ALIBABA CLOUD

阿里云

Serverless 工作流

最佳实践

文档版本：20211125

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.任务状态轮询

本文介绍了任务状态轮询和Serverless工作流实现的具体步骤。

简介

在长时间任务的场景中如果任务结束后没有回调机制，开发者通常会采用轮询的方式来判断任务的结束。可靠的轮询实现需要维护状态的持久化以保证即使当前轮询进程失败退出，进程恢复后轮询也会继续进行。本示例通过一个假设场景：用户调用函数计算提交了一个多媒体处理任务，该任务耗时从1分钟到几小时不等，任务执行状态可以通过API查询，介绍如何使用Serverless工作流实现一个通用可靠的任务轮询工作流。

Serverless工作流实现

下面的教程会将两个FC函数编排成一个任务轮询工作流，该示例需要以下3个步骤：

1. 创建FC函数
2. 创建Serverless工作流流程
3. 开始执行并查看结果

步骤1：创建FC函数

1. 首先创建一个名为 `fnf-demo` 的FC服务，并在该服务下创建两个Python2.7的函数，详细步骤，请参见使用控制台创建函数。
   - StartJob函数：模拟通过调用API开始一个长时间的任务，返回一个任务ID。

     ```
     import logging
     import uuid
     def handler(event, context):
       logger = logging.getLogger()
       id = uuid.uuid4()
       logger.info('Started job with ID %s' % id)
       return {"job_id": str(id)}
     ```

   - GetJobStatus函数：模拟通过调用API获取指定任务的执行结果，比较当前的时间和函数第一次执行的时间的差值和输入中 `delay` 的值，返回不同的状态：“success”或“running”。

```
import logging
import uuid
import time
import json
start_time = int(time.time())
def handler(event, context):
 evt = json.loads(event)
 logger = logging.getLogger()
 job_id = evt["job_id"]
 logger.info('Started job with ID %s' % job_id)
 now = int(time.time())
 status = "running"
 delay = 60
 if "delay" in evt:
  delay = evt["delay"]
 if now - start_time > delay:
  status = "success"
 try_count = 0
 if "try_count" in evt:
  try_count = evt["try_count"]
 try_count = try_count + 1
 logger.info('Job %s, status %s, try_count %d' % (job_id, status, try_count))
 return {"job_id": job_id, "job_status":status, "try_count":try_count}
```

步骤2：创建Serverless工作流流程

该流程的主要逻辑描述如下：

1. StartJob步骤：调用 `StartJob` 函数开始一个任务。
2. Wait10s步骤：等待10秒。
3. GetJobStatus步骤：调用 `GetJobStatus`
4. CheckJobComplete步骤：检查 `GetJobStatus` 函数返回的结果：
   - 如果返回"success"整个流程执行成功。
   - 如果轮询尝试次数大于3次，认为任务执行失败，流程执行失败。
   - 如果返回"running"则跳回到 `Wait10s` 步骤，继续执行。

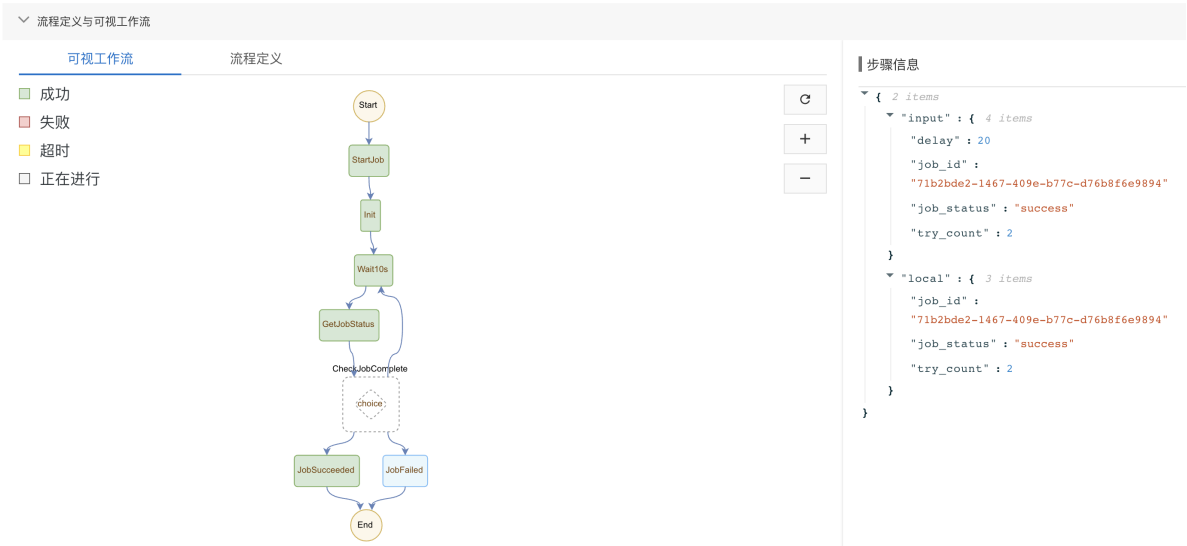
```
version: v1
type: flow
steps:
 - type: task
   name: StartJob
   resourceArn: acs:fc:cn-hangzhou:{accountID}:services/fnf-demo/functions/StartJob
 - type: pass
   name: Init
   outputMappings:
    - target: try_count
      source: 0
 - type: wait
   name: Wait10s
   duration: 10
 - type: task
   name: GetJobStatus
   resourceArn: acs:fc:cn-hangzhou:{accountID}:services/fnf-demo/functions/GetJobStatus
   inputMappings:
    - target: job_id
      source: $local.job_id
    - target: delay
      source: $input.delay
    - target: try_count
      source: $local.try_count
 - type: choice
   name: CheckJobComplete
   inputMappings:
    - target: status
      source: $local.job_status
    - target: try_count
      source: $local.try_count
   choices:
     - condition: $.status == "success"
       goto: JobSucceeded
     - condition: $.try_count > 3
       goto: JobFailed
     - condition: $.status == "running"
       goto: Wait10s
 - type: succeed
   name: JobSucceeded
 - type: fail
   name: JobFailed
```

步骤3：开始执行并查看结果

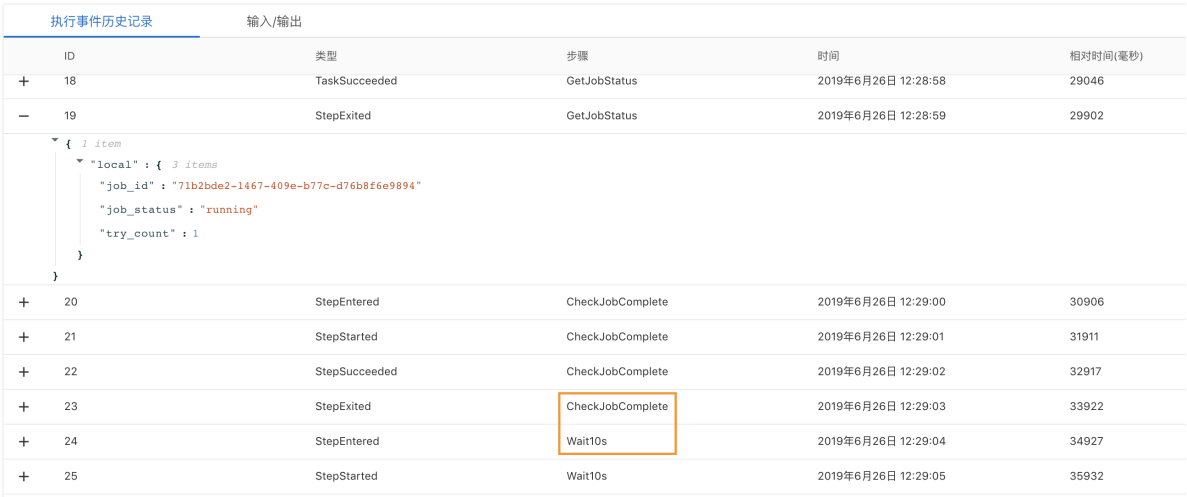
在控制台创建好的流程中单击**新执行**并提供以下JSON对象作为输入，其中 `delay` 字段的值模拟任务完成需要的时间，这里预期任务在开始20秒后， `GetJobStatus` 函数返回“success”，在此之前均返回“running”，您可以调整 `delay` 的值观察不同的执行结果。

```
{
 "delay": 20
}
```

- 下图展示的是轮询从开始到结束的流程执行可视化。



- 下图展示的是任务需要20秒完成，可以看到流程执行历史中第一次 `GetJobStatus` 返回“running”因此 `CheckJobComplete` 的后续步骤跳回到 `Wait10s` 进行等待和下一次查询，第二次查询返回“success”，流程执行结束。



ID	类型	步骤	时间	相对时间(毫秒)
18	TaskSucceeded	GetJobStatus	2019年6月26日 12:28:58	29046
19	StepExited	GetJobStatus	2019年6月26日 12:28:59	29902
20	StepEntered	CheckJobComplete	2019年6月26日 12:29:00	30906
21	StepStarted	CheckJobComplete	2019年6月26日 12:29:01	31911
22	StepSucceeded	CheckJobComplete	2019年6月26日 12:29:02	32917
23	StepExited	CheckJobComplete	2019年6月26日 12:29:03	33922
24	StepEntered	Wait10s	2019年6月26日 12:29:04	34927
25	StepStarted	Wait10s	2019年6月26日 12:29:05	35932

2.分布式多步骤事务

本文介绍了如何使用Serverless工作流提供长流程分布式事务保证，帮助用户聚焦于自身业务逻辑。

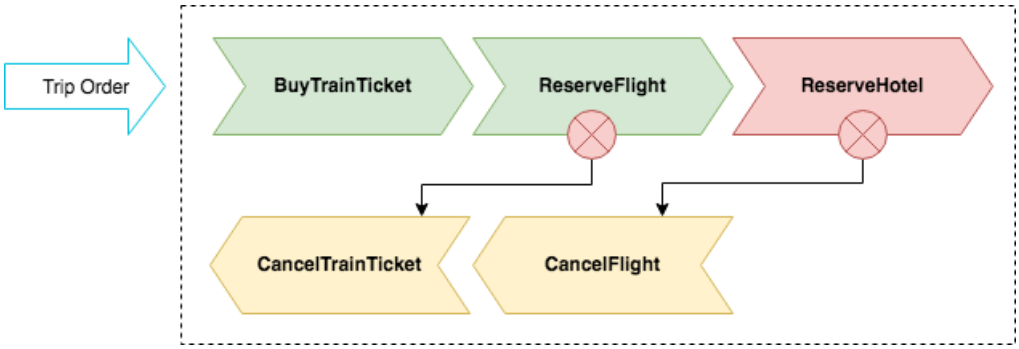
简介

复杂的业务场景例如电商网站、酒店、航班预定这类涉及订单管理的应用通常要访问多个远程服务，并且对操作事务性语义（即所有步骤全部成功或全部失败，不存在中间状态）有较高要求。在流量较小、数据存储集中的应用中，事务性可以通过关系型数据库提供的ACID特性满足。然而在大流量场景下，为了高可用和可扩展性，业务通常选择向微服务的分布式架构方向演进。在这样的架构中提供多步骤事务性的保证通常需要引入队列和数据库来持久化消息以及展现流程状态，这类系统的开发和运维会给业务方带来额外的成本和负担。而使用Serverless工作流提供长流程分布式事务保证会帮您解决这些问题。

