# 安装pomelo

pomelo是基于Node.js的，目前pomelo已经完全支持Windows、Linux、Mac等多种平台。

### 准备

* 确保你的机器可以上网,因为安装pomelo的过程需要从网上下载其依赖的包。
* 确保你的系统上已经要安装了Node，目前最新的Node提供了已经编译好的二进制安装包，包括Windows，Mac和Linux等平台。想省事的话，直接去[这里](http://nodejs.org/download/)下载对应的安装包，直接安装就好了。Node同时也提供了传统的从源码编译的方式安装，不过比起直接使用二进制的方式要麻烦。
* 确保你的系统中安装有python(2.5 < version < 3.0)以及C++的编译器。Node的源码主要由C++代码和JavaScript代码构成，但是却用[gyp](http://code.google.com/p/gyp/)工具来做源码的项目管理，该工具采用Python语言写成的。对于非windows平台，一般都会预装Python以及C++编译工具；对于Windows系统，请确保你的Windows系统包含源码编译工具。在Windows平台上，Node.js采用gyp来生成Visual Studio Solution文件，最终通过VC++的编译器将其编译为二进制文件。
* 虽然pomelo是用Javascript写成，但是pomelo依赖的库中，有使用了C++语言写的扩展，因此安装pomelo的过程中会使用到C++编译器。 所以，在安装之前请确保你的Windows系统满足以下两个条件：
  + [python](http://python.org/)(2.5<version<3.0)。
  + VC++ 编译器，包含在[Visual Studio 2010](http://msdn.microsoft.com/en-us/vstudio/hh388567)中（VC++ 2010 Express亦可）。对于windows8的用户，需要安装Microsoft Visual Studio C++ 2012。
* 如果你使用的是Mac OS X系统, 则需要安装[Xcode Command Line Tools](https://developer.apple.com/downloads/index.action?q=xcode)或者[Xcode](https://developer.apple.com/xcode/)的完整包以及make工具.

### 安装pomelo

使用npm(node包管理工具)全局安装pomelo:

$ npm install pomelo -g

可以通过如下命令下载源代码的方式安装

$ git clone https://github.com/NetEase/pomelo.git

$ cd pomelo

$ npm install -g

其中-g表示全局安装，关于npm的使用问题，可以参考[npm的文档](https://npmjs.org/doc/)，里面有详细的npm使用的介绍。如果安装过程中没有报错误，说明安装成功。

windows下安装经验：

1. node,vs2010 和 python(2.5<v<3) 都是32位或者都是64位的。

2. 配置 PYTHON=d:\Python27\python.exe(设置成你自己的路径)。注意不是path里面,而是和path同级的，直接在全局或者当前用户下配置。

3. 保证环境变量path里面有 %SystemRoot%\system32;%SystemRoot%;%SystemRoot%\System32\Wbem;

注： 这三个环境变量中貌似只有%SystemRoot%\system32这个环境变量有用，没具体试（没有他会报CreateProcessW找不到的错误)。

4. 如果在命令行界面安装pomelo失败，可以在Visual Studio的命令行界面安装。

下面我们将通过一个[HelloWorld项目](https://github.com/NetEase/pomelo/wiki/pomelo%E7%9A%84HelloWorld)来检验我们的安装是否成功。

# 游戏服务器的app.js配置参考

在使用pomelo框架创建的项目中有两个app.js，它们分别在game-server目录和web-server目录下，分别对应游戏服务器与web服务器，并且它们都是运行各自服务器的入口。以下内容主要是针对game-server中的app.js的配置说明。

## 应用入口设置

app.js是运行pomelo项目的入口，在使用pomelo命令行创建一个新项目时，会根据项目的相关信息生成默认的app.js文件。其主要内容如下：

var pomelo = require('pomelo');

var app = pomelo.createApp();

app.set('name', 'nameofproject');

app.start();

首先根据pomelo创建一个应用；然后会设置应用的名称、最后启动服务，这些配置都是创建一个项目必要条件；最后配置完成后就可以启动项目。

## 设置变量与状态

应用的变量可以通过set和get方法进行存取，例如要存取server对象，具体代码如下：

app.set('server',server);

var server = app.get('server');

应用的功能选项可以通过enable和disable来打开和关闭；另外，用户可以通过enabled和disabled对相应的状态进行判断，如果该状态存在则返回true,反之返回false。例如要打开或者关闭应用的filter并查看其状态是否存在, 具体代码如下：

app.enable('filter');

app.enabled('filter'); //返回true

app.disable('filter');

app.disabled('filter'); //返回true

用户可以通过loadConfig加载配置文件，加载后文件中的参数将直接挂载到application对象上。例如需要加载mysql.json文件，具体代码如下：

{

"development":

{

"host":"127.0.0.1",

"port":"3306",

"database":"pomelo"

}

}

app.loadConfig('mysql.json');

var host = app.mysql.host; //返回 127.0.0.1

## app.configure规则

服务器的配置主要由configure()方法完成，完整的app.configure配置参数如下：

app.configure([env], [serverType], [function]);

前两个参数是可选的， 以下是参数说明：

* env: 运行环境， 可设成development, production或development|production
* serverType: 服务器类型，设置了这个参数只会对当前参数类型服务器做初始化，不设置则对所有服务器执行初始化function
* function: 具体的初始化操作， 内部可以写任何js方法

以下是一些配置实例：

### 实例一

app.configure(function(){

});

这种配置将对所有的模式（development/production）下的所有服务器生效。

### 实例二

app.configure('development', function(){

});

这种配置则只针对development模式下所有服务器生效。

### 实例三

app.configure('development', 'chat', function(){

});

这种配置则针对development模式下的chat服务器生效。

### 初始内容实例

在configure中用户可以根据应用的不同需求在不同的服务器中进行相关配置，例如在全局设置mysql参数：

app.configure('development|production', function(){

app.loadConfig('mysql', app.getBase() + '/config/mysql.json');

});

另外也可以选择在具体的服务器中进行应用的配置，例如：

var initArea = function(){

//area init

};

app.configure('development|production', 'area', function(){

initArea();

});

## 加载component

component是pomelo的组件，它是具有生命周期的应用组件，通常在框架初始化时载入。 pomelo自带的component如下：master、monitor、filter、proxy、handler、remote、server、sync和connection，其主要功能如下：

* master： master组件主要负责启动master服务器。
* monitor： monitor组件主要负责启动各个服务器的monitor服务，该服务负责收集服务器的信息并定期向master进行消息推送，保持master与各个服务器的心跳连接。
* proxy： proxy组件主要负责生成服务器rpc客户端，由于系统中存在多个服务器进程，不同服务器进程之间相互通信需要通过rpc调用（master服务器除外）。
* handler：handler组件主要负责加载前端服务器中的handler目录下的文件。
* filter: filter组件主要负责加载服务器请求的filter服务，包括rpc调用前的filter及rpc调用后的filter。
* remote： remote组件主要负责加载后端服务器的服务并生成服务器rpc服务端。
* server：server组件主要负责启动所有服务器的用户请求处理服务。
* connector: connector组件主要负责启动前端服务器的session服务和接收用户请求。
* sync： sync组件主要负责启动数据同步模块并对外提供数据同步功能。
* connection: connection组件主要负责启动用户连接信息的统计服务。

pomelo会默认根据服务器的类型加载不同的组件，另外用户可以根据应用需求自定义组件。组件的加载主要是使用load方法，例如：

app.load(pomelo.proxy, [options]); //options参数可选

## 配置router

router主要负责路由信息的维护，路由计算，路由结果缓存等工作，并能根据需要切换路由策略，更新路由信息等。用户可以自定义不同服务器的不同路由规则，然后在app.js中进行配置即可。例如为chat服务器配置路由规则：

//routeUtil.js

app.route('chat', routeUtil.chat);

在routerUtil中可以具体的定义不同服务器的路由规则，例如：

routeUtil.chat = function(session, msg, app, callback) {

var chatServers = app.getServersByType('chat');

if (!chatServers) {

callback(new Error('can not find chat servers.'));

return;

}

var server = dispatcher.dispatch(session.rid, chatServers);

callback(null, server.id);

};

在回调函数中返回服务器的id即可，这里使用dispatcher对session.rid进行hash处理从而完成服务器选择。

## 配置filter

当一个客户端请求到达服务器后，经过filter链和handler处理，最后生成响应返回给客户端。handler是业务逻辑实现的地方，filter则是执行业务前进行预处理的地方。为了开发者方便，系统提供了默认的一些filter，例如：serialFilter,timeFilter，timeOutFilter，另外用户可以根据应用的需要自定义filter。filter默认提供的配置类型如下：

* serial： 主要负责保证所有从客户端到服务端的请求能够按顺序地处理。

app.filter(pomelo.filters.serial());

* time： 主要负责记录请求的相应时间。

app.filter(pomelo.filters.time());

* timeOut: 主要负责监控请求响应时间，如果超时就给出警告。

app.filter(pomelo.filters.timeout());

## 完整样例参考

var pomelo = require('pomelo');

var routeUtil = require('./app/util/routeUtil');

/\*\*

\* Init app for client.

\*/

var app = pomelo.createApp();

app.set('name', 'chatofpomelo');

app.defaultConfiguration();//此方法已经不能用

// app configure

app.configure('production|development', function() {

// route configures

app.route('chat', routeUtil.chat);

app.route('connector', routeUtil.connector);

// remote configures

app.set('remoteConfig', {

cacheMsg: true,

interval: 30

});

// filter configures

app.filter(pomelo.filters.timeout());

// mysql configures

app.loadConfig('mysql', app.get('dirname') + '/config/mysql.json');

});

// start app

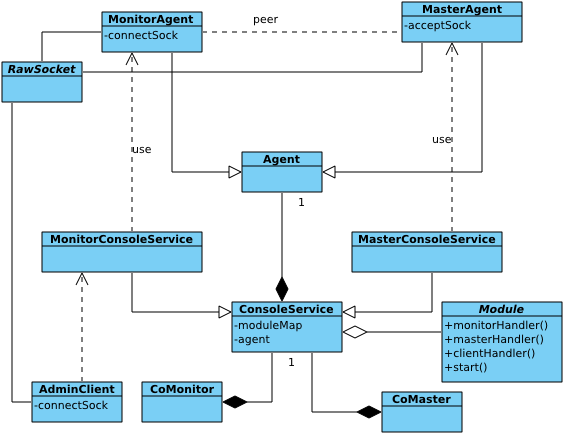
app.start();

# 服务器管理框架

在前面的介绍中，我们知道pomelo支持动态地增加以及删除某一个服务器，于是我们会问当增加一个应用服务器的时候，已经启动的应用服务器是怎么知道新增加了一个服务器的；当停止某个服务器时，其他服务器又是怎样获知这些消息，并使得以后的rpc或者消息不路由到已经停止的服务器的。还有，如果查看整个服务器群的信息，或者整个服务器群的运行状态。诸如此类的问题，pomelo提供了一个管理框架，通过对这个框架进行扩展，用户将可以自己定制一些需要监控的信息，下面将对此进行介绍。

### pomelo管理框架

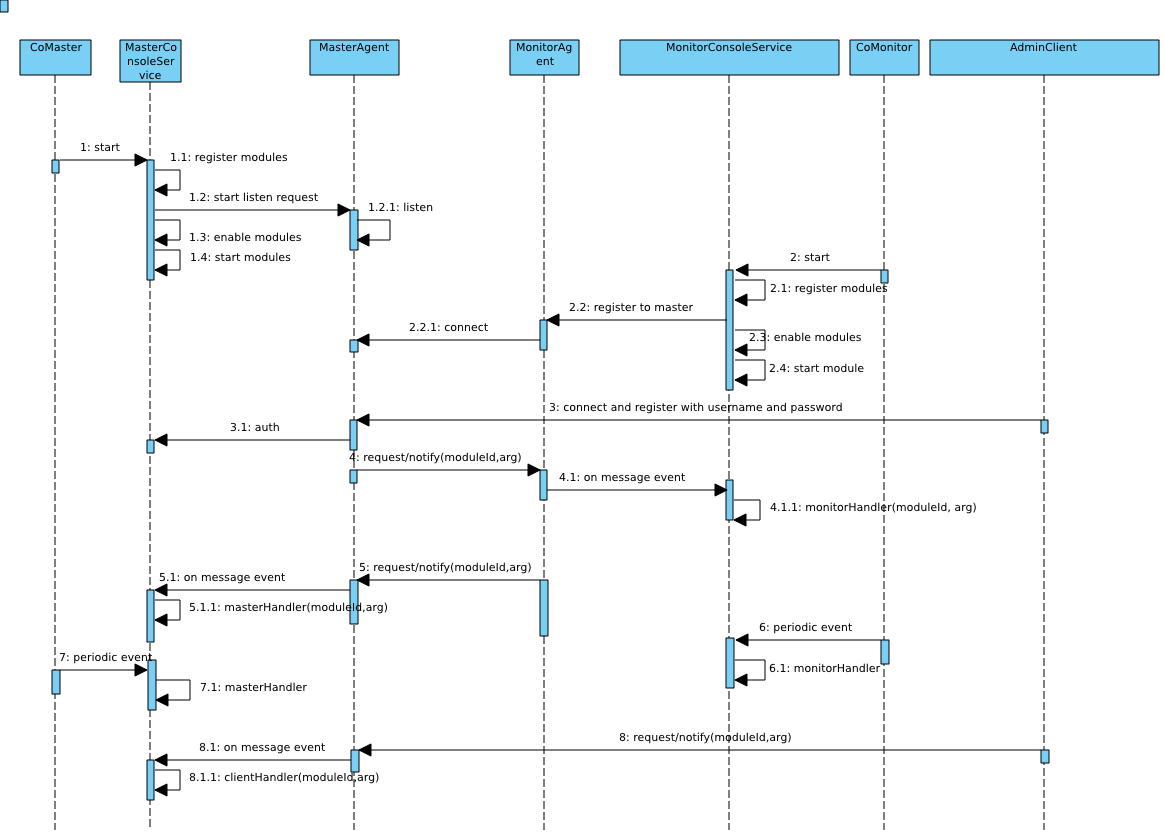
* pomelo的管理框架中，将对应的主体分为三种角色，分别为master，monitor，client。其中master可以认为对应于master服务器的Master组件，monitor可以认为是所有服务器都加载的Monitor组件。而client，可以认为是第三方的管理工具，pomelo提供的命令行工具，pomelo-cli以及admin-console-web都扮演了client角色。
* 一般情况下，监控管理模型大致可以分为两类：
  + master向monitor发出请求某些监控信息，monitor向master上报其信息，master获得到信息后进行缓存。第三方的client会连接到master上，请求整个服务器群的监控信息，master将缓存的监控信息返回给client。
  + client连接到master上后，给master发送一条命令，比如请求关服务器命令，master获得这个命令后，也就是关闭服务器命令后，master会向monitor广播此关闭命令，每一个monitor收到关闭命令后，关闭自身。
* 下面展示了监控框架相关的类图,这个是从前面的整个框架的类图中取出来的，与管理框架相关的类:



在上面类图中，Master组件，Monitor组件，AdminClient分别扮演我们上面提到的master，monitor和client角色，下面的叙述中将对这些术语不加区分的混用。

* 对于Master端，MasterConsoleService会管理所有已注册的Module，生成以ModuleId为主键的Module的map。MasterAgent监听端口，承受来自monitor和client的连接与注册，接受monitor和client的request/notify，向monitor发送request/notify，需要注意的是，master不会向client发出request/notify,只会针对client的request进行response。
* 对于Monitor端，MonitorConsoleService会管理所有已注册的Module，维护Module的map。所有的服务器通过配置文件都能获知到master服务器的监听地址和端口，MonitorAgent会主动发起连接到master,并维护对应的连接，然后通过此连接与master端通信。
* 对于client端，要求使用一个用户名和口令主动发起到master的连接及注册，验证通过后，AdminClient会维护到master的连接，然后就可以向master发送request/notify了。
* 所有与通信相关的类，都为维护自己的socket信息，对于Master来说，不仅仅有连接的socket，还有监听的socket。在整个管理框架中，通信层使用的socket.io。
* 每一个Module会定义四个回调monitorHandler，masterHandler，clientHandler，start，对于不同用途的Module，可能会省略掉一些用不着的回调函数定义，也就是说这四个回调都是可选的。每一个回调函数的签名为XXXHandler(agent, msg, cb),第一个参数指出调用的Agent，可以是MonitorAgent或者MasterAgent，第二个参数是request/notify中的请求体，第三个是回调函数，如果是请求的话，最后结果通过回调第三个参数返回结果，否则的话，忽略第三个参数。

下面我们用一个类时序图，来说明整个框架的控制流程:



实际上，控制流程是多个，这里为了省事，将所有流程都画到一个图中了，对于后面的一些行为实际上是可以没有先后顺序的，希望读者能够甄辨。

#### master组件的启动

master服务器总是率先启动，Master组件在其start调用的最后才会调用Starter.start,Starter.start才会启动所有的应用服务器，因此Master组件总是最先start。在Master组件的start调用中，会完成以下几步:

* 加载注册Module到MasterConsoleService，Module的导出方式有两种，可以导出工厂函数，也可以导出对象，如果导出工厂函数的话，其签名应该是 FacFunc(opts, ConsoleServicee),其中opts是用户调用app.registerAdmin的时候传入的，ConsoleService则是具体的加载注册Module的MasterConsoleService。
* 在加载注册完所有的Module后，会开启MasterAgent对端口的监听，此时，master就已经可以接收来自monitor和client的request和notify了。
* 开启监听后，MasterConsoleService会enable所有的module，这步操作主要是看看有没有module配置了周期性地拉去monitor信息，也就是module的配置中有type选项和interval选项，且type的值为'pull'，interval指定了周期，则认为其配置了周期性监控操作，此时会完成周期性事件的调度，使得master可以周期性地获取监控信息。
* 最后如果有Module定义了start回调，将会在这里调用，一般在start回调里会做一些初始化信息。 经历了这些步骤后，master完成启动。

#### monitor组件的启动

由于应用服务器是在Master组件启动后期才创建，因此monitor总是后于master启动。monitor的启动过程与master类似，唯一不同的就是，monitor会发起到master的连接，而不是监听接口。monitor中同样也会使用与master完全相同的方式，加载注册Module，如果有Module配置了周期性地推送监控数据到master的话，即其配置type的值为'push',这里也会调度对应的事件，使得能够周期地推送数据。最后如果有Module定义了start的话，则会回调start。Monitor的启动过程与master基本一致。

#### client的连接注册

client会连接注册到master上，因此，client需要在master开始监听端口后，才能成功连接上，在连接到master上后，基于安全的缘故，master需要使用用户名和密码对其进行验证，具体的客户端验证的配置文件为config/adminUser.json，例子程序里有相应的配置示例。如果master提供的用户名和密码，通过了验证，那么客户端就在master上注册成功，就可以给master发request/notify了。

#### 周期触发

对于配置了type和interval的Module，它们被认为是需要周期性进行回调的，在前面的enable module阶段，已经对其进行了调度。如果配置的type为'pull',那么每隔interval秒，在master端，其对应的masterHandler将会被回调，回调的时候，不会传入参数。

#### 主动请求

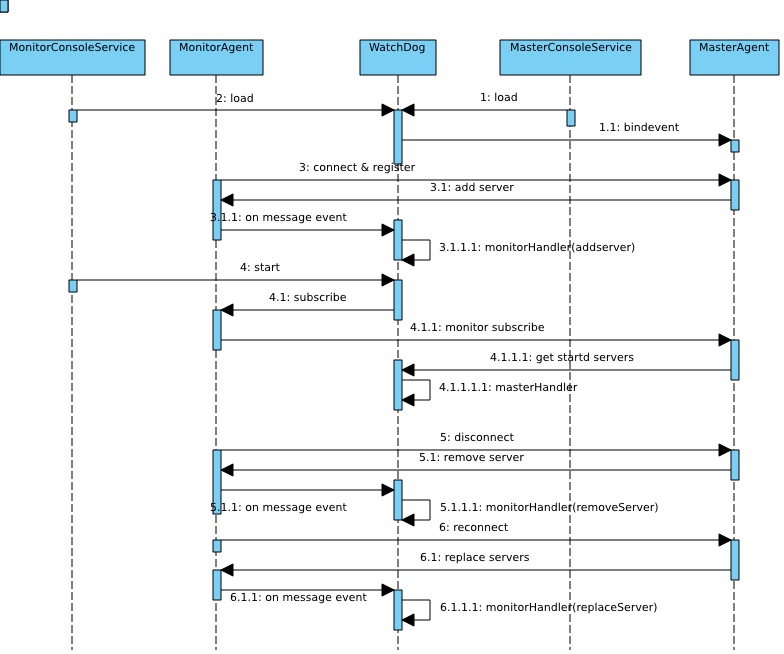
* 当有monitor向master发出request/notify的时候，请求参数会指出相应的ModuleId以及回调调用的参数，在master端，对应的Module的masterHandler将会被回调，此时回调会使用monitor请求中携带的参数。因此，通过对masterHandler请求的参数进行判断就可以区分到底是周期性任务还是monitor请求。
* 对于monitor端，与master类似，当某个Module配置了type为'push'的时候，其对应的monitorHandler将会被回调，当master给monitor发送request/notify的时候，其对应的monitorHandler也同样会被回调。与master一样，可以通过调用是否有参数进行区分是周期性的任务还是接收到了master的消息。当然，即使是master的消息，也可能没有携带任何参数，这种情况只能由用户自己处理了，一般来说，为了便于区分，不要发送不带参数的request/notify。
* client在连接到master上后，client可以向master发送request/notify，请求信息中带有ModuleId和对应的回调参数。master接受后，对应Module的clientHandler将会被回调。这里要注意的是master不会主动向client发送request/notify,只会对client的消息进行响应。

以上的用例行为基本上描述了pomelo的管理框架的执行流，下面会对pomelo内建的两个module做一些分析。

### watchdog分析

我们知道对于服务器配置的静态信息可以从配置文件中直接读取，但是由于服务器可以在运行时增加和停止，而整个服务器群的其他服务器也需要获知具体的服务器的动态信息，就需要一种机制来实现这一切。

pomelo是通过内建的Module watchdog实现这一切的,下面继续通过选取典型的用例场景，使用非正式的时序图来说明其流程:



#### 绑定连接/断开事件

在加载watchdog这个module时，在master端，除了会监听端口外，还做了一件很重要的工作，就是将底层socket的事件由MasterAgent捕捉后，重新抛出，由MasterConsoleService捕捉后进行处理，这些事件为：

* register事件，即一旦有MonitorAgent发起到MasterAgent的连接时触发，MasterConsoleService会在这个事件的处理中，发起广播通知增加的服务器信息。
* disconnect事件，即一旦有MonitorAgent断开连接时触发，MasterConsoleService会在这个事件处理中，发起广播通知有服务器离开的消息。
* reconnect事件，是当有应用服务器重连时触发，MasterConsoleService会在这个事件处理中，发起广播通知有服务器重连的消息。

#### 新增服务器

用例行为4展示了在monitor端，加载watchdog这个module的时候，会在start阶段，执行watchdog的start，在这里，monitor会向master发起一个订阅请求，也就是说此时monitor请求订阅所有的服务器变化信息，当MasterAgent接收到请求时，masterHandler会回调，通过检查参数，获知是一个subscribe操作，masterHandler的回调中会返回所有目前已经启动的服务器的信息给这个新启动的服务器，并将其加入到监听者列表，以后每次再有服务器变动的时候，也会将具体的服务器变动消息发送给此服务器。

用例行为3展示了，当新增一个服务器的时候的交互行为，有monitor发起到master的注册请求，从而激发了master端的register事件处理，其行为为通过MasterAgent广播新增服务器的notify到所有已经订阅注册的服务器，这些服务器收到notify后，其monitorHandler被回调，在monitorHandler中会调用app.addServers方法，这样所有的服务器群都会获知新增的服务器。

#### 停止服务器

用例行为5展示了当有应用服务器断开的情况，当有服务器断开的时候，会激发MonitorAgent的disconnect事件，在这个事件处理中，MasterConsoleService会发起广播notify到所有的服务器，其他的服务器收到此notify后，monitorHandler会被回调，在回调中，通过判断参数，获知是有服务器停止工作，调用app.removeServer删除相应的服务器。同时，在master端，watchdog的监听者列表里也会将这个服务器的信息删除。

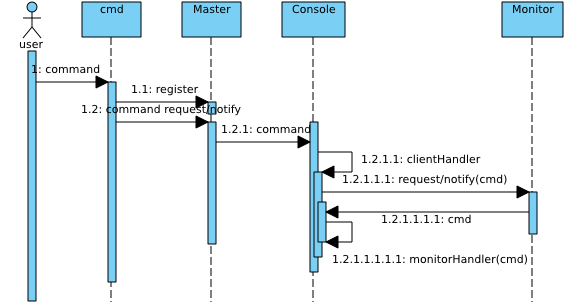
#### 服务器重连

用例行为6展示了当有应用服务器断开重连的情况，具体行为跟前面的服务器加入和离开类似，读者可以结合源码自行分析。

watchdog是pomelo内部很核心的一个module，用来完成服务器状态信息交换。因此在这个module中仅仅涉及到master，monitor角色，没有client角色,因此省略了与client有关的clientHandler回调函数的定义。其事件是由底层socket连接来激发的，而不是周期性地由定时器激发,所以在module的定义中并没有指定其type和interval配置。

### console分析

这里在简单分析一下console组件，console组件主要为pomelo命令行工具服务，



图中所示的用例为一般情况下，用户通过一个命令来管理整个服务器群的方式。我们以命令行工具的list命令为例子来说明：

* 用户执行pomleo list [options],此时，命令行工具会创建一个AdminClient，然后向master发出注册请求，后面的参数用来指定master的位置，端口号以及向master注册所使用的用户名和口令，master用此来进行校验身份，给予不同的权限。其具体的参数以及参数默认值，在pomelo命令行工具里面会有详细的介绍。
* 当AdminClient向Master注册成功后，给其发送请求，参数部分指定了moduleId和具体要做的操作，在我们这个例子中，moduleId就是console，具体的操作为list命令。
* 在master端，收到请求后，其console module的clientHandler被回调，在回调中，通过判断其操作是list，于是向所有与其连接的monitor请求服务器信息。
* 在monitor端，收到master发出的请求后，其console module的monitorHandler被回调，在回调中会获得自己的服务器信息，包括pid，heapused等信息响应给master。
* master在收集完所有monitor的响应后，将获得到的服务器信息数据响应给AdminClient，也就是会由命令行工具收到，并显示出来。

大部分的客户端执行命令流程都是如上面所示，一般情况下，如果配置了周期性地由master去拉取或者由monitor主动推送消息的话，那么当客户端发出请求时，master就可以直接返回其缓存的东西，而不需要一旦客户端发起请求，就向所有的服务器轮询，我们在前面的教程里实现的timeReport module就是使用了周期拉取，当客户端请求时，直接返回缓存的方式。

在console这个module中，由于没有monitor给master发request/notify，所以console的masterHandler回调可以省略，同样console还省略了start回调，因为这里没有什么需要在正常的请求响应之前要执行的东西。

需要注意： 在命令行工具中，add命令现在已经过时了，也就说console module中关于处理add方法的部分现在已经过时，当需要add服务器的时候，推荐使用pomelo-cli，那是一个更强大的交互式命令行工具，pomelo-cli使用的module定义在pomelo-admin中，包括watchServer等。读者可以按照前面的分析方式，自行分析pomelo-cli以及pomelo-admin中定义的module。在pomelo应用中，通过app.enable('systemMonitor')，将会使得应用默认注册pomelo-admin中定义的module，否则，仅仅有console和watchdog会被默认注册。还有一个基于web的监控工具pomelo-admin-web,它提供了通过web页面来看服务器管理监控信息的方式，它也是一个监控管理的客户端。

### 权限管理

在客户端连接到master上的时候，需要对客户端进行身份验证，验证使用用户名和口令，用户的权限分为三个等级，他们分别对应不同的权限。用户的配置信息在adminUser.json中，其中level 1的权限可以执行任何操作，其他level的权限的控制权限收到限制。关于权限控制以及用户信息的配置，可以参考[pomelo-cli的相关文档](https://github.com/NetEase/pomelo/wiki/pomelo-cli%E4%BD%BF%E7%94%A8)。

### 小结

本部分详细介绍地介绍了pomelo的监控管理框架的工作流程，分析了pomelo核心的两个module Watchdog和console的工作原理。结合前面教程中关于module的例子，用户可以很容易地完成自己特殊需要的module的定制。

# adminConsole 开发文档

#### adminConsole 有三种角色，分别是 master,monitor,client

* master : 运行在master进程中，监听端口等待monitor和client的连接。主要负责维护所有已注册的连接，消息路由和处理，以及缓存服务器集群状态信息。
* monitor : 运行在各个需要监控的服务器进程中（包括master进程）。启动后连接并注册到master，主要负责收集被监控的进程信息，向master汇报。
* client : 运行在admin console的web页面。启动后连接并注册到master。主要负责响应用户操作和呈现master返回结果。

#### 三者之间通讯的消息分为两类：需要响应的消息request和不需要响应的消息notify。

* request：如单点请求某一具体服务器上面的信息。
* notify：如收集服务器状态信息的消息。

request消息格式定义：

{

id: 消息id，标志请求和响应的对应关系。Notify不需要这个字段；

moduleId: 消息的路由字段，指明处理该消息由哪个模块处理；

body: 消息内容，处理消息所需的key/value数据

}

notify消息格式定义：

{

moduleId: 消息的路由字段，指明处理该消息由哪个模块处理；

body: 消息内容，处理消息所需的key/value数据

}

#### 类说明

* consoleService  
  是整个监控模块的总入口，master和monitor上都需要创建。各个进程向consoleService注册各自的module。ConsoleService根据当前服务器类型，创建底层的agent实例，并负责agent的启动和关闭。consoleService同时也充当了map的角色，缓存收集到的状态信息。
* masterAgent  
  在master进程中启动，负责底层网络通讯相关工作。如：监听端口，接收monitor和client连接，分组维护这些连接以及将消息推送给指定的接收者等。
* monitorAgent  
  在各个需要监控进程中启动（包括master进程）。启动后连接并注册到masterAgent，负责与masterAgent之间的通讯。
* module  
  监控模块接口，实现监控相关的具体逻辑。定义三个回调接口，分别对应于master，monitor和client上的逻辑。

#### 自定义监控模块开发

adminConsole 采用‘类插件’的开发模式，因此为 adminConsole 开发一个新的监控模块是很容易的  
下面就通过一个 adminConsole helloPomelo 模块来进行说明  
通过请求该监控模块，我们可以从游戏服务器集群中，得到 xxx hello Pomelo (其中 xxx 指的是 serverId)  
进行开发前请先仔细阅读上面的介绍文档以及 [adminConsole-api-文档](https://github.com/NetEase/pomelo/wiki/adminConsole-%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3)

* 首先我们要在项目路径下 game-server/app/modules/ 文件夹下面新建一个 helloPomelo.js 文件

var Module = function(app, opts) {

opts = opts || {};

this.type = opts.type || 'pull'; // 设置该参数，表明该模块监控是由 master 主动拉数据

this.interval = opts.interval || 5; //master 主动拉数据间隔

};

Module.moduleId = 'helloPomelo';

module.exports = Module;

Module.prototype.monitorHandler = function(agent, msg) {

var word = agent.id + ' hello pomelo';

//把 monitor 监控的数据 notify 给 master

agent.notify(Module.moduleId, {serverId: agent.id, body: word});

};

Module.prototype.masterHandler = function(agent, msg) {

//如果没有消息，则notify 所有的 monitor 拉取数据

if(!msg) {

agent.notifyAll(Module.moduleId);

return;

}

//收集从 monitor 拉取的数据

var data = agent.get(Module.moduleId);

if(!data) {

data = {};

agent.set(Module.moduleId, data);

}

data[msg.serverId] = msg;

};

Module.prototype.clientHandler = function(agent, msg, cb) {

//客户端请求，直接返回 master 中cache的数据

cb(null, agent.get(Module.moduleId) || {});

};

* 然后在项目目录下的 app.js 中，添加注册模块的代码

app.configure('production|development', function() {

});

在configure里面添加如下代码就完成了 helloPomelo 监控模块的注册工作

app.registerAdmin('helloPomelo',new helloPomelo());

* 然后我们可以在 adminConsole web 项目中的 views 文件夹下，添加一个 helloPomelo.html 来进行简单的测试

<html>

<head>

<title></title>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script src="/js/socket.io.js"></script>

<script src="/js/util/protocol.js"></script>

<script src="/js/client.js"></script>

</head>

<body>

<div id = 'main'></div>

<script type="text/javascript">

var client = window.client = new ConsoleClient();

var host = 'localhost';

var port = '3005';

client.connect('xxx', host, port, function(err){

if(err) throw err;

client.request('helloPomelo',null,function(err,msg){

var dom = document.getElementById('main');

for(var key in msg){

var elem = document.createElement('p');

elem.innerText = msg[key]['body'];

dom.appendChild(elem);

}

console.log(msg);

});

});

