

# DAC 编排器优化预研说明文档

2021/5/28

## DAC 编排器优化预研说明文档

### 预研目标

#### 核心术语

Resource 资源

Instance 实例

### 相关代码

orchestration-service 编排器服务

REST API 脚手架

MQTT 服务

配置

默认主题

模型

@iota/device-model

@iota/things-model

@iota/rules-model

中间件

MQTT `cm/notify` 消息发布

dc 模型变更 hook

定义 `notify` 属性的模型

@iota/orchestrator 中间件

REST API

编排资源

查询编排资源

查询资源运行所在的实例

前端 重编排 功能

Web UI

接口调用 (资源编排、实例查询)

## Orchestrator.js (编排器核心)

消息订阅与事件监听

MQTT `cm/notify` 消息订阅

资源 `'create'` 和 `'delete'` 事件监听

ZooKeeper 客户端

`this.znodeRoot`

`connect()`: ZK 客户端连接

ZK 客户端操作

Kafka 客户端

`onHeartBeat()`: Kafka 消息处理

Who is the Kafka Producer that produces `'HeartBeat'` topic?

心跳

`this.heartBeatStatus` (心跳已接收 标记) 管理

`startHeartBeat()`

`checkHeart()`

`stopHeartBeat()`

Orchestrator 对象属性

`this.opts` (`initInstances` | `maxInstances` | `resourcePerInstance` 等)

`this.orchestrating`

`this.orchestrateSuccess`

@TODO: `orchestrate()`

`opts = ctx.request.body` 请求参数

编排对象  
orchestration (临时变量)  
**this.lastOrchestration**  
fetchOrchestration()  
makeOrchestration()  
    算法: instances  
    算法: resourcePerInstance  
Instance 实例管理  
    createInstance()  
    deleteInstance()  
    checkInstance()  
Resource 资源管理  
    createResource()  
        choose a instance  
        create a instance  
    deleteResource()  
    monitor(): 监控实例状态  
代码问题  
COMING SOON...

---

## 预研目标

新增 Things (结构物) 时, 实现 DAC 弹性扩容和动态编排机制。

- 弹性扩容: 根据 k8s pod 的 CPU 和 memory 指标, 实现 pod 的弹性扩容。
- 动态编排: 弹性扩容后, 编排器 重编排。

## 核心术语

### Resource 资源

```
'Resource': {  
  'id': {  
    tmp: {  
      status: // monitor signal for Orchestrator.  
    },  
    _reload: // data: times, monitor signal for Instance.  
    resources: // data: reources ids  
  }  
}
```

### Instance 实例

Pod 实例: 目前编排对象为 Thing 和 Rule 两个资源。

- Thing 的 pod 模板: src/templates/iota-dac/pod-template.yml
- Rule 的 pod 模板: src/templates/iota-rules-engine/pod-template.yml

---

## 相关代码

编排器 REST 服务 SVN: <http://10.8.30.22/lota/trunk/code/web/iota-service/orchestrator>

编排器中间件 SVN: <http://10.8.30.22/lota/trunk/code/web/iota-pkg/iota-orchestrator>

## orchestration-service 编排器服务

### REST API 脚手架

服务框架: @iota/api-server-scaffold

### MQTT 服务

#### 配置

iota-service\orchestrator\config.js

```
const mqtt = { broker: args.mqttBrokers || process.env.IOTA_MQTT_BROKERS ||  
'mqtt://localhost:1883', reconnectPeriod: 10000 };
```

#### 默认主题

```
static DEFAULT_CM_NOTIFY_TOPIC = 'cm/notify';
```

默认主题定义涉及模块:

- @iota/dc
- @iota/orchestrator
- @iota/rules-engine

### 模型

@iota/device-model

```
export function models(dc) {  
  interface_type(dc);  
  property_type(dc);  
  protocol_meta(dc);  
  interface_meta(dc);  
  interface_property(dc);  
  device_meta(dc);  
  device_property(dc);  
  device_purpose(dc);  
  formula_meta(dc);  
  formula_property(dc);  
  capability_category(dc);  
  capability_meta(dc);  
  capability_property(dc);  
  device_meta_interface(dc);  
  capability_meta_interface(dc);  
}
```

