ActionChain

ActionChain是一个简洁的线性工作流程。这是一个简单的操作链调用。检查每个操作的结果（成功或失败）并确定下一个要运行的操作。这提供了基于成功或失败的简单分支逻辑。

数据可以在动作之间传递，每个动作都会发布结果。

从StackStorm的角度来看，ActionChain本身就是一个动作。因此，它具有相同的操作和功能，例如定义，注册，CLI执行，规则使用等。ActionChain也可以被称为来自另一个ActionChain或来自Mistral工作流的Action。

注意

如果您需要更复杂的工作流逻辑，例如分叉，加入，重试，延迟和错误处理策略，请使用Mistral。

创建一个ActionChain

ActionChains在`/opt/stackstorm/packs/<pack>/actions/`目录下的包中定义。一个ActionChain需要两个文件：一个YAML元数据文件，和简单动作一样，以及ActionChain定义本身。元数据文件位于`<pack>/actions`目录中，ActionChain定义位于`<pack>/actions/chains`目录中。

ActionChain元数据

ActionChain元数据与用于任何其他操作的元数据非常相似。关键的区别在于它将工作流定义文件指定`action-chain`为`runner_type`和`entry_point`点。

这里是表示动作定义的元数据的示例echochain.meta.yaml用于ActionChain echochain.yaml：

```
---
#动作定义元数据
名称： “echochain”
描述： “简单动作链工作流程”

#`runner_type`具有`action-chain`的值来标识动作是一个ActionChain。
runner_type : “动作链”

ActionChain定义文件的``entry_point`路径，相对于包的动作目录。
entry_point : “chains / echochain.yaml”

enabled : true
参数：
  skip_notify :
    默认值：
      - c2
notify :
  on-complete :
    message : “\”@channel：Action succeeded。\“” routes :- “slack”
```

ActionChain定义

这个echochain.yaml：是上面引用的相应的ActionChain工作流定义：

```
---
    链：
        -
            名称： “c1”
            ref ： “core.local”
            参数：
                cmd ： “echo c1” on-success ：“c2” on-failure ：“c4” - name ：“c2” ref
：“core .local“ 参数：cmd ：”echo “c2：父执行程序是{{action_context.parent.execution_id}}。”“
on-success ：”c3“ on-failure ：”c4“- 名称：









             “c3”
            ref ： “core.local”
            参数：
                cmd ： “echo c3” on-failure ：“c4” - name ：“c4” ref ：“core.local” 参数：
cmd ：“echo fail c4” default ：“ C1"
```

定义细节：

- `chain` 是包含任务的数组属性，它封装了动作调用。
- 任务被命名为以列表形式提供的操作执行规范。该名称的作用范围是一个ActionChain，并用作对任务的引用。
- `ref` 任务的属性指向在StackStorm中注册的Action。这可以在任何包中。
- `on-success` 是成功执行动作后下一个要调用的任务的链接。如果没有提供，ActionChain将终止，状态设置为 `success` 。
- `on-failure` 是一个可选的链接，用于在执行失败的情况下调用下一个任务。如果没有提供，ActionChain将终止并且状态设置为 `error` 。
- `default` 是一个可选的顶级属性，用于指定ActionChain的开始。如果 `default` 没有明确指定，ActionChain从第一个动作开始。

注册ActionChain

一旦创建操作定义和元数据文件，就加载操作：

```
#注册动作
st2动作创建/opt/stackstorm/packs/examples/actions/echochain.meta.yaml
#检查它是否可用
st2 action list --pack = examples
 #运行它
st2运行examples.echochain
```

ActionChain工作流定义中的任何更改都会自动提取。但是，如果您更改操作元数据（例如重命名或添加参数），则必须使用更新操作 。或者，完全重新加载 将会提取所有更改。 `st2 action update <action.ref> <action.metadata.file>` `st2ctl reload --register-all`

提供输入

为了向ActionChain提供输入，必须在动作元数据中定义输入参数：

```
---
#动作定义元数据
名称： "echochain-param"
描述： "动作链工作流程传递变量" #"runner_type"标识动作链定义文件的跑步者runner_type ："action-
chain" #"entry_point"路径，相对于该包的动作目录。entry_point ：启用"chains /
echochain_param.yaml" ：true 参数：input1 ：type ："string" required ：true 描述："Any
字符串来传递下来"
```

