DOCKER

官网地址⟶docker

tips：笔记大多数为命令的执行

一、安装

此次安装基于 `CentOS7` 版本 安装文档 ⟶ https://docs.docker.com/engine/install/centos/

1）卸载旧程序

```
 sudo yum remove docker \
                  docker-client \
                  docker-client-latest \
                  docker-common \
                  docker-latest \
                  docker-latest-logrotate \
                  docker-logrotate \
                  docker-engine
```

2）环境搭建

```
# docker编译依赖于gcc
yum -y install gcc

# yum-utils
sudo yum install -y yum-utils

# 由于官网提供的仓库地址是国外网络...采用阿里云
# 设置docker镜像源
yum-config-manager \
    --add-repo \
    https://mirrors.aliyun.com/docker-
ce/linux/centos/docker-ce.repo
# 上一条命令的第二步
sed -i
's/download.docker.com/mirrors.aliyun.com\/docker-
ce/g' /etc/yum.repos.d/docker-ce.repo

# 可忽略 更新yum索引软件包 变的快一点
yum makecache fast
```

3）依照官网执行

```
 # 补充 `sudo`root用户执行的声明
 sudo yum install docker-ce docker-ce-cli
containerd.io
```

4）启动测试

```
systemctl start docker # 开启docker服务
```

```
 sudo docker run hello-world
 # 执行命令出现以下表示安装成功

 Hello from Docker!
 This message shows that your installation appears
to be working correctly.

 docker --version
 Docker version 20.10.12, build e91ed57
```

5） 阿里云镜像加速

来源 黑马 自己可以申请阿里云个人实例镜像加速

官网⟶ https://cr.console.aliyun.com/cn-hangzhou/instances/mirrors

```
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<-'EOF'
{
  "registry-mirrors":
["https://uc51lue1.mirror.aliyuncs.com"]
}
EOF
sudo systemctl daemon-reload
sudo systemctl restart docker
```

6）相应卸载命令

```
# systemctl stop docker
# yum remove docker-ce docker-ce-cli containerd.io
# rm -rf /var/lib/docker
# rm -rf /var/lib/containerd
```

二、docker常用命令

1） run干了什么?

```
docker run (images) ->{
    if(本地存在(images)){
        // 直接以该镜像作为模板容器实例运行
        images.run();
        return;
    }
    // 本地没有 就去dockerHub找
    Images img = DockerHub.get(images.getName())
    if(img != null){
        // 下载到本地运行
```

```
            images.run();
            return;
        }
        throw new Exception("无法找到该镜像，执行失
败！！！");
    }
```

2）常用命令

帮助类

```
systemctl stop docker  # 停止docker服务

systemctl restart docker  # 重启docker服务

systemctl status docker # 查看状态 active (running)代
表运行

systemctl enable docker # 配置开机启动

docker info # 查看摘要信息

docker --help # 查看帮助文档

docker 命令 --help # 查看精确命令的帮助文档
```

镜像命令

具体命令

--附加参数功能

1. `docker images` 直接运行查看所有镜像
    1. `-a` 列出所有本地镜像
    2. `-q` 只显示镜像id

`docker images` 选项说明

```
REPOSITORY     TAG         IMAGE ID        CREATED
    SIZE

镜像源          标签版本号     镜像id          创建时间
占用大小
```

2. `docker search` 查看远程库是否存在某个镜像
    1. `docker search --limit 5 镜像名` 分页

```
NAME          DESCRIPTION       STARS      OFFICIAL
AUTOMATED
名称           说明               点赞数       是否官方
是否自动构建
```

3. `docker pull 镜像名:tag` 下载镜像 tag版本号
    1. 不写tag默认 `latest` 也就是最新版
4. `docker system df` 查看镜像/容器/数据卷占用空间 `df -h`
5. `docker rmi 镜像名` 删除镜像
    1. `docker rmi -f image:tag` 删除单个
    2. `docker rmi -f image:tag image:tag...` 删除多个
    3. `docker rmi -f $(docker images -qa)` 删除全部

docker的虚悬镜像是什么?

仓库名，镜像名都是的镜像 `dangling` image

```
REPOSITORY     TAG         IMAGE ID        CREATED
SIZE

none           none       fsfdsdfds        ....
12232MB
```

容器命令

- `docker run`
  - `--name` 为容器起一个名称
  - `-d` 后台守护进程运行
  - `-i` 以交互模式运行 与 `-t` 一起使用
  - `-t` 为容器分配一个伪终端使用
  - `-P` 随件端口映射
  - `-p` 指定端口映射

```
docker run -it ubuntu:latest bash
# 交互式运行ubuntu容器 需要交互式 bash shell脚本
# 输入exit退出
```

- `docker ps` 查看当前运行的容器
  - `-a` 包括没运行的
  - `-l` 最近创建de
  - `-n 1` 最近创建过的1个
  - `-q` 静默模式 只显示容器编号
- 退出容器内部的命令
  - `exit` 退出结束容器停止
  - `ctrl+p+q` 退出容器不停止

```
# 不停止可以直接
docker exec -it ub bash # 再次进入容器
```

- `docker start 容器名/id` 重新启动
- `docker rm 容器名/id` 删除容器
  - `-f` 硬删
  - `docker rm -f $(docker ps -a -q)` 删除一堆
  - `docker ps -a -q | xargs docker rm -f` 同上

问题：

执行`docker run -d ubuntu` 以后台模式运行某个容器

`docker ps -a` 查看容器已经退出

注意点：docker容器后台运行，必须要有一个前台进程，容器运行的命令如果不是一直挂起的那一种（top,yail），就会自动退出

比如nginx这种web容器，配置服务只需要启动响应的额service ⟶ service nginx start,但是导致前台没有运行的应用，就会认为无事可做，自杀

解决方案：将要运行的容器以前台进程的形式运行，也就是命令行模式，表示还有交互，别中断

`docker run -it redis:latest` 前台运行

`docker run --name rs -p 6379:6379 -d redis` 守护进程启动 指定端口6379

- `docker logs 容器名/id` 查看容器日志
- `docker top 容器名/id` 查看容器内进程
- `docker exec` 进入容器
  - `-it` 交互伪终端
  - `bash/容器特有命令` 执行相关命令

```
# 开启一个容器 后台运行
docker run --name rs -p 6379:6379 -d redis

# 直接进入容器内部
docker exec -it redis bash
# 执行redis客户端
redis-cli

# 或者直接执行客户端
docker exec -it redis redis-cli
```

- 重新进入容器 `docker attach 容器id`

与exec的区别是?

> atach直接进入容器启动命令的终端，不会启动新的进程 exit会导致直接停止容器
>
> exec 是在容器中打开新的终端，可以启动新的进程，exit不会导致容器的停止

- `docker cp 容器名称/id 容器文件路径  主机路径` 拷贝容器重要文件

```
docker cp redis:/data/dump.rdb /opt
```

镜像的导入与导出

- docker save image:atg -o 文件路径 导出
- docker load docker load -i 文件路径 加载镜像

```
# 导出
docker save redis:latest -o /opt/redis.tar
docker save redis:latest > /opt/redis.tar

# 加载
docker load -i /opt/redis.tar
```

容器的导入导出

- docker export 容器名/id > 文件路径
- cat 文件.tar | docker import - 包名/容器名:版本号

```
# 导出
docker export redis > aa.tar

# 导入
cat aa.tar | docker import - wjl/redis:666
```

三、 docker镜像

所谓的镜像是一种轻量级，可执行的软件包，它包含运行某个软件所需的所有内容，把应用程序和配置依赖打包形成一个可交互的运行环境，这个打包好的运行环境就是image镜像文件

docker run ≈ new Java()

容器镜像层都是只读的，容器是可写的

> 当容器启动时，一个新的可写层被加载到镜像的顶部。这一层通常被称为“容器层”，容器层之下都叫“镜像层”。
>
> 所有对容器的改动-无论添加、删除、还是修改文件都只会发生在容器中。只有容器是可写的，容器层下面的镜像层都是只读的

1）commit命令

案例⟶ 让原生Ubuntu镜像安装vim

docker run -it --name ub ubuntu 进入容器

```
apt-get update # 更新包管理工具 `apt-get`类似与CentOS的yum

apt-get -y install vim # 安装vim

exit
```

执行commit命令

```
 docker commit -m="add vim cmd" -a="wjl"
d036ca38c53b ubuntu-vim:1.0

 -m 描述信息
 -a 作者
 d036ca38c53b 刚刚安装vim的Ubuntu
 ubuntu-vim:1.0  容器名:版本号
```

