分布式系统 Lab5 实验报告

赵悦棋 517021910158

1. 实验环境

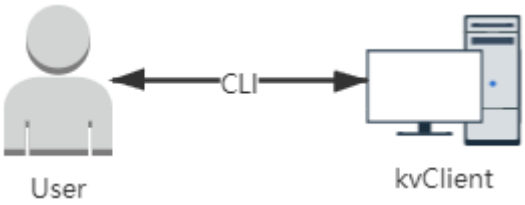
操作系统：macOS 10.15.5
开发语言：Java 8
RPC 框架：RMI
Zookeeper：3.6.1
依赖：Apache Commons Lang
IDE：IntelliJ IDEA

2. 开发重心

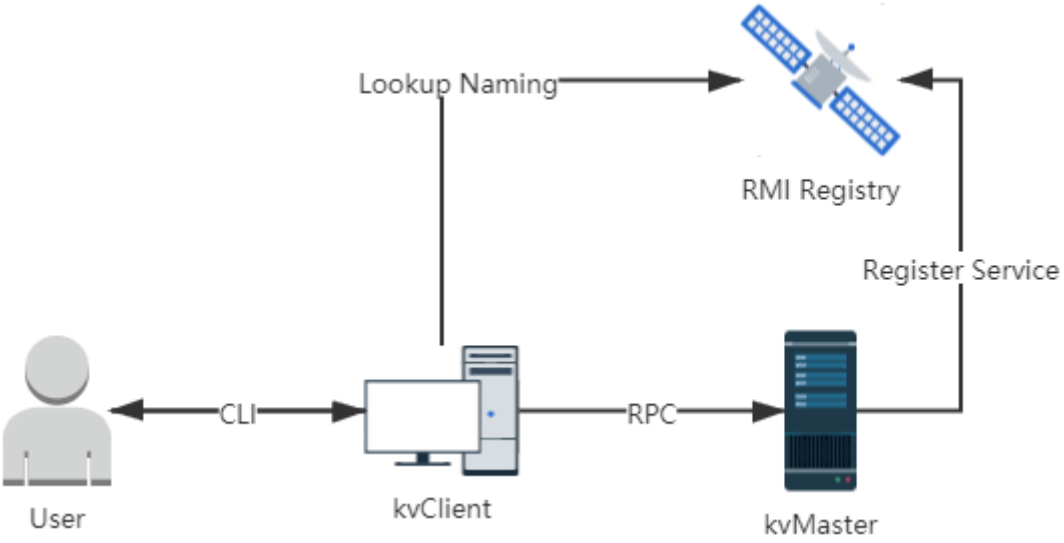
在该实验的开发过程中，我越发觉得整套系统的功能可以无止尽地扩充，比如在实现了基本的内存存储之后就会想要持久化地写到磁盘文件里，一旦设计到写文件又想把日志功能也开发出来……但是时间是有限的，因此在本次实验中我选择了只做适当的功能扩展，但把每一个写好的功能都写到极致，比如我的客户端具备完整的 CLI 属性，比如我的所有程序最终都能够优雅地自然终止不需要手动强制停止等，比如我没有任何写死的数据，因此可以不做任何修改直接上云。具体参见下文。

3. 开发过程

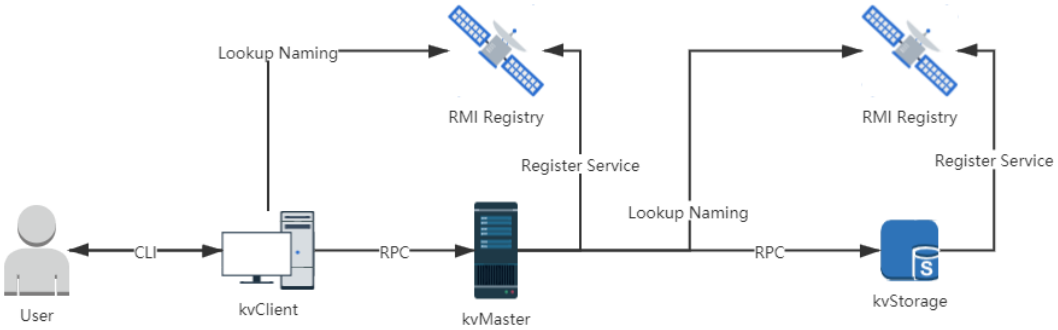
我先写了客户端的基本逻辑，实现了一个 CLI 的可交互客户端，需要用到远程方法调用的地方暂时留空，首先保证我可以对整个系统进行输入。



