22 | 撬动离线业务：Job与CronJob

22 | 撬动离线业务：Job与CronJob

张磊 2018-10-12

19:30

讲述：张磊 大小：8.94M

你好，我是张磊。今天我和你分享的主题是：撬动离线业务之 Job 与 CronJob。

在前面的几篇文章中，我和你详细分享了 Deployment、StatefulSet，以及 DaemonSet 这三个编排概念。你有没有发现它们的共同之处呢？

实际上，它们主要编排的对象，都是“在线业务”，即：Long Running Task（长作业）。比如，我在前面举例时常用的 Nginx、Tomcat，以及 MySQL 等等。这些应用一旦运行起来，除非出错或者停止，它的容器进程会一直保持在 Running 状态。

但是，有一类作业显然不满足这样的条件，这就是“离线业务”，或者叫作 Batch Job（计算业务）。这种业务在计算完成后就直接退出了，而此时如果你依然用 Deployment 来管理这种业务的话，就会发现 Pod 会在计算结束后退出，然后被

Deployment Controller 不断地重启；而像“滚动更新”这样的编排功能，更无从谈起了。

所以，早在 Borg 项目中，Google 就已经对作业进行了分类处理，提出了 LRS（Long Running Service）和 Batch Jobs 两种作业形态，对它们进行“分别管理”和“混合调度”。

不过，在 2015 年 Borg 论文刚刚发布的时候，Kubernetes 项目并不支持对 Batch Job 的管理。直到 v1.4 版本之后，社区才逐步设计出了一个用来描述离线业务的 API 对象，它的名字就是：Job。

Job API 对象的定义非常简单，我来举个例子，如下所示：

```
apiVersion: batch/v1
kind: Job
metadata:
name: pi
spec:
template:
spec:
containers:
- name: pi
image: resouer/ubuntu-bc
command: ["sh", "-c", "echo 'scale=10000; 4*a(1)' | bc -l "]
restartPolicy: Never
backoffLimit: 4
```

□复制代码

此时，相信你对 Kubernetes 的 API 对象已经不再陌生了。在这个 Job 的 YAML 文件里，你肯定一眼就会看到一位“老熟人”：Pod 模板，即 spec.template 字段。

在这个 Pod 模板中，我定义了一个 Ubuntu 镜像的容器（准确地说，是一个安装了 bc 命令的 Ubuntu 镜像），它运行的程序是：

```
echo "scale=10000; 4*a(1)" | bc -l
```

□复制代码

其中，bc 命令是 Linux 里的“计算器”；-l 表示，我现在要使用标准数学库；而 a(1)，则是调用数学库中的 arctangent 函数，计算 atan(1)。这是什么意思呢？

中学知识告诉我们：tan(π/4) = 1。所以，4*atan(1)正好就是π，也就是3.1415926...。

> *备注：如果你不熟悉这个知识也不必担心，我也是在查阅资料后才知道的。*

所以，这其实就是一个计算π值的容器。而通过 scale=10000，我指定了输出的小数点后的位数是 10000。在我的计算机上，这个计算大概用时 1 分 54 秒。

但是，跟其他控制器不同的是，Job 对象并不要求你定义一个 spec.selector 来描述要控制哪些 Pod。具体原因，我马上会讲解到。

现在，我们就可以创建这个 Job 了：

```
$ kubectl create -f job.yaml
```

□复制代码

在成功创建后，我们来查看一下这个 Job 对象，如下所示：

```
$ kubectl describe jobs/pi
Name: pi
Namespace: default
Selector: controller-uid=c2db599a-2c9d-11e6-b324-0209dc45a495
Labels: controller-uid=c2db599a-2c9d-11e6-b324-0209dc45a495
job-name=pi
Annotations: <none>
Parallelism: 1
Completions: 1
..
Pods Statuses: 0 Running / 1 Succeeded / 0 Failed
Pod Template:
Labels: controller-uid=c2db599a-2c9d-11e6-b324-0209dc45a495
job-name=pi
Containers:
...
Volumes: <none>
Events:
FirstSeen LastSeen Count From SubobjectPath Type Reason Message
```

```
--------- -------- ----- ---- ------------- -------- ------ -------
1m 1m 1 {job-controller } Normal SuccessfulCreate Created pod: pi-rq5rl
```