执行docker images

```
ubuntu-vim   1.0       de9852d7a6f6   About a
minute ago   174MB
```

提交阿里云

步骤

登录阿里云——> 控制台——>容器镜像服务——>个人实例——>命名空间——>创建镜像仓库选择自己的命名空间——>添加描述——>创建本地仓库

此时阿里云会生成相关命令 直接cv

2）创建并提交私有库

```
docker pull registry

# 映射5000端口
docker run --name reg -v /usr/wjl:/tmp/myregistry
 -p 5000:5000 -d --privileged=true  registry:latest

# 找一个容器做点改动 执行commit命令
 docker commit -m="add vim cmd" -a="wjl"
d036ca38c53b nginx_biaoji:1.0

# 验证本地私服库是否包含镜像
 curl -XGET http://192.168.0.103:5000/v2/_catalog

# 提交
docker tag nginx_biaoji:1.0
192.168.0.103:5000/nginx_biaoji:1.0

# 执行docker images 所出现的就是要推送的镜像
192.168.0.103:5000/nginx_biaoji

# 推送私有库下面有详细的配置
```

```
# 推送私有库
 docker push 192.168.0.103:5000/nginx_biaoji:1.0

# 验证是否成功
curl -XGET http://192.168.0.103:5000/v2/_catalog


# 拉取镜像
docker pull 192.168.0.103:5000/nginx_biaoji:1.0
```

由于docker不支持http发送 ，所以采用配置

```
vim /etc/docker/daemon.json
# 注意ip
{
"registry-mirrors":
["https://uc51lue1.mirror.aliyuncs.com"],
"insecure-registries":["192.168.0.103:5000"]
}
```

配置完重启

四、挂载数据卷

CentOS7安全模块比之前系统版本强，不安全的会先禁止，所以目录挂载的情况默认为不安全的行为，在SELinux里挂载的目录被禁止掉了，如果要开启，一般使用 `--privileged=true` 命令，扩大容器的权限解决挂载目录没有权限的问题，否则，container内的root只是外部的一个普通用户权限。

所谓的数据卷就类似于活动硬盘，容器死了挂载数据卷的文件夹还会存放数据

语法

`docker run -it --privileged=true -v 宿主机目录(绝对路径):/容器目录 image:tag`

```
docker volume inspect 容器id # 查看挂载信息
```

以上的配置是默认加了 `rw` 表示可读可写

要想限制容器只读可以在卷参数后面加上 `:ro`

ro: read-only

数据卷的继承

`docker run -it --volumes-from u1  --name u2 --privileged=true ubuntu bash`

--volumes-from u1 参数表示继承u1的数据卷

五、常用软件运行示例

1) tomcat

无需下载最新版 docker pull billygoo/tomcat8-jdk8

```
docker run --name tom -d -p 8080:8080 billygoo/tomcat8-jdk8
```

Tips:注意最新版需要将webapps.dict更改为webapps

2) Mysql

dcker pull mysql:latest

```
docker run --name mysql -e
MYSQL_ROOT_PASSWORD=123456 -d -p 3306:3306
mysql:latest
# -e 指定环境变量
# MYSQL_ROOT_PASSWORD MySQL的密码
```

create database if not exists mysqldb character set utf8;

连接Navicat连接测试成功

但是mysql很重要 需要数据卷

```
 docker run --name mysql
 -d -p 3306:3306
 --privileged=true
 -v /tmp/mysql/log:/var/log/mysql
 -v /tmp/mysql/data:var/lib/mysql
 -v /tmp/mysql/conf:/etc/mysql/conf.d
 -e MYSQL_ROOT_PASSWORD=123456
 mysql:latest
```

在/tmp/mysql/conf目录下创建 `my.cnf` 文件

```
[client]
default_character_set=utf8
[mysqld]
collation_server = utf8_general_ci
character_set_server = utf8
```

docker restart mysql

```
mysql> select * from vip;
+----+------+
| id | name |
+----+------+
|  1 | 王   |
|  2 | 李   |
+----+------+
2 rows in set (0.00 sec)
```

3) redis

创建redis/conf 目录 找一份redis.conf拷贝至此目录

1. 后台启动 daemonize no
2. 允许远程访问 bind 0.0.0.0
3. 可以选择数据持久化 appen
4. 保护模式 protected-mode no

```
docker run -p 6379:6379
--name redis --privileged=true
-v /opt/redis/conf/redis.conf:/etc/redis/redis.conf
-v /opt/redis/data:/data
-d redis:6.0.8
redis-server /etc/redis/redis.conf
```

```
docker exec -it redis redis-cli
```

4) MySql主从复制

主机

粘贴以下内容：注意 两个Mysql进程可能起不来

```
5.7
```

```
docker run -p 3307:3306 --name mysql-master -v
/tmp/mysql2/mysql-master/log:/var/log/mysql -v
/tmp/mysql2/mysql-master/data:/var/lib/mysql -v
/tmp/mysql2/mysql-master/conf:/etc/mysql  -e
MYSQL_ROOT_PASSWORD=123456 -d mysql:latest
```

8.0

```
docker run -p 3307:3306 --name mysql-master -v
/tmp/mysql2/mysql-master/log:/var/log/mysql -v
/tmp/mysql2/mysql-master/data:/var/lib/mysql -v
/tmp/mysql2/mysql-master/conf:/etc/mysql  -v
/tmp/mysql2/mysql-master/mysql-
files:/var/lib/mysql-files  -e
MYSQL_ROOT_PASSWORD=123456 -d mysql:latest
```

在 `/tmp/mysql2/mysql-master/conf/` 下创建 `my.cnf`

```
[mysqld]
## 设置server_id 同一区域网唯一
server_id=101
## 执行不需要同步的数据库名称
binlog-ignore-db=mysql
## 开启二进制日志
log-bin=mall-mysql-bin
## 设置二进制日志使用内存大小（事务）
binlog_cache_size=1M
## 设置二进制格式‘
binlog_format=mixed
## 二进制日志过期时间清理
expire_logs_days=7
## 跳过主从复制终于到的所有错误避免slave复制中断 1062主键重
复
slave_skip_errors=1062
```

https://blog.csdn.net/qq_40604437/article/details/106680762 8.0错误问题解决

> 注意：以下语法有很多在8.0中过时 详细信息查看日志
>
> `docker logs -f imageName/id`

```
[mysqld]
server_id=101
binlog-ignore-db=mysql
log-bin=mall-mysql-bin
binlog_cache_size=1M
binlog_format=mixed
expire_logs_days=7
slave_skip_errors=1062


secure_file_priv=/var/lib/mysql
```

如果报错

```
ERROR 1045 (28000): Access denied for user
'root'@'localhost' (using password: YES)
root@782942b739bb:/# mysql -u root -p
```

加上这一行

```
skip-grant-tables
```

进入容器登录MySQL

```
# 创建一个从机用户
CREATE USER 'slave'@'%' IDENTIFIED BY '123456';
# 授权
GRANT REPLICATION SLAVE,REPLICATION CLIENT ON *.*
TO'slave'@'%';
```

从机

5.7

```
docker run -p 3307:3306 --name mysql-master -v
/tmp/mysql2/mysql-master/log:/var/log/mysql -v
/tmp/mysql2/mysql-master/data:/var/lib/mysql -v
/tmp/mysql2/mysql-master/conf:/etc/mysql  -e
MYSQL_ROOT_PASSWORD=123456 -d mysql:latest
```

8.0

```
docker run -p 3308:3306 --name mysql-slave -v
/tmp/mysql2/mysql-master/log:/var/log/mysql -v
/tmp/mysql2/mysql-slave/data:/var/lib/mysql -v
/tmp/mysql2/mysql-slave/conf:/etc/mysql  -v
/tmp/mysql2/mysql-slave/mysql-files:/var/lib/mysql-
files  -e MYSQL_ROOT_PASSWORD=123456 -d
mysql:latest
```

```
[mysqld]
server_id=102
binlog-ignore-db=mysql
log-bin=mall-mysql-bin
binlog_cache_size=1M
binlog_format=mixed
expire_logs_days=7
slave_skip_errors=1062

log_slave_updates=1

##设置为只读 有surper权限的除外

read_only=1

secure_file_priv=/var/lib/mysql
```

