利用 AWS 无服务器架构(serverless)

让游戏实现全球同服

2016 年 6 月

近年来，随着全球社交化和移动互联网技术的发展，全球同服的游戏架构已经形成一种趋势。通过 AWS 全球分布的云基础设施可以非常便利的完成这种游戏后台架构的部署：

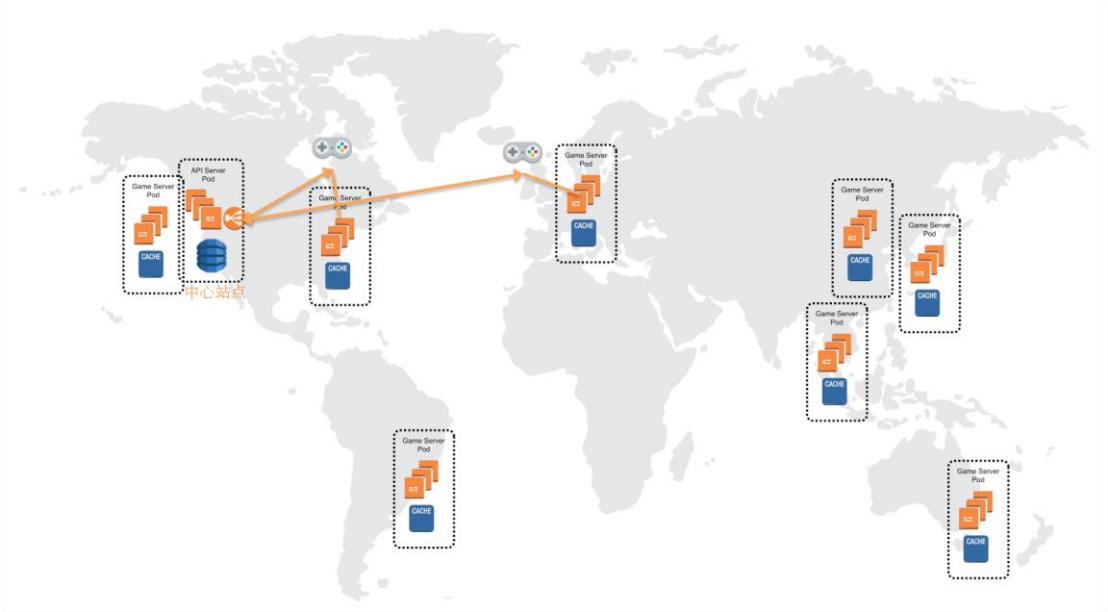

图 1：全球同服的游戏架构

a) 全球所有玩家的持久化信息（包括用户基本信息，等级，装备，进度等状态信息）都保存在中心站点。玩家统一通过 HTTP（s）登录到中心站点并获取状态信息。
b) 对战初始，由中心站点对玩家进行重定向到对应的 Game Server。在对战过程中使用 TCP 长连接从而保证更好的游戏体验。
c) 对战结束后，客户端与 Game Server 中断 TCP 连接，对战结果数据回写中心站点并保存最终的状态信息。

基于上述的架构，游戏完全构建在统一的“大世界”中（唯一中心站点），并且由分布在全球 Game Server 来保证游戏的低延迟。但接下来的问题随之而来——由于 Game Server 分布在全球不同的地区，如何能做到资源的快速扩展和按需伸缩？本文下面将以 Serverless 的方式阐述实现这一需求。

首先，AWS 平台提供了非常完整的 API 接口，开发者可以选择各种语言的 SDK 完成对资源的调度，这里我们可以将代码运行在 Lambda 中。如下示例，我们的中心站点即 Lambda 部署的站点选择的是弗吉尼亚（美东）地区，通过 Nodejs SDK 跨地区（东京）启动 EC2 服务器：

```
var AWS = require('aws-sdk');

exports.handler = function(event, context) {
    console.log("Received data as:",event);

    var ec2 = new AWS.EC2({region: 'ap-northeast-1'});

    var params = {
      ImageId: 'ami-29160d47',
      InstanceType: 't2.micro',
      KeyName:'Tech-labs',
      SecurityGroupIds:['sg-d0aa1bb4'],
      IamInstanceProfile: {
       Name: 'EC2-Admin'
      },
      MinCount: 1, MaxCount: 1
    };

    // Create the instance
    ec2.runInstances(params, function(err, data) {
      if (err) {
          console.log("Could not create instance", err);
          context.fail(err); }

      var instanceId = data.Instances[0].InstanceId;
      console.log("Created instance", instanceId);
     //ToDo persistent instance id and set status as starting
      context.succeed(instanceId);

    });
};
```

