Kubernetes 诞生之初是为在线应用服务的，或者说 long-running 类型的应用服务。但是随着越累越多的企业开始拥抱 Kubernetes，批处理的需求也逐渐显现出来。那么到底什么是批处理呢?

简单来说，所有有明确结束标志的应用都可以统称为批处理应用。比如大数据领域就将作业 (Application) 分为两种：实时处理和批处理，其中批处理就是类似于 MapReduce 这种作业。

Kubernetes 对于批处理作业提供了两种 API 对象参考：Job 和 CronJob，从名字我们也可以看出来 CronJob 就是 Job 的定时调度。

1. Job

1.1 运行一个 Job Demo

下面是一个非常简单的 Job 的描述文件，这个 Job 会通过 perl 计算 pi 的小数点后两千位数，并输出。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      restartPolicy: Never
  backoffLimit: 4
```

同样地，我们还是使用 `kubectl apply` 去运行这个 Job，然后通过 `kubectl get` 查看一些概览信息。

```
$ kubectl apply -f pi-job.yaml -n imooc
job.batch/pi created
$ kubectl get jobs -n imooc
NAME   COMPLETIONS   DURATION   AGE
pi     0/1           20s        20s
```

我们可以看到 Job 的概览信息包括：

- NAME：Job 的名字；

- COMPLETIONS：是否完成，因为 Job 也有可能包含多个容器，或者说 Task，所以这里 COMPLETIONS 的表示左边是完成的 task 数，右边是 task 总数。
- DURATION：Job 的作业持续时间；
- AGE：Job 的存活时间，关于 DURATION 和 AGE 的区别，我们过 20s 再看一下就能看出来。

```
$ kubectl get jobs -n imooc
NAME    COMPLETIONS    DURATION    AGE
pi      1/1            44s         3m42s
```

没错，如上显示，DURATION 为 task 执行的时间，而 AGE 为 Job 存活的时间，Job 执行完之后 Job 对象还是存在的。

安装惯例，我们还要通过 `kubectl describe jobs` 查看一下这个 Job 运行起来都包含哪些信息。

```
$ kubectl describe jobs pi -n imooc
Name:           pi
Namespace:      imooc
Selector:       controller-uid=63385de0-7da9-11ea-a328-00163e16aee6
Labels:         controller-uid=63385de0-7da9-11ea-a328-00163e16aee6
                job-name=pi
Annotations:    kubectl.kubernetes.io/last-applied-configuration:
                  {"apiVersion":"batch/v1","kind":"Job","metadata":
{"annotations":{},"name":"pi","namespace":"imooc"},"spec":
{"backoffLimit":4,"template":{"...
Parallelism:    1
Completions:    1
Start Time:     Tue, 14 Apr 2020 01:08:25 +0800
Completed At:   Tue, 14 Apr 2020 01:09:09 +0800
Duration:       44s
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=63385de0-7da9-11ea-a328-00163e16aee6
           job-name=pi
  Containers:
   pi:
    Image:      perl
    Port:       <none>
    Host Port:  <none>
    Command:
      perl
      -Mbignum=bpi
      -wle
      print bpi(2000)
    Limits:
      cpu:     100m
      memory:  200Mi
    Requests:
      cpu:        100m
      memory:     200Mi
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From            Message
```

```
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  7m2s  job-controller  Created pod: pi-wsgmm
```

上面的 Job 的描述信息中主要包括：

- 基本信息：包括名字、标签（labels）、注释（annotations）等；
- Parallelism：并行度，这个下一小节再细说；
- Completions：完成的 Task 个数；
- Duration：Task 执行持续时间；
- Events：主要包括创建 Pod 的事件信息，因为 Pod 作为 Kubernetes 的基本调度单温，Job 的执行最后也是通过 Pod 来运行的，对于这个示例要查看最后的运行结果，我们可以通过 `kubectl logs` 来查看 Pod 的日志。

```
$ kubectl logs pi-wsgmm -n imooc
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555964462294895493038196442881097566593344612847564823378678316527120190914564856692346034861045432664821339360726024914127372458700660631558817488152092096282925409171536436789259036001133053054882046652138414695194151160943305727036575959195309218611738193261179310511854807446237996274956735188575272489122793818301194912983367336244065664308602139494639522473719070217986094370277053921717629317675238467481846766940513200056812714526356082778577134275778960917363717872146844090122495343014654958537105079227968925892354201995611212902196086403441815981362977477130996051870721134999999837297804995105973173281609631859502445945534690830264252230825334468503526193118817101000313783875288658753320838142061717766914730359825349042875546873115956286388235378759375195778185778053217122680661300192787661119590921642019893809525720106548586327886593615338182796823030195203530185296899577362259941389124972177528347913151557485724245415069595082953311686172785588907509838175463746493931925506040092770167113900984882401285836160356370766010471018194295559619894676783744944825537977472684710404753464620804668425906949129331367702898915210475216205696602405803815019351125338243003558764024749647326391419927260426992279678235478163600934172164121992458631503028618297455570674983850549458858692699569092721079750930295532116534498720275596023648066549911988183479775356636980742654252786255181841757467289097777279380008164706001614524919217321721477235014144197356854816136115735255213347574184946843852332390739414333454776241686251898356948556209921922218427255025425688767179049460165346680498862723279178608578438382796797668145410095388378636095068006422512520511739298489608412848862694560424196528502221066118630674427862203919494504712371378696095636437191728746776465757396241389086583264599581339047802759015
```

1.2 编写 Job 对象描述

还是以下面这个简单的 Job 描述文件为例看一下 Job 对象的 yaml 文件或者说 spec 如何编写。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
```

```
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      restartPolicy: Never
  backoffLimit: 4
```