主服务器查看状态

```
mysql> show master status;
+------------------------+----------+--------------
+------------------+-------------------+
| File                   | Position | Binlog_Do_DB |
Binlog_Ignore_DB | Executed_Gtid_Set |
+------------------------+----------+--------------
+------------------+-------------------+
| mall-mysql-bin.000001 |      713 |              |
mysql             |                   |
+------------------------+----------+--------------
+------------------+-------------------+
1 row in set (0.00 sec)
```

进入从服务器

```
[root@192 ~]# docker exec -it mysql-slave bash
root@c7ac51046759:/# mysql -u root -p123456
```

执行认主

```
change master to
master_host='192.168.0.102',master_user='slave',mas
ter_password='123456',master_port=3307,master_log_f
ile='mall-mysql-bin.000001',master_log_pos=
713,master_connect_retry=30;
```

参数说明

```
master_host 主数据库ip
master_port 主数据库端口
master_user 之前在主数据库创建用于同步数据的账号
master_password 密码
master_log_file 指定从主数据库复制的日志文件，查看主数据的状态 获取File参数
master_log_pos 指定从主数据库哪个地方开始复制数据，查看主数据的状态 获取Position参数
# 就是之前在主数据库获取的  Position ↑
master_connect_retry 连接失败重试的事件间隔 秒为单位
```

在从机执行

```
show slave status\G;


# 表示还没开始
Slave_IO_Running: No
Slave_SQL_Running: No
```

在从机开启主从

```
mysql> start slave;
Query OK, 0 rows affected, 1 warning (0.02 sec)

# 表示成功
Slave_IO_Running:Yes
Slave_SQL_Running: Yes
```

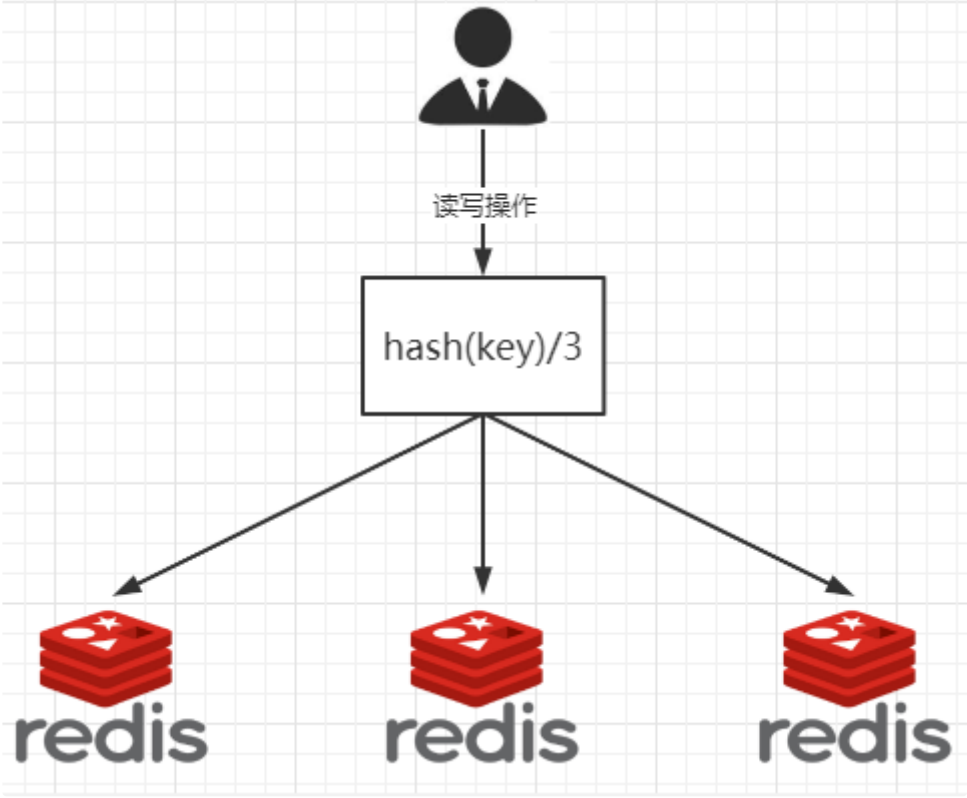
此时主机创建表 插入数据从机能看到查询到表示成功

(19条消息) Mysql主从同步时Slave_IO_Running: Connecting ; Slave_SQL_Running: Yes的情况故障排除*mbytes*的博客-*CSDN*博客
mysql slave_io_running

5） redis集群

· 1~2亿条数据需要缓存，请问如何设计这个存储案例

单机单台100%不可能，肯定是分布式存储，用redis如何落地?



2亿条记录就是2亿个k,v，我们单机不行必须要分布式多机，假设有3台机器构成一个集群，用户每次读写操作都是根据公式：hash(key) % N个机器台数，计算出哈希值，用来决定数据映射到哪一个节点上。
优点： 简单粗暴，直接有效，只需要预估好数据规划好节点，例如3台、8台、10台，就能保证一段时间的数据支撑。使用Hash算法让固定的一部分请求落到同一台服务器上，这样每台服务器固定处理一部分请求（并维护这些请求的信息），起到负载均衡+分而治之的作用。
缺点： 原来规划好的节点，进行扩容或者缩容就比较麻烦了额，不管扩缩，每次数据变动导致节点有变动，映射关系需要重新进行计算，在服务器个数固定不变时没有问题，如果需要弹性扩容或故障停机的情况下，原来的取模公式就会发生变化：Hash(key)/3会变成Hash(key)/?。此时地址经过取余运算的结果将发生很大变化，根据公式获取的服务器也会变得不可控。某个redis机器宕机了，由于台数数量变化，会导致hash取余全部数据重新洗牌。

一致性Hash算法背景

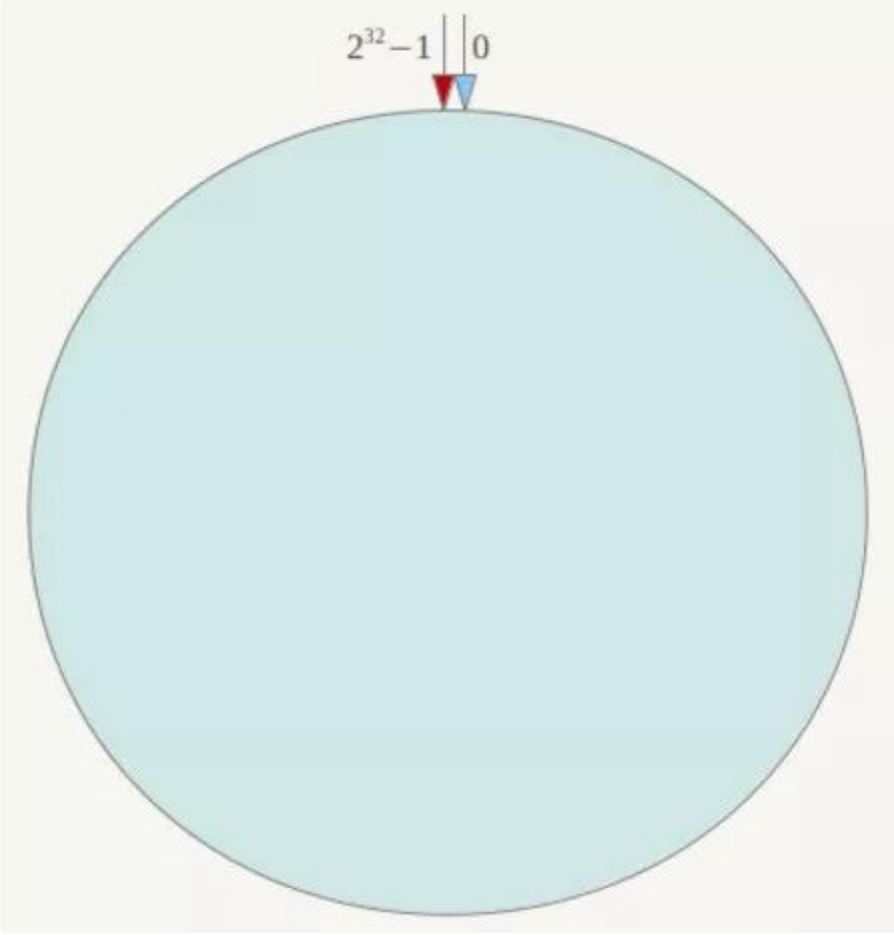
一致性哈希算法在1997年由麻省理工学院中提出的，设计目标是为了解决

分布式缓存数据变动和映射问题，某个机器宕机了，分母数量改变了，自然取余数不OK了。

一致性哈希环

一致性哈希算法必然有个hash函数并按照算法产生hash值，这个算法的所有可能哈希值会构成一个全量集，这个集合可以成为一个hash空间[0,2^32-1]，这个是一个线性空间，但是在算法中，我们通过适当的逻辑控制将它首尾相连(0 = 2^32),这样让它逻辑上形成了一个环形空间。

