分布式系统作业

大作业

姓名：郝裕玮
班级：计科 1 班
学号：18329015

目录

1 实验项目简介

设计并实现一个分布式键值（key-value）存储系统，可以是基于磁盘的存储系统，也可以是基于内存的存储系统，可以是主从结构的集中式分布式系统，也可以是 P2P 式的非集中式分布式系统。能够完成基本的读、写、删除等功能，支持缓存、多用户和数据一致性保证。

具体要求如下：

（1）必须是分布式的键值存储系统，至少在两个节点或者两个进程中测试；

（2）可以是集中式的也可以是非集中式；

（3）能够完成基本的操作如：PUT、GET、DEL 等；

（4）支持多用户同时操作；

（5）至少实现一种面向客户的一致性如单调写；

（6）需要完整的功能测试用例；

（7）涉及到节点通信时须采用 RPC 机制。

加分项：

（1）具备性能优化措施如 cache 等；

（2）具备失效容错方法如：Paxos、Raft 等；

（3）具备安全防护功能；

（4）其他高级功能；

2 环境配置

编程语言：Python 3.7.0

第三方库：rpyc + sqlitedict

对上述 2 个库执行 pip install xxx 即可。

3 实验内容

3.1 原理分析

我实现的分布式键值系统主要由以下 4 个部分组成：

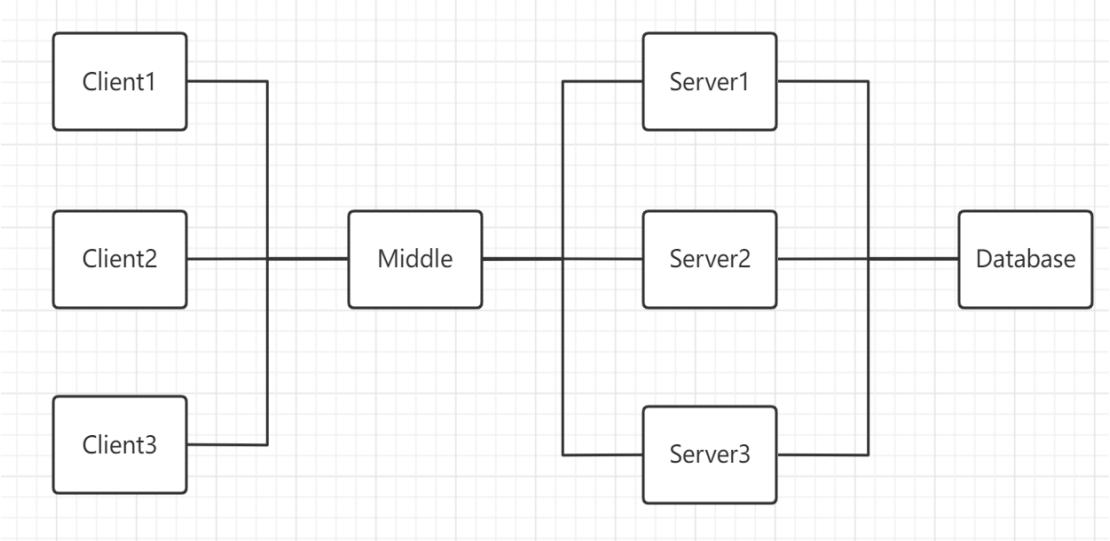
（1）客户端 Client；

（2）分布式系统中间件 Middle；

（3）服务器 Server；

（4）数据库 Database。

他们的关系图如下所示：



其中 Client_x 通过 Middle 中间件与对应的 Server_x 相连接：Client 发出去请求给 Middle 中间件，Middle 中间件再调用 Server_x 上的功能函数，从而最终对 Database 进行查询或修改，并将结果返回给 Client。同时由于所有 Server 都连接到同一个 Database，所以 Database 的任意变化都会在该次操作结束后同步到所有 Server 上。

所以很容易证明我实现的分布式系统具有单调读一致性和单调写一致性。

为了使种类各异的计算机和网络都呈现为单个的系统，分布式系统常常通过一个“软件层”组织起来，该“软件层”在逻辑上位于由用户和应用程序组成的高层与由操作系统组成的低层之间，如图 1.1 所示。这样的分布式系统有时又称为**中间件**(middleware)。

