该action是在case文件中指定的，数字是其在文件中的行号
  - +号：表示没有在case文件指定的中间状态
  - s号：如果使用了skip指令，表示被跳过的动作和状态变化
  - 譬如：

```
./dsn.rep_tests.simple_kv case-000.ini case-000.act
      6  set:load_balance_for_test=1
      +  config:{0,-,[]}
      +  state:{{r1,ina,0,0}}
      +  config:{1,r1,[]}
      +  state:{{r1,pri,1,0}}
      +  state:{{r1,pri,1,0},{r2,pot,1,0}}
      +  state:{{r1,ina,1,0},{r2,pot,1,0}}
      +  config:{2,r1,[r2]}
      +  state:{{r1,pri,2,0},{r2,pot,1,0}}
      +  state:{{r1,pri,2,0},{r2,sec,2,0}}
      +  state:{{r1,pri,2,0},{r2,sec,2,0},{r3,pot,2,0}}
      +  state:{{r1,ina,2,0},{r2,sec,2,0},{r3,pot,2,0}}
      9  config:{3,r1,[r2,r3]}
      +  state:{{r1,pri,3,0},{r2,sec,2,0},{r3,pot,2,0}}
      +  state:{{r1,pri,3,0},{r2,sec,2,0},{r3,sec,3,0}}
     10  state:{{r1,pri,3,0},{r2,sec,3,0},{r3,sec,3,0}}
     13  client:begin_write:id=1,key=k1,value=v1,timeout=0
     16  state:{{r1,pri,3,1},{r2,sec,3,0},{r3,sec,3,0}}
     19  client:end_write:id=1,err=ERR_OK,resp=0
     22  client:begin_read:id=1,key=k1,timeout=0
     25  client:end_read:id=1,err=ERR_OK,resp=v1
     28  client:begin_write:id=2,key=k2,value=v2,timeout=0
      +  state:{{r1,pri,3,1},{r2,sec,3,1},{r3,sec,3,0}}
      +  state:{{r1,pri,3,1},{r2,sec,3,1},{r3,sec,3,1}}
     31  state:{{r1,pri,3,2},{r2,sec,3,1},{r3,sec,3,1}}
     34  client:end_write:id=2,err=ERR_OK,resp=0
```

- case-{id}.out：该case运行过程中动作与状态变化，与stdout基本一样，但没有在行首标明类型
- case-{id}.log：rDSN的日志输出，已经过滤掉了failure_detector相关的日志

在分析日志文件case-{id}.log时，可以根据task_id和rpc_id来将各种task事件连起来，方便理解和跟踪。

譬如下面几条日志就将m和r1上的两个task（0202000000000001和030031f600000001）通过rpc（8162773449786149）给串起来了。

```
I00:00:15.001 (15001000000 31f6)      m.THREAD_POOL_META_SERVER0.0202000000000001: load.balancer:178:send_proposal(): send proposal CT_ASSIGN_PRIMARY of 10.235.114.34:34801, current ballot = 0
D00:00:15.001 (15001000000 31f6)      m.THREAD_POOL_META_SERVER0.0202000000000001: simple_kv.case:1028:on_event(): === on_rpc_call:rpc_id=8162773449786149,rpc_name=RPC_CONFIG_PROPOSAL,from=m,to=r1
D00:00:15.011 (15011000000 31f9)     r1.io-thrd.12793: simple_kv.case:1028:on_event(): === on_rpc_request_enqueue:node=r1,task_id=030031f600000001,task_code=RPC_CONFIG_PROPOSAL,delay=0,rpc_id=8162773449786149,rpc_name=RPC_CONFIG_PROPOSAL,from=m,to=r1
D00:00:15.011 (15011000000 31fe)     r1.replication1.030031f600000001: simple_kv.case:1028:on_event(): === on_task_begin:node=r1,task_id=030031f600000001,task_code=RPC_CONFIG_PROPOSAL,delay=0
```


```
I00:00:15.001 (15001000000 31f6)      m.THREAD_POOL_META_SERVER0.0202000000000001: load.balancer:178:send_proposal(): send proposal CT_ASSIGN_PRIMARY of 10.235.114.34:34801, current ballot = 0
D00:00:15.001 (15001000000 31f6)      m.THREAD_POOL_META_SERVER0.0202000000000001: simple_kv.case:1028:on_event(): === on_rpc_call:rpc_id=8162773449786149,rpc_name=RPC_CONFIG_PROPOSAL,from=m,to=r1
D00:00:15.011 (15011000000 31f9)     r1.io-thrd.12793: simple_kv.case:1028:on_event(): === on_rpc_request_enqueue:node=r1,task_id=030031f600000001,task_code=RPC_CONFIG_PROPOSAL,delay=0,rpc_id=8162773449786149,rpc_name=RPC_CONFIG_PROPOSAL,from=m,to=r1
D00:00:15.011 (15011000000 31fe)     r1.replication1.030031f600000001: simple_kv.case:1028:on_event(): === on_task_begin:node=r1,task_id=030031f600000001,task_code=RPC_CONFIG_PROPOSAL,delay=0
```

```
I00:00:15.001 (15001000000 31f6)      m.THREAD_POOL_META_SERVER0.0202000000000001: load.balancer:178:send_proposal(): send proposal CT_ASSIGN_PRIMARY of 10.235.114.34:34801, current ballot = 0
D00:00:15.001 (15001000000 31f6)      m.THREAD_POOL_META_SERVER0.0202000000000001: simple_kv.case:1028:on_event(): === on_rpc_call:rpc_id=8162773449786149,rpc_name=RPC_CONFIG_PROPOSAL,from=m,to=r1
D00:00:15.011 (15011000000 31f9)     r1.io-thrd.12793: simple_kv.case:1028:on_event(): === on_rpc_request_enqueue:node=r1,task_id=030031f600000001,task_code=RPC_CONFIG_PROPOSAL,delay=0,rpc_id=8162773449786149,rpc_name=RPC_CONFIG_PROPOSAL,from=m,to=r1
D00:00:15.011 (15011000000 31fe)     r1.replication1.030031f600000001: simple_kv.case:1028:on_event(): === on_task_begin:node=r1,task_id=030031f600000001,task_code=RPC_CONFIG_PROPOSAL,delay=0
```


