服务云端部署&DevOps 流水线部署模式实战

1 下单性能优化

1.1 业务改造思考

```
// 从数据库查询商品数据
// 优化一： 从缓存中获取数据
TbSeckillGoods seckillGoods =
seckillGoodsMapper.selectByPrimaryKey(killId);
//库存扣减

// 优化二：操作缓存，先不考虑数据一致性问题
seckillGoods.setStockCount(seckillGoods.getStockCount() - 1);
//更新库存
seckillGoodsMapper.updateByPrimaryKeySelective(seckillGoods);


//下单
// 优化三：写操作，异步的操作
TbSeckillOrder order = new TbSeckillOrder();
order.setSeckillId(killId);
order.setUserId(userId);
order.setCreateTime(new Date());
order.setStatus("0");
order.setMoney(seckillGoods.getCostPrice());

//保存订单
seckillOrderMapper.insertSelective(order);
```

1.2 缓存改造

首先需求把秒杀商品存储在 Redis 缓存中，同时把秒杀商品库存单独存储在 Redis 服务器中；

Redis

```
# redis 商品
key: seckill_goods_1
value: {"id":1,name:vivo,stock:2}

# 库存
key: seckill_goods_stock_1
value:2
```

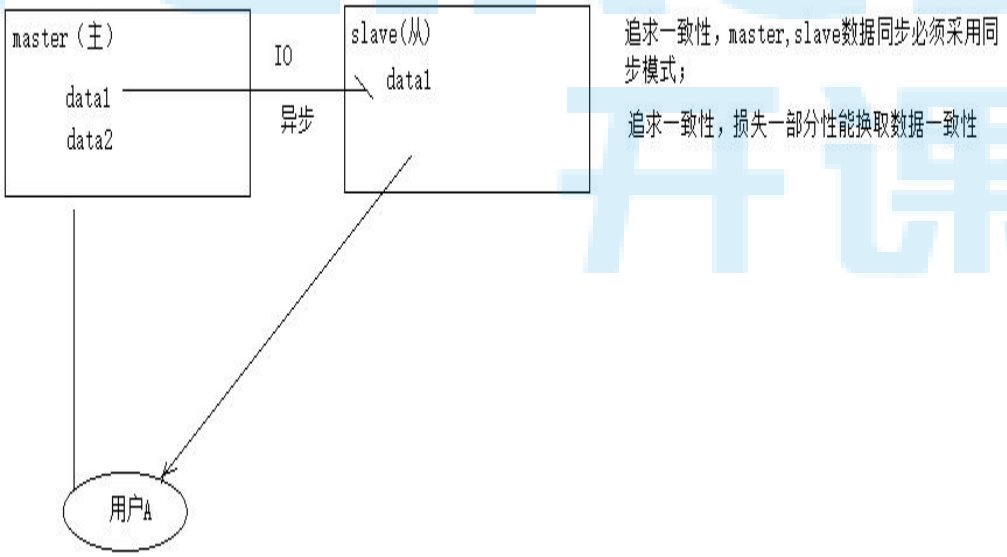
实现下单业务优化改造工作：

```
// 1、 从缓存中获取秒杀商品数据
TbSeckillGoods seckillGoods = (TbSeckillGoods)
redisTemplate.opsForValue().get("seckill_goods_"+killId);
//2、利用 redis 原子性操作扣减库存，不需要上锁
boolean res = reduceStock(killId);


//3、异步实现（blockingQueue,disruptor,rocketMQ 队列实现异步）
//下单
TbSeckillOrder order = new TbSeckillOrder();
// 队列实现异步下单操作
Boolean produce = SeckillQueue.getMailQueue().produce(order);

if(!produce){
   return HttpResult.error("秒杀失败");
}

return HttpResult.ok("秒杀成功");
```

下单操作经过 3 个步骤的优化，吞吐能力显著提升：

2 数据一致性问题

2.1 CAP 定理



因此在分布式模式下，CAP 理论要求不做一个平衡，不能同时要求可用性，一致性；

2.2 数据一致性问题

1、扣减库存是扣减的数据库的库存

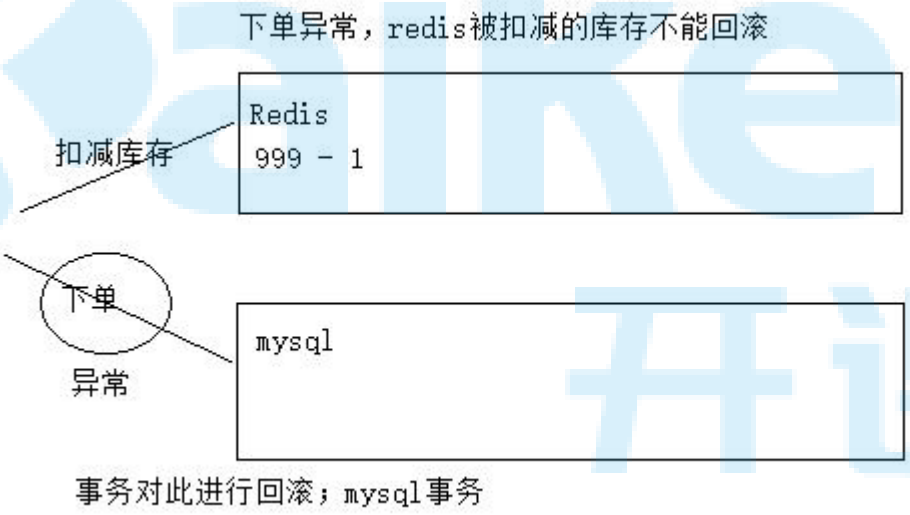
```
//2、利用redis原子性操作扣减库存，不需要上锁
boolean res = reduceStock(killId);
```