图 1.1　作为中间件组织的分布式系统。中间件层延伸到了多台机器上，且为每个应用程序提供了相同的接口

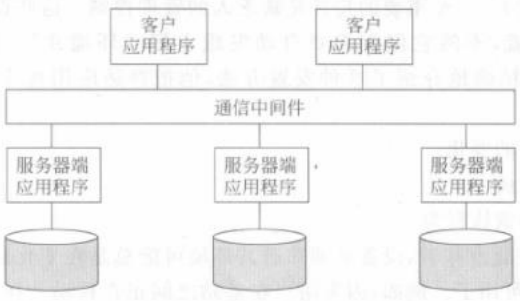


图 1.11　在企业应用集成中，中间件作为一种通信工具

现在已经有了几种通信中间件了。**远程过程调用**(remote procedure call，RPC)就是这样一种应用程序组件，它可以通过一个本地过程调用，有效地往另一个应用程序组件发送一个请求。本地过程调用可以使其请求封装成一个消息，并发送给被调用者。同样，过程调用的结果将被发送回来并返给应用程序。

3.2 代码分析

由 3.1 可知我们需要实现 3 个部分的代码：Client，Middle，Server（Database 在 Server 中初始化并连接即可）。

具体代码如下所示（设计思路和功能分析均已包含在代码注释中）：

（1）Client.py

```
#coding=utf-8
import argparse
import rpyc
# argparse 库:用于解析参数,并为这些参数自动生成帮助和使用信息(用于美化 UI
界面)
# rpyc 库:它是一个 Python 的库,用于实现 RPC 和分布式计算

#Client 类:客户端
class Client(object):
    #在 cmd 中输入 help 会显示如下信息
    """
    Commands Help:
    PUT key value —— Generate/Modify (key, value)
    GET key —— Query (key, value) by the key
    GETALL —— Get all (key, value) in the database
    DEL key —— Delete (key, value) by the key
    DELALL —— Delete all (key, value) in the database
    GETLOG —— Get the log
    """

    #客户端与 master 节点连接
    def connect(self):
        self.conn = rpyc.connect('localhost', 21000)
        #获取自己独有的客户 ID
        #client 要访问 Master 节点的代码必须通过 self.conn.root.xxx 才能访
问
        self.id = self.conn.root.get_id()
        return self.id

    #进入运行界面
    def run(self):
```

```
        try:
            while True:
                #输入指令
                command = input("Client %d >> " %self.id)
                #指令为 help 则输出指南
                if command == 'help':
                    print(self.__doc__)
                #反之则调用 Master 节点中对应的功能函数
                else:
                    msg = self.conn.root.function(self.id, command)
                    #打印结果信息
                    if msg != None:
                        print(msg)
        except KeyboardInterrupt:
            pass


#主程序
if __name__ == '__main__':
    users = {} #初始用户列表为空
    #打开文件读取
    with open('user.txt') as file:
        for i in file.readlines():
            #每一行是用户名+空格+密码,用 split 函数将其分开
            #users[i[0]] = i[1]代表建立字典对应值,即 users[用户名]=密码
            i = i.split()
            users[i[0]] = i[1]

    #用户登录
    username = input('Please input your username:')
    password = input('Please input your password:')
    #检测用户名和密码是否匹配
    if username in users.keys() and users[username] == password:
        client = Client()
        client_id = client.connect()

        if client_id == None:
            print('Connection Failed.')
        else:
            print('\nWelcome to Distributed Key-Value System!\n')
            print('Your client ID is %d\n' %client_id)
            print("Enter \"help\" for the list of commands:\n")
            client.run()
    else:
```

```
        print('Your username or password has something wrong.Please
try again!')
```