□复制代码

可以看到，这个 Job 对象在创建后，它的 Pod 模板，被自动加上了一个 controller-uid=< 一个随机字符串 > 这样的 Label。而这个 Job 对象本身，则被自动加上了这个 Label 对应的 Selector，从而 保证了 Job 与它所管理的 Pod 之间的匹配关系。

而 Job Controller 之所以要使用这种携带了 UID 的 Label，就是为了避免不同 Job 对象所管理的 Pod 发生重合。需要注意的是，**这种自动生成的 Label 对用户来说并不友好，所以不太适合推广到 Deployment 等长作业编排对象上。**

接下来，我们可以看到这个 Job 创建的 Pod 进入了 Running 状态，这意味着它正在计算 Pi 的值。

```
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
pi-rq5rl 1/1 Running 0 10s
```

□复制代码

而几分钟后计算结束，这个 Pod 就会进入 Completed 状态：

```
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
pi-rq5rl 0/1 Completed 0 4m
```

□复制代码

这也是我们需要在 Pod 模板中定义 restartPolicy=Never 的原因：离线计算的 Pod 永远都不应该被重启，否则它们会再重新计算一遍。

> *事实上，restartPolicy 在 Job 对象里只允许被设置为 Never 和 OnFailure；而在 Deployment 对象里，restartPolicy 则只允许被设置为 Always。*

此时，我们通过 kubectl logs 查看一下这个 Pod 的日志，就可以看到计算得到的 Pi 值已经被打印了出来：

```
$ kubectl logs pi-rq5rl
3.14159265358979323846264338327 9...
```

□复制代码

这时候，你一定会想到这样一个问题，**如果这个离线作业失败了要怎么办？**

比如，我们在这个例子中**定义了 restartPolicy=Never，那么离线作业失败后 Job Controller 就会不断地尝试创建一个新 Pod**，如下所示：

```
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
pi-55h89 0/1 ContainerCreating 0 2s
pi-tqbcz 0/1 Error 0 5s
```

复制代码

可以看到，这时候会不断地有新 Pod 被创建出来。

当然，这个尝试肯定不能无限进行下去。所以，我们就在 Job 对象的 spec.backoffLimit 字段里定义了重试次数为 4（即，backoffLimit=4），而这个字段的默认值是 6。

需要注意的是，Job Controller 重新创建 Pod 的间隔是呈指数增加的，即下一次重新创建 Pod 的动作会分别发生在 10 s、20 s、40 s ...后。

而如果你**定义的 restartPolicy=OnFailure，那么离线作业失败后，Job Controller 就不会去尝试创建新的 Pod。但是，它会不断地尝试重启 Pod 里的容器。**这也正好对应了 restartPolicy 的含义（你也可以借此机会再回顾一下第 15 篇文章《深入解析 Pod 对象（二）：使用进阶》中的相关内容）。

如前所述，当一个 Job 的 Pod 运行结束后，它会进入 Completed 状态。但是，如果这个 Pod 因为某种原因一直不肯结束呢?

在 Job 的 API 对象里，有一个 spec.activeDeadlineSeconds 字段可以设置最长运行时间，比如：

```
spec:
backoffLimit: 5
activeDeadlineSeconds: 100
```

复制代码

一旦运行超过了 100 s，这个 Job 的所有 Pod 都会被终止。并且，你可以在 Pod 的状态里看到终止的原因是 reason: DeadlineExceeded。

以上，就是一个 Job API 对象最主要的概念和用法了。不过，离线业务之所以被称为 Batch Job，当然是因为它们可以以“Batch”，也就是并行的方式去运行。

接下来，我就来为你讲解一下Job Controller 对并行作业的控制方法。

在 Job 对象中，负责并行控制的参数有两个：

1. spec.parallelism，它定义的是一个 Job 在任意时间最多可以启动多少个 Pod 同时运行；
2. spec.completions，它定义的是 Job 至少要完成的 Pod 数目，即 Job 的最小完成数。