场景描述

假设某应用为其用户提供预定火车票、航班和酒店的功能，要求三个步骤保证事务性。该功能需要三个远程调用实现（例如预定火车票需要调用12306接口），如果三个调用都成功则该订单成功。然而实际上任何一个远程调用都有可能会失败，因此该应用需要对不同的失败场景做出相应的补偿逻辑，回退已完成操作。如下图所示：

- 如果预定火车票（BuyTrainTicket）成功，而预定航班（ReserveFlight）失败，则需要取消已经购买的火车票（CancelTrainTicket），并告知您订单失败。
- 如果预定火车票（BuyTrainTicket）和预定航班（ReserveFlight）均成功，但是预订酒店（ReserveHotel）失败，则需要取消已经预定的航班（CancelFlight）和火车票（CancelTrainTicket），并告知您订单失败。



Serverless工作流实现

下文的示例将FC函数编排成一个Serverless工作流流程从而实现了一个可靠的多步骤长流程，该示例分为3步：

1. 创建FC函数
2. 创建流程
3. 执行并查看结果

步骤1：创建FC函数

本步骤是模拟场景案例中提示的三个操作即预订火车票、预订航班及预定酒店。

1. 创建下面的Python2.7的函数，关于创建的详细步骤，请参见FC文档，建议命名：
   - Service: fnf-demo
   - Function: Operation

Operation函数模拟各操作（例如预定航班、预定酒店）的实现，根据输入决定该操作执行结果（成功或失败）。

```
import json
import logging
import uuid
def handler(event, context):
 evt = json.loads(event)
 logger = logging.getLogger()
 id = uuid.uuid4()
 op = "operation"
 if 'operation' in evt:
  op = evt['operation']
  if op in evt:
   result = evt[op]
   if result == False:
    logger.info("%s failed" % op)
    exit()
 logger.info("%s succeeded, id %s" % (op, id))
 return '{"%s":"success", "%s_txnID": "%s"}' % (op, op, id)
```

步骤2：创建流程

使用Serverless工作流控制台创建下面的流程。

1. 配置流程RAM角色

```
{
  "Statement": [
    {
      "Action": "sts:AssumeRole",
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "fnf.aliyuncs.com"
        ]
      }
    }
  ],
  "Version": "1"
}
```

2. 流程定义

```
version: v1
type: flow
steps:
 - type: task
  resourceArn: acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation
  name: BuyTrainTicket
  inputMappings:
  - target: operation
   source: buy_train_ticket
  - target: buy_train_ticket
   source: $input.buy_train_ticket_result
```

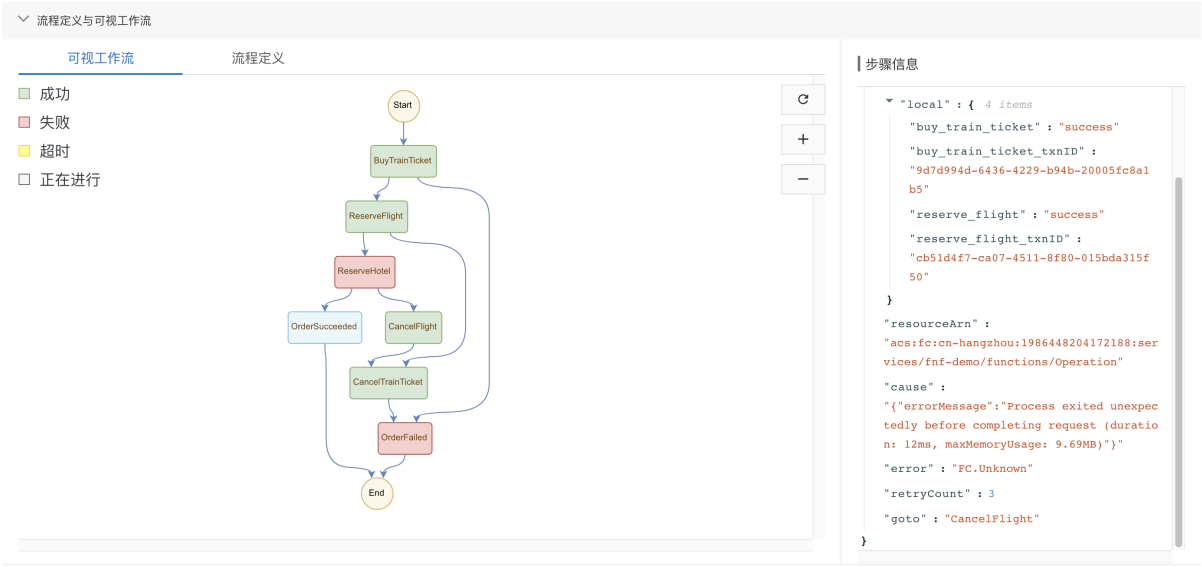
```
  catch:
  - errors:
    - FC.Unknown
    goto: OrderFailed
- type: task
  resourceArn: acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation
  name: ReserveFlight
  inputMappings:
  - target: operation
    source: reserve_flight
  - target: reserve_flight
    source: $input.reserve_flight_result
  catch: # 捕获ReserveFlight task抛出的FC.Unknown错误，跳转到CancelTrainTicket。
  - errors:
    - FC.Unknown
    goto: CancelTrainTicket
- type: task
  resourceArn: acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation
  name: ReserveHotel
  inputMappings:
  - target: operation
    source: reserve_hotel
  - target: reserve_hotel
    source: $input.reserve_hotel_result
  retry: # 对FC.Unknown类型的错误最多指数退避重试3次，初始间隔1s，后续间隔=上次间隔*2。
  - errors:
    - FC.Unknown
    intervalSeconds: 1
    maxAttempts: 3
    multiplier: 2
  catch: # 捕获ReserveHotel task抛出的FC.Unknown错误，跳转到CancelFlight。
    - errors:
      - FC.Unknown
      goto: CancelFlight
- type: succeed
  name: OrderSucceeded
- type: task
  resourceArn: acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation
  name: CancelFlight
  inputMappings:
  - target: operation
    source: cancel_flight
  - target: reserve_flight_txnID
    source: $local.reserve_flight_txnID
- type: task
  resourceArn: acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation
  name: CancelTrainTicket
  inputMappings:
  - target: operation
    source: cancel_train_ticket
  - target: reserve_flight_txnID
    source: $local.reserve_flight_txnID
- type: fail
  name: OrderFailed
```

步骤3：执行并查看结果

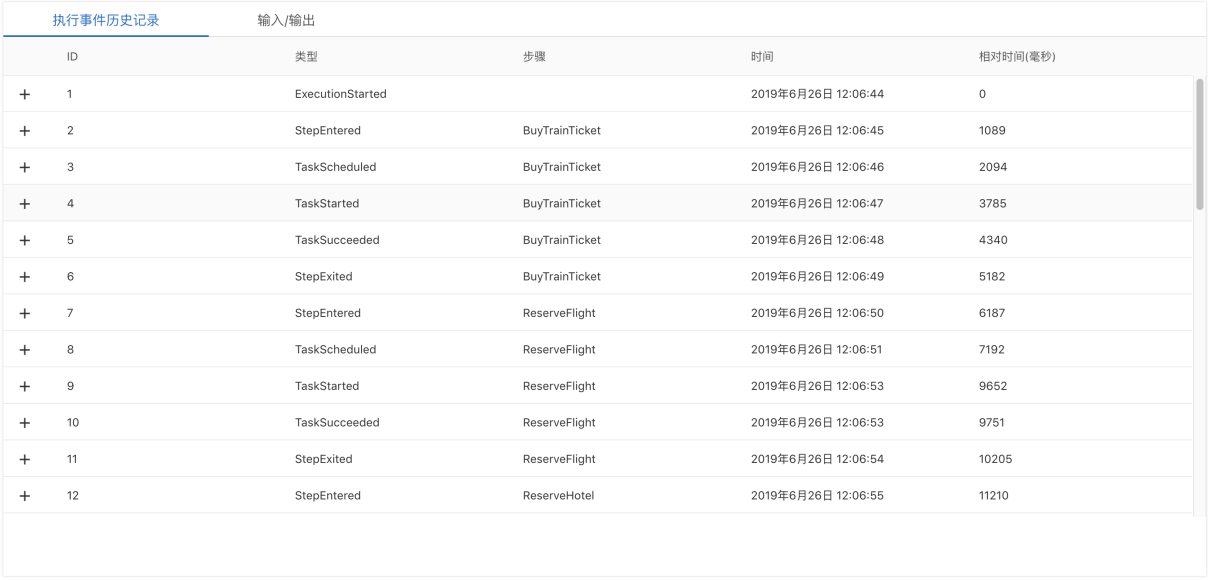
在控制台上对创建好的流程（Flow）开始一个新的执行（Execution）。StartExecution API要求传入JSON格式的输入。下面的JSON对象可以模拟每个步骤的成功或失败（例如"reserve_hotel_result":"fail"代表模拟预定酒店这步失败）。StartExecution是一个异步API，调用结束后，Serverless工作流会返回一个执行名字用来查询流程执行状态。

```
{
 "buy_train_ticket_result":"success",
 "reserve_flight_result":"success",
 "reserve_hotel_result":"fail"
}
```

流程执行开始后，在Serverless工作流控制台可以查看流程的执行过程和结果。从步骤信息页签您可以看到，由于 "reserve_hotel_result":"fail" 和 ReserveHotel 函数调用失败，Serverless工作流按照流程定义，依次取消航班（CancelFlight）、取消火车票（CancelTrainTicket）。Serverless工作流每个步骤转换有持久化的保证，因此网络中断或进程崩溃等失败场景不会影响流程事务性的保证。



流程执行会产生执行历史事件（event），这些事件可以通过控制台、SDK或CLI调用 GetExecutionHistory API查询。



执行事件历史记录 输入/输出

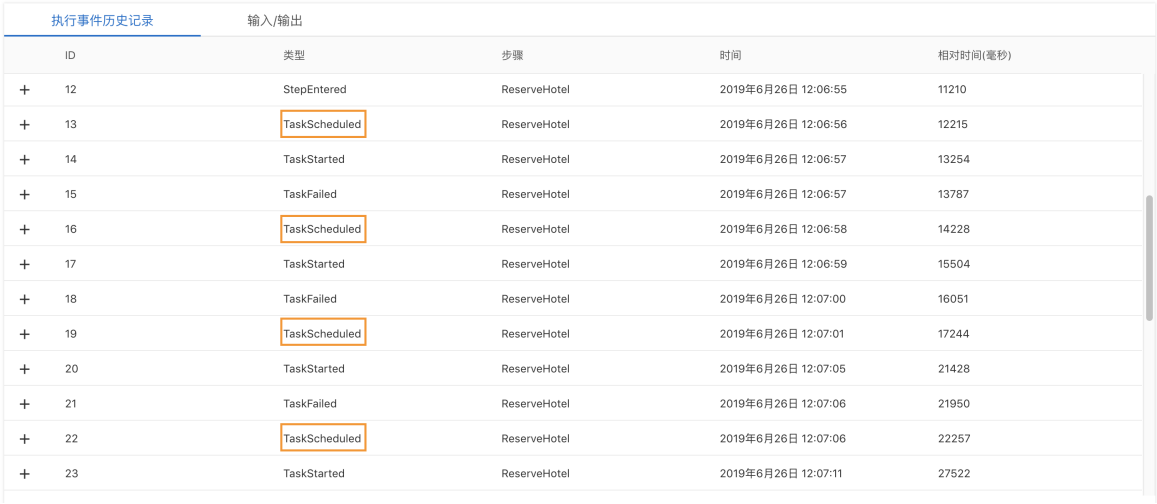
	ID	类型	步骤	时间	相对时间(毫秒)
+	1	ExecutionStarted		2019年6月26日 12:06:44	0
+	2	StepEntered	BuyTrainTicket	2019年6月26日 12:06:45	1089
+	3	TaskScheduled	BuyTrainTicket	2019年6月26日 12:06:46	2094
+	4	TaskStarted	BuyTrainTicket	2019年6月26日 12:06:47	3785
+	5	TaskSucceeded	BuyTrainTicket	2019年6月26日 12:06:48	4340
+	6	StepExited	BuyTrainTicket	2019年6月26日 12:06:49	5182
+	7	StepEntered	ReserveFlight	2019年6月26日 12:06:50	6187
+	8	TaskScheduled	ReserveFlight	2019年6月26日 12:06:51	7192
+	9	TaskStarted	ReserveFlight	2019年6月26日 12:06:53	9652
+	10	TaskSucceeded	ReserveFlight	2019年6月26日 12:06:53	9751
+	11	StepExited	ReserveFlight	2019年6月26日 12:06:54	10205
+	12	StepEntered	ReserveHotel	2019年6月26日 12:06:55	11210