（2）Middle.py

```
#coding=utf-8
import argparse
import rpyc
from rpyc.utils.server import ThreadedServer
# argparse 库:用于解析参数,并为这些参数自动生成帮助和使用信息(用于美化 UI
界面)
# rpyc 库:它是一个 Python 的库,用于实现 RPC 和分布式计算

#Middle 类:分布式系统中间件
class Middle(rpyc.Service):
    #当客户端到服务器的连接建立时,on_connect 函数会被执行
    def on_connect(self, conn):
        pass

    #当客户端到服务器的连接断开时,on_disconnect 函数会被执行
    def on_disconnect(self, conn):
        #依次断开客户端和服务器的连接
        for temp in clients:
            temp.close()

    #为当前客户分配 ID
    def exposed_get_id(self):
        #遍历 client_ids 表,寻找尚未分配的 id
        for i in range(len(client_ids)):
            #若 client_ids[i]尚未分配则设置其为 True(已使用)并返回序号 i
            if client_ids[i] == False:
                client_ids[i] = True
                return i
        return None

    #执行各种 command
    def exposed_function(self, client_id, clause):
        clause = clause.strip().split()
        #将 command 根据空格进行分割
        #strip()可消去字符串前后的空格(不包括中间)
        #split()将字符串根据空格进行分割
        lens = len(clause)
```

```
        WRONG_MSG = 'Wrong command. Enter help if necessary.'
        if lens < 1:
            return WRONG_MSG

        #将指令的第一个单词全部转为小写,便于后续判定
        #易知指令的第一个单词小写化后只可能
是:put,get,getall,del,delall,getlog
        command = clause[0].lower()

        #开始进行条件判断
        #对于 PUT key value
        if command == 'put':
            if lens == 3:
                key = clause[1]
                value = clause[2]
                #client 要访问服务器的代码必须通过 self.conn.root.xxx 才
能访问
                clients[client_id].root.Put(key, value)
                #将本次操作 PUT key value 写到对应客户的日志中
                clients[client_id].root.Write_Log('PUT
('+str(key)+','+str(value)+')')
            else:
                return WRONG_MSG

        #对于 GET key
        if command == 'get':
            if lens == 2:
                key = clause[1]
                #client 要访问服务器的代码必须通过 self.conn.root.xxx 才
能访问
                result = clients[client_id].root.Get(key)
                #将本次操作 GET key 写到对应客户的日志中
                clients[client_id].root.Write_Log('GET '+str(key))
            else:
                return WRONG_MSG

            if result == None:
                return 'Key %s not found.' %key
            else:
                return result

        #对于 GETALL
        if command == 'getall':
            if lens == 1:
```

```
                #client 要访问服务器的代码必须通过 self.conn.root.xxx 才
能访问
                #将本次操作 GETALL 写到对应客户的日志中
                clients[client_id].root.Write_Log('GETALL')
                return clients[client_id].root.Get_All()
            else:
                return WRONG_MSG

        #对于 DEL key
        if command == 'del':
            if lens == 2:
                key = clause[1]
                #client 要访问服务器的代码必须通过 self.conn.root.xxx 才
能访问
                clients[client_id].root.Delete(key)
                #将本次操作 DEL key 写到对应客户的日志中
                clients[client_id].root.Write_Log('DEL '+str(key))
            else:
                return WRONG_MSG

        #对于 DELALL
        if command == 'delall':
            if lens == 1:
                #client 要访问服务器的代码必须通过 self.conn.root.xxx 才
能访问
                #将本次操作 GETALL 写到对应客户的日志中
                clients[client_id].root.Write_Log('DELALL')
                return clients[client_id].root.Delete_All()
            else:
                return WRONG_MSG

        #对于 GETLOG
        if command == 'getlog':
            if lens == 1:
                #client 要访问服务器的代码必须通过 self.conn.root.xxx 才
能访问
                return clients[client_id].root.Get_Log()
            else:
                return WRONG_MSG


#主程序
if __name__ == '__main__':
    #(1)创建 ArgumentParser()对象
```

```
    #(2)调用 add_argument()方法添加参数
    #(3)使用 parse_args()解析添加的参数
    parser = argparse.ArgumentParser()
    parser.add_argument('--p', type=int, default=1)
    args = parser.parse_args()

    #建立一个 bool 数组,初始化为 args.p 个 False 变量,表示这些 id 都未被使用
    client_ids = [False] * args.p

    #设置中间件的监听端口号用于和不同的服务器节点相连接
    clients = [rpyc.connect('localhost', 20000+i) for i in
range(args.p)]
    #设置中间件与客户端连接的端口号
    middle = ThreadedServer(Middle, port=21000)
    print("Middleware is running...\n")
    middle.start()
```