case组织

每个case都指定了一个case id，其为定长3位的数字，譬如000, 100, 201等。

case有两类：

- 单次运行即可完成的case：case文件为case-{id}.act，配置文件为case-{id}.ini。譬如case-000对应的文件为：case-000.act、case-000.ini。
- 多次运行才能完成的case：
    - 整个case由多个subcase组成，每个subcase都指定了一个subid，其为定长1位的数组。
      每个sub-case都有独立的case文件和配置文件，case文件为case-{id}-{subid}.act，配置文件为case-{id}-{subid}.ini。
      譬如case-300包含3个subcase，分别为case-300-0、case-300-1、case-300-2。
    - 整个case按照subid顺序执行，subcase运行之间不会清理环境。

case id分类规则（仅供建议）：

- 000：正常运行
- 100~199：所有replica_server之间的RPC错误
- 200~299：所有replica_server的aio错误
- 300~399：所有replica_server与meta_server之间的RPC错误，不含fd通信
- 400~499：以上错误的组合
- 500~599：failure detector测试

新建case：可以运行./addcase.sh <new-case-id> <from-case-id> 来增加新的case，会以<from-case-id>作为模板生成新的case。

case语法

case输入文件以行为单位，每一行可以是：

- 空行，会被忽略
- #号开头的注释行，会被忽略
- 一条指令

指令的格式总是"type:params"，type是该指令的类型，params是该指令的参数。

目前支持的指令类型有：

- state
- config
- inject
- client
- skip
- set

STATE指令

state指令是检查类指令，用于检查系统中所有replica-server的状态是否符合期望。譬如：

```
state:{{r1,pri,3,0}{r2,sec,3,0}{r3,sec,3,0}}
```

其中：

- params是状态集合
- {r1,pri,3,0}表示一个replica-server的状态：r1表示所在节点名，pri表示状态为primary，3表示ballot，0表示last_committed_decree

执行规则：

- 当遇到该指令，会检查当前所有replica-server的状态，如果与期望状态一致，则前进到下一条指令
- 如果不一致：
  - 如果当前状态位于期望状态之前（ballot和last_committed_decree都不大于期望状态的值），则继续执行当前指令，直到状态变化到期望状态
  - 否则，case失败

CONFIG指令

config指令是检查类指令（和state指令很类似），用于检查partition_configuration的状态是否符合期望。譬如：

```
config:{2,r1,[r2,r3]}
```

其中：

- {2,r1,[r2,r3],0}表示：2表示ballot，r1表示primary节点名，r2表示secondary节点列表

执行规则：

- 当遇到该指令，会检查当前partition_configuration的状态，如果与期望状态一致，则前进到下一条指令
- 如果不一致：
  - 如果当前状态位于期望状态之前（ballot不大于期望状态的值），则继续执行当前指令，直到状态变化到期望状态
  - 否则，case失败

INJECT指令

inject指令是动作类指令，用于通过钩子函数注入错误（譬如RPC错误和IO错误）。

on_rpc_call

在发送RPC请求时，如果满足条件，则注入错误。譬如：

```
inject:on_rpc_call:rpc_name=rpc_prepare,from=r1,to=r2
```