错误处理和重试

1. 上面示例中的预定航班、预定酒店等远程调用都有可能受到网络或服务错误等原因导致调用失败，而增加对瞬时错误的重试可以提高订单流程成功率。Serverless工作流在任务（Task）类型的步骤（Step）自带重试功能，如预定酒店这个步骤用下面的写法可以实现对FC.Unknown类型的错误指数退避。假设重试到达最大次数后 `ReserveHotel` 都无法成功，按照该步骤中 `catch` 的定义，ReserveHotel函数抛出的FC.Unknown错误会被捕获并将跳转到 `CancelFlight` 执行定义好的补偿逻辑。

```
- type: task
  resourceArn: acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation
  name: ReserveHotel
  inputMappings:
  - target: operation
    source: reserve_hotel
  retry: # 对FC.Unknown类型的错误最多指数退避重试3次，初始间隔1s，后续间隔=上次间隔*2。
  - errors:
    - FC.Unknown
    intervalSeconds: 1
    maxAttempts: 3
    multiplier: 2
  catch: # 捕获ReserveHotel task抛出的FC.Unknown错误，跳转到CancelFlight。
    - errors:
      - FC.Unknown
      goto: CancelFlight
```

2. 下图可以看到加入重试之后预订酒店（ReserveHotel）任务执行了多次直到最大重试数。



执行事件历史记录 输入/输出

	ID	类型	步骤	时间	相对时间(毫秒)
+	12	StepEntered	ReserveHotel	2019年6月26日 12:06:55	11210
+	13	TaskScheduled	ReserveHotel	2019年6月26日 12:06:56	12215
+	14	TaskStarted	ReserveHotel	2019年6月26日 12:06:57	13254
+	15	TaskFailed	ReserveHotel	2019年6月26日 12:06:57	13787
+	16	TaskScheduled	ReserveHotel	2019年6月26日 12:06:58	14228
+	17	TaskStarted	ReserveHotel	2019年6月26日 12:06:59	15504
+	18	TaskFailed	ReserveHotel	2019年6月26日 12:07:00	16051
+	19	TaskScheduled	ReserveHotel	2019年6月26日 12:07:01	17244
+	20	TaskStarted	ReserveHotel	2019年6月26日 12:07:05	21428
+	21	TaskFailed	ReserveHotel	2019年6月26日 12:07:06	21950
+	22	TaskScheduled	ReserveHotel	2019年6月26日 12:07:06	22257
+	23	TaskStarted	ReserveHotel	2019年6月26日 12:07:11	27522

步骤间的数据传递

1. 预定酒店失败后需要取消航班和火车票，这两部分别需要用到预定航班和预定火车票返回的交易ID（txnID），下面的 `inputMapping` 对象描述了如何将之前步骤产生的输出传入 `CancelFlight` 这个步骤中。

```
- type: task
  resourceArn: acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation
  name: CancelFlight
  inputMappings:
  - target: operation
    source: cancel_flight
  - target: reserve_flight_txnID
    source: $local.reserve_flight_txnID
```

2. 流程执行各步骤结束的输出都会被放在 `StepExited` 事件详情（EventDetail）的local对象中。

```
{
  "input":{
    "operation":"reserve_hotel",
    "reserve_hotel_result":"fail"
  },
  "local":{
    "buy_train_ticket":"success",
    "buy_train_ticket_txnID":"d37412b3-bb68-4d04-9d90-c8c15643d45e",
    "reserve_flight_result":"success",
    "reserve_flight_txnID":"024caecf-cfa3-43a6-b561-9b6fe0571b55"
  },
  "resourceArn":"acs:fc:{region}:{accountID}:services/fnf-demo/functions/Operation",
  "cause":"{\"errorMessage\":\"Process exited unexpectedly before completing request (duration: 12ms, maxMemoryUsage: 9.18MB)\"}",
  "error":"FC.Unknown",
  "retryCount":3,
  "goto":"CancelFlight"
}
```

3. 结合上面的 `EventDetail` 和 `inputMappings` 的映射之后，传入到 `CancelFlight` 步骤的输入变成如下

JSON对象，这样 `CancelFlight` 函数的输入会包含 `reserve_flight_txnID` 字段。

```
"input":{
 "operation":"cancel_flight",
 "reserve_flight_txnID":"024caecf-cfa3-43a6-b561-9b6fe0571b55"
}
```

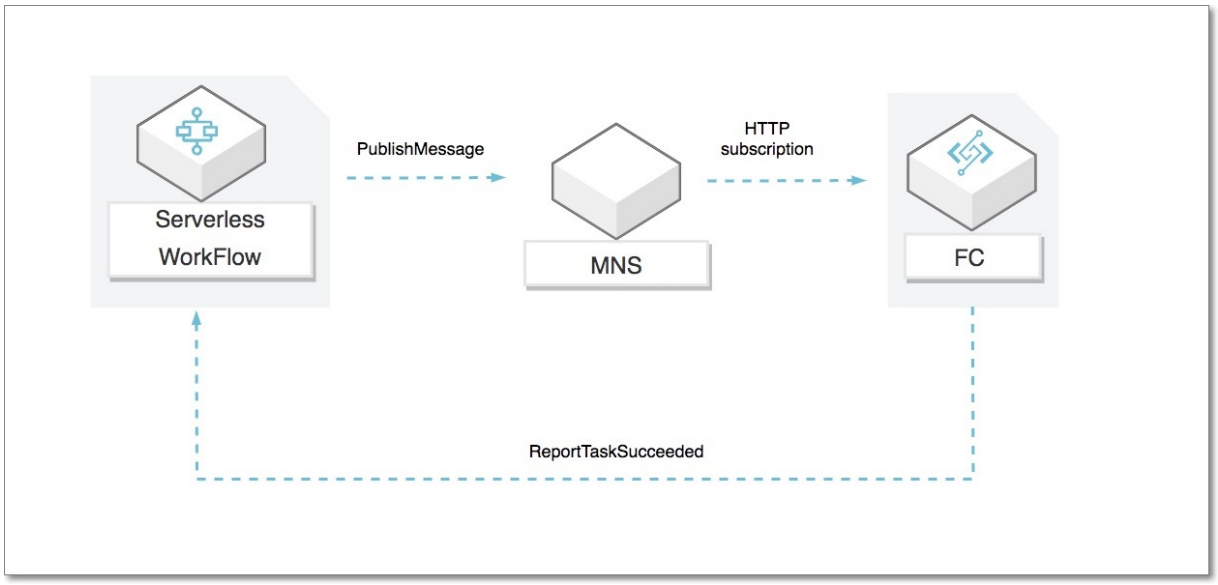
3.MNS主题集成和消息发布

本文介绍了如何使用任务步骤的等待回调（wait ForCallback）模式集成MNS主题，并发布消息到主题。MNS主题接收到消息后，调用工作流ReportTaskSucceeded或ReportTaskFailed API回调任务状态。

框架原理

应用部署后执行流程如下：

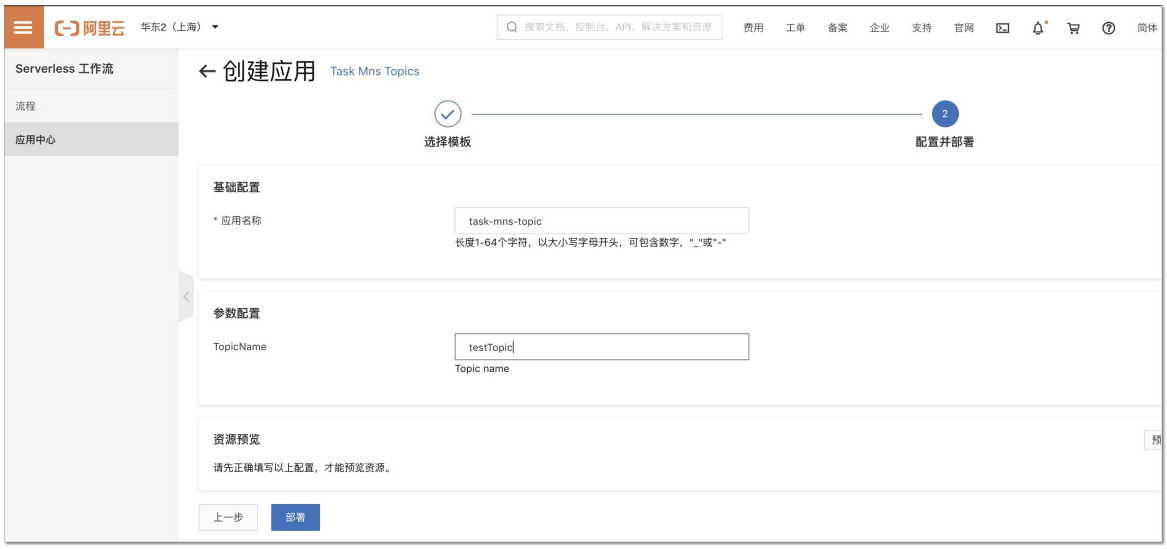
1. 执行工作流，任务步骤发布消息到MNS主题。任务步骤的 `TaskToken` 会被放入消息体一起发送到主题。
2. 工作流任务步骤暂停执行，等待任务回调。
3. MNS主题接收到消息后，将消息和 `TaskToken` 通过HTTP推送发送到函数计算FC的函数HTTP触发器，触发函数执行。
4. 函数计算函数最终获取到 `TaskToken` ，并调用ReportTaskSucceeded来报告任务状态。
5. 流程继续执行。