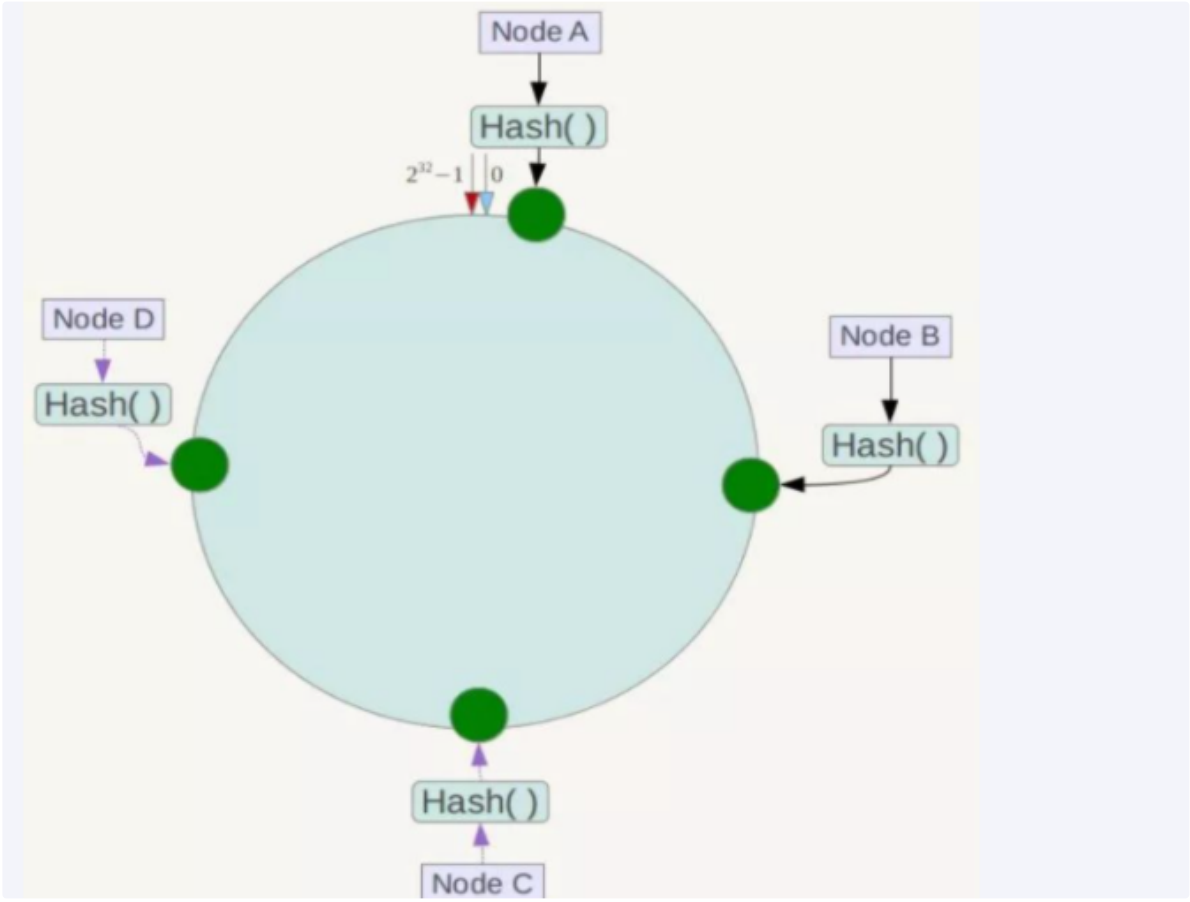
它也是按照使用取模的方法，前面笔记介绍的节点取模法是对节点（服务器）的数量进行取模。而一致性Hash算法是对2^32取模，简单来说，一致性Hash算法将整个哈希值空间组织成一个虚拟的圆环，如假设某哈希函数H的值空间为0-2^32-1（即哈希值是一个32位无符号整形），整个哈希环如下图：整个空间按顺时针方向组织，圆环的正上方的点代表0，0点右侧的第一个点代表1，以此类推，2、3、4、……直到2^32-1，也就是说0点左侧的第一个点代表2^32-1， 0和2^32-1在零点中方向重合，我们把这个由2^32个点组成的圆环称为Hash环。



节点映射

将集群中各个IP节点映射到环上的某一个位置。

将各个服务器使用Hash进行一个哈希，具体可以选择服务器的IP或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置。假如4个节点NodeA、B、C、D，经过IP地址的哈希函数计算(hash(ip))，使用IP地址哈希后在环空间的位置如下：



当我们需要存储一个kv键值对时，首先计算key的hash值，hash(key)，将这个key使用相同的函数Hash计算出哈希值并确定此数据在环上的位置，**从此位置沿环顺时针“行走”**，第一台遇到的服务器就是其应该定位到的服务器，并将该键值对存储在该节点上。

如我们有Object A、Object B、Object C、Object D四个数据对象，经过哈希计算后，在环空间上的位置如下：根据一致性Hash算法，数据A会被定为到Node A上，B被定为到Node B上，C被定为到Node C上，D被定为到Node D上。

容错性

假设Node C宕机，可以看到此时对象A、B、D不会受到影响，只有C对象被重定位到Node D。一般的，在一致性Hash算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

