



**虚拟化与云计算**

**课程作业**

**题 目**  基于Docker的数据存储服务系统实现

**学 院**  计算机学院

**专业班级**  计算机科学与技术3班

**学 号**  3117004523

**姓 名**  李国延

**指导教师**  杨劲涛

**2020 年 6 月 30 日**

目录

[一 选题背景 1](#_Toc42122128)

[二 基础原理 1](#_Toc42122129)

[2.1 Docker原理 1](#_Toc42122130)

[2.2 存储服务系统原理 3](#_Toc42122131)

[三 系统设计 4](#_Toc42122132)

[3.1 API层设计 4](#_Toc42122133)

[3.2 引擎设计 5](#_Toc42122134)

[3.3 底层存储设计 5](#_Toc42122135)

[四 系统实现 6](#_Toc42122136)

[4.1 引擎实现 6](#_Toc42122137)

[4.2 整体实现 8](#_Toc42122138)

[4.3 测试分析 11](#_Toc42122139)

[4.4 Docker部署 13](#_Toc42122140)

[五 总结 15](#_Toc42122141)

# 一 选题背景

当下的科技发展已经处于一个云计算的时代，科技巨头都已经在云计算方面发展了些年，比如亚马逊云、谷歌云还是国内的阿里云。记得19年阿里双十一狂欢，对于淘宝系的服务，已经实现了全面上云，这也标志着云计算是成功的，在这个领域未来会有很强的生命力。

与云计算息息相关的具体技术，就是容器化，因为云计算的本质是通过网络，将分布的计算节点，构造成一张网，而具体的服务部署、运行，都是基于这张大网的，那么，就像我们以前发布一个应用一样，应用运行需要CPU、内存、磁盘资源，而容器化，就是将计算资源进行虚拟化的基础上，再封装成为容器，容器也是一个载体。

Docker就是容器化的一个典型的代表，Docker构造于操作系统之上，将资源虚拟化、容器化，而通过镜像，来运行容器，Docker就类似于一辆深海邮轮，承载着许多应用。

再观云计算发展，很主流的就是云存储，比如OSS、COS、OZONE等技术，这些云存储技术本来就是基于容器，以及存储服务来提供的。云存储服务可以说是云计算时代的一个很好的产物，因为云存储从一定程度上解决了工程中的许多问题，能提供容灾、CDN加速访问、调用链过滤等功能。这也从一定程度上激发了我对基于容器化实现存储服务的兴趣。

在本次课程设计中，个人选择了基于Docker的容器化来实现具备容灾的键值存储服务系统，从而加深自己对虚拟化与云计算这门课程的理解与掌握。

# 二 基础原理

## 2.1 Docker原理

LXC为Linux Container的简写。可以提供轻量级的虚拟化，以便隔离进程和资源，而且不需要提供指令解释机制以及全虚拟化的其他复杂性。相当于C++中的NameSpace。容器有效地将由单个操作系统管理的资源划分到孤立的组中，以更好地在孤立的组之间平衡有冲突的资源使用需求。LXC便是Docker的前身。

与传统虚拟化技术相比，它的优势在于：

（1）与宿主机使用同一个内核，性能损耗小；

（2）不需要指令级模拟；

（3）不需要即时(Just-in-time)编译；

（4）容器可以在CPU核心的本地运行指令，不需要任何专门的解释机制；

（5）避免了准虚拟化和系统调用替换中的复杂性；

（6）轻量级隔离，在隔离的同时还提供共享机制，以实现容器与宿主机的资源共享。

Linux Container提供了在单一可控主机节点上支持多个相互隔离的server container同时执行的机制。Linux Container有点像chroot，提供了一个拥有自己进程和网络空间的虚拟环境，但又有别于虚拟机，因为lxc是一种操作系统层次上的资源的虚拟化。

docker并不是LXC替代品，docker底层使用了LXC来实现，LXC将linux进程沙盒化，使得进程之间相互隔离，并且能够控制该进程的资源。

docker是一个开源的应用容器引擎，基于go语言开发并遵循了apache2.0协议开源。

docker可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的linux服务器，也可以实现虚拟化。

容器是完全使用沙箱机制，相互之间不会有任何接口（类iphone的app），并且容器开销极其低。

讲讲Docker中的核心概念：镜像和容器、容器和虚拟机。

docker镜像就是一个只读模板，比如，一个镜像可以包含一个完整的centos，里面仅安装apache或用户的其他应用，镜像可以用来创建docker容器，另外docker提供了一个很简单的机制来创建镜像或者更新现有的镜像，用户甚至可以直接从其他人那里下周一个已经做好的镜像来直接使用。

docker利用容器来运行应用，容器是从镜像创建的运行实例，它可以被启动，开始、停止、删除、每个容器都是互相隔离的，保证安全的平台，可以吧容器看做是要给简易版的linux环境（包括root用户权限、镜像空间、用户空间和网络空间等）和运行在其中的应用程序。