## @iota/things-model

```
export function models(dc) {
  thing(dc);
  dimension(dc);
  layout(dc);
  device_instance(dc);
  layout_node(dc);
  device_interface(dc);
  device_capability(dc);
  layout_link(dc);
  dimension_capability(dc);
  capability_formula(dc);
  scheme(dc);
}
```

## @iota/rules-model

一个 `Rule` 数据表模型定义。

```
export default function (dc) {
  let Rule = dc.orm.define('Rule', {
    id: {
      type: dc.ORM.STRING,
      primaryKey: true,
      defaultValue: dc.ORM.UUIDV4,
      allowNull: false
    },
    name: {
      type: dc.ORM.STRING,
      allowNull: false
    },
    ...
  }, {
    tableName: 'Rule'
  });
  Rule.notify = {
    c: true,
    u: true,
    d: true,
  };
  dc.models.Rule = Rule;
  return Rule;
};
```

## 中间件

一个 `@iota/orchestrator` 中间件，指定了两个资源：`Thing` 和 `Rule`。

```
const orchestratorEntry = require('@iota/orchestrator').entry;
const orchestrator = {
  entry: orchestratorEntry,
  opts: {
    // template path
  }
};
```

```

templatesPath: path.join(__dirname, 'templates'),
// notify
mqtt: mqtt,
// config
zookeeper: zookeeper,
// kafka
kafka: brokers,
// resources to be orchestrate
resources: [{
  resource: "Thing",
  instanceName: 'iota-dac',
  heart: true,
  initInstances: 10
}, {
  resource: "Rule",
  instanceName: 'iota-rules-engine',
  initInstances: 10
}],
runInPod: runInPod,
kubernetes: {
  url: kubernetes,
  insecureSkipTlsverify: true,
  version: 'v1',
  promises: true,
  namespace: 'iota',
  auth: {
    bearer: args.authToken || process.env.IOTA_AUTH_TOKEN ||
'eyJ...4Dg'
  }
}
};

config.mws.push(orchestrator);

```

## MQTT cm/notify 消息发布

### dc 模型变更 hook

@iota/dc -> 数据表记录 增/删/改 钩子回调 notifyAfterCommit() 创建并发送 mqtt 消息。

```

//add hook for create update destroy
dc.orm.addHook('afterCreate', function (ins, opts) {
  notifyAfterCommit(ins, opts, 'c');
});
dc.orm.addHook('afterUpdate', function (ins, opts) {
  notifyAfterCommit(ins, opts, 'u');
});
dc.orm.addHook('afterDestroy', function (ins, opts) {
  notifyAfterCommit(ins, opts, 'd');
});

//add hook for bulk c u d
dc.orm.addHook('afterBulkCreate', function (inss, opts) {

```

```

    notifyAfterCommit(inss, opts, 'bc');
  });
  dc.orm.addHook('afterBulkUpdate', function (opts) {
    notifyAfterCommit(null, opts, 'bu');
  });
  dc.orm.addHook('afterBulkDestroy', function (opts) {
    notifyAfterCommit(null, opts, 'bd');
  });

```

mqtt 消息:

```

// type: c u d bc bu bd

// hook: bu bd
msg = JSON.stringify({ o: type, t: table, where: opts.where });

// hook: bc
msg = JSON.stringify({ o: type, t: table, keys: ins.primarykeys });

// hook: c u d
msg = JSON.stringify({ o: type, t: table, k: ins.primarykey, instance: ins });

// default
msg = JSON.stringify({ o: type, t: table });