简单说，就是C挂了，受到影响的只是B、C之间的数据，并且这些数据会转移到D进行存储。

扩展性

数据量增加了，需要增加一台节点NodeX，X的位置在A和B之间，那收到影响的也就是A到X之间的数据，重新把A到X的数据录入到X上即可，

不会导致hash取余全部数据重新洗牌。

为了在节点数目发生改变时尽可能少的迁移数据

将所有的存储节点排列在收尾相接的Hash环上，每个key在计算Hash后会顺时针找到临近的存储节点存放。

而当有节点加入或退出时仅影响该节点在Hash环上顺时针相邻的后续节点。

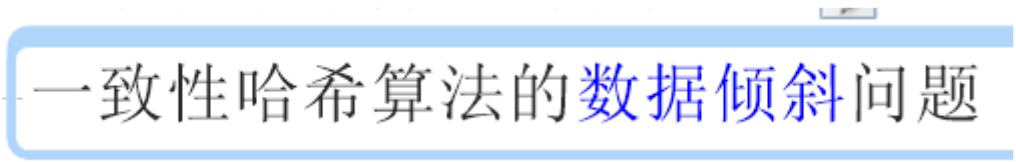
优点

加入和删除节点只影响哈希环中顺时针方向的相邻的节点，对其他节点无影响。

缺点

数据的分布和节点的位置有关，因为这些节点不是均匀的分布在哈希环上的，所以数据在进行存储时达不到均匀分布的效果。

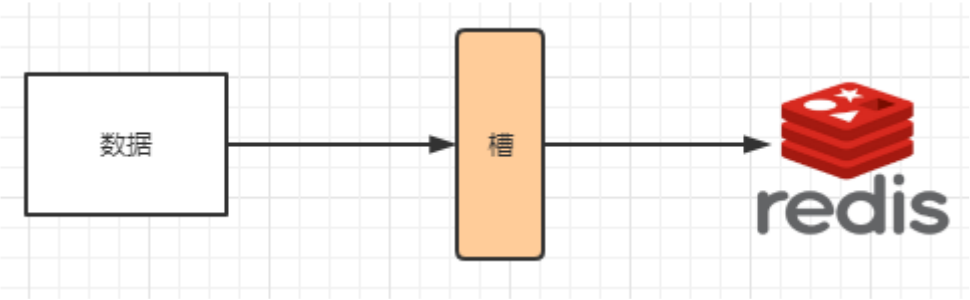
1 为什么出现



哈希槽实质就是一个数组，数组[0,2^14 -1]形成hash slot空间。

2 能干什么

解决均匀分配的问题，在数据和节点之间又加入了一层，把这层称为哈希槽（slot），用于管理数据和节点之间的关系，现在就相当于节点上放的是槽，槽里放的是数据。



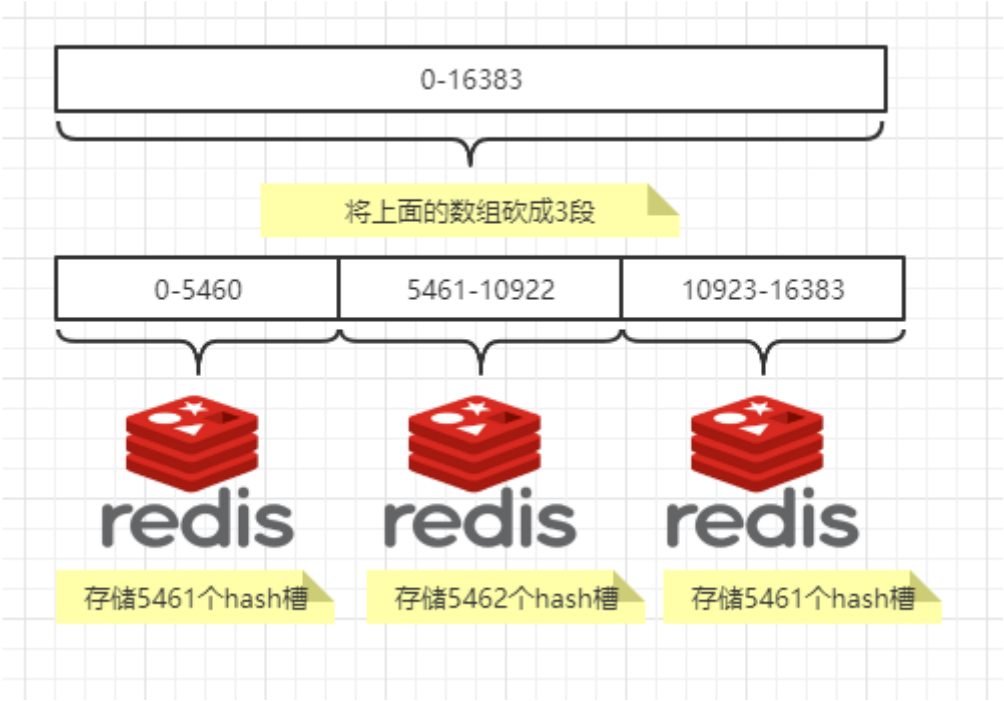
槽解决的是粒度问题，相当于把粒度变大了，这样便于数据移动。

哈希解决的是映射问题，使用key的哈希值来计算所在的槽，便于数据分配。

3 多少个hash槽

一个集群只能有16384个槽，编号0-16383（0-2^14-1）。这些槽会分配给集群中的所有主节点，分配策略没有要求。可以指定哪些编号的槽分配给哪个主节点。集群会记录节点和槽的对应关系。解决了节点和槽的关系后，接下来就需要对key求哈希值，然后对16384取余，余数是几key就落入对应的槽里。slot = CRC16(key) % 16384。以槽为单位移动数据，因为槽的数目是固定的，处理起来比较容易，这样数据移动问题就解决了。

Redis 集群中内置了 16384 个哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点。当需要在 Redis 集群中放置一个 key-value时，redis 先对 key 使用 crc16 算法算出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，也就是映射到某个节点上。如下代码，key之A 、B在Node2， key之C落在Node3上




```java
@Test
public void test3()
{
    //import io.lettuce.core.cluster.SlotHash;
    System.out.println(SlotHash.getSlot( key: "A")); //6373
    System.out.println(SlotHash.getSlot( key: "B")); //10374
    System.out.println(SlotHash.getSlot( key: "C")); //14503
    System.out.println(SlotHash.getSlot( key: "hello")); //866
}
```

搭建

```
docker run --name redis-node-1 -d --net host --
privileged=true -v /opt/redis-nodes/redis-node-
1:/data redis:6.0.8 --cluster-enabled yes --
appendonly yes --port 6381



docker run --name redis-node-2 -d --net host --
privileged=true -v /opt/redis-nodes/redis-node-
2:/data redis:6.0.8 --cluster-enabled yes --
appendonly yes --port 6382
```

```
docker run --name redis-node-3 -d --net host --
privileged=true -v /opt/redis-nodes/redis-node-
3:/data redis:6.0.8 --cluster-enabled yes --
appendonly yes --port 6383



docker run --name redis-node-4 -d --net host --
privileged=true -v /opt/redis-nodes/redis-node-
4:/data  redis:6.0.8 --cluster-enabled yes --
appendonly yes --port 6384



docker run --name redis-node-5 -d --net host --
privileged=true -v /opt/redis-nodes/redis-node-
5:/data redis:6.0.8 --cluster-enabled yes --
appendonly yes --port 6385



docker run --name redis-node-6 -d --net host --
privileged=true -v /opt/redis-nodes/redis-node-
6:/data redis:6.0.8 --cluster-enabled yes --
appendonly yes --port 6386
```

进入某一容器之后执行

```
redis-cli --cluster create 192.168.0.106:6381
192.168.0.106:6382 192.168.0.106:6383
192.168.0.106:6384 192.168.0.106:6385
192.168.0.106:6386 --cluster-replicas 1

# --cluster-replicas 1表示为每一个master创建一个节点



# 粘贴之后输入‘yes’
# 哈希槽的分配
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
```

```
Master[2] -> Slots 10923 - 16383


>>> Performing Cluster Check (using node
192.168.0.106:6381)
M: e9cf6cd8b07d0fd420f6f6d09ff67f58ffb524e5
192.168.0.106:6381
   slots:[0-5460] (5461 slots) master
   1 additional replica(s)
S: dea8451728dd8005a3c513e89ce6c24f306a2a88
192.168.0.106:6385
   slots: (0 slots) slave
   replicates
ab92082b8d1e79db3e96cfa231be5a4817c7eb85
M: 0890efd031ac30c23949cb770773ea898254f587
192.168.0.106:6383
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
S: ddfbb3e515bca09210e190dc80871cc43302738b
192.168.0.106:6386
   slots: (0 slots) slave
   replicates
0890efd031ac30c23949cb770773ea898254f587
M: ab92082b8d1e79db3e96cfa231be5a4817c7eb85
192.168.0.106:6382
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
S: 3e349f1ebbe5e130aa6e0bb47bfca06b44b9f2bd
192.168.0.106:6384
   slots: (0 slots) slave
   replicates
e9cf6cd8b07d0fd420f6f6d09ff67f58ffb524e5
```

查看集群信息

```
redis-cli -p 6381
# cluster info
127.0.0.1:6381> cluster info
cluster_state:ok
cluster_slots_assigned:16384
```

```
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:6
cluster_size:3
cluster_current_epoch:6
cluster_my_epoch:1
cluster_stats_messages_ping_sent:286
cluster_stats_messages_pong_sent:302
cluster_stats_messages_sent:588
cluster_stats_messages_ping_received:297
cluster_stats_messages_pong_received:286
cluster_stats_messages_meet_received:5
cluster_stats_messages_received:588
# cluster nodes
127.0.0.1:6381> cluster nodes
dea8451728dd8005a3c513e89ce6c24f306a2a88
192.168.0.106:6385@16385 slave
ab92082b8d1e79db3e96cfa231be5a4817c7eb85 0
1644155610030 2 connected
0890efd031ac30c23949cb770773ea898254f587
192.168.0.106:6383@16383 master - 0 1644155610000 3
connected 10923-16383
ddfbb3e515bca09210e190dc80871cc43302738b
192.168.0.106:6386@16386 slave
0890efd031ac30c23949cb770773ea898254f587 0
1644155608005 3 connected
ab92082b8d1e79db3e96cfa231be5a4817c7eb85
192.168.0.106:6382@16382 master - 0 1644155609014 2
connected 5461-10922
3e349f1ebbe5e130aa6e0bb47bfca06b44b9f2bd
192.168.0.106:6384@16384 slave
e9cf6cd8b07d0fd420f6f6d09ff67f58ffb524e5 0
1644155611043 1 connected
e9cf6cd8b07d0fd420f6f6d09ff67f58ffb524e5
192.168.0.106:6381@16381 myself,master - 0
1644155608000 1 connected 0-5460
```