</script>

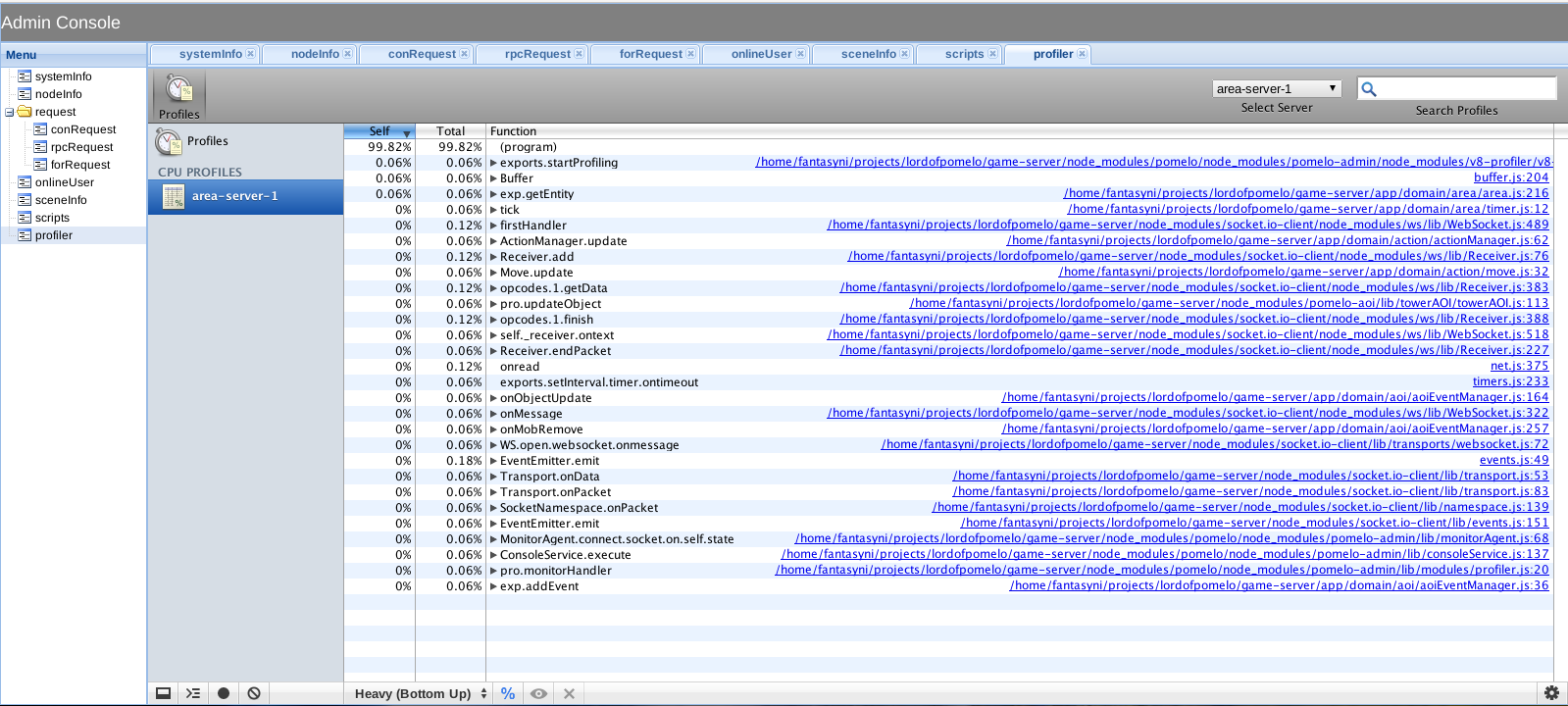
</body>

</html>

* 启动项目主程序以及 adminConsole web  
  打开支持websocket的浏览器，如 google chrome，输入<http://0.0.0.0:7001/module/helloPomelo>  
  就可以看到从游戏服务器集群中获取到的数据  
  [](http://oss.aliyuncs.com/pomelo/1.png)

# adminConsole安装与简介

#### adminConsole 参考文档

adminConsole 是 pomelo 框架中用于监控的模块，可以通过 web 端的方式来对游戏服务器集群的运行状态，性能，日志等进行实时的监控，它采用‘类插件’的开发模式，开发者可以很方便的扩展具体的监控模块逻辑，目前在 adminConsole 中，集成的监控模块有如下几个：[](http://oss.aliyuncs.com/pomelo/adminConsole1.png)

* systemInfo  
  用于监控各个服务器上的系统信息，包括 loadavg,men,CPU(I/0),DISK(I/0)
* nodeInfo  
  用于监控各个服务器上的进程信息，包括 pid,cpu%,mem%，vsz，rss
* conRequest  
  用于监控由 connector 请求所产生的日志，包括玩家的登入，移动，切换场景等所花费的时间，并给出具体的路由(route)
* rpcRequest  
  用于监控游戏服务器中 rpc 的调用情况，所花费的时间
* forRequest  
  用于监控由 forward 请求所产生的日志
* onlineUser  
  用于实时监控在线玩家的信息，包括user login name，login ip，login time
* sceneInfo  
  用于实时监控玩家的场景信息，包括玩家所在的服务器，玩家所在的坐标等
* scripts  
  该模块提供了可以在 adminConsole 端在具体的服务器上执行脚本(script)
* profiler  
  该模块集成了chrome控制台下面的 Profiles 性能分析工具，可以用来对Pomelo服务器端的代码进行性能分析

#### adminConsole安装与使用

运行环境：linux or mac os x

git clone <https://github.com/NetEase/pomelo-admin-web.git>

cd pomelo-admin-web

node app

浏览器中访问： http://localhost:7001， 就可以打开管理控制台界面。  
如果在此之前已经启动了 pomelo 项目，就可以在 adminConsole 上面进行监控了  
如果端口有冲突，请在config/admin.json修改端口，访问的浏览器必须支持websocket，推荐使用chrome.

#### scripts模块脚本编写注意事项

scripts模块使用了 node.js 中的 [vm](http://nodejs.org/api/vm.html) module 来执行脚本  
内置提供了  
app --- pomelo application 实例  
os --- os 模块  
fs --- fs 模块  
process --- process 模块  
util --- util 模块  
来做为 vm 的 sandbox 上下文环境，即我们可以直接在脚本中调用这些  
为了便于输出结果，在 adminConsole 中，把执行的结果统一赋值给了全局 result 变量  
因此，在编写脚本的时候，要输出的结果要赋值给 result 变量（不要用 var 进行声明，它是全局的）  
比如，可以编写一个脚本来获取服务器的cpu数量信息

var cpus = os.cpus();

result = util.inspect(cpus,true,null);

# Connector实现

在与客户端通信的时候，pomelo目前提供了hybridconnector和sioconnector，其中hybridconnector支持tcp，websocket;sioconnector支持socket.io。但是实际编程中，只有这些connector可能还无法满足我们的需求，我们可能需要自己定制自己的connector，pomelo提供了定制connector的接口，在这部分就是要说明这个问题。

#### 内建的connector分析

pomelo内建的connector包括sioconnector和hybridconnector，这里以sioconnector为例来分析，说明如何实现一个connector。 首先sioconnector的构造函数里需要三个参数host、port、opts，(host,port)是要监听socket绑定的，在sioconnector的start调用中，会开启对应的监听，并使得当有连接事件发生时，能够将连接事件抛出。抛出连接事件的时候，对应的通信socket是SioSocket。SioSocket的事件中，当客户端主动断开连接时，需要触发disconnect事件，当通信出现错误时，触发error事件，当有消息获得的时候触发message事件。当收到客户端的请求message或者需要给客户端发送回应或者推送消息的时候，pomelo会使用connector的decode函数对数据进行解码。对于客户端请求的消息，其上报到应用层的格式应为：

{

id : <requestId>,

route: <handlerRoute>,

body: <requestBody>

}

* 这里id是客户端请求的requestId，这个数值由客户端产生的，因此不同session的请求id是互相不干涉的。如果不是请求，只是一个notify的话，id值为空。route是对应应用服务器的请求位置，格式为".."。body为具体请求时携带的参数信息,当我们发起filter-handler链对请求进行处理的时候，参数msg就是这里的body值，是在这里产生的。
* 对于服务器端给客户端的响应或者服务器端的推送消息，会使用connector的encode进行编码，编码函数的签名为:

encode(reqId, route, msg);

其中，如果是响应的话，reqId是对应的请求id，route为请求携带的route,实际上可以省略，msg会具体的相应内容;

* 如果是推送消息的话，reqId为空值，route为客户端的route，也就是"on"这种格式。其返回值应该是能被socket发送的缓冲区。
* decode/endcode函数还可以通过配置选项opts来进行配置，如果在opts里指定了encode/decode，那么将会优先使用opts中指定的encode/decode。
* connector还应实现stop方法，对于sioconnector来说，其stop方法里面关闭了监听socket连接。

经过对sioconnector的分析，我们完成了pomelo中connector的抽象。

#### connector抽象

在涉及到客户端与服务器端进行通信的时候，往往都会使用类似与tcp服务器客户端通信的模式，我们的connector抽象也不例外。在这里，我们假定要实现一个FooConnector，看看需要实现些什么。

* 首先需要一个FooConnector，FooConnector的构造函数需要一些自己的参数，这些参数要包括创建一个连接监听所需要的所有信息，当然比较经典的参数信息是(host,port)对，但是对于一些比较特殊的connector来说，其连接监听所需要的信息可能就不是(host,port)对了，比如用于本地通信的share memory以及命名管道等。
* FooConnector需要实现一个start函数，在这个start函数中，会启动对连接的监听，最经典的操作应该是listen调用。当然在具体实现一些比较特殊的connector的时候，可能不会是listen调用。start中还应该对实现当有连接到来的时候，触发connector的connection事件，对应经典实现的accept返回，当然是异步的触发。在触发connection事件的时候，应该传出参数FooSocket。
* FooSocket是FooConnector触发connection事件时传出的参数，可以类比经典tcp通信中，accept返回的用于通信的socket。当FooSocket出现通信错误时，触发error事件;当遇到客户端主动断开连接时，应该触发事件disconnect;当收到客户端传来的消息时，应该能够触发message事件，message事件携带的参数为FooBuffer类型。
* FooBuffer类型可以类比经典tcp通信中的byte数组，在javascript中，FooBuffer一般为string或者Buffer类型，当然用户也可以定制自己的类型。需要注意的每一次message事件的触发，携带的都应该是一个完整的包，不能出现半包以及粘包，用户的connector实现要保证这一点。当然，只有类似与tcp这样的流协议的时候才可能出现半包粘包的问题，一些基于数据报的协议是不会出现这种问题的。
* Connector还需要提供一个decode函数，这个decode函数的参数为上面FooSocket的message事件携带的FooBuffer，这个decode函数应该能够把这个FooBuffer解码成具体的应用层能够直接使用的请求对象，也就是说decode函数的返回值应该为如下格式的对象:

{

id: <requestId>,

route: <handlerRoute>,

body: <requestBody>

}

* 当服务端向用户发起消息推送或者响应的时候，Connector应该提供encode函数，完成具体的打包操作，Connector提供的encode函数签名应该为:

encode(reqId, route, body);

encode函数的返回值类型为FooBuffer，也就是FooSocket可以处理的类型，encode完成用户的请求数据的打包。如果是服务端推送数据的话，reqId的值应该为空。

* 由于我们自己定义了encode/decode函数，因此我们可以设计实现我们自己的线上协议，只要在应用层抛数据的时候保持一致即可。
* 当具体发送打包后的数据时，需要FooSocket提供一个发送方法send，send的参数就是encode打包后的FooBuffer，其签名为：

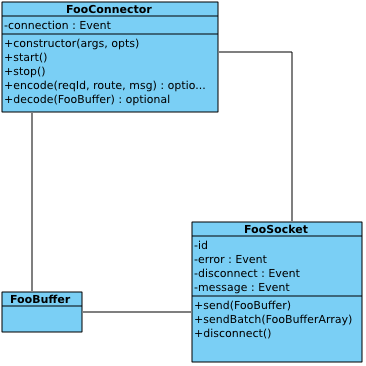
send(FooBuffer);

* 在具体实现数据往客户端发送的时候，有时候并不是一旦数据产生就往外发出，而是会先缓存，定时发送数据，这样对于大量小包产生的场景，将会带来很大的性能提升。因为产生的很多小包，可以批量发出，因此FooSocket还要支持批量发消息，对应的函数签名为:

sendBatch(FooBufferArray);

一般来说，对于底层协议是二进制协议的话，在打包的时候由于已经定义了包边界，因此当有多个包的FooBuffer要发出的时候，只需要将所有的Buffer打包成一个大的Buffer就可以了，hybridconnector中就是这样实现的。而在sioconnector中，因为其底层协议使用的json，因此其将这些消息组成了一个json数组，具体可以参考它们对应的实现。我们在实现FooSocket的sendBatch方法时，可以根据具体的实际情况进行实现，其语义就是一次批量发送多个包。

* FooSocket还应该有主动断开连接的方法disconnect，用于当服务器想把某个用户kick掉的时候调用。
* 当最后应用程序关闭的时候，FooConnector需要提供stop方法，用来关闭相应的连接监听，可以类比tcp服务器中最后关闭用来监听的socket。
* 综上所述，我们得出与定制Connector相关的类图，如下:



* 在app.js中，如果我们的FooConnector的构造函数使用的绑定地址信息是(host,port)对的话，我们通过如下的调用，启用我们自己定制的connector:

app.set('connectorConfig', {

connector: FooConnector,

encode : <encode func>, //optional

decode : <decode func>, //optional

others: <others>

});

这里需要指出，通过opts配置的encode/decode会优先使用，如果没有通过opts配置encode/decode的话，将会使用connector配置的encode/decode，如果既没有配置encode/decode，又没有对connector实现decode/encode，将会出现错误。这里的connector配置项使用的是FooConnector的构造函数，当pomelo构造connector时，如果发现配置的是一个构造函数的话，会按如下的方式进行构造:

new FooConnector(host, clientPort, opts);

如果定制的FooConnector的构造的时候使用的地址信息不是(host,port),那么你就不能使用这种方式进行配置，你可以使用如下方式进行配置:

var conn = new FooConnector(<addr\_args>, opts);

app.set('connectorConfig', {

connector: conn,

// ....

});

#### 小结

在本部分，阐述了如何实现自定义connector，给出了connector的抽象，介绍了当实现一个全新的connector的时候应该需要实现的内容。pomelo对于connector的实现是完全开放的，用户可以根据自己的需求定制connector，定制自己的通信协议，只要自己的客户端与服务端线上协议保持一致就行了。

# LordOfPomelo介绍

#### LordOfPomelo介绍

[LordOfPomelo](https://github.com/NetEase/lordofpomelo)是一个基于Pomelo框架开发的分布式MMORPG游戏[Demo](http://pomelo.netease.com/lordofpomelo/).

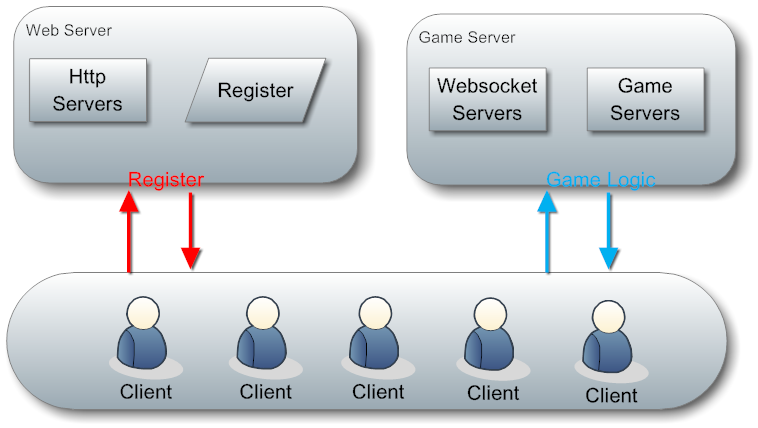
#### LordOfPomelo的主要内容

涵盖了主流MMORPG的核心内容: 多个不同游戏场景, 两种职业, 多种类型的任务, 丰富多彩的道具和武器, 组队, 副本, 以及与各种怪物和Boss的战斗. 玩家可以在多个虚拟场景中穿梭, 完成各种任务, 提升等级, 并和其他玩家互动.

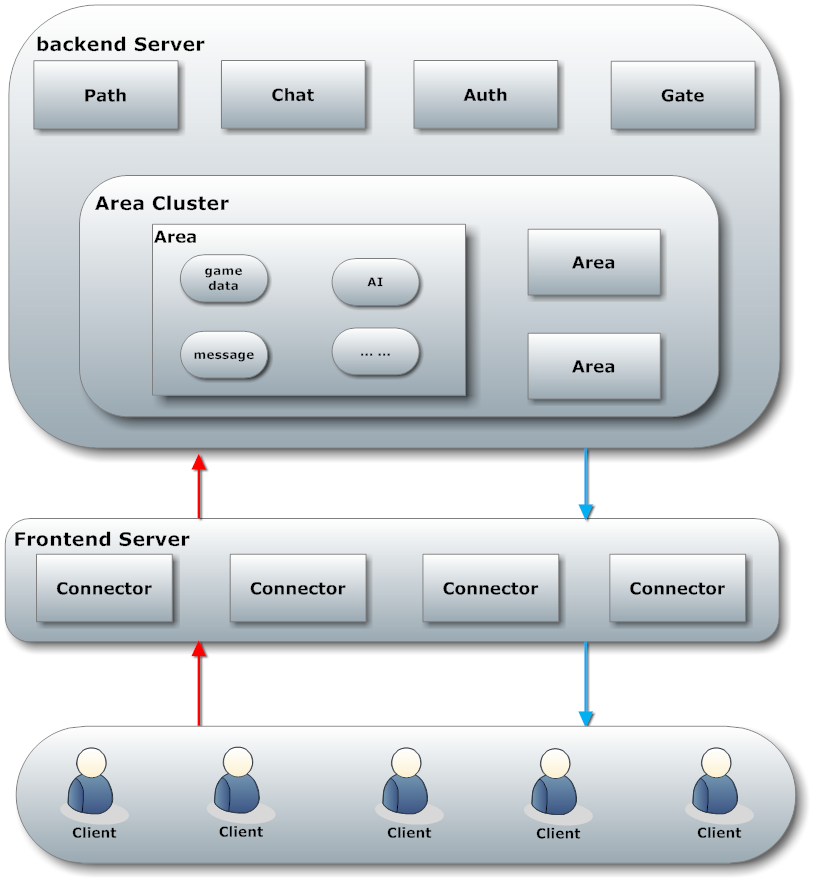
LordOfPomelo采用了基于分区的场景管理, 一张地图表示一个游戏场景, 与一个独立的场景服务器对应, 由该服务器提供对应的服务. 这种设计在避免了复杂的跨服事务的同时提供了对服务器扩展的支持, 开发者可以通过加入新的游戏场景来提高整个服务端的负载能力. 为了考察Pomelo服务器的响应能力, 我们在场景中采用了实时游戏模式: 玩家的攻击, 技能释放, 道具的检取、使用等行为都是实时进行的.

整个游戏采用了Pomelo框架的标准开发模式, 实现了集群化服务器管理, 并且可以使用LordOfPomelo中对线性扩展的支持, 加入新的服务器来提高总体的负载能力. 在经过多轮性能测试与优化后, 达到了单场景800人的负载能力, 同时可以保证良好的响应时间(100ms左右).

#### LordOfPomelo整体架构



如上图所示, LordOfPomelo包括两种类型的服务器: game-server和web-server. web-server是基于HTTP的web服务器, 玩家通过web-server实现注册和登录逻辑. 在玩家完成验证之后就会通过websocket连接到game-server集群, 进入实际的游戏场景之中. game-server是LordOfPomelo的核心服务器集群, 包括一组前端的websocket服务器, 以及后端的游戏逻辑服务器集群, game-server的架构如下图:



上图中的Client可以是任何支持websocket的客户端, LordOfPomelo中自带的客户端是通过HTML5实现的, 不但可以运行在PC的浏览器上, 而且可以运行在其它支持HTML5的终端中（如iPhone, iPad和配置较高的android设备）. 不同平台之间的玩家可以进行对等、实时的互动.

如图所示, game-server中的服务器也分为两类: frontend server和backend server, frontend server是一组websocket服务器集群, 用来处理与websoket客户端之间消息通讯, 负责消息的转发, 过滤, 以及消息广播等功能. backend server则主要用来处理游戏逻辑, 包括各种不同类型的游戏逻辑服务器. 其中, 场景服务器是最重要的游戏服务器, 主要负责游戏场景管理, 游戏数据的更新和保存, 客户端请求的处理, 以及怪物和NPC行为的驱动等. 这些功能是通过与其他服务器的协同工作来实现的. 同时, 场景服务器也采取了可扩展的形式, 每个场景对应一个独立的场景服务器. 可以通过增加游戏场景来分散单个服务器的压力, 提高整体负载.

#### LordOfPomelo分析

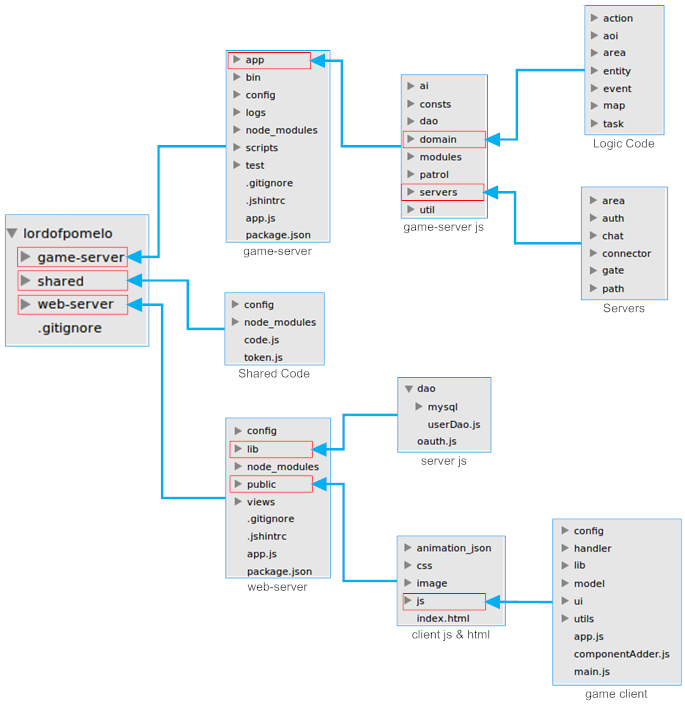
* [LordOfPomelo服务器介绍](https://github.com/NetEase/pomelo/wiki/LordOfPomelo-%E6%9C%8D%E5%8A%A1%E5%99%A8%E4%BB%8B%E7%BB%8D)
* [LordOfPomelo中的数据压缩](https://github.com/NetEase/pomelo/wiki/Pomelo-%E6%95%B0%E6%8D%AE%E5%8E%8B%E7%BC%A9%E5%8D%8F%E8%AE%AE)
* [LordOfPomelo代码组织](https://github.com/NetEase/pomelo/wiki/LordOfPomelo-%E4%BB%A3%E7%A0%81%E7%BB%84%E7%BB%87)
* [LordOfPomelo启动流程](https://github.com/NetEase/pomelo/wiki/LordOfPomelo-%E5%90%AF%E5%8A%A8%E6%B5%81%E7%A8%8B)

# LordOfPomelo 代码组织

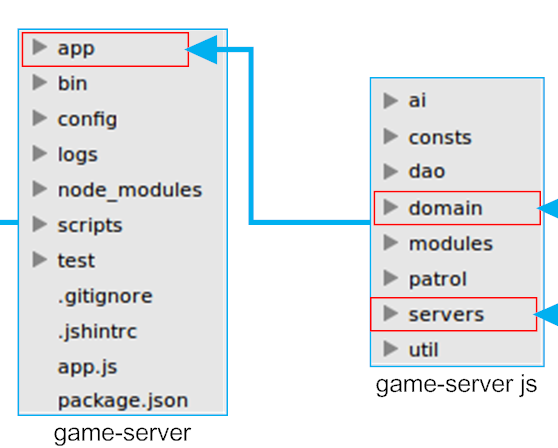
#### LordOfPomelo代码组织

LordOfPomelo的代码主要包括两部分: 后端服务器代码game-server和前端客户端代码web-server. game-server是游戏服务端, 包括所有的游戏逻辑代码和游戏服务器代码. web-server是游戏客户端, 包括用户注册和登录界面代码, 以及一个用HTML5编写的游戏客户端. 除了这两部分之外, 还有一个公用的shared目录, 用来存放前后端共用的代码和配置.

作为一个分布式的游戏服务器, LordOfPomelo可以同时运行在多台服务器上, 却统一使用同一套代码, 不同的服务器会根据配置加载各自的目录代码. 下图是LordOfPomelo的代码结构:



#### game-server服务端代码分析

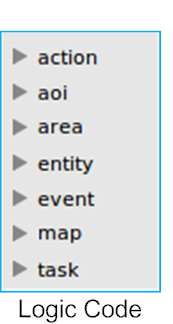


game-server根目录下的app.js是服务器代码的入口, 其他目录的功能如下:

* /app : 服务端js代码, 包括服务器代码和游戏逻辑代码.
* /config : LordOfPomelo中的配置文件.
* /logs : 服务器端运行时产生的日志文件.
* /scripts: 统计模块对应的本地脚本.
* /test : LordOfPomelo中的测试用例

#### 逻辑代码

逻辑代码主要用来完成具体的业务逻辑, 如用来驱动怪物的AI代码, 用来计算地图中路径的寻路代码等, 逻辑代码在/app/domain目录下:



* /action : 负责处理客户端的请求. 由于场景是由tick驱动的, 而tick的间隔一般较短(默认100ms), 当请求需要在多个tick中执行的时候就会被封装为一个action来执行.
* /aoi : aoi相关逻辑, 包括aoi消息的封装, 以及对aoi消息的处理.
* /area : 场景相关逻辑, 提供场景中的主要接口. 包括: 场景中实体的加入、更新和删除, 广播消息的推送, 场景中服务的访问(AOI, AI等), 场景信息的获取等. 同时还包括一个Timer, 用来驱动场景中的逻辑.
* /entity : 场景中的所有实体, 包括玩家, 怪物, npc, 宝物, 装备, 队伍等.
* /event : 用来集中处理场景逻辑中产生的各种事件, 包括玩家消息, 怪物消息等.
* /map : 用来完成地图的加载和解析, 以及地图中区域的抽象.
* /task : 任务相关的代码, 控制任务的执行和取消, 以及任务奖励的获得.

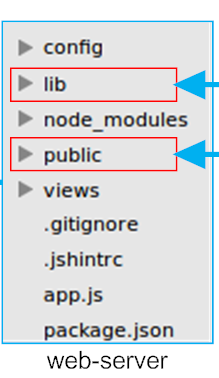
#### 服务器代码.



服务器代码在/servers目录下, 通过规约的形式组织, 对外提供rpc接口, 处理客户端和服务端的请求并返回结果. LordOfPomelo中使用的服务器包括:

* /area : 场景服务器, 用来储存场景信息, 处理客户端的请求, 如用户添加, 删除, 攻击等操作.
* /chat : 聊天服务器, 处理聊天信息
* /connector: 连接服务器, 负责维护用户session, 接受用户数据, 并将服务端的广播数据推送给玩家
* /login : 登录服务器, 用来验证用户登录信息
* /path : 寻路服务器, 用来完成路径计算功能.
* /manager : 副本/组队服务器, 用来管理全局的副本和组队功能.

#### web-server代码架构



LordOfPomelo的页面端代码主要分为两个部分: 基于HTML5开发的ui代码和使用colorbox开发的游戏逻辑代码. ui代码包括注册/登录页面, 游戏场景中的各种选项和菜单. 这些代码基于HTML5开发, 使用css3进行渲染. 游戏场景的绘制和游戏逻辑的驱动则是基于colorbox开发, 并使用到了HTML5中的很多特性.

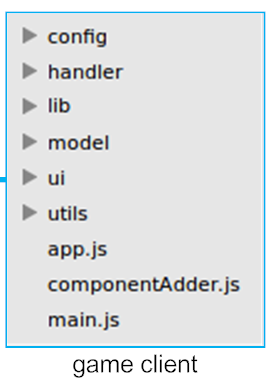
除此之外, web-server中还包括用户注册代码和oauth验证的逻辑, 这些代码在lib目录下.

下图是页面端的内容:



* /animation\_json : 动画相关的json描述.
* /css : 代码中所用到的css文件.
* /image : 客户端中用到的图片资源.
* /js : 所有客户端的js文件.
* /index.html : 是LordOfPomelo的入口文件.

下图是游戏js代码组织:



* /config : 客户端的配置信息.
* /handler : 客户端的handler, 用来处理服务端response请求.
* /lib : colorbox和pomelo的客户端通讯库代码.
* /model : 客户端的游戏逻辑代码.
* /ui : ui代码.
* /utils : 客户端用到的工具类.
* /app.js : 客户端的初始化入口, 负责初始化客户端的逻辑代码.

# LordOfPomelo 启动流程

LordOfPomelo的启动流程采用了Pomelo中的启动模式. 在阅读下面内容前, 请先阅读[pomelo启动流程](https://github.com/NetEase/pomelo/wiki/pomelo%E5%90%AF%E5%8A%A8%E6%B5%81%E7%A8%8B).

app.js是LordOfPomelo的入口, 主要负责所有服务器的配置, 以及组件的加载和启动. LordOfPomelo的启动主要分为两步：启动master服务器, 再由master服务器分别启动其他服务器.

#### 组件的配置和加载

LordOfPomelo中使用了多个外部组件, 这些组件在服务器启动时加载, 提供各种服务：如数据统计, 路由功能的替换, 游戏场景的初始化等.

LordOfPomelo中使用了基于脚本的统计, 这些组件通过运行自定义的脚本, 收集服务器运行数据并生成报告, 更加具体的功能, 请见：

var sceneInfo = require('./app/modules/sceneInfo');

var onlineUser = require('./app/modules/onlineUser');

if(typeof app.registerAdmin === 'function'){

app.registerAdmin('sceneInfo', new sceneInfo());

app.registerAdmin('onlineUser',new onlineUser(app));

}

LordOfPomelo启动时还会加载areas的配置文件, 用来建立场景和服务器之间的映射:

//Set areasIdMap, a map from area id to serverId.

if (app.serverType !== 'master') {

var areas = app.get('servers').area;

var areaIdMap = {};

for(var id in areas){

areaIdMap[areas[id].area] = areas[id].id;

}

app.set('areaIdMap', areaIdMap);

}

为了能在多个场景服务器中正确的路由, LordOfPomelo中加载了自定义的路由组件, 通过使用场景与服务器之间的映射信息, 可以确保玩家的请求被分发到对应的场景服务器上:

// route configures

app.route('area', routeUtil.area);

app.route('connector', routeUtil.connector);

除了服务器的通用配置以外, app.js中还负责不同服务的初始化工作: 如全局服务器的初始化, 场景的初始化, 以及寻路服务器的初始化, 这些初始化会根据服务器的类型进行不同的初始化过程:

app.configure('production|development', 'area', function(){

app.filter(pomelo.filters.serial());

app.before(playerFilter());

//Load scene server and instance server

var server = app.curServer;

if(server.instance){

instancePool.init(require('./config/instance.json'));

app.areaManager = instancePool;

}else{

scene.init(dataApi.area.findById(server.area));

app.areaManager = scene;

}

//Init areaService

areaService.init();

});

数据同步插件和MySql的初始化:

// Configure database

app.configure('production|development', 'area|auth|connector|master', function() {

var dbclient = require('./app/dao/mysql/mysql').init(app);

app.set('dbclient', dbclient);

app.use(sync, {sync: {path:\_\_dirname + '/app/dao/mapping', dbclient: dbclient}});

});

#### 服务器的启动

LordOfPomelo的启动也采用了Pomelo框架中的启动方式, 即将master作为一个默认组件, 在app.js调用app.start()方法后加载, 启动master服务.

master组件会负责启动其他所有服务. 这个启动过程分为两个阶段：第一阶段, master服务启动其他所有服务器, 在服务器启动完毕后, 其中的monitor组件会连到master对应的监听端口上, 表明该服务器启动完毕. 第二阶段, 在所有服务器都启动完毕之后, mater会调用所有服务器上的afterStart接口, 来进行启动后的处理工作.

# LordOfPomelo 安装指南

#### LordOfPomelo介绍

LordOfPomelo是一个基于Pomelo游戏框架开发的MMORPG(大型多人在线角色扮演游戏)的游戏Demo, 具有角色、怪物、装备、战斗、聊天、技能、升级系统、任务系统、组队、副本等较为完整的游戏功能.

LordOfPomelo服务端采用了Pomelo框架, 客户端采用了基于HTML5的colorbox框架, 在大约3个月的时间内实现快速开发. 服务端约8000行代码, 客户端约6000行代码. 支持单场景800人以上的并发访问, 响应时间控制在100ms左右.