```

注入 mqtt 对象:

```

let mqtt = new Mqtt(app, options.mqtt); // MQTT client

dc.notify = mqtt;

// mqtt 对象方法
notifyCm(msg) {
  this.client.publish(this.cmNotifyTopic, msg, { qos: 1 }); //
  this.cmNotifyTopic = DEFAULT_CM_NOTIFY_TOPIC = 'cm/notify';
}

```

## 定义 notify 属性的模型

仅对定义了 notify 属性的模型, `notifyAfterCommit()` 会创建并发送 mqtt 消息。

定义了 notify 属性的模型 (model) 有: `Thing`、`Rule`、`ProtocolMeta`、`DeviceMeta`。

```

Thing.notify = {
  c: true,
  d: true,
  u: true,
};

Rule.notify = {
  c: true,
  u: true,
  d: true,
};

```

```
ProtocolMeta.notify = {  
  c: true,  
  u: true,  
  d: true,  
};  
  
DeviceMeta.notify = {  
  c: true,  
  u: true,  
  d: true,  
};
```

## @iota/orchestrator 中间件

---

@iota/orchestrator 中间件以 REST API 形式提供相关编排服务。

### REST API

#### 编排资源

```
POST /v1/api/orchestrations/:resource/orchestrate
```

#### 查询编排资源

```
GET /v1/api/orchestrations/:resource
```

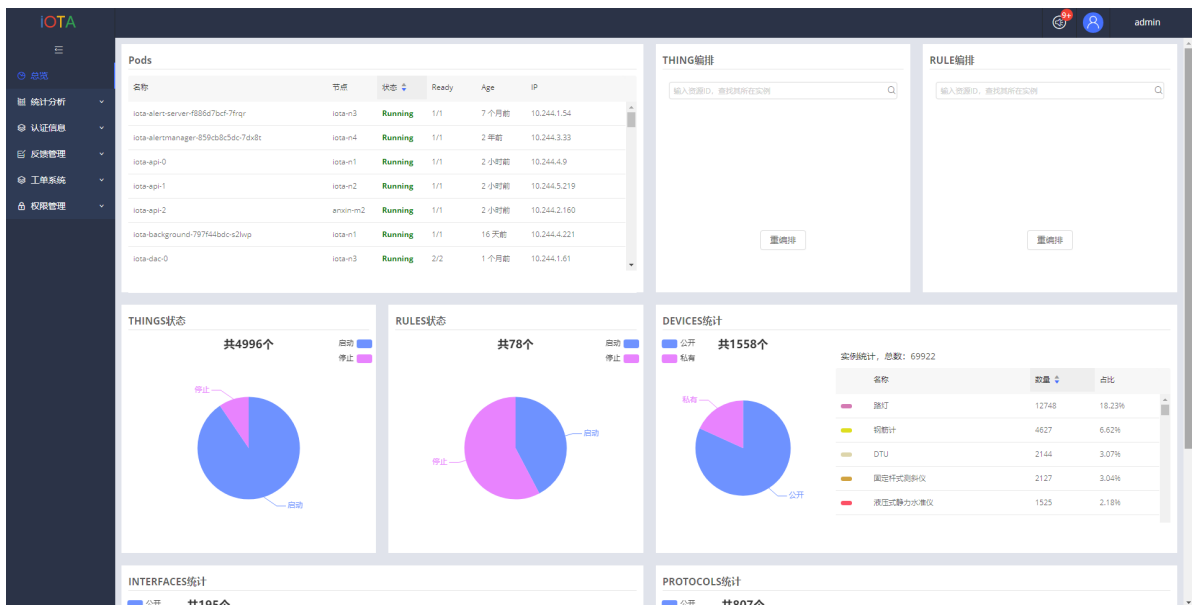
#### 查询资源运行所在的实例

```
GET /v1/api/orchestrations/:resource/:id/instances
```

## 前端 重编排 功能

### Web UI

iOTA 后台管理界面, 重编排 对象: Thing 和 Rule。



## 接口调用 (资源编排、实例查询)

SVN: <http://10.8.30.22/lota/trunk/code/web/iota-web-client-scaffold/src/iota/fe-background-dashboard/src/dashboard/Orchestrator.jsx>

