AI平台开发指南

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 时间 | 作者 | 说明 |
| v0.9 | 2018-08-29 | 彭行雄 | 初始化 |
| 2018-09-03 | 李娟 | 为评审意见修改文档；并加入数据对接DEMO |

# python项目

## 新建maven工程

在编译器IDEA中新建一个maven项目，其命名如下所示：

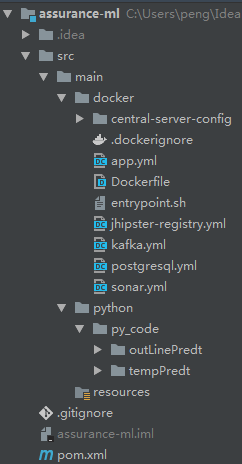
|  |
| --- |
| <groupId>com.ruijie</groupId>  <artifactId>assurance-ml</artifactId>  <version>1.0-SNAPSHOT</version> |

创建成功后删除java、test等文件夹。

docker文件夹为如下压缩包：



将docker.zip解压放到项目的src/main文件夹下，最终的目录结构为:



在main路径下新建文件夹路径python/py\_code，此目录名称固定，主要是放入后续python的代码。

**注意：需要加入.gitignore文件，用于忽略.idea和.iml文件。具体做法如下：**

**在与pom.xml同一路径下，添加名称为.gitignore文件。**

****

## 修改pom.xml

在pom.xml文件的项目信息（version）后面加入属性描述和插件配置

<modelVersion>4.0.0</modelVersion>

<groupId>com.ruijie</groupId>  
<artifactId>assurance-ml</artifactId>  
<version>1.0-SNAPSHOT</version>

（1）加入属性描述

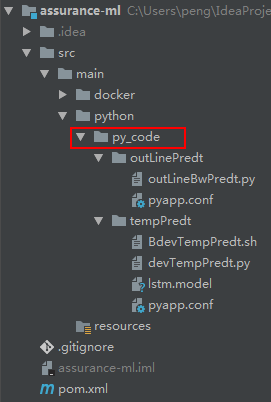
|  |
| --- |
| <properties>  <maven-resources-plugin.version>3.0.1</maven-resources-plugin.version>  <dockerfile-maven-plugin.version>1.4.3</dockerfile-maven-plugin.version>  <!-- jhipster-needle-maven-property -->  <docker.registry.url>harbor.mig.ruijie.net</docker.registry.url>  <docker.registry.prefix>rgibns-snapshot</docker.registry.prefix>  <docker.registry.username>admin</docker.registry.username>  <docker.registry.password>rgharbor</docker.registry.password> </properties> |