#### 运行环境

* [nodejs](http://nodejs.org/)
* Windows、Linux 或 MacOS 操作系统
* MySql 数据库

#### 安装LordOfPomelo

#### 下载源码

git clone https://github.com/NetEase/lordofpomelo.git

#### 安装依赖包

进入目录： cd lordofpomelo

安装依赖包： sh npm-install.sh(Windows: npm-install.bat)

#### 创建MySql数据库

#### 创建数据库

sql文件路径：./game-server/config/schema/Pomelo.sql

* 安装MySql数据库(略)
* 登录MySql: mysql –u用户名 –p密码 (登录成功提示符：mysql>)
* 创建数据库: mysql> create database Pomelo;
* 选择数据库: mysql> use Pomelo;
* 导入sql文件: mysql> source ./game-server/config/schema/Pomelo.sql

#### 修改数据库配置

数据库配置文件为./shared/config/mysql.json

{

"development": {

"host" : "127.0.0.1",

"port" : "3306",

"database" : "Pomelo",

"user" : "xy",

"password" : "dev"

},

"production": {

"host" : "127.0.0.1",

"port" : "3306",

"database" : "Pomelo",

"user" : "xy",

"password" : "dev"

}

}

将"development"环境下的的数据库配置修改为实际的配置.

#### 运行游戏

需要分别启动game-server和web-server. game-server的启动方式：

* pomelo start (pomelo的安装方法参考[pomelo快速使用指南](https://github.com/NetEase/pomelo/wiki/pomelo%E5%BF%AB%E9%80%9F%E4%BD%BF%E7%94%A8%E6%8C%87%E5%8D%97)) 注: 如果上次启动的进程没有完全退出, 可以使用pomelo kill --force来结束所有node进程.

web-server的启动方式：

* cd web-server && node app

#### 访问游戏

本地运行, 直接在浏览器中访问 http://localhost:3001 或者 [http://127.0.0.1:3001](http://127.0.0.1:3001/)

浏览器需支持websocket, 推荐使用chrome.

#### 相关问题解决办法

1. 端口冲突

修改服务器配置文件./game-server/config/servers.json,内容如下：

{

"development": {

"connector": [

{"id": "connector-server-1", "host": "127.0.0.1", "port": 3150, "clientPort": 3010, "frontend": true},

{"id": "connector-server-2", "host": "127.0.0.1", "port": 3151, "clientPort":3011, "frontend": true}

],

"area": [

{"id": "area-server-1", "host": "127.0.0.1", "port": 3250, "area": 1},

{"id": "area-server-2", "host": "127.0.0.1", "port": 3251, "area": 2},

{"id": "area-server-3", "host": "127.0.0.1", "port": 3252, "area": 3},

{"id": "instance-server-1", "host": "127.0.0.1", "port": 3260, "instance": true},

{"id": "instance-server-2", "host": "127.0.0.1", "port": 3261, "instance": true},

{"id": "instance-server-3", "host": "127.0.0.1", "port": 3262, "instance": true}

],

"chat": [

{"id":"chat-server-1","host":"127.0.0.1","port":3450}

],

"path": [

{"id": "path-server-1", "host": "127.0.0.1", "port": 3550}

],

"auth": [

{"id": "auth-server-1", "host": "127.0.0.1", "port": 3650}

],

"gate": [

{"id": "gate-server-1", "host": "127.0.0.1", "clientPort": 3014, "frontend": true}

],

"manager": [

{"id":"manager-server-1","host":"127.0.0.1","port":3750}

]

},

"production": {

// ...

}

}

该配置文件分别定义了development和production环境下的各个服务器的配置信息, 包括服务器类型, 地址, 端口号等, production环境下的参数和development环境的结构类似. frontend参数为true时表示前端服务器. 端口冲突时可以修改相应的端口号.

或者修改文件./web-server/public/js/config/config.js, 内容如下：

\_\_resources\_\_["/config.js"] = {meta: {mimetype: "application/javascript"}, data: function(exports, require, module, \_\_filename, \_\_dirname) {

module.exports = {

IMAGE\_URL:'http://pomelo.netease.com/art/',

GATE\_HOST: window.location.hostname,

GATE\_PORT:3014

};

}};

该文件是web服务器的常用配置信息: IMAGE\_URL是图片等静态资源的地址; GATE\_HOST是gate服务器接受websocket的入口, 所有的websocket请求须先经过gate服务器获取connector服务器的id, 然后再和connector建立连接; GATE\_PORT是前端gate的端口号. manager服务器主要负责是副本和组队功能.

#### 配置文件汇总说明

* ./game-server/config/master.json

master服务器的配置信息, 包括development和production环境下的服务器地址、端口号等.

master服务器负责启动、关闭各服务器, 并监控所有服务器的状态信息.

* ./game-server/config/servers.json

area、connector等服务器的配置信息, 包括development和production环境下的服务器地址、端口号等. 由于connector是前端服务器, 用于接收并转发玩家的请求, 所以会有clientPort.

* ./shared/config/mysql.json

数据库配置信息, 在安装LordOfPomelo之后需要根据数据库安装的实际情况修改development和production环境的参数.

* ./web-server/public/js/config/config.js

客户端图片等静态资源及HTTP访问地址的配置.

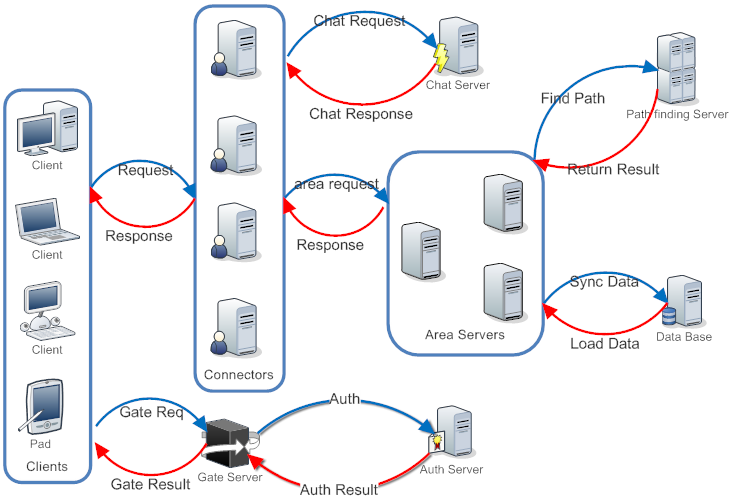
#### 登录问题说明

LordOfPomelo提供了注册新用户以及使用github, google, facebook, twitter和新浪微博授权的方式进行登录. 在使用github等授权方式进行登录时, 需要自己通过OAuth授权认证, 然后修改配置文件中的对应信息(./web-server/config/oauth.json).

# LordOfPomelo 服务器介绍

#### 各类服务器介绍

LordOfPomelo采用分布式的设计, 服务端是由一个服务器集群组成的, 包括：多台场景服务器, 一台或多台寻路服务器, 聊天服务器, 副本/组队服务器, 全局服务器, 长连接服务器等.



#### connecter服务器

与web的短连接模式不同, 在网络游戏中客户端与服务端建立的都是长连接. 而长连接本身是需要一定的资源来维持的, 在LordOfPomelo中, 我们使用websocket协议在客户端和服务器之间建立连接. 而connector服务器就是用来维护这些连接, 并中转客户端和服务端之间的消息. 在LordOfPomelo中, 客户端和服务端的连接状态是通过一个抽象的session来维护的, session是一个客户端在服务端的标识, 用来维护用户的登录状态, 用户的基本信息, 以及用户的websocket连接信息.

#### gate服务器

**gate服务器的主要作用是为用户提供一个统一的websocket入口**, **并负责用户验证和connector服务器的分配.** 与connector服务器不同, LordOfPomelo中只有一台gate服务器. gate服务器会向所有客户端暴露一个固定的websocket接口, **当用户登录时, 会首先连接gate服务器, 完成验证, 并获得由gate服务器分配的对应connetor服务器的信息.** 之后, 客户端会断开与gate服务器的连接, 通过获取的信息连接对应的connector服务器, 获取对应的服务.

#### 验证服务器

验证服务器负责用户注册和验证. 作为用户验证的统一入口, 提供远程调用接口供其他服务器调用, 来进行用户的验证. 验证服务器的主要作用是屏蔽认证验证的细节, 为其他服务器提供统一的验证接口.

#### 场景服务器

在网络游戏中, 出于性能和负载的考量, 大的游戏世界总是会被分成多个区域, 这些不同区域就是场景. 在LordOfPomelo中, **一张游戏地图就是一个游戏场景, 与一台独立的场景服务器对应**. 场景是构成LordOfPomelo游戏世界的基本单位, 不能进行分割和简单的并行扩展. 场景服务器负责维护场景中所有实体, 并驱动实体AI运行游戏逻辑.

场景服务器负责处理游戏中的几乎所有逻辑, 同时为其他服务器提供操纵场景数据的接口. 在LordOfPomelo中, 虽然场景本身不能分割, 但可以通过加入新的游戏场景的方法来分散用户, 从而提高游戏服务器总的负载量. 而一些与场景相关的服务也可以通过独立运行的方式进行水平扩展.

#### 寻路服务器

寻路服务是游戏服务器的基本服务之一, 玩家跑动, 怪物移动都需要寻路服务提供支持. 其功能是根据地图中的起点和终点, 得到一条这两点之间的最优路径. 由于寻路是典型的无状态、计算密集型服务, 在LordOfPomelo中, 我们将寻路逻辑与场景逻辑分离, 放在单独的寻路服务器中, 从而减轻了场景服务器的压力. 而寻路服务器也可以根据需要进行简单的并行扩展. LordOfPomelo中的寻路算法使用A\*实现, 提供了通用的计算接口, 并封装为一个模块, 具体信息见[pomelo-pathfinding](https://github.com/NetEase/pomelo-pathfinding).

#### 聊天服务器

聊天服务是网游的基本服务之一. 在LordOfPomelo中, 聊天服务是与场景服务分离的, 通过一个独立的服务器来实现. 聊天服务器会维护一份所有在线用户的数据, 通过这些数据与connector服务器通讯, 来实现玩家之间的即时通讯.

#### 副本/组队服务器

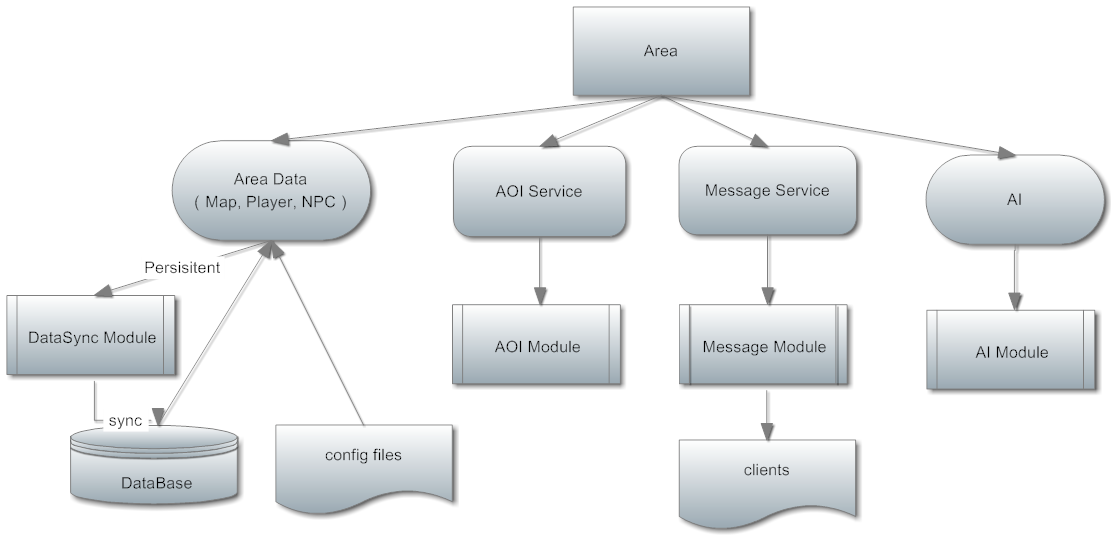
副本/组队服务器是后端服务器集群中负责全局管理副本全生命周期和组队相关操作的功能服务器.具体可参考[lordofpomelo 0.3新特性](https://github.com/NetEase/pomelo/wiki/lordofpomelo-0.3%E6%96%B0%E7%89%B9%E6%80%A7)

#### 场景管理与介绍

LordOfPomelo中的场景服务负责管理游戏中所有实体和整个游戏世界. 而出于安全性和一致性等方面的考虑, 所有的游戏逻辑验证都是在后端进行的, 因此场景服务还包括了游戏中所有的逻辑判断和处理. LordOfPomelo的场景服务是游戏服务的核心, 也是逻辑最为复杂的部分, 下面就对LordOfPomelo中的场景服务进行分析：

#### LordOfPomelo中的场景管理

LordOfPomelo中每个场景对应一个独立的场景服务器, 所有的业务逻辑都在场景服务器内部进行. 对于跨场景的操作, 则是通过一个全局服务器来进行处理. 下图就是一台单独的场景服务器的功能：



#### 实体管理

**游戏场景中的所有实体都会在进入场景时加载到内存中,** **之后所有的修改都会直接在内存中进行, 并通过数据同步模块来定时同步到数据库中**. 这一设计将场景中所有的**数据操作都变成了直接的内存操作, 提高了整体性能, 规避了频繁的缓慢IO操作.**

LordOfPomelo中实体的定义都在domain目录下的entity文件夹中, 采用了面向对象的设计思想, 所有实体都继承自entity类. 实体的加入和删除都通过area中的接口(addEntity/removeEntity)来处理, 从而减轻了对象的管理负担.

#### 消息服务

LordOfPomelo中的消息服务可以分为两种：一对一的RPC请求<-->响应, 以及一对多的广播消息. 由于LordOfPomelo中的逻辑验证是在服务端进行的, 对于客户端的大部分请求, 服务端都会有一个直接的回复来进行处理, 这种回复是使用了类似于web中的request/response模式来实现的.

对于广播服务, LordOfPomelo中使用了基于AOI的区域通知服务. 即在收到广播消息后, 根据消息类型, 通过AOI服务获得需要通知的玩家列表, 然后对该列表中的玩家进行广播, 从而大幅减少了消息的数量.

#### AOI服务

玩家的视野一般远小于场景的大小, 因此对于场景中的绝大部分消息, 进行简单的全场景广播是没有必要而且无法承受的. 当有消息需要发送时, 如何确定该消息需要通知的玩家列表就是AOI模块的功能了.

**LordOfPomelo中实现的是基于灯塔的AOI服务**. **其基本的实现方法是将整个地图划分为若干个等大的tower, 每个tower负责维护一个对象列表**, 指向所有在这个tower范围内的对象. 然后在这个数据结构的基础上来触发各种AOI事件.

# plugin文档

为了方便开发者根据自身的需求对pomelo原有的功能进行有效的扩展，pomelo在0.6版本提供了一种灵活的插件机制。

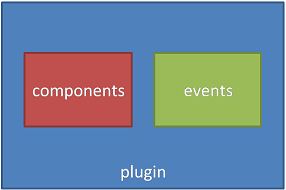
#### Plugins

在pomelo 0.6版中，已开发完成组件包括：[pomelo-sync-plugin](https://github.com/NetEase/pomelo-sync-plugin)，[pomelo-globalchannel-plugin](https://github.com/NetEase/pomelo-globalchannel-plugin)，[pomelo-status-plugin](https://github.com/NetEase/pomelo-status-plugin)，[pomelo-masterha-plugin](https://github.com/NetEase/pomelo-masterha-plugin)；另外可以参考使用插件机制完成的 [chatofpomelo-plugins](https://github.com/py8765/chatofpomelo-plugins)。

* [pomelo-sync-plugin](https://github.com/NetEase/pomelo-sync-plugin)：提供数据同步服务，将[pomelo-sync](https://github.com/NetEase/pomelo-sync)以plugin形式提供使用。
* [pomelo-globalchannel-plugin](https://github.com/NetEase/pomelo-globalchannel-plugin)：提供全局channel服务，默认使用redis存储。
* [pomelo-status-plugin](https://github.com/NetEase/pomelo-status-plugin)：提供用户状态服务，同时提供对指定用户进行消息推送服务。
* [pomelo-masterha-plugin](https://github.com/NetEase/pomelo-masterha-plugin)：提供master节点高可用服务。

#### 结构

插件的结构主要包括两个部分：components和events，其中components是必要的，events则可以根据插件自身所需功能进行配置。如下图所示：



components跟pomelo中原有的组件功能一致，具体可以参考 [pomelo 组件](https://github.com/NetEase/pomelo/wiki/Pomelo-Framework)。它是服务器生命周期的服务单元，一个组件负责实现一类具体的功能。在plugin中开发者可以根据需要定义多个组件，开发者可以根据需要实现服务器不同生命周期的回调方法，包括start、afterStart、stop三个生命周期过程。

events是为了开发者可以对pomelo中基本事件进行处理，开发者可以根据自身的需求对不同的事件进行监听并作出相应的处理。现在主要包括add\_servers、remove\_servers、replace\_servers、bind\_session、close\_session。

* add\_servers：系统添加服务器事件，参数为添加的服务器信息，参数类型是数组。
* remove\_servers: 系统移除服务器事件，参数为移除的服务器信息，参数类型是数组。
* replace\_servers: 系统中有(除master)的服务器断网后重新连接事件，参数为断网的服务器收到系统中现存服务器信息，参数类型是数组。
* bind\_session: 系统中有用户进行session绑定操作事件，参数为session对象。
* close\_session: 系统中有用户session关闭事件（包括连接断开和连接异常），参数为session对象。

#### 使用方法

pomelo中使用plugin的相应API如下：

##### API

##### app.use(plugin, opts)

使用相应的pomelo插件

##### Arguments

* plugin - plugin that uses in pomelo
* opts - attach parameters

pomelo中使用组件只需要在app.js中进行相应配置即可，具体可以参考如下代码：

var statusPlugin = require('pomelo-status-plugin');

app.use(statusPlugin, {

status:{

host: '127.0.0.1',

port: 6379

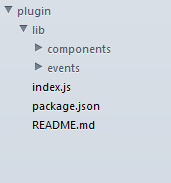
}

});

ps: 由于plugin中可以有多个component，所以相应的配置参数也可能为多个，为了区分不同组件的配置参数，在use方法的第二个参数中可以配置多个组件的配置参数，key为相应组件的文件名，value为配置参数。

### 构建方法

首先，需要创建一个符合plugin规范的空项目，具体的项目目录结构如下图所示：



其次，需要在index.js中进行相关配置，由于在plugin中components是必要的，所以必须在index.js中指明components的路径信息，events如果有用到可以进行配置，具体配置参考如下代码：

module.exports = {

components: \_\_dirname + '/lib/components/',

events: \_\_dirname + '/lib/events/'

};

最后，就可以进行components和events中相关代码的编写。对于component，需要对外提供相应的构造函数，pomelo在加载过程中会将相应的服务器上下文信息和配置参数进行注入，具体可参考如下代码：

module.exports = function(app, opts) {

return new Component(app, opts);

};

var Component = function(app, opts) {

//do construction

};

Component.prototype.start = function(cb) {

// do something application start

};

Component.prototype.afterStart = function(cb) {

// do something after application started

};

Component.prototype.stop = function(force, cb) {

// do something on application stop

};

对于event, 同样需要对外提供其构造函数，pomelo会在加载过程中将相应服务器的上下文信息注入，开发者只需根据自身需要编写相应的回调函数即可，具体可参考如下代码：

module.exports = function(app) {

return new Event(app, opts);

};

var Event = function(app) {

//do construction

};

Event.prototype.add\_servers = function(servers) {

//do something when application add servers

};

Event.prototype.remove\_servers = function(ids) {

//do something when application remove servers

};

Event.prototype.replace\_servers = function(servers) {

//do something when server reconnected

};

Event.prototype.bind\_session = function(session) {

//do something when session binded

};

Event.prototype.close\_session = function(session) {

//do something when session closed

};

# Pomelo 0.2到0.3升级指南

随着Pomelo支持协议的增多，配置文件原先定义的wsPort（代表websocket port）已不适用，现调整为clientPort。并新增一个字段frontend来标识该服务器是否是前端服务器，前端服务器 必须 设置该字段为true，后端服务器可以不设置或设置成false。

以[lordofpomelo](https://github.com/NetEase/lordofpomelo)为例，[servers.js](https://github.com/NetEase/lordofpomelo/blob/master/game-server/config/servers.json)文件修改如下。

connector服务器配置：

"connector": [

{"id": "connector-server-1", "host": "127.0.0.1", "port": 3150, "clientPort": 3010, "frontend": true},

{"id": "connector-server-2", "host": "127.0.0.1", "port": 3151, "clientPort":3011, "frontend": true}

]

gate服务器配置：

"gate": [

{"id": "gate-server-1", "host": "127.0.0.1", "clientPort": 3014, "frontend": true}

]

服务器代码中有依赖到配置信息wsPort的地方也需要做相应的调整。[lordofpomelo](https://github.com/NetEase/lordofpomelo)中的[gateHandler](https://github.com/NetEase/lordofpomelo/blob/master/game-server/app/servers/gate/handler/gateHandler.js#L29)中的res.wsPort需要改为res.clientPort。

#### connector配置

0.2中基于socket.io connector的服务器代码可以无需修改就能使用。

如要使用支持socket和websocket的connector，在app.js配置指定使用hybridconnector即可。使用例子如下：

app.configure('production|development', 'connector', function(){

app.set('connectorConfig', {

connector: pomelo.connectors.hybridconnector,

heartbeat: 3,

useDict: true,

useProtobuf: true,

checkClient: function(type, version) {

// check the client type and version then return true or false

},

handshake: function(msg, cb){

cb(null, {/\* message pass to client in handshake phase \*/});

}

});

});

更详细的信息和多connector的共存请参考[Pomelo 0.3新特性](https://github.com/NetEase/pomelo/wiki/Pomelo-0.3%E6%96%B0%E7%89%B9%E6%80%A7)文档。

#### route字典压缩配置

route字典压缩支持在传输过程中对route字段进行压缩，这一修改对用户是完全透明的，用户可以在不修改任何代码的情况下应用这一功能。只要在app.js中加入配置开关就可以启用字典压缩功能，具体配置如下：

app.set('connectorConfig', {

connector: pomelo.connectors.hybridconnector,

heartbeat: 3,

useDict: true,

handshake: function(msg, cb){

cb(null, {/\* message pass to client in handshake phase \*/});

}

});

其中“useDict：true”项就会打开字典压缩功能，之后所有的系统生成route在传输时就会自动进行压缩，而这一过程对用户是完全透明的。 对于用户自定义的route，需要在项目中加入/game-server/config/dictionary.json文件，在其中定义需要压缩的route，系统在传输时会自动进行替换，以[lordofpomelo](https://github.com/NetEase/lordofpomelo)为例，dictionary.json的格式如下：

[

"onDropItem",

"onAttack",

"onDied",

"onMove",

"onUpgrage",

"onPickItem",

"onRevive",

"addEntities",

"onRemoveEntities",

"onPathCheckout"

]

关于route压缩的更多内容，见以[Pomelo 压缩协议](https://github.com/NetEase/pomelo/wiki/Pomelo-%E6%95%B0%E6%8D%AE%E5%8E%8B%E7%BC%A9%E5%8D%8F%E8%AE%AE)。

#### protobuf编码配置

protobuf消息内容的编码格式，与之前的json相比，可以大大的减少数据的传输量。在lordofpomelo的测试中，编码后的消息大小只有json格式的20%左右。 pomelo中的porotobuf编码与之前是完全兼容的，可以在不修改代码的情况下直接开启这一功能，只需要在app.js中配置开启并加入消息的proto文件即可，app.js中的配置如下：

app.set('connectorConfig', {

connector: pomelo.connectors.hybridconnector,

heartbeat: 3,

useProtobuf: true,

handshake: function(msg, cb){

cb(null, {/\* message pass to client in handshake phase \*/});

}

});

其中的“useProtobuf：true”选项就会开启protobuf编码功能。 要对某个消息进行编码，只需要在pomelo指定文件中加入对应消息的protobuf编码就可以了，默认的protobuf编码文件位置是 /game-sever/config/serverProtos.json 和 /game-server/config/clientProtos.json，分别表示服务端->客户端的消息和客户端->服务端的消息，以[lordofpomelo](https://github.com/NetEase/lordofpomelo)为例，其内容格式如下：

"onMove" : {

"required uInt32 entityId" : 1,

"message Path": {

"required uInt32 x" : 1,

"required uInt32 y" : 2

},

"repeated Path path" : 2,

"required uInt32 speed" : 3

},

"onAttack" : {

"required uInt32 attacker" : 1,

"required uInt32 target" : 2

}

关于protobuf压缩的更多内容，见以[Pomelo 压缩协议](https://github.com/NetEase/pomelo/wiki/Pomelo-%E6%95%B0%E6%8D%AE%E5%8E%8B%E7%BC%A9%E5%8D%8F%E8%AE%AE)。

#### web客户端build配置

客户端js代码目前采用以[component](https://github.com/component/component)的形式统一管理与维护。如果升级客户端代码则需要进行简单的配置，比如在lordofpomelo的web-server下面的js库代码就只剩下，component.json，local文件夹，以及colorbox文件夹  
其中，component.json包含了指向local这个文件夹所指定的本地component，用于在component build的时候作为初始化操作  
而local文件夹里面有一个boot component，里面对lordofpomelo需要用到的客户端js库进行了配置，配置类似于npm的配置方式

{

"name": "boot",

"description": "Main app boot component",

"dependencies": {

"component/emitter":"master",

"NetEase/pomelo-protocol": "0.3.x",

"pomelonode/pomelo-protobuf": "\*",

"pomelonode/pomelo-jsclient-websocket": "master",

"component/jquery": "\*"

},

"scripts": ["index.js"]

}

其中index.js 则是对component的具体使用开发逻辑，在lordofpomelo中，我们只是把各个依赖的库给挂到windows作用域下面，以兼容老的代码  
如果你是新开发的话，完全可以利用component的规范采用类似node.js的开发方式来开发客户端js代码

最后使用component之前，需要进行build，比如在lordofpomelo中，为了便于开发，在web-server目录的bin文件夹下面，有一个 component.sh web-server目录下执行命令，即可完成install和build客户端代码的工作

sh bin/component.sh

如果你只是修改了本地的component而不需要install线上的版本，那么你可以在web-server目录下执行命令

sh bin/build-component.sh

即可对本地的component进行build操作，而不去线上install

最后你只需要把build完成之后的 build.js 引入到你的前端页面中即可

具体详情还请到 [component](https://github.com/component/component) 上去了解

# Pomelo 0.3新特性

### Pomelo 0.3版本新特性

Pomelo 0.3版为移动端性能优化做了很多工作， 新协议的数据包压缩后的传输量仅为0.2版的20%， 并保留了0.2版基于socket.io的传输协议。socket.io对开发浏览器端器端的实时应用非常适合，而socket（websocket）、protobuf、二进制等协议则对移动端、桌面客户端的开发更具优势。pomelo对它们的同时支持使同时支持浏览器、移动、桌面客户端的高实时应用或游戏变得非常容易。

服务器的动态扩展是另一个重要的特性， 这不仅使系统适应了弹性的工作环境，也为游戏的一些动态功能提供了更多方便， 如MMORPG游戏的动态副本等。

0.3版还提供了其它很多特性，如新的广播接口， 新的客户端支持等。

#### 1 新协议支持

0.3版Pomelo开始支持二进制协议，并支持对请求route的字典压缩和请求内容进行protobuf压缩。0.3版同时兼容以前版本基于socket.io的通讯协议。通过在应用中配置不同的connector component来实现协议的切换或共存。

目前Pomelo服务器提供两类connector：sioconnector和hybridconnector，分别对于基于socket.io和二进制的通讯。

##### 1.1 sioconnector

支持基于socket.io的通讯协议，也是Pomelo框架默认采用的connector（主要是兼容老版本）。之前基于socket.io的服务器和客户端代码不用修改就可以使用。

##### 1.2 hybridconnector

支持socket和websocket，使用二进制通讯协议，并且支持route字典压缩和protobuf压缩的connector，需要在app.js中显式配置。以下是一个hybridconnector的配置例子：

app.configure('production|development', 'connector', function(){

app.set('connectorConfig', {

connector: pomelo.connectors.hybridconnector,

heartbeat: 3,

useDict: true,

useProtobuf: true,

checkClient: function(type, version) {

// check the client type and version then return true or false

},

handshake: function(msg, cb){

cb(null, {/\* message pass to client in handshake phase \*/});

}

});

});

首先，通过app.set('connectorConfig', opts)更改了connector组件的默认配置。opts是最终将被传递给connector，不同的connector可以有不同的opts内容。

* connector - 指定所采用的connector实现，用来支持不同的底层通讯协议。Pomelo提供了pomelo.connectors.sioconnector和pomelo.connectors.hybridconnector。开发者也可以实现自己的connector，以支持不同的通讯协议。
* heartbeat - 客户端与服务器的心跳间隔。不指定则表示不需要心跳。具体心跳流程在协议文档心跳小结介绍。
* useDict - 是否进行route字典压缩，默认为false。具体字典压缩原理请参考协议文档字典压缩小结。
* useProtobuf - 是否对协议内容进行protobuf压缩，默认为false。protobuf的流程请参考协议文档protobuf压缩小结。
* checkClient - 可选的客户端鉴定函数。如果设置该字段，则要求客户端在握手阶段必须传递其版本号给服务器，并通过此函数来鉴定该版本客户端是否适用。如果该函数的返回值为false，则拒绝该客户端的后续操作。
* handshake - 可选的握手函数，在客户端和服务器握手过程中被调用，用来在握手阶段向客户端传递自定义的信息。

##### 1.3 不同connector的共存

不同的connector可以在同一个项目中共存，支持不同的客户端连接，只要配置不同的前端服务器，监听不同的端口即可。例如：

// frontend server based on socket.io

app.configure('production|development', 'sio-connector', function(){

app.set('connectorConfig', {

connector: pomelo.connectors.sioconnector

});

});

// frontend server based on socket and websocket

app.configure('production|development', 'hybrid-connector', function(){

app.set('connectorConfig', {

connector: pomelo.connectors.hybridconnector

});

});

servers.json的配置

{

"development": {

"sio-connector": [

{"id": "sio-server-1", "host": "127.0.0.1", "port": 3150, "clientPort": 3010, "frontend": true}

],

"hybrid-connector": [

{"id": "hybrid-server-1", "host": "127.0.0.1", "port": 3250, "clientPort": 3020, "frontend": true}

]

}

}

关于Pomelo二进制协议的具体格式，请参考[Pomelo协议文档](https://github.com/NetEase/pomelo/wiki/Pomelo-%E9%80%9A%E8%AE%AF%E5%8D%8F%E8%AE%AE)，字典压缩和protobuf压缩相关内容请参考[Pomelo数据压缩协议](https://github.com/NetEase/pomelo/wiki/Pomelo-%E6%95%B0%E6%8D%AE%E5%8E%8B%E7%BC%A9%E5%8D%8F%E8%AE%AE)。

#### 2 动态服务器扩展

Pomelo 0.3版开始支持动态增加和移除服务器进程机制，并提供相应的命令行工具。每台新增的服务器都会连接到master服务器上进行注册。Master再将新服务器的信息广播给集群中的所有服务器进程。原有的服务器进程再对新增服务器事件进行响应。

当一个服务器进程接收到一个新增服务器的消息后，会将该服务器信息保存到本地的app上下文中，之后可以通过app.getServers等系列方法查看到新服务器的信息，从而影响之后的消息路由。

如果新加的服务器的类型之前尚未存在于app上下文中，Pomelo会尝试着为其创建对应的rpc代理对象。如果该类型服务器需要提供rpc服务，则需要在约定的目录（servers/server-type/remote/）下提供rpc服务代码。

动态移除服务器进程的流程也与上面类似，当服务器断开与master的连接后，master会将该服务器的信息广播给其他进程，其他进程再进行相应处理。

注意: Pomelo仅提供动态增加和移除服务器进程的机制，但由此而导致的路由规则的变化应当由具体的应用自己来维护。需要谨慎处理好诸如数据状态等问题。

#### 命令行支持

Pomelo 0.3版本的命令行工具根据新特性增加了一个add命令，同时对之前的stop命令做了一定的修改。

pomelo add命令主要作用是动态的添加服务器，暂时不支持前端服务器的添加(0.3后面的版本可能会支持)。命令需要的参数有四个，分别是服务器地址，服务器端口号，服务器id（用来标识服务器）,服务器类型。具体的命令格式如下：

pomelo add host=[host] port=[port] id=[id] serverType=[serverType]

pomelo stop命令主要作用是动态的停止服务器，可以动态的停止某一台服务器或者同时停止多台服务器。命令主要需要的参数就是服务器id, 可以同时有多个id,这样就是停止多台服务器，如果没有id，默认就是停止所有的服务器。具体的命令格式如下：

pomelo stop [id]