这两个参数听起来有点儿抽象，所以我准备了一个例子来帮助你理解。

现在，我在之前计算 Pi 值的 Job 里，添加这两个参数：

```
apiVersion: batch/v1
kind: Job
metadata:
name: pi
spec:
parallelism: 2
completions: 4
template:
spec:
containers:
- name: pi
image: resouer/ubuntu-bc
command: ["sh", "-c", "echo 'scale=5000; 4*a(1)' | bc -l "]
restartPolicy: Never
backoffLimit: 4
```

□复制代码

这样，我们就指定了这个 Job 最大的并行数是 2，而最小的完成数是 4。

接下来，我们来创建这个 Job 对象：

```
$ kubectl create -f job.yaml
```

□复制代码

可以看到，这个 Job 其实也维护了两个状态字段，即 DESIRED 和 SUCCESSFUL，如下所示：

```
$ kubectl get job
NAME DESIRED SUCCESSFUL AGE
pi 4 0 3s
```

□复制代码
其中，DESIRED 的值，正是 completions 定义的最小完成数。

然后，我们可以看到，这个 Job 首先创建了两个并行运行的 Pod 来计算 Pi：

```
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
pi-5mt88 1/1 Running 0 6s
pi-gmcq5 1/1 Running 0 6s
```

□复制代码
而在 40 s 后，这两个 Pod 相继完成计算。

这时我们可以看到，每当有一个 Pod 完成计算进入 Completed 状态时，就会有一个新的 Pod 被自动创建出来，并且快速地从 Pending 状态进入到 ContainerCreating 状态：

```
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
pi-gmcq5 0/1 Completed 0 40s
pi-84ww8 0/1 Pending 0 0s
pi-5mt88 0/1 Completed 0 41s
pi-62rbt 0/1 Pending 0 0s
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
pi-gmcq5 0/1 Completed 0 40s
pi-84ww8 0/1 ContainerCreating 0 0s
pi-5mt88 0/1 Completed 0 41s
pi-62rbt 0/1 ContainerCreating 0 0s
```

□复制代码
紧接着，Job Controller 第二次创建出来的两个并行的 Pod 也进入了 Running 状态：

```
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
pi-5mt88 0/1 Completed 0 54s
pi-62rbt 1/1 Running 0 13s
```

```
pi-84ww8 1/1 Running 0 14s
pi-gmcq5 0/1 Completed 0 54s
```

□复制代码

最终，后面创建的这两个 Pod 也完成了计算，进入了 Completed 状态。

这时，由于所有的 Pod 均已经成功退出，这个 Job 也就执行完了，所以你会看到它的 SUCCESSFUL 字段的值变成了 4：

```
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
pi-5mt88 0/1 Completed 0 5m
pi-62rbt 0/1 Completed 0 4m
pi-84ww8 0/1 Completed 0 4m
pi-gmcq5 0/1 Completed 0 5m
$ kubectl get job
NAME DESIRED SUCCESSFUL AGE
pi 4 4 5m
```

□复制代码

通过上述 Job 的 DESIRED 和 SUCCESSFUL 字段的关系，我们就可以很容易地理解Job Controller 的工作原理了。

首先，Job Controller 控制的对象，直接就是 Pod。

其次，Job Controller 在控制循环中进行的调谐（Reconcile）操作，是根据实际在 Running 状态 Pod 的数目、已经成功退出的 Pod 的数目，以及 parallelism、completions 参数的值共同计算出在这个周期里，应该创建或者删除的 Pod 数目，然后调用 Kubernetes API 来执行这个操作。

以创建 Pod 为例。在上面计算 Pi 值的这个例子中，当 Job 一开始创建出来时，实际处于 Running 状态的 Pod 数目 =0，已经成功退出的 Pod 数目 =0，而用户定义的 completions，也就是最终用户需要的 Pod 数目 =4。