部署应用

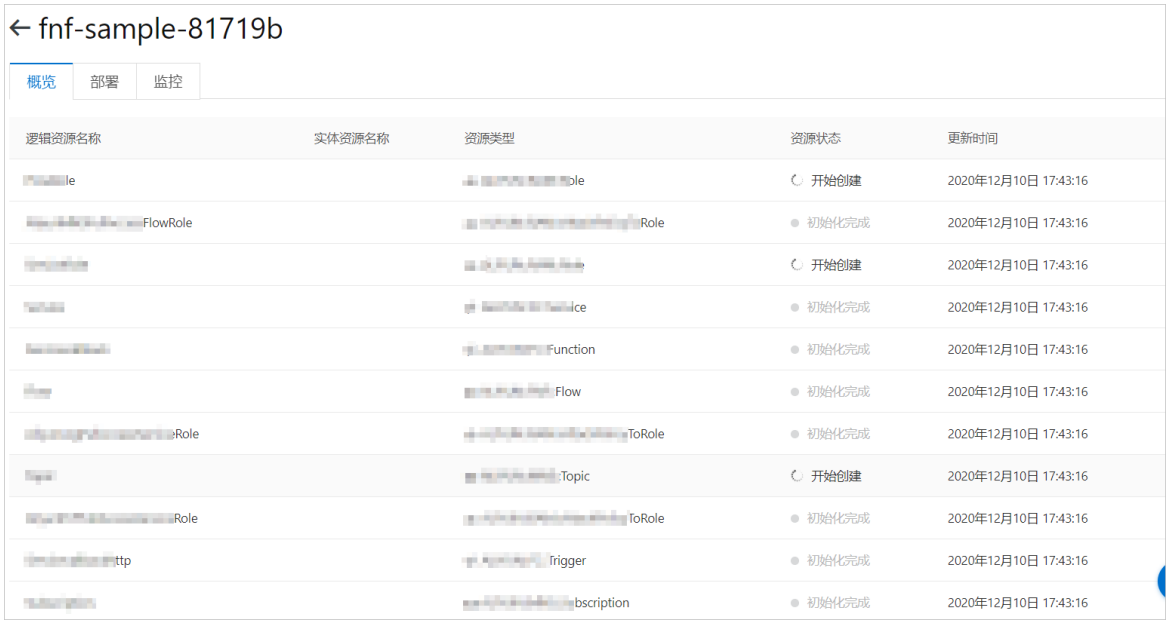
1. 登录Serverless工作流控制台。
2. 在**流程**页面，单击**创建流程**。
3. 在**创建流程**页面，选择**示例项目 > 任务步骤编排MNS主题**模板，单击**下一步**。

4. 在**创建应用**页面，配置相关信息，创建模板对应的应用，并单击**部署**。



- **应用名称**：自定义参数，同一账号下必须唯一。
- **TopicName**：自定义参数，如果对应MNS主题不存在会自动创建。

单击**部署**后，会显示应用下创建的所有资源。



5. 执行工作流。

   执行以下命令。

```
{
  "messageBody": "hello world"
}
```

   执行成功后，您可以看到执行结果的状态。

应用代码

1. 编排MNS主题的工作流。

   将任务步骤回调的 `TaskToken` 封装在消息的 `MessageBody` 中，用于后续的回调。 `outputMappings` 中读取ReportTaskSucceeded设置的 `output` 。

```
version: v1
type: flow
steps:
 - type: task
   name: mns-topic-task
   resourceArn: acs:mns:::/topics/<topic>/messages
   pattern: waitForCallback
   inputMappings:
    - target: messageBody
      source: $input.messageBody
    - target: taskToken
      source: $context.task.token
   outputMappings:
    - target: status
      source: $local.status
   serviceParams:
     MessageBody: $
```

2. 回调任务步骤的FC函数。

   读取 `MessageBody` 中封装的 `TaskToken` ，回调任务状态设置 `output` 为 `{"status":"success"}` 。

```
def handler(environ, start_response):
  # Get request body
  try:
    request_body_size = int(environ.get('CONTENT_LENGTH',
0))
  except ValueError:
    request_body_size = 0
  request_body =
environ['wsgi.input'].read(request_body_size)
  print('Request body:
{}'.format(request_body))
  body = json.loads(request_body)
  message_body_str =
body['Message']
  # Read MessageBody and TaskToken from
message body
  message_body =
json.loads(message_body_str)
  task_token =
message_body['taskToken']
  ori_message_body =
message_body['messageBody']
  print('Task token: {}\norigin message
body: {}'.format(task_token, ori_message_body))
  # Init fnf client use sts token
  context = environ['fc.context']
  creds = context.credentials
  sts_creds =
StsTokenCredential(creds.access_key_id, creds.access_key_secret, creds.security_token)
  fnf_client =
AcsClient(credential=sts_creds, region_id=context.region)
  # Report task succeeded to serverless
workflow
  req =
ReportTaskSucceededRequest()
  req.set_TaskToken(task_token)
  req.set_Output('{"status":
"success"}')
  resp =
fnf_client.do_action_with_exception(req)
  print('Report task response:
{}'.format(resp))
  # Response to http request
  status = '200 OK'
  response_headers = [('Content-type',
'text/plain')]
  start_response(status,
response_headers)
  return [b'OK']
```

更多信息

任务步骤编排MNS主题应用的更多信息，请参见task-mns-topics应用代码。

4.使用MNS服务集成及回调编排任意任务类型

Serverless工作流的服务集成功能可以简化用户与云服务的交互。本文示例中，我们使用MNS队列集成的功能，结合回调 （callback）完成更多非函数计算的计算任务的编排。

简介

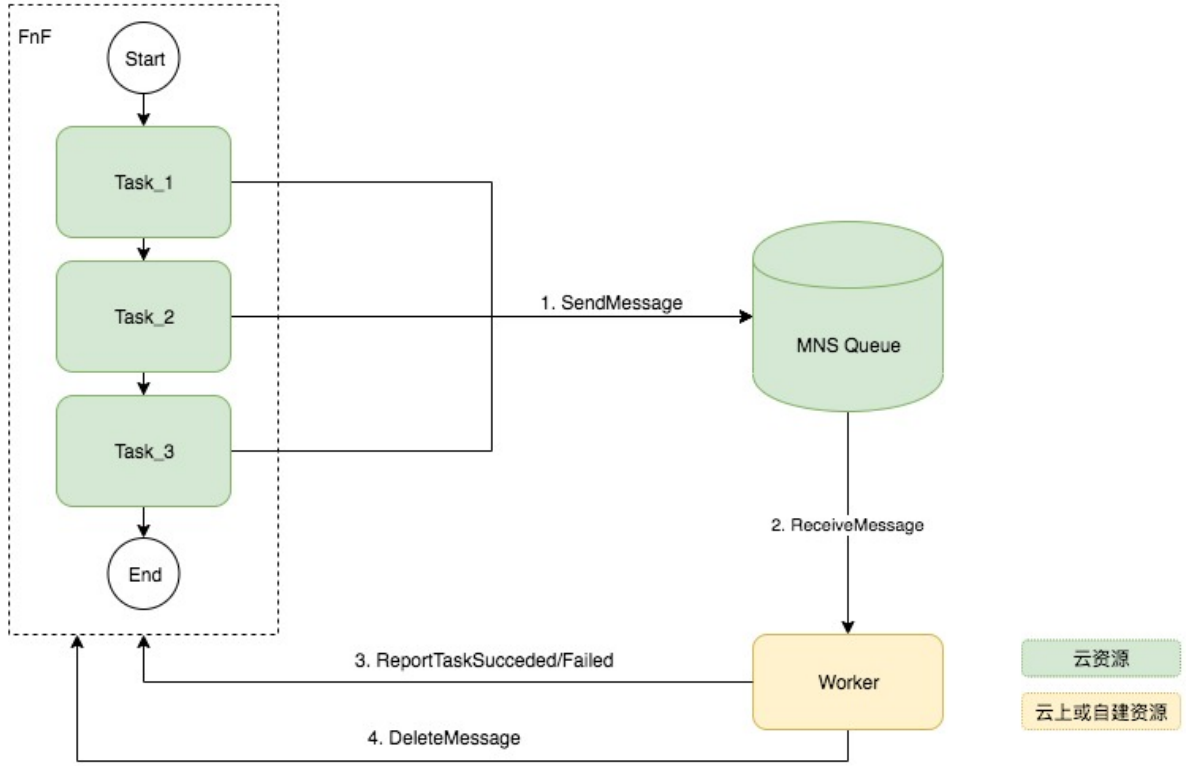
Serverless工作流的使用场景不仅限于编排函数计算FC（Function Compute）的FaaS函数，也包括广义上任意的计算任务。在上一篇最佳实践文档异步任务回调中介绍了利用函数计算的函数向MNS队列中发送消息，自定义环境中的任务执行者 （worker） 接收到消息，结合回调（callback）通知Serverless工作流任务执行结果。在下文中，将介绍如何使用Serverless工作流的新功能MNS队列。MNS队列进一步简化编排自定义任务类型。Serverless工作流可以直接向MNS的队列发送消息，省去了发送消息的函数计算的函数的开发、测试和维护，提高了可用性，降低了延迟。使用MNS集成相比通过函数计算的函数发送消息到MNS的做法有以下好处：

- 无需为发送消息做函数计算的函数的开发，降低了开发、测试和维护成本。
- 降低了消息传递的延时、少了一次远程访问、避免了函数计算的冷启动。
- 去除了一个服务依赖、提高了容错性。

Serverless工作流未来会推出更多的云服务集成，让不同类型任务组成的工作流编排变得更加容易。

服务集成功能

下图中3个串行的任务由Serverless工作流负责依次发送至用户指定的MNS队列中。消息发送成功之后Serverless工作流将会在该步骤暂定等待回调。用户在自定义环境中的worker（例如ECS VM、容器、自建机房内的机器）调用MNS `ReceiveMessage` 接口拉取消息。收到消息后，worker根据消息内容执行相应的任务。任务结束后，调用Serverless工作流 `ReportTaskSucceeded/Failed` 接口，Serverless工作流收到任务结果后继续该步骤执行。worker在汇报任务结果成功后删除MNS队列消息。



步骤详解

下文将详细介绍使用该功能的步骤。

1. 准备工作
2. 填写流程（Flow）
3. 编写worker
4. 执行并查看结果

步骤一：准备工作

1. 通过MNS控制台创建MNS队列，详细步骤请参见创建队列。
2. Serverless工作流需要扮演用户在Flow中指定的执行角色（RAM role）向用户账号下的MNS队列发送消息，因此需要为该RAM role添加MNS SendMessage相关的权限策略 (policy)，细粒度的策略示例如下。如没有细粒度权限控制的需求，可以通过Serverless工作流控制台向Flow RAM role添加系统策略 `AliyunMNSFullAccess` 。

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "mns:SendMessage"
      ],
      "Resource": [
        "acs:mns:$region:$account_id:/queues/$queue_name/messages"
      ]
    }
  ],
  "Version": "1"
}
```

步骤二：编写流程 (Flow)

下面的FDL是一个可以向fnf-demo这个MNS队列发送消息并且等待回调的任务（Task）步骤。

```
version: v1
type: flow
steps:
 - type: task
  name: Task_1
  resourceArn: acs:mns:::/queues/fnf-demo/messages # 表示该任务（Task）步骤会向同区域，同账号下的MNS队列fnf-demo发送消息。pattern: waitForCallback # 表示该任务步骤在发送MNS消息成功后会暂停，直到收到回调。inputMappings:
    - target: task_token
      source: $context.task.token # 从context对象中获取标识该任务的令牌 （task token）。
    - target: key
      source: value
  serviceParams: # 服务集成参数。
  MessageBody: $ # 用映射后的input作为要发送消息的内容。
  Priority: 1 # 消息队列的优先级。
```

步骤三：编写worker

下面的Python 2.7代码模拟一个执行任务的worker，它可以运行在任何可以访问到Serverless工作流和MNS服务的环境中。该worker长轮询调用MNS `ReceiveMessage`，当一个带有MNS配置的任务步骤进入时，Serverless工作流会向 `fnf-demo` 这个队列中发送一个消息。该worker执行相应任务结束后回调（callback）Serverless工作流 `ReportTaskSucceeded/Failed` 接口，在任务结果汇报完成后Serverless工作流会继续当前任务步骤执行，worker可以删除消息。

1. 在虚拟环境中，安装fnf、mns、Python SDK。

   ```
   cd /tmp; mkdir -p fnf-demo-callback; cd fnf-demo-callback
   virtualenv env; source env/bin/activate
   pip install -t . aliyun-python-sdk-core -t . aliyun-python-sdk-fnf -t . aliyun-mns
   ```

2. 编写本地任务执行者worker.py代码 。