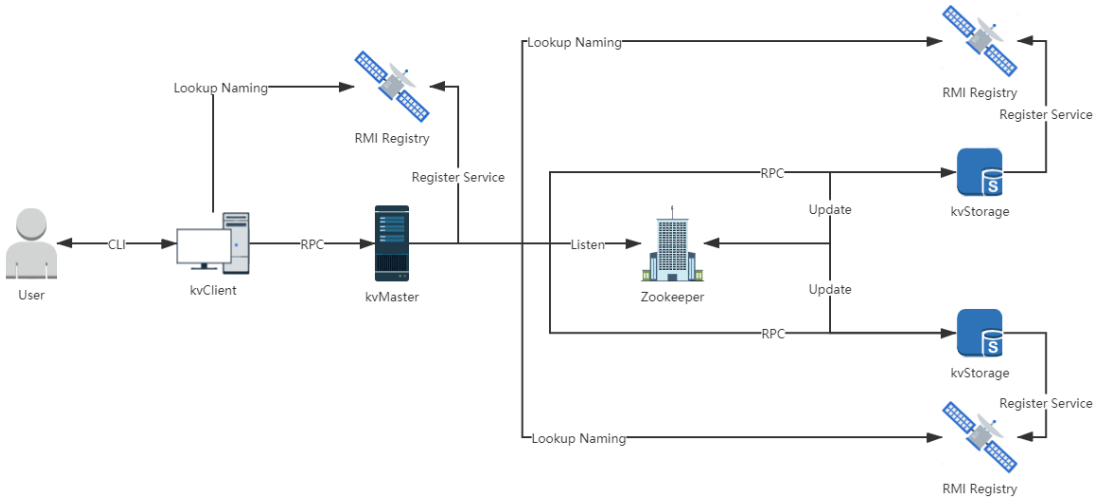
接着开始写服务端 Master 节点，参考网上的教程我完成了 RMI 的启动，之后按照“实现一个接口，去客户端补齐远程方法调用”的流程依次将 PUT、READ、DELETE 操作以及额外实现的 SHUTDOWN 操作在 Client/Master 这一侧实现，客户端至此完毕。



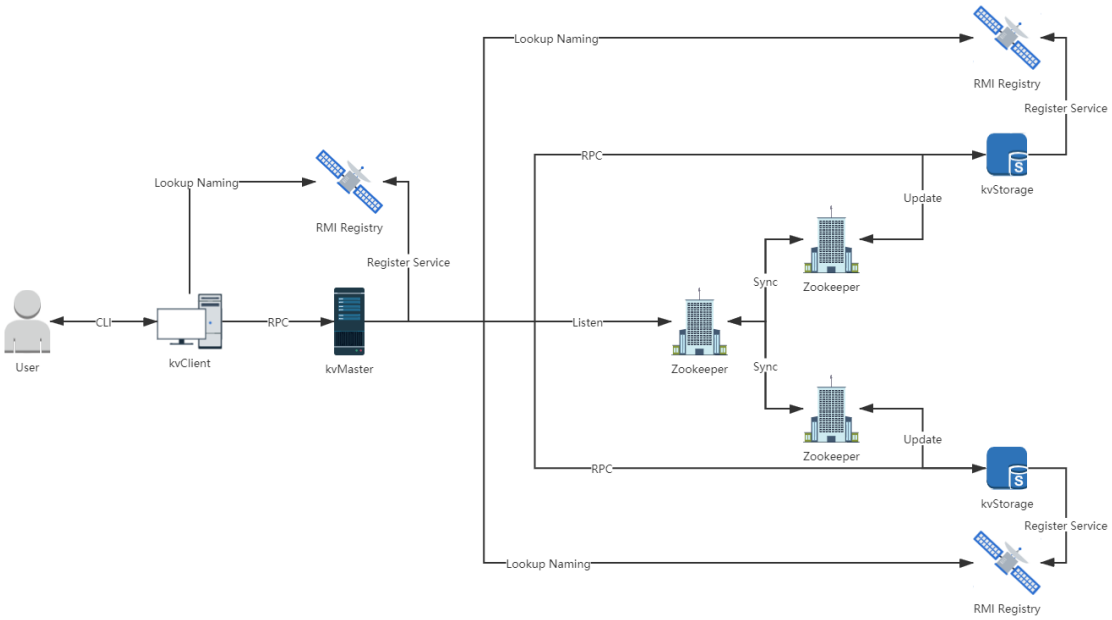
然后开始写 Storage 节点，基本和之前的 Client/Master 的开发流程一致，至此实现了 Client→Master →Storage 的基础架构，并在此基础上添加了 Master 的并发控制。