（2）加入插件

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-resources-plugin</artifactId>  
 <version>${maven-resources-plugin.version}</version>  
 <executions>  
 <execution>  
 <id>default-resources</id>  
 <phase>validate</phase>  
 <goals>  
 <goal>copy-resources</goal>  
 </goals>  
 <configuration>  
 <outputDirectory>target/classes</outputDirectory>  
 <useDefaultDelimiters>false</useDefaultDelimiters>  
 <delimiters>  
 <delimiter>#</delimiter>  
 </delimiters>  
 <resources>  
 <resource>  
 <directory>src/main/resources/</directory>  
 <filtering>true</filtering>  
 <includes>  
 <include>config/\*.yml</include>  
 </includes>  
 </resource>  
 <resource>  
 <directory>src/main/resources/</directory>  
 <filtering>false</filtering>  
 <excludes>  
 <exclude>config/\*.yml</exclude>  
 </excludes>  
 </resource>  
 </resources>  
 </configuration>  
 </execution>  
 <execution>  
 <id>docker-resources</id>  
 <phase>validate</phase>  
 <goals>  
 <goal>copy-resources</goal>  
 </goals>  
 <configuration>  
 <outputDirectory>target/</outputDirectory>  
 <resources>  
 <resource>  
 <directory>src/main/docker/</directory>  
 <filtering>false</filtering>  
 <excludes>  
 <exclude>\*\*/\*.yml</exclude>  
 </excludes>  
 </resource>  
 <resource>  
 <directory>src/main/python/</directory>  
 </resource>  
 </resources>  
 </configuration>  
 </execution>  
 </executions>  
 </plugin>  
 <plugin>  
 <groupId>com.spotify</groupId>  
 <artifactId>dockerfile-maven-plugin</artifactId>  
 <version>${dockerfile-maven-plugin.version}</version>  
 <executions>  
 <execution>  
 <id>tag-snapshot</id>  
 <configuration>  
 <tag>${project.version}-SNAPSHOT</tag>  
 </configuration>  
 </execution>  
 <execution>  
 <id>push-snapshot</id>  
 <configuration>  
 <tag>${project.version}-SNAPSHOT</tag>  
 </configuration>  
 </execution>  
 <execution>  
 <id>tag-latest</id>  
 <configuration>  
 <tag>latest</tag>  
 </configuration>  
 </execution>  
 <execution>  
 <id>push-latest</id>  
 <configuration>  
 <tag>latest</tag>  
 </configuration>  
 </execution>  
 </executions>  
 <configuration>  
 <contextDirectory>${project.build.directory}</contextDirectory>  
 <repository>${docker.registry.url}/${docker.registry.prefix}/${project.artifactId}</repository>  
 <tag>${project.version}</tag>  
 <contextDirectory>${project.build.directory}</contextDirectory>  
 <username>${docker.registry.username}</username>  
 <password>${docker.registry.password}</password>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

## 加入python代码

在python/py\_code目录下加入业务需要的python项目代码：



## 依赖库申明

对于部署而言，python算法是黑盒的，因此依赖库需要显式的给出，方法是在python项目的resource目录中创建requirement.txt文件，显式的申明需要使用的依赖。格式如下：

|  |
| --- |
| pystan  fbprophet  pymongo  scikit-learn==0.19  pykafka  scipy  torch==0.3.1  pyspark==2.3.1  py4j |

通常情况下只需要列出每个依赖的名称即可，表示使用最新版本的依赖库。但是有些时候还需要具体的版本号，例如pyspark，AI环境使用的spark是2.3.1版本，为了更好的兼容性体验，也建议pyspark使用2.3.1版本。

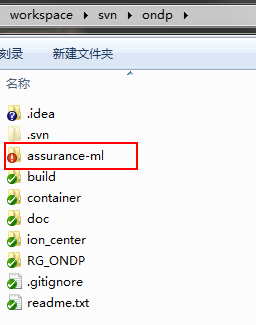
然后在安装有Python3的环境中，使用pip命令测试一下是否能正常安装（意在确认requirements.txt写法均正确），命令为：

|  |
| --- |
| pip3 install -i https://pypi.tuna.tsinghua.edu.cn/simple -r /tmp/requirements.txt |

这里给出的 <https://pypi.tuna.tsinghua.edu.cn/simple> 表示国内镜像。如果能够顺利安装成功，则每次修改requirement.txt时都请告知平台组部署人员来更新基础的docker镜像。

## 移入svn

将整个assurance-ml放入到svn项目中相关目录，例如放入到下图所示位置：



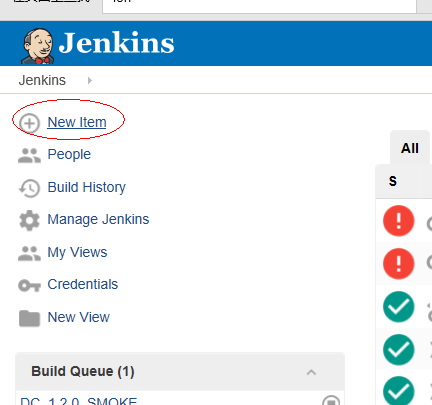
假设当前的ondp为svn根目录（统一放在svn根目录），则assurance-ml所在的路径为“svn根目录/assurance-ml”。