仓库是集中存储镜像文件的沧桑，registry是仓库主从服务器，实际上参考注册服务器上存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签（tag）

仓库分为两种，公有参考，和私有仓库，最大的公开仓库是docker Hub，存放了数量庞大的镜像供用户下周，国内的docker pool，这里仓库的概念与Git类似，registry可以理解为github这样的托管服务。

那么Docker解决了什么问题呢？

发布服务不用担心服务器的运行环境，所有的服务器都是自动分配docker，自动部署，自动安装，自动运行。不用担心其他服务引擎的磁盘问题，cpu问题，系统问题了。应用运行于沙盒，对底层系统侵入性小，迁移方便，可以制作镜像，迁移使用自定义的镜像即可迁移。

当然，Docker也并非是完美：如在资源开销上，Docker由于宿主机上的所有容器是共享相同的内核和相同的资源，如果对某些资源（CPU、内存、磁盘等）的访问不受限制，那么异常的容器将占用整个宿主机的资源，从而影响其他容器的运行，影响应用程序。套接字问题，容器在默认情况下都安装了docker Unix套接字（/var/run/docker.sock），此套接字，可以关闭、启动或者创建新的镜像。当你的容器启动并共享套接字的时候，你就给了容器操控宿主机的权限，它将可以启动或终止其它容器，在宿主机拖入或创建镜像，甚至写入到宿主机的文件系统。正确配置和保护，可以使用docker容器实现高级别的安全性，但它的安全性还是低于正确配置的VM。内核漏洞，Docker内核攻击对于容器化环境来说可能是致命性的，因为容器与主机共享相同的系统内核，因此单独信任容器内置保护机制是不够的。

容器的隔离性使得某个应用程序的漏洞不会直接影响到其他容器的应用程序，但是漏洞可能会破坏与其他容器所共享的单一的操作系统，进而影响机器上的其他容器。如果漏洞允许代码执行，那么它将在主机操作系统上执行，而不是在容器内执行；如果此漏洞允许任意内存访问，则攻击者可以更改或读取任何其他容器的任何数据。最后是Docker做企业集成的问题，Docker做持续性交付的企业最能体会到这种苦恼，Ghosh说这对于企业来说可能会更加复杂，而且还会有不同的工作负载、各种各样的应用堆栈，异构的基础设施和有限的资源，更不用说IT企业需要的可视化、可控制和安全性。

因此，Docker并非十全十美，当然，本身算是瑕不掩瑜，比如缺少的编排功能，现在已经有Google的K8S支持，虽然Docker Swarn也在尝试进行编排工作。但是，仍然需要肯定Docker的贡献。

## 2.2 存储服务系统原理

存储服务系统具备如下特点：简易的API访问、快速响应、具备容灾。

简易API访问，作为存储服务，这里不像MySQL等数据库，而更像是Redis之类的键值对数据库，提供k-v的增删查改操作即可，k是字符串类型，而v可以是复杂数据结构也可以是单一基础数据类型。

对于快速响应特性，则存储服务系统本身支持多节点部署，因为单节点的负载、运算能力有限，可以通过多节点实现负载均衡，提高响应速度，同时，从实现层面上来看，应当选择高效的编程语言，如C++、GO之类。

具备容灾，由于该存储服务基于容器化部署，因此，运行环境上支持容灾是相对容易的，而从设计实现上，必须支持多节点发布，如副本机制、主从机制、分布式机制（满足一致性）等。在该系统中，个人选择了相对易于实现的副本机制，来保证一定的容灾能力。