缓存数据 ，数据库库存数据一致性问题； 此时数据库的库存和缓存的库存不一致；

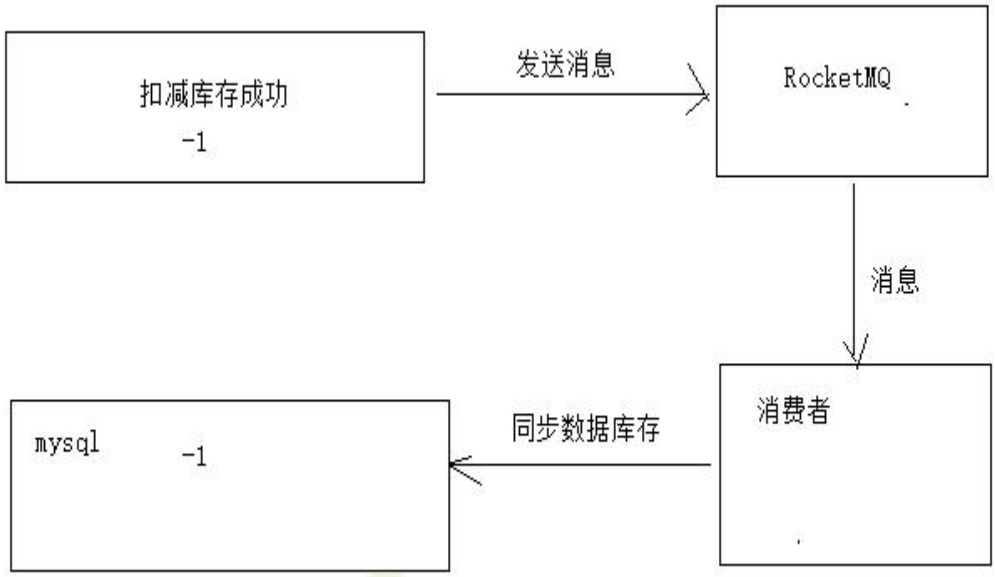
2、下单（操作的数据库）

```
//下单
TbSeckillOrder order = new TbSeckillOrder();
order.setSeckillId(killId);
order.setUserId(userId);
order.setCreateTime(new Date());
order.setStatus("0");
order.setMoney(seckillGoods.getCostPrice());
// 队列实现异步下单操作
Boolean produce = SeckillQueue.getMailQueue().produce(order);
```

下单（出现异常）失败了（下单操作是 MySQL 事务，可以回滚），扣减库存确成功了（扣减库存是 redis 操作，无法回滚）！！！



解决问题：缓存数据库库存一致性问题



使用消息中间件保证 缓存库存， 和数据库库存 进行消息同步的；是的 redis 数据，mysql 的数据保持同步；

1）发送消息

```
* @Description: 库存扣减方法
* @Author: hubin
* @CreateDate: 2021/1/24 22:27
* @UpdateUser: hubin
* @UpdateDate: 2021/1/24 22:27
* @UpdateRemark: 修改内容
* @Version: 1.0
*/
private boolean reduceStock(Long killId) {

    Long res = redisTemplate.opsForValue().increment( k: "seckill_goods_stock_" + killId, l: -1);
    if(res > 0){
        // 发送消息
        producer.asncSendMsg(killId);    扣减Redis库存，发送一个消息
        return true;
    }else if(res == 0){
        producer.asncSendMsg(killId);
        //记录标识,表示此商品已经售卖结束
        redisTemplate.opsForValue().set("seckill_goods_stock_end_"+killId,"STOCK_END");
        return true;
    }else {
        // 网络异常，扣减库存发生失败
        redisTemplate.opsForValue().increment( k: "seckill_goods_stock_" + killId, l: 1);

        return false;
    }
```

2）消息消费，同步数据库库存

```
consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
consumer.registerMessageListener((MessageListenerConcurrently) (list, context) -> {
    try {
        for (MessageExt messageExt : list) {
            String killId = new String(messageExt.getBody(), RemotingHelper.DEFAULT_CHARSET);
            //执行扣减库存的操作
            //同步数据库的库存                    更新数据库的库存，从而保证缓存库存， 数据库库存一致
            seckillGoodsMapper.updateSeckillGoodsByPrimaryKeyByLock(Long.parseLong(killId));
            System.out.println("[Consumer] msgID(" + messageExt.getMsgId() + ") msgBody : " + ki
        }
    } catch (Exception e) {
        e.printStackTrace();
        //如果出现异常，必须告知消息进行重试
        return ConsumeConcurrentlyStatus.RECONSUME_LATER;
    }
    return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
```

2.3 存在问题？

以上数据一致性解决方案中： 解决了缓存的库存，和数据库库存一致性问题；那么以上的解决方案是否可以行？ 如果不可行，存在什么样问题？？

问题 1： 下单操作出现异常（下单失败，下单操作是数据库操作，数据库事务可以回滚），但是在下单这个操作之前的缓存扣减库存的操作不能回滚；因此 下单操作，扣减库存操作不一致的，因为他们不是一个原子操作；