然后更新svn->提交svn代码。

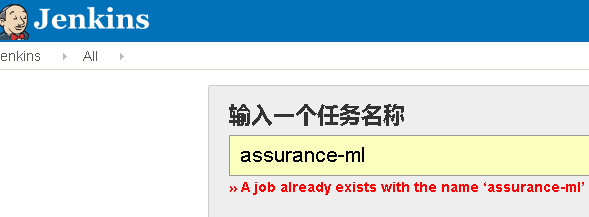
# Jenkins自动化

## 新建项目

登录到[Jenkins网站](http://172.18.103.242:8080/)，账号可找pm获取。新建一个项目

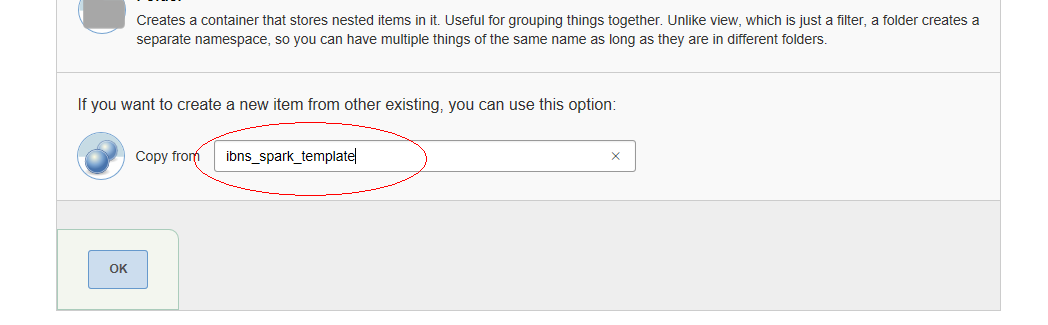


给项目命名，如果已经有同名的项目了，会提示。

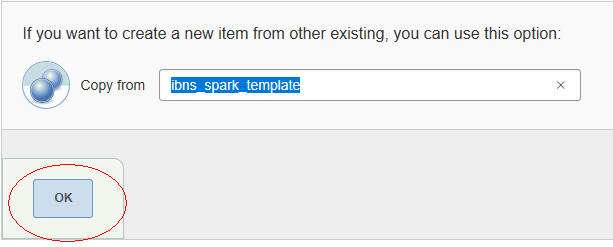


该名称可以自定义，但是要记住。

然后从ibns\_spark\_template模板拷贝



点击OK按钮进行保存



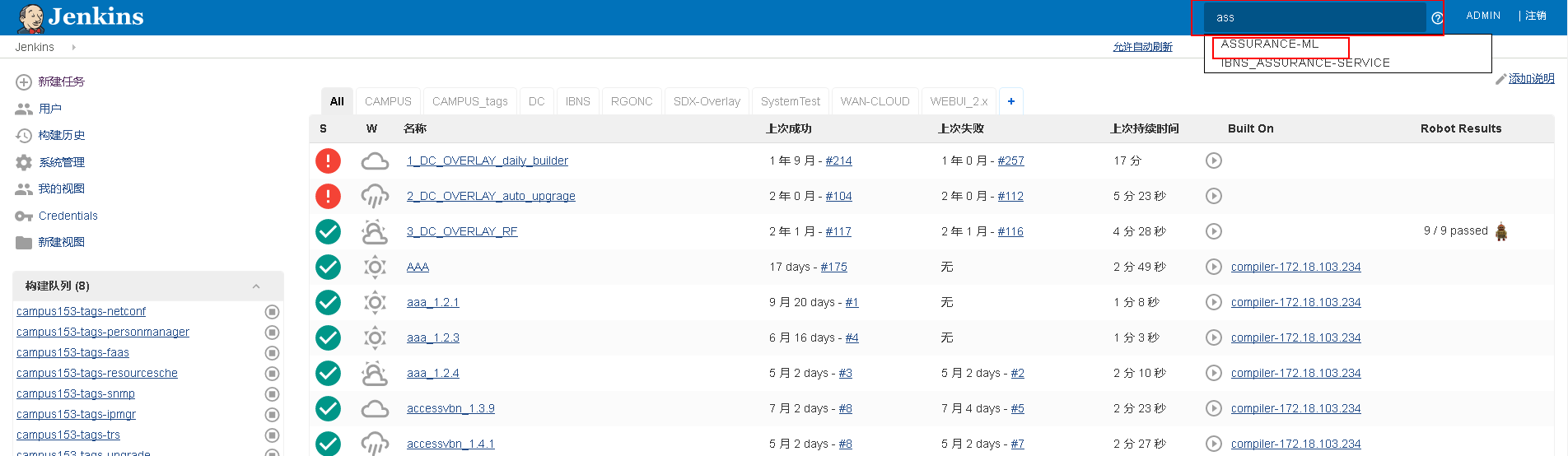
点击ok保存后，会出现如下配置界面