此前 Master 的主机信息是在 Client 端作为启动参数给定的，这符合常理，而 Storage 的主机信息在 Master 则是写死的，并不合理，引入单节点的 Zookeeper 修改为 Storage 在启动时向 Zookeeper 注册信息，同时在自己状态变更时更新 Znode 信息，而 Master 通过监听 Zookeeper 获取 Storage 的状态，并在此基础上添加了多节点、负载均衡、高可用（备份）等功能。



最后把单节点的 Zookeeper 扩展为多节点的 Zookeeper 集群，并让 Storage 和 Master 分别访问不同的 Zookeeper 节点，整套系统就搭建完毕了。



4. 集群搭建

由于手头的设备有限，性能也有限（MacBook Pro 开一个虚拟机已经够呛了，开三个直接驾崩），因此最终选择在单机上的不同端口运行三个 Zookeeper 构成一个伪分布式状态。具体搭建方法如下：从官网下载 Zookeeper 的最新发布版本，分别解压到三个不同的目录，命名分别为 node0、node1、node2。在每一个目录的子目录 conf 下创建 zoo.cfg 文件，重点关注的内容是修改 dataDir 到指定目录，我的情况是在/Users/Shared 目录下创建了 zookeeper 目录并依次为三个节点创建子目录，并在子目录内创建一个名字为 myid 的文件用来让 zookeeper 知道自己对应配置文件中的哪一个节点，node0 的配置文件如下：