所以，在这个时刻，需要创建的 Pod 数目 = 最终需要的 Pod 数目 - 实际在 Running 状态 Pod 数目 - 已经成功退出的 Pod 数目 = 4 - 0 - 0= 4。也就是说，Job Controller 需要创建 4 个 Pod 来纠正这个不一致状态。

可是，我们又定义了这个 Job 的 parallelism=2。也就是说，我们规定了每次并发创建的 Pod 个数不能超过 2 个。所以，Job Controller 会对前面的计算结果做一个修正，修正后的期望创建的 Pod 数目应该是：2 个。

这时候，Job Controller 就会并发地向 kube-apiserver 发起两个创建 Pod 的请求。

类似地，如果在这次调谐周期里，Job Controller 发现实际在 Running 状态的 Pod 数目，比 parallelism 还大，那么它就会删除一些 Pod，使两者相等。

综上所述，Job Controller 实际上控制了，作业执行的**并行度**，以及总共需要完成的**任务数**这两个重要参数。而在实际使用时，你需要根据作业的特性，来决定并行度（parallelism）和任务数（completions）的合理取值。

接下来，我再和你分享三种常用的、使用 Job 对象的方法。

第一种用法，也是最简单粗暴的用法：外部管理器 +Job 模板。

这种模式的特定用法是：把 Job 的 YAML 文件定义为一个“模板”，然后用一个外部工具控制这些“模板”来生成 Job。这时，Job 的定义方式如下所示：

```
apiVersion: batch/v1
kind: Job
metadata:
name: process-item-$ITEM
labels:
jobgroup: jobexample
spec:
template:
metadata:
name: jobexample
labels:
jobgroup: jobexample
spec:
containers:
- name: c
image: busybox
command: ["sh", "-c", "echo Processing item $ITEM && sleep 5"]
restartPolicy: Never
```

□复制代码

可以看到，我们在这个 Job 的 YAML 里，定义了 $ITEM 这样的“变量”。

所以，在控制这种 Job 时，我们只要注意如下两个方面即可：

1. 创建 Job 时，替换掉 $ITEM 这样的变量；
2. 所有来自于同一个模板的 Job，都有一个 jobgroup: jobexample 标签，也就是说这一组 Job 使用这样一个相同的标识。

而做到第一点非常简单。比如，你可以通过这样一句 shell 把 $ITEM 替换掉：

```
$ mkdir ./jobs
$ for i in apple banana cherry
do
cat job-tmpl.yaml | sed "s/\$ITEM/$i/" > ./jobs/job-$i.yaml
done
```

复制代码

这样，一组来自于同一个模板的不同 Job 的 yaml 就生成了。接下来，你就可以通过一句 kubectl create 指令创建这些 Job 了：

```
$ kubectl create -f ./jobs
$ kubectl get pods -l jobgroup=jobexample
NAME READY STATUS RESTARTS AGE
process-item-apple-kixwv 0/1 Completed 0 4m
process-item-banana-wrsf7 0/1 Completed 0 4m
process-item-cherry-dnfu9 0/1 Completed 0 4m
```

复制代码

这个模式看起来虽然很“傻”，但却是 Kubernetes 社区里使用 Job 的一个很普遍的模式。

原因很简单：大多数用户在需要管理 Batch Job 的时候，都已经有了一套自己的方案，需要做的往往就是集成工作。这时候，Kubernetes 项目对这些方案来说最有价值的，就是 Job 这个 API 对象。所以，你只需要编写一个外部工具（等同于我们这里的 for 循环）来管理这些 Job 即可。

这种模式最典型的应用，就是 TensorFlow 社区的 KubeFlow 项目。

很容易理解，在这种模式下使用 Job 对象，completions 和 parallelism 这两个字段都应该使用默认值 1，而不应该由我们自行设置。而作业 Pod 的并行控制，应该完全交由外部工具来进行管理（比如，KubeFlow）。

第二种用法：拥有固定任务数目的并行 Job。

这种模式下，我只关心最后是否有指定数目（spec.completions）个任务成功退出。至于执行时的并行度是多少，我并不关心。

比如，我们这个计算 Pi 值的例子，就是这样一个典型的、拥有固定任务数目（completions=4）的应用场景。 它的 parallelism 值是 2；或者，你可以干脆不指定 parallelism，直接使用默认的并行度（即：1）。