#### 3 其他新特性

#### 3.1 servers.json配置文件的修改

随着Pomelo支持协议的增多，配置文件原先定义的wsPort（代表websocket port）已不适用，现调整为clientPort。并新增一个字段frontend来标识该服务器是否是前端服务器，前端服务器 必须 设置该字段为true，后端服务器可以不设置或设置成false。

以[lordofpomelo](https://github.com/NetEase/lordofpomelo)为例，[servers.js](https://github.com/NetEase/lordofpomelo/blob/master/game-server/config/servers.json)文件修改如下。

connector服务器配置：

"connector": [

{"id": "connector-server-1", "host": "127.0.0.1", "port": 3150, "clientPort": 3010, "frontend": true},

{"id": "connector-server-2", "host": "127.0.0.1", "port": 3151, "clientPort":3011, "frontend": true}

]

gate服务器配置：

"gate": [

{"id": "gate-server-1", "host": "127.0.0.1", "clientPort": 3014, "frontend": true}

]

服务器代码中有依赖到配置信息wsPort的地方也需要做相应的调整。[lordofpomelo](https://github.com/NetEase/lordofpomelo)中的[gateHandler](https://github.com/NetEase/lordofpomelo/blob/master/game-server/app/servers/gate/handler/gateHandler.js#L29)中的res.wsPort需要改为res.clientPort。

#### 3.2 新增channel广播接口

ChannelService新增broadcast接口，适用于给全世界广播的场景。该接口会将消息推送到指定类型的所有前端服务器进程，再由后者将消息推送给连接的所有客户端。详细接口描述请参考[Pomelo API](http://pomelo.netease.com/api.html)。

使用例子：

var frontendType = 'connector';

var route = 'test.hello';

var msg = {msg: 'hello world'};

var opts = {binded: true};

app.get('channelService').broadcast(frontendType, route, msg, opts, function(err) {

// check the broadcast result

});

#### 3.3 新增session获取接口

LocalSessionService新增根据session id获取localSession接口get和根据user id获取localSession接口getByUid。

使用例子：

var frontendId = 'connector-server-1';

var sid = 1;

var uid = '123456';

app.get('localSessionService').get(frontendId, sid, function(err, localSession) {

// do something with the local session

});

app.get('localSessionService').getByUid(frontendId, uid, function(err, localSession) {

// do something with the local session

});

#### 3.4 javascript客户端支持

#### 3.4.1 支持两类javascript客户端 --- socket.io与websocket

目前socket.io与websocket两套协议在javascript都有用武之地。socket.io的兼容好，可以适应各种浏览器，适合开发类似聊天室这样的高实时应用。websocket客户端则在数据压缩上做到了很多优化，大大减少了消息的传输量，适合开发基于HTML 5的游戏应用。

两个客户端的发布地址如下：

* [websocket客户端](https://github.com/pomelonode/pomelo-jsclient-websocket)
* [socket.io客户端](https://github.com/pomelonode/pomelo-jsclient-socket.io)

#### 3.4.2 客户端javscript build系统

以前的pomelo-jsclient版本管理异常混乱， 经常将好几个文件复制到不同的项目各自修改。因此我们引入了[component](https://github.com/component/component)来管理js的库。 客户端js代码目前采用以[component](https://github.com/component/component)的形式统一管理与维护。 因此在lordofpomelo的web-server下面的js库代码就只剩下，component.json，local文件夹，以及colorbox文件夹  
其中，component.json包含了指向local这个文件夹所指定的本地component，用于在component build的时候作为初始化操作  
而local文件夹里面有一个boot component，里面对lordofpomelo需要用到的客户端js库进行了配置，配置类似于npm的配置方式

{

"name": "boot",

"description": "Main app boot component",

"dependencies": {

"component/emitter":"master",

"NetEase/pomelo-protocol": "0.3.x",

"pomelonode/pomelo-protobuf": "\*",

"pomelonode/pomelo-jsclient-websocket": "master",

"component/jquery": "\*"

},

"scripts": ["index.js"]

}

使用component之前，需要进行build，为了便于开发，在web-server目录的bin文件夹下面，有一个 component.sh web-server目录下执行命令，即可完成install和build客户端代码的工作

sh bin/component.sh

如果你只是修改了本地的component而不需要install线上的版本，那么你可以在web-server目录下执行命令

sh bin/build-component.sh

即可对本地的component进行build操作，而不去线上install

具体详情还请到 [component](https://github.com/component/component) 上去了解

#### 3.5 其它客户端支持

* 提供新的C客户端，支持socket协议，基于libuv开发，提供了完整的数据与消息压缩， 支持cocos2d-x
* 其余客户端，如flash,android, ios, unity3d， 目前还只支持socket.io协议，后续会推出基于socket协议客户端

# Pomelo0.4新特性

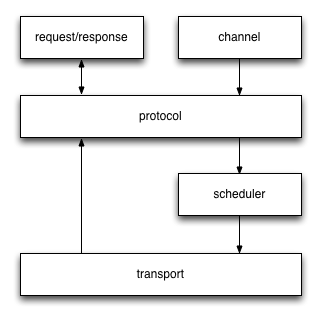
#### Pomelo0.4新特性

在0.4版本中，主要对Pomelo内部结构进行了调整，提供更友好的自定义通讯协议支持，以及自定义的数据发送包调度策略，为各种应用场景提供更灵活的支持。

#### 自定义协议

#### 服务器端实现

根据网友的建议，Pomelo框架对协议模块做了更细致的划分，以支持更灵活的定制特性。主要结构如下图所示：



在上层模块和底层的transport模块之间加入protocol层对协议数据包进行编解码。Protocol层主要针对上层通讯数据进行编解码，并与具体的connector实现相关联。对于hybridconnector，编解码的层次对应于Message层的协议数据。对于sioconnector，编解码的层次则对应于socket.io协议之上的数据。

Protocol层的编解码算法可以通过向connector组件传递encode/decode初始化参数来定制。具体例子如下：

var encode = function(reqId, route, msg) {

// do some customized encode with reqId, route and msg

return result; // return encode result

};

var decode = function(msg) {

// do some customized decode with msg

return result; // return decode result

};

app.configure('production|development', 'connector', function(){

app.set('connectorConfig', {

connector : pomelo.connectors.hybridconnector,

encode: encode,

decode: decode

});

});

每当connector上有数据包需要编解码即会触发对应的encode/decode函数。

* encode(reqId, route, msg) - 负责将reqId, route和msg进行编码。可以通过reqId对判断协议包的类型，如果reqId > 0，表示是request消息，如果reqId === 0，表示是push消息。返回值即编码结果，将直接投递给底层传输模块发送出去。
* decode(msg) - 负责将底层获取到的数据包解码成上层的消息。msg为底层传输模块获取到的数据，与客户端相同层次所传输的数据一致。返回结果为解码后的结果，是上层框架所能处理的消息，格式为：{id: id, route: route, body: body}。其中id为消息的id，request类型的消息id为正整数，notify消息可以没有id字段。route为String类型，消息的route字段；body为Object类型，即消息体。

可以通过 pomelo.connectors.hybridconnector.encode/decode 和pomelo.connectors.sioconnector.encode/decode 获取对应connector的默认编解码方法。

#### JS客户端实现

与服务器端类似，Pomelo客户端也提供了相应的定制参数，具体例子如下：

var encode = function(reqId, route, msg) {

// do some customized encode with reqId, route and msg

return result; // return encode result

};

var decode = function(msg) {

// do some customized decode with msg

return result; // return decode result

};

pomelo.init({

host: host,

port: port,

log: true,

encode: encode,

decode: decode}, function() {

});

#### 数据包发送的调度

从上面的图可以看到，消息经过protocol层编码后，还需要经过一个scheduler层才能到达最下面的transport层。

scheduler层负责决定消息发送的调度策略，比如：消息的发送优先级，广播时的分批发送等。

var SchedulerService = function() {

};

SchedulerService.prototype.schedule = function(route, msg, recvs, opts, cb) {

// do the schedule

};

app.configure('production|development', 'connector', function(){

app.set('schedulerConfig', {

scheduler: new SchedulerService()

});

});

schedule(route, msg, recvs, opts, cb)负责实现具体的调度逻辑。在schedule函数内部可以调用sessionService.sendMessage将消息发送出去。

* route - 消息的route字符串，response消息route参数为空。
* msg - 编码后的消息内容。
* recvs - 接收者的session id列表。
* opts - 附加参数。opts.isBroadcast表示当前消息是否是广播，opts.isResponse表示当前消息是否是response，opts.isPush表示当前消息是push消息。

Pomelo框架默认提供两类scheduler。

* pomelo.schedulers.direct - 默认采用的scheduler，直接将上层下来的消息发送出去。
* pomelo.schedulers.buffer - 对上层的消息进行缓存，并定期批量发送出去。可以通过flushInterval初始化参数设置刷新间隔，单位为毫秒，默认值为20毫秒。

#### 支持同一账号多处登录

0.4版本之前同一个账号在同一个前端服务器中只能存在一个session。0.4之后支持同一个账号多地登录，以支持在web应用中多个页面登录应用的情景。

该特性会影响到下列接口：

* localSessionService.getByUid - 语义改为获取指定uid下所有的session信息，返回的session改为session数组。
* localSessionService.kickByUid - 语义改为踢对应uid所有的session下线。

以及sessionServie中的对应私有方法：

* sessionService.getByUid - 语义改为获取指定uid下所有的session信息，返回的session改为session数组。
* sessionService.kickByUid - 语义改为踢对应uid所有的session下线。

#### HybridConnector添加心跳超时断开连接选项

0.3版本中，hybridconnector服务器端检测到心跳超时不会主动断开连接。在0.4.2版本中，添加disconnectOnTimeout选项，当服务器检测到心跳超时后会主动断开该客户端连接。使用的例子如下：

app.set('connectorConfig', {

connector : pomelo.connectors.hybridconnector,

heartbeat : 3,

disconnectOnTimeout: true

});

# Pomelo 0.5新特性

#### Pomelo0.5新特性

在0.5版本中，主要增强了Pomelo高可用的特性，包括master服务器的高可用和其它服务器的可配置自动重启，另外还提供一个全局的globalChannel和服务器进程与cpu绑定的功能。

#### master 高可用

Pomelo 虽然是分布式服务器框架，但是作为全局控制器的master节点却是唯一的。为了提高整体服务的可用性，我们0.5版中引入了基于Zookeeper的master高可用功能。 Master高可用是通过Zookeepr 实现的，因此如果要使用这一功能，需要第三方Zookeeper服务的支持。而Master高可用依赖于Zookpeer服务，因此建议作为第三方服务的Zookeeper本身要实现高可用。

#### 实现原理

master高可用采用多机热备技术，采用主Master+多个备份Master节点的方式实现高可用。当主master宕机时，某一个从master会自动接管主master的工作，并与其他服务器建立连接，继续提供服务，这一过程全部是自动进行的。

#### 使用说明

要开启Master高可用，只需要在app.js中加入以下配置：

app.configure('production|development', function(){

app.enable('masterHA');

app.set('masterHAConfig',

{

server : '127.0.0.1:2181',

path : '/pomelo/master'

});

});

app.enable('masterHA')表示开启master ha功能，这时服务器会加载zookeeper客户端，并尝试连接zookeeper默认端口“127.0.0.1:2181"。用户可以通过app.set('masterHAConfig'来加载自定义zookpeeper配置。server表示zookeeper服务器，path表示使用的zookeeper节点。该节点必须是zookeeper上已经存在并可以正常访问的节点。

现在有两种方式启动从Master节点：

* 第一种方式是在加入高可用配置后修改/config/master.json，采用与主master不同的Ip和端口，然后采用正常方式启动。这时，如果有主master启动的话，Pomelo会自动识别出主Master的存在，在启动时只会启动一个master备份节点，而不会重复启动其他服务。
* 第二种方式是使用命令行工具，直接启动一个独立的master节点。

#### globalChannel

globalChannel是提供全局的channel服务，其默认实现是通过redis将相关信息存储，开发者可以根据自身需求开发其它实现;Pomelo原有的channelService只能在具体某个服务器中创建channel，这种channel只能存储该服务器的用户信息，而globalChannelService则可以创建全局的globalChannel，所有服务器的用户信息都可以通过globalChannel进行存储。

#### 使用说明

globalChannel默认不加载，需要使用只需要在app.js中进行配置即可,参考配置如下(开启redis-server)。

app.configure('production|development', function(){

app.set('globalChannelConfig',

{

host: '127.0.0.1',

port: 6379,

//optional

channelManager: mysqlManager

});

});

var mysqlManager = function() {

// necessary methods (refer to redisGlobalChannelManager.js)

}

需要使用只需要从application中获取，即app.get('globalChannelService')；如果需要自己配置manager的实现，只需要在globalChannelConfig中配置channelManager,并参考Pomelo默认的redis的实现完成相应的方法就可以配置自己的globalChannel;具体的接口可以参考[Pomelo的API说明文档](http://pomelo.netease.com/api.html)。

#### 服务器自动重启

根据网友的需求，在Pomelo0.5版本中增加了服务器（非master）自动重启的功能，服务器的自动重启是以服务器与master服务器的连接状态为判断依据，即当服务器与master服务器断开后触发该服务器的重新启动。建议只对无状态的服务器配置自动重启，这样能够保证服务器重启后不影响原系统的运行。默认情况下服务器不会自动重启，如果需要开启自动重启功能需要在servers.json中进行配置auto-restart，具体配置如下：

{

"development":{

"connector":[

{"id":"connector-server-1", "host":"127.0.0.1", "port":4050, "clientPort": 3050, "frontend": true}

]

"chat":[

{"id":"chat-server-1", "host":"127.0.0.1", "port":6050, "auto-restart": true}

]

"gate":[

{"id": "gate-server-1", "host": "127.0.0.1", "clientPort": 3014, "frontend": true}

]

}

}

#### 服务器绑定CPU

为了更加充分的利用服务器的CPU，Pomelo在0.5版本中增加了服务器进程与指定CPU进行绑定，该功能限于linux系统的多核服务器，如果需要将服务器与具体CPU进行绑定，只需要在servers.json中进行配置，具体配置如下：

{

"development":{

"connector":[

{"id":"connector-server-1", "host":"127.0.0.1", "port":4050, "clientPort": 3050, "frontend": true, "cpu": 2}

]

"chat":[

{"id":"chat-server-1", "host":"127.0.0.1", "port":6050, "cpu": 1}

]

"gate":[

{"id": "gate-server-1", "host": "127.0.0.1", "clientPort": 3014, "frontend": true, "cpu": 3}

]

}

}

#### 自定义服务器关闭前事件

为了让开发者能够在服务器关闭前自定义处理事件，在Pomelo0.5版本中增加了这种服务器生命周期的自定义事件。开发者可以通过application对象的beforeStopHook方法添加自定义事件，示例代码如下：

app.configure('production|development', 'connector', function() {

var fun = function(app, cb){

//do something

cb();

}

app.beforeStopHook(fun);

});

# pomelo 0.6版新特性

在Pomelo 0.6版本中，对pomelo部分结构进行了调整，将原有的globalChannel组件和master高可用组件从框架中移出，以一种插件的形式提供给用户，具体可以参考[pomelo-globalchannel-plugin](https://github.com/NetEase/pomelo-globalchannel-plugin), [pomelo-masterha-plugin](https://github.com/NetEase/pomelo-masterha-plugin)；同时在新版本中提供了一个交互式命令行工具，相比之前的命令行工具，使用更为方便、安全，对于项目的运维会更加方便；另外为了进一步提高pomelo的安全性，在0.6版本中引入了数据的签名验证，同时对hybridconnector的非法连接进行了相关处理；其它新的特性还包括详细的rpc debug日志和前端服务器的基本过载保护功能。

#### 交互式命令行

为了方便开发者更好的对使用pomelo开发的服务进行运维，在新版本中提供了交互式命令行工具，具体情况可以参考[pomelo-cli](https://github.com/NetEase/pomelo-cli)

#### 服务器连接认证

服务器与master之间的连接需要进行认证以提高服务的安全性，目前在pomelo-admin中提供了一个简单的服务器认证，可以看 [admin auth](https://github.com/NetEase/pomelo-admin/blob/master/lib/util/utils.js#L117)  
使用连接认证，需要在 config 目录下添加 adminServer.json 的文件

[{

"type": "connector",

"token": "agarxhqb98rpajloaxn34ga8xrunpagkjwlaw3ruxnpaagl29w4rxn"

}, {

"type": "chat",

"token": "agarxhqb98rpajloaxn34ga8xrunpagkjwlaw3ruxnpaagl29w4rxn"

},{

"type": "gate",

"token": "agarxhqb98rpajloaxn34ga8xrunpagkjwlaw3ruxnpaagl29w4rxn"

}

]

**type** 是serverType, **token** 是一个字符串，你可以自己生成  
你可以通过自己定义的认证还是来完成认证的工作  
具体情况可以参考 [admin auth server](https://github.com/NetEase/pomelo-admin#server-master-auth)

#### plugin机制

为了方便开发者根据自身的需求对pomelo原有的功能进行有效的扩展，在新版本中提供了插件机制的功能，同时将之前版本中的globalChannel和master服务器高可用部分以plugin的形式提供，具体情况可以参考 [pomelo plugin](https://github.com/NetEase/pomelo/wiki/plugin%E6%96%87%E6%A1%A3)。如果要将pomelo升级到0.6，对于之前的pomelo-sync库需要重新修改在app.js中的配置，参考配置修改如下：

var sync = require('pomelo-sync-plugin');

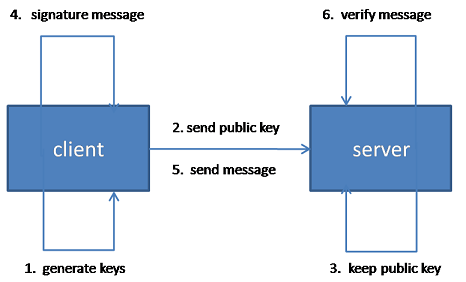
app.use(sync, {sync: {

// opts parameters passed to constructor

}});

#### 连接加密

在pomelo 0.6版本中，对hybridconnector增加了数据签名的功能。客户端首先产生rsa的密钥对，客户端保留rsa私钥，在握手阶段客户端将公钥发到服务端；在发送消息阶段，客户端使用私钥对消息进行签名，客户端将消息和签名一起发送到服务端，服务端进行签名验证，如果验证成功后面的流程继续，如果验证失败则该数据包则不进行处理。具体流程可以参考下图：



#### 使用方法

在客户端连接的过程中增加encrypt:true，在服务端app.js中配置useCrypt，具体代码参考：

客户端

pomelo.init({

host:'127.0.0.1',

port:3014,

encrypt:true

}, function() {

// do something connected

});

服务端

app.set('connectorConfig', {

connector: pomelo.connectors.hybridconnector,

heartbeat: 3,

useDict: true,

useProtobuf: true,

useCrypto: true

});

#### 非法连接处理

在pomelo中sioconnector是基于socket.io，socket.io本身是有对非socket.io的连接进行处理的。对于hybridconnector，底层是基于socket的，在pomelo 0.6版本中增加了对不符合规定协议的连接进行拒绝处理。主要包括两个部分：1.对空连接进行了超时处理；2.对不符合协议规范的连接进行拒绝处理。

#### rpc debug日志

根据网友的建议，在pomelo 0.6版本中增加了更多的rpc日志。开发者只需要在app.js中使用app.enable('rpcDebugLog')即可，另外需要在game-server/config/servers.json中配置category为rpc-debug的appender，具体配置可以参考如下代码：

{

"type": "file",

"filename": "./logs/rpc-debug-${opts:serverId}.log",

"maxLogSize": 1048576,

"layout": {

"type": "basic"

},

"backups":5,

"category": "rpc-debug"

}

#### 过载保护

在pomelo之前的版本中有toobusy模块对服务器进行过载保护，在新版本中增加了一个对connector连接数的限制功能，开发者只需要在servers.json中对不同的connector进行最大连接数量的配置，当connector超过配置的最大数量，服务器会拒绝连接。配置可以参考如下代码：

{"id":"connector-server-1", "host":"127.0.0.1", "port":4050, "clientPort":3050, "frontend":true, "max-connections": 100}

#### 服务器断网重连机制

在之前pomelo的版本中，在分布式环境下，存在服务器(非master)网络短时间断开，然后网络恢复后服务无法恢复的情况。在pomelo 0.6版本中，如果服务器(非master)短时间断开，网络恢复后服务器可以正常工作。

#### Daemon启动模式

pomelo-daemon提供对pomelo服务进行分布式环境下的启动以及rpc debug日志的收集  
详细情况请看 [pomelo-daemon](https://github.com/NetEase/pomelo-daemon)

#### 升级的logger

在新版中对pomelo中的logger进行了升级

* logger支持自定义prefix输出，把prefix打印在消息的头部，prefix可以是文件名，serverId, host 等等
* logger在debug模式下支持log行号的打印，方便开发者分析调试
* 在pomelo中的日志统一输出到了category为pomelo的appenders  
  详细情况请看 [pomelo-logger](https://github.com/NetEase/pomelo-logger)

#### pomelo-protobuf支持rootMsg的proto文件定义

通过定义rootMsg，可以很好的进行复用，简化protos文件的大小

{

"message Path": {

"required double x" : 1,

"required double y" : 2

},

"message Equipment" : {

"required uInt32 entityId" : 1,

"required uInt32 kindId" : 2

},

"onMove" : {

"required uInt32 entityId" : 1,

"repeated Path path" : 2,

"required float speed" : 3

},

"area.playerHandler.enterScene" : {

"message Player" : {

"message Bag" : {

"message Item" : {

"required uInt32 id" : 1,

"optional string type" : 2

},

"repeated Item items" : 1

},

"required uInt32 entityId" : 1,

"required uInt32 kindId" : 2,

"required Bag bag" : 3,

"repeated Equipment equipments" : 4

},

"optional Player curPlayer" : 2

}

}

以上定义的Equipment和Path都是可能复用的root message。 详细情况情况 [pomelo-protobuf](https://github.com/pomelonode/pomelo-protobuf#rootmessage-support)

# pomelo 0.7版新特性

在pomelo 0.7版本中，增加了用户自定义定时任务功能，用户能够在不同服务器中动态地增删定时任务；根据网友的建议，在0.7版中增加了全局filter的功能，用户能够在前端服务器对请求进行统一处理；另外新版中增加了事务的机制，提供了一个简单的事务方法，包括条件方法和处理方法；其它新特性还包括[pomelo-cli](https://github.com/NetEase/pomelo-cli)的命令自动补全功能。

#### 定时任务

用户能够通过配置文件或者[pomelo-cli](https://github.com/NetEase/pomelo-cli)的命令addCron和removeCron对定时任务进行动态调度。 定时任务是针对具体服务器而言，例如需要在chat服务器中配置定时任务：

首先在game-server/app/servers/chat目录下增加cron目录，在game-server/app/servers/chat/cron目录下编写具体的执行的任务的代码chatCron.js，例如：

module.exports = function(app) {

return new Cron(app);

};

var Cron = function(app) {

this.app = app;

};

var cron = Cron.prototype;

cron.sendMoney = function() {

console.log('%s server is sending money now!', this.app.serverId);

};

然后在game-server/config/目录下增加定时任务配置文件crons.json，具体配置文件如下所示：

{

"development":{

"chat":[

{"id":1, "time": "0 30 10 \* \* \*", "action": "chatCron.sendMoney"},

{"id":2, "serverId":"chat-server-1", "time": "0 30 10 \* \* \*", "action": "chatCron.sendMoney"}

]

},

"production":{

"chat":[

{"id":1, "time": "0 30 10 \* \* \*", "action": "chatCron.sendMoney"},

{"id":2, "serverId":"chat-server-1", "time": "0 30 10 \* \* \*", "action": "chatCron.sendMoney"}

]

}

}

在配置文件crons.json中，id是定时任务在具体服务器的唯一标识，且不能在同一服务器中重复；time是定时任务执行的具体时间，时间的定义跟linux的定时任务类似，一共包括7个字段，每个字段的具体定义如下：

\* \* \* \* \* \* command to be executed

- - - - - -

| | | | | |

| | | | | +----- day of week (0 - 6) (Sunday=0)

| | | | +------- month (0 - 11)

| | | +--------- day of month (1 - 31)

| | +----------- hour (0 - 23)

| +------------- min (0 - 59)

+------------- second (0 - 59)

0 30 10 \* \* \* 这就代表每天10:30执行相应任务；serverId是一个可选字段，如果有写该字段则该任务只在该服务器下执行，如果没有该字段则该定时任务在所有同类服务器中执行；action是具体执行任务方法，chatCron.sendMoney则代表执行game-server/app/servers/chat/cron/chatCron.js中的sendMoney方法。

通过[pomelo-cli](https://github.com/NetEase/pomelo-cli)的addCron和removeCron命令可以动态地增加和删除定时任务，其中addCron的必要参数包括：id,action,time；removeCron的必要参数包括：id；serverId和serverType是两者选其一即可。例如：

addCron id=8 'time=0 30 11 \* \* \*' action=chatCron.sendMoney serverId=chat-server-3

removeCron id=8

#### 全局filter

在之前pomelo的版本中，filter是在后端服务器进行消息拦截并进行相应处理；根据网友的意见，在0.7版中在前端服务器增加了filter,请求在前端服务器就可以进行统一处理。请求的处理过程由之前的：前端服务器 -> beforeFilter -> 后端服务器 -> afterFilter 变为：globalBeforeFilter -> 前端服务器 -> beforeFilter -> 后端服务器 -> afterFilter -> globalAfterFilter。同之前的filter的错误处理过程一样，全局filter的错误全部进入globalErrorHandler处理。

全局filter与以前的filter可以相互通用，具体的配置样例如下：

app.configure('production|development', function() {

app.globalFilter(pomelo.serial());

});

#### Transaction

在新版本中，pomelo提供简单的事务处理的功能；开发者可以设置相应的事务处理条件和实际事务处理的方法，同时开发还可以定义事务处理的重试次数。事务的具体执行过程是先执行开发者定义的事务处理条件，如果条件报错则直接终止整个事务，如果条件执行通过，再开始执行相应的事务处理方法，当在事务处理方法执行的过程中出现错误则根据开发者定义的重试次数进行执行重试，默认重试次数为1；所有的事务处理的结果都会在相应的日志文件中进行详细记录，开发者可以根据错误日志对失败的事务进行相应处理；相应的API如下所示：

##### transaction(name, conditions, handlers, retry)

事务处理方法

###### Arguments

* name - transaction name
* conditions - transaction conditions
* handlers - transaction handlers
* retry - retry times of handlers if error occurs in handlers

具体的使用示例如下：

var conditions = {

test1: function(cb) {

console.log('condition1');

cb();

},

test2: function(cb) {

console.log('condition2');

cb();

}

};

var handlers = {

do1: function(cb) {

console.log('handler1');

cb();

},

do2: function(cb) {

console.log('handler2');

cb();

}

};

app.transaction('test', conditions, handlers, 3);

#### pomelo-cli自动提示

在最新版本的pomelo-cli中提供命令自动补全的功能，具体可以参考[pomelo-cli](https://github.com/NetEase/pomelo-cli)。

#### pomelo 部分术语改名

在最新版本中，对pomelo中一些含混并且可能对用户造成误导的命名做了修改，这样可以使得名字含义更明确，以使得开发者能更好地理解，具体修改如下：

* LocalSession以及LocalSessionService改名为BackendSession和BackendSessionService，具体功能不变，代码中需要注意如下：

// deprecated

var localSessionService = app.get("localSessionService");

// recommanded

var backendSessionService = app.get("backendSessionService");

* MockLocalSession 修改为 FrontendSession。
* 组件scheduler改名为pushScheduler， 对pushScheduler组件进行配置选项的时候使用：

app.set("pushSchedulerConfig", opts);

* 组件proxy以及remote的配置选项 cacheMsg 本版本中改名为bufferMsg，使得含义更明确：

// deprecated

app.set("proxyConfig", {cacheMsg: true});

app.set("remoteConfig", {cacheMsg: true});

//recommended

app.set("proxyConfig", {bufferMsg: true});

app.set("remoteConfig", {bufferMsg: true});

这些改名对以前的代码保持兼容。

# pomelo 0.8新特性

#### pomelo-cocos2d-jsb

对 cocos2d-x javaScript binding 环境有了支持，开发者可以很方便的使用 javaScript 来完成前后端的开发，并发布到 ios,android,web平台上  
详细请看 [pomelo-cocos2d-jsb](https://github.com/NetEase/pomelo-cocos2d-jsb)

#### pomelo命令行重构

##### 执行pomelo命令时，去除了对特定目录的依赖。

当在非game-server目录下执行pomelo start的时候，需要通过命令行选项指出代码的位置。对于log4js配置文件来说，需要将其日志文件的根目录设置为game-server下的目录。这个目录使用${opts:base}来指定, 因此，如果在其他目录下启动应用服务器，请使用新的log4js配置，在template目录下有相应的示例，如下：

{

"appenders": [

{

"type": "console"

},

{

"type": "file",

"filename": "${opts:base}/logs/con-log-${opts:serverId}.log",

"pattern": "connector",

"maxLogSize": 1048576,

"layout": {

"type": "basic"

},

"backups": 5,

"category": "con-log"

},

{

"type": "file",

"filename": "${opts:base}/logs/crash.log",

"maxLogSize": 1048576,

"layout": {

"type": "basic"

},

"backups": 5,

"category":"crash-log"

},

// other appenders...

],

"levels": {

"rpc-log" : "ERROR",

"forward-log": "ERROR"

},

"replaceConsole": true,

"lineDebug": false

}

对于其他的pomelo子命令，如pomelo list， pomelo add， pomelo stop等命令的执行也不再依赖目录，在执行时需要指定master服务器的地址及端口号，如果不指定，默认使用127.0.0.1:3005，因此这些命令可以远程执行。当然执行这些命令的时候，需要相应的权限控制，默认的用户名和密码均为admin，用户可以根据具体需求进行修改。

对于pomelo masterha命令，同pomelo start类似，也需要指出要执行代码的位置。这些修改保持与前面版本的兼容。

##### 对pomelo start的改进

当以daemon方式启动pomelo应用的时候，去除了对第三方模块forever的依赖，此时启动的所有应用进程将脱离控制终端在后台运行，要查看所有的启动以及运行日志，只能通过相应的日志文件。

对于pomelo start的env，可以支持由用户来自由配置，使用 -e|--env 选项。

使用pomelo --help 可以获得更详细的pomelo命令行支持的选项。

#### pushScheduler功能增强

##### 对response，push增加了一个可选参数

在新版中，支持了更细粒度的pushScheduler选择。对respone，push增加了一个可选参数userOption; 对broadcast，扩展了原有的option参数的使用。用户可以把它定义为任意值，这个参数将会被用来选择具体的pushScheduler以及在schedule调用中使用，每一次的response，push以及broadcast都可以选择使用不同的pushScheduler。具体示例如下：

在Handler中，可以传入一个userOption参数，如下：

Handler.prototype.handle = function(msg, session, next) {

// do handling

// var resp = ...

// var userOption = ...

cb(null, resp, userOption);

}

对于push操作，示例如下:

var aChannel = channelService.getChannel('aChannel');

// var userOptions = ...

aChannel.pushMessage(route, msg, userOptions, cb);

对于broadcast操作，以前的版本已经支持了option配置，具体的配置项仅仅支持了binded和filterParam，新版本中对此进行了扩展，这两个选项继续有效，而且用户可以自定义更多新的选项，使用方式不变。

对于errorHandler， 以前版本的errorHandler为：

var errorHandler = function (err, msg, resp, session, cb) {

// ...

}

由于增加了userOption选项，故相应的errorHandler也多了一个参数，如下：

var errorHandler = function (err, msg, resp, session, opts, cb) {

// ...