- 资源编排接口调用:

```
orchestrator = () => {

  fetch(`${this.props.apiServer}/orchestrations/${this.props.resource}/orchestrations`, {
    method: 'POST',
    headers: { "Content-Type": "application/json" },
    credentials: 'include'
  }).then(async res => {
    let result = await res.json();
    let status = res.status;
    return status !== 200 ? Promise.reject({ result }) : Promise.resolve({
      result });
  }).then(result => {
    message.success('资源编排成功。');
  }).catch(err => {
    message.error('资源编排失败。');
  });
}
```

- 资源运行所在实例获取:

```
findResource = () => {

  fetch(`${this.props.apiServer}/orchestrations/${this.props.resource}/${this.state.resourceId}/instances`, {
    method: 'GET',
    headers: { "Content-Type": "application/json" },
    credentials: 'include'
  }).then(async res => {
    let result = await res.json();
    let status = res.status;
```



```
        return status !== 200 ? Promise.reject({ result }) : Promise.resolve({
result });
    }).then(result => {
        this.setState({ runIn: result.result.runIn })
    }).catch(err => {
        this.setState({ runIn: 'N/A' })
    })
}
```

## Orchestrator.js (编排器核心)

源码: src\lib\Orchestrator.js

### 消息订阅与事件监听

#### MQTT `cm/notify` 消息订阅

```
// that = this : ResourceNotify (extends EventEmitter) 对象

that.client.subscribe(that.cmNotifyTopic); // that.cmNotifyTopic =
DEFAULT_CM_NOTIFY_TOPIC = 'cm/notify';

this.client.on('message', async function (topic, message) {
    var notify = JSON.parse(message.toString());

    switch (notify.o) {
        case 'c':
            that.emit('create', { id: notify.k.id })
            break;
        case 'bd':
            that.emit('delete', { id: notify.where.id })
            break;
    }
}
```

#### 资源 'create' 和 'delete' 事件监听

```
// this : Orchestrator (extends EventEmitter) 对象

this.resourceNotify.on('create', this.createResource);
this.resourceNotify.on('delete', this.deleteResource);
```

### ZooKeeper 客户端

`@iota/zookeeper`: 基于 `node-zookeeper-client` 封装的 ZooKeeper 客户端模块。

```
import Zookeeper, { State, Event } from '@iota/zookeeper';
```

## this.znodeRoot

```
// this.znodeRoot = '/iota/orchestration'  
// opts.resource : ['Thing', 'Rule']  
this.znodeRoot = `${this.znodeRoot}/${opts.resource}`;
```

## connect(): ZK 客户端连接

```
connect = async () => {  
  if (!this.zookeeper) {  
    this.zookeeper = new Zookeeper(this.opts.zookeeper || 'localhost:2181');  
    await this.zookeeper.connect();  
  }  
  // ZK 客户端不是"已连接"状态, 则重连  
  else if (this.zookeeper.getState() !== State.SYNC_CONNECTED) {  
    this.logger.error('reconnect');  
  
    await this.zookeeper.close()  
  
    this.zookeeper = new Zookeeper(this.opts.zookeeper || 'localhost:2181');  
    await this.zookeeper.connect();  
  }  
}
```

## ZK 客户端操作

主要进行 zk 存储管理。

## Kafka 客户端

@iota-fork/kafka-node -> ConsumerGroup: Kafka 消费者组。

```
if (opts.heart) {  
  this.heartBeatStatus = {};  
  this.kafkaConsumer = new Kafka.ConsumerGroup({  
    kafkaHost: opts.kafka || 'localhost:9092',  
    groupId: `iota-orchestrator-${opts.resource}`, // opts.resource =  
    'Thing' or 'Rule'  
    fromOffset: 'latest'  
  }, ['HeartBeat']); // topics = ['HeartBeat']  
  this.kafkaConsumer.on('error', this.onError);  
  this.kafkaConsumer.on('message', this.onHeartBeat);  
}
```