此外，你还可以使用一个工作队列（Work Queue）进行任务分发。这时，Job 的 YAML 文件定义如下所示：

```
apiVersion: batch/v1
kind: Job
metadata:
name: job-wq-1
spec:
completions: 8
parallelism: 2
template:
metadata:
name: job-wq-1
spec:
containers:
- name: c
image: myrepo/job-wq-1
env:
- name: BROKER_URL
value: amqp://guest:guest@rabbitmq-service:5672
- name: QUEUE
value: job1
restartPolicy: OnFailure
```

□复制代码

我们可以看到，它的 completions 的值是：8，这意味着我们总共要处理的任务数目是 8 个。也就是说，总共会有 8 个任务会被逐一放入工作队列里（你可以运行一个外部小程序作为生产者，来提交任务）。

在这个实例中，我选择充当工作队列的是一个运行在 Kubernetes 里的 RabbitMQ。所以，我们需要在 Pod 模板里定义 BROKER_URL，来作为消费者。

所以，一旦你用 kubectl create 创建了这个 Job，它就会以并发度为 2 的方式，每两个 Pod 一组，创建出 8 个 Pod。每个 Pod 都会去连接 BROKER_URL，从 RabbitMQ 里读取任务，然后各自进行处理。这个 Pod 里的执行逻辑，我们可以用这样一段伪代码来表示：

```
/* job-wq-1 的伪代码 */
queue := newQueue($BROKER_URL, $QUEUE)
task := queue.Pop()
process(task)
exit
```

复制代码

可以看到，每个 Pod 只需要将任务信息读取出来，处理完成，然后退出即可。而作为用户，我只关心最终一共有 8 个计算任务启动并且退出，只要这个目标达到，我就认为整个 Job 处理完成了。所以说，这种用法，对应的就是“任务总数固定”的场景。

第三种用法，也是很常用的一个用法：指定并行度（parallelism），但不设置固定的 completions 的值。

此时，你就必须自己想办法，来决定什么时候启动新 Pod，什么时候 Job 才算执行完成。在这种情况下，任务的总数是未知的，所以你不仅需要一个工作队列来负责任务分发，还需要能够判断工作队列已经为空（即：所有的工作已经结束了）。

这时候，Job 的定义基本上没变化，只不过是不再需要定义 completions 的值了而已：

```
apiVersion: batch/v1
kind: Job
metadata:
name: job-wq-2
spec:
parallelism: 2
template:
metadata:
name: job-wq-2
```

```
spec:
containers:
- name: c
image: gcr.io/myproject/job-wq-2
env:
- name: BROKER_URL
value: amqp://guest:guest@rabbitmq-service:5672
- name: QUEUE
value: job2
restartPolicy: OnFailure
```

复制代码

而对应的 Pod 的逻辑会稍微复杂一些，我可以用这样一段伪代码来描述：

```
/* job-wq-2 的伪代码 */
for !queue.IsEmpty($BROKER_URL, $QUEUE) {
task := queue.Pop()
process(task)
}
print("Queue empty, exiting")
exit
```

复制代码

由于任务数目的总数不固定，所以每一个 Pod 必须能够知道，自己什么时候可以退出。比如，在这个例子中，我简单地以“队列为空”，作为任务全部完成的标志。所以说，这种用法，对应的是“任务总数不固定”的场景。

不过，在实际的应用中，你需要处理的条件往往会非常复杂。比如，任务完成后的输出、每个任务 Pod 之间是不是有资源的竞争和协同等等。

所以，在今天这篇文章中，我就不再展开 Job 的用法了。因为，在实际场景里，要么干脆就用第一种用法来自己管理作业；要么，这些任务 Pod 之间的关系就不那么“单纯”，甚至还是“有状态应用”（比如，任务的输入 / 输出是在持久化数据卷里）。在这种情况下，我在后面要重点讲解的 Operator，加上 Job 对象一起，可能才能更好的满足实际离线任务的编排需求。