`input1` 现在可以在ActionChain定义中的任务的参数字段中引用输入参数：

```
---
   #...
   链：
      -
         名称： "action1"
         ref ： "core.local"
         参数：
            action1_input ： "{{input1}}"
   #...
```

`action1_input` 有价值 `{{input1}}` 。这个语法是由Jinja模板支持的可变引用 。

规则标准和操作字段中也使用了类似的构造。

变量

ActionChain提供了命名变量的便利。全局变量被设置在 `var` 关键字定义的顶部。

任务使用 `publish` 关键字发布新变量。当需要混合输入，全局变量，数据存储区值和多个操作执行结果中的可重用值时，变量非常方便。

所有变量都使用Jinja语法来引用。 `published` 如果将 `display_published` 属性提供给ActionChain Runner，则累积发布的变量也可在属性 下的ActionChain执行结果中使用。

```
---
vars :
    domain : “{{ st2kv.system.domain }}” #全局变量端口：9101


链：
    -
        名称： get_service_data
        ref :  my_pack.get_services
        publish :
            url_1 : http：//“{{get_service_data.result [0] .host.name}}。{{domain}}：
{{port}}”
```

*示例*包中的publish_data.yaml工作流程显示了使用和的完整工作示例：`vars` `publish`

```
---
vars :
    domain : “{{ st2kv.system.domain }}” #`system`引用DataStore键值。如果没有设置，则为空。
api_port ：9101 webui_port ：8080 链：- 名称：“get_host” ref ：“core.local” 参数：cmd ：
hostname | tr -d'\ n' 发布：#发布一个变量来快速生成一个长表达式api_url ：“http：// {{
get_host.stdout }}：{{ api_port }}” webui_url :

                “http：// {{ get_host.stdout }}：{{ webui_port }}” host ：“{{ get_host.stdout
}}” #用于基本逻辑的Jinja woodoo fqdn ：“{{ get_host.stdout }} {%if 域%} {{。域}} {%ENDIF
%} “ #A复杂的对象可以被发布路径：API ：[ ”V1 /动作“ ，”V1 /触发器“ ] WebUI中：”/“ auth_port
：”{{ api_port + 1 }}“

                    #该变量引用作为动作参数
                    system_info的“system_info_path”变量： “{{ system_info_path }}”

        成功： say_the_names
    -
        名称： say_the_names
```

在任务之间传递数据

以前任务的输出可以以与输入到ActionChain类似的方式引用。

这个示例echochain_param.yaml 显示了向下传递工作流的输入和数据：

```
---
    链：
        -
            名称： “c1”
            ref ： “core.local”
            参数：
                cmd ： “echo c1 , input {{input1}}” #引用动作的输入参数on-success ：“c2” on-
failure ：“c4” - name ：“c2” ref ：“core.local” 参数：cmd ：“echo c2 {{c1.stdout}}” #引用上
一个动作的成功输出：“c3”失败：









             “c4”
        -
            名称： “c3”
            ref ： “core.local”
            参数：
                cmd ： “echo c3 {{c2.stdout}}” on-failure ：“c4” - name ：“c4” ref
：“core.local “ 参数：cmd ：”echo fail c4“ 默认值：”c1“
```

细节：

- 任务输出总是以任务名称为前缀。例如in ， 指的是'c1'的输出，并进一步深入到输出的属性中。参考点是对象的字段。 `{"cmd":"echo c2 {{c1.stdout}}"}` `c1.stdout` `result` `action execution`
- 一个特殊的 `__results` 键可以访问整个链的整个结果直到执行点。

在不同的工作流程之间传递数据

在StackStorm中，工作流只是一个动作。这意味着您将数据从一个工作流传递到另一个工作流的方式完全相同，您可以将数据传递给某个操作 - 使用操作参数。

在下面的例子中，我们有两个工作流程 - `workflow1` 并且 `workflow2` 。任务命名为 `task2` 内部 `workflow1` 调用 `workflow2` ，并将该变量 `date` 作为操作参数传递给它。 `workflow2` 然后使用此值并将其打印到标准输出。

`workflow1.yaml`

```
---
    chain :
        -
            name : “
            task1 ” ref : “core.local”
            参数：
                cmd : “date”
            on-success : “task2”
        -
            name : “task2”
            ref : “mypack.workflow2”
            参数：
                date : “ {{ task1.stdout }}“ #这里我们将结果从”task1“作为”日期“动作参数传递给
动作”workflow2“
```