## 配置项目

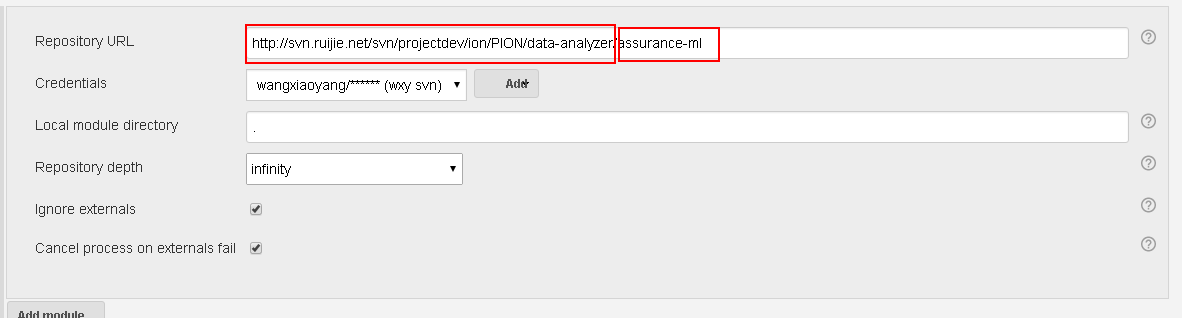
### 进入配置

找到自己的项目并点击进入





修改代码路径与svn权限，可以看到，这里的结构就是svn中提到的svn根目录/assurance-ml目录。



注：svn权限如果没有自己专属的权限，可选择如图wangxiaoyang/\*\*\*\*\*\*(wxy svn)

### 自动化打包

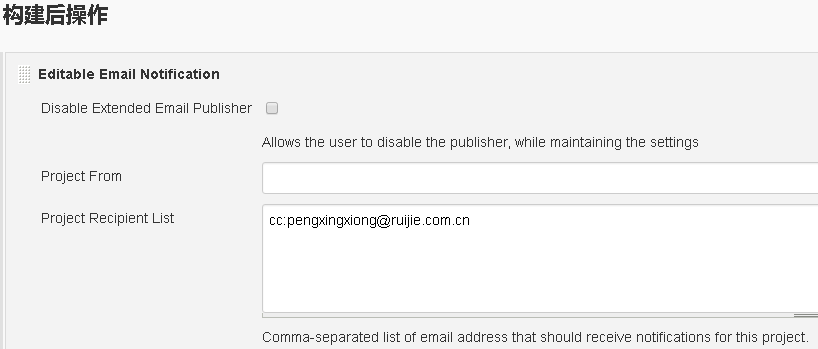
由于是python代码，不同于java/scala那样需要打成jar包，因此命令较为简单：

|  |
| --- |
| cd /home/ruijie/workspace/${JOB\_NAME}/  svn update  mvn clean  mvn validate  mvn dockerfile:build dockerfile:push |