# 三 系统设计

## 3.1 API层设计

API层分别设计如下三个接口，针对K-V操作

|  |  |
| --- | --- |
| 接口 | 操 作 描 述 |
| Get | 通过键获取对应的值，如果不存在，则进行相应的提示；否则返回存在的数据。该接口无需做replication操作，从存储引擎层，要解决并发读取的线程安全问题。 |
| Put | 通过键值将数据写入存储层，如果不存在，则写入，提示写入成功信息，否则进行数据覆盖，该操作需要做replication处理，同时需要与Get、Del操作进行同步，保证数据的一致性。 |
| Del | 通过键删除对应的值，如果不存在，则进行相应的提示；否则返回删除成功的提示。该接口需做replication操作，从存储引擎层，要解决并发读取的线程安全问题。 |

## 3.2 引擎设计

锁同步引擎与CopyOnWrite存储引擎：两者都是为了解决并发的问题，在这里，锁同步更加均衡读写性能，而COW思想的存储引擎，更加注重读的无锁化处理，该存储服务系统内置两种存储引擎实现，可由用户自行选择合适情景下的存储引擎。

这里主要讲讲COW思想的存储引擎设计：和单词描述的一样，他的实现就是写时复制， 在往集合中添加数据的时候，先拷贝存储的数组，然后添加元素到拷贝好的数组中，然后用现在的数组去替换成员变量的数组（就是get等读取操作读取的数组）。这个机制和读写锁是一样的，但是比读写锁有改进的地方，那就是读取的时候可以写入的 ，这样省去了读写之间的竞争，看了这个过程，你也发现了问题，同时写入的时候怎么办呢，当然果断还是加锁。 这体现了读写分离的思想：在写操作的线程，会将数组复制出来一份进行操作。而原本的数组不会做改变。读线程则不会受到影响，但是可能读到的是一个过期的数据。只能保证最终的一致性，不能保证实时的一致性。但是从根本上，能够提高无锁化读以及数据存储服务系统所能提供的吞吐量。

## 3.3 底层存储设计

该存储系统服务当前版本只支持内存存储，采用哈希数据结构进行键值对存储，后续版本会添加持久化数据支持。

# 四 系统实现

## 4.1 引擎实现

基于golang语言进行实现

|  |
| --- |
| //define kv storage engine  //This version does not support data persistence operations  //The next version will do consistent synchronization based on raft algorithm on data persistence  type StorageEngine struct {  // Package sync provides basic synchronization primitives such as mutual  // exclusion locks. Other than the Once and WaitGroup types, most are intended  // for use by low-level library routines. Higher-level synchronization is  // better done via channels and communication.  //  // Values containing the types defined in this package should not be copied.  sync.Mutex //Ensure thread safety  Get func(key string) (interface{}, error) //get operation  Put func(key string, val interface{}) error //put operation  Del func(key string) bool //delete operating  }  //Use cow thinking to keep safe  //Read and write will not conflict  //Best read performance  var CopyOnWriteEngine StorageEngine  //Storage engine using synchronizer  //Balanced performance  var SyncEngine StorageEngine  //init  func init() {  KV = make(map[string]interface{})  //init cow engine  CopyOnWriteEngine = StorageEngine{  Mutex: sync.Mutex{},  //The get operation only needs to be unlocked  Get: func(key string) (interface{}, error) {  if v, ok := KV[key]; ok {  return v, nil  }  return nil, errors.New("fail")  },  Del: func(key string) bool {  delete(KV, key)  return true  },  }  CopyOnWriteEngine.Put = func(key string, val interface{}) error {  CopyOnWriteEngine.Lock()  defer CopyOnWriteEngine.Unlock()  KV = Copy(KV)  KV[key] = val  return nil  }  //sync engine  SyncEngine = StorageEngine{  Mutex: sync.Mutex{},  }  SyncEngine.Get = func(key string) (interface{}, error) {  SyncEngine.Lock()  defer SyncEngine.Unlock()  if v, ok := KV[key]; ok {  return v, nil  }  return nil, errors.New("fail")  }  SyncEngine.Put = func(key string, val interface{}) error {  SyncEngine.Lock()  defer SyncEngine.Unlock()  KV[key] = val  return nil  }  SyncEngine.Del = func(key string) bool {  SyncEngine.Lock()  defer SyncEngine.Unlock()  delete(KV, key)  return true  }  } |