`workflow2.meta.yaml`

```
---
名称： “workflow2”
的描述： “......”
runner_type : ‘行动链’
入口点： ‘workflow2.yaml’
启用： 真正的
参数：
 日期：
    类型： ‘串’
    的描述： “日期该节目被印刷到标准输出“ 必需：是
```

`workflow2.yaml`

```
---
    链：
        -
            名称： “
            task1 ” ref : “core.local”
            参数：
                cmd : “echo {{ date }}” #在这里我们回传作为操作参数传递给工作流的变量“date”
```

上面的示例适用于您有两个相关工作流而另一个调用另一个的情况。

如果您有两个独立的工作流程，并且您希望在它们之间传递数据或在另一个工作流程中使用数据，则最常用的方法是使用内置的键值 数据存储。

在第一个工作流程中，您将数据存储在数据存储中，在第二个工作流程中，您可以从数据存储中检索这些数据。这种方法可以在两个工作流程之间建立更紧密的耦合，并且使得它们更少重复使用，更难以彼此独立运行。在可能的情况下，我们鼓励您设计工作流程，以便您可以使用操作参数传递数据。

使用操作参数意味着第二个工作流程仍然可以重复使用并独立于第一个工作流程运行 - 您只需将所需参数传递给它即可。

暂停和恢复动作链执行

可以通过运行暂停ActionChain的执行。执行必须处于运行状态，以便暂停成功。执行最初将进入一个状态，那么将进入一个状态时，没有更多的任务处于活动状态，，，或。当动作链暂停时，可以通过运行恢
复。 `st2 execution pause <execution-id>` `pausing` `paused` `running` `pausing` `canceling` `st2 execution resume <execu`

发布的变量在暂停时保存在执行上下文中，并在恢复时恢复。

> **注意**
>
> 在此版本中，发布的变量在执行上下文中未加密存储。

The `pause` and `resume` operation will cascade down to subworkflows, whether it's another StackStorm action that is a Mistral workflow or ActionChain. If the `pause` operation is performed from a subworkflow or subchain, then the `pause` will cascade up to the parent workflow or parent chain. However, if the `resume` operation is performed from a subworkflow or subchain, the `resume` will not cascade up to the parent workflow or parent chain. This allows users to resume and troubleshoot branches individually.

Gotchas

Using YAML and Jinja implies some constraints on how to name and reference variables:

- Variable names can use letters, underscores, and numbers. No dashes! This applies to all variables: global vars, input parameters, DataStore keys, and published variables.
- The same naming rules apply to task names: `this-task-name-is-wrong`! Use `task_names_with_underscores`.
- Always quote variable references `"{{ my_variable.or.expression }}"` (remember that `{ }` is a YAML dictionary). The types are respected inside the Jinja template but converted to strings outside: `"{{ 1 + 2 }} + 3"` resolves to `"3 + 3"`.

Error Reporting

ActionChain errors are classified as:

- Errors reported by a specific task in the chain. In this case the error is reported as per behavior of the particular action in the task.

  Sample output:

```
{
     "result": {
         "tasks": [
             {
                 "created_at": "2015-02-27T19:29:02.057885+00:00",
                 "execution_id": "54f0c57e0640fd177f278052",
                 "id": "c1",
                 "name": "c1",
                 "result": {
                     "failed": true,
                     "return_code": 127,
                     "stderr": "bash: borg: command not found\n",
                     "stdout": "",
                     "succeeded": false
                 },
                 "state": "failed",
                 "updated_at": "2015-02-27T19:29:03.149547+00:00",
                 "workflow": null
             }
         ]
     }
 }
```


- Errors experienced by the ActionChain runtime while determining the flow. In this case the error is reported as the error property of the ActionChain result.

  Sample output:


```
{
    "result": {
        "error": "Failed to run task \"c2\". Parameter rendering failed: 's1' is
undefined",
        "traceback": "Traceback (most recent call last):...",
        "tasks": [
            {
                "created_at": "2015-02-27T19:19:34.536558+00:00",
                "execution_id": "54f0c3460640fd15a843957d",
                "id": "c1",
                "name": "c1",
                "result": {
                    "failed": false,
                    "return_code": 0,
                    “stderr” : “” ,
                    “stdout” : “Fri Feb 27 27:19:34 UTC 2015 \ n” ,
                    “succeeded” : true
                },
                “state” : “succeeded” ,
                “updated_at” : “2015-02 -27T19 : 19 : 35.591297 + 00 : 00“ ,
                ”workflow“ : null
            }
        ]
    }
}
```