问题 2： 发送消息失败了呢？？（发送消息，扣减库存也不是一个原子操作）

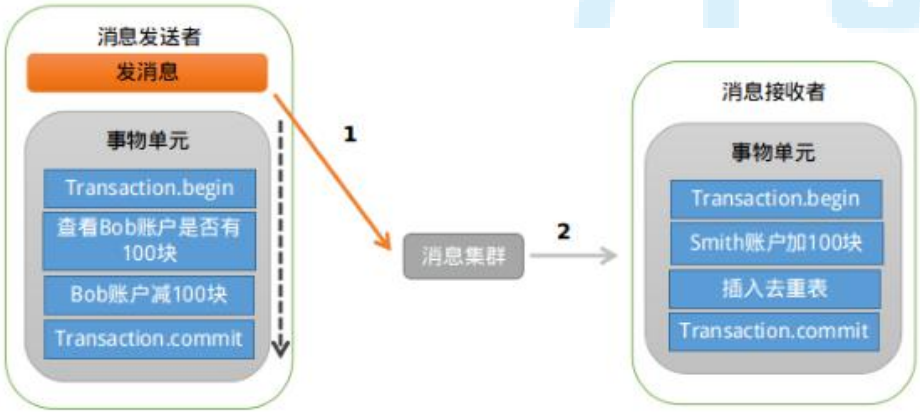
备选方案： 事务提交之后，再去发送消息？？

问题 3： 库存回补失败了怎么办呢？？

思考以上的问题，该如何解决？？？ 提出你的解决方案？？

解决方案： 利用 RocketMQ 事务消息实现最终消息一致性；

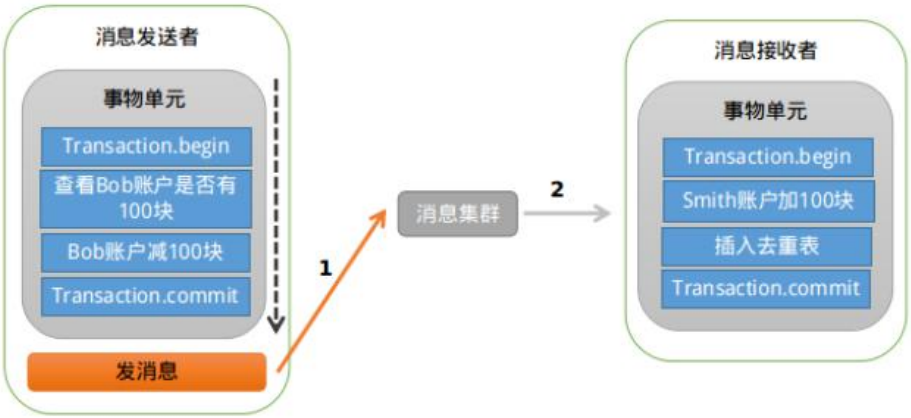
情形一： 本地事务执行之前发送消息



业务场景： 先发送消息，然后再执行本地事务

存在问题： 发送消息成功，本地事务执行失败（本地事务回滚），但是对于消息消费者来说执行业务；

2）情形二： 本地事务提交之后，再发消息

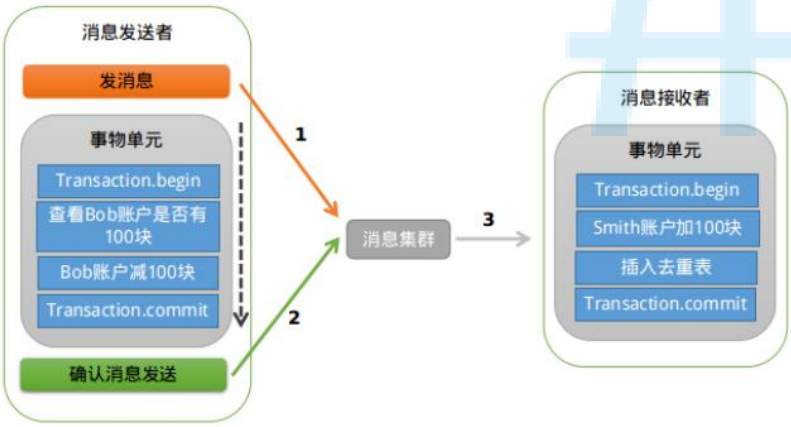


业务场景： 本地事务执行结束后，再发送消息

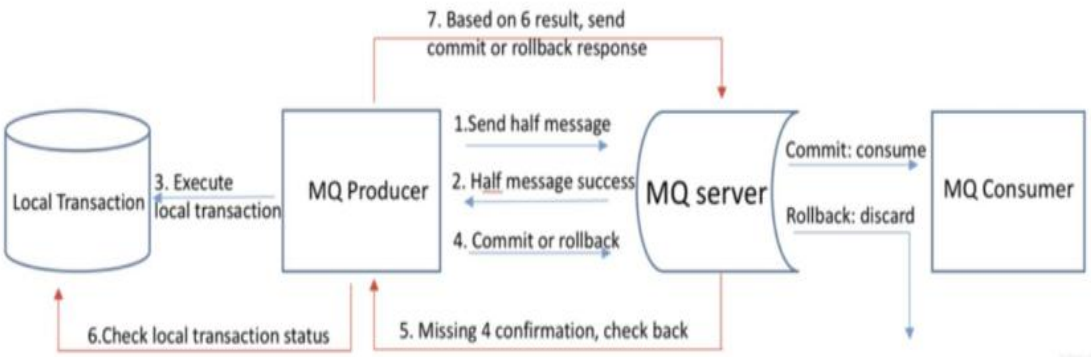
```
// 事务提交之后发送消息
TransactionSynchronizationManager.registerSynchronization(new TransactionSynchronizationAdapter() {
    // 等待事务提交之后再 do something
    @Override
    public void afterCommit() {
        super.afterCommit();
    }
});
```

存在问题：本地事务执行成功，发送消息异常（发送消息失败），也会导致两边的数据不一致；

3）Half message 事务消息



事务消息： half message 半消息机制方式解决最终消息一致性；



3 事务消息

解决数据一致性问题，使用 rocketmq 最终消息一致性,充分考虑到了业务接口的性能，以及数据一致性问题；

3.1 发送事务消息

```
package com.sugo.seckill.mq;


@Component
public class MqProducer {

    private static final String producerGroup = "seckillGroup";
    private static final String namesrvAddr = "127.0.0.1:9876";
    //private DefaultMQProducer producer;
    private TransactionMQProducer producer;

    //注入订单服务
    @Autowired
    private SeckillOrderService orderService;

    //注入
    @Autowired
    private SeckillGoodsMapper seckillGoodsMapper;

    @PostConstruct
    public void initProducer() {
```

```
        producer = new TransactionMQProducer(producerGroup);
        producer.setNamesrvAddr(namesrvAddr);
        producer.setRetryTimesWhenSendFailed(3);
        try {
            producer.start();

            //注入一个监听器
            producer.setTransactionListener(new
TransactionListener() {


                /**
                 * @Description: 执行本地业务的方法
                 * @Author: hubin
                 * @CreateDate: 2021/1/26 20:39
                 * @UpdateUser: hubin
                 * @UpdateDate: 2021/1/26 20:39
                 * @UpdateRemark: 修改内容
                 * @Version: 1.0
                 */
                @Override
                public LocalTransactionState
executeLocalTransaction(Message message, Object o) {

                    String seckillId = null;
                    try {
                        // 接受消息内容
                        String msg = new
String(message.getBody(),RemotingHelper.DEFAULT_CHARSET);

                        // 获取消息内存map
                        Map<String,String> maps =
JSON.parseObject(msg,Map.class);

                        // 获取消息内容
                        seckillId = maps.get("seckillId");
                        String userId = maps.get("userId");

                        // 调用下单的方法

orderService.startKilledWithRedis(Long.parseLong(seckillId),us
erId);
```

```
                } catch (UnsupportedEncodingException e) {
                    e.printStackTrace();
                }catch (BaseException e){
                    // 业务处理中出现一个预知的异常
                    // 设置事务回滚状态
                    TbSeckillGoods seckillGoods =
seckillGoodsMapper.selectByPrimaryKey(seckillId);
                    seckillGoods.setTransactionStatus(2);
                    seckillGoods.setStockCount(null);


seckillGoodsMapper.updateByPrimaryKeySelective(seckillGoods);

                    // 返回回滚
                    return
LocalTransactionState.ROLLBACK_MESSAGE;

                }

                // 业务执行成功，确定事务提交状态
                return LocalTransactionState.COMMIT_MESSAGE;
            }

            /**
             * @Description: 事务状态回查方法
             * @Author: hubin
             * @CreateDate: 2021/1/26 20:39
             * @UpdateUser: hubin
             * @UpdateDate: 2021/1/26 20:39
             * @UpdateRemark: 修改内容
             * @Version: 1.0
             */
            @Override
            public LocalTransactionState
checkLocalTransaction(MessageExt messageExt) {

                try {
                    // 接受消息内容
                    String msg = new
String(messageExt.getBody(),RemotingHelper.DEFAULT_CHARSET);

                    // 获取消息内存map
                    Map<String,String> maps =
```

```
JSON.parseObject(msg,Map.class);

                        // 获取消息内容
                        String seckillId = maps.get("seckillId");
                        String userId = maps.get("userId");

                        // 查询事务状态
                        TbSeckillGoods seckillGoods =
seckillGoodsMapper.selectByPrimaryKey(seckillId);

                        // 根据事务状态，判定事务 commit ,
rollback,unkown
                        if(seckillGoods.getTransactionStatus() ==
0){
                            return LocalTransactionState.UNKNOW;
                        }
                        if(seckillGoods.getTransactionStatus() ==
1){
                            return
LocalTransactionState.COMMIT_MESSAGE;
                        }
                        if(seckillGoods.getTransactionStatus() ==
2){
                            return
LocalTransactionState.ROLLBACK_MESSAGE;
                        }


                    } catch (UnsupportedEncodingException e) {
                        e.printStackTrace();
                    }
                    // 确认提交
                    return LocalTransactionState.COMMIT_MESSAGE;
                }
            });
```

```
            System.out.println("[Producer 已启动]");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public String send(String topic, String tags, String msg) {
        SendResult result = null;
        try {
            Message message = new Message(topic, tags,
msg.getBytes(RemotingHelper.DEFAULT_CHARSET));
            result = producer.send(message);
            System.out.println("[Producer] msgID(" +
result.getMsgId() + ") " + result.getSendStatus());
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "{\"MsgId\":\"" + result.getMsgId() + "\"}";
    }

    @PreDestroy
    public void shutDownProducer() {
        if (producer != null) {
            producer.shutdown();
        }
    }

    /**
     * @Description: 发送消息，同步数据库库存
     * @Author: hubin
     * @CreateDate: 2020/10/26 21:59
     * @UpdateUser: hubin
     * @UpdateDate: 2020/10/26 21:59
     * @UpdateRemark: 修改内容
     * @Version: 1.0
     */
    public boolean asncSendMsg(Long seckillId) {
        try {
            Message message = new
Message("seckill_goods_asnc_stock", "increase",
(seckillId+"").getBytes(RemotingHelper.DEFAULT_CHARSET));
            //发送消息
```

```
            producer.send(message);
        } catch (Exception e) {
            e.printStackTrace();
            //发送失败
            return false;
        }
        return true;
    }



    /**
     * @Description: 发送消息，使用事务型消息把所有的操作原子化
     * @Author: hubin
     * @CreateDate: 2020/10/26 21:59
     * @UpdateUser: hubin
     * @UpdateDate: 2020/10/26 21:59
     * @UpdateRemark: 修改内容
     * @Version: 1.0
     */
    public boolean asncSendTransactionMsg(Long seckillId,String
userId) {
        try {

            Map<String,String> maps = new HashMap<>();
            maps.put("seckillId",seckillId+"");
            maps.put("userId",userId);

            //把对象转换为字符串
            String jsonStr = JSON.toJSONString(maps);

            Message message = new
Message("seckill_goods_asnc_stock", "increase",
jsonStr.getBytes(RemotingHelper.DEFAULT_CHARSET));
            //发送事务消息
           producer.sendMessageInTransaction(message,null);
        } catch (Exception e) {
            e.printStackTrace();
            //发送失败
            return false;
        }
        return true;
```

```
    }
}
```