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/Users/Shared/zookeeper/node0
clientPort=2181
server.0=127.0.0.1:2888:3888
server.1=127.0.0.1:2887:3887
server.2=127.0.0.1:2889:3889
```

其中 server.x 的 x 就是 dataDir 中 myid 文件的值，clientPort 是应用程序连接到该节点的端口，server 信息中的第一个端口号是 Follower 连接到 Leader 用的端口号，第二个端口号是用来选举用的端口号，node1 的所有端口选择 node0 对应的端口号减去 1，node2 的所有端口选择 node0 对应的端口号加上 1。最后在各自的解压目录下的 bin 文件中运行 zkServer.sh 加上 start 参数启动集群。在从单节点扩展到集群后，Master 监听 node0，Storage 主节点连接到 node1，Storage 备份节点连接到 node2，经测试集群运作正常。

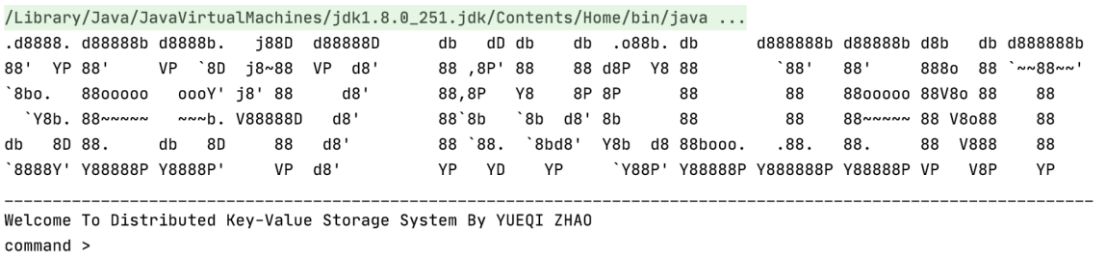
5. 具体实现

1) 数据结构

共有 3 个自定义数据结构，分别是 KeyValuePair，Message 和 Node。KeyValuePair 是对键值对的简单封装，可以方便我后续开发可以统一函数的参数；Message 是 Client 与 Master 之间的 RPC 通讯的返回类型，包含 MsgType 和 MsgContent，前者用于标记结果的属性，后者则是具体内容，比如类型为 SUCCEES 时表示操作成功了，此时可以无视内容，而类型为 ERROR 时，则可能出现了错误，Master 端的错误就可以通过内容传递到客户端作为输出让我看到，类型为 WARNING 时表示 PUT 操作试图覆盖一个 KEY 先前存的 VALUE，就可以通过内容让用户看到先前的 VALUE 是什么再决定是否要覆盖……；Node 是对 Storage 的 metadata 的封装，包含 Storage 节点的 IP 地址、端口号、别名、在 Zookeeper 集群中的 Path、是否为备份节点、是谁的备份、当前利用率等，用于 Master 对 Storage 节点进行管理，如替换挂掉的主节点、负载均衡等。

2) 客户端

客户端接收 2 个必要参数和 1 个可选参数，分别对应 Master 的 IP 地址和端口号以及是否开启 DEBUG 模式，在 DEBUG 模式下异常会被打印。在华丽的欢迎界面之后开始循环要求用户输入，华丽的界面如下：



支持的用户输入有：

a) PUT [KEY] [VALUE]
b) READ [KEY]
c) DELETE [KEY]
d) HELP
e) SHUTDOWN
f) EXIT

其中 PUT 操作在遇到当前 KEY 在系统中已有一个 VALUE 时会再次询问用户是否要更新 KEY 的值，用户输入 Y 时才会更新，否则放弃本次操作；HELP 操作会打印帮助信息；SHUTDOWN 操作会关闭 Master 和所有的 Storage 并退出 Client；EXIT 操作仅退出 Client，远程系统保持运行。所有其它的非法输入均会有相应的提示，并且所有的错误和异常都会以系统提示的方式告知用户，即所有的内部错误和异常都会被捕获而不让用户看到。

3) 服务端

服务端接收 2 个必要参数，分别对应 Zookeeper 节点的 IP 地址和端口号。启动时会有同样华丽的欢迎特效文字，此处不再放图片。服务端会先讲各个服务绑定到 RMI Registry，然后开始监听 Zookeeper 中 kvStorage 的状态。Master 的实现非常复杂此处仅挑选比较重要的几个详细描述。

a) 并发控制

Master 中所有重要代码段（一下简称临界区）都被锁保护，一共使用了 4 类锁，分别是 SystemLock，KeyLock，CacheLock 和 NodeLock，为了最大化性能，四者均用读写锁实现，因为在现实场景下并行读操作比写操作要频繁得多。SystemLock 为系统锁，在 Master 绑定服务和连接 Zookeeper 时会上锁，在关机时会上锁，避免系统在非就绪状态开始处理请求；KeyLock 为数据锁，也就是最核心的用户可以同时读一个 KEY 的 VALUE 但不能同时修改一个 KEY 的 VALUE；NodeLock 为节点信息锁，主要是考虑到可能有多个节点的信息同时变更，或者在进行调度的时候又有节点信息变更，为了保证每一次更新的原子性引入；CacheLock 为缓存锁，理由基本同上。

b) 对节点信息的监听

设置监听器监听 Zookeeper 中路径名为/kvStorage 的 Znode 的所有子节点，当有新节点加入或节点宕机时，子节点列表会发生改变，Zookeeper 会回调监听器的处理函数，处理函数用当前子节点列表和上一次回调的子节点列表进行比较，挑出新上线的节点和下线的节点，从 Zookeeper 中依次拉取新的节点的 Node 信息并设置监听器，并交由 Master 进一步处理。

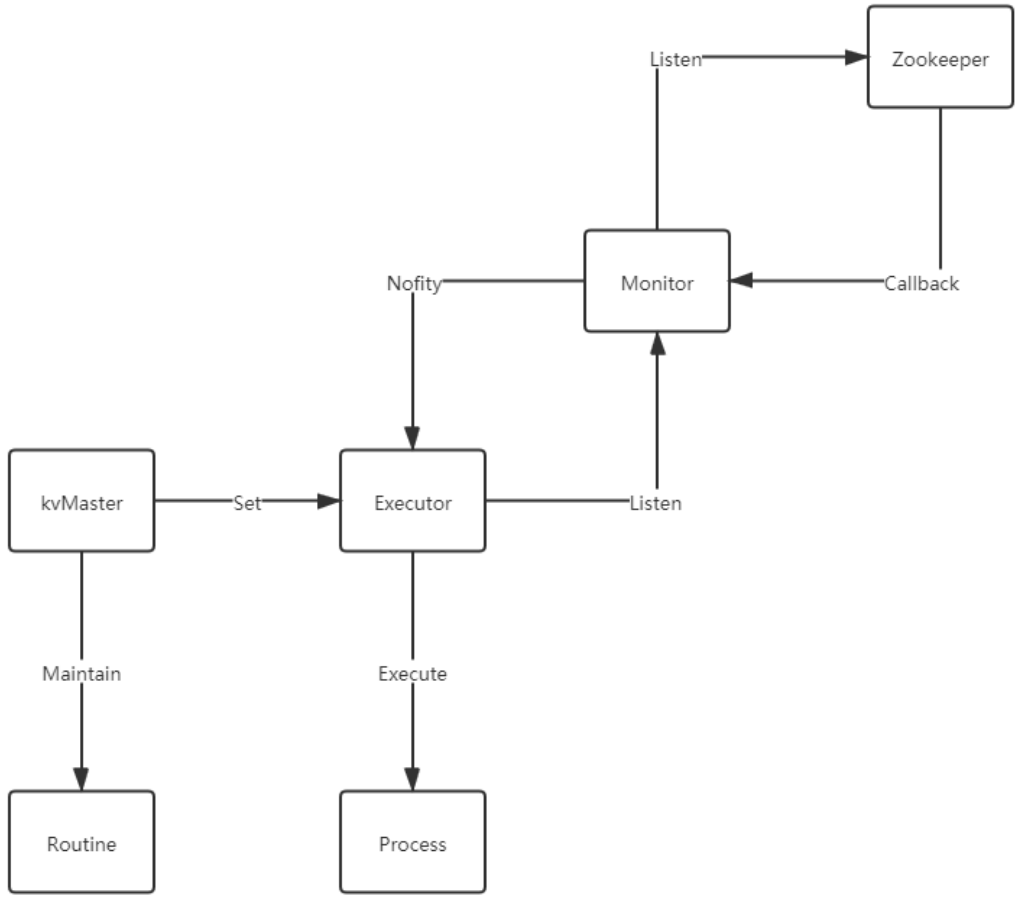
c) 对节点状态的监听

监听模式基本同上，只要是上线过的节点 Master 都会为其设定监视器，当节点信息更新的时候回调函数会把信息送回来，然后监听器交由 Master 处

理。对于下线了的节点，由于下线后其作为临时节点超时后会被 Zookeeper 清理，Master 中的监听器会注意到 Exist 回调状态变化，并设置 isDead 标记最后自我销毁。当节点信息或节点状态变化时 Master 输出如下：