## onHeartBeat(): Kafka 消息处理

- kafka message: heartBeat
- 函数功能: 设置 this.heartBeatStatus 对象中指定实例的心跳已接收 标记为 true

```

onHeartBeat = (heartBeat) => {
  try {
    this.logger.info(`receive heartbeat: ${JSON.stringify(heartBeat)}`)
    let tmp = JSON.parse(heartBeat.value);
    if (tmp.resource === this.opts.resource) {
      this.heartBeatStatus[tmp.instanceId] = true;
    }
  } catch (err) {
    this.logger.error(err)
  }
}

```

## Who is the Kafka Producer that produces 'HeartBeat' topic ?

DAC 通过 Kafka 发送服务心跳 (SendHeartBeat): `func (s *SyncService) SendHeartBeat() {}`

## 心跳

### this.heartBeatStatus (心跳已接收 标记) 管理

- Orchestrator -> constructor(): 初始化 `resetHeartBeatStatus`
- resetHeartBeatStatus(): 重置 `resetHeartBeatStatus`

```
this.heartBeatStatus = {};
```

- addHeartBeatCheck(instanceId): 指定实例的心跳已接收 标记为 `false`
- checkHeart(): 指定实例的心跳已接收 标记为 `false`

```
this.heartBeatStatus[instanceId] = false;
```

- onHeartBeat(): 接收 kafka 消息, 将指定实例的心跳已接收 标记为 `true`

```

this.kafkaConsumer.on('message', this.onHeartBeat);

onHeartBeat = (heartBeat) => {
  this.heartBeatStatus[tmp.instanceId] = true; // tmp: heartBeat message JSON
  object
}

```

### startHeartBeat()

启动心跳, caller:

- orchestrate(): 重编排 触发
- createResource(): 新增资源后触发

```
this.heartBeatTimer = setTimeout(this.checkHeart, 90 * 1000)
```

## checkHeart()

心跳程序。

```
checkHeart = async () => {
  if (this.opts.heart) {
    for (let id in this.lastOrchestration) {
      if (!this.heartBeatStatus[id]) {
        this.logger.info(`Not receive heartBeat rightly, delete instance
[${id}]`);
        await this.deleteInstance(id);
        await this.createInstance(id);
      }
      this.heartBeatStatus[id] = false;
    }
    this.heartBeatTimer = setTimeout(this.checkHeart, 90 * 1000);
  }
}
```

## stopHeartBeat()

停止 `setTimeout` 计数器。caller:

- `orchestrate()`: 重编排触发
- `createResource()`: 新增资源后触发

## Orchestrator 对象属性

**this.opts ( `initInstances` | `maxInstances` | `resourcePerInstance` 等)**

- `initInstances` 默认值: 10
- `maxInstances` 有效值范围: [3, 100]
- `resourcePerInstance` 有效值范围: (0, 5000]

```
this.opts.initInstances = this.opts.initInstances || 10;
this.opts.maxInstances = (3 <= opts.maxInstances && opts.maxInstances < 100) ?
opts.maxInstances : 100;
this.opts.resourcePerInstance = 0 < opts.resourcePerInstance &&
opts.resourcePerInstance < 5000 ?
  opts.resourcePerInstance :
  5000;
```

## this.orchestrating

编排是否正在进行中的标志。由 `orchestrate()` 对该标志进行管理。

## this.orchestrateSuccess

编排成功与否的标志。由 `orchestrate()` 对该标志进行管理。

## @TODO: orchestrate()

orchestrate(): 资源编排 `router.post('/orchestrations/:resource/orchestrate')` 的 Controller

```
orchestrate = async (opts) => { // const opts = ctx.request.body;
}
```

### opts = ctx.request.body 请求参数

- `maxInstances` 请求参数的有效值: [3, 100)
- `resourcePerInstance` 请求参数的有效值: (0, 5000)