3.2 接受事务消息

```
@Bean
public DefaultMQPushConsumer defaultMQPushConsumer() {
    DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer(consumerGroup);
    consumer.setNamesrvAddr(namesrvAddr);
    try {
        //广播模式消费
        //consumer.setMessageModel(MessageModel.BROADCASTING);
        consumer.subscribe("seckill_goods_asnc_stock", "*");

        // 如果是第一次启动，从队列头部开始消费
        // 如果不是第一次启动，从上次消费的位置继续消费

consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIR
ST_OFFSET);

consumer.registerMessageListener((MessageListenerConcurrently)
(list, context) -> {
            try {
                for (MessageExt messageExt : list) {
                    String msg = new String(messageExt.getBody(),
RemotingHelper.DEFAULT_CHARSET);

                    // 获取消息内存map
                    Map<String,String> maps =
JSON.parseObject(msg,Map.class);

                    // 获取消息内容
                    String seckillId = maps.get("seckillId");

                    //执行扣减库存的操作
```

```
                    //同步数据库的库存

seckillGoodsMapper.updateSeckillGoodsByPrimaryKeyByLock(Long.p
arseLong(seckillId));
                    System.out.println("[Consumer] msgID(" +
messageExt.getMsgId() + ") msgBody : " + seckillId);
                }
            } catch (Exception e) {
                e.printStackTrace();
                //如果出现异常，必须告知消息进行重试
                return ConsumeConcurrentlyStatus.RECONSUME_LATER;
            }
            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
        });
        consumer.start();
        System.out.println("[Consumer 已启动]");
    } catch (Exception e) {
        e.printStackTrace();
    }
    return consumer;
}
```

3.3 业务执行动作

```
/**
 * @Description:  redis 原子操作，实现库存控制，实现缓存优化
 * @Author: hubin
 * @CreateDate: 2020/11/27 22:01
 * @UpdateUser: hubin
 * @UpdateDate: 2020/11/27 22:01
 * @UpdateRemark: 修改内容
 * @Version: 1.0
 */
@Override
public HttpResult startKilledWithRedis(Long killId, String userId)
throws BaseException {
   try {
      // 1、 从缓存中获取秒杀商品数据
```

```
        TbSeckillGoods seckillGoods = (TbSeckillGoods)
redisTemplate.opsForValue().get("seckill_goods_"+killId);
        //判断
        if(seckillGoods == null){
            return
HttpResult.error(HttpStatus.SEC_GOODS_NOT_EXSISTS,"商品不存在");
        }
        if(seckillGoods.getStatus() != 1){
            return HttpResult.error(HttpStatus.SEC_NOT_UP,"商品未审
核");
        }
        if(seckillGoods.getStockCount() <= 0){
            return HttpResult.error(HttpStatus.SEC_GOODS_END,"商品
已售罄");
        }
        if(seckillGoods.getStartTimeDate().getTime() > new
Date().getTime()){
            return
HttpResult.error(HttpStatus.SEC_ACTIVE_NOT_START,"活动未开始");
        }
        if(seckillGoods.getEndTimeDate().getTime() <= new
Date().getTime()){
            return HttpResult.error(HttpStatus.SEC_ACTIVE_END,"活动
结束");
        }

        //2、利用redis原子性操作扣减库存，不需要上锁
        boolean res = reduceStock(killId);

        // 判定
        if(!res){
            throw new
BaseException(HttpStatus.SEC_GOODS_STOCK_FAIL,"扣减库存失败");
        }


        //3、异步实现（blockingQueue,disruptor,rocketMQ队列实现异步）

        //下单
        TbSeckillOrder order = new TbSeckillOrder();
        order.setSeckillId(killId);
        order.setUserId(userId);
        order.setCreateTime(new Date());
```

```
        order.setStatus("0");
        order.setMoney(seckillGoods.getCostPrice());
        // 队列实现异步下单操作
        Boolean produce =
SeckillQueue.getMailQueue().produce(order);

        if(!produce){
            throw new
BaseException(HttpStatus.SEC_GOODS_STOCK_FAIL,"下单失败");
        }

        // 设置事务状态
        seckillGoods.setStockCount(null);
        // 提交状态
        seckillGoods.setTransactionStatus(1);

        // 更新事务状态

seckillGoodsMapper.updateByPrimaryKeySelective(seckillGoods);


        return HttpResult.ok("秒杀成功");

    } catch (Exception e) {
        e.printStackTrace();
        throw new BaseException(HttpStatus.SEC_GOODS_STOCK_FAIL,"
下单失败");
    }


    //return null;
}

/**
 * @Description: 库存扣减方法
 * @Author: hubin
 * @CreateDate: 2021/1/24 22:27
 * @UpdateUser: hubin
 * @UpdateDate: 2021/1/24 22:27
 * @UpdateRemark: 修改内容
 * @Version: 1.0
 */
private boolean reduceStock(Long killId) {
```

```
   Long res =
redisTemplate.opsForValue().increment("seckill_goods_stock_" +
killId, -1);
   if(res > 0){
      // 发送消息
      //producer.asncSendMsg(killId);
      return true;
   }else if(res == 0){
      //producer.asncSendMsg(killId);
      //记录标识,表示此商品已经售卖结束

redisTemplate.opsForValue().set("seckill_goods_stock_end_"+kil
lId,"STOCK_END");
      return true;
   }else {
      // 网络异常，扣减库存发生失败

redisTemplate.opsForValue().increment("seckill_goods_stock_" +
killId, 1);

      return false;
   }

}
```