存一个key

```
127.0.0.1:6381> set k1 v1
(error) MOVED 12706 192.168.0.106:6383
# ???
# 因为是集群

# 进入客户端前需要加参数 -c 防止路由失效
 redis-cli -p 6381 -c

root@192:/data# redis-cli -p 6381 -c
127.0.0.1:6381> set k1 v1
-> Redirected to slot [12706] located at
192.168.0.106:6383
OK

# 获取集群信息
redis-cli --cluster check 192.168.0.106:6383
```

主从切换

```
# 干掉6381
SHUTDOWN
# 查看集群
CLUSTER NODES
master,fail # 代表死亡

# 再次启动6381 变成slave
```

扩容

```
docker run --name redis-node-7 -d --net host --
privileged=true -v /opt/redis-nodes/redis-node-
7:/data redis:6.0.8 --cluster-enabled yes --
appendonly yes --port 6387



docker run --name redis-node-8 -d --net host --
privileged=true -v /opt/redis-nodes/redis-node-
8:/data redis:6.0.8 --cluster-enabled yes --
appendonly yes --port 6388
```

运行之后加入集群(在各自的容器中运行)

```
redis-cli --cluster add-node 实际ip:集群中的端口  要加
入的端口ip

redis-cli --cluster add-node 192.168.0.106:6387
192.168.0.106:6381

# 检查
redis-cli --cluster check 192.168.0.106:6383

# 没有槽位
M: 254016403991879c282b8586dfd60c99888e30f6
192.168.0.106:6387
   slots: (0 slots) master

# 分配槽号
# 重新洗牌
redis-cli --cluster reshard 192.168.0.106:6381

# 您要移动多少个插槽?  AA 槽数16384/master
How many slots do you want to move (from 1 to
16384)?  4096
# 分配给哪一个id
What is the receiving node ID?
254016403991879c282b8586dfd60c99888e30f6

# 输入all 全部重新分配
```

```
# 接着输入yes
# 前面的master每一个匀一点
redis-cli --cluster add-node 192.168.0.106:6388
192.168.0.106:6381
```

为什么6387是3个新的区间，以前的还是连续?

重新分配成本太高，所以前3家各自匀出来一部分，从6381/6382/6383三个旧节点分别匀出1364个坑位给新节点6387

将6388作为6387de从节点

```
redis-cli --cluster add-node 192.168.0.106:6388
192.168.0.106:6387 --cluster-slave --cluster-
master-id 254016403991879c282b8586dfd60c99888e30f6
# ⟶ 主节点的编号

# 检查四主四从
redis-cli --cluster check 192.168.0.106:6381
```

缩容

```
# 6387的ID
254016403991879c282b8586dfd60c99888e30f6
# 6388的id
1bc0e35ae1296f2c00ee5bd50efd6a6e38191257

# 删除6388节点  redis-cli --cluster del-node  地址 id
redis-cli --cluster del-node 192.168.0.106:6388
1bc0e35ae1296f2c00ee5bd50efd6a6e38191257

# 检查
redis-cli --cluster check 192.168.0.106:6381

# 清空6387槽号
 redis-cli --cluster reshard 192.168.0.106:6381

 # 按照公式 AA 16387/master
```

```
 How many slots do you want to move (from 1 to
16384)?  4096

# 根据id分配给6383
What is the receiving node ID?
0890efd031ac30c23949cb770773ea898254f587

# Type 'done' once you entered all the source nodes
IDs. done的意思是从哪里剥去节点
#
Source node #1:
254016403991879c282b8586dfd60c99888e30f6
# 剥去交给6381
Source node #2: done


# 检查 槽位为0
M: 254016403991879c282b8586dfd60c99888e30f6
192.168.0.106:6387
   slots: (0 slots) master

# 删除87节点
 redis-cli --cluster del-node 192.168.0.106:6387
254016403991879c282b8586dfd60c99888e30f6
```

六、DockerFile

基础知识

1. 每条保留字指令都必须为大写字母切后面至少要跟随一个参数，
2. 每条指令从上向下，顺序执行，
3. #表示注释
4. 每条指令都会创建一个新的镜像层并对镜像进行提交

1） 大致执行流程

（1） docker从基础镜像运行一一个容器

（2)执行一条指令并对容器作出修改

（3） 执行类似docker commit的操作提交一个新的镜像层

（4） docker再基于刚提交的镜像运行一个新容器

（5)执行dockerfile中的 下一条指令直到所有指令都执行完成

从应用软件的角度来看，Dockerfile、Docker镜像与Docker容器分别代表软件的三个不同阶段，

* Dockerfile是软件的原材料

* Docker镜像是软件的交付品

* Docker容器则可以认为是软件镜像的运行态，也即依照镜像运行的容器实例

Dockerfile面向开发，Docker镜像成为交付标准，Docker容器则涉及部署与运维，三者缺一不可，合力充当Docker体系的基石。

1 Dockerfile，需要定义一个Dockerfile，Dockerfile定义了进程需要的一切东西。Dockerfile涉及的内容包括执行代码或者是文件、环境变量、依赖包、运行时环境、动态链接库、操作系统的发行版、服务进程和内核进程(当应用进程需要和系统服务和内核进程打交道，这时需要考虑如何设计namespace的权限控制)等等;

2 Docker镜像，在用Dockerfile定义一个文件之后，docker build时会产生一个Docker镜像，当运行 Docker镜像时会真正开始提供服务;

3 Docker容器，容器是直接提供服务的

2） 常用保留字指令

1. FROM 基础镜像，当前新镜像是基于哪个镜像的，指定一个已经存在的镜像作为模板，第一条必须是from
2. MAINTAINER 镜像维护者的姓名和邮箱地址|
3. RUN 容器构建时所需要的命令 格式1 RUN < > 2.RUN ["",""]
4. EXPOSE 对外暴露的端口
5. WORKDIR 终端默认登录的路径
6. USER 以什么用户执行 默认root
7. ENV 环境变量 `ENV MYSQL_ROOT_PASSWORD 123456` 引用 $MYSQL_ROOT_PASSWORD
8. ADD 将宿主机目录下的文件拷贝进镜像且会自动处理URL和解压tar压缩包
9. COPY 同上
10. VOLUME 数据卷
11. CMD 指定容器启动后要干的事情 会被覆盖
12. ENTRYPOINT 类似于常量 不会被覆盖

- FROM 第一行必须是 来源自哪一个镜像?
- RUN 后面执行一些shell脚本之类的 例如 `RUN yum -y install vim` , 也支持json数组的形式 `RUN ["yum","-y","install","vim"]` 这样
- WORKDIR 默认进入的路径 没有会自动创建 `WORKDIR /usr/local`
- COPY 拷贝 `COPY wjl.txt /data/txt` JSON写法 `COPY "wjl.txt" "/data/txt"`
- ADD 支持url 自动下载 `ADD url /data/tar` url不会自动解压 本地打包的tar会自动解压
- VOLUME `VOLUME ["/data"]`
- ENV
- ENTRYPOINT 后面也是shell或者JSON数组 想覆盖掉它的命令的话 必须在docker run时执行 `--entrypoint=shell`
- CMD 配合以上使用时必须JSON形式

```
ENTRYPOINT ["ls /usr"]
CMD ["/aaa"] # 此时cmd被覆盖 合起来就是 ls /usr/【用户执行run输入的路径】
```

3） 案例

`docker pull centos`

下载jdk 在同一目录下创建 `Dockerfile` 文件

```
FROM centos
LABEL wjl="zzyybs@126.com"

ENV MYPATH /usr/local
WORKDIR $MYPATH

#安装vim编辑器
RUN yum -y install vim
#安装ifconfig命令查看网络IP
RUN yum -y install net-tools
#安装java8及lib库
RUN yum -y install glibc.i686
RUN mkdir /usr/local/java
#ADD 是相对路径jar,把jdk-8u171-linux-x64.tar.gz添加到
容器中,安装包必须要和Dockerfile文件在同一位置
ADD jdk-8u202-linux-x64.tar.gz  /usr/local/java/
#配置java环境变量
ENV JAVA_HOME /usr/local/java/jdk1.8.0_202
ENV JRE_HOME $JAVA_HOME/jre
ENV CLASSPATH
$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar:$JRE
_HOME/lib:$CLASSPATH
ENV PATH $JAVA_HOME/bin:$PATH

EXPOSE 80

CMD echo $MYPATH
CMD echo "success--------------ok"
CMD /bin/bash
```

```
docker bulid -t imageName:tag .
```