```
nodeExecutor: <--------incoming change-------->
nodeExecutor: <<<<<<<< nodes to add <<<<<<<<<<
nodeExecutor: Storage-030000000007
nodeExecutor: >>>>>>>> nodes to remove >>>>>>>>
nodeExecutor: empty
nodeExecutor: <--------incoming change-------->
nodeExecutor: number of newly fetched nodes: 1
kvServer: updating node information...done
kvServer: current available nodes: 2
kvServer: current backup nodes: 2
dataExecutor: <--------incoming change-------->
dataExecutor: node 'Storage-03' updated its utilization to 0.0
dataExecutor: <--------incoming change-------->
kvServer: updating node information...done
```

监听器的设计框架如下：



d) 支持多节点后读写问题

在支持多节点后，除了 PUT 一个新的 KEY 以外别的都需要重新设计。对于 READ 操作，现在多了一个去哪 READ 的问题，一个不可避免的操作就是轮询，每一个节点都问一遍，直到找到为止，为了提升性能后续引入了缓存；对于 PUT 和 DELETE 操作，由于存在覆盖的情况，面临和 READ 一样的问题，解决方法也一样。简单概括就是靠轮询和缓存解决数据在哪里的问题，毕竟完整记录 KEY 和所属 Storage 是不合理的，这会让 Master 本身也变成 Storage。

e) 节点管理、可扩展性

Master 维护两个列表，一个 AvailableNode 表和 BackupNode 表，所有节点信息的变更最终都会作用在这两个表中，只有 AvailableNode 表是活跃的，即参与负载均衡调度以及轮询的节点仅来自 AvailableNode 表，BackupNode 表只有在“高可用”中一个节点挂掉需要替换的顶上才会用到。因此理论上可以支持任意多个主节点加入，由于设计上的一些问题，虽然一个主节点可以有多个备份节点加入，但容错只能主节点挂掉，替换它的备份节点再挂掉目前无法再调度另一个备份节点上来。

f) 负载均衡

每个节点都有一个 Capacity 属性，记录自己的虚拟最大容量，并根据存的 KEY 的总数计算占用率，Master 会以 1 秒 1 次的周期更新占用率最低的节点作为新 KEY 的目标节点，调度效果图如下。