```
import json
import os
from aliyunsdkcore.client import AcsClient
from aliyunsdkcore.acs_exception.exceptions import ServerException
from aliyunsdkcore.client import AcsClient
from aliyunsdkfnf.request.v20190315 import ReportTaskSucceededRequest
from mns.account import Account # pip install aliyun-mns
from mns.queue import *
def main():
 region = os.environ['REGION']
 account_id = os.environ['ACCOUNT_ID']
 ak_id = os.environ['AK_ID']
 ak_secret = os.environ['AK_SECRET']
 queue_name = "fnf-demo"
 fnf_client = AcsClient(
  ak_id,
  ak_secret,
  region
 )
 mns_endpoint = "https://%s.mns.%s.aliyuncs.com" % (account_id, region)
 my_account = Account(mns_endpoint, ak_id, ak_secret)
 my_queue = my_account.get_queue("fnf-demo")
 my_queue.set_encoding(False)
 wait_seconds = 10
 try:
  while True:
   try:
    print "Receiving messages"
    recv_msg = my_queue.receive_message(wait_seconds)
    print "Received message %s, body %s" % (recv_msg.message_id, recv_msg.message_body)
    body = json.loads(recv_msg.message_body)
    task_token = body["task_token"]
    output = "{\"key\": \"value\"}"
    request = ReportTaskSucceededRequest.ReportTaskSucceededRequest()
    request.set_Output(output)
    request.set_TaskToken(task_token)
    resp = fnf_client.do_action_with_exception(request)
    print "Report task succeeded finished"
    my_queue.delete_message(recv_msg.receipt_handle)
    print "Deleted message " + recv_msg.message_id
   except MNSExceptionBase as e:
    print(e)
   except ServerException as e:
    print(e)
    if e.error_code == 'TaskAlreadyCompleted':
     my_queue.delete_message(recv_msg.receipt_handle)
     print "Task already completed, deleted message " + recv_msg.message_id
 except ServerException as e:
  print(e)
if __name__ == '__main__':
  main()
```

3. 开启worker，长轮询fnf-demo队列，收到消息后回调Serverless工作流汇报结果。

```
# 执行worker进程。export REGION={your-region}
export ACCOUNT_ID={your-account-id}
export AK_ID={your-ak-id}
export AK_SECRET={your-ak-secret}
python worker.py
```

步骤四：执行并查看结果

在Serverless工作流控制台开始一个流程的执行，配合worker可以看到流程成功执行。

5.异步任务回调

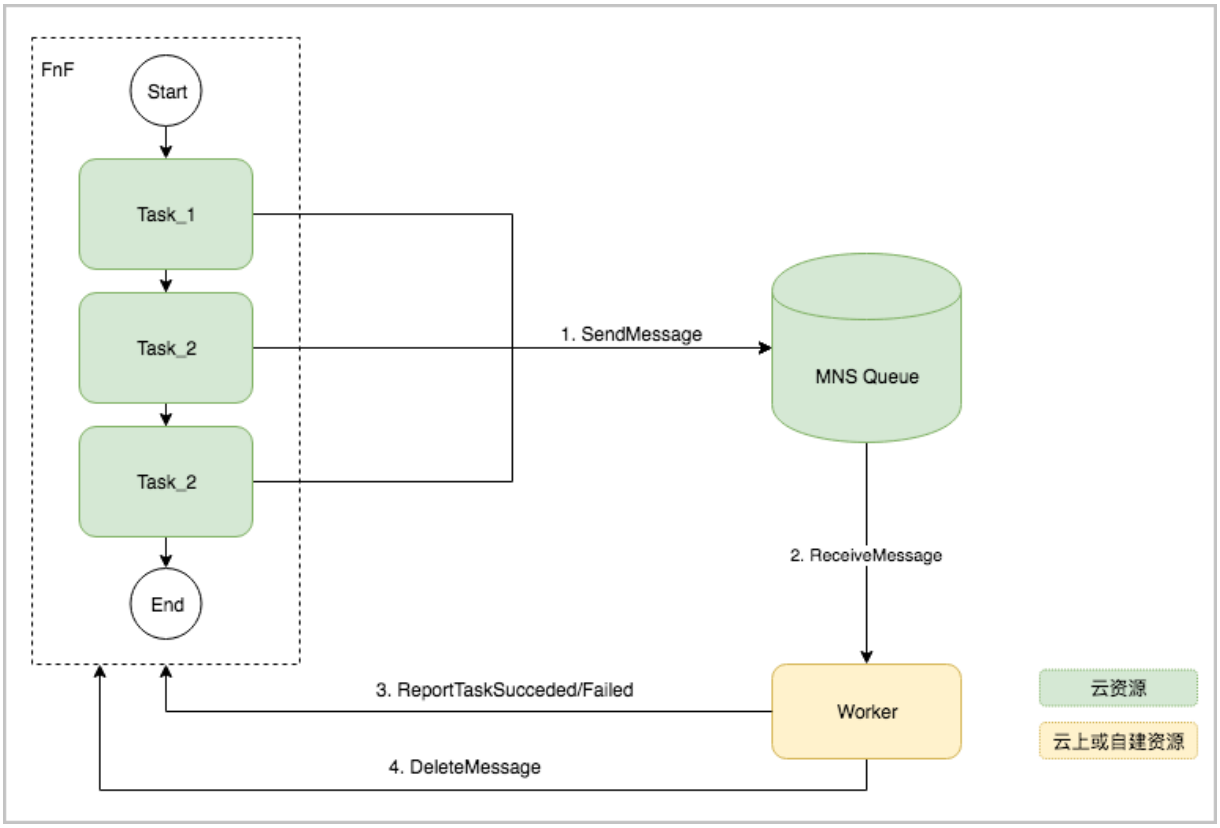
本文介绍了Serverless工作流的回调功能。相比较轮询，使用回调有效地降低了延迟、减少了轮询对服务器造成的不必要压力。另外，回调功能配合队列可以实现对非FC任务的编排，将Serverless工作流的编排范围扩展到任意类型的计算资源。

简介

长时间执行的任务通常会采用异步提交任务并返回任务标识（ID），判断异步任务结束的方法通常有两种：轮询（polling）和回调（callback），在任务状态轮询中我们介绍了使用轮询来判断任务结束。Serverless工作流的回调（callback）功能，覆盖以下的痛点或场景：

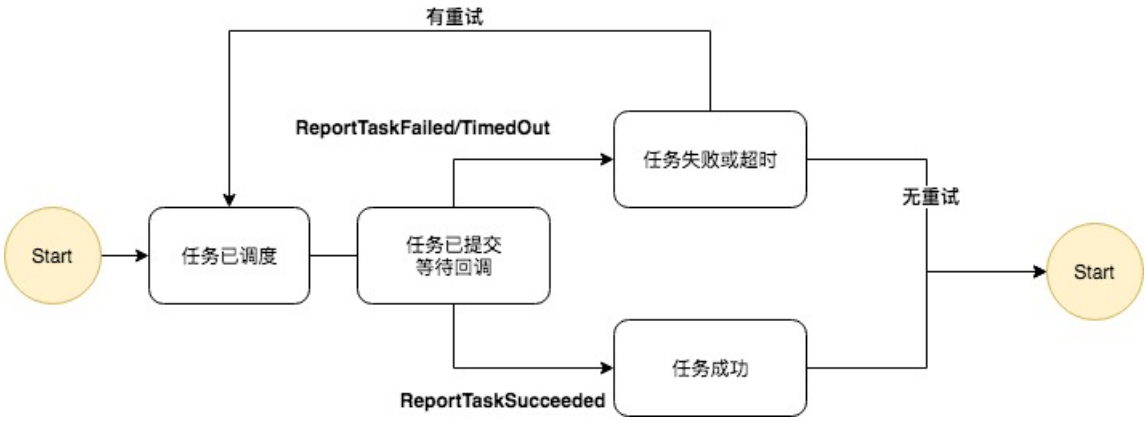
- 消除轮询周期长带来的不必要延迟。
- 消除大流量场景下高并发的轮询造成不必要的服务器资源压力和浪费。
- 编排非FC Function的任务，例如运行在自建机房或ECS上的进程。
- 需要人工干预的步骤，例如通知审批通过。

下图展示了使用MNS队列服务集成结合回调API编排自建资源，拓宽Serverless工作流的适用场景。



回调使用详解

在Task步骤中指定 `pattern: waitForCallback` ，如下图状态机所示：该步骤会在提交 `resourceArn` 指定的任务后（如FC invocation）该步骤会将一个 `taskToken` 存入到该步骤的 `context` 对象并进入一个暂停的状态，直到Serverless工作流收到回调或指定的任务超时。将 `taskToken` 传入 `ReportTaskSucceed` 或 `ReportTaskFailed` 接口去回调会使得该步骤继续执行。




```
- type: task
  name: mytask
  resourceArn: acs:fc:::services/{fc-service}/functions/{fc-function}
  pattern: waitForCallback # 指定该Task步骤在提交任务后等待回调。
  inputMappings:
   - target: taskToken
     source: $context.task.token # 将context中的taskToken作为input传入resourceArn指定的函数。
  outputMappings:
   - target: k
     source: $local.key # 将ReportTaskSucceeded中output {"key": "value"}映射成 {"k": "value"}并作为该步骤的输出。
```

示例

该示例共分为以下3个步骤：

1. 准备Task Function
2. 开始工作流
3. 回调

步骤1：准备Task Function

1. 创建下面一个简单的函数，该函数会将输入直接返回。
   - 服务：fnf-demo。
   - 函数：echo。
   - 运行环境：python2.7。
   - 函数入口：index.handler。

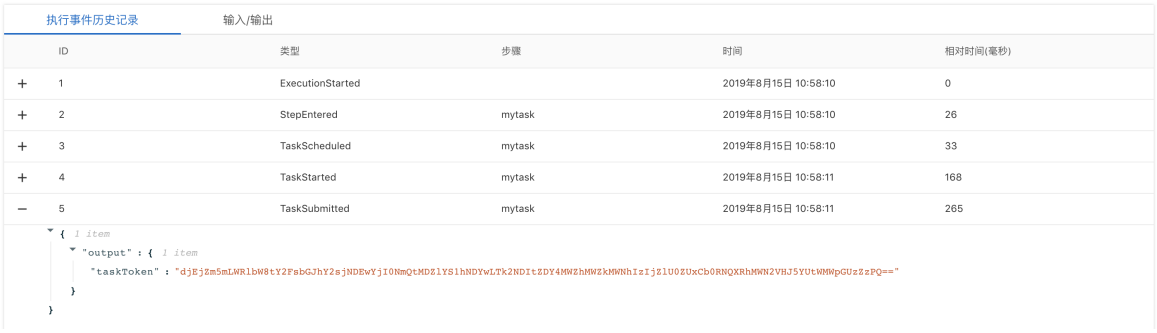
```
#!/usr/bin/env python
import json
def handler(event, context):
  return event
```

步骤2：开始工作流

1. 创建流程，并开始执行。
   - 流程名称：fnf-demo-callback。
   - 流程角色：配置一个有FC Invocation权限的角色。