（3）Server.py

```
#coding=utf-8
import argparse
import rpyc
from rpyc.utils.server import ThreadedServer
from multiprocessing import Process
from sqlitedict import SqliteDict
# argparse 库:用于解析参数,并为这些参数自动生成帮助和使用信息(用于美化 UI
界面)
# rpyc 库:它是一个 Python 的库,用于实现 RPC 和分布式计算
# multiprocessing 库:用于编写多线程/多进程程序
# sqlitedict 库:用于将字典信息存入数据库中,且支持多线程访问

#用于记录当前进程下的每步操作(一个客户对应一个进程)
log=[]

#Server 类:服务器
class Server(rpyc.Service):
    #当客户端到服务器的连接建立时,on_connect 函数会被执行
    def on_connect(self, conn):
        #连接建立时,创建一个字典数据库用于存储键值
        #'./database.sqlite'代表数据库文件存储路径和文件名
        #autocommit=True 代表会对每次对数据库操作的结果自动提交
```

```
        self.database = SqliteDict('./database.sqlite',
autocommit=True)

    #当客户端到服务器的连接断开时,on_disconnect 函数会被执行
    def on_disconnect(self, conn):
        #pass 代表不进行任何操作
        pass

    #注意,Server 类中"exposed_"开头的函数才能被客户端调用
    #调用时需将对应函数的 exposed_删去再调用
    #生成/修改键值对(key,value)
    def exposed_Put(self, key, value):
        self.database[key] = value

    #查询键值对(key,value)
    def exposed_Get(self, key):
        #通过 key 来查询对应的 value
        if key not in self.database:
            return None
        else:
            return self.database[key]

    #删除键值对(key,value)
    def exposed_Delete(self, key):
        if key not in self.database:
            #return
            return None
        else:
            #根据键值 k 删除键值对(key,value)
            del self.database[key]

    #获取数据库中所有键值对(key,value)
    def exposed_Get_All(self):
        res = [(key, self.database[key]) for key in self.database]
        res.sort()#对键值对进行字典序排序(根据 key 值排序)
        return res

    #删除数据库中所有键值对(key,value)
    def exposed_Delete_All(self):
        all_keys = [key for key in self.database]
        for key in all_keys:
            del self.database[key]

    #将操作记录写入日志
```

```
    def exposed_Write_Log(self,msg):
        log.append(msg)

    #展示日志
    def exposed_Get_Log(self):
        return log

#设置服务器监听端口号并运行服务器
def run(id):
    port1 = id + 20000
    server = ThreadedServer(Server, port=port1)
    try:
        #运行服务器
        server.start()
    except KeyboardInterrupt:
        server.close()

#主程序
if __name__ == '__main__':
    #(1)创建 ArgumentParser()对象
    #(2)调用 add_argument()方法添加参数
    #(3)使用 parse_args()解析添加的参数
    parser = argparse.ArgumentParser()
    parser.add_argument('--p', type=int, default=1)
    args = parser.parse_args()

    #限制服务器在同一时间段内最多只能接受 10 个客户端的连接和请求
    if args.p > 10:
        raise Exception("The max number of clients is 10.")

    #创建 args.p 个进程
    processes = [Process(target=run,args=(i,)) for i in
range(args.p)]
    print("Server is running and it can connect with %d clients at
the same time." %args.p)

    #启动 args.p 个进程,并用 join 函数进行堵塞
    #join 函数:在进程中可以阻塞主进程的执行,直到等待子线程全部完成之后,再继
续运行主线程后面的代码
    for i in range(args.p):
        processes[i].start()
    for i in range(args.p):
        processes[i].join()
```

3.3 功能测试

详见压缩包中“功能测试.mp4”（包含语音讲解）。

4 总结

4.1 遇到的问题和解决办法

（1）由于这次使用了 python 的 rpyc 库（Python 的 RPC）和 sqlitedict 库（Python 的 sqlite3 数据库的轻量级包装器，具有简单的 Pythonic 类 dict 接口并支持多线程访问），所以在调用函数和函数接口上花费了很多时间去查阅资料和学习。

（2）一开始不知道自己应该怎样设计系统结构，在重新回归书本后我得到了答案：利用中间件来连接客户端和服务器即可。

为了使种类各异的计算机和网络都呈现为单个的系统，分布式系统常常通过一个“软件层”组织起来，该“软件层”在逻辑上位于由用户和应用程序组成的高层与由操作系统组成的低层之间，如图 1.1 所示。这样的分布式系统有时又称为**中间件**（middleware）。

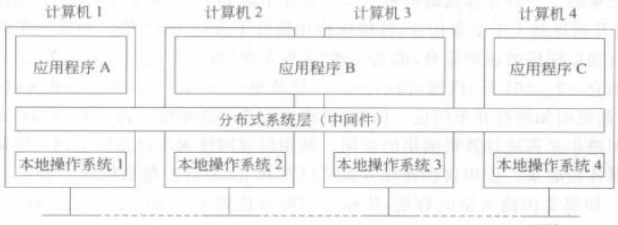


图 1.1 作为中间件组织的分布式系统。中间件层延伸到了多台机器上，且为每个应用程序提供了相同的接口

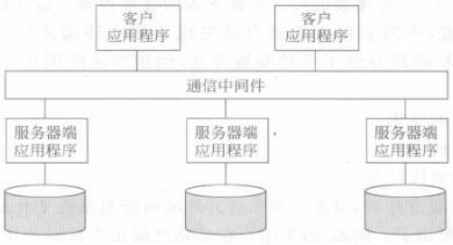


图 1.11 在企业应用集成中，中间件作为一种通信工具

现在已经有了几种通信中间件了。**远程过程调用**（remote procedure call，RPC）就是这样一种应用程序组件，它可以通过一个本地过程调用，有效地往另一个应用程序组件发送一个请求。本地过程调用可以使其请求封装成一个消息，并发送给被调用者。同样，过程调用的结果将被发送回来并返给应用程序。

4.2 实验心得

经过这次实验，我对分布式系统有了更深一步的理解，明白了分布式系统在日常生活中的重要作用（比如不同客户使用不同功能时，感觉上是只和一台服务器交互，实际上是多个服务器共同运行的结果）。同时我也对 RPC 的原理及其运用更加熟练。

至此，本次实验圆满完成。