其中：

- rpc_name=xxx：rpc名（不区分大小写）
- from=xxx：源节点
- to=xxx：目标节点
- 以上条件为AND关系

执行规则：

- 当遇到该指令，在执行on_rpc_call钩子的时候会检查条件，如果条件满足，则注入错误，并前进到下一条指令

on_rpc_reply

在发送RPC回复时，如果满足条件，则注入错误。譬如：

```
inject:on_rpc_reply:rpc_name=rpc_prepare_ack,from=r2,to=r1
```

其中：

- rpc_name=xxx：rpc名（不区分大小写）
- from=xxx：源节点
- to=xxx：目标节点
- 以上条件为AND关系，对于不设置的条件不会做检查

执行规则：

- 当遇到该指令，在执行on_rpc_reply钩子的时候会检查条件，如果条件满足，则注入错误，并前进到下一条指令

on_aio_call

在发起IO调用时，如果满足条件，则注入错误。譬如：

```
inject:on_aio_call:node=r1,type=read,file_offset=0,buffer_size=100
```

其中：

- node=xxx：所在节点
- type=read|write：aio类型，read为读，write为写（不区分大小写）
- file_offset=xxx：读写文件起始位置
- buffer_size=xxx：读写buffer大小
- 以上条件为AND关系，对于不设置的条件不会做检查

执行规则：

- 当遇到该指令，在执行on_aio_call钩子的时候会检查条件，如果条件满足，则注入错误，并前进到下一条指令

on_aio_enqueue

在IO完成后enqueue回调task时，如果满足条件，则注入错误。譬如：

```
inject:on_aio_enqueue:node=r1,type=read,file_offset=0,buffer_size=100,err=ERR_OK,transfered_size=100
```

其中：

- node=xxx：所在节点
- type=read|write：aio类型，read为读，write为写（不区分大小写）
- file_offset=xxx：读写文件起始位置
- buffer_size=xxx：读写buffer大小
- err=xxx：返回码（不区分大小写）

- transfered_size=xxx：成功的字节数
- 以上条件为AND关系，对于不设置的条件不会做检查

执行规则：

- 当遇到该指令，在执行on_aio_call钩子的时候会检查条件，如果条件满足，则注入错误，并前进到下一条指令

CLIENT指令

client指令主要用于发起读写请求、检查读写结果。其中发起读写请求是动作类指令，检查读写结果是检查类指令。

begin_write

发起异步写请求，动作类指令。譬如：

```
client:begin_write:id=1,key=aaa,value=bbb,timeout=0
```

其中：

- id用于和end_write进行匹配（类型为数字），key和value是写请求的数据（区分大小写），timeout是超时时间（单位ms，0表示使用默认超时）

执行规则：

- 当执行到该条指令时，立即发起异步写操作，然后前进到下一条指令

end_write

检查异步写结果，检查类指令。譬如

```
client:end_write:id=1,err=err_ok,resp=0
```

其中：

- id用于和begin_write进行匹配，err是期望返回的error_code的字符串形式（不区分大小写），resp是期望返回的resp结果

执行规则：

- 当执行到该条指令时，等待client收到write回复，并检查结果，如果结果和期望的结果一致，则前进到下一条指令
- 如果结果不一致，则case失败

begin_read

发起异步读请求，动作类指令。譬如：

```
client:begin_read:id=2,key=aaa,timeout=0
```

其中：

- id用于和end_read进行匹配，key是读请求的数据，timeout是超时时间（单位ms，0表示使用默认超时）

执行规则：

- 当执行到该条指令时，立即发起异步读操作，然后前进到下一条指令

end_read

检查异步读结果，检查类指令。譬如：

```
client:end_read:id=2,err=err_ok,resp=bbb
```

其中：

- id用于和begin_read进行匹配，err是期望返回的error_code的字符串形式（不区分大小写），resp是期望返回的resp结果

执行规则：

- 当执行到该条指令时，等待client收到read回复，并检查结果，如果结果和期望的结果一致，则前进到下一条指令
- 如果结果不一致，则case失败

SKIP指令

特殊动作类执行，用于跳过接下来的N次检查。譬如：

```
skip:100
```

SET指令

特殊动作类指令，用于设置系统的options参数。譬如：

```
set:null_loop=1000,disable_load_balance=1
```

options参数包括：

null_loop

null_loop=N：系统空转次数。如果系统内部连续调度N次的过程中，一致没有出现state变化、config变化、指令前进中的一种，则case失败。该参数用于避免系统进入死循环状态。

disable_load_balance

disable_load_balance=1|0：是否关闭load balance功能。如果=1，表示关闭，则在此之后meta_server都将不再进行load balance操作；关闭后可重新开启，只需设置为0即可。

misc.sh辅助脚本

使用前配置

在.bashrc中添加:

export PROJ_ROOT=/path/to/your/rdsn-frame

source $PROJ_ROOT/misc.sh

使用

r

进入$PROJ_ROOT

g

对于一个CMake Target(executable, static library, shared library),在源代码目录和编译的目标目录间相互切换.