```
version: v1
type: flow
steps:
 - type: task
   name: mytask
   resourceArn: acs:fc:::services/fnf-demo/functions/echo
   pattern: waitForCallback
   inputMappings:
    - target: taskToken
      source: $context.task.token
   outputMappings:
    - target: s
      source: $local.status
```

流程开始后可以看到 mytask 步骤暂停在 TaskSubmitted 事件，等待回调。该事件的 output 中含有 taskToken 作为回调任务的标识。



步骤3：回调

1. 使用Serverless工作流的Python SDK在本地（或其他可以运行Python的环境）运行 callback.py 脚本，将{task-token}替换为 TaskSubmitted 事件中的值。

```
cd /tmp
mkdir fnf-demo-callback
cd fnf-demo-callback
# 在虚拟环境中，安装fnf python SDK。
virtualenv env
source env/bin/activate
pip install -t . aliyun-python-sdk-core
pip install -t . aliyun-python-sdk-fnf
# 执行worker进程。
export ACCOUNT_ID={your-account-id}; export AK_ID={your-ak-id}; export AK_SECRET={your-ak-secret}
python worker.py {task-token-from-TaskSubmitted}
```

```
# worker.py 代码:
import os
import sys
from aliyunsdkcore.acs_exception.exceptions import ServerException
from aliyunsdkcore.client import AcsClient
from aliyunsdkfnf.request.v20190315 import ReportTaskSucceededRequest
def main():
 account_id = os.environ['ACCOUNT_ID']
 akid = os.environ['AK_ID']
 ak_secret = os.environ['AK_SECRET']
 fnf_client = AcsClient(akid, ak_secret, "cn-hangzhou")
 task_token = sys.argv[1]
 print "task token " + task_token
 try:
  request = ReportTaskSucceededRequest.ReportTaskSucceededRequest()
  request.set_Output("{\"status\": \"ok\"}")
  request.set_TaskToken(task_token)
  resp = fnf_client.do_action_with_exception(request)
  print "Report task succeeded finished"
 except ServerException as e:
  print(e)
if __name__ == '__main__':
  main()
```

上述脚本回调成功后可以看到 `mytask` 步骤继续执行，`ReportTaskSucceeded` 中指定的输出"{"status": "ok"}"经过outputMappings的映射后变成"{"s": "ok"}"。

6.工作流调度固定版本或预留资源函数

本文介绍了工作流调度固定版本或预留资源函数和Serverless工作流实现的具体步骤。

简介

在实际生产场景中，任务流所调度的函数因为业务场景变化可能需要频繁变更，您会考虑到如何避免变更带来的非预期行为，控制变更稳定性。在Serverless工作流任务步骤使用固定版本的函数将对以下场景产生帮助：

- 流程A编排了多个函数f1、f2和f3，同一次任务必须执行函数的相同版本。假如流程A正在执行中，已执行完了函数f1，但是此时进行了函数更新，则正在执行的流程可能会执行函数f2、f3的最新版本造成非预期情况发生。那么必须保证，每次流程执行函数的版本在执行时已经固定。
- 某个函数的快速回滚。上线后，发现流程执行失败是由于新变更引起的，那么需要快速回滚流程执行上一固定版本。
- 通过函数别名调用预留资源函数、降低函数冷启动时间及优化函数成本。

函数计算版本及别名可以有效支持类似上述场景的持续集成、持续发布的各种需求。下文将通过一个具体示例展示如何在流程中使用别名调用预留资源函数。预留资源函数是依赖于固定版本函数的，因此对于其他需要固定版本的场景，也可参见本示例操作。

流程

本次示例操作包含以下步骤：

1. 创建函数预留实例
2. 创建工作流
3. 使用命令行或控制台执行并观察预留函数执行

步骤一：创建函数预留实例

1. 首先创建一个名为 `fnf-demo` 的函数计算服务，并在该服务下创建一个名为provision的Python3函数并发布版本、别名，生成预留实例，详细步骤，请参见弹性伸缩（含预留资源）。

   假设创建的函数版本为1，别名为online，并发布了一个预留实例。函数内容如下。

```
import logging
def handler(event, context):
 logger = logging.getLogger()
 logger.info('Started function test')
 return {"success": True}
```

步骤二：创建工作流

Serverless工作流对函数计算版本及别名进行了原生支持。

在Serverless工作流的Task步骤中，默认填写的 `resourceArn` 一般为 `acs:fc:{region}:{accID}:services/fnf/functions/test` 。按照函数执行规则，默认会选择最新的函数版本执行。您可以发布版本或别名，并在工作流Task步骤中填写 `resourceArn` 一项为 `acs:fc:{region}:{accID}:services/fnf.{别名or版本}/functions/test` 来实现对特定版本函数的调用。因此，流程可以定义如下：

```
version: v1
type: flow
steps:
 - type: task
   resourceArn: acs:fc:::services/fnf-demo.online/functions/provision
   # 也可以使用版本号，resourceArn: acs:fc:::services/fnf-demo.1/functions/provision。
   name: TestFCProvision
```

步骤三：使用命令行或控制台执行并观察预留函数执行

1. 启动工作流执行。使用预留模式前。

+	4	TaskStarted	TestFCProvision	2019年11月15日 17:35:31	57
+	5	TaskSucceeded	TestFCProvision	2019年11月15日 17:35:31	564

2. 使用预留模式后。

+	4	TaskStarted	TestFCProvision	2019年11月15日 17:44:40	61
+	5	TaskSucceeded	TestFCProvision	2019年11月15日 17:44:40	298

从工作流的执行时间来看，在task步骤从未使用预留模式前的耗时500 ms缩短为预留模式后的230 ms。

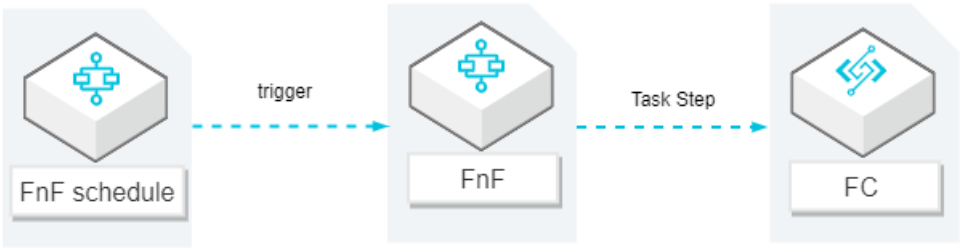
7.定时触发工作流

本文介绍如何使用Serverless工作流提供的定时调度功能，来定时执行工作流以调用函数计算FC（Function Compute）的函数。

执行流程

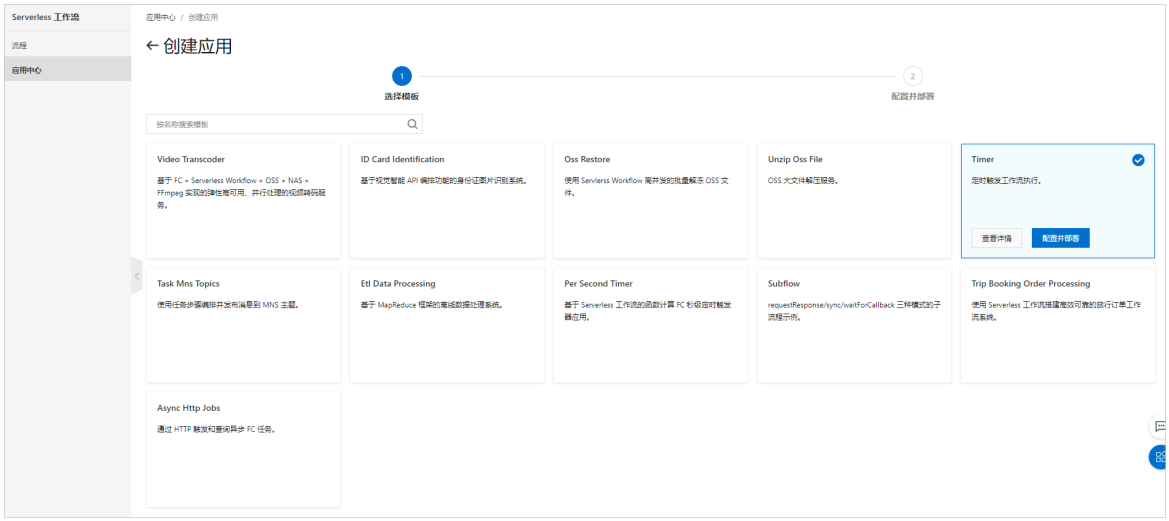
定时触发工作流的执行流程如下：

1. 在Serverless工作流定义调用函数计算的函数的任务步骤。
2. 在Serverless工作流创建定时调度，工作流被定时执行，然后任务步骤中的函数也会被定时执行。

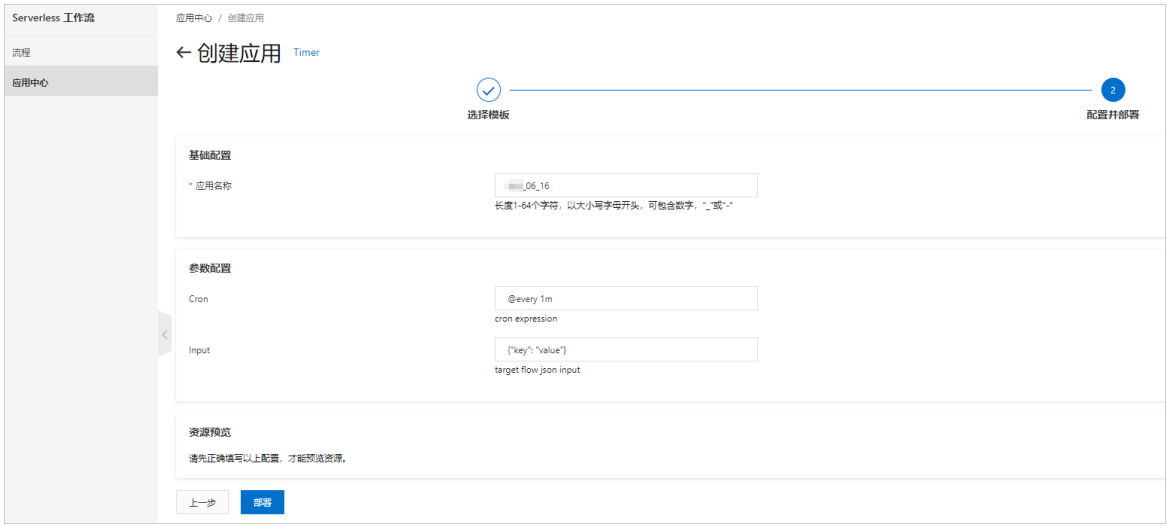


操作步骤

1. 登录Serverless工作流控制台。
2. 在顶部菜单栏，选择地域。
3. 在左侧导航栏，单击**应用中心**。
4. 在**应用中心**页面，单击**创建应用**。
5. 在**创建应用**页面的**选择模板**页签，选择Timer模板，然后单击**配置并部署**。



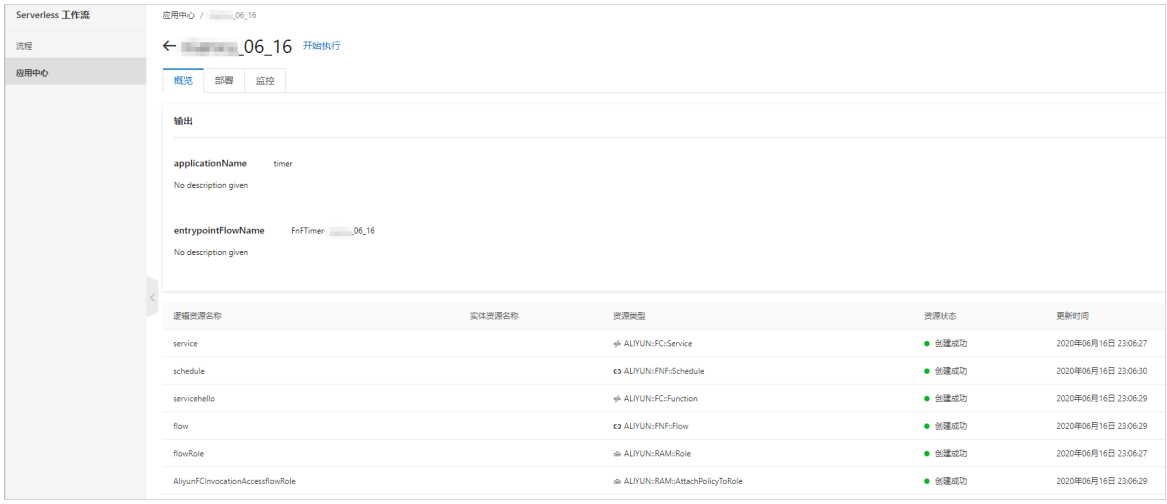
6. 在**配置并部署**页签，配置相应参数，单击**部署**。



参数说明如下。

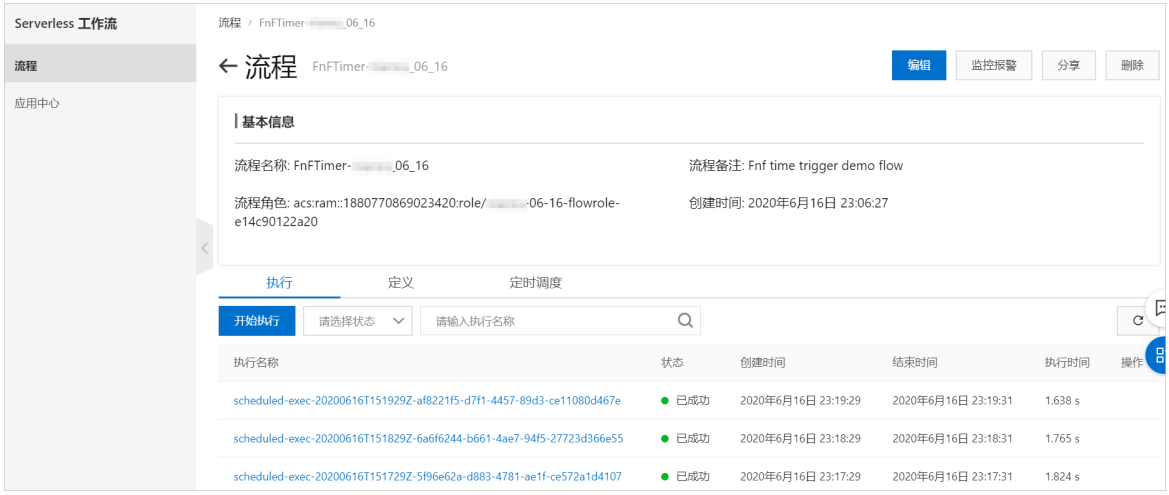
参数	说明
应用名称	您的应用名称，同一个账号下需保证唯一。 **说明** 您的应用是自定义的资源编排服务ROS资源，可登录至资源编排服务ROS控制台查看。
Cron	定时调度工作流的Cron表达式，详情请参见调度时间参数说明。
Input	定时调度工作流的输入，必须为JSON格式，默认为空。详情请参见Input格式。

部署成功后可看到应用创建的所有资源。



- RAM角色：函数调用权限AliyunFCInvocationAccessflowRole、工作流权限flowRole。
- 函数计算资源：服务service、函数servicehello。
- Serverless工作流资源：工作流flow、定时调度ALIYUN::FNF::Schedule。

7. 登录Serverless工作流控制台，查看您刚创建的工作流已成功地被定时触发。



通过任务步骤调用函数计算的函数 `hello` 的示例工作流定义如下。