3.4 发送消息实践

```
/**
 * @Description:  redis 原子操作，实现库存控制，实现缓存优化
 * @Author: hubin
 * @CreateDate: 2020/11/27 22:01
 * @UpdateUser: hubin
 * @UpdateDate: 2020/11/27 22:01
 * @UpdateRemark: 修改内容
 * @Version: 1.0
 */
@RequestMapping("/order/kill/asyc/{killId}/{token}")
```

```
public HttpResult startKilledWithRedis(@PathVariable Long killId,
@PathVariable String token) throws BaseException{
   //判断
   if(StringUtils.isBlank(token)){
      return HttpResult.error(HttpStatus.SC_EXPECTATION_FAILED,"
用户未登录");
   }
   //获取用户数据
   FrontUser user =
seckillOrderService.getUserInfoFromRedis(token);

   //判断
   if(user == null){
      return HttpResult.error(HttpStatus.SC_EXPECTATION_FAILED,"
用户未登录");
   }

   //获取userid
   String userId = user.getId()+"";
   //发送消息
   //线程同步的调用方法，20 个等待队列，流量泄洪
   Future<Object> future = executorService.submit(new
Callable<Object>() {

      @Override
      public Object call() throws Exception {
         // 发送消息
         boolean res = producer.asncSendTransactionMsg(killId,
userId);

         // 判断消息发送成功，还是失败
         if(!res){
            throw new
BaseException(HttpStatus.SEC_GOODS_STOCK_FAIL,"消息发送失败");
         }


         return null;
      }
   });

   try {
      future.get();
```

```
   } catch (InterruptedException e) {
      e.printStackTrace();
      throw new BaseException(HttpStatus.SEC_GOODS_STOCK_FAIL,"
消息发送失败");
   } catch (ExecutionException e) {
      e.printStackTrace();
      throw new BaseException(HttpStatus.SEC_GOODS_STOCK_FAIL,"
消息发送失败");
   }

   return null;

}
```

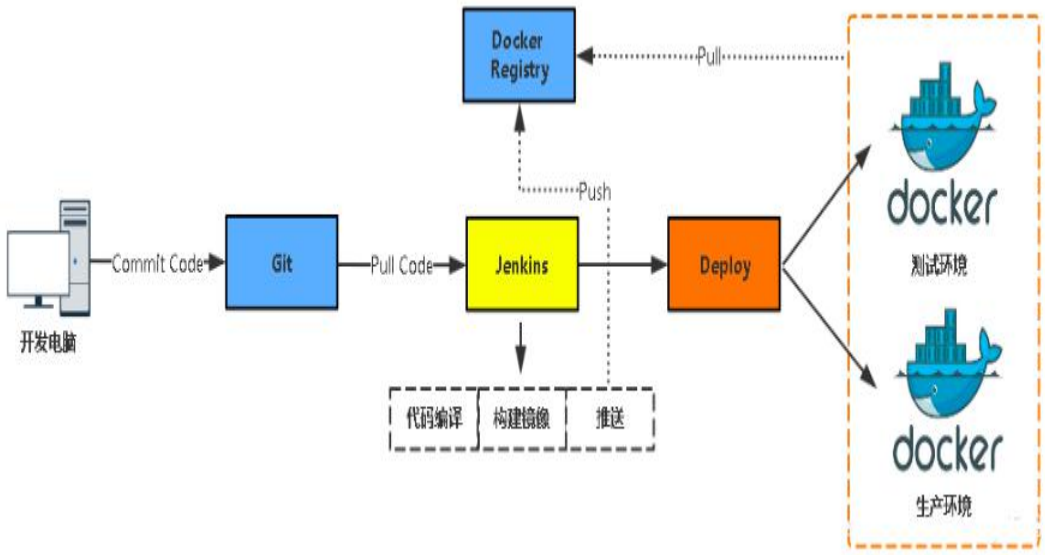
4 DevOps

4.1 DevOps 是什么？

DevOps ： 开发 + 运营（测试，运维，文档……）

→ 开发 + 运维 一种文化体系，建立的一套流水线生产架构模式； 从项目开发 ，测试，发布能够更加快速，高效； DevOps 项目能够快递迭代，频繁的更新更可靠的发生；因此基于这些问题，提出了敏捷开发的思想；

➔ 软件开发交付的流程实现自动化，可持续交付，可持续部署



4.2 我们能干什么？

开发人员：

→和运维进行深度结合，协同工作；编写一些运维的相关的代码（脚本化代码：shell、kubernetes 相关，dockerfile.....）,创建一个更好的产品；