}

这个修改对以前的版本包持兼容，建议使用新的函数签名。

##### 多个pushScheduler的配置及选择

在新版中，一个connector可以配置多个pushScheduler，并根据自定义规则selector来进行选择，response以及push新增加的可选参数userOptions可以用于selector的选择计算。下面的示例中就定义了三个不同的pushScheduler，并应用于不同的情况：

app.configure('development', 'connector', function() {

app.set('pushSchedulerConfig', {

scheduler: [

{

id: 'direct',

scheduler: pomelo.pushSchedulers.direct

},

{

id: 'buffer5',

scheduler: pomelo.pushSchedulers.buffer,

options: {flushInterval: 5000}

},

{

id: 'buffer10',

scheduler: pomelo.pushSchedulers.buffer,

options: {flushInterval: 10000}

}

],

selector: function(reqId, route, msg, recvs, opts, cb) {

// opts.userOptions is passed by response/push/broadcast

console.log('user options is: ', opts.userOptions);

if(opts.type === 'push') {

cb('buffer5');

return;

}

if (opts.type === 'response') {

cb('direct');

return ;

}

if (opts.type === 'broadcast') {

cb('buffer10');

return ;

}

}

});

新的pushScheduler配置与原有的一个connector仅仅支持一个pushScheduler的配置方式保持兼容。

#### rpc调用方式改进

当进行rpc调用的时候，增加了跳过路由计算而直接将调用发送到一个具体的服务器或者广播到一类服务器的调用方式，代码示例如下：

// route

var routeParam = session;

app.rpc.area.playerRemote.leaveTeam(routeParam, args..., cb);

// to specified server 'area-server-1'

app.rpc.area.playerRemote.leaveTeam.toServer('area-server-1', args..., cb);

// broadcast to all the area servers

app.rpc.area.playerRemote.leaveTeam.toServer('\*', args..., cb);

## 生命周期回调

在pomelo 0.8版中增加对外提供application的生命周期回调，这样能够让开发者在不同类型的服务器生命周期中进行具体操作。提供的生命周期回调函数包括：beforeStartup，afterStartup，beforeShutdown，afterStartAll。其具体的功能说明如下：

### beforeStartup(app, cb)

before application start components callback

#### Arguments

* app - application object
* cb - callback function

##### afterStartup(app, cb)

after application start components callback

#### Arguments

* app - application object
* cb - callback function

##### beforeShutdown(app, cb)

before application stop components callback

#### Arguments

* app - application object
* cb - callback function

##### afterStartAll(app)

after all applications started callback

#### Arguments

* app - application object

具体使用方法：在game-server/app/servers/某一类型服务器/ 目录下添加lifecycle.js文件，具体文件内容如下：

module.exports.beforeStartup = function(app, cb) {

// do some operations before application start up

cb();

};

module.exports.afterStartup = function(app, cb) {

// do some operations after application start up

cb();

};

module.exports.beforeShutdown = function(app, cb) {

// do some operations before application shutdown down

cb();

};

module.exports.afterStartAll = function(app) {

// do some operations after all applications start up

};

## rpc filter提供对外接口

根据网友的建议，在新版本中对外提供了添加rpc filter的接口，包括rpcBefore、rpcAfter、rpcFilter，其功能分别为添加before filter, 添加after filter，同时添加两种filter；使用方法与handler的filter类似。在早期版本中提供的enableRpcLog选项，可以通过添加rpc filter代替。

app.configure('production|development', function() {

// configurations

app.filter(pomelo.filters.time());

app.rpcFilter(pomelo.rpcFilters.rpcLog());

});

## 简化配置文件

在0.8版的pomelo中对配置文件servers.json进行了精简，通过增加clusterCount字段将原有的配置进行了简化，这样将更加适合大规模应用的部署和运维管理。

原有的配置：

"connector":[

{"id":"connector-server-1", "host":"127.0.0.1", "port":4050, "clientPort": 3050, "frontend": true},

{"id":"connector-server-2", "host":"127.0.0.1", "port":4051, "clientPort": 3051, "frontend": true},

{"id":"connector-server-3", "host":"127.0.0.1", "port":4052, "clientPort": 3052, "frontend": true}

],

"chat":[

{"id":"chat-server-1", "host":"127.0.0.1", "port":6050},

{"id":"chat-server-2", "host":"127.0.0.1", "port":6051},

{"id":"chat-server-3", "host":"127.0.0.1", "port":6052}

],

"gate":[

{"id": "gate-server-1", "host": "127.0.0.1", "clientPort": 3014, "frontend": true}

]

0.8版本pomelo支持的简化配置：

"connector":[

{"host":"127.0.0.1", "port":"4050++", "clientPort": "3050++", "frontend": true, "clusterCount": 3}

],

"chat":[

{"host":"127.0.0.1", "port":"6050++", "clusterCount": 3}

],

"gate":[

{"host": "127.0.0.1", "clientPort": 3014, "frontend": true, "clusterCount": 1}

]

同时在采用pomelo-cli进行动态增加服务器的时候，同样可以使用add host=127.0.0.1 port=9000++ serverType=chat clusterCount=3 这样的形式同时增加多台服务器。

## pomelo-logger支持动态日志级别

在[pomelo-logger](https://github.com/NetEase/pomelo-logger)0.1.2中，增加了动态改变日志级别的功能，开发者可以在原有的config/log4js.json中配置reloadSecs参数，该参数表示定期检查配置文件是否有更新，如果有更新则重新装载并根据配置文件，更改日志级别。log4js.json配置如下：

{

......

"levels": {

"pomelo" : "INFO",

"rpc-log" : "INFO",

"forward-log": "ERROR",

"con-log": "INFO"

},

"replaceConsole": true,

"reloadSecs": 60 \* 3

}

该配置表示每3分钟检查一次配置文件是否有更新。

## 安全性方面

### RPC调用的IP白名单

该功能可以为各个服务器的RPC调用提供IP白名单过滤功能.

###### 原理

RPC服务端每接受一个连接都会抛出一个连接事件, 这个事件中含有该连接的socket.id和RPC客户端IP. RPC服务端会捕获该连接事件, 并调用用户传入的获取IP白名单的函数, 如果该RPC客户端IP不在白名单中, 则立刻将对应的socket断开. 以此来实现RPC调用白名单过滤功能.

###### 使用

使用时只需要向remoteConfig的配置中传入一个获取IP白名单的函数(whitelist: rpcWhitelist.whitelistFunc)即可, 这个函数需要接受一个回调函数作为其参数, 该回调函数形如function(err, tmpList) {...}. 在获取IP白名单的函数内, 拿到IP白名单时(该白名单应为一维JS Array), 以类似于cb(null, self.gWhitelist)的形式调用IP过滤回调函数.

./game-server/app/util/whitelist.js

... ...

var self = this;

self.gWhitelist = ['192.168.146.100', '192.168.146.101'];

module.exports.whitelistFunc = function(cb) {

cb(null, self.gWhitelist);

};

... ...

./game-server/app.js

var rpcWhitelist = require('./app/util/whitelist');

... ...

// configure for global

app.configure('production|development', function() {

... ...

// remote configures

app.set('remoteConfig', {

cacheMsg: true

, interval: 30

, whitelist: rpcWhitelist.whitelistFunc

});

... ...

}

* 具体请参考[lordofpomelo](https://github.com/NetEase/lordofpomelo/tree/dev)

### pomelo-admin的IP白名单

该功能可以为master服务器的admin提供IP白名单过滤功能.

###### 原理

admin服务端每接受一个连接都会抛出一个连接事件, 这个事件中含有该连接的socket.id和admin客户端IP. admin服务端会捕获该连接事件, 并调用用户传入的获取IP白名单的函数, 如果该admin客户端IP不在白名单中, 则立刻将对应的socket断开. 以此来实现master服务器的admin白名单过滤功能.

###### 使用

使用时只需要向masterConfig的配置中传入一个获取IP白名单的函数(whitelist: adminWhitelist.whitelistFunc)即可, 这个函数需要接受一个回调函数作为其参数, 该回调函数形如function(err, tmpList) {...}. 在获取IP白名单的函数内, 拿到IP白名单时(该白名单应为一维JS Array), 以类似于cb(null, self.gWhitelist)的形式调用IP过滤回调函数.

./game-server/app/util/whitelist.js

... ...

var self = this;

self.gWhitelist = ['192.168.146.100', '192.168.146.101'];

module.exports.whitelistFunc = function(cb) {

cb(null, self.gWhitelist);

};

... ...

./game-server/app.js

var adminWhitelist = require('./app/util/whitelist');

... ...

// configure for global

app.configure('production|development', function() {

... ...

app.set('masterConfig', {

authUser: app.get('adminAuthUser') // auth client function

, authServer: app.get('adminAuthServerMaster') // auth server function

, whitelist: adminWhitelist.whitelistFunc

});

... ...

}

* 具体请参考[lordofpomelo](https://github.com/NetEase/lordofpomelo/tree/dev)

## csv配置文件自动热加载插件pomelo-data-plugin

该插件可以监控指定目录下的所有csv格式的配置文件, 并在某个文件被改变时自动地将其热加载进入Pomelo.

#### 原理

该插件主要使用[csv模块](https://npmjs.org/package/csv)来解析csv配置文件, 使用fs.watchFile函数来监控文件变化事件. 当Pomelo框架启动该插件中的组件时, 组件会加载给定文件夹中的所有csv配置文件, 并为每个文件加一个watcher. 以此来实现csv配置文件自动热加载的功能.

#### 安装

npm install pomelo-data-plugin

#### 使用

./pomelo-data-plugin-demo/config/data/team.csv

# 队伍名称配置表

# 队名编号, 队名

id,teamName

5,The Lord of the Rings

6,The Fast and the Furious

上面的csv配置文件中, 以#开头的行是注释语句, 正文的第一行应为列名(id为该文件的主键列名, 为必须列), 下面的行是对应列名的具体数据.

var dataPlugin = require('pomelo-data-plugin');

... ...

app.configure('production|development', function() {

...

app.use(dataPlugin, {

watcher: {

dir: \_\_dirname + '/config/data',

idx: 'id',

interval: 3000

}

});

...

});

... ...

... ...

var teamConf = app.get('dataService').get('team');

... ...

... ...

上面代码中的dir即为需要监控的配置文件夹; idx为所有csv配置文件的主键列名(如:team.csv所示的id); interval为fs.watchFile函数测试其所监控文件改变的时间间隔, 单位为毫秒.

* 具体请参考[pomelo-data-plugin](https://github.com/NetEase/pomelo-data-plugin)和[pomelo-data-plugin-demo](https://github.com/NetEase/pomelo-data-plugin-demo)

# pomelo 0.9新特性

## pomelo rpc支持zeromq通信

在pomelo 0.9中提供了基于zmq的rpc调用，开发者可以根据需要选择原有的pomelo-rpc或者pomelo-rpc-zeromq。基于zeromq和原有的pomelo-rpc的性能对比测试结果可以参考：

具体使用方法：

1. 安装zeromq
2. 在app.js中进行配置，具体配置如下所示：

var zmq = require('pomelo-rpc-zeromq');

app.configure('production|development', function() {

app.set('proxyConfig', {

rpcClient: zmq.client

});

app.set('remoteConfig', {

rpcServer: zmq.server

});

});

具体使用示例可以参考 [chatofpomelo](https://github.com/NetEase/chatofpomelo/tree/zmq) zmq分支

[pomelo-rpc-zeromq性能测试报告](https://github.com/NetEase/pomelo/wiki/pomelo-rpc-zeromq%E6%80%A7%E8%83%BD%E6%B5%8B%E8%AF%95%E6%8A%A5%E5%91%8A)与原有的[pomelo-rpc的性能测试报告](https://github.com/NetEase/pomelo/wiki/pomelo-rpc%E6%80%A7%E8%83%BD%E6%B5%8B%E8%AF%95%E6%8A%A5%E5%91%8A)可以参考。

## pomelo命令行改进

根据网友的建议，在pomelo 0.9版本中增加了重启的命令和服务器单独启动的命令；具体命令如下：

pomelo start -t [server type] 启动某类型的服务器，如果需要分开启动不同类型服务器时候使用，首先必须启动master服务器，例如：

pomelo start -t master

pomelo start -t connector

pomelo start -i [server id] 启动具体某个服务器，同上首先需要启动master，master服务器无需提供具体id, 例如：

pomelo start -i master

pomelo start -i chat-server-1

pomelo restart 重启除了master以为的其它服务器，可能会出现重启失败的情况。

pomelo restart -t [server type] 重启某类型的服务器，不包括master服务器

pomelo restart -i [server id] 重启具体某个服务器，不包括master服务器

pomelo restart -t connector

pomelo restart -i chat-server-1

## pomelo rpc增加回调超时机制

针对之前网友提出的[rpc回调函数积压问题](http://nodejs.netease.com/topic/52b2d8470a516e1851b256e4), 在新版本的pomelo-rpc中通过增加了回调函数的超时机制解决，rpc客户端在记录应用层的回调函数的同时添加对应的定时器，如果rpc服务端收到对应的消息则将定时器清除，如果定时器超时则将对应的回调函数清除。定时器的超时时间开发者可以进行设置，默认是10s， 具体使用如下：

app.set('proxyConfig', {

timeout: 1000 \* 20

});

## pomelo支持连接黑名单机制

在新版本的pomelo中连接服务器支持静态或者动态添加ip黑名单功能，服务端可以在连接服务器中增加黑名单对攻击的ip进行屏蔽。

### 原理

connector每接受一个连接都会抛出一个连接事件, 这个事件中含有该连接的客户端IP. connector会捕获该连接事件, 并调用用户传入的获取IP黑名单的函数, 如果该客户端IP在黑名单中, 则立刻将对应的socket断开. 以此来实现连接服务器的黑名单过滤功能.

### 使用方法

#### 静态添加黑名单

使用时只需要向在connector的connectionConfig配置中传入一个获取IP黑名单的函数即可, 这个函数需要接受一个回调函数作为其参数, 该回调函数形如function(err, list) {...}. 在获取IP黑名单的函数内, 拿到IP黑名单时(该黑名单应为一维JS Array), 以类似于cb(null, self.list)的形式调用IP过滤回调函数，具体使用方法如下：

./game-server/app/util/blackList.js

... ...

var self = this;

self.blackList = ['192.168.100.1', '192.168.100.2'];

module.exports.blackListFun = function(cb) {

cb(null, self.blackList);

};

... ...

./game-server/app.js

var blackList = require('./app/util/blackList');

... ...

app.configure('production|development', function() {

... ...

app.set('connectorConfig', {

blacklistFun: blackList.blackListFun

});

... ...

}

#### 动态添加黑名单

动态添加黑名单可以通过pomelo-cli完成，其中运行输入具体ip或者正则表达式，具体命令如下：

blacklist 192.168.100.1

blacklist (([01]?d?d|2[0-4]d|25[0-5]).){3}([01]?d?d|2[0-4]d|25[0-5])

## channel 序列化接口

在新版本的pomelo中提供channel的序列化接口，开发者可以通过实现该接口将系统中创建的channel进行保存；同时当服务器重新启动后，系统会将之前保存的channel恢复到系统中。开发者需要实现如下四个接口：

#### add(key, value, cb)

add key value pairs

#### remove(key, value, cb)

remove key value pairs

#### load(key, cb)

load all values

#### removeAll(key, cb)

remove all values

具体的使用方法如下所示：

var store = require('./store');

app.set('channelConfig', {

store : store,

prefix : 'pomelo'

});

//store.js

var redis = require('redis');

var StoreManager = function() {

this.redis = redis.createClient(6379, '127.0.0.1', {});

};

module.exports = new StoreManager();

StoreManager.prototype.add = function(key, value, cb) {

this.redis.sadd(key, value, function(err) {

cb(err);

});

};

StoreManager.prototype.remove = function(key, value, cb) {

this.redis.srem(key, value, function(err) {

cb(err);

});

};

StoreManager.prototype.load = function(key, cb) {

this.redis.smembers(key, function(err, list) {

cb(err, list);

});

};

StoreManager.prototype.removeAll = function(key, cb) {

this.redis.del(key, function(err) {

cb(err);

});

};

## 热重启的部分支持

在新版本的pomelo中对rpc模块进行了改进，在rpc服务端断开连接后，上层应用的rpc请求会在rpc客户端缓存；当rpc服务端恢复后，再次发起rpc请求时，会把之前的rpc请求一起发到rpc服务端。

在系统中如果是rpc单向依赖，也就是说系统中只有A类服务器发送rpc请求到B类服务器，没有B类服务器发送rpc请求到A类服务器，同时B类服务器是没有任何状态信息和实例化信息，这样B类服务器就可以在pomelo 0.9版本中重启，且不会影响系统的正常运行。

具体可以参考[chatofpomelo](https://github.com/NetEase/chatofpomelo) store分支。在chatofpomelo中只有connector到chat服务器的单向rpc依赖，对于chat服务器有channel的实例存在，所以使用channel序列化接口将channel存储，所以需要使用到redis。

操作步骤如下：

1. 在不同的命令行界面分别执行 pomelo start -t master， pomelo start -t connector, pomelo start -t gate, pomelo start -t chat;
2. 打开web服务器，运行chatofpomelo;
3. 关闭chat服务器，并重新启动；

## pomelo支持decodeIO protobuf

在pomelo 0.9版本中提供了对decodeIO的protobuf的支持，对于decodeIO的protobuf的介绍可以参考[decodeIO-protobufjs](https://github.com/dcodeIO/ProtoBuf.js).

### 使用方法

#### 客户端

使用最新的pomelo-jsclient-websocket, 同时在客户端添加命名为pomelo-decodeIO-protobuf的component，并将其挂载到window对象下。对应的component.js如下所示：

{

"name": "boot",

"description": "Main app boot component",

"dependencies": {

"component/emitter":"master",

"NetEase/pomelo-protocol": "master",

"pomelonode/pomelo-decodeIO-protobuf": "master",

"pomelonode/pomelo-jsclient-websocket": "master",

"component/jquery": "\*"

},

"scripts": ["index.js"]

}

对应的index.js如下所示：

var Emitter = require('emitter');

window.EventEmitter = Emitter;

var protocol = require('pomelo-protocol');

window.Protocol = protocol;

var protobuf = require('pomelo-decodeIO-protobuf');

window.decodeIO\_protobuf = protobuf;

var pomelo = require('pomelo-jsclient-websocket');

window.pomelo = pomelo;

var jquery = require('jquery');

window.$ = jquery;

#### 服务端

在服务端需要使用pomelo-protobuf-plugin，并在app.js中使用对应的插件，具体配置如下：

app.configure('production|development', function() {

app.use(protobuf, {

protobuf: {

}

});

});

#### 注意事项

pomelo原有的protobuf和decodeIO-protobufjs不能同时使用，即不能同时使用pomelo-protobuf-plugin插件并在前端服务器开启useProtobuf。

考虑到与原有的protobuf保持一致，pomelo 0.9版本中支持的decodeIO-protobuf同样采用serverProtos.json和clientProtos.json，不支持decodeIO-protobufjs中的.proto格式，对于.proto格式可以采用decodeIO-protobufjs提供的命令行工具转换成json格式。

具体的使用示例可以参考[lordofpomelo](https://github.com/NetEase/lordofpomelo/tree/decodeIO_protobuf) decodeIO-protobuf分支。

## pomelo websocket支持自动重连

在pomelo 0.9版本中，pomelo-jsclient-websocket 支持自动重连。重连发生在连接断开后的5s后，在重连失败后下次重连的时间将是上次重连时间的2倍；所以重连时间依次为5s, 10s，20s,依次类推。默认最大的重连次数是10次，该参数可以在连接初始化过程中进行配置。

//设置客户端重连

pomelo.init({

host: 127.0.0.1,

port: 3050,

reconnect: true

}, function() {

});

//设置客户端重连最大次数

pomelo.init({

host: 127.0.0.1,

port: 3050,

reconnect: true，

maxReconnectAttempts： 20

}, function() {

});

# pomelo 1.0新特性

## pomelo udpconnector提供

根据网友的要求，在pomelo 1.0中提供了udpconnector。在该udpconnector中，采用了pomelo之前hybridconnector提供的传输协议，包括握手、心跳及数据包的格式。该connector中通过客户端的ip和port来对客户端进行唯一标识，客户端的断开则是通过心跳超时进行判定。

使用方法：

// app configuration

app.configure('production|development', 'connector', function() {

app.set('connectorConfig',

{

connector : pomelo.connectors.udpconnector,

heartbeat : 3

});

});

由于pomelo中提供的客户端还没有支持udp的，所以在1.0.0中提供一个node版本的udp client，配合通过pomelo init出来的demo使用，具体可以参考[udpclient](https://github.com/py8765/udpclient/tree/master)

## pomelo-rpc 负载均衡及容错机制

在pomelo 1.0版本中，pomelo-rpc提供了相关的负载均衡算法，开发者在调用rpc时，可以选择相应的路由算法，rpc框架会根据不同的配置在rpc客户端进行路由算法的选择，从而完成相应的rpc调用。提供的负载均衡算法包括：rr（ROUNDROBIN）, wrr(WEIGHT\_ROUNDROBIN), la(LEAST\_ACTIVE), ch(CONSISTENT\_HASH)。

配置方法：

// rr

app.configure('production|development', 'connector', function() {

app.set('proxyConfig',

{

routerType: 'rr'

});

});

//基于权重的rr，在servers.json中对rpc目标服务器进行权重配置

app.configure('production|development', 'connector', function() {

app.set('proxyConfig',

{

routerType: 'wrr'

});

});

"read": [

{"id": "read-server-1", "host": "127.0.0.1", "port": 4150, "weight": 1},

{"id": "read-server-2", "host": "127.0.0.1", "port": 4151, "weight": 5},

{"id": "read-server-3", "host": "127.0.0.1", "port": 4152, "weight": 8}

]

//la

app.configure('production|development', 'connector', function() {

app.set('proxyConfig',

{

routerType: 'la'

});

});

//consistent\_hash

app.configure('production|development', 'connector', function() {

app.set('proxyConfig',

{

routerType: 'ch',

replicas: '100', //虚拟节点数量

algorithm: 'md5', //hash算法

hashFieldIndex: 0 //根据rpc参数列表中的具体参数进行hash

});

});

ps:　如果使用toServer('chat-server-1')这种指定rpc目标服务器的rpc调用则不会使用相关的负载均衡算法，如果是指定了routerType则之前在application对象中设置的route函数则无效；综合这三种方式的优先级是toServer > 指定routerType > 指定路由函数。

在新版本中,pomelo-rpc模块提供了相关的容错机制，包括failover，即失败自动切换，当出现失败，重试其它同类型服务器，这种模式通常用于读操作，但重试可能会带来更长延迟；failfast是快速失败，其策略为只发起一次rpc调用，失败后就立即将错误信息返回；failsafe则是一种安全策略，也是rpc默认采用的，即根据rpc的不同类型错误进行不同的处理策略，主要是发起连接重试和发送重试操作。

配置方法：

//快速失败

app.configure('production|development', 'connector', function() {

app.set('proxyConfig',

{

failMode : 'failfast'

});

});

//切换服务器

app.configure('production|development', 'connector', function() {

app.set('proxyConfig',

{

failMode : 'failover'

});

});

//安全策略

app.configure('production|development', 'connector', function() {

app.set('proxyConfig',

{

failMode : 'failsafe',

retryTimes:　3， //重试次数

retryConnectTime：　5 \* 1000 //重连间隔时间

});

});

ps: rpc默认采用安全策略。

## pomelo 支持tls及wss

考虑到安全性方面的问题，pomelo 1.0版本中增加了对tls及wss协议的支持；在之前的pomelo版本中，提供了hybridconnector和sioconnector；对于sioconnector,其底层使用的是socket.io，socket.io提供了包括websocket和长轮询等几种传输方式，其中websocket默认采用的ws协议，现在pomelo支持wss协议，即用户可以在sioconnector中配置基于wss协议的websocket的通信方式；对于hybridconnector,提供包括原生socket的支持和websocket的支持，针对这两种方式，在pomelo 1.0中提供了两种连接的安全版本即tls和wss,用户可以在使用hybridconnector时，采用安全级别较高的tls或者wss。

使用方法：

支持tls客户端，pomelo现在提供的tls客户端有libpomelo.

app.configure('production|development', 'connector', function() {

app.set('connectorConfig',

{

connector : pomelo.connectors.hybridconnector,

heartbeat : 3,

useDict : true,

useProtobuf : true,

ssl: {

key: fs.readFileSync('./keys/server.key'),

cert: fs.readFileSync('./keys/server.crt'),

}

});

});

支持wss客户端, pomelo提供的wss客户端有js客户端.

app.configure('production|development', 'connector', function() {

app.set('connectorConfig',

{

connector : pomelo.connectors.hybridconnector,

heartbeat : 3,

useDict : true,

useProtobuf : true,

ssl: {

key: fs.readFileSync('./keys/server.key'),

cert: fs.readFileSync('./keys/server.crt'),

}

});

});

支持socket.io的wss客户端, pomelo提供的socket.io的wss客户端有js客户端.

app.configure('production|development', 'connector', function() {

app.set('connectorConfig',

{

connector : pomelo.connectors.sioconnector,

key: fs.readFileSync('./keys/server.key'),

cert: fs.readFileSync('./keys/server.crt')

});

});

在pomelo 1.0中增加了通过pomelo init 获取wss和socket.io的wss两种客户端及服务端的初始化项目，同时初始化的项目中提供了相应的密钥及证书。注意由于证书是和域名绑定的，所以在打开客户端的时候输入的ip地址为 [https://127.0.0.1:3001](https://127.0.0.1:3001/)

## pomelo 提供zookeeper集群管理

在pomelo之前版本中，提供了master服务器进行集群管理，对于中小型游戏项目及非大型分布式集群项目来说已经能够满足需求；对于大型的分布式项目的集群管理，zookeeper则是最佳选择。为了让pomelo在更广泛的领域适用，在1.0版本中pomelo提供了zookeeper的插件。

其主要原理是利用zookeeper的集群管理，使得pomelo集群中服务器信息保持一致，确保pomelo对外提供稳定的服务。具体实现则是master服务器启动后主动向zookeeper注册一个根节点，其它服务器启动后作为这个根节点的孩子注册到zookeeper,并将服务器相关信息写入，同时非master服务器监听该根节点的孩子节点变化情况；一旦孩子节点有变化，则其它节点重新从根节点或者当前集群中服务器信息。

使用方法：

var zookeeper = require('pomelo-zookeeper-plugin');

app.configure('production|development', function() {

//关闭master集群管理功能

app.set('masterConfig', {

closeWatcher: true

});

app.set('monitorConfig', {

closeWatcher: true

});

app.use(zookeeper, {

zookeeper: {

server: '127.0.0.1:2181',

path: '/pomelo/servers',

username: 'pomelo',

password: 'pomelo'

}

});

}

github地址： [pomelo-zookeeper-plugin](https://github.com/NetEase/pomelo-zookeeper-plugin)

## libpomelo 提供自动重连及SSL/TLS支持

#### libpomelo自动重连

关于libpomelo的自动重连，目前使用的策略为一旦检测到读写socket错误或者数据包解析遇到非法包时，将重置连接，并发起自动重连，重连成功后抛出reconnect事件。 用户可以通过显式地调用pc\_client\_disconnect来断开连接，这样的话，将不会发起自动重连。

新增API:

* pc\_client\_t\* pc\_client\_new\_with\_reconnect(int delay, int delay\_max, int exp\_backoff); 参数说明： delay - 这个参数是重连时，用来退避的延时值，单位为秒 delay\_max - 这个参数是重连时最大延时值，单位也为秒 exp\_backoff - 是否开启指数退避，非0时为开启指数退避，为0时不开启指数退避 返回值： 成功，返回创建成功的实例 失败，返回NULL

例如，如果delay为2s，delay\_max为10s，exp\_backoff为0，不开启指数退避，那么重连时的连接重试间隔将为 2s， 4s， 6s， 8s， 10s， 10s， 10s......直到连接上为止

如果delay为2s， delay\_max为30s，exp\_backoff为1， 开启指数退避，那么重连时的连接重试间隔将为

2s， 4s， 8s， 16s， 30s， 30s.....直到连接上为止

* void pc\_client\_disconnect(pc\_client\_t\* client); 参数说明: client 为pomelo client实例

#### libpomelo SSL/TLS支持

libpomelo目前计划支持SSL/TLS，使用OpenSSL。在使用TLS支持时，在编译时需开启宏 WITH\_TLS。 **拟**增加API如下：

* int pc\_client\_lib\_init(); 这个函数主要用来做一些全局SSL/TLS相关的以及库相关的初始化，此函数应该在所有其他库函数执行前调用一次，不用检查返回值，返回值永远为0。
* int pc\_client\_lib\_cleanup(); 这个函数与pc\_client\_lib\_init相对应，用来在应用关闭时，对整个库做一些全局化的清理工作，其应当是最后一个被调用的库函数，返回值永远为0
* int pc\_client\_set\_tls\_ca(pc\_client\_t\* client, const char\* cafile, const char\* capath) 这个函数用来设置SSL/TLS用到的ca信息. 参数说明： client - pomelo client实例. cafile,capath - 这两个参数用来指定ca信息，具体请参见SSL的调用[SSL\_CTX\_load\_verify\_locations](https://www.openssl.org/docs/ssl/SSL_CTX_load_verify_locations.html). 返回值: 成功，返回值为0 参数错误时，返回值为-1
* int pc\_client\_set\_tls\_cert(pc\_client\_t\* client, const char\* certfile, const char\* keyfile, int (\*pw\_callback)(char\* buf, int size, int rwflag, void\* userdata)); 这个函数是服务端要求对pc client进行验证时，用来设置客户端的证书以及私钥信息， 如果不需要双向验证，则不需要设置。

参数说明: cerfile - 存放证书链的文件，PEM格式，具体请参见OpenSSL的调用[SSL\_CTX\_use\_certificate\_chain\_file](https://www.openssl.org/docs/ssl/SSL_CTX_use_certificate.html) keyfile - 存放私钥的文件，PEM格式。 pw\_callback - 如果私钥文件使用口令加密的话，此参数会被回调用来传递解密私钥文件的口令，否则为NULL即可，具体参见 [SSL\_CTX\_set\_default\_passwd\_cb](https://www.openssl.org/docs/ssl/SSL_CTX_set_default_passwd_cb.html)

* int pc\_client\_set\_tls\_opts(pc\_client\_t\* client, int enable\_verify, const char\* ciphers); 这个参数是用来设置SSL的一些选项的，OpenSSL的提供的选项很多，这里仅提供了两处选项设置 参数说明： enable\_verify - 是否开启服务端证书验证，将此值设置为0，将不验证服务端证书，设置为1将验证服务端证书，缺省时不对服务端证书进行验证 ciphers - 一个字符串用来说明可用的加密算法，如果使用缺省值的话，只需设置为NULL即可
* int pc\_client\_set\_tls\_hostname\_verify(pc\_client\_t\* client, int (\*hostname\_verify\_cb)(pc\_client\_t\* client, const char\*\* names, int len)); 这个函数用来设置对服务端的hostname进行验证的回调的

参数说明： hostname\_verify\_cb - 这个参数是一个回调用来验证服务器端证书中的SAN以及CN字段的，如果设置了此回调，那么当收到证书后，libpomelo会从证书中解析出SAN和CN，并调用此回调函数，此函数返回1表示验证通过，返回0表示验证失败，参数 names 和 len 是用来表示一个字符串数组，其内部存储着证书中解析出来的SAN和CN。对于names，用户在实现回调时，只需要访问其获取SAN和CN即可，不要对其进行其他的操作，libpomelo会负责其内存的申请和释放。

### 说明

关于自动重连和TLS支持，目前还处于实验阶段，其中自动重连已经可以测试，但是TLS实现尚未加完，在github上有auto-reconn分支和ssl分支，也欢迎各位把你修改后的或者更好的实现代码贡献出来, 谢谢。 关于libpomelo任何想法，无论是代码组织，还是具体的实现策略，欢迎大家互相交流。

#### pomelo 提供自动扩展插件

在pomelo1.0里提供了一个服务器自动扩展的插件，其主要原理是监控某一类型的服务器，监控的指标现在暂时提供cpu和memory，当这一类型的服务器的某项监控指标超过之前设置的阈值时，服务器就自动扩展，扩展服务器的数量可以由用户进行配置。

使用方法：

//app.js配置方法

app.configure('production|development', 'master', function() {

app.use(scale, {

scale: {

cpu: {

chat: 5,

interval: 10 \* 1000,

increasement: 1

},

memory: {

connector: 5,

interval: 15 \* 1000,

increasement: 1

},

backup: 'config/development/backupServers.json'

}

});

});

//backupServer.json配置

{

"connector":[

{"id":"backup-connector-server-1", "host":"127.0.0.1", "port":4053, "clientPort": 3053, "frontend": true},

{"id":"backup-connector-server-2", "host":"127.0.0.1", "port":4054, "clientPort": 3054, "frontend": true},

{"id":"backup-connector-server-3", "host":"127.0.0.1", "port":4055, "clientPort": 3055, "frontend": true}

],

"chat":[

{"id":"backup-chat-server-1", "host":"127.0.0.1", "port":6053},

{"id":"backup-chat-server-2", "host":"127.0.0.1", "port":6054},

{"id":"backup-chat-server-3", "host":"127.0.0.1", "port":6055}

]

}

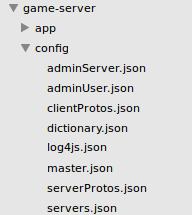
配置参数说明：

现在监控指标包括cpu和memory两项，在每一个监控指标内可以有监控的服务器类型，例如chat：5，这样就表示chat类型的服务器的阈值为5%，当chat类型的服务器cpu的平均值超过5%后，系统将自动扩展服务器，服务器一次扩展的数量由increasement参数决定，例如increasement参数为1，则表示每次超过阈值后扩展1个服务器，扩展服务器的列表由用户指定，backup参数就是扩展的服务器列表；另外interval参数表示系统检测时间，单位是秒，例如interval: 15 \* 1000表示系统每15秒检测一次相应的指标，如果超过该指标则进行相应的扩展。

github地址： [pomelo-scale-plugin](https://github.com/NetEase/pomelo-scale-plugin)

#### 支持按环境目录配置

在pomelo1.0中支持按照目录结构进行配置相关的配置文件，在之前的版本中pomelo的配置文件如下图所示：



不同环境是根据具体配置文件里的key进行区分，例如：

// servers.json配置

{

"development":{

"connector":[

{"id":"connector-server-1", "host":"127.0.0.1", "port":4050, "clientPort": 3050, "frontend": true}