查看虚悬镜像

```
 docker image ls -f dangling=true
```

删除

```
docker image prune
```

4） 案例2 jar

```
FROM java:8
MAINTAINER WJL
VOLUME /tmp
ADD demo-0.0.1-SNAPSHOT.jar demo.jar
RUN bash -c 'touch /demo.jar'
ENTRYPOINT ["java","-jar","/demo.jar"]
EXPOSE 8080
```

```
docker bulid -t demo:1.0 .

docker run --name demo -d -p 8080:8080  demo:1.0
```

访问成功

七、docker network

从其架构和运行流程来看，Docker 是一个 C/S 模式的架构，后端是一个松耦合架构，众多模块各司其职。

Docker 运行的基本流程为：

1 用户是使用 Docker Client 与 Docker Daemon 建立通信，并发送请求给后者。

2 Docker Daemon 作为 Docker 架构中的主体部分，首先提供 Docker Server 的功能使其可以接受 Docker Client 的请求。

3 Docker Engine 执行 Docker 内部的一系列工作，每一项工作都是以一个 Job 的形式的存在。

4 Job 的运行过程中，当需要容器镜像时，则从 Docker Registry 中下载镜像，并通过镜像管理驱动 Graph driver将下载镜像以Graph的形式存储。

5 当需要为 Docker 创建网络环境时，通过网络管理驱动 Network driver 创建并配置 Docker 容器网络环境。

6 当需要限制 Docker 容器运行资源或执行用户指令等操作时，则通过 Execdriver 来完成。

7 Libcontainer是一项独立的容器管理包，Network driver以及 Exec driver都是通过Libcontainer来实现具体对容器进行的操作。

- 查看docker网络 `docker network ls`

docker 启动后 默认创建的三个网络

```
NETWORK ID     NAME      DRIVER    SCOPE
e870d25f9a27   bridge    bridge    local
1deb4410558a   host      host      local
1b29da45b6a7   none      null      local
```

```
[root@192 myfile]# docker network --help

Usage:  docker network COMMAND

Manage networks

Commands:
  connect     Connect a container to a network
  create      Create a network
  disconnect  Disconnect a container from a network
  inspect     Display detailed information on one
or more networks
```

```
  ls          List networks
  prune       Remove all unused networks
  rm          Remove one or more networks
```

1）网络模式

网络模式	简介说明
bridge	为每一个容器分配设置ip等，将容器连接到一个 docker0 的虚拟网桥【默认】
host	容器不会虚拟出自己的网卡，配置自己的ip，使用宿主机的ip端口
none	容器有独立的network namespace 但并没有对其进行任何网络设置，如非配veth pair和网桥连接，IP等
container	新创建的容器不会创建自己的网卡和配置自己的ip。而是和一个指定容器共享ip,端口范围等

- --network bridge
- --network host
- --network none
- --network container:NAME/id

Docker 服务默认会创建一个 docker0 网桥（其上有一个 docker0 内部接口），该桥接网络的名称为docker0，它在内核层连通了其他的物理或虚拟网卡，这就将所有容器和本地主机都放到同一个物理网络。
Docker 默认指定了 docker0 接口 的 IP 地址和子网掩码，让主机和容器之间可以通过网桥相互通信。

```
# 查看 bridge 网络的详细信息，并通过 grep 获取名称项

docker network inspect bridge | grep name
```

```
[root@zzyy tmp]# docker network inspect bridge | grep name
            "com.docker.network.bridge.name": "docker0",
[root@zzyy tmp]#
```

ifconfig

```
[root@zzyy tmp]# ifconfig | grep docker
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
[root@zzyy tmp]#
```

1 Docker使用Linux桥接，在宿主机虚拟一个Docker容器网桥(docker0)，Docker启动一个容器时会根据Docker网桥的网段分配给容器一个IP地址，称为Container-IP，同时Docker网桥是每个容器的默认网关。因为在同一宿主机内的容器都接入同一个网桥，这样容器之间就能够通过容器的Container-IP直接通信。

2 docker run 的时候，没有指定network的话默认使用的网桥模式就是bridge，使用的就是docker0。在宿主机ifconfig,就可以看到docker0和自己create的network(后面讲)eth0，eth1，eth2……代表网卡一，网卡二，网卡三……，lo代表127.0.0.1，即localhost，inet addr用来表示网卡的IP地址

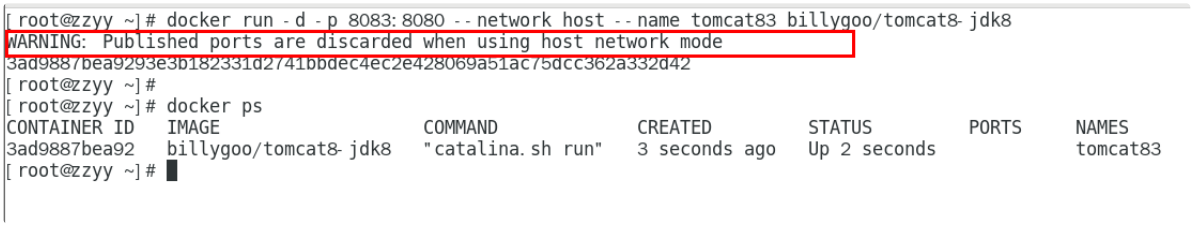
3 网桥docker0创建一对对等虚拟设备接口一个叫veth，另一个叫eth0，成对匹配。

3.1 整个宿主机的网桥模式都是docker0，类似一个交换机有一堆接口，每个接口叫veth，在本地主机和容器内分别创建一个虚拟接口，并让他们彼此联通（这样一对接口叫veth pair）；

3.2 每个容器实例内部也有一块网卡，每个接口叫eth0;

3.3 docker0上面的每个veth匹配某个容器实例内部的eth0，两两配对，一一匹配。

通过上述，将宿主机上的所有容器都连接到这个内部网络上，两个容器在同一个网络下,会从这个网关下各自拿到分配的ip，此时两个容器的网络是互通的。



问题：

docke启动时总是遇见标题中的警告

原因：

docker启动时指定--network=host或-net=host，如果还指定了-p映射端口，那这个时候就会有此警告，

并且通过-p设置的参数将不会起到任何作用，端口号会以主机端口号为主，重复时则递增。

解决：

解决的办法就是使用docker的其他网络模式，例如--network=bridge，这样就可以解决问题，或者直接无视

2）自定义网络

```
# 自定义网络
docker network create ap

# 使用网络
docker run -it --name a1 --network ap  -p 8080:80 alpine:latest

docker run -it --name a2 --network ap  -p 8081:80 alpine:latest
```

效果 在两个容器内部 a1 直接 `ping a2`

a2直接 `ping a1` 都可以成功

八、docker-compose容器编排

1） 简介

Compose 是 Docker 公司推出的一个工具软件，可以管理多个 Docker 容器组成一个应用。你需要定义一个 YAML 格式的配置文件docker-compose.yml，写好多个容器之间的调用关系。然后，只要一个命令，就能同时启动/关闭这些容器

docker建议我们每一个容器中只运行一个服务,因为docker容器本身占用资源极少,所以最好是将每个服务单独的分割开来但是这样我们又面临了一个问题?

如果我需要同时部署好多个服务,难道要每个服务单独写Dockerfile然后在构建镜像,构建容器,这样累都累死了,所以docker官方给我们提供了docker-compose多服务部署的工具

例如要实现一个Web微服务项目，除了Web服务容器本身，往往还需要再加上后端的数据库mysql服务容器，redis服务器，注册中心eureka，甚至还包括负载均衡容器等等。。。。。。

Compose允许用户通过一个单独的docker-compose.yml模板文件（YAML 格式）来定义一组相关联的应用容器为一个项目（project）。

可以很容易地用一个配置文件定义一个多容器的应用，然后使用一条指令安装这个应用的所有依赖，完成构建。Docker-Compose 解决了容器与容器之间如何管理编排的问题。

2） 安装

官网教程⟶ https://docs.docker.com/compose/install/

```
sudo curl -L
"https://github.com/docker/compose/releases/downloa
d/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o
/usr/local/bin/docker-compose

# 执行权限
sudo chmod +x /usr/local/bin/docker-compose
```