## 4.2 整体实现

|  |
| --- |
| package main  import (  "errors"  "fmt"  "net/http"  "os"  "sync"  )  /\*\*  \* @student no 3117004523  \* @author liguoyan  \* @date 2020/6/26  \*/  //the global KV storage  var KV map[string]interface{}  //define kv storage engine  //This version does not support data persistence operations  //The next version will do consistent synchronization based on raft algorithm on data persistence  type StorageEngine struct {  sync.Mutex //Ensure thread safety  Get func(key string) (interface{}, error) //get operation  Put func(key string, val interface{}) error //put operation  Del func(key string) bool //delete operating  }  //Use cow thinking to keep safe  //Read and write will not conflict  //Best read performance  var CopyOnWriteEngine StorageEngine  //Storage engine using synchronizer  //Balanced performance  var SyncEngine StorageEngine  //config  var Config ClusterConfig  //init  func init() {  KV = make(map[string]interface{})  //init cow engine  CopyOnWriteEngine = StorageEngine{  Mutex: sync.Mutex{},  //The get operation only needs to be unlocked  Get: func(key string) (interface{}, error) {  if v, ok := KV[key]; ok {  return v, nil  }  return nil, errors.New("fail")  },  Del: func(key string) bool {  delete(KV, key)  return true  },  }  CopyOnWriteEngine.Put = func(key string, val interface{}) error {  CopyOnWriteEngine.Lock()  defer CopyOnWriteEngine.Unlock()  KV = Copy(KV)  KV[key] = val  return nil  }  //sync engine  SyncEngine = StorageEngine{  Mutex: sync.Mutex{},  }  SyncEngine.Get = func(key string) (interface{}, error) {  SyncEngine.Lock()  defer SyncEngine.Unlock()  if v, ok := KV[key]; ok {  return v, nil  }  return nil, errors.New("fail")  }  SyncEngine.Put = func(key string, val interface{}) error {  SyncEngine.Lock()  defer SyncEngine.Unlock()  KV[key] = val  return nil  }  SyncEngine.Del = func(key string) bool {  SyncEngine.Lock()  defer SyncEngine.Unlock()  delete(KV, key)  return true  }  }  //main function  func main() {  if len(os.Args) < 3 {  fmt.Println("Usage : go run xx <master>{:port} <slave>{:port}")  return  }  //config by args  Config = ClusterConfig{  master: os.Args[1],  slave: os.Args[2],  }  //Register route  mux := http.DefaultServeMux  mux.HandleFunc("/get", HandlerGet)  mux.HandleFunc("/put", HandlerPut)  mux.HandleFunc("/del", HandlerDel)  //Open service  \_ = http.ListenAndServe(Config.master, mux)  }  //Get without Disaster recovery  func HandlerGet(w http.ResponseWriter, r \*http.Request) {  key := r.FormValue("k")  v, err := SyncEngine.Get(key)  if err != nil {  w.Write([]byte("key not exist"))  return  }  //autonomy  \_, \_ = w.Write([]byte(v.(string)))  }  //KV request processor with master-slave disaster recovery  func HandlerPut(w http.ResponseWriter, r \*http.Request) {  key := r.FormValue("k")  val := r.FormValue("v")  //replica  \_ = SyncEngine.Put(key, val)  //容灾  \_, \_ = http.Get(Config.slave + "/put?k=" + key + "&v=" + val)  \_, \_ = w.Write([]byte("ok"))  }  //KV request processor with master-slave disaster recovery  func HandlerDel(w http.ResponseWriter, r \*http.Request) {  key := r.FormValue("k")  SyncEngine.Del(key)  http.Get(Config.slave + "/del?k=" + key)  \_, \_ = w.Write([]byte("ok"))  }  //Shallow clone  func Copy(value map[string]interface{}) map[string]interface{} {  newMap := make(map[string]interface{})  for k, v := range value {  newMap[k] = v  }  return newMap  }  //Cluster configuration  type ClusterConfig struct {  master string  slave string  //Now only support a master with a slave.  //Both are masters themselves, the other is slave  } |