→ 云原生架构：项目符合云原生架构体系，考虑写代码（jdk,spring,springcloud）

DevOps 工程师：

开发一些自动化的脚本，实现项目自动化发布，使得项目发布更加智能化；从而使得企业降本增效；

4.3 DevOps 三架马车

1)Jenkins

实现项目代码的编译，构建，打包镜像，push 到镜像仓库；借助一些脚本实现流水线的生产模式；

2)docker

容器化可以实现跨平台；节省环境差异性造成的影响；

3)Kubernetes

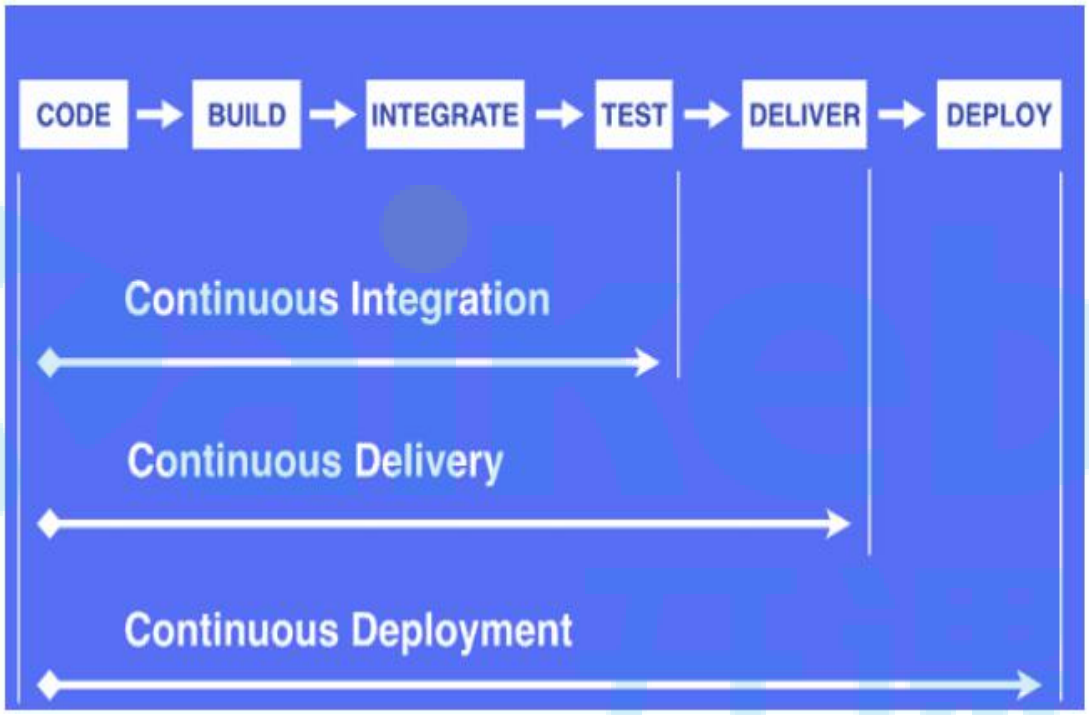
容器越来越多以后，kubernetes 构建容器的管理系统；

5 Jenkins

5.1 CI/CD

互联网软件的开发和发布，已经形成了一套标准流程，假如把开发工作流程分为以下几个阶段：

编码 --> 构建 --> 集成 --> 测试 --> 交付 --> 部署



CI： 持续集成 集成测试
CD: 持续发布，持续部署

5.2 Jenkins 部署

```
docker run \
  -u root \
  --rm \ ❶
  -d \ ❷
  -p 8080:8080 \ ❸
  -p 50000:50000 \ ❹
  -v jenkins-data:/var/jenkins_home \ ❺
  -v /var/run/docker.sock:/var/run/docker.sock \ ❻
  jenkinsci/blueocean ❼
```

❶ （可选） `jenkinsci/blueocean` 关闭时自动删除Docker容器（下图为实例）。如果您洁。

❷ （可选） `jenkinsci/blueocean` 在后台运行容器（即"分离"模式）并输出容器ID。如果中输出正在运行的此容器的Docker日志。

❸ 映射（例如"发布"） `jenkinsci/blueocean` 容器的端口8080到主机上的端口8080。第后一个代表容器的端口。因此，如果您为此选项指定 `-p 49000:8080` ，您将通过端口

5.3 Shell 脚本

Shell

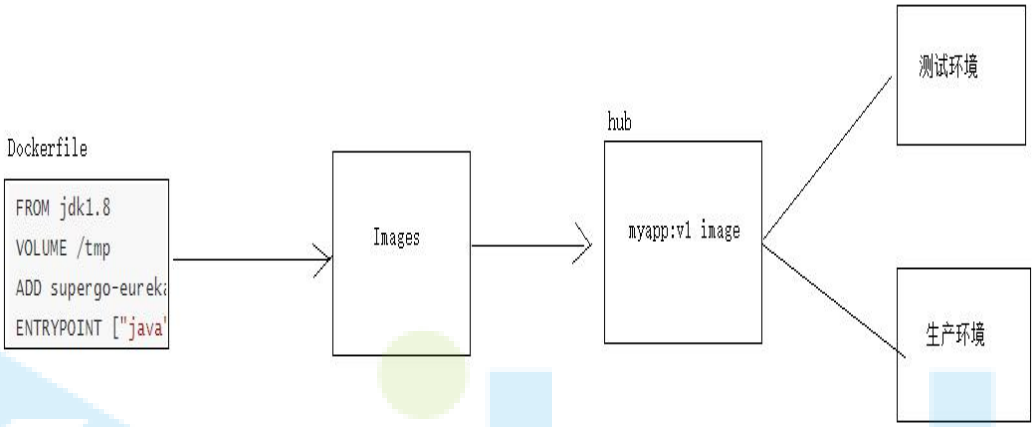
命令

```
#磁盘空间不足，导致启动失败，必须加 '#!/bin/bash' ，否则会启动不起来
#!/bin/bash
#export BUILD_ID=dontKillMe这一句很重要，这样指定了，项目启动之后才不会被Jenkins杀掉。
export BUILD_ID=dontKillMe
#Jenkins中编译好的jar位置
jar_path=/root/.jenkins/workspace/eureka-one/supergo-eureka/target
#Jenkins中编译好的jar名称
jar_name=app.jar

echo "start服务脚本开始"
dir=/opt/jenkins_jars
#将编译好的jar复制到最后指定的位置
cp  ${jar_path}/${jar_name} $dir
cd $dir
echo dir=$dir
echo ----------------------------------------
nohup java -jar $dir/${jar_name} > $dir/$(date +'%Y%m%d').log 2>&1 &
echo "start服务脚本结"
```

使用这个脚本（让 jenkins 帮助执行脚本）部署服务，但是无法解决测试环境，生成环境一致性问题；

6 Docker 容器部署

6.1 部署流程



Dockerfile – docker 指令 构建镜像，通过镜像实现服务部署；

6.2 Dockerfile

Dockerfile 是由一系列命令和参数构成的脚本，这些命令应用于基础镜像并最终创建一个新的镜像。

1、对于开发人员：可以为开发团队提供一个完全一致的开发环境；

2、对于测试人员：可以直接拿开发时所构建的镜像或者通过 Dockerfile 文件构建一个新的镜像开始工作了；

3、对于运维人员：在部署时，可以实现应用的无缝移植。

1.2 常用命令

命令	作用
FROM image_name:tag	定义了使用哪个基础镜像启动构建流程
MAINTAINER user_name	声明镜像的创建者
ENV key value	设置环境变量 (可以写多条)
RUN command	是Dockerfile的核心部分(可以写多条)
ADD source_dir/file dest_dir/file	将宿主机的文件复制到容器内，如果是一个压缩文件， 将会在复制后自动解压
COPY source_dir/file dest_dir/file	和ADD相似，但是如果有压缩文件并不能解压
WORKDIR path_dir	设置工作目录
EXPOSE port1 prot2	用来指定端口，使容器内的应用可以通过端口和外界交 互
CMD argument	在构建容器时使用，会被docker run 后的argument覆 盖
ENTRYPOINT argument	和CMD相似，但是并不会被docker run指定的参数覆盖
VOLUME	将本地文件夹或者其他容器的文件挂载到容器中

构建镜像基本逻辑：必须有有一个基础镜像（相当于安装一个软件，此软件必须依赖于一个基础软件-操作系统）

1、jdk from baseImage(centos)

2、eureka from baseImage(jdk—> centos)

6.3 JDK 镜像

JDK 镜像必须依赖 centos 操作系统的镜像，也就是说必须把 jdk 安装在操作系统中；centos 是一个容器镜像；

Dockerfile:

```
FROM hub.kaikeba.com/library/centos:v1
MAINTAINER jackhu
ADD jdk-8u261-linux-x64.tar.gz /usr/local/java
ENV JAVA_HOME /usr/local/java/jdk1.8.0_65
ENV PATH $JAVA_HOME/bin:$PATH
```

构建镜像： docker build -t jdk:v1 .