]

},

"production":{

"connector":[

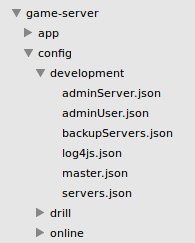
{"id":"connector-server-1", "host":"127.0.0.1", "port":4050, "clientPort": 3050, "frontend": true}

]

}

}

在pomelo1.0支持根据目录进行配置，如下图所示：



config 目录下是根据环境进行文件配置， 在启动过程中选择不同的环境就会根据相应的环境名称目录加载该目录下的所有配置文件。例如 pomelo start env=online 这样就会加载config/online目录下的所有配置文件。默认会加载development下面的配置文件。在这种情况下，对应的servers.json就不需要根据环境配置，具体配置如下：

{

"connector":[

{"id":"connector-server-1", "host":"127.0.0.1", "port":4050, "clientPort": 3050, "frontend": true}

]

}

PS: 默认还是会直接加载config目录下的配置文件，当config目录下面没有对应的文件系统将才会加载环境名称对应的目录下的配置文件；所以要使用根据环境名称目录进行配置时需要先将config目录下之前的配置文件删除。

#### 支持pomelo-cli输入脚本

pomelo 1.0版本中，在pomelo-cli中增加了对脚本输入的支持，主要是方便通过pomelo-cli对不同服务器进行动态地查看相关信息，例如在前端服务器查看相应的session数量，某一个用户的session，或者某一个用户的地址信息等等。

使用方法：

pomelo-cli　　//进入pomelo-cli

use connector-server-1 //使用具体服务器

run app.get("sessionService").getSessionsCount() //执行相应的命令

ps: 脚本默认app为每个服务器的application对象，考虑到通过application对象可以获取到这个服务器的所有信息，所以在这个执行脚本中只提供application对象。

#### sioconnector支持低版本的IE浏览器

之前有很多网友提出希望sioconnector支持低版本的IE浏览器，因为之前sioconnector的通信方法使用了pomelo-protocol,在encode/decode过程中使用了一些低版本IE浏览器不支持的数据结构，在1.0正式版本的pomelo中提供了新的js客户端对IE低版本的客户端支持。客户端代码地址：[pomeloclient.js](https://github.com/pomelonode/pomeloclient)

使用方法：

在web-server中将之前的pomeloclient.js替换为pomeloclient\_ie.js，在game-server中进行对app.js进行配置如下：

var encode = function(reqId, route, msg) {

if (!!reqId) {

return{

id: reqId,

body: msg

};

} else {

var m = {

route: route,

body: msg

};

return JSON.stringify(m);

}

};

var decode = function(data) {

data = JSON.parse(data);

return {

id: data.id,

route: data.route,

body: data.msg

};

};

//将对应的前端服务器进行配置

app.configure('production|development', 'connector|gate', function() {

app.set('connectorConfig', {

connector: pomelo.connectors.sioconnector,

encode: encode,

decode: decode

});

});

#### mqtt connector 支持

在1.0正式版本的pomelo中，发布了mqtt connector.考虑到pomelo本身的体系的兼容，mqtt connector实际上并未采用publish/subscribe的模式。因为mqtt本身协议具有精简、灵活等优势，比较适合移动设备使用，所以pomelo引入mqtt connector主要是使用其协议。

使用方法：

app.configure('production|development', 'connector', function() {

app.set('connectorConfig', {

connector: mqttconnector,

publishRoute: 'connector.mqttHandler.publish',

subscribeRoute: 'connector.mqttHandler.subscribe'

});

});

其中publishRoute指的是客户端通过publish方法发到服务端的消息所导向的handler具体方法，例如publishRoute: 'connector.mqttHandler.publish'表示客户端的publish消息将导入connector的mqttHandler文件的publish方法，subscribeRoute与publishRoute类似。

#### handler及remote的部分热更新支持

在1.0正式版本中，提供了handler和remote的热更新的部分支持。在pomelo1.0正式版中，提供对handler和remote两个目录中的文件进行监控，当有文件更新时，系统会重新load对应的handler和remote服务，这时当有新的handler请求和rpc请求进入时，系统就会自动执行更新后的服务。但是对于在非handler和remote目录的文件就不支持热更新。热更新开关默认是关闭状态，用户可以通过配置打开热更新服务。

使用方法：

// handler 热更新开关

app.set('serverConfig',

{

reloadHandlers: true

});

// remote 热更新开关

app.set('remoteConfig',

{

reloadRemotes: true

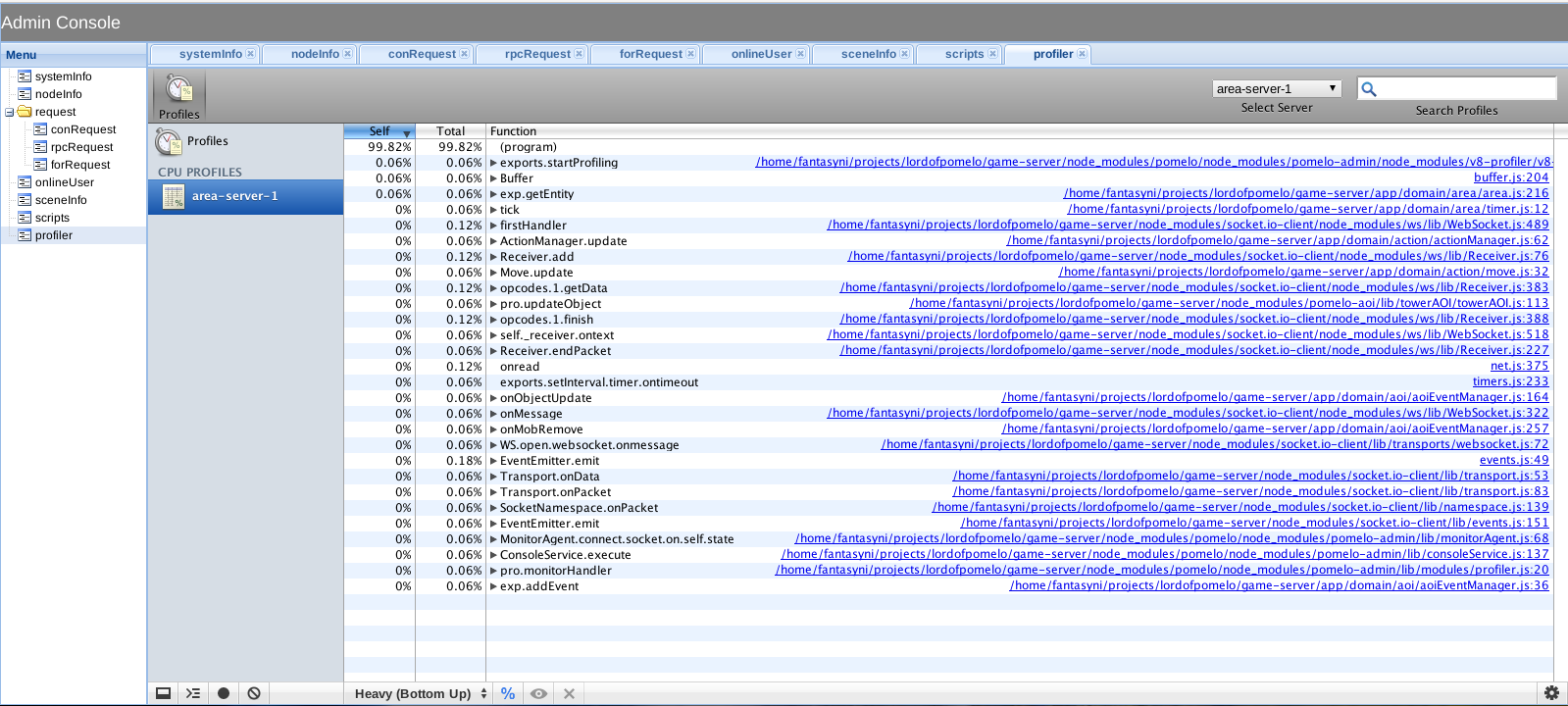
});

#### 其它

在1.0版本中取消对toobusy和ndump模块的支持，但保留toobusy的filter,如果需要可以自己在项目中安装toobusy模块；对于ndump则采用node-heapdump取代，pomelo-cli暂时只支持对内存的dump，停止对cpu dump的支持。

#### pomelo admin web工具的使用

pomelo-admin-web 是 pomelo 框架中基于[pomelo-admin](https://github.com/NetEase/pomelo-admin)开发的web端监控的模块，可以通过 web 端的方式来对游戏服务器集群的运行状态，性能，日志等进行实时的监控，它采用‘类插件’的开发模式，开发者可以很方便的扩展具体的监控模块逻辑，目前在 adminConsole 中，集成的监控模块有如下几个：



* systemInfo  
  用于监控各个服务器上的系统信息，包括 loadavg,men,CPU(I/0),DISK(I/0)
* nodeInfo  
  用于监控各个服务器上的进程信息，包括 pid,cpu%,mem%，vsz，rss
* conRequest  
  用于监控由 connector 请求所产生的日志，包括玩家的登入，移动，切换场景等所花费的时间，并给出具体的路由(route)
* rpcRequest  
  用于监控游戏服务器中 rpc 的调用情况，所花费的时间
* forRequest  
  用于监控由 forward 请求所产生的日志
* onlineUser  
  用于实时监控在线玩家的信息，包括user login name，login ip，login time
* sceneInfo  
  用于实时监控玩家的场景信息，包括玩家所在的服务器，玩家所在的坐标等
* scripts  
  该模块提供了可以在 adminConsole 端在具体的服务器上执行脚本(script)
* profiler  
  该模块集成了chrome控制台下面的 Profiles 性能分析工具，可以用来对Pomelo服务器端的代码进行性能分析

## adminConsole安装与使用

运行环境：linux or mac os x

$ git clone https://github.com/NetEase/pomelo-admin-web.git

$ cd pomelo-admin-web

$ node app

浏览器中访问： http://localhost:7001， 就可以打开管理控制台界面。  
如果在此之前已经启动了 pomelo 项目，就可以在 adminConsole 上面进行监控了  
如果端口有冲突，请在config/admin.json修改端口，访问的浏览器必须支持websocket，推荐使用chrome.

## scripts模块脚本编写注意事项

scripts模块使用了 node.js 中的 [vm](http://nodejs.org/api/vm.html) module 来执行脚本  
内置提供了  
app --- pomelo application 实例  
os --- os 模块  
fs --- fs 模块  
process --- process 模块  
util --- util 模块  
来做为 vm 的 sandbox 上下文环境，即我们可以直接在脚本中调用这些  
为了便于输出结果，在 adminConsole 中，把执行的结果统一赋值给了全局 result 变量  
因此，在编写脚本的时候，要输出的结果要赋值给 result 变量（不要用 var 进行声明，它是全局的）  
比如，可以编写一个脚本来获取服务器的cpu数量信息

var cpus = os.cpus();

result = util.inspect(cpus,true,null);

# Pomelo cli使用

## pomelo-cli 交互式命令行

pomelo-cli 提供了一个交互式命令行，开发者可以使用这个工具对使用[pomelo](https://github.com/NetEase/pomelo)框架开发的应用和服务进行运维

### 安装

npm install -g pomelo-cli

### 登录

使用命令

pomelo-cli -h host -P port -u username -p password

默认的pomelo-cli登录参数

pomelo-cli -h 127.0.0.1 -P 3005 -u monitor -p monitor

登录用户的用户名密码权限等级是在config/adminUser.json里面进行配置的  
具体可以参考[admin user](https://github.com/NetEase/pomelo-admin#user-level-control)

对于kill, stop, add ,enable, disable 等命令是受权限限制的，非admin用户将无权进行操作

### 命令

#### use

使用某个context来进行操作，context可以是serverId或者是all  
use {serverId|all}  
之后你的命令就会被应用到该context

example: use area-server-1

example: use all

#### quit

敲入 **quit** 你就会退出 pomelo-cli

#### kill

关闭所有服务器

example: kill

**注意**：小心使用该命令

#### exec

执行脚本文件  
exec {filepath}  
filepath 可以是相对于pomelo-cli命令执行的路径

example : exec xxx.js

等价于 exec pwd/xxx.js

filepath 也可以是以 / 开头的绝对路径

example : exec /home/user/xxx.js

脚本文件的执行是通过 [vm](http://nodejs.org/api/vm.html) 模块  
vm context 是

var context = {

app: this.app,

require: require,

os: require("os"),

fs: require("fs"),

process: process,

util: util

};

执行结果是通过 **result** 参数  
因此，你在脚本文件里，你需要使用**result**来得到返回结果  
getCPUs.js

var cpus = os.cpus();

result = util.inspect(cpus,true,null);

#### get

等价于 app.get(key)  
get {key}

#### set

等价于 app.set(key, value)  
set {key} {value}  
**注意**：value 必须是 string 或者简单的value

#### add

动态添加服务器到pomelo集群中  
add 参数是来自于servers.json配置文件中的 key = value 对

example: add host=127.0.0.1 port=3451 serverType=chat id=chat-server-2

example: add host=127.0.0.1 port=3152 serverType=connector id=connector-server-3 clientPort=3012 frontend=true

**注意**：添加服务器必须使用正确完整的参数，否则添加的服务器会处于bad模式

#### stop

停止服务器，以serverId作为参数  
stop {serverId}

example: stop area-server-1

#### show

查看信息比如：servers, connections  
可以查看如下信息：  
servers, connections, logins, modules, status, proxy, handler, components, settings

example: show servers

example: show connections

example: show proxy

example: show handler

example: show logins

**注意**: 只有查看servers的时候是可以在任何一个context下面的，其他的所有的信息查看必须在某一个server的context下进行。

#### enable

enable admin module 设置 或者 enable app 设置  
enable module {moduleId}  
enable app {settings}

example: enable module systemInfo

example: enable app systemMonitor

#### disable

disable admin module 设置 或者 disable app 设置  
disable module {moduleId}  
disable app {settings}

example: disable module systemInfo

example: disable app systemMonitor

#### dump

dump v8 heap 和 cpu 以供之后的分析  
dump cpu|memory {filepath} [times] [--force]  
times 是 cpu dump 所需要的时间，以秒为单位

example: dump cpu /home/xxx/test 5

example: dump memory /home/xxx/test

**注意**：你可以使用 --force 来覆盖写入已经存在的文件

example: dump cpu /home/xxx/test 5 --force

example: dump memory /home/xxx/test --force

##### 分析dump结果

打开[google chrome](https://www.google.com/intl/en/chrome/browser/)浏览器，按下F12来打开浏览器控制台  
找到**profile**标签，右键选择**Load profile...**  
选择dump文件，点击**open**。dump 文件就会被载入，你就可以进行分析  
关于dump的更多信息，你可以访问[ndump](https://github.com/piaohai/ndump)

# Pomelo daemon的使用

## pomelo-daemon

pomelo-daemon 提供了一个 daemon 服务，可以用这个服务来进行分布式部署以及日志收集

### 安装

npm install -g pomelo-daemon

### 使用

#### 启动pomelo集群

* 在服务器上面部署代码
* 把servers.json上面的host配置称为正确的host，而不是 '127.0.0.1'
* 在config文件夹下面添加daemon.json文件  
  daemon.json 例子

{

"id": "dh37fgj492je",

"key": "agarxhqb98rpajloaxn34ga8xrunpagkjwlaw3ruxnpaagl29w4rxn",

"algorithm": "sha256",

"user": "pomelo"

}

**注意**：pomelo-daemon 使用 [hawk](https://github.com/hueniverse/hawk) 来提供服务间的请求认证

* cd 到 /game-server 路径下面
* 在master服务器上，敲入命令

pomelo-daemon

* 在其它服务器上，敲入命令

pomelo-daemon --mode=server

**注意**：你可以使用nohup来部署daemon

nohup pomelo-daemon --mode=server

* 在master服务器上，pomelo-daemon client，敲入命令

start all

* pomelo 集群启动起来了

#### daemon rpc 日志收集

pomelo-daemon 提供了 pomelo rpc 日志收集同步到 mongodb，然后可以通过 [pomelo-admin-web](https://github.com/NetEase/pomelo-admin-web) 来进行分析查看

* 添加 mongo.json 文件到 config 文件夹下面  
  mongo.json 例子

{

"host": "localhost",

"port": 27017,

"username": "pomelo",

"password": "pomelo",

"database": "test",

"collection": "cpomelo"

}

* 启动 pomelo-daemon rpc logger 收集，使用 --pattern 参数来设置 rpc-log 文件的patterns

pomelo-daemon --mode=server --log --pattern=rpc-log

**注意**：rpc-logs 日志收集仅仅用于调试，在生产环境下面不建议使用rpc-logs模式

# pomelo robot使用文档

### pomelo-robot

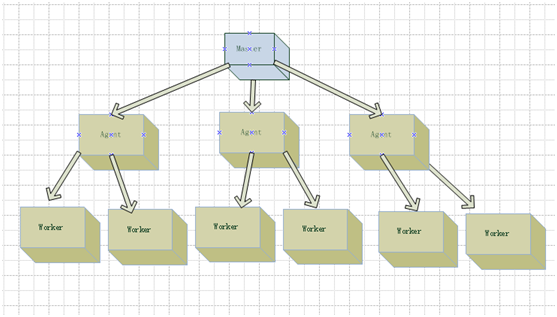
pomelo-robot 是一个用来对pomelo游戏框架进行性能测试的工具，也可以测试其他基于socket.io的服务的性能。 本模块可以采用单机测试模式，也可以采用分布式测试模式。

### 目的

为了能对游戏项目进行性能对比分析与自动化操作，同时为游戏服务器提供机器人的功能和指令配置方式对游戏服务器进行测试。 目前主要考虑二种功能：运行简单指令的机器人和多线程的性能测试。后者依赖于前者的功能实现，同时还可能考虑输出性能分析报告。 pomelo-robot 模块通过沙箱的方式执行用户自定义的Javascript脚本，运行中各客户端会自动把数据汇报给主节点， 主节点把所有节点的测试数据进行清洗与合并，算出最大最小及平均响应时间等统计数据，然后定时发关给内置的HTTP服务器进行界面展示。

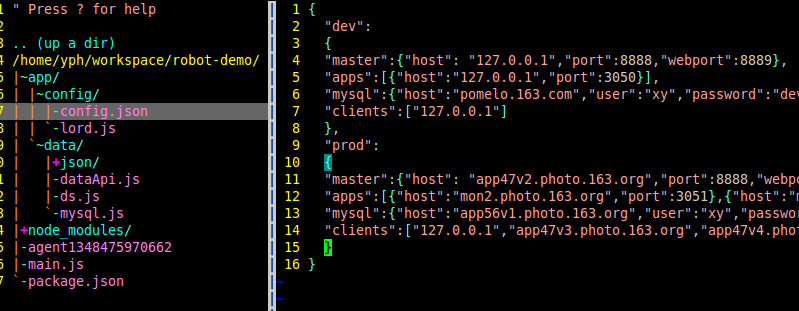
### 模块结构

模块内部运行结构如下：



Master负责启动各Agent客户端，同时在界面展示报告统计中心与数据来源等相关配置。  
Agent负责启动游戏客户端的连接及运行自定义脚本同时向Master汇报运行数据。

#### 使用示例

新建NODEJS测试工程,工程目录结构如下图所示：  
  
安装依赖库  
npm install pomelo-robot

#### app.json配置

app.json分为二种运行环境本地开发或线上测试，文件内容如下：

{

"dev":

{

"master":{"host": "127.0.0.1","port":8888,"webport":8889},

"apps":[{"host":"127.0.0.1","port":3050}],

"mysql":{"host":"pomelo.163.com","user":"xy","password":"dev","database":"Pomelo"},

"clients":["127.0.0.1"]

},

"prod":

{

"master":{"host": "app47v2.photo.163.org","port":8888,"webport":8889},

"apps":[{"host":"mon2.photo.163.org","port":3051}],

"mysql":{"host":"app56v1.photo.163.org","user":"xy","password":"dev","database":"Pomelo"},

"clients":["127.0.0.1","app47v3.photo.163.org","app47v4.photo.163.org"]

}

}

master：主服务器的IP，客户端通信端口，WEB界面端口。  
apps：应用服务器即游戏服务器的对外提供的接口，支持多台服务器配置，客户端会均衡的连接各服务器。  
mysql：配置测试用的数据来源。  
clients：测试客户端，线上运行时采用ssh运行时，此模式需要各机器配置同样环境无密码的ssh登录。

#### 实现自定义脚本配置

库本身对用户提供Iuser对象和robot对象，Iuser示例中主要包括用户名和密码，robot采用event的模式支持如pomele client类型的request和on方法。  
实现自定义脚本LORD.JS。

var login = function(){

var data = {route:'connector.loginHandler.login', username:Iuser.username, password:Iuser.passwd};

robot.request(data,loginRes,true);

};

/\*\*

\* 处理登录请求

\*/

var loginRes = function(data){

//console.log('longined %j',data);

};

login();

上面代码运行在沙箱引擎中，首先登录游戏服务器，成功后会回调loginRes函数，用户可以继续后续的相关操作，这里不再给出具体代码。

#### 配置启动文件main.js

示例启动文件main.js因为本身简单，直接负责了根据参数判断启动master服务或agent服务。  
master服务负责启动agent服务并传入相关的启动参数，包括（agent名称，运行的机器人数量）  
整体代码如下：

var Robot = require('pomelo-robot').Robot;

//服务相关配置

var config = require('./app/config/config');

var robot = new Robot(config);

if (robot.server==='master') { //启动master 服务

robot.runMaster(\_\_filename);

} else {

//示例中第5个参数为mysql数据源分页参数

var i = 5;

var limit = process.argv[i++];

var offset= process.argv[i++];

//载入用户自定义的脚本。

var script = require('fs').readFileSync(process.cwd() + '/app/config/lord.js', 'utf8');

queryHero(client,limit,offset,function(error,users){ //启动Agent 服务

robot.runAgent(users,script)

});

}

上面代码示例，获得Mysql数据源的代码如下：

//数据库配置

var mysql =config[robot.env].mysql;

var Client = require('mysql').Client;

var client = new Client();

client.host = mysql.host;

client.user = mysql.user;

client.password = mysql.password;

client.database = mysql.database;

queryHero = function(client,limit,offset,cb){

var users = [];

var sql = "SELECT User.\* FROM User,Player where User.id = Player.userId and User.name like 'pomelo%' limit ? offset ? ";

var args = [parseInt(limit),parseInt(offset)];

client.query(sql,args,function selectCb(error, results, fields) {

if (!!error) {

console.log('queryHero Error: ' + error.message);

cb(null,users);

}

for (var i = 0;i<results.length;i++) {

var user = {uid:results[i]['id'],username:results[i]['name'],passwd:results[i]['passwd']||'123'};

users.push(user);

};

cb(null,users);

});

};

#### 运行测试

运行如下命令：  
node main.js 默认启动开发环境，如需要启动线上测试环境，在线上master机器上执行node main.js prod即可。  
打开浏览器访问如下地址  
http://masterIp:8889

点击准备按钮，可以看到连接上来的客户端，当所有的客户端准备好了，点击运行即可。  
管理界面会定时得到后台数据与展示出来。  
运行界面如下图所示：



#### 注意

当分布时使用时，各客户端配置的机器的目录是统一的，并且主节点能通过SSH无密码访问到各节点。  
另外，开发人员也可以自行登录到各客户端通过参数根据主服务器生成的命令手工启动。

#### 其他

详细的使用及完全的源代码请参考[工程](https://github.com/NetEase/pomelo-robot)

# pomelo rpc zeromq性能测试报告

### 测试环境

##### 1.1 服务端测试环境

|  |  |  |
| --- | --- | --- |
| **服务** | **机器** | **硬件配置** |
| GameServer | pomelo3.server.163.org | 云主机 1CPU 8核心 CPU型号 GenuineIntel QEMU Virtual CPU version 1.1.2@2.0GHz 16G 内存 1网卡 linux/64位 OS |

##### 1.2 客户端测试环境

|  |  |  |
| --- | --- | --- |
| **服务** | **机器** | **硬件配置** |
| Clients | pomelo16~18.server.163.org | 云主机 1CPU 1核心 CPU型号 GenuineIntel Westmere E56xx/L56xx/X56xx (Nehalem-C)@2.6GHz 1G 内存 1网卡 linux/64位 OS |

### 测试结果

#### 场景A

1. 1个服务进程.
2. 2个客户端并发, 每隔1ms发起一次RPC调用(msg='Hello World'), 每个客户端总计发送1w次, 服务器对每个RPC请求回复一个200.
3. 服务器完成2w次RPC请求的时间为14.595s, 平均1370次/s.
4. 服务器完成一次RPC调用的时间约为: 1~4ms
5. 在服务器运行过程中: server进程对CPU的占用平均值为: 37.8% [CPU占用的采样点为: 36%, 38%, 43%, 35%, 37%]
6. 在客户端运行过程中: client进程对CPU的占用平均值为: 31.8% [CPU占用的采样点为: 37%, 35%, 31%, 31%, 25%]

#### 场景B

1. 4个connector和1个echo业务进程.
2. 4个客户端并发且分别连接1个connector, 每隔1ms发起一次request请求(msg='Hello World'), 每个客户端总计发送1w次, 服务器对每个request回复一个200.
3. 服务器完成4w次请求的时间为15.12s, 平均2645次/s.
4. 服务器完成一次RPC调用的时间约为: 1~8ms
5. 在服务器运行过程中: connector进程对CPU的占用平均值为: 71.25% [CPU占用的采样点为: 70%, 65%, 76%, 74%]; echo进程对CPU的占用平均值为: 80.75% [CPU占用的采样点为: 85%, 78%, 82%, 78%]
6. 在客户端运行过程中: client进程对CPU的占用平均值为: 27.6% [CPU占用的采样点为: 26.2%, 26.5%, 28.2%, 29.4%]

# pomelo rpc性能测试报告

### 测试环境

##### 1.1 服务端测试环境

|  |  |  |
| --- | --- | --- |
| **服务** | **机器** | **硬件配置** |
| GameServer | pomelo3.server.163.org | 云主机 1CPU 8核心 CPU型号 GenuineIntel QEMU Virtual CPU version 1.1.2@2.0GHz 16G 内存 1网卡 linux/64位 OS |

##### 1.2 客户端测试环境

|  |  |  |
| --- | --- | --- |
| **服务** | **机器** | **硬件配置** |
| Clients | pomelo16~18.server.163.org | 云主机 1CPU 1核心 CPU型号 GenuineIntel Westmere E56xx/L56xx/X56xx (Nehalem-C)@2.6GHz 1G 内存 1网卡 linux/64位 OS |

### 测试结果

#### 场景A

1. connector和echo业务进程各1个.
2. 2个客户端并发, 每隔1ms发起一次request请求(msg='Hello World'), 每个客户端总计发送1w次, 服务器对每个request回复一个200.
3. 服务器完成2w次请求的时间为14.835s, 平均1348次/s.
4. 服务器完成一次RPC调用的时间约为: 2~8ms
5. 在服务器运行过程中: connector进程对CPU的占用平均值为: 91.6% [CPU占用的采样点为: 92%, 94%, 95%, 87%, 84%, 96%, 93%]; echo进程对CPU的占用平均值为: 28.1% [CPU占用的采样点为: 30%, 20%, 33%, 22%, 25%, 46%, 21%]
6. 在客户端运行过程中: client进程对CPU的占用平均值为: 30.1% [CPU占用的采样点为: 18%, 24%, 25%, 40%, 16%, 49%, 39%]

#### 场景B

1. 4个connector和1个echo业务进程.
2. 4个客户端并发且分别连接1个connector, 每隔1ms发起一次request请求(msg='Hello World'), 每个客户端总计发送1w次, 服务器对每个request回复一个200.
3. 服务器完成4w次请求的时间为14.866s, 平均2690次/s.
4. 服务器完成一次RPC调用的时间约为: 1~25ms
5. 在服务器运行过程中: connector进程对CPU的占用平均值为: 71.8% [CPU占用的采样点为: 75%, 71%, 71%, 74%, 68%]; echo进程对CPU的占用平均值为: 81.3% [CPU占用的采样点为: 81%, 82%, 83%, 79%]
6. 在客户端运行过程中: client进程对CPU的占用平均值为: 28.0% [CPU占用的采样点为: 28%, 29%, 29%, 26%]

# pomelo sync使用文档

## 关于

pomelo sync 模块是管理游戏进程内需要持久化的数据在内存和存储系统之间的同步，提供一种异步的同步方式，根据用户配置定时的同步数据到持久层，如MYSQL，Redis，文件等。

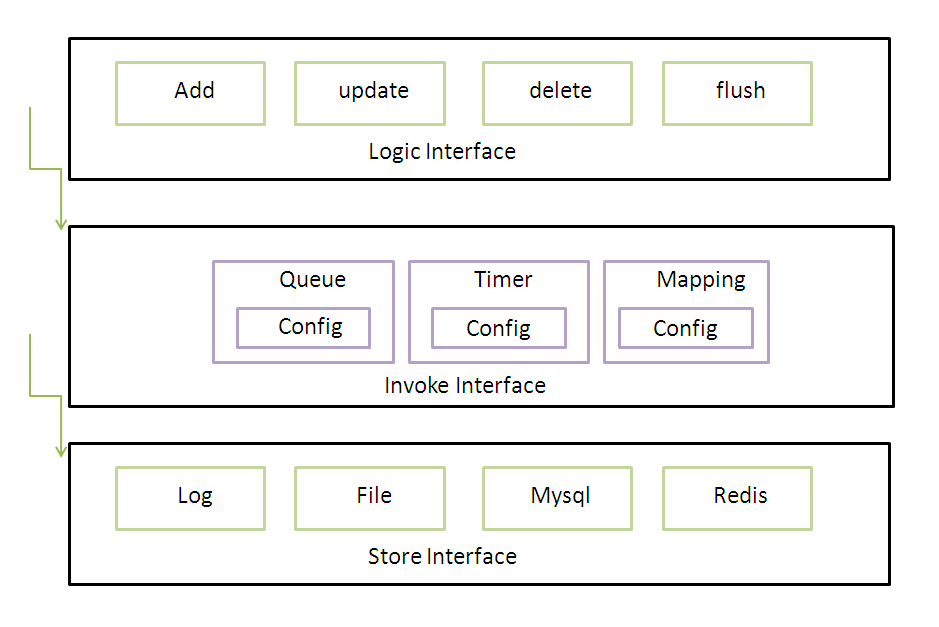
## 目的

由于在游戏应用场景中，需要大量的数据更新与同步，如用户的位置，血量，装备值等，如果频繁的操作数据层，会产生很大的IO操作开销，采用定时批量的方式处理变动的数据是避免压力过大的方法之一，pomelo sync就是为了满足这样的需求而开发的。

类似与IBatis的配置，但是ORM完全由用户配置控制，具有很大的灵活性，因此，可以应用于自己的不同数据层中，如其他的Mongdb等。

## 模块结构

pomelo sync模块内部结构如下：



Logic Interface：提供侵入式的数据更新调用接口，包括常规的add、update、delete和立即执行持久化的接口

Invoke Interface：提供持久化的实际调用执行：  
Queue：提供本周期内需要持久化的对象，可支持多节点转发的数据  
Timer：提供定时静态配置、动态修改  
Mapping：提供在持久化时同步方法的映射

Store Interface：封装了各存储引擎的持久化接口以  
Log：提供对数据变动时的AOF日志记录  
Mysql,Redis,File：根据用户传入的对象执行用户自定义的同步方法。

## 安装

npm install pomelo-sync

## 使用示例

### 创建sync对象

//引用库

var DBsync = require('pomelo-sync');

//配置客户端连接,即同步的连接，如mysqlclient;

var dbclient = mysqlclient;

var opts = opts || {};

//配置映射的路径方法

opts.client = dbclient;

opts.interval = opts.interval || 60 \* 1000;

var sync = new DBsync(opts) ;

### 配置同步对象映射关系

目前，本库支持用户自行定义和设置映射路径，代码分别如下：

#### 自定义传入的方法

var updateTask = function(dbclient,val) {

var sql = 'update Task set taskState = ?, startTime = ?, taskData = ? where id = ?';

var taskData = val.taskData;

if (typeof taskData !== 'string') {

taskData = JSON.stringify(taskData);

}

var args = [val.taskState, val.startTime, taskData, val.id];

dbclient.query(sql, args, function(err, res) {

if (err) {

console.error('write mysql failed! ' + sql + JSON.stringify(val));

} else {

callback && callback(!!err);

}

});

}

var optKey = 'taskSync.updateTask';

sync.mapping = {optKey:updateTask};

​

#### 传入用户自定义路径

由模块扫描配置的文件夹的所有JS文件，并把JS的文件名和导出的方法作为Key与对应的导出方法映射。

var mappingPath = \_\_dirname+ '/mapping';

sync.mapping = sync.loadMapping(mappingPath);

mapping文件夹下的taskSync.js文件内容

module.exports = {

updateTask: function(dbclient, val, callback) {

var sql = 'update Task set taskState = ?, startTime = ?, taskData = ? where id = ?';

var taskData = val.taskData;

if (typeof taskData !== 'string') {

taskData = JSON.stringify(taskData);