Job 对象的描述主要包括：

- **apiVersion**：batch/v1；
- **kind**：Job；
- **metadata**：比如 name，labels 等；
- **spec**：主要信息都包含在 spec 中：
  - ***template***：唯一必填字段，Pod 模板，定义和 Pod 的编写一致，除了不需要 apiVersion 和 kind；
  - ***selector***：表示 Pod 选择器，默认空缺即可。

下面再介绍一下 Job 的并行度。Job 可以用来运行三种类型的任务，包括：

- **非并行任务**：一般情况下，只会启动一个 Pod，Pod 成功结束就表示 Job 正常完成了。
- **带有固定 completion 数目的并行任务**：spec.completions 定义 Job 至少要完成的 Pod 数据，即 Job 的最小完成数。
- **具有工作队列的并行任务**：通过参数 spec.parallelism 指定一个 Job 在任意时间最多可以启动运行的 Pod 数。

1.3 Job 结束和清理

当 Job 完成时，为了方便查看任务执行状态或者日志，Job 创建的 Job 和 Pod 对象一般情况下不会被自动清理。我们可以通过命令 `kubectl delete jobs` 来删除指定的 Job，这样 Job 以及连带的 Pod 都会被删除。

1.4 Job 自动清理

很多情况下，Job 结束了之后我们是期望可以清理掉的，因为残留的 Job 对象会额外增加 Kubernetes 的 ApiServer 的压力。那么如何自动清理结束的 Job 呢?

1. 通过上层控制器来清理，比如 CronJob。
2. TTL：引入 TTL 控制器，TTL 是 Time To Live 的简称，也就是存活时间。很多存储系统中都有这么一个叫做 TTL 的参数。要使用 TTL 控制器非常简单，只需要在 Job 的 spec 中增加参数 `ttlSecondsAfterFinished` 即可，这个参数的含义很明显，就是 Job 结束之后的存活时间。下面是一个添加了该参数的 Job 资源文件示例。

   ```
   apiVersion: batch/v1
   kind: Job
   metadata:
     name: pi-with-ttl
   ```

```
spec:
  ttlSecondsAfterFinished: 100
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

3. CronJob

在正式介绍 Kubernetes 的 CronJob 之前，我们先介绍一下 Linux 系统的 Crontab，对 Linux 熟悉的同学肯定都使用过，简而言之，Crontab 可以用来设置定时和周期性的任务。Crontab 常用命令如下：

```
crontab [-u username]
        -e  (编辑任务表)
        -l  (列出任务表)
        -r  (删除任务表)
```

我们可以通过命令 `crontab -e` 进入当前用户的任务表编辑页面，每行是一条命令，格式为时间 + 任务。其中时间共有五个域，分别是分、时、日、月、周五种，任务可以是一个 shell 命令或者一个可执行程序。其中对时间的操作符有四种：

- `*` 取值范围内的所有数字
- `/` 每过多少个数字
- `-` 从 X 到 Z
- `,` 一组数字集合

下面举几个具体的任务例子。

每分钟都执行一次 job。

每小时的第 3 和第 30 分钟执行一次 job。

在上午 8 点到 11 点的第 3 和第 15 分钟执行一次 job。

每隔两天的上午 8 点到 11 点的第 3 和第 15 分钟执行一次 job。

2.1 创建 CronJob

下面正式开始介绍 Kubernetes 的 CronJob API。CronJob 和 Linux 的 Crontab 非常类似，只不过 CronJob 的周期性任务是相对于整个 Kubernetes 集群而言的，而 Crontab 执行的任务被限定在一台 Linux 机器上。我们先看一个 CronJob 示例。

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
    name: cronjob-demo
spec:
    schedule: "*/2 * * * *"
    jobTemplate:
```

```
        spec:
            template:
                spec:
                    containers:
                    - name: busybox
                        image: busybox
                        args:
                        - /bin/sh
                        - -c
                        - date; echo Hello from the Kubernetes cluster
                    restartPolicy: OnFailure
```