```
version: v1
type: flow
steps:
 # task step to invoke FC function hello
 - type: task
   name: hello
   resourceArn: acs:fc:::services/service-CD946B9A9F36/functions/hello
```

您可以修改该工作流的定义实现自身的业务逻辑。详情请参见修改流程。

8.使用函数计算实现游戏渠道包构建

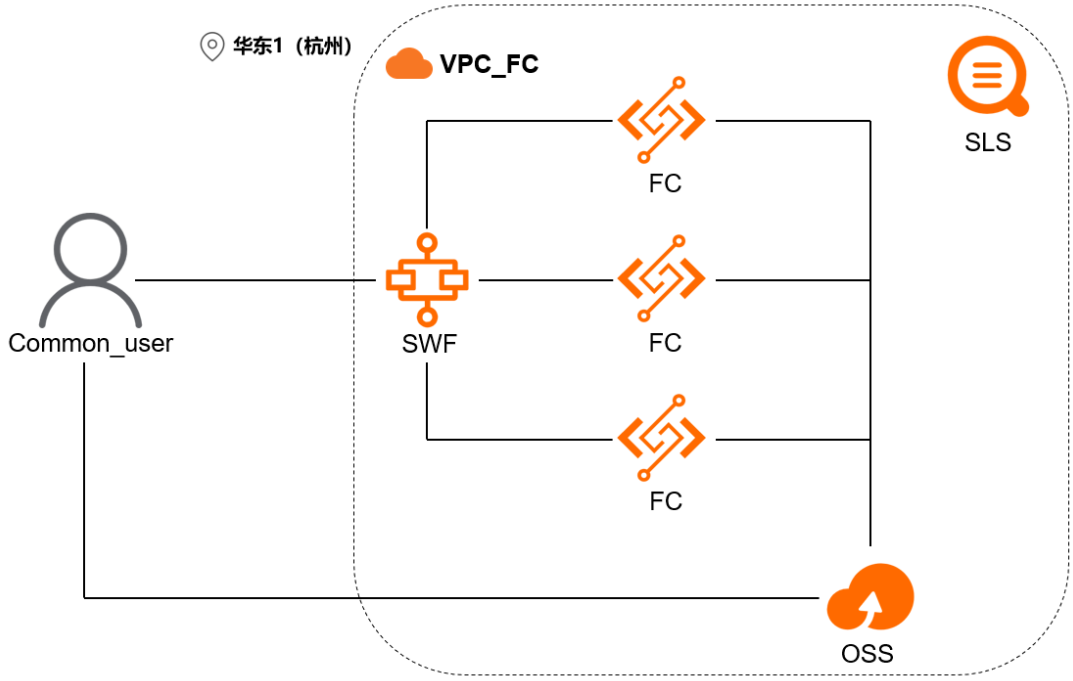
本文介绍如何使用Serverless工作流、函数计算、对象存储和日志服务的组合方案，实现游戏在发行过程中自动化、并行化一键式地构建游戏渠道包。

应用场景

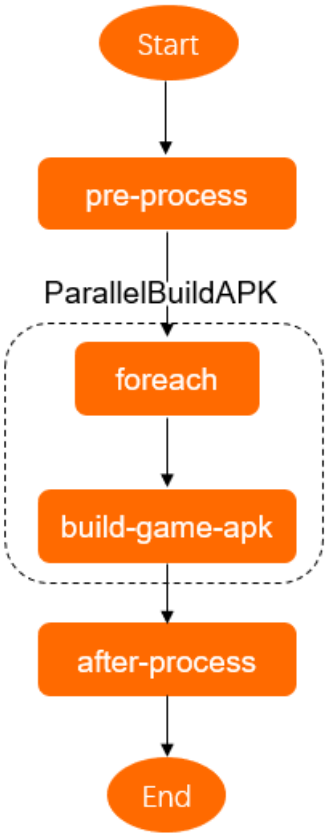
游戏发行商在发行游戏之前，通常会针对不同的发行渠道，将游戏母包和不同的渠道资料包构建成渠道包。在这个构建的过程中，部分厂商会通过手工方式进行构建，但手工方式构建存在易出错、低效率和高成本等缺点。

方案简介

方案架构图



方案流程图



方案流程如下：

1. pre-process：从输入参数中获取并整理对象存储、签名等信息。
2. build-game-akp：从对象存储拉取母包、apktool和渠道资料包等，并发执行打包。
3. after-process：执行清理工作。

方案优势

- 自动化：事件一键触发、自动运行，无需人工干预，Serverless工作流完整跟踪记录整个打包流程，提高成功率。
- 免运维：函数执行级别的监控、告警和日志。
- 高效率：多个渠道包构建过程同时进行，配置模板化，无需修改代码，提高打包效率。
- 低成本：基于函数计算资源利用率高的优势，可以极大程度地降低计算资源成本。

方案详情

具体步骤，请参见基于函数计算的游戏打包。

9.音视频弹性处理

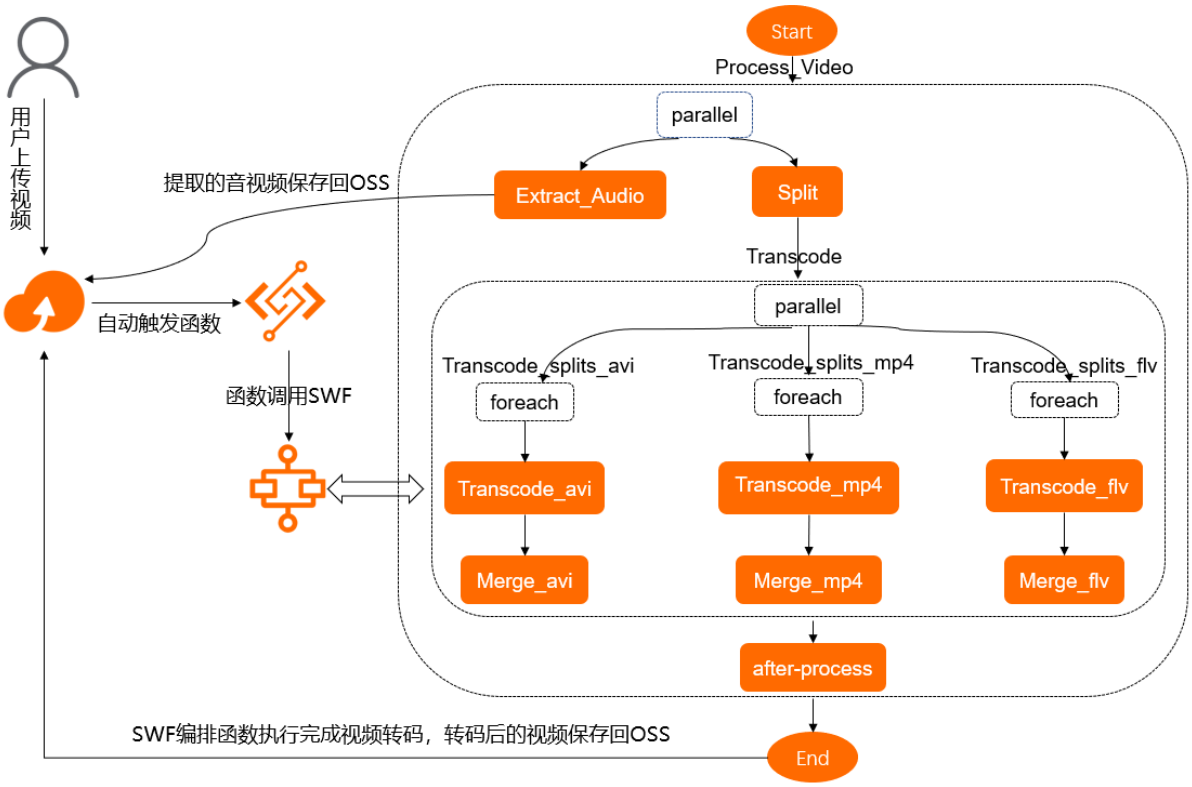
本文介绍如何通过Serverless工作流、函数计算、文件存储、对象存储、及日志服务的组合方案，部署一个高弹性高可用的音视频处理系统。

应用场景

- 有大量的视频需要上传。
- 上传的视频需要及时处理去适配各种终端及网络条件。
- 可以在短时间内准备大量的计算资源进行大规模并行转码处理。
- 能简单迁移基于FFmpeg自建的转码服务。

方案简介

本示例是将MOV格式文件转换为FLV、AVI、MP4格式的文件，并将转换后的文件存储到对象存储的指定目录中。



方案实施如下：

1. 上传视频文件到对象存储指定目录下。
2. 文件上传到对象存储后，Serverless工作流及函数计算的OSS触发器会自动触发函数计算服务。
3. 函数计算调用Serverless工作流的定制流程，自动处理音视频文件。

方案优势

- 快速迁移：基于FFmpeg自建的转码服务，Serverless工作流及函数计算支持您的命令无缝迁移，FFmpeg的版本也可以自定义。
- 弹性高可用：Serverless工作流及函数计算可以快速调动大量计算资源加速、并行转码。

- 自定义Serverless工作流：Serverless工作流不仅可以实现高度自定义，例如并行转码、打水印、元信息插入数据库等复杂组合操作，还可以实现Serverless工作流的安全升级更新。
- 降低成本：视频转码是CPU密集型，在Serverless工作流及函数计算的资源利用率高的情况下，实现了转码成本的降低。
- 提升效率：降低学习和使用成本例如不用学习新的语言或其他云产品，极大程度上缩短了项目周期，加快开发部署。

方案详情

具体步骤，请参见函数计算实现弹性音视频处理系统。

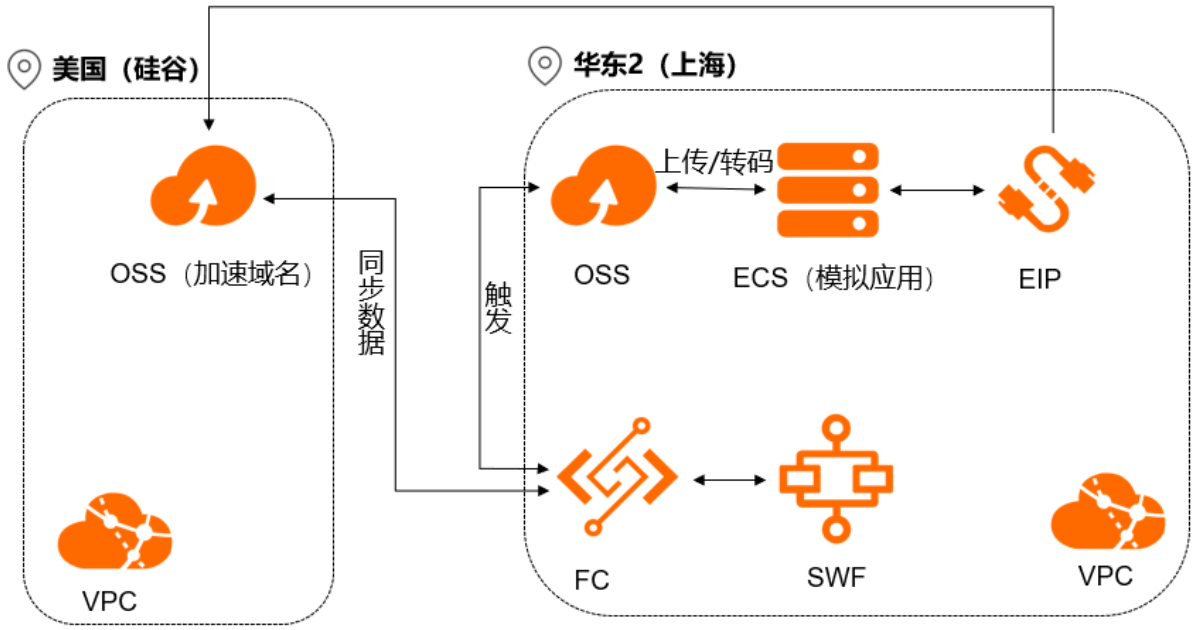
10.低成本跨境文件传输

本文介绍如何通过Serverless工作流、函数计算、对象存储、专用网络、云服务器及弹性公网IP的组合方案，实现低成本跨境文件的传输。

应用场景

- 跨境对象存储数据传输。
- 对数据的跨境传输成本有严格的控制。
- 对文件同步延时不敏感，能接受一定程度上网络抖动带来同步文件时的延迟。
- 希望传输数据的系统有足够的弹性和扩展性应对大规模文件的写入。

方案简介



1. 云服务器模拟应用访问华东2（上海）和美国（硅谷）的对象存储，负责内容的制作和上传。
2. 内容通过内网域名上传到华东2（上海）的对象存储。
3. 文件完成上传到对象存储后触发函数计算调用Serverless工作流，Serverless工作流内的一个步骤负责一个文件的上传，保障文件通过美国（硅谷）的对象存储加速域名被快速同步。
4. 云服务器模拟美国（硅谷）当地用户读取对象存储文件，并校验数据是否正确。

方案优势

- 运维低成本：开发人员只需关注代码逻辑。
- 网络成本低：相比云企业高速通道的方式，降低了网络成本。
- 同步服务部署成本低：文件需要进行传输触发函数计算任务时，无需准备云服务器。
- 弹性高效：一个文件同步触发一个Serverless工作流任务，充分利用资源且高效同步。

方案详情

具体步骤，请参见低成本跨境文件传输。

11.错误处理

Serverless工作流支持与多个云服务集成，当使用云服务作为Serverless工作流任务步骤的执行节点时，您可以根据业务场景对执行的错误进行重试或捕获处理，使您的任务在生产场景中更稳定地运行。本文介绍错误处理的方式及如何在不同的业务场景中进行错误处理的最佳实践。

错误处理方式

Serverless工作流的任务步骤不仅支持对错误的捕获，而且也支持对错误捕获后的处理，例如重试及跳转。更多信息，请参见任务步骤。

- 错误重试。