}

var args = [val.taskState, val.startTime, taskData, val.id];

dbclient.query(sql, args, function(err, res) {

if (err) {

console.error('write mysql failed! ' + sql + JSON.stringify(val));

} else {

callback && callback(!!err);

}

});

}

};

### 添加需要同步的对象

需要传入同步时回调方法的关键词，同步对象的主键及同步对象，定时器会把添加的对象同步到数据库中。

var id = 10001;

var task = {"taskState":1,"startTime":Date.now(),"taskData":1,"id":id};

sync.exec(optKey,id,task);

​

## API

### sync.exec(key,id,val,cb)

添加异步定时执行的操作

#### Arguments

* key - 方法映射的关键词，使用时需要唯一。
* id - 实体对象主键。
* val - 需要同步对象，添加后会克隆此对象。
* cb - 同步完成后的异步回调，可以为空。

具体使用见上示例

### sync.flush(key,id,val,cb)

立即同步某个对象到持久层

#### Arguments

* key - 方法映射的关键词，使用时需要唯一。
* id - 实体对象主键。
* val - 需要同步对象。
* cb - 同步完成后的异步回调，可以为空。

具体使用同上示例。

### sync.sync()

用户手工立即执行同步过程，不等到定时器调用。一般使用在服务器关闭关调用。代码如下：

var stop = function(cb) {

var self = this;

self.sync.sync();

self.state = STATE\_STOPED;

var interval = setInterval(function(){

if (self.sync.isDone()) {

clearInterval(interval);

cb();

}

}, 200);

};

​

### sync.isDone

获得内存是否还有需要同步对象的状态，有的话为FALSE，用户可以根据状态判断是否关闭。 一般配置sync()方法一起使用，具体使用见上面的代码

## 其他参数选项

系统默认的同步间隔时间为 1000 \* 60（即1分钟）,需要修改的在opts.interval传入即可。  
系统默认是关闭aof日志记录，如需要可使用opts.aof = ture打开。

## 注意

使用外部的持久化时需要设置对应的CLIENT对象，在调用方法时会按第一个参数传给调用方法。需要同步的对象传出在第二个参数位置上。 本模块需要采用持久层的内置事务模型来保证对事务的支持，本身不提供。

## 其他

更详细的示例，请参考库的[源代码](https://github.com/NetEase/pomelo-sync)与[游戏DEMO](https://github.com/NetEase/lordofpomelo)。

# Pomelo协议

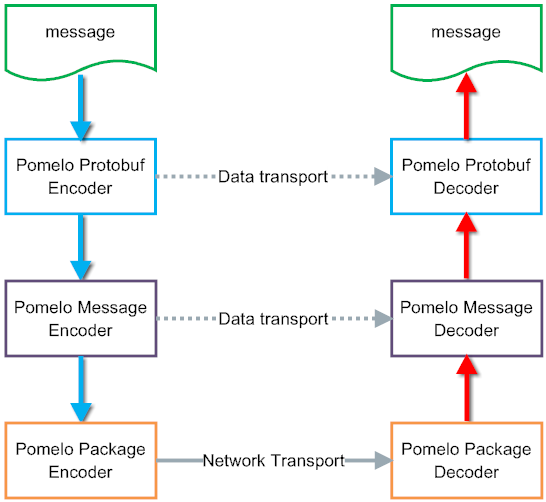
### Pomelo协议

### 背景

0.3版本之前的Pomelo采用json格式的文本作为客户端和服务器的通讯协议，方便协议定制和修改，但同时也带来了较多的通讯冗余数据。Pomelo在0.3版本，底层通讯提供了二进制版本通讯协议的支持，同时提供了route字典压缩和protobuf压缩，提高带宽利用率，以满足诸如移动环境的需求。同时上层接口仍保持json格式的接口，对0.3版本之前的代码不产生任何影响，也保留了开发的便利。

Pomelo的二进制协议包含两层编码：package和message。Message层主要实现route压缩和protobuf压缩，message层的编码结果将传递给package层。Package层主要实现Pomelo应用基于二进制协议的握手过程，心跳和数据传输编码，package层的编码结果可以通过tcp，websocket等协议以二进制数据的形式进行传输。Message层编码可选，也替换成其他二进制编码格式，都不影响package层编码和发送。

Pomelo协议层的结构如下图所示：

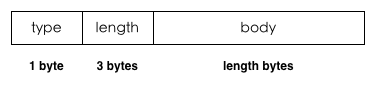


### Pomelo Package

Package协议主要用来封装在面向连接的二进制流的通讯协议（如：tcp）上的Pomelo数据包。Package分为控制包和数据包两种类型。前者用来实现Pomelo应用层面的控制流程，包括客户端和服务器的握手，心跳和服务器主动断开连接的通知等控制信息。后者则是用来在客户端和服务器之间传输应用数据。

#### Package格式

Package分为header和body两部分。Header描述package包的类型和包的长度，body则是需要传输的数据内容。具体格式如下：



* type - package类型，1个byte，取值如下。
  + 0x01: 客户端到服务器的握手请求以及服务器到客户端的握手响应
  + 0x02: 客户端到服务器的握手ack
  + 0x03: 心跳包
  + 0x04: 数据包
  + 0x05: 服务器主动断开连接通知
* length - body内容长度，3个byte的大端整数。
* body - 二进制的传输内容。

各个package类型的具体描述和控制流程如下。

#### 握手

握手流程主要提供一个机会，让客户端和服务器在连接建立后，进行一些初始化的数据交换。交换的数据分为系统和用户两部分。系统部分为Pomelo框架所需信息，用户部分则是用户可以在具体应用中自定义的内容。

握手的内容为utf-8编码的json字符串（不压缩），通过body字段传输。

握手请求：

{

'sys': {

'version': '1.1.1',

'type': 'js-websocket'

},

'user': {

// any customized request data

}

}

* sys.version - 客户端的版本号。每个客户端SDK的每一个版本都有一个固定的版本号。在握手阶段客户端将该版本号上传给服务器，服务器可以由此来判断当前客户端是否合适与服务器通讯。
* sys.type - 客户端的类型。可以通过客户端类型和版本号一起来确定客户端是否合适。

握手响应：

{

'code': 200, // result code

'sys': {

'heartbeat': 3, // heartbeat interval in second

'dict': {}, // route dictionary

'protos': {} // protobuf definition data

},

'user': {

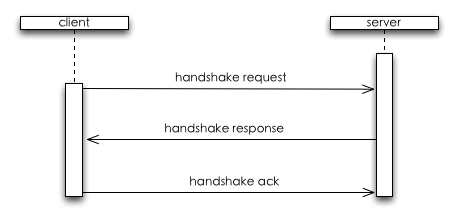
// any customized response data

}

}

* code - 握手响应的状态码。目前的取值：200代表成功，500为处理用户自定义握手流程时失败，501为客户端版本号不符合要求。
* sys.heartbeat - 心跳时间间隔，单位为秒。
* dict - route字段压缩的映射表
* protos - protobuf压缩的数据定义

握手的流程如下：

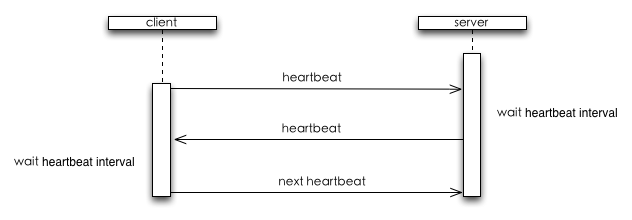


当底层连接建立后，客户端向服务器发起握手请求，并附带必要的数据。服务器检验握手数据后，返回握手响应。如果握手成功，客户端向服务器发送一个握手ack，握手阶段至此成功结束。

#### 心跳

心跳包的length字段为0，body为空。

心跳的流程如下：



服务器可以配置心跳时间间隔。当握手结束后，服务器发起第一个心跳。服务器和客户端收到心跳包后，延迟心跳间隔的时间后再向对方发送一个心跳包。

心跳超时时间为2倍的心跳间隔时间。服务器检测到心跳超时并不会主动断开客户端的连接。客户端检测到心跳超时，可以根据策略选择是否要主动断开连接。

#### 数据

数据包用来在客户端和服务器之间传输数据所用。数据包的body是由上层传下来的任意二进制数据，package层不会对body内容做任何处理。

#### 服务器主动断开

当服务器主动断开客户端连接时（如：踢掉某个在线玩家），会先向客户端发送一个控制消息，然后再断开连接。客户端可以通过这个消息来判断是否是服务器主动断开连接的。

### Pomelo Message

Message 协议的主要作用是封装消息头，包括route和消息类型两部分，不同的消息类型有着不同的消息头。

#### Message 头格式

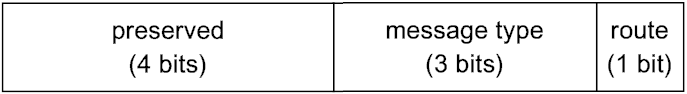
消息头分为三部分，flag，message id，route。如下图所示：



从上图可以看出，Pomelo消息头是可变的，会根据具体的消息类型和内容而改变。其中，flag位是必须的，占用一个byte，它决定了后面的消息类型和内容的格式。message id和route则是可选的，具体的长度要根据实际内容和flag来决定。

##### 标志位Flag

Flag占用Message头的第一个byte，其内容如下

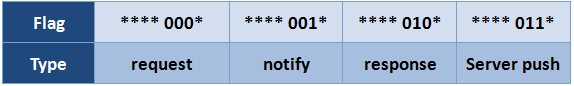


现在只用到了其中的4个bit，这四个bit包括两部分，占用4个字节的message type字段和占用1个字节的route标识，其中：

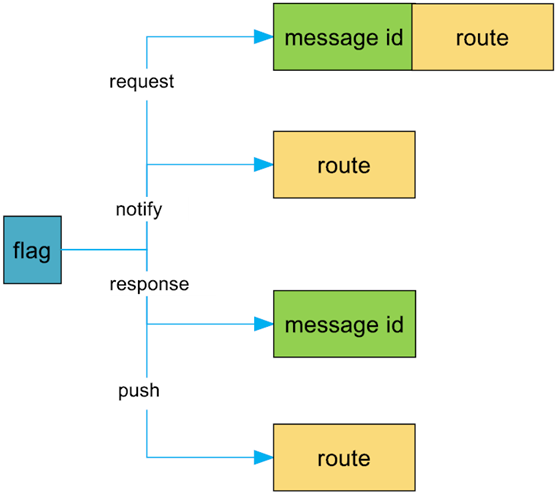
* message type用来标识消息类型,范围为0～7，现在消息共有四类，request，notify，response，push，值的范围是0～3。不同的消息类型有着不同的消息内容，下面会有详细分析。
* 最后一位的route表示route是否压缩，影响route字段的长度。 这两部分之间相互独立，互不影响。

##### 消息类型Message Type 解析

Message type与消息类型的关系如下表：

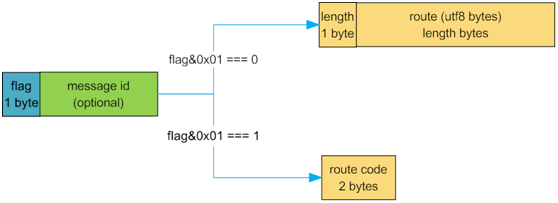


不同的消息类型对应着不同的消息头，具体的对应关系如下图：



##### Route标志位

route主要分为压缩和未压缩两种，由flag的最后一位（route位）指定，当flag中的route标志为0时，表示未压缩的route，为1则表示是压缩route。route通过系统生成和用户自定义的字典进行压缩，具体内容见（pomelo压缩协议-字典压缩）。



上图是不同的flag标志对应的route字段的内容：

* flag的最后一位为1时，后面跟的是一个uInt16表示的route字典编号，需要通过查询字典来获取route。
* flag最后一位为0是，后面第一位unt8的byte，用来表示route的字节长度。之后是通过utf8编码后的route，其长度就是前面一位byte的uInt8的值。

# Pomelo 数据压缩协议

### Pomelo 数据压缩协议

在pomelo 0.3中，为了减少数据传输带宽，提高传输效率，我们支持了基于字典的route压缩和基于protobuf的传输数据压缩。

### 基于字典的route压缩

#### route字段分析

pomelo中的route是用来确定消息的分发路径，将其交给相应的服务器和服务处理的。route分为两类，由客户端发给服务端消息时使用的route和服务端向客户端广播时使用的route。

* 前一种route是由服务器自动生成的，其中的字段就代表了对应的方法在服务端的位置。如“area.playerHandler.attack”则表示在“area”服务器上的“playerHandler”接口中提供的“attack”方法。
* 后一种route是服务端向客户端推送消息时使用，如“onMove”，“onAttack”等，这些字段是由用户自己定义的。 在一般的web应用等带宽不敏感的环境中，route字段的开销是可以接受的。而在一些移动应用中，带宽～money的情况下，精简route字段就变得有必要了。

#### Pomelo中route压缩的问题

原有的route模式不能支持方便的数据压缩；

* 对pomelo生成的route来说，如“area.playerHandler.attack”,其中的每一个都表示系统中的路径，因此除非修改目录和方法名称，否则无法进行精简。
* 用户自定义的route，如“onMove”，“onAttack“等可以较方便的进行精简。但是用户直接可对内容修改会导致route名与实际的语义脱离，从而降低代码的可读性，而简化后的route也增加了路由冲突的可能性，更整个项目的管理带来困难。

#### Pomelo 中的route压缩

Pomelo 0.3版中实现了基于字典的route压缩，其实现如下：

* 对于系统生成的route，如“area.playerHandler.attack"，在系统启动时会对每一个route生成唯一的字典（16位的short int）。
* 对于用户自定义的route，如“onMove”，“onAttack”，则需要用户提供一个自定义的route列表，pomelo会根据这一个列表对每个用户自定义的route生成一个字典项。 在开启字典功能的状态下，当有消息传递时，其中的route在发送时会被替换为在字典项（16bit short int），而接收端会自动还原，这一过程对于用户而言是完全透明的。要使用这一功能，只需要在app.js中配置开启就可以了，具体示例如下：

app.set('connectorConfig',

{

connector : pomelo.connectors.hybridconnector,

heartbeat : 3,

useDict : true,

handshake : function(msg, cb){

cb(null, {});

}

});

其中的userDic：true就表示打开字典压缩。默认字典包括所有系统自动生成的route，如"area.playerHandler.move"等。用户自定义的route则需要用户自己指定，具体方式是在项目的/game-server/config/dictionary.json文件中列出需要压缩的自定义route，其内容示例如下：

[

"onDropItem",

"onAttack",

"onDied",

"onMove",

"onUpgrade",

"onPickItem",

"onRevive",

"addEntities",

"onRemoveEntities",

"onPathCheckout"

]

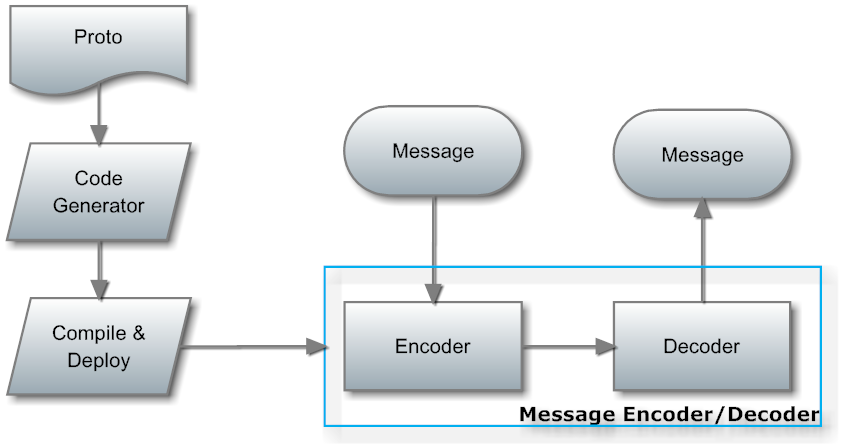
可以看到，dictionary.json的内容就是一个string的数组，其中的内容是需要压缩的route。Pomelo在启动时会自动读取其中的内容，生成数据字典，并在数据传输时进行替换/还原。

### 基于protobuf的数据编码协议

在Pomelo 0.3中，我们实现了基于protobuf的数据编码协议，与其他的编码协议如xml，json相比，protobuf有着更好的传输效率和压缩比率。在我们的lordofpomelo项目中，使用protobuf进行数据编码后的消息大小只有基于Json的编码的20%左右。

#### protobuf协议介绍

protobuf协议是由Google制定的，主要用于其内部的rpc调用和文件编码。原生的protobuf包括两部分内容：基于二进制的数据编码协议和基于proto元数据的代码生成器。首先，需要根据每条消息来编写对应的proto文件，然后使用google提供的代码生成器，基于proto文件来生成相应的编码器和解码器，然后使用生成的编/解码器来进行编/解码操作，对应的流程如下图：



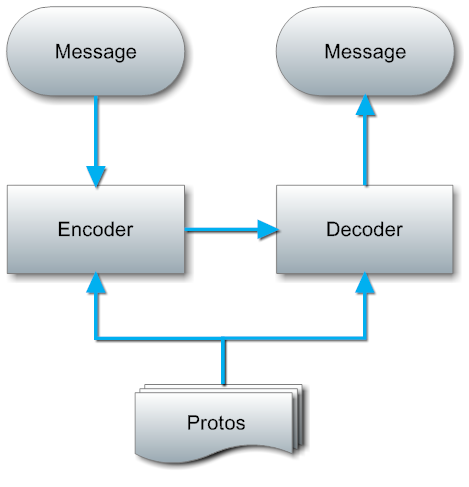
这种方式的优势是代码静态生成，运行时不需要proto文件信息，而且可以根据具体的信息内容对代码进行优化。但缺点也十分明显：使用复杂（涉及到代码生成，编译，部署），改动成本高昂（需要重新生成，编译代码，并对代码进行部署），需要生成大量新代码（每个消息都需要一个独立的编码/解码器）。 关于protobuf协议的更多内容，可以参见其官网[Protobuf项目](https://code.google.com/p/protobuf/)。

#### Pomelo 中的protobuf

原生的带有代码生成器的protobuf过于重量级，缺乏灵活性，任何消息的修改都会是一个非常重量级的操作，而这个在pomelo中应该是经常发生的。而随之带来的大量的代码生成会大大增加客户端的体积和部署难度。因此，我们没有采用生成代码的方式，而是根据proto文件的定义，对消息进行即时的解析。

#### Pomelo protobuf实现

在pomelo中，我们实现了一个通用的protobuf编/解码器，以及一个proto文件解析器。通过分析proto文件内容，实现了对消息的编码/解码。这样，当修改/添加消息类型时，只需要修改对应的proto文件就可以了。具体的运行流程如下图：



从上图可以看出，与原生的protobuf生成代码的方式相比，pomelo中的解决方案要更将灵活，轻量。不需要生成任何代码，在运行时通过proto文件中对消息的定义，实现对消息的动态编码/解码功能。

#### Proto 文件定义

原生的protobuf中，每一个消息与一个proto文件对应，而在生成编码/解码器之后，这个proto文件就不再被使用。 而在pomelo中，因为我们需要proto的内容来动态的对消息进行编码/解码，因此需要维护一个完整的protos信息表，一一对应的方式不但管理困难，也没有必要。因此将所有的proto定义放在一个json文件中，通过一个独立的key来进行区分，在pomelo中，key就是消息的route，我们的protos文件格式如下：

"onMove" : {

"required uInt32 entityId" : 1,

"message Path": {

"required uInt32 x" : 1,

"required uInt32 y" : 2

},

"repeated Path path" : 2,

"required uInt32 speed" : 3

},

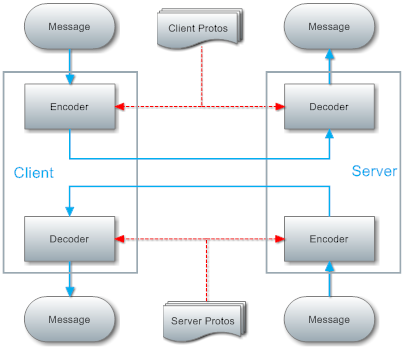
"onAttack" : {

"required uInt32 attacker" : 1,

"required uInt32 target" : 2

}

这里的key就相当于原生protobuf中的proto文件中的消息名称，proto定义具体的语法也是按照protobuf标准来实现的，只是采用了对于js更容易解析的json形式。

在pomelo中，对于同样route的消息，如‘area.playerHander.attack'，在客户端和服务端的格式可能完全不同，这就意味着对于客户端的编码器和解码器对于同样route的消息需要不同的定义。因此，我们需要两套protos文件，server protos和client protos，具体的关系如下图： 

#### 使用Protobuf

虽然protobuf的实现看上去十分复杂，但由于这一层对用户是完全透明的，使用会非常简单。用户只需要通过简单的两步定义就可以在原有的项目中开启protobuf功能。 首先，需要在connector组件上打开protobuf开关，在app.js中的配置如下：

app.set('connectorConfig',

{

connector : pomelo.connectors.hybridconnector,

heartbeat : 3,

useProtobuf : true,

handshake : function(msg, cb){

cb(null, {});

}

});

实际上需要加入的就是“useProtobuf：true”这一项。当设置这一标识后，pomelo会在客户端握手时将protos内容同步到客户端，并默认开启protobuf压缩功能。 在protobuf功能开启用，用户还需要加入protos定义来实现对具体消息的编码/解码。protos文件默认在/game-server/config目录下，包括两个文件：serverProtos.json和clientProtos.json，分别表示服务端->客户端消息的protos和 客户端->服务端消息的protos。只要在其中加入有效的proto定义，就可以开启对应消息的protobuf编码功能。

#### 与老项目的兼容性

Pomelo中的protobuf实现对原有项目是完全兼容的，你可以直接在老的项目中打开protobuf开关而不会引起任何问题。只是当proto定义是空的，默认所有的消息都不会经过protobuf压缩，而是采用默认的二进制编码进行传输。 当你相对某个消息进行protobuf编码时，只需要在对应的protos文件（serverProtos.json或clientProtos）中加入对应的protobuf项，pomelo在启动时就会自动识别并对消息进行压缩，而不会对其他未定义的消息产生任何影响。

# Pomelo 组件

## Pomelo组件

组件（component）是纳入服务器生命周期管理的服务单元。Pomelo服务器启动的过程实际上就是加载和启动一些列组件的过程（更多关于组件的细节请参考[这里](https://github.com/NetEase/pomelo/wiki/Pomelo-Framework#-6)）。Pomelo框架内部提供了一系列的组件，并默认在启动过程中加载并启动它们，以提供Pomelo中所需的各项服务。本文档将介绍这些内置组件的作用，以及开发者如何对这些组件进行配置。但对于一些不稳定的特性和配置选项暂时先隐藏。

Pomelo内置组件的代码放置在[lib/components/](https://github.com/NetEase/pomelo/tree/master/lib/components)目录下。它们当中大部分充当一个包装器角色，将Pomelo框架内部的服务包装后，使之纳入容器的生命周期管理并暴露给外界使用。

## Pomelo组件配置

开发者可以在app.js文件中对各个内置组件进行配置。配置的方式是在app中设置名为componentNameConfig的属性，该属性的value将被作为初始化参数传递给组件。其中componentName 为对应组件的名字。例如，配置connector组件的属性的例子如下：

app.set('connectorConfig', {

connector: pomelo.connectors.hybridconnector,

heartbeat: 3,

useDict: true,

useProtobuf: true

});

注意 :具体的配置参数由对应的组件决定，可能会随组件的升级发生改变。

## 组件说明

### channel

提供channel相关的服务的组件。加载该组件后，会在app上下文加入channelService，可以通过app.get('channelService')获取。channelService相关的接口信息请参考[Pomelo API](http://pomelo.netease.com/api.html)。

#### 配置

* broadcastFilter - broadcast的过滤函数。会在执行broadcast操作时，在每个frontend server上，在将消息发送给每个session之前触发。返回true表示可以将消息发送给该session；false则消息将不会发送给对应的session。

##### 参数

\* session - 消息将发往的session。

\* msg - 待发消息。

\* param - broadcast附带参数，在`channelService.broadcast(type, route, {filterParam: param}, cb)`中传递。

#### 使用例子

app.set('channelConfig', {

broadcastFilter: function(session, msg, param) {

// check some condition

return true;

}

});

### connection

无配置项。提供统计frontend server连接数服务的组件，组件名\_\_connection\_\_，在frontend server上会被加载，主要工作是将connectionService封装成组件。pomelo-admin依赖这个组件进行frontend server的连接数和登录用户数进行统计。

### connector

管理frontend server底层的连接和通信协议的实现，组件名\_\_connector\_\_。配置参数由所使用的connector实现决定。

#### 配置

* connector - connector工厂方法，创建并返回新的connector实例。

##### 参数

\* port - 监听端口

\* host - 监听的host名

\* opts - 额外初始化参数

目前Pomelo提供两个connector的实现：sioconnector和hybridconnector。关于这两个connector的更多信息请参考[Pomelo 0.3新特性](https://github.com/NetEase/pomelo/wiki/Pomelo-0.3%E6%96%B0%E7%89%B9%E6%80%A7#1-)。

### dictionary

生成route字符串压缩编码的组件。该组件会遍历所有handler的route字符串，并为之生成唯一的压缩编码。同时也支持用户配置额外的route字符串，默认配置文件为config/dictionary.json，也可以通过配置来指定该文件的位置。

注意 dictionary组件只有在Pomelo 0.3版本中使用hybridconnector并且配置useDict为true时才有效。

#### 配置

指定用户自定义route字符串配置文件

app.set('dictionaryConfig', {

dict: path.join(app.getBase(), '/config/dictionary.json')

});

### backendSession

无配置项。提供BackendSession相关服务的组件。加载该组件后，会在app上下文中加入backendSessionService，可以通过app.get('backendSessionService')获取。backendSessionService相关的接口信息请参考[Pomelo API](http://pomelo.netease.com/api.html)。

### master

无配置项。在master进程上加载。提供master相关功能，如：根据配置启动各个服务器进程，运行pomelo-adminmaster端并加载master端modules，监控各个进程状态信息等。

### monitor

无配置项。在master之外的各个进程上加载。运行pomelo-adminmonitor端并加载monitor端modules。

### protobuf

无配置项。该组件负责加载protobuf数据定义文件和提供protobuf相关encode和decode服务（内部使用）。

注意 protobuf组件只有在Pomelo 0.3版本中使用hybridconnector并且配置useProtobuf为true时才有效。

具体pomelo-protobuf的使用，请参考[这里](https://github.com/pomelonode/pomelo-protobuf)。

### proxy

提供rpc客户端代理生成和rpc客户端请求路由计算服务。该模块加载后，会在app上下文加入rpc字段。app.rpc是rpc代理对象的根节点，根据各个服务器路径下remote/目录下的服务代码自动生成，可以通过app.rpc.serverType.service.method的形式发起远程调用。

可以通过app.route方法对特定类型的服务器类型配置路由计算函数。路由计算函数的主要工作就是决定某一个消息应该发往哪一个远程服务器。

#### 使用例子

为area服务器配置路由函数routeFn。

app.route('area', routeFn);

配置默认的路由函数，所有发往没配置路由函数的服务器类型的请求都会交给默认路由函数计算。

app.route('default', defaultRouteFn);

路由函数的定义：

var routeFn = function(session, msg, app, cb) {

// 实现路由策略，计算得到目标服务器id

cb(null, serverId);

};

### remote

提供远程服务暴露服务。根据当前服务器类型，加载对应remote/目录下的服务器代码，并根据配置的端口将远程服务暴露出来。该模块启动后，其他服务器即可连接配置的端口，向当前服务器发起rpc调用。

更多关于pomelo-rpc的使用细节，请参考[这里](https://github.com/NetEase/pomelo-rpc)。

### server

server模块使服务器具备处理客户端请求的能力。该模块主要实现了filter服务，根据当前服务器类型，加载对应handler/目录下的代码，并决定一个请求应该是在当前服务器处理还是应该路由给其他服务器处理。

### session

无配置项。提供globalSession相关服务的组件。加载该组件后，会在app上下文中加入sessionService，可以通过app.get('sessionService')获取。sessionService相关的接口信息请参考[Pomelo API](http://pomelo.netease.com/api.html)。

### sync

提供定时同步内存数据到数据库的服务。该组件加载后会在app向下文中加上sync属性，可以通过app.get('sync')来获取。

#### 配置

配置sync组件

app.load(pomelo.sync, {path:\_\_dirname + '/app/dao/mapping', dbclient: dbclient});

其中，path是实现底层数据同步服务器的目录。sync会加载该目录下所有服务，并建立映射关系。dbclient是用来实现数据同步服务的回调参数。

数据同步服务示例

module.exports = {

updateBag: function (dbclient, val, cb) {

var sql = 'update Bag set items = ? where id = ?';

var items = val.items;

if (typeof items !== 'string') {

items = JSON.stringify(items);

}

var args = [items, val.id];

dbclient.query(sql, args, function (err, res) {

if (err) {

console.error('write mysql failed!　' + sql + ' ' + JSON.stringify(val));

}

cb(!!err);

});

}

};

通过sync更新数据

app.get('sync').exec('bagSync.updateBag', player.bag.id, player.bag);

更多关于pomelo-sync的使用细节，请参考[这里](https://github.com/NetEase/pomelo-sync)。

# Pomelo 通讯协议

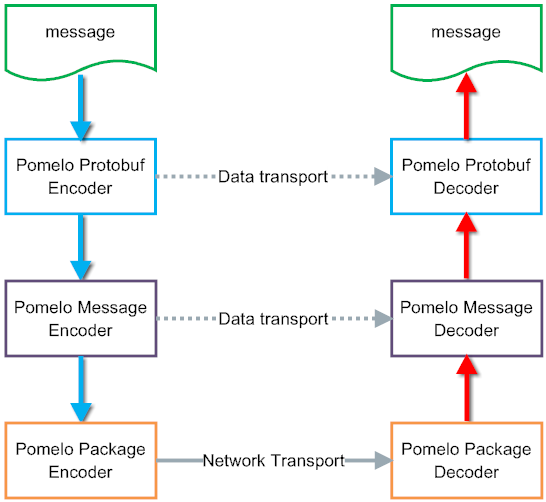
## Pomelo通讯协议

### 背景

0.3版本之前的Pomelo采用json格式的文本作为客户端和服务器的通讯协议，方便协议定制和修改，但同时也带来了较多的通讯冗余数据。Pomelo在0.3版本，底层通讯提供了二进制版本通讯协议的支持，同时提供了route字典压缩和protobuf压缩，提高带宽利用率，以满足诸如移动环境的需求。同时上层接口仍保持json格式的接口，对0.3版本之前的代码不产生任何影响，也保留了开发的便利。

Pomelo的二进制协议包含两层编码：package和message。Message层主要实现route压缩和protobuf压缩，message层的编码结果将传递给package层。Package层主要实现Pomelo应用基于二进制协议的握手过程，心跳和数据传输编码，package层的编码结果可以通过tcp，websocket等协议以二进制数据的形式进行传输。Message层编码可选，也可替换成其他二进制编码格式，都不影响package层编码和发送。

Pomelo协议层的结构如下图所示：

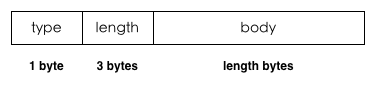


### Pomelo Package

Package协议主要用来封装在面向连接的二进制流的通讯协议（如：tcp）上的Pomelo数据包。Package分为控制包和数据包两种类型。前者用来实现Pomelo应用层面的控制流程，包括客户端和服务器的握手，心跳和服务器主动断开连接的通知等控制信息。后者则是用来在客户端和服务器之间传输应用数据。

#### Package格式

Package分为header和body两部分。Header描述package包的类型和包的长度，body则是需要传输的数据内容。具体格式如下：



* type - package类型，1个byte，取值如下。
  + 0x01: 客户端到服务器的握手请求以及服务器到客户端的握手响应
  + 0x02: 客户端到服务器的握手ack
  + 0x03: 心跳包
  + 0x04: 数据包
  + 0x05: 服务器主动断开连接通知
* length - body内容长度，3个byte的大端整数。
* body - 二进制的传输内容。

各个package类型的具体描述和控制流程如下。

#### 握手

握手流程主要提供一个机会，让客户端和服务器在连接建立后，进行一些初始化的数据交换。交换的数据分为系统和用户两部分。系统部分为Pomelo框架所需信息，用户部分则是用户可以在具体应用中自定义的内容。

握手的内容为utf-8编码的json字符串（不压缩），通过body字段传输。

握手请求：

{

'sys': {

'version': '1.1.1',

'type': 'js-websocket'

},

'user': {

// any customized request data

}

}

* sys.version - 客户端的版本号。每个客户端SDK的每一个版本都有一个固定的版本号。在握手阶段客户端将该版本号上传给服务器，服务器可以由此来判断当前客户端是否合适与服务器通讯。
* sys.type - 客户端的类型。可以通过客户端类型和版本号一起来确定客户端是否合适。