这个 CronJob 做的事情非常简单，每隔 2 分钟输出当前时间和一串文本信息 “Hello from the Kubernetes cluster”。下面我们部署一下来看看效果。

```
➜  kubectl apply -f cronjob-demo.yaml
cronjob.batch/cronjob-demo created
```

和其他 API 对象一样，我们通过 `kubectl get cronjob` 来查看我们刚刚部署的 CronJob。

```
➜  kubectl get cronjob
NAME           SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
cronjob-demo   */2 * * * *   False     0        <none>          39s
➜  kubectl get cronjob
NAME           SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
cronjob-demo   */2 * * * *   False     1        64s             2m3s
```

第一次查看的时候可以看到 `LAST SCHEDULE` 字段为 `<none>` 就表示没有被调度，然后过了 2 分钟再次查看可以看到上一次的调度时间。

下面我们再通过 `kubectl describe cronjob` 来查看一下 CronJob 的明细信息。

```
➜ kubectl describe cronjob cronjob-demo
Name:                          cronjob-demo
Namespace:                     default
Labels:                        <none>
Annotations:                   kubectl.kubernetes.io/last-applied-configuration:

{"apiVersion":"batch/v1beta1","kind":"CronJob","metadata":{"annotations":
{},"name":"cronjob-demo","namespace":"default"},"spec":{"jobTempl...
Schedule:                      */2 * * * *
Concurrency Policy:            Allow
Suspend:                       False
Successful Job History Limit:  3
Failed Job History Limit:      1
Starting Deadline Seconds:     <unset>
Selector:                      <unset>
Parallelism:                   <unset>
Completions:                   <unset>
Pod Template:
```

```
  Labels:  <none>
  Containers:
   busybox:
    Image:      busybox
    Port:       <none>
    Host Port:  <none>
    Args:
      /bin/sh
      -c
      date; echo Hello from the Kubernetes cluster
    Environment:     <none>
    Mounts:          <none>
  Volumes:           <none>
Last Schedule Time:  Sat, 23 May 2020 15:46:00 +0800
Active Jobs:         cronjob-demo-1590219720, cronjob-demo-1590219840, cronjob-
demo-1590219960
Events:
  Type    Reason            Age    From                Message
  ----    ------            ----   ----                -------
  Normal  SuccessfulCreate  4m33s  cronjob-controller  Created job cronjob-demo-
1590219720
  Normal  SuccessfulCreate  2m33s  cronjob-controller  Created job cronjob-demo-
1590219840
  Normal  SuccessfulCreate  33s    cronjob-controller  Created job cronjob-demo-
1590219960
```

我们可以在 Events 中看到每隔两分钟，CronJob 对象就会创建出来一个新的 Job。

2.2 删除 CronJob

删除 CronJob 和删除其他资源类似。

```
➜ kubectl delete cronjob cronjob-demo
cronjob.batch "cronjob-demo" deleted
```

2.3 Spec 说明

CronJob 的资源文件编写主要包括：

- **apiVersion**：batch/v1beta1
- **kind**：cronjob
- **metadata**：一些元信息，比如 name 之类的
- **spec**：CronJob 的主要信息都在 spec 域下
  - ***schedule***：调度策略，格式遵从 Linux 的 Cron 标准
  - ***jobTemplate***：任务模板，和 Job API 的语法完全一样，只不过缺少一些 apiVersion 和 kind 等信息
  - ***startingDeadlineSeconds***：可选，表示任务如果由于某种原因错过了调度时间，开始该任务的截止时间的秒数。过了截止时间，CronJob 就不会再调度任务了，这种任务被统计为失败任务。如果该域没有声明，那么任务就没有最后期限。
  - ***concurrencyPolicy***：可选，定义任务执行时发生重叠如何处理，支持下面三种方式：

    - Allow：允许并发任务执行。默认选项为 Allow。
    - Forbid：不允许并发任务执行，也就是说如果新任务的执行时间到了而老的任务还没有执行完，则不会执行新的任务。这种情况在某些情况下是必要的，比如多个任务同时操作一个共享资源时可能出错。
    - Replace：如果新任务的执行时间到了而老任务没有执行完，CronJob 会用新的任务替换当前正在运行的任务。
  - ***suspend***：可选，如果设置为 `true`，后续发生的执行都会被挂起。这个设置对已经开始的执行不起作用。默认关闭。
- **successfulJobsHistoryLimit**：可选，表示多少执行完成的任务会被保留，默认值为 3。
- **failedJobHistoryLimit**：可选，表示多少执行失败的任务会被保留，默认值为 1。有的时候保留执行失败的任务对于我们排查任务失败的原因比较有用。

3. 总结

本篇文章介绍了 Kubernetes 中的批处理调度 Job 和 CronJob。尽管 Kubernetes 的主要应用场景是 Long-Running 的应用，但是某些情况下批处理调度还是需要的，比如我们通过 Job 去初始化环境，通过 CronJob 去定时清理集群中的某些资源等。

}