将该命令加入到下图位置：

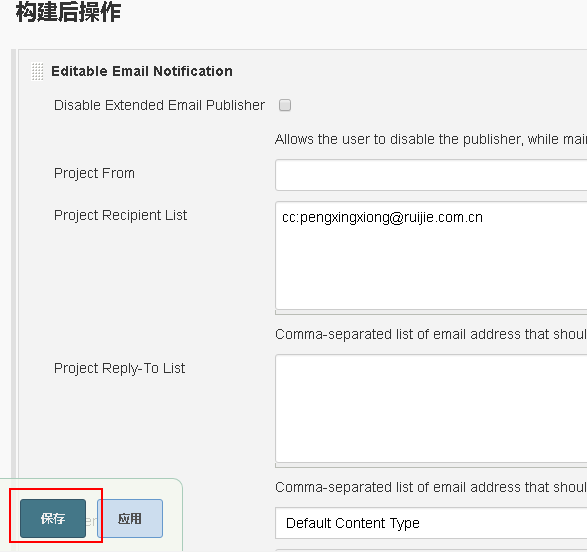


### 修改构建失败抄送对象



一般情况下，都是抄送给自己，格式为“cc:工作邮箱地址”

### 保存



保存正常进入如下页面



## 构建项目

构建项目有两种方式： SVN定时拉取构建与手动立刻构建

### SVN定时拉取构建

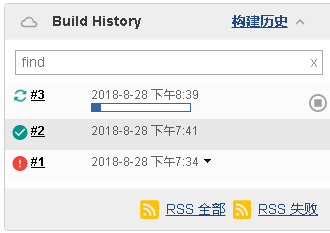
项目配置好后，每三分钟会检查一次SVN，如SVN代码有更新会自动构建，无需人工干预。

### 手动立刻构建

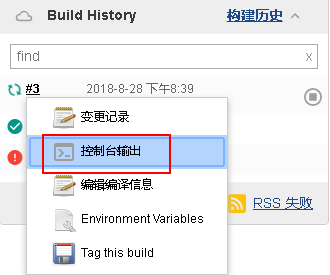


开始构建时，会出现一个构建实例。





点击进入控制台



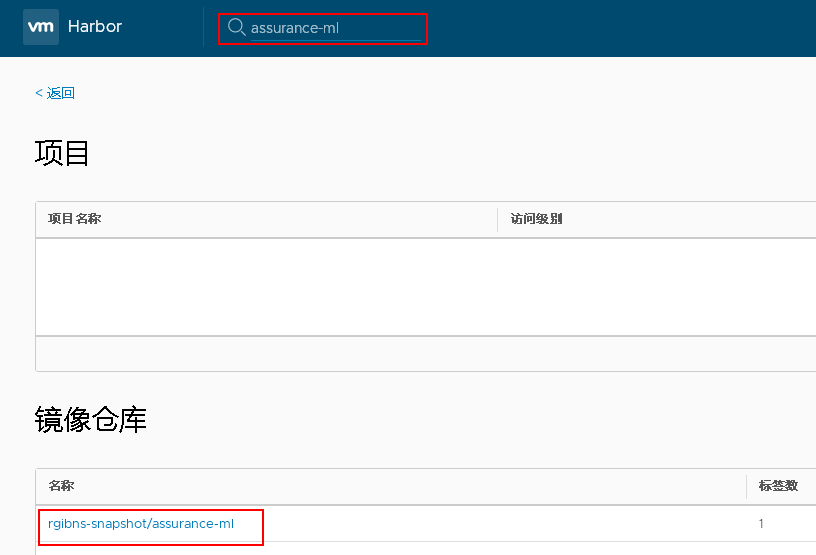
查看日志信息



可以看到最后构建成功，则表示顺利打包。

## 检查Harbor仓库

构建成功后，在pom文件中定义的[私库地址](https://172.18.135.100/harbor/sign-in)与私库镜像路径中查看上传的镜像



这里的assurance-ml是pom.xml中定义的artifactId。点击进入后界面为：



标签即为pom.xml中定义的version。即这个代码镜像为:

镜像：harbor.mig.ruijie.net/rgibns-snapshot/assurance-ml

标签：1.0-SNAPSHOT

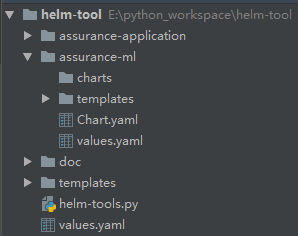
# 使用AI环境

## 配置helm包

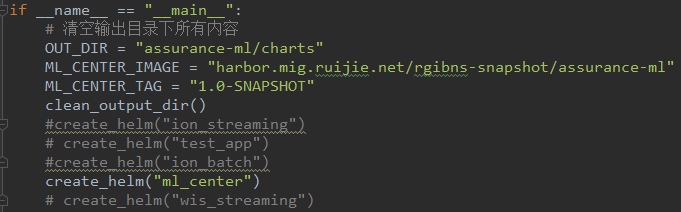
将helm-tools.zip解压到本地目录



使用编译器或者直接文件夹打开，目录结构为



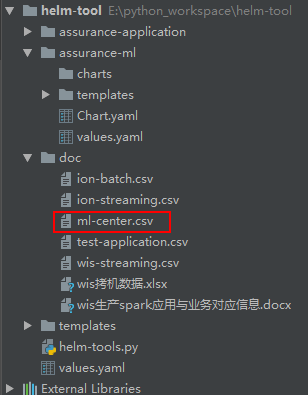
修改helm-tools.py文件，加入以下代码



表示要创建AI应用。

其中需要修改的是ML\_CENTER\_IMAGE和ML\_CENTER\_TAG。这两个值分别代表着Harbor仓库对应的镜像和标签名。请保持一致。

## 更新模板



使用EXCEL软件打开ml-center.csv文件，严格按照给出的样例格式填充。



各字段含义

|  |  |
| --- | --- |
| 字段 | 说明 |
| APPNAME | 应用名称，bml前缀表示批处理类型的ml算法，注意不要使用大写字母 |
| SCHEDULE | 时间调度，使用cron表达式 |
| SUBMIT | 提交命令，支持spark-submit/python3/python/sh |
| PYTHON\_VERSION | 使用python命令时涉及的python版本，主要是用于pyspark，可以为空。 |
| CPU | 最少使用的CPU资源数 |
| MEMORY | 最少使用的内存数 |

## 创建helm包

方式1：使用IDEA编译器运行helm-tools.py文件

方式2：使用python命令运行helm-tools.py文件

# python2 相对路径  
cd helm-tool  
python helm-tools.py

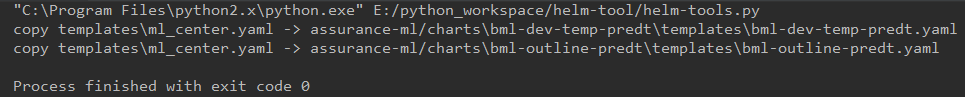
如果用Python3执行，需修改如下：

1.73和79行，print后边内容添加()

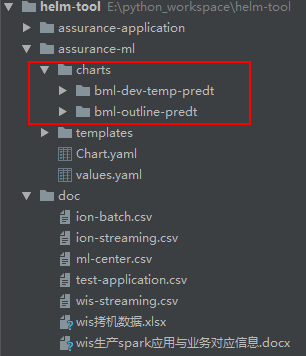
2.49行csv\_iter.next()改为next(csv\_iter)

# python3 相对路径  
cd helm-tool  
python3 helm-tools.py

成功执行后会在控制台打印如下信息：



表示在模板中的应用都成功创建了，同时在assurance-ml/charts目录中可以看到创建的helm文件：



## 上传Helm包

使用ftp将assurance-ml上传到具有Helm服务（输入helm，能够输出帮助信息）的节点的/root/charts目录，如果/root/charts目录已经存在assurance-ml目录，则进入到/root/charts/assurance-ml/charts中删除与本地重复的文件夹。