## 4.3 测试分析

命令行执行：（因为需要测试容灾，运行两个节点）

go run main.go :6789 :5961

go run main.go :5961 :6789

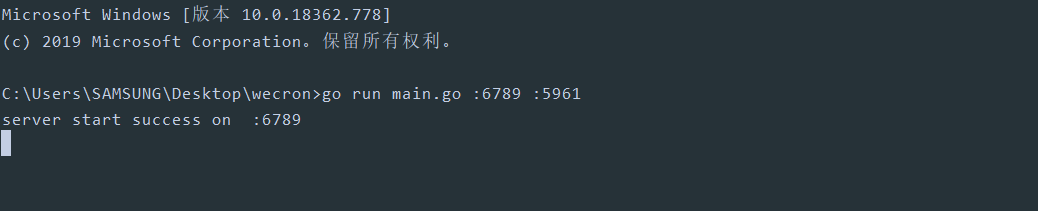


图 1 在6789运行

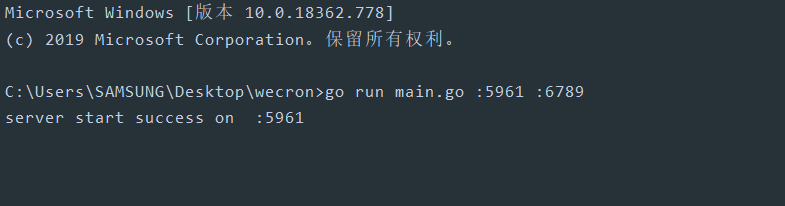


图 2 在5961运行

运行成功，先进行Put操作：



图 3 进行put操作

在6789端口节点执行put ligy1 3117004523操作，为了验证数据副本机制，从另外节点执行get ligy1操作

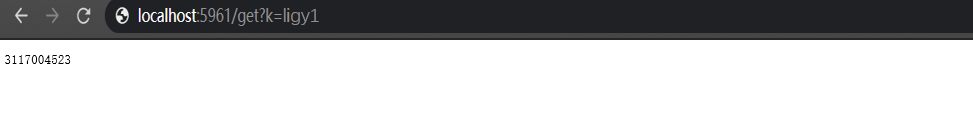


图 4 执行get ligy1操作验证

执行del操作，则两个节点都会删除数据：



图 5 执行del操作



图 6 再次get检验

以上是基础的两个节点的数据同步操作，采用了同步锁存储引擎，如果需要拓展多节点，只需要配置一下集群配置信息即可。

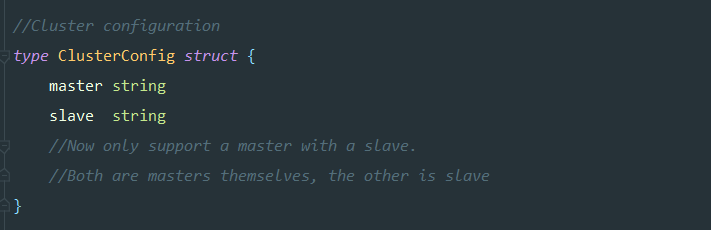


图 7 cluster配置

## 4.4 Docker部署

基于Docker部署数据存储服务，步骤如下：

* 构建具备go语言环境的容器
* 编译构建存储服务
* 启动容器并配置路由映射
* 打包容器为镜像，用于迁移与扩容操作

首先为docker引擎分配如下资源：

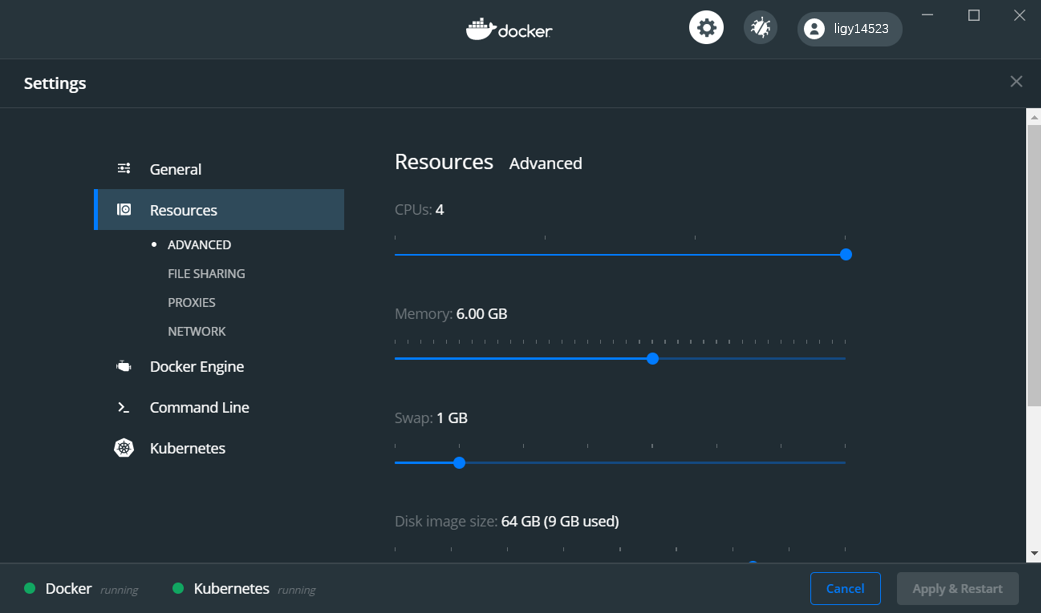


图 8 为docker分配资源

在Linux的AMD64架构上编译存储服务：



图 9 编译存储服务

测试运行：（以下结果证明编译成功）



图 10 测试结果