```
dataExecutor: <--------incoming change-------->
dataExecutor: node 'Storage-02' updated its utilization to 0.1
dataExecutor: <--------incoming change-------->
kvServer: updating node information...done
kvServer: scheduling...done
kvServer: node 'Storage-01' is the default target now
```

g) 高可用

当监视器通知要删掉一个节点的时候，先在 AvailableNode 表中定位到要删除的节点，然后去 BackupNode 表中找有没有它的备份，有的话从 BackupNode 表中把备份节点移到 AvailableNode 中覆盖被删除节点，没有的话就地移除待删除节点，**并且清空与待删除节点有关的缓存信息（非常重要，否则下一次 READ 会触发依次 Internal Error**）。

h) 支持高可用后的读写问题

READ 操作的问题不大，主要是 PUT 和 DELETE，为了保证备份节点和主节点的数据一致性，对主节点的所有 PUT 和 DELETE 操作会对其备份节点同步执行。

i) 缓存

目前设计的缓存数量为 5，采用 LRU 的方式。初始为空，因此前 5 次 READ 操作都不得不轮询所有的 Storage，但轮询的结果会被保存在缓存中，下一次再访问这个 KEY 的 VALUE 时，在开始轮询前会先检索缓存中是否存在该 KEY，存在的话会直接返回 Storage 信息，避免了轮询。需要注意的是当主备份节点切换时，必须把缓存中与主节点相关的信息清楚，否则下一次 READ 会去访问一个已经下线的服务器，出现错误（但即使出现了这样的情况，我设置了在 READ 失败时从缓存中尝试删除本次的 KEY 与访问 Storage 的绑定，因此最差

情况用户也只会看到一次 Internal Errror，重试时将可以访问到正确的结果）。

4) 存储端

启动时将自己以临时 Znode 的方式注册进 Zookeeper 集群，确保自己下线时 Znode 会在超时后被删除从而通知到 Master 更新信息。Znode 的 Data 为自己的 Node 信息。每间隔 1 秒刷新一次自己的占用率，占用率精确到小数点后 1 位（可定制）。当占用率发生变化时修改自己的 Znode 的 Data 通知到 Master 更新信息。kvStorage 被设计为一个 Runnable 的类，并且所有信息都需要在构造函数中给出，因此可以快速创建一个新的节点并运行（2 行代码），来帮助实现可扩展性。

6. 总结

本次实验严格意义上说是念大学至今第一个仅给要求不给代码的非测试类实验，对于习惯了在助教给定的框架下发挥的我来说是一次巨大的挑战，系统架构的设计和开发流程的构思耗费了不少的精力，尤其当系统已经开发到具备一定规模之后突然发现后续的功能需求和前序的部分设计冲突而前序的设计已经被广泛地被应用在系统中，回去返工的体验着实让人崩溃。万幸的是我构思的开发流程和每个功能的实现方式基本都是正确的，并且在边开发边维护 TODO 的鼓励下，没有在某一个环节停滞，最终还是顺利开发出了整套系统。同时由于我使用的是 Zookeeper 的官方 API，因此在开发完之后对 Zookeeper 在用户侧的运行方式有了深刻的体会。由于时间限制到截止日期为止我还有很多想法没能实现，比如在可扩展性方面，我区分了 Storage 的主节点和备份节点，导致了一个在可扩展性上的瓶颈，虽然一个节点可以有多个备份节点，但是主节点挂掉后备份节点目前无法升级为主节点（因为它的别名和主节点不一致），因此当这个备份节点挂掉的时候，剩下的备份节点不会被调度上来替换它，我会在未来修正这个问题。不得不说本次实验虽然让人一度很痛苦，但学到的东西远超其它的实验，并且首次让我有强烈的渴望在 DDL 提交之后还要继续完善这个实验。