由于启动 EC2 的过程是一个异步过程，所以我们需要记录相关的服务器启动信息（InstanceId），并定义另一接口接受 Game Server 在服务就绪后返回的回执信息。如以下示例：

```
var AWS = require('aws-sdk');

exports.handler = function(event, context) {
    console.log("Received data as:",event);

    var instanceId=event.instanceId;
    var region=event.region;
    var publicIp=event.publicIp;
    var version=event.version;
    ...

    //TODO check instanceId and update instance status to online

};
```

同时，这种回执接口的 API（包括其他 API）都可以考虑使用 Amazon API Gateway 服务进行部署。API Gateway 可以帮助我们将现有函数快速的发布 Restful 的 API 接口，并同时利用 CloudFront 的边缘节点进行部署，以保证访问端能获得更低的延迟。按照上例的回执 Lambda 函数可以构造 API Gateway 的配置如下：

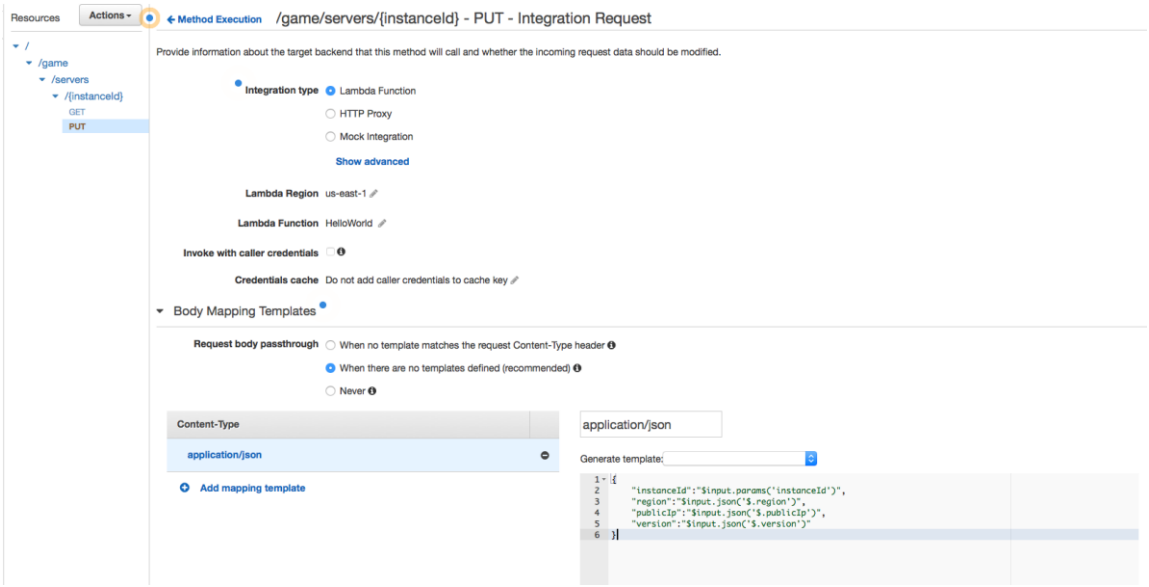

图 2：API Gateway 映射 Lambda 函数

请求示例：

```
/game/servers/i-dd861842

{
      “region”:”ap-northeast-1”,
      “publicIp”:” 52.193.34.102”,
      “version”:”110”
}
```

接下来，为了确保 Game Server 的状态是正常并使得玩家能被路由到正确的，可以构造另一类似心跳的 Lambda 函数用以接受 Game Server 的状态信息，心跳频率可根据需求进行调整。当然，如果频率不需要很高的情况下（>=1 分钟），也可以利用 CloudWatch 来发起报警，并同时发起 SNS 通知 Lambda 函数以更新 Game Server 的状态。

最后，在 Game Server 具备了自动按需扩展（Scale out）的能力后，我们就需要考虑如何解决 Game Server 的缩减（Scale in）了。这里，我们采用 CloudWatch->SNS->Lambda(cross region)的方式来实现用 Game Server 的缩减（Scale in），具体流程说明如下：