## 编排对象

### orchestration (临时变量)

```
let orchestration = await this.makeOrchestration(opts);
```

- `makeOrchestration()` -> `orchestration[]` 数据结构:

```
orchestration[i] = {
  data: {
    used: 0,
    count: resourcePerInstance
  },
  children: {
    tmp: {},
    _reload: { data: { times: 0 } },
    resources: { data: { ids: [] } }
  }
};
```

### this.lastOrchestration

```
orchestrate = async (opts) => {
  this.lastOrchestration = await this.fetchOrchestration(); // 取旧值
  ...
  this.lastOrchestration = orchestration; // 更新
}

createResource = async (resource) => {
  this.lastOrchestration[newInstanceId] = newInstance;
}
```

- `fetchOrchestration()` -> `orchestration[]` 数据结构:

```
orchestration[id] = {
  data: {},
  children: {
    _reload: { data: { times: 0 } },
    resources: { data: { ids: [] } }
  }
}
```

## fetchOrchestration()

从 ZooKeeper 中读数据，返回编排器对象。

```
/**
 * Get lastOrchestration orchestration.
 */
fetchOrchestration = async () => {
  let instances = await this.zookeeper.getChildren(znode).catch(err => {
    this.logger.error('fetch zookeeper children failed', err);
    return [];
  });

  for (let i = 0; i < instances.length; i++) {
    // zk = this.zookeeper
    orchestration[id].data = await zk.getData(`${znode}/${id}`)
    orchestration[id].children.resources.data = await
    zk.getData(`${znode}/${id}/resources`)
    orchestration[id].children._reload.data = await
    zk.getData(`${znode}/${id}/_reload`)
  }

  return orchestration;
}
```

## makeOrchestration()

### 算法: instances

根据 Thing 或 Rule 的 model.count() 计算需要的实例数 (instances):

```
// count = await model.count();
// resourcePerInstance = opts.resourcePerInstance;
let instances = Math.floor(count / resourcePerInstance) + 1;
```

- maxInstances 实例数限制: 服务允许的最大实例数: 100 个

```
if (instances > opts.maxInstances) {
  throw new Error(`require instances more than the max:
  ${instances}/${opts.maxInstances}`);
}
```

## 算法: resourcePerInstance

resourcePerInstance 重新设置:

- 如果 `instances` (模型需要的实例数) 少于 `initInstances` (指定的实例数), 则使用指定的实例数 `initInstances` 重新计算 `resourcePerInstance` (每个实例的资源数)
- `instances` 实例数区间: `[opts.initInstances, opts.maxInstances]`, 其中 `opts.maxInstances < 100`

```
if (instances < opts.initInstances) {
  instances = opts.initInstances;
  resourcePerInstance = Math.floor(count / instances) + 1; // count = await
model.count();
}
```

## Instance 实例管理

### createInstance()

从 `pod-template.yml` 读取, 更新信息:

- `metadata.name`
- `metadata.labels.instanceId`
- `spec.containers[0].env[0].value`

```
createInstance = async (id) => {
  let name = `${this.opts.instanceName}-${id}`;
  let podInstance = Object.assign({}, this.templates.pod);
  podInstance.metadata.name = name;
  podInstance.metadata.labels.instanceId = `${id}`;
  podInstance.spec.containers[0].env[0].value = `${id}`;

  await Orchestrator.createPromise(async () => {
    await this.kubeApi.ns.po.post({ body: podInstance });
  });
}
```

### deleteInstance()

```
deleteInstance = async (id) => {
  let name = `${this.opts.instanceName}-${id}`;
  await this.kubeApi.ns.po.delete({ name: name, gracePeriodSeconds: 0 });
}
```