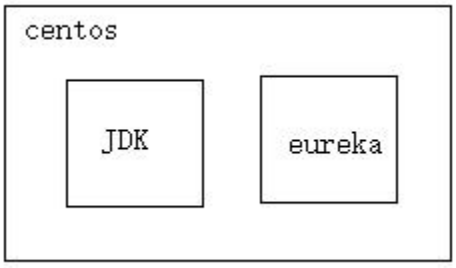
```
[root@k8s-java11-master-01 jdk]# docker build -t jdk:v1 .
Sending build context to Docker daemon  143.1MB
Step 1/5 : FROM hub.kaikeba.com/library/centos:v1
 ---> 0d120b6ccaa8
Step 2/5 : MAINTAINER jackhu
 ---> Using cache
 ---> de05362a89ac
Step 3/5 : ADD jdk-8u261-linux-x64.tar.gz /usr/local/java
 ---> 74400370424e
Step 4/5 : ENV JAVA_HOME /usr/local/java/jdk1.8.0_65
 ---> Running in 355b4947fc74
Removing intermediate container 355b4947fc74
 ---> c5f838d84200
Step 5/5 : ENV PATH $JAVA_HOME/bin:$PATH
 ---> Running in fdda17206e46
Removing intermediate container fdda17206e46
 ---> 7ff899773d65
Successfully built 7ff899773d65
Successfully tagged jdk:v1
```

验证此镜像是否 ok： docker run -di –name=myjdk jdk:v1 /bin/bash

发布一个版本： docker tag jdk:v1 hub.kaikeba.com/dev/jdk:v1
Push 一个镜像： docker push hub.kaikeba.com/dev/jdk:v1

6.4 Eureka 镜像

Eureka 镜像是一个应用程序的镜像，eureka 镜像运行起来，必须依赖一个 jdk 镜像（jdk 依赖 centos）



Dockerfile: docker build -t eureka:v1 .

```
FROM hub.kaikeba.com/dev/jdk:v1
MAINTAINER jackhu
ADD app.jar /
ENTRYPOINT ["java","-jar","/app.jar"]
```

开始构建应用镜像：

```
[root@k8s-java11-master-01 eureka]# docker build -t myeu:v1 .
Sending build context to Docker daemon  44.74MB
Step 1/4 : FROM hub.kaikeba.com/dev/jdk:v1
 ---> 7ff899773d65
Step 2/4 : MAINTAINER jackhu
 ---> Running in 55688331cca7
Removing intermediate container 55688331cca7
 ---> 6626bf1a3e30
Step 3/4 : ADD app.jar /
 ---> f509ef648566
Step 4/4 : ENTRYPOINT ["java","-jar","/app.jar"]
 ---> Running in 93ea66c41b3a
Removing intermediate container 93ea66c41b3a
 ---> 2824a32401d0
Successfully built 2824a32401d0
Successfully tagged myeu:v1
[root@k8s-java11-master-01 eureka]#
```

验证此镜像是否 ok： docker run -di –name=eu -p 9999:10086 myeu:v1 /bin/bash

发布一个版本： docker tag eu:v1 hub.kaikeba.com/dev/eu:v1
Push 一个镜像： docker push hub.kaikeba.com/dev/eu:v1

6.5 后端 dockerfile

```
<plugin>
            <groupId>com.spotify</groupId>
            <artifactId>docker-maven-plugin</artifactId>
            <version>0.4.13</version>
            <configuration>
                <!--dockerfile 指令：变成插件配置-->
                <!--用于指定镜像名称-->

<imageName>hub.kaikeba.com/kubernetes/${project.artifactId}:${project.v
ersion}</imageName>
                <!--用于指定基础镜像，相当于 Dockerfile 中的 FROM 指令-->
                <!--FROM jdk1.8:v1-->
                <baseImage>hub.kaikeba.com/library/jdk1.8:v3</baseImage>
                <!--指定工作目录-->
                <!--<workdir>/</workdir>-->
                <maintainer>ithubin@163.com</maintainer>
                <cmd>["java","-version"]</cmd>
                <!--相当于 Dockerfile 的 ENTRYPOINT 指令-->
                <!--dockerfile : entryPoint-->
```

```
<entryPoint>["java","-jar","/${project.build.finalName}.jar"]</entryPoi
nt>
                <!--指定 harbor 镜像仓库地址,指定：镜像仓库用户名，密码-->
                <serverId>my-docker-registry</serverId>
                <!--是否跳过 docker build-->
                <!--<skipDockerBuild>true</skipDockerBuild>-->
                <resources>
                    <resource>
                        <!--workdir ADD xx.jar / -->
                        <!--workdir 工作目录
-->
                        <targetPath>/</targetPath>
                        <!--用于指定需要复制的根目录，
${project.build.directory}表示 target 目录-->
                        <directory>${project.build.directory}</directory>
                        <!--用于指定需要复制的文件。
${project.build.finalName}.jar 指的是打包后的 jar 包文件
-->

<include>${project.build.finalName}.jar</include>

                    </resource>
                </resources>
                <!--使用本地镜像仓库使用-->
               <!-- <dockerHost>http://192.168.66.66:2375</dockerHost>-->
            </configuration>
        </plugin>
    </plugins>
</build>
```

指定镜像仓库地址:

```
<!--指定harbor镜像仓库地址,指定：镜像仓库用户名，密码-->
<serverId>my-docker-registry</serverId>
<!--是否跳过docker build-->
```

Jenkins 关联的镜像仓库地址：

```
<server>
  <id>my-docker-registry</id>
  <username>admin</username>
  <password>Harbor12345</password>
  <configuration>
    <email>ithubin@mail.com</email>
  </configuration>
</server>
```