注意:对于library, 因为多个目标可能共享一个lib目录,所以g只能部分工作.即只支持从源代码目录切换到目标目录

```
g
[weijiesun@WeijieSun-PC simple_kv]$ pwd
/home/weijiesun/rDSN/rdsn-frame/src/rep_tests/simple_kv
[weijiesun@WeijieSun-PC simple_kv]$ g
[weijiesun@WeijieSun-PC dsn.rep_tests.simple_kv]$ pwd
/home/weijiesun/rDSN/rdsn-frame/builder/bin/dsn.rep_tests.simple_kv
[weijiesun@WeijieSun-PC dsn.rep_tests.simple_kv]$ g
[weijiesun@WeijieSun-PC simple_kv]$ pwd
/home/weijiesun/rDSN/rdsn-frame/src/rep_tests/simple_kv
[weijiesun@WeijieSun-PC simple_kv]$
```

b

进入$PROJ_ROOT/builder

gcp

对于一个CMake Target.如果当前在源代码目录,则把指定文件拷贝到目标目录;如果当前在目标目录,则拷贝到源代码目录.

例:

```
gcp
[weijiesun@WeijieSun-PC simple_kv]$ touch aaa bbb ccc
[weijiesun@WeijieSun-PC simple_kv]$ ls -l aaa bbb ccc
-rw-r--r-- 1 weijiesun users 0 11月 12 15:30 aaa
-rw-r--r-- 1 weijiesun users 0 11月 12 15:30 bbb
-rw-r--r-- 1 weijiesun users 0 11月 12 15:30 ccc
[weijiesun@WeijieSun-PC simple_kv]$ gcp aaa bbb ccc
[weijiesun@WeijieSun-PC simple_kv]$ g
[weijiesun@WeijieSun-PC dsn.rep_tests.simple_kv]$ ls -l aaa bbb ccc
-rw-r--r-- 1 weijiesun users 0 11月 12 15:32 aaa
-rw-r--r-- 1 weijiesun users 0 11月 12 15:32 bbb
-rw-r--r-- 1 weijiesun users 0 11月 12 15:32 ccc
[weijiesun@WeijieSun-PC dsn.rep_tests.simple_kv]$
```

src

进入指定CMake Target项目的源代码目录

例:

```
src
[weijiesun@WeijieSun-PC rdsn-frame]$ src echo
[weijiesun@WeijieSun-PC echo]$ pwd
/home/weijiesun/rDSN/rdsn-frame/src/apps/echo
[weijiesun@WeijieSun-PC echo]$ src failure
[weijiesun@WeijieSun-PC failure_detector]$ pwd
/home/weijiesun/rDSN/rdsn-frame/src/dist/failure_detector
[weijiesun@WeijieSun-PC failure_detector]$ src rep_test
[weijiesun@WeijieSun-PC simple_kv]$ pwd
/home/weijiesun/rDSN/rdsn-frame/src/rep_tests/simple_kv
[weijiesun@WeijieSun-PC simple_kv]$ src pegasus_meta_server
[weijiesun@WeijieSun-PC meta]$ pwd
/home/weijiesun/rDSN/rdsn-frame/src/apps/pegasus/meta
```

dst

进入指定CMake Target项目的目标目录

发现问题

- mutation_log::garbage_collection_when_as_commit_logs()返回值为count，但是在replica::init_commit_log_service()中却认为其返回值为error_code
- replica::on_update_configuration_on_meta_server_reply中,如果meta_server返回INVALID_VERSION,replica在重传时候会将request的引用计数加一,导致内存泄露
- mutation_log在replay的时候,如果最后一个文件是空的,replay的过程会coredump
- rpc如果出错（譬如连接失败），是否能立即得到失败的callback，还是要等到timeout？
- 在replica重启的时候:
    - 如果发生数据损坏,replica会直接把原prepare log的目录改为log.err,如果之前还有log.err,则删除.如果此处的重启为supervisor自动重启,数据是否要直接删除需要商榷
    - 删除数据后,replica会重新初始化一个新的空log目录,原来的数据全部都没了;如果只有一个replica不可用,这是没问题的;但如果多个挂了,或者整个集群重启,meta server的lb就不能随便assigin了
- group_check返回值问题