### checkInstance()

对于异常状态的实例, 删除后重新创建。

```
checkInstance = async (id) => {
  let name = `${this.opts.instanceName}-${id}`;
  try {
    let dacPod = await this.kubeApi.ns.po.get({ name: name });
  }
}
```

```

    if (dacPod && (dacPod.status.phase !== 'Running' && dacPod.status.phase
    !== 'Pending')) {
        await this.deleteInstance(id);
        await this.createInstance(id);
        // cannot delete , cause kubernetes cannot delete it immediately
    }
} catch (err) {
    if (err.code === 404) {
        await this.createInstance(id);
    } else {
        this.logger.error(`check instance failed, name: ${name}`, err, err ?
err.message : '');
        throw err;
    }
}
}
}

```

## Resource 资源管理

### createResource()

createResource(): 资源新增的处理器。

- 选择一个实例以添加资源
- 创建一个实例以添加资源

### choose a instance

选择一个实例，添加资源，通知 DAC 。

```

/**
 * Handle resource added.
 * choose a instance or create one
 * choose: add resource to instances.reources and notice instance
 * @param resource
 */
createResource = async (resource) => {
    //find one
    let instanceIds = Object.keys(this.lastOrchestration);
    instanceIds.reverse();

    for (let i = 0; i < instanceIds.length; i++) {
        let id = instanceIds[i];
        let instance = this.lastOrchestration[id];
        count = instance.data.count;
        if (instance.data.used < instance.data.count) {

            let transaction = this.zookeeper.transaction();
            transaction.setData(`${this.znodeRoot}/${id}`,
                new Buffer(JSON.stringify(instance.data)))
                .setData(`${this.znodeRoot}/${id}/_reload`,
                    new Buffer(JSON.stringify(instance.children._reload.data)))
                .setData(`${this.znodeRoot}/${id}/resources`,
                    new
Buffer(JSON.stringify(instance.children.resources.data)));
            await transaction.commit();

```



```

        break;
    }
}
}

```

## create a instance

创建一个实例，添加资源，更新 `this.lastOrchestration`。

```

createResource = async (resource) => {
  let newInstanceId = 0;
  while (true) {
    if (this.lastOrchestration[newInstanceId]) {
      newInstanceId++;
    } else {
      break;
    }
  }
  let newInstance = {
    data: {
      used: 1,
      count: count
    },
    children: {
      tmp: {},
      _reload: { data: { times: 0 } },
      resources: {
        data: { ids: [resource.id + ''] }
      }
    }
  }
  let transaction = this.zookeeper.transaction();
  transaction.create(`${this.znodeRoot}/${newInstanceId}`, new
Buffer(JSON.stringify(newInstance.data)))
    .create(`${this.znodeRoot}/${newInstanceId}/tmp`)
    .create(`${this.znodeRoot}/${newInstanceId}/_reload`, new
Buffer(JSON.stringify(newInstance.children._reload.data)))
    .create(`${this.znodeRoot}/${newInstanceId}/resources`, new
Buffer(JSON.stringify(newInstance.children.resources.data)));
  await transaction.commit();
  this.stopHeartBeat();
  await this.checkInstance(newInstanceId);
  this.addHeartBeatCheck(newInstanceId);
  this.startHeartBeat();
  this.lastOrchestration[newInstanceId] = newInstance;
}

```

## deleteResource()

`deleteResource()`: 资源删除的处理器，查找实例 - 删除资源 - 通知 DAC。

```

deleteResource = async (resource) => {
  ...
}

```

## monitor(): 监控实例状态

监控周期 60 s, Succeeded | Failed | Unknown 异常状态下重新创建实例, 以保证实例的正常运行。

```
monitor = async () => {
  let ids = Object.keys(this.lastOrchestration);
  for (let i = 0; i < ids.length; i++) {
    await this.checkInstance(ids[i]).catch(err => {});
  }
  setTimeout(this.monitor, 60 * 1000);
}
```

## 代码问题

---

createResource() 和 deleteResource() 两个方法中均有如下代码:

```
instance.children._reload.data.times + 1;
```

这个 times + 1 之后, 但没赋值, times 不变, 无意义。

## COMING SOON...

---

未完, 待续...