1. Game Server 自定义指标（custom metrics）发送当前服务器的在线人数到 CloudWatch 中。

```
#!/bin/bash
#get instance-id from local meta-data
id=$(curl -s http://169.254.169.254/latest/meta-data/instance-id)
#get current onlinePlayers
players=$(...)
aws cloudwatch put-metric-data --metric-name "OnlinePlayers" --namespace
"GameServer" --dimension InstanceId=$id --value $players
```

2. 设定 CloudWatch 的报警规则，当服务器在线人数为零时，会触发 SNS 通知。

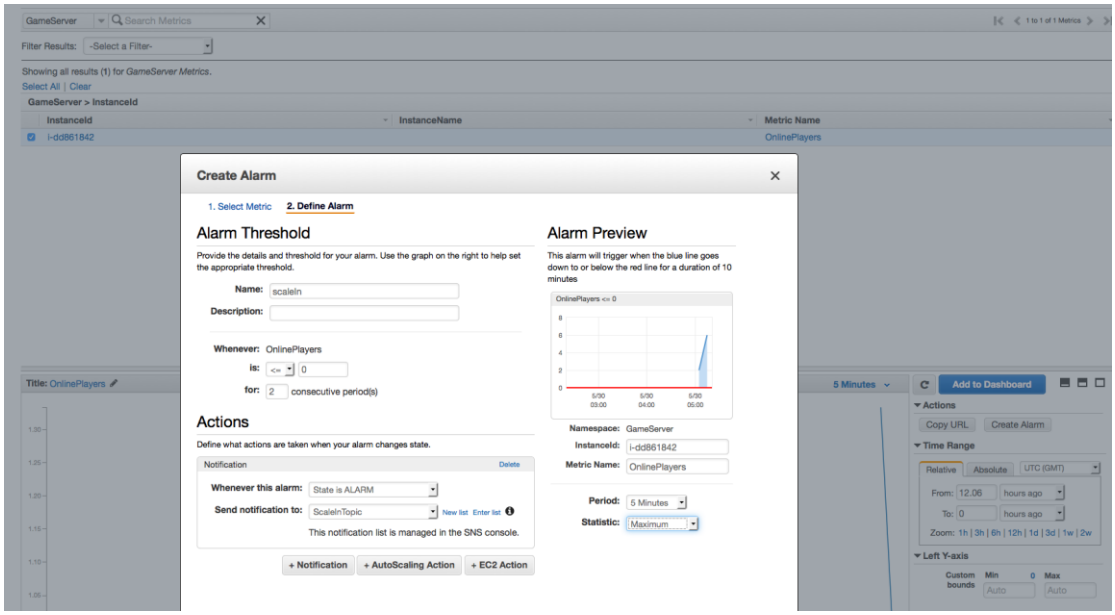

图 3：CloudWatch 自定义指标报警

在实际场景中， 需要通过如下脚本自动建立报警：

```
aws cloudwatch put-metric-alarm --alarm-name $id-players--alarm-description "Alarm
when online players less than 0" --metric-name "OnlinePlayers" --namespace
"GameServer" --dimension "Name=InstanceId,Value=$id" --statistic Maximum --period 300
--threshold 0 --comparison-operator LessThanOrEqualToThreshold  --evaluation-periods 2
--alarm-actions arn:aws:sns:ap-northeast-1:111111111222:ScaleInTopic
```