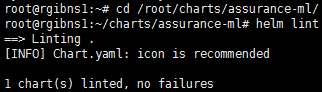
## Helm部署

### 格式检查

进入/root/charts/assurance-ml目录对assurance-ml进行格式检查。

|  |
| --- |
| cd /root/charts/assurance-ml/  helm lint |

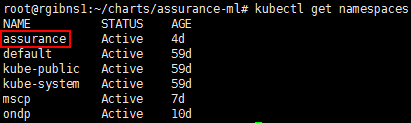
如果出现以下信息表示格式正确



### 命名空间检查

|  |
| --- |
| kubectl get namespaces |

使用该命令能够查看所有的命名空间，请从输出结果中确认存在assurance命名空间



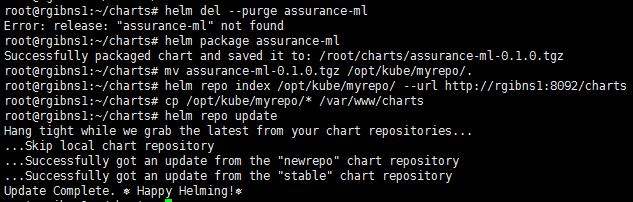
如果assurance命名空间不存在，则要创建该命名空间。

|  |
| --- |
| kubectl create namespace assurance |

### 上传到Helm仓库

|  |
| --- |
| cd /root/charts  helm del --purge assurance-ml  helm package assurance-ml  mv assurance-ml-0.1.0.tgz /opt/kube/myrepo/.  helm repo index /opt/kube/myrepo/ --url http://rgibns1:8092/charts  cp /opt/kube/myrepo/\* /var/www/charts  helm repo update |

出现如下信息表示上传成功

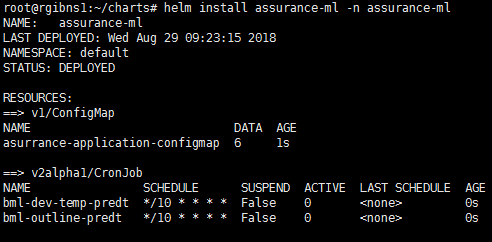


第二行有个错误，这个主要是因为之前未创建过assurance-ml的helm，仓库中就没有，可以忽略该报错。

### 部署helm包

|  |
| --- |
| helm install assurance-ml -n assurance-ml --set global.cluster.IP1=\*.\*.\*.\*,global.cluster.IP2=\*.\*.\*.\*,global.cluster.IP3=\*.\*.\*.\* |