```
steps:
 - type: task
   name: hello
   resourceArn: acs:fc:{region}:{accountID}:xxx
   retry:
    - errors:
       - FnF.ALL
      intervalSeconds: 10
      maxIntervalSeconds: 300
      maxAttempts: 3
      multiplier: 2
```

错误重试的参数说明

参数	描述
retry	表示该任务步骤错误处理类型为重试。
errors	表示需要捕获的错误列表。
intervalSeconds	表示重试的初始间隔时间，最大值是86400秒，默认值是1秒。单位：秒。
maxAttempts	表示最多重试次数，默认值是3次。
multiplier	表示后一次重试比前一次重试间隔时间的倍数，默认值是2。按照上文的代码示例，第二次重试时间间隔为20秒，第三次重试时间间隔为40秒。

- 错误跳转。

```
steps:
 - type: task
   name: hello
   resourceArn: acs:fc:{region}:{accountID}:xxx
   errorMappings:
    - target: errMsg
      source: $local.cause #该值为系统预留，在本步骤发生错误时可直接使用。
    - target: errCode
      source: $local.error #该值为系统预留，在本步骤发生错误时可直接使用。
   catch:
    - errors:
      - FnF.ALL
      goto: final
```

错误跳转的参数说明

参数	描述
errorMappings	指定跳转支持传递本步骤中的错误字段。
catch	表示该任务节点的捕获策略。
errors	表示需要捕获的错误列表。
goto	表示当任务抛出对应错误后，进行跳转的对象。

使用函数计算作为Serverless工作流的执行节点

当使用函数计算作为Serverless工作流的执行节点时，您需要关注以下错误类型：

- 函数计算系统提示的异常错误。
- 函数代码错误。

这些错误可以在Serverless工作流的 errors 的任务中进行捕获。

常见的错误类型如下：

```
- errors:
  - FC.ResourceThrottled
  - FC.ResourceExhausted
  - FC.InternalServerError
  - FC.Unknown
  - FnF.TaskTimeout
  - FnF.ALL
```

常见的错误类型

错误类型	描述

错误类型	描述
FC.{ErrorCode}	函数计算服务返回除200的HTTP Code。常见的错误类型如下所示： • FC.ResourceThrottled：您的函数因为并发度过高被限流。您所有的函数由一个总的并发度控制。Serverless工作流执行到任务类型节点时会同步调用函数计算，此数值与其他调用方式的并发度共用。您可以申请调整该值。 • FC.ResourceExhausted：您的函数因为资源不足被限流。当出现这类错误时，请联系我们。 • FC.InternalServerError：函数计算出现系统错误，请重新执行流程。 说明 {Error code}是函数计算的错误码。详细信息，请参见错误码列表。
FC.Unknown	函数计算服务调用函数成功，但函数执行出错，且该错误码未被捕获，例如UnhandledInvocationError等。
{CustomError}	函数计算服务调用函数成功，但函数主动抛出异常错误。
FnF.TaskTimeout	Serverless工作流某步骤执行超时。
FnF.ALL	捕获Serverless工作流系统的所有错误。
FnF.Timeout	Serverless工作流整体执行超时。

除了函数计算和Serverless工作流系统常见的错误类型外，您也可以自定义错误类型，在函数代码中主动抛出异常，方便将函数执行的状态或错误传递给Serverless工作流，然后Serverless工作流再根据流程对任务进行重试或跳转。下文以Python为例介绍如何在函数代码中自定义错误类型，并在Serverless工作流的任务中进行重试处理。具体步骤如下：

1. 自定义函数代码。

```
...
class ErrorNeedsRetry(Exception):
  pass
def handler(event, context):
  try:
    # do sth
  except ServerException:
    raise ErrorNeedsRetry("custom error message")
```

2. 按需修改Serverless工作流的任务流程进行函数计算的错误捕获及重试。

```
retry:
 - errors:
   - ErrorNeedsRetry
 intervalSeconds: 10
 maxAttempts: 3
 multiplier: 2
```

函数计算或Serverless工作流系统常见的系统错误

自定义代码错误

使用MNS等其他云服务作为任务的执行节点

当使用其他第三方云服务作为任务的执行节点时，Serverless工作流将直接调用相应服务的API对任务进行下发。

以MNS为例，当Serverless工作流需要发送消息时，将调用MNS的SendMessage接口。详细信息，请参见SendMessage。这类任务通常是API调用，不存在等待函数执行结果的情况，且Serverless工作流内部会对可重试的错误进行多次合理重试。因此，当您使用MNS、视觉智能API等云服务作为任务的执行节点时，在流程中您无需关注错误处理。