最后，我再来和你分享一个非常有用的 Job 对象，叫作：CronJob。

顾名思义，CronJob 描述的，正是定时任务。它的 API 对象，如下所示：

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
name: hello
spec:
schedule: "*/1 * * * *"
jobTemplate:
spec:
template:
spec:
containers:
- name: hello
image: busybox
args:
- /bin/sh
- -c
- date; echo Hello from the Kubernetes cluster
restartPolicy: OnFailure
```

复制代码

在这个 YAML 文件中，最重要的关键词就是**jobTemplate**。看到它，你一定恍然大悟，原来 CronJob 是一个 Job 对象的控制器（Controller）！

没错，CronJob 与 Job 的关系，正如同 Deployment 与 Pod 的关系一样。CronJob 是一个专门用来管理 Job 对象的控制器。只不过，它创建和删除 Job 的依据，是 schedule 字段定义的、一个标准的Unix Cron格式的表达式。

比如，"*/1 * * * *"。

这个 Cron 表达式里 */1 中的 * 表示从 0 开始，/ 表示“每”，1 表示偏移量。所以，它的意思就是：从 0 开始，每 1 个时间单位执行一次。

那么，时间单位又是什么呢?

Cron 表达式中的五个部分分别代表：分钟、小时、日、月、星期。

所以，上面这句 Cron 表达式的意思是：从当前开始，每分钟执行一次。

而这里要执行的内容，就是 jobTemplate 定义的 Job 了。

所以，这个 CronJob 对象在创建 1 分钟后，就会有一个 Job 产生了，如下所示：

```
$ kubectl create -f ./cronjob.yaml
cronjob "hello" created
# 一分钟后
$ kubectl get jobs
NAME DESIRED SUCCESSFUL AGE
hello-4111706356 1 1 2s
```

□复制代码

此时，CronJob 对象会记录下这次 Job 执行的时间：

```
$ kubectl get cronjob hello
NAME SCHEDULE SUSPEND ACTIVE LAST-SCHEDULE
hello */1 * * * * False 0 Thu, 6 Sep 2018 14:34:00 -070
```

□复制代码

需要注意的是，由于定时任务的特殊性，很可能某个 Job 还没有执行完，另外一个新 Job 就产生了。这时候，你可以通过 spec.concurrencyPolicy 字段来定义具体的处理策略。比如：

1. concurrencyPolicy=Allow，这也是默认情况，这意味着这些 Job 可以同时存在；
2. concurrencyPolicy=Forbid，这意味着不会创建新的 Pod，该创建周期被跳过；
3. concurrencyPolicy=Replace，这意味着新产生的 Job 会替换旧的、没有执行完的 Job。

而如果某一次 Job 创建失败，这次创建就会被标记为“miss”。当在指定的时间窗口内，miss 的数目达到 100 时，那么 CronJob 会停止再创建这个 Job。

这个时间窗口，可以由 spec.startingDeadlineSeconds 字段指定。比如 startingDeadlineSeconds=200，意味着在过去 200 s 里，如果 miss 的数目达到了 100 次，那么这个 Job 就不会被创建执行了。

总结

在今天这篇文章中，我主要和你分享了 Job 这个离线业务的编排方法，讲解了 completions 和 parallelism 字段的含义，以及 Job Controller 的执行原理。

紧接着，我通过实例和你分享了 Job 对象三种常见的使用方法。但是，根据我在社区和生产环境中的经验，大多数情况下用户还是更倾向于自己控制 Job 对象。所以，相比于这些固定的“模式”，掌握 Job 的 API 对象，和它各个字段的准确含义会更加重要。

最后，我还介绍了一种 Job 的控制器，叫作：CronJob。这也印证了我在前面的分享中所说的：用一个对象控制另一个对象，是 Kubernetes 编排的精髓所在。

思考题

根据 Job 控制器的工作原理，如果你定义的 parallelism 比 completions 还大的话，比如：

```
parallelism: 4
completions: 2
```

复制代码

那么，这个 Job 最开始创建的时候，会同时启动几个 Pod 呢？原因是什么？