3） 常用命令

```
Compose常用命令

docker-compose -h                                    # 查看帮
助

docker-compose up                                    # 启动所
有docker-compose服务

docker-compose up -d                                 # 启动所
有docker-compose服务并后台运行

docker-compose down                                  # 停止并
删除容器、网络、卷、镜像。

docker-compose exec  yml里面的服务id       # 进入容器实
例内部  docker-compose exec docker-compose.yml文件中
写的服务id /bin/bash

docker-compose ps                              # 展示当前
docker-compose编排过的运行的所有容器
```

```
docker-compose top                          # 展示当前
docker-compose编排过的容器进程

docker-compose logs  yml里面的服务id       # 查看容器输出
日志

docker-compose config     # 检查配置

docker-compose config -q  # 检查配置，有问题才有输出

docker-compose restart   # 重启服务

docker-compose start     # 启动服务

docker-compose stop      # 停止服务
```

4) docker-compose.yml

```
services: # 代表多个服务
  tomcat: # 服务名
    image: billygoo/tomcat8-jdk8:latest # 指定运行的
镜像
    container_name: "tom"
    ports: # 指定映射
      - "8080:8080"
```

volumes

```
version: "3.0" # 选择使用的版本
services: # 代表多个服务
  tomcat: # 服务名 可以自己起
    image: billygoo/tomcat8-jdk8:latest # 指定运行的
镜像
    container_name: "tom"
    ports: # 指定映射
      - "8080:8080"
```

```
    volumes:
      - tomcatwebapps:/usr/local/tomcat/webapps


# 不支持自动创建所以声明指定的卷名"idea_tomcatwebapps
volumes:
  tomcatwebapps:
# 创建出来会以项目（当前文件夹）加文件名
    external: true
    # 注意使用自定义卷名要自己创建
    # docker volumes create tomcatwebapps
```

networks

```
version: "3.0" # 选择使用的版本
services: # 代表多个服务或者一个应用
  tomcat: # 服务名
    image: billygoo/tomcat8-jdk8:latest # 指定运行的
镜像
    container_name: "tom"
    ports: # 指定映射
      - "8080:8080"
    networks:
      - hello
  #  tomcat1 tom2 都是可以ping通的
  tomcat1: # 服务名
    image: billygoo/tomcat8-jdk8:latest # 指定运行的
镜像
    container_name: "tom2"
    ports: # 指定映射
      - "8081:8080"
    networks:
      - hello
# 自定义网络桥
networks:
  hello:
```

指定命令

```
version: "3.0" # 选择使用的版本
services: # 代表多个服务
  tomcat: # 服务名
    image: billygoo/tomcat8-jdk8:latest # 指定运行的
镜像
    container_name: "tom"
    ports: # 指定映射
      - "8080:8080"
    networks:
      - hello
  tomcat1: # 服务名
    image: billygoo/tomcat8-jdk8:latest # 指定运行的
镜像
    container_name: "tom2"
    ports: # 指定映射
      - "8081:8080"
    networks:
      - hello
  mysqlgo:
    container_name: "mysql"
    image: mysql:latest
    volumes:
      - mysqldata:/var/lib/mysql
      - mysqlconf:/etc/mysql
    environment: # 加 - 就是等号
      - MYSQL_ROOT_PASSWORD=123456
    ports:
      - "3306:3306"
    networks:
      - hello
  myredis:
    container_name: "redis"
    image: redis:6.0.8
    volumes:
      - redisdata:/data
    ports:
      - "6379:6379"

    networks:
```

```
      - hello
    command: # "redis-server  --appendonly yes"
      - "redis-server"
      - "--appendonly yes"
# 自定义网络桥 external 必须自己创建
networks:
  hello:
    external: true
volumes:
  mysqldata:
  mysqlconf:
  redisdata:
```

文件外环境变量

```
version: "3.0"
services:
  nacos:
    image: nacos/nacos-server:latest
    ports:
      - "8848:8848"
    env_file:
      - nacos.env
```

env文件

```
# nacos 必须加一行注释
MODE=standalone  # nacos单实例启动
```

启动顺序

```
version: "3.0"
services:
  nacos:
```

```
    image: nacos/nacos-server:latest
    ports:
      - "8848:8848"
    env_file:
      - nacos.env
    networks:
      - hello
    depends_on: # 指定这个容器必须依赖于哪一个容器启动才能
启动
      - tomcat-service # 指定服务名而不是镜像容器名
  tomcat-service:
    image: billygoo/tomcat8-jdk8:latest
    container_name: "tom"
    ports:
      - "8080:8080"
    networks:
      - hello
networks:
  hello:
    external: true

# 此时启动后台日志可以明显看到nacos在tom之后启动
```

心跳检测

```
healthcheck:
      test: ["CMD","curl","http:localhost"]
      interval: 1m30s
      timeout: 10s
      retries: 3
```

系统参数[少]

```
sysctls: # 修修改容器执行的系统参数
      - net.core.somaxconn=1024
```

容器内进程[少]

```
ulimits:
      nproc: 65535
      nofile:
        soft: 20000
        hard: 40000
```

bulid

Dockerfile

```
FROM java:8
ADD demo.jar /tmp/demo.jar
ENTRYPOINT ["java -jar","demo.jar"]
EXPOSE 8084
```

build: 运行docker bulid 并直接运行镜像

```
version: "3.0"
services:
  demo:
    #build: . 这样也可以但是警告
    build:
      # 指定路径
      context: /docker
      dockerfile: Dockerfile
    container_name: "demo"
    ports:
      - "8084:8084"
```

```yaml
version: "3"


services:
  microService:
    image: zzyy_docker:1.6
    container_name: ms01
    ports:
      - "6001:6001"
    volumes:
      - /app/microService:/data
    networks:
      - atguigu_net
    depends_on:
      - redis
      - mysql


  redis:
    image: redis:6.0.8
    ports:
      - "6379:6379"
    volumes:
```

```
      - /app/redis/redis.conf:/etc/redis/redis.conf
      - /app/redis/data:/data
    networks:
      - atguigu_net
    command: redis-server /etc/redis/redis.conf


  mysql:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: '123456'
      MYSQL_ALLOW_EMPTY_PASSWORD: 'no'
      MYSQL_DATABASE: 'db2021'
      MYSQL_USER: 'zzyy'
      MYSQL_PASSWORD: 'zzyy123'
    ports:
       - "3306:3306"
    volumes:
       - /app/mysql/db:/var/lib/mysql
       - /app/mysql/conf/my.cnf:/etc/my.cnf
       - /app/mysql/init:/docker-entrypoint-
initdb.d
```

```
    networks:
      - atguigu_net
    command: --default-authentication-
plugin=mysql_native_password #解决外部无法访问


networks:
   atguigu_net:
```

九、docker 可视化工具

1) 安装

官方文档⟶ https://docs.portainer.io/v/ce-2.6/start/install

```
 docker search poratiner # 搜索 有官方和中文版
 此存储库已弃用。从 2022 年 1 月开始，此存储库的最新标签将指向 Portainer CE 2.X。请改用 portainer/portainer-ce。

 docker pull portainer/portainer-ce

# 1.--restart=always 表示跟随docker重启
 docker run -d -p 8000:8000 -p 9000:9000 --name=portainer --restart=always -v /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data portainer/portainer-ce:2.6.3
# 访问9000
```

界面没啥可说的

十、容器监控CIG

docker system df/ docker stats命令的进阶

安装

```
# 创建一个目录
mkdir
```

```
version: '3.1'
volumes:
  grafana_data: {}
services:
  influxdb:
    image: tutum/influxdb:0.9
    restart: always
    environment:
      - PRE_CREATE_DB=cadvisor
```

```
    ports:
      - "8083:8083"
      - "8086:8086"
    volumes:
      - ./data/influxdb:/data
  cadvisor:
    image: google/cadvisor
    links:
      - influxdb:influxsrv
    command:
      - storage_driver=influxdb -
storage_driver_db=cadvisor -
storage_driver_host=influxsrv:8086
    restart: always
    ports:
      - "8080:8080"
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:rw
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro


  grafana:
    user: "104"
    image: grafana/grafana
    user: "104"
    restart: always
    links:
      - influxdb:influxsrv
    ports:
      - "3000:3000"
    volumes:
      - grafana_data:/var/lib/grafana
    environment:
      - HTTP_USER=admin
      - HTTP_PASS=admin
      - INFLUXDB_HOST=influxsrv
      - INFLUXDB_PORT=8086
      - INFLUXDB_NAME=cadvisor
      - INFLUXDB_USER=root
```

```
      - INFLUXDB_PASS=root
```

访问8080, 8083,3000

都能出现图形化界面

.....

十一、教程链接

https://www.bilibili.com/video/BV1gr4y1U7CY

https://www.bilibili.com/video/BV1ZT4y1K75K