编写docker file

|  |
| --- |
| FROM golang:latest  MAINTAINER lgy14523 <1642801201@qq.com>  ADD main /  EXPOSE 8991 8992  ENTRYPOINT ["./main"] #入口 |

根据镜像启动存储服务：



图 11 启动存储服务

查看启动情况：



图 12 启动情况

通过网页访问存储服务：

在8991节点存储一个KV

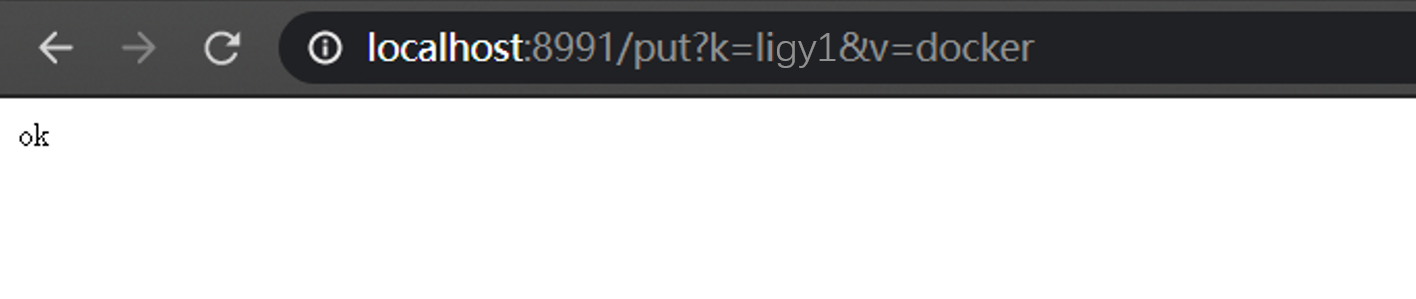


图 13 存储KV

在8992节点尝试获取KV，测试主从双写

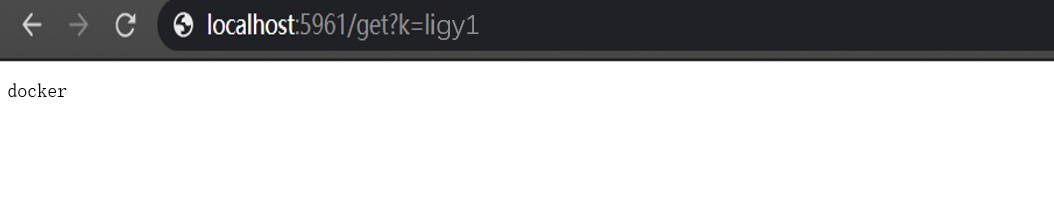


图 14 尝试获取KV

以上过程基于docker单节点容器部署，使用两个端口模拟主从节点，当然，如果加上K8S编排工具，且docker具备多个节点，就能够实现真正的多机器容灾。

# 五 总结

在本次课程设计中，选择了基于Docker的存储服务系统实现，分别是Docker的实践以及存储服务的设计与实现，在Docker实践部分，学习了Docker的原理以及如何使用Docker部署服务，且具备双写容灾，这个过程加深了对《虚拟化与云计算》理论知识的理解，也增强了对Docker容器化的知识的理解。

再者，通过设计主从双写的KV存储服务系统，了解到了更多关于云存储的知识，增强了对《虚拟化与云计算》这门课程的兴趣，也锻炼了动手编程能力。

在这个过程中遇到一些问题，通过查找资料，阅读老师课程讲的内容，都很好地进行解决，在本次课程设计以及课程学习中，特别感谢老师的指导，无论是理论知识、课后作业、课程实验、课程设计，老师都给予了指导，在老师指导下，最终完成了课程的设计。