6.6 Jenkins push


```
ec65b9354391: Pushing [=========================================>       ]   57.9MB/66.35MB
  [2B  [3A  [2K
ec65b9354391: Pushing [==========================================>      ]  59.57MB/66.35MB
  [2B  [3A  [2K
ec65b9354391: Pushing [==========================================>      ]  61.24MB/66.35MB
  [2B  [3A  [2K
ec65b9354391: Pushing [==========================================>      ]  62.36MB/66.35MB
  [2B  [3A  [2K
ec65b9354391: Pushing [===========================================>     ]  63.47MB/66.35MB
  [2B  [3A  [2K
ec65b9354391: Pushing [============================================>    ]  65.14MB/66.35MB
  [2B  [3A  [2K
ec65b9354391: Pushing [============================================>]  66.35MB
  [2B  [3A  [2K
ec65b9354391: Pushed
  [2B1.0-SNAPSHOT: digest: sha256:84813578bfc6ce8beb6755f83bc3bf99e0a9b146a37928afe05bee0d652a1bf2 size: {
null: null
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  33.516 s
[INFO] Finished at: 2021-01-26T15:40:03Z
[INFO] ------------------------------------------------------------------------
Waiting for Jenkins to finish collecting data
[JENKINS] Archiving /var/jenkins_home/workspace/jshop/jshop-web/pom.xml to com.jshop/jshop-web/1.0-SNAPSH(
[JENKINS] Archiving /var/jenkins_home/workspace/jshop/jshop-web/target/jshop.jar to com.jshop/jshop-web/1.
SNAPSHOT.jar
channel stopped
Finished: SUCCESS
```

7 K8s 云部署

7.1 Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: jshop-server
  name: jshop-server
  namespace: edu
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: jshop-server
      release: jshopserver
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: jshop-server
        release: jshopserver
    spec:
      containers:
      - env:
        - name: K
          value: V
        image: registry.cn-beijing.aliyuncs.com/kkb2/jshop:v7
        imagePullPolicy: Always
        livenessProbe:
          failureThreshold: 3
          initialDelaySeconds: 120
          periodSeconds: 10
```

```
            successThreshold: 1
            tcpSocket:
              port: 9000
            timeoutSeconds: 5
          name: admin-server
          readinessProbe:
            failureThreshold: 3
            initialDelaySeconds: 120
            periodSeconds: 10
            successThreshold: 1
            tcpSocket:
              port: 9000
            timeoutSeconds: 5
          resources:
            limits:
              cpu: "1"
              memory: 2Gi
            requests:
              cpu: 500m
              memory: 1Gi
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
        dnsPolicy: ClusterFirst
        imagePullSecrets:
        - name: kkb100
        restartPolicy: Always
        schedulerName: default-scheduler
        securityContext: {}
        terminationGracePeriodSeconds: 30
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: jshop-server
  name: jshopserver
  namespace: edu
spec:
  ports:
  - name: jshop-server
    port: 80
    protocol: TCP
    targetPort: 9000
```

```
  selector:
    app: jshop-server
  type: ClusterIP
```

7.2 Ingress

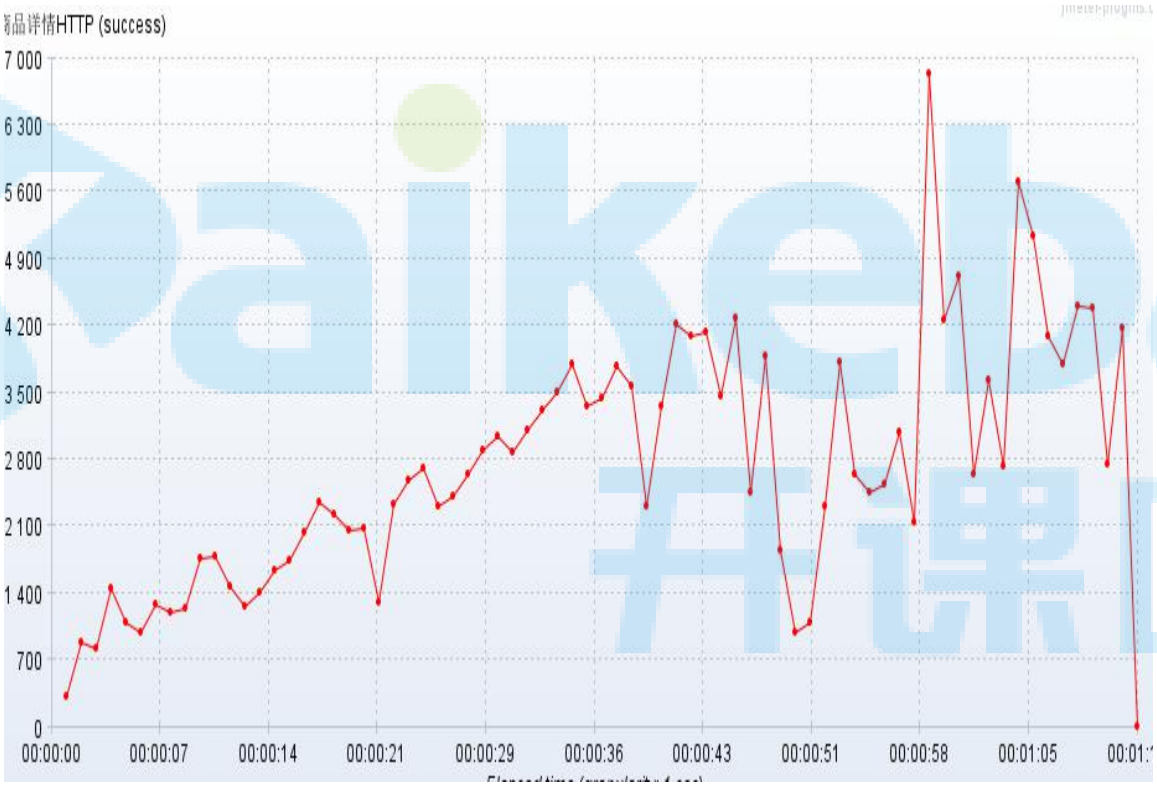
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/proxy-body-size: 5m
  name: jshop-ingress
  namespace: edu
spec:
  rules:
  - host: edu1.kaikeba.com
    http:
      paths:
      - backend:
          serviceName: jshopserver
          servicePort: 80
        path: /
  tls:
  - hosts:
    - edu1.kaikeba.com
```

7.3 动态伸缩容

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-cpu-jshopserver
  namespace: edu
spec:
  maxReplicas: 30
  metrics:
```

```
  - resource:
      name: cpu
      target:
        averageUtilization: 70
        type: Utilization
    type: Resource
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: jshop-server
```

施加压力：实现动态的扩缩容



发现目前扩容到了 4 个服务（POD）

```
root@k8s-m001 jshop]# kubectl get pod -n edu -o wide
AME                            READY   STATUS    RESTARTS   AGE     IP              NODE
ESS GATES
shop-server-5dd7776dc4-7vqsk   1/1     Running   0          5m32s   10.244.9.209    k8s-s209
shop-server-5dd7776dc4-bq9s4   1/1     Running   0          2m26s   10.244.48.155   k8s-s003
shop-server-5dd7776dc4-js7bk   1/1     Running   0          14m     10.244.116.153  k8s-s006
shop-server-5dd7776dc4-q99dj   1/1     Running   0          5m32s   10.244.25.23    k8s-s177
root@k8s-m001 jshop]#
```

一段时间后： 自动缩容

问题：

1、探针
2、promuthues ,granfana
3、push
4、微服务问题（部署到云端）