出现如下信息表示部署成功



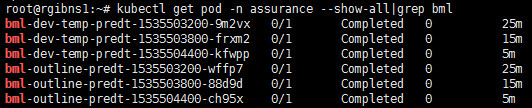
## 查看运行结果

Helm部署本质上是K8s的容器化，通过查看容器日志获取到运行结果。因为上述两个任务都是10分钟的定时任务，因此在十分钟后使用以下命令来获取日志。

### 获取运行的pod

|  |
| --- |
| kubectl get pod -n assurance --show-all|grep bml |

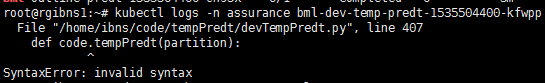
使用该命令能够查看所有已经完成的定时任务，会输出以下结果



最后一列的25m表示该pod是25分钟前创建的。我们查看其日志，则是将第一列的pod名称作为查看对象。

|  |
| --- |
| kubectl logs -n assurance bml-dev-temp-predt-1535504400-kfwpp |

输出结果为：

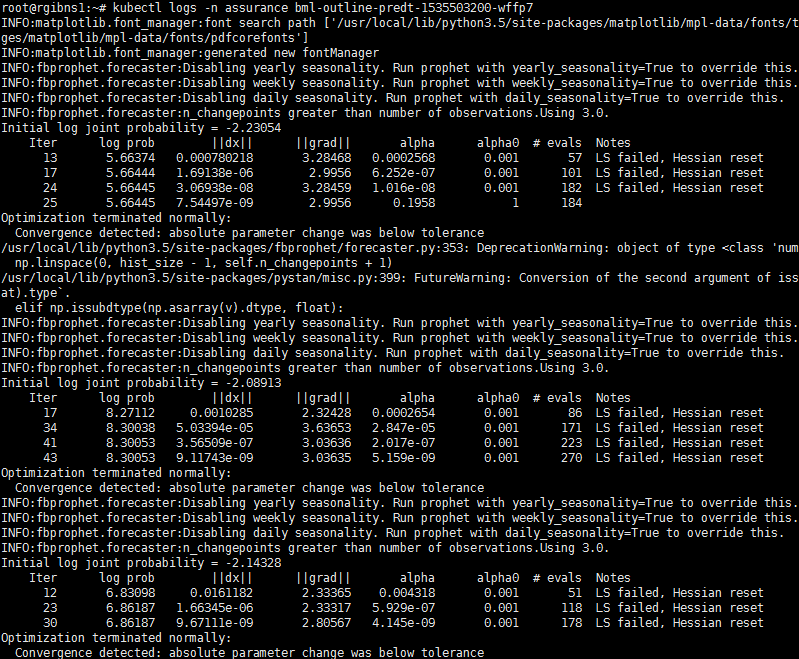


这个可以看到代码有误。则可以从本地修改代码重新使用svn自动化打包。

现在查看另外一个pod：

|  |
| --- |
| kubectl logs -n assurance bml-outline-predt-1535503200-wffp7 |

输出结果为：



可以发现，该算法运行正常，且正常结束了。

建议：在算法中多输出日志有助于部署时的错误检查。

## 部署更新

对于开发调试而言，常常需要多次部署和调试，调试的方法为以下描述

### 更新代码镜像

更新python代码到svn后，从Jenkins中查看是否完成了新的自动化打包。



并且在Harbor仓库中查看是否更新到最新的时间（时间与真实时间不一致，但是能够从时间变化中看出已更新了）。



在定时任务的下次任务到达后，将自动从Harbor仓库中拉取最新的代码镜像使用。

## 数据库连接使用

### Mongodb

此处介绍使用Python连接mongodb，并且介绍查询和更新使用方法。



#### 连接

# 连接mongodb  
mongodbhosts = "172.31.159.11:23001,172.31.159.12:23002,172.31.159.13:23003"  
connection = pymongo.MongoClient('mongodb://' + mongodbhosts)  
# 获取数据库  
db = connection.ion  
# 获取Collection  
coll = db['outLineSumInfo']

#### 查询

# 条件查询Collection数据  
dc\_list = list(coll.find({'ts': {'$gte': twelve\_hour\_ago, '$lte': now\_time}, 'count': {"$gt": 0}}, {'ts': 1, 'items': 1}).sort("ts", 1))

#### 更新

coll.update\_one({"\_id": zone\_id\_map["\_id"]}, {"$set": {"items": zone\_id\_map["items"]}})

### Kafka

Python连接kafka的标准库，有kafka-python和pykafka。kafka-python使用的人多是比较成熟的库，但kafka-python并没有zk的支持。pykafka是Samsa的升级版本，使用samsa连接zookeeper，生产者直接连接kafka服务器列表，消费者才用zookeeper。

1. Pykafka





1. kafka-python





以下以kafka-python举例：

#### 消费者

#consumer客户端

consumer = KafkaConsumer('dev-terminal', group\_id='test\_group', bootstrap\_servers=['172.24.26.66:9092', '172.24.26.67:9092', '172.24.26.68:9092'])

#接收生产者的消息

**for** msg **in** consumer:  
 print("%s:%d:%d: key=%s value=%s" % (msg.topic, msg.partition,msg.offset, msg.key, msg.value))

#producer客户端

producer = KafkaProducer(bootstrap\_servers=['172.24.26.66:9092', '172.24.26.67:9092', '172.24.26.68:9092'])

#选择topic

topic = 'dev-terminal'

#生产者生成数据

producer.send(topic, str(n).encode())

#关闭生产者客户端

producer.close()