3. SNS 通知主题的订阅者为中心站点的 Lambda 函数用于把机器终止。

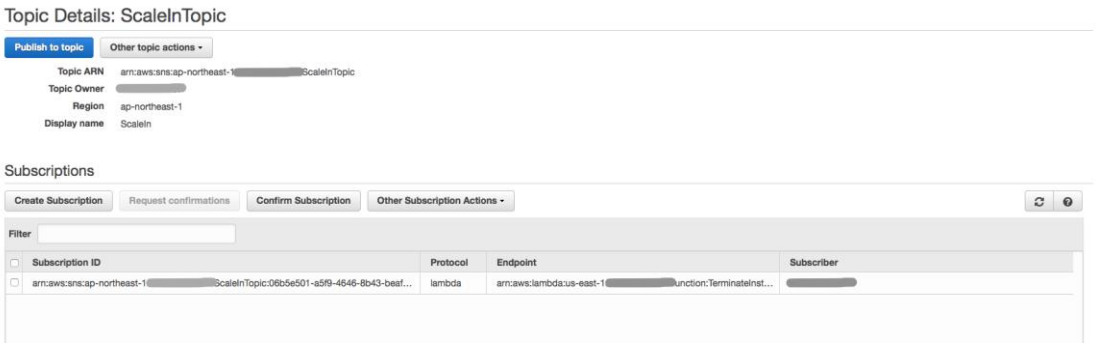

图 4：Lambda 函数订阅 SNS 服务通知

用于终止服务器的 Lambda 函数：

```
var AWS = require('aws-sdk');

exports.handler = function(event, context) {
    console.log("Received data as:",event);
    var message=JSON.parse(event.Records[0].Sns.Message);
    var region=event.Records[0].EventSubscriptionArn.split(":")[3];
    console.log("Need to terminate the server in region:",region);
    var ec2 = new AWS.EC2({region: region});
    console.log("Need to terminate the server:",message);
    var instanceId=message.Trigger.Dimensions[0].value;
    console.log("Need to terminate the server:",instanceId);

    //TODO check if instance can be terminated from DynamoDB and update
instance as terminating

    var params = {InstanceIds:[instanceId]};


    //terminate the instance
    ec2.terminateInstances(params, function(err, data) {
      if (err) {
          console.log("Could not terminate instance", err);
         //TODO rollback terminating the instance
          context.fail(err);

      }
      for(var i in data.TerminatingInstances) {
            var instance = data.TerminatingInstances[i];
            console.log('TERM:\t' + instance.InstanceId);
         //TODO delete terminated the instance
        }
      context.succeed(data.TerminatingInstances);

    });
};
```

通过之上的方法，我们已基本实现了以基于事件触发的 Serverless 架构对全球分布的 Game Server 的调度，具体参见下图架构。

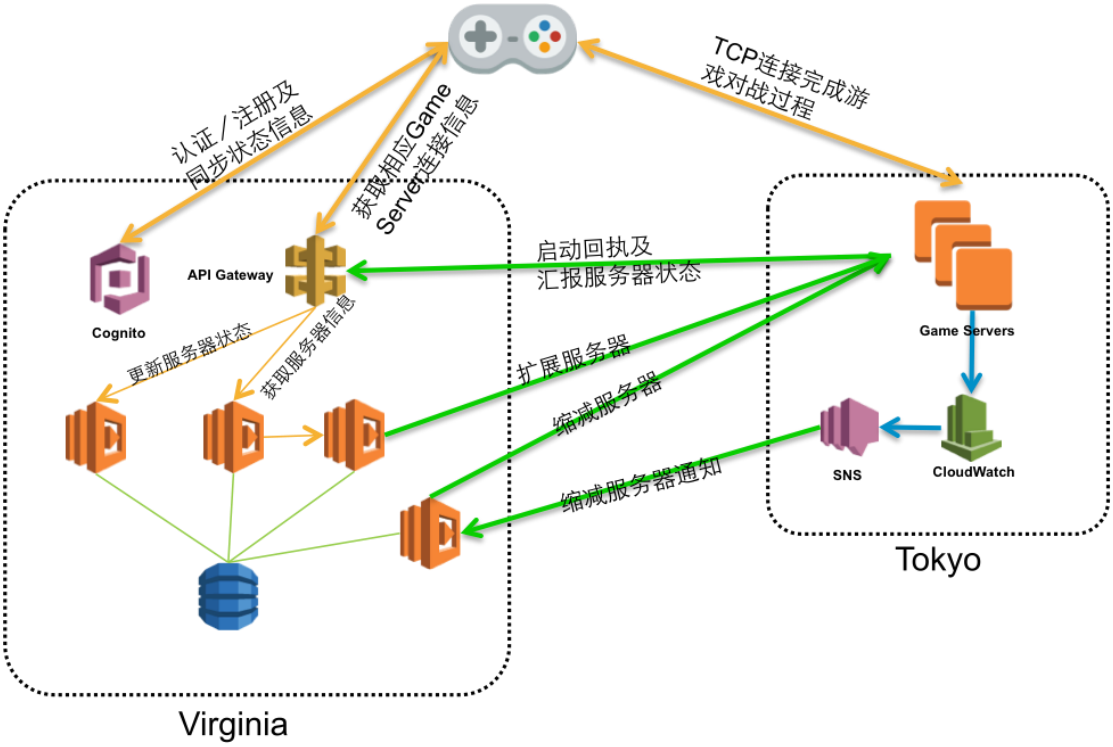

图 5：Serverless 全球同服游戏架构

关于架构里中心站点的 Cognito 和 DynamoDB 的应用方案将在以后的文章里详细介绍，敬请期待。