握手响应：

{

'code': 200, // result code

'sys': {

'heartbeat': 3, // heartbeat interval in second

'dict': {}, // route dictionary

'protos': {} // protobuf definition data

},

'user': {

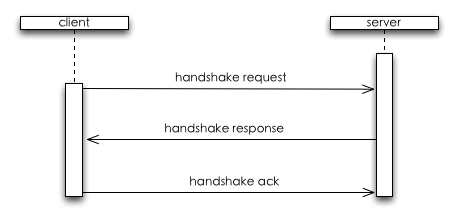
// any customized response data

}

}

* code - 握手响应的状态码。目前的取值：200代表成功，500为处理用户自定义握手流程时失败，501为客户端版本号不符合要求。
* sys.heartbeat - 可选，心跳时间间隔，单位为秒，没指定表示不需要心跳。
* dict - 可选，route字段压缩的映射表，没指定表示没有字典压缩。
* protos - 可选，protobuf压缩的数据定义，没有表示protobuf压缩。
* user - 可选，用户自定义的握手数据，没有表示没有用户自定义的握手数据。

握手的流程如下：

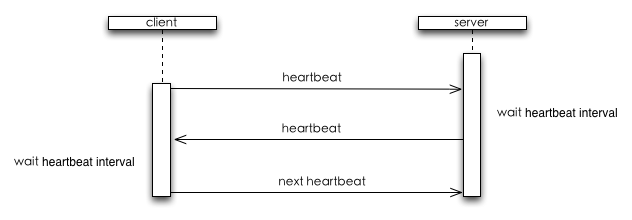


当底层连接建立后，客户端向服务器发起握手请求，并附带必要的数据。服务器检验握手数据后，返回握手响应。如果握手成功，客户端向服务器发送一个握手ack，握手阶段至此成功结束。

#### 心跳

心跳包的length字段为0，body为空。

心跳的流程如下：



服务器可以配置心跳时间间隔。当握手结束后，服务器发起第一个心跳。服务器和客户端收到心跳包后，延迟心跳间隔的时间后再向对方发送一个心跳包。

心跳超时时间为2倍的心跳间隔时间。服务器检测到心跳超时并不会主动断开客户端的连接。客户端检测到心跳超时，可以根据策略选择是否要主动断开连接。

#### 数据

数据包用来在客户端和服务器之间传输数据所用。数据包的body是由上层传下来的任意二进制数据，package层不会对body内容做任何处理。

#### 服务器主动断开

当服务器主动断开客户端连接时（如：踢掉某个在线玩家），会先向客户端发送一个控制消息，然后再断开连接。客户端可以通过这个消息来判断是否是服务器主动断开连接的。

### Pomelo Message

Message 协议的主要作用是封装消息头，包括route和消息类型两部分，不同的消息类型有着不同的消息头。

#### Message 头格式

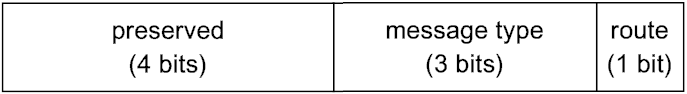
消息头分为三部分，flag，message id，route。如下图所示：



从上图可以看出，Pomelo消息头是可变的，会根据具体的消息类型和内容而改变。其中，flag位是必须的，占用一个byte，它决定了后面的消息类型和内容的格式。message id和route则是可选的。其中Message id采用variant 128变长编码方式，根据值的大小，长度在0～5byte之间。Route则根据消息类型以及内容的大小，长度在0～255byte之间。

##### 标志位Flag

Flag占用Message头的第一个byte，其内容如下

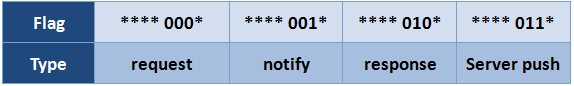


现在只用到了其中的4个bit，这四个bit包括两部分，占用3个bit的message type字段和占用1个bit的route标识，其中：

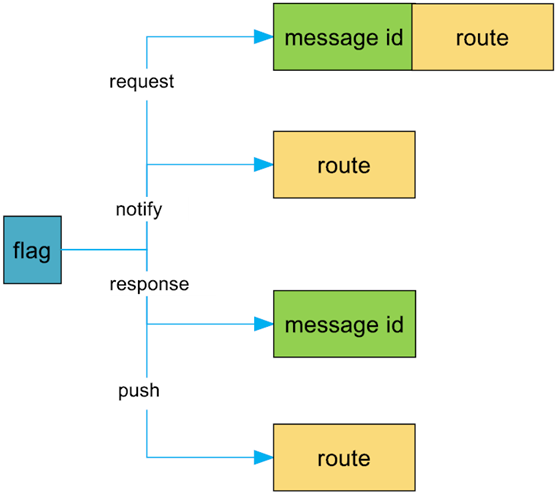
* message type用来标识消息类型,范围为0～7，现在消息共有四类，request，notify，response，push，值的范围是0～3。不同的消息类型有着不同的消息内容，下面会有详细分析。
* 最后一位的route表示route是否压缩，影响route字段的长度。 这两部分之间相互独立，互不影响。

##### 消息类型Message Type 解析

Message type与消息类型的关系如下表：

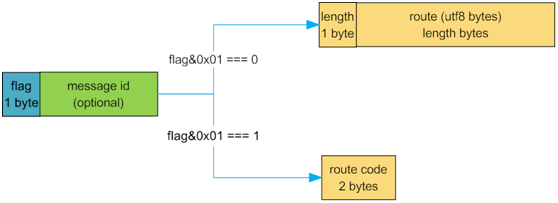


不同的消息类型对应着不同的消息头，具体的对应关系如下图：



##### Route标志位

route主要分为压缩和未压缩两种，由flag的最后一位（route位）指定，当flag中的route标志为0时，表示未压缩的route，为1则表示是压缩route。route通过系统生成和用户自定义的字典进行压缩，具体内容见（pomelo压缩协议-字典压缩）。



上图是不同的flag标志对应的route字段的内容：

* flag的最后一位为1时，后面跟的是一个uInt16表示的route字典编号，需要通过查询字典来获取route。
* flag最后一位为0是，后面第一位uInt8的byte，用来表示route的字节长度。之后是通过utf8编码后的route，其长度就是前面一位byte的uInt8的值。

# PomeloRobot 使用文档

## pomelo-robot

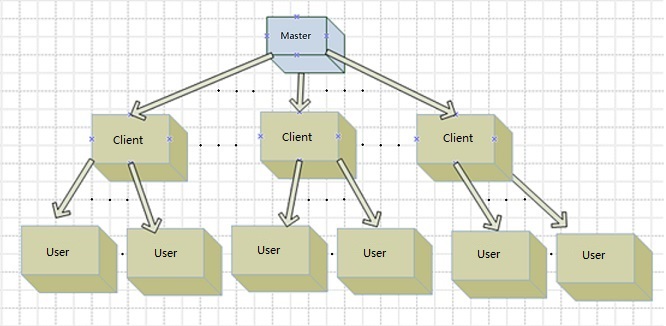
pomelo-robot 是一个用来对pomelo游戏服务器框架进行性能测试的工具, 也可以测试其他基于socket.io服务的性能. 该模块可以采用单机或者分布式测试两种模式.

## 功能

对游戏项目进行自动化的性能测试和分析, 为游戏服务器提供机器人及其运行脚本, 最终输出性能测试分析报告. pomelo-robot 模块通过沙箱的方式执行用户自定义的JS脚本. 运行过程中各客户端会自动把数据汇报给主节点, 主节点把所有节点的测试数据进行提炼汇总, 计算出平均响应时间等统计数据, 然后定时发给内置的HTTP服务器进行界面展示.

## 模块结构

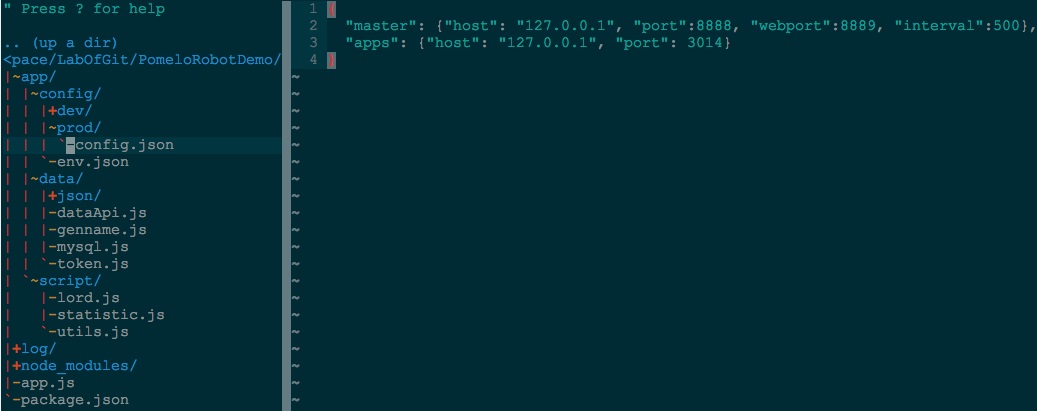
模块内部运行结构如下：



"Master"负责收集所有"Client"的运行数据并展示统计结果.   
"Client"负责在多个沙箱("User")中运行自定义脚本, 同时向"Master"汇报运行数据.

### 使用示例

创建Node.js测试工程, 工程目录结构如下图所示:



安装依赖库:

npm install pomelo-robot

#### config.json配置

"config.json"分为两种运行环境: 开发(dev)或生产(prod)环境, "prod"环境的文件内容如下：

{

"master": {"host": "127.0.0.1", "port":8888, "webport":8889, "interval":500}

}

"master"：主服务器的IP, 与客户端的通讯端口, WEB界面端口, 间隔多少毫秒运行一个自定义JS脚本(如:lord.js).

#### 实现自定义脚本配置

##### 在"env.json"中配置自定义脚本路径:

{

"env": "prod",

"script": "/app/script/lord.js"

}

##### 实现自定义脚本"lord.js":

// ...

var queryHero = require(cwd + '/app/data/mysql').queryHero;

// ...

function entry(host, port, token, callback) {

// ...

// 初始化socketClient

pomelo.init({host: host, port: port, log: true}, function() {

pomelo.request('connector.entryHandler.entry', {token: token}, function(data) {

// ...

afterLogin(pomelo,data);

});

});

}

// ...

"/app/data/mysql.js"中查找登录角色的代码如下：

// ...

queryHero = function(client,limit,offset,cb){

var users = [];

var sql = "SELECT User.\* FROM User,Player where User.id = Player.userId and User.name like 'pomelo%' limit ? offset ? ";

// ...

};

// ...

上面的代码运行在沙箱中: 首先登录游戏服务器, 登录成功后回调"afterLogin"函数, 用户可以进行后续的相关操作.

#### 启动入口文件app.js

"app.js"会根据启动参数判断是启动"master"服务还是"client"服务.

var envConfig = require('./app/config/env.json');

var config = require('./app/config/' + envConfig.env + '/config');

//...

if (mode === 'master') {

robot.runMaster(\_\_filename);

} else {

var script = (process.cwd() + envConfig.script);

robot.runAgent(script);

}

// ...

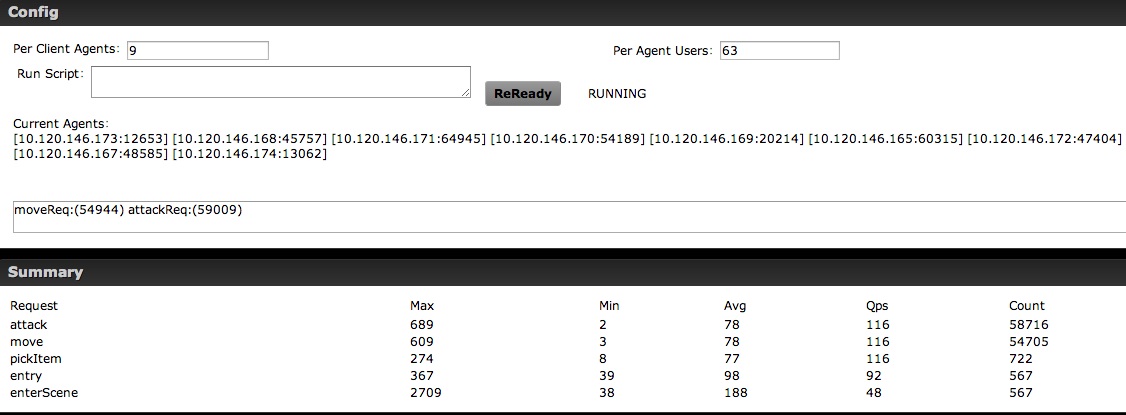
具体代码可参考[工程](https://github.com/NetEase/pomelo-robot-demo).

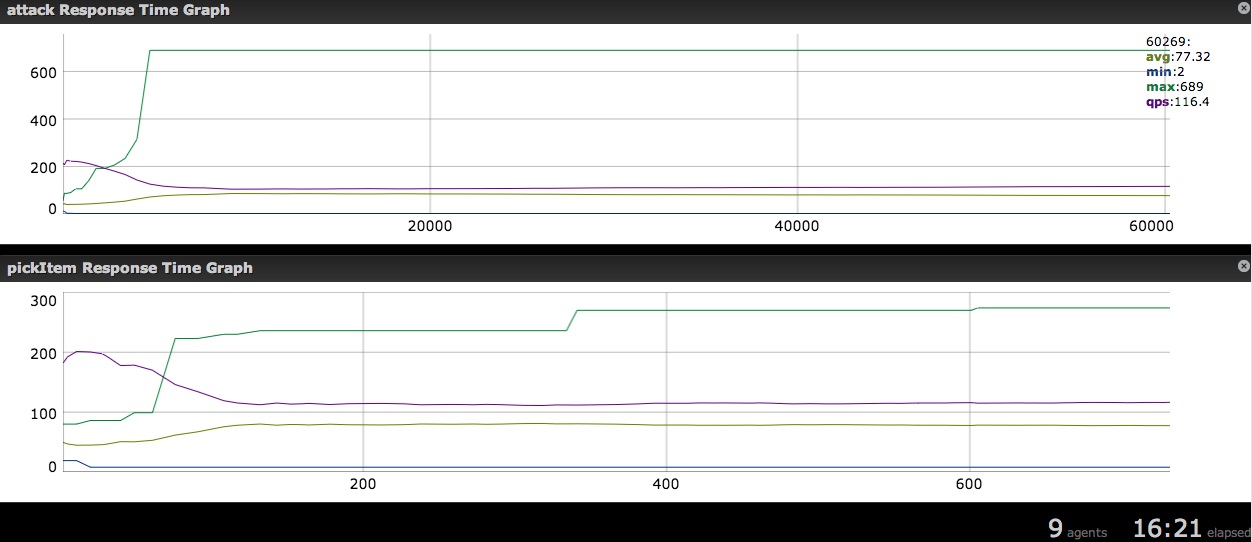
#### 运行测试

运行如下命令, 启动"master"服务:   
"node app.js master"   
打开浏览器访问地址"http://masterIp:8889"

运行如下命令, 启动"client"服务:   
"node app.js client"   
注: 可以在多台机器上启动"client"服务进行性能和压力测试.

WEB界面会显示连接到"master"的"client"数量, 可以在"Per Agent Users"一栏中配置每个"client"将要运行的沙箱数量, 点击"Go"按钮通知所有的"client"开始运行. WEB界面会定时获取后台数据进行展示.   
运行界面如下图所示：





#### 其他

源代码请参考[工程](https://github.com/NetEase/pomelo-robot)

# Pomelo日志管理

## 日志管理

pomelo 日志是通过 [pomelo-logger](https://github.com/NetEase/pomelo-logger) 模块来管理的，[pomelo-logger](https://github.com/NetEase/pomelo-logger) 是对 [log4js](https://github.com/nomiddlename/log4js-node) 的简单封装，并提供了一些非常有用的 feature。

日志是通过 category 来进行管理与维护的，可以在log4js.json文件中进行配置

{

"appenders": [

{

"type": "console"

},

{

"type": "file",

"filename": "./logs/con-log-${opts:serverId}.log",

"pattern": "connector",

"maxLogSize": 1048576,

"layout": {

"type": "basic"

},

"backups": 5,

"category": "con-log"

},

{

"type": "file",

"filename": "./logs/rpc-log-${opts:serverId}.log",

"maxLogSize": 1048576,

"layout": {

"type": "basic"

},

"backups": 5,

"category": "rpc-log"

},

{

"type": "file",

"filename": "./logs/forward-log-${opts:serverId}.log",

"maxLogSize": 1048576,

"layout": {

"type": "basic"

},

"backups": 5,

"category": "forward-log"

},

{

"type": "file",

"filename": "./logs/rpc-debug-${opts:serverId}.log",

"maxLogSize": 1048576,

"layout": {

"type": "basic"

},

"backups": 5,

"category": "rpc-debug"

},

{

"type": "file",

"filename": "./logs/crash.log",

"maxLogSize": 1048576,

"layout": {

"type": "basic"

},

"backups": 5,

"category":"crash-log"

},

{

"type": "file",

"filename": "./logs/admin.log",

"maxLogSize": 1048576,

"layout": {

"type": "basic"

}

,"backups": 5,

"category":"admin-log"

},

{

"type": "file",

"filename": "./logs/pomelo.log",

"maxLogSize": 1048576,

"layout": {

"type": "basic"

}

,"backups": 5,

"category":"pomelo"

}

],

"levels": {

"rpc-log" : "ERROR",

"forward-log": "ERROR"

},

"replaceConsole": true,

"lineDebug": false

}

从配置文件中可以看出，每一项（除了console项）都配了category，pomelo-logger 通过 getLogger 的第一个参数指定 category 来把该logger输出的日志定向到该category配置的文件或者其它输出方案。  
你可以添加自己的category，并在getLogger指定该category，你就可以把日志定向到该category所配的输出方案  
**注意**：不建议使用不指定category的方式来进行配置，这样子所有的logger都会定向到该全局的输出方案

### 日志category

在pomelo中有些指定的category用于输出日志：

* pomelo： 输出 pomelo 框架里的日志
* admin-log： 输出 pomelo-admin 用于监控client登陆master时输出的日志
* crash-log： 输出服务器crash异常时的日志信息
* rpc-debug： 输出 rpc-debug 的日志，需要开启 [rpc-debug 模式](https://github.com/NetEase/pomelo/wiki/pomelo-0.6%E7%89%88%E6%96%B0%E7%89%B9%E6%80%A7#rpc-debug%E6%97%A5%E5%BF%97)
* forward-log : 输出从前端服务器转发到后端服务器的请求日志
* rpc-log: 输出 rpc filter 上的日志
* con-log: 输出 handler filter 上的日志

### 日志levels

可以通过指定日志的levels来控制输出的日志

"levels": {

"rpc-log" : "ERROR",

"forward-log": "ERROR"

}

日志等级从左到右依次提升：

TRACE, DEBUG, INFO, WARN, ERROR, FATAL

在levels上等级配的越低，输出的日志范围则越大  
相反，则输出的日志范围越小  
比如：

var rpc\_logger = require('pomelo-logger').getLogger('rpc-log', \_\_filename);

rpc\_logger.info("msg");

这里rpc\_logger的输出日志等级是 INFO，而 levels 上配的是 ERROR  
那么该日志就不会被输出到对应的appenders上面  
你需要levels改成低于 INFO 的级别，比如 DEBUG 才会把 rpc\_logger 的日志输出

### 日志配置项说明

* type： 指定appenders的类型，可以是console， dataFile， file 等，具体详见 [log4js](https://github.com/nomiddlename/log4js-node/wiki/Appenders)
* filename： 指定输出文件的路径
* pattern：指定输出日志的pattern
* maxLogSize：指定输出日志的最大大小
* layout：指定输出的layout样式
* backups：指定最大输出的文件数目
* category：指定该appender对应的category，如果没有该项，说明该appender是一个全局的appender
* replaceConsole：指定是否替换默认的console
* lineDebug：指定是否开启debug显示日志行数

# Pomelo的master服务器高可用

## Pomelo的master服务器高可用(以LordOfPomelo为例)

### 使用master高可用的方法和步骤

#### 1. 启动和配置zookeeper相关服务

$ zkServer.sh start

* 在lordofpomelo/game-server目录下执行./scripts/createZKMasterhaNode.js或者./scripts/createZKMasterhaNode.js /pomelo/master都会在zookeeper中创建/pomelo/masterznode. 可以使用$ zkCli.sh和[zk: localhost:2181(CONNECTED) 1] ls /pomelo/master来查看.

#### 2. lordofpomelo相关配置

* 在lordofpomelo/game-server目录下执行npm install pomelo-masterha-plugin安装master高可用插件.
* 创建文件lordofpomelo/game-server/config/masterha.json, 文件内容为:

{

"masterha":[

{"id": "master-server-1", "host": "127.0.0.1", "port":3006},

{"id": "master-server-1", "host": "127.0.0.1", "port":3007}

]

}

注: 如果是在[分布式部署的环境](https://github.com/NetEase/pomelo/wiki/Pomelo%E7%9A%84%E5%88%86%E5%B8%83%E5%BC%8F%E9%83%A8%E7%BD%B2%E6%96%B9%E6%B3%95)下使用master高可用, 则上面的host应填写相应机器的IP地址, 如: pomelo16.server.163.org, 注意: 这时不要填写127.0.0.1或者localhost.

* 在文件lordofpomelo/game-server/app.js中添加master高可用相关代码:

...

var masterhaPlugin = require('pomelo-masterha-plugin');

...

// master high availability

app.use(masterhaPlugin, {

zookeeper: {

server: '127.0.0.1:2181',

path: '/pomelo/master'

}

});

...

注: 如果是在[分布式部署的环境](https://github.com/NetEase/pomelo/wiki/Pomelo%E7%9A%84%E5%88%86%E5%B8%83%E5%BC%8F%E9%83%A8%E7%BD%B2%E6%96%B9%E6%B3%95)下使用master高可用, 则上面的server应填写zookeeper服务所在机器的IP地址和端口, 如: pomelo17.server.163.org:2181, 注意: 这时不要填写127.0.0.1:2181或者localhost:2181.

至此, 相关配置就完成了.

#### 3. 启动master高可用服务

* 在目录lordofpomelo/game-server下执行pomelo start -e production启动game-server服务器集群; 在目录lordofpomelo/game-server下执行./scripts/startMasterhaNode.sh启动master高可用热备节点. 可以使用$ zkCli.sh和[zk: localhost:2181(CONNECTED) 2] ls /pomelo/master/lock来查看, 当前应有的3个master节点. 在目录lordofpomelo/web-server下执行node app.js启动web-server, 这时应可以正常登录并进行游戏.

注: ./scripts/startMasterhaNode.sh文件的内容如下:

#!/usr/bin/env bash

pomelo masterha /config/masterha.json

该文件一定要在目录lordofpomelo/game-server下执行.

#### 4. 检验master高可用服务

* 使用$ pomelo-cli和monitor@pomelo : all>show servers来查看master服务器状态, 可以看到当前主master服务器的相关信息, 如master-server-1 master 127.0.0.1 3005 4305 14.55 8.93.
* 我们使用$ kill 4305来kill掉当前的主master服务进程. 切换到启动master高可用服务的终端窗口, 可以看到某个master高可用热备节点被提升为主master服务进程的信息, 如server host: 127.0.0.1, port: 3007 now is promoted to master!. 使用$ zkCli.sh和[zk: localhost:2181(CONNECTED) 3] ls /pomelo/master/lock来查看, 当前应有的2个master节点.
* 使用$ pomelo-cli -P 3007和monitor@pomelo : all>show servers来查看master服务器状态, 可以看到当前主master服务器的相关信息, 如master-server-1 master 127.0.0.1 3007 4421 21.18 34.53. 并且, 此时其它服务进程不受任何应用, 用户仍然可以正常进行游戏.
* 这时可以在目录lordofpomelo/game-server下使用pomelo stop -P 3007来关闭game-server服务器集群.

#### 5. 说明

* 具体代码可以参考[lordofpomelo](https://github.com/NetEase/lordofpomelo)的master分支.
* 该功能需要 [pomelo](https://npmjs.org/package/pomelo)@0.7.2及以上版本, [pomelo-masterha-plugin](https://npmjs.org/package/pomelo-masterha-plugin)@0.0.4及以上版本支持.

# Pomelo的分布式部署方法

## Pomelo的分布式部署(以LordOfPomelo为例)

### 分布式部署的方法和步骤

#### 1. 系统及应用软件环境搭建和配置

所有参与分布式部署的机器:

* 必须为同类操作系统(建议为完全相同的操作系统, 本文所示例的4台机器的操作系统均为"Debian GNU/Linux 7.0").
* 必须都有一个同名的用户(如:"pomelo"等, 本文所示例的4台机器均有一个名为"pomelo"的用户).
* Node.js的安装版本必须完全相同, 安装的绝对路径也必须完全相同(本文所示例的安装绝对路径为"/home/pomelo/node-v0.10.21-linux-x64").
* "lordofpomelo"所放置的绝对路径也必须完全相同(本文所示例的绝对路径为"/home/pomelo/lordofpomelo").
* 在所有参与分布式部署的机器上配置ssh登录选项. 方法为: 在"~/.ssh"目录下创建一个名为"config"的文件(本文所示例的目录为"/home/pomelo/.ssh"), 文件内容如下:

Host \*

HashKnownHosts no

CheckHostIP no

StrictHostKeyChecking no

上述文件的目的是使得各个机器之间可以进行顺畅的ssh登录. 各选项的含义请参考[ssh\_config](http://man.he.net/man5/ssh_config).

#### 2. 全局安装Pomelo, 安装lordofpomelo依赖包

$ npm install pomelo -g

$ cd lordofpomelo

$ sh npm-install.sh

详细的步骤请参考[安装pomelo](https://github.com/NetEase/pomelo/wiki/%E5%AE%89%E8%A3%85pomelo)和[LordOfPomelo-安装指南](https://github.com/NetEase/pomelo/wiki/LordOfPomelo-%E5%AE%89%E8%A3%85%E6%8C%87%E5%8D%97).

#### 3. 修改lordofpomelo中相关配置文件

* 修改"lordofpomelo/shared/config/mysql.json": 将其中的host的地址修改为MySql所在机器的IP地址, 注意: 不要填写"127.0.0.1"或者"localhost". 具体的配置如下所示, 大家可以根据实际情况修改对应配置项:

{

"development": {

"host" : "pomelo3.server.163.org",

"port" : "3306",

"database" : "Pomelo",

"user" : "xy",

"password" : "dev"

},

"production": {

...

}

}

* 修改"lordofpomelo/game-server/config/master.json": 将其中的host的地址修改为master所在机器的IP地址(即, 将要在哪台机器上使用pomelo start来启动game-server服务器集群), 注意: 不要填写"127.0.0.1"或者"localhost". 具体的配置如下所示, 大家可以根据实际情况修改对应配置项:

{

"development":{

"id": "master-server-1", "host": "pomelo16.server.163.org", "port": 3005

},

"production":

{

...

}

}

* 修改"lordofpomelo/game-server/config/servers.json": 将其中的host的地址修改为相应服务进程所在机器的IP地址(即, 将要在哪台机器上运行该服务进程), 注意: 不要填写"127.0.0.1"或者"localhost". 具体的配置如下所示, 大家可以根据实际情况修改对应配置项:

{

"development": {

...

"area": [

{"id": "area-server-1", "host": "pomelo16.server.163.org", "port": 3250, "area": 1},

{"id": "area-server-2", "host": "pomelo18.server.163.org", "port": 3251, "area": 2},

{"id": "area-server-3", "host": "pomelo19.server.163.org", "port": 3252, "area": 3},

...

],

...

"gate": [

{"id": "gate-server-1", "host": "pomelo16.server.163.org", "clientPort": 3014, "frontend": true}

],

...

},

"production": {

...

}

}

* 修改"lordofpomelo/web-server/public/js/config/config.js": 将其中的GATE\_HOST和GATE\_PORT修改为game-server的gate服务进程所在机器的IP地址和端口, 注意: 如果web-server与game-server的gate服务进程在同一台机器上则可将GATE\_HOST配置为window.location.hostname, 否则配置相应的IP; 该配置应与"lordofpomelo/game-server/config/servers.json"中gate的配置相对应. 具体的配置如下所示, 大家可以根据实际情况修改对应配置项:

...

IMAGE\_URL: 'http://pomelo.netease.com/art/',

GATE\_HOST: 'pomelo16.server.163.org',

GATE\_PORT: 3014

...

上述步骤都完成后就可以在master所在机器(本文所示例的是"pomelo16.server.163.org")的lordofpomelo/game-server目录下使用pomelo start命令启动game-server服务器集群; 在lordofpomelo/game-server目录下使用pomelo stop命令停止game-server服务器集群了. 在另外一台机器(本文所示例的是"pomelo17.server.163.org"; 当然也可以和上面的master在同一台机器上)的lordofpomelo/web-server目录下使用命令node app.js来启动web-server; 由于web-server是无状态的web服务器, 则可以通过kill/Ctrl+c来停止.

#### 4. 说明

* 在分布式部署中, 启动/停止各应用服务器的代码可以参考lordofpomelo/game-server/node\_modules/pomelo/lib/master/starter.js中的sshrun函数相关部分.

# Rpc调用原理

在这部分，我们将讨论关于rpc调用相关的问题。在pomelo中rpc的调用主要是通过proxy组件和remote组件实现，其中proxy组件主要负责创建rpc客户端代理，让开发者在pomelo中更方便地进行rpc调用；remote组件主要负责加载rpc服务，包括系统的rpc服务和用户的rpc服务。Pomelo的rpc框架主要解决了两个问题，第一个就是进程间的路由策略，第二个则是rpc底层的通信协议的选择。对于第一个问题，pomelo提供了一套灵活的路由机制，并允许开发者根据需要自由地控制路由信息；对于第二个问题，pomelo现在支持基于socket.io的通信机制和基于原生socket的通信机制。下面我们就分别介绍rpc客户端和服务端的具体实现：

## RPC客户端

rpc客户端主要负责产生代理对象，加载路由策略和进行消息的转发。

### proxy组件

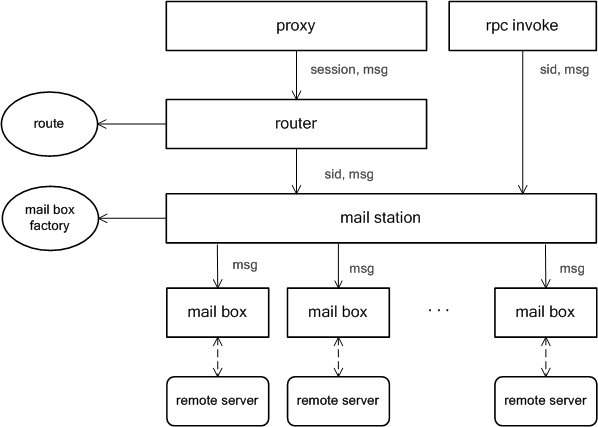
在进行pomelo开发的过程中，进行rpc调用的代码如下：

app.rpc.chat.chatRemote.add(session, uid, serverId, param, cb);

在pomelo中之所以能够如此简洁地进行rpc调用是因为javascript的语言特性和pomelo底层对rpc客户端进行的封装。proxy组件在启动时首先会生成一个rpc client，同时监听系统中服务器增加、服务器移除、服务器替换事件；当这些事件被触发时，proxy组件会根据相应的事件信息对服务器代理对象进行相应的动态变化。例如，当有新的服务器增加时，proxy组件会增加该服务器的代理对象；当有服务器被移除后，proxy组件会移除该服务器的代理对象。在proxy组件启动完成时会将rpc client生成的代理对象挂载到app.rpc下，这样开发者在进行rpc调用时就可以匹配到对应的代理对象，从而通过rpc client进行相应的rpc调用。

### RPC client

对于rpc client，其整体架构图如下所示：



在最底层，使用mail box的抽象隐藏了底层通讯协议的细节。一个mail box对应一个远程服务器的连接。Mail box对上提供了统一的接口，如：连接，发送，关闭等。Mail box内部则可以提供不同的实现，包括底层的传输协议，消息缓冲队列，传输数据的包装等。开发者可以根据实际需要，实现不同的mail box，来满足不同的底层协议的需求。现在pomelo提供基于socket.io的mail box和基于原生socket的mail box，默认使用socket.io。

在mail box上面，是mail station层，负责管理底层所有mail box实例的创建和销毁，以及对上层提供统一的消息分发接口。上层代码只要传递一个目标mail box的id，mail station则可以知道如何通过底层相应的mail box实例将这个消息发送出去。开发者可以给mail station传递一个mail box的工厂方法，即可以定制底层的mail box实例的创建过程了，比如：连接到某个服务器，使用某一类型的mail box，而其他的服务器，则使用另外一个类型的mail box。

再往上的是路由层。路由层的主要工作就是提供消息路由的算法。路由函数是可以从外面定制的，开发者通过注入自定义的路由函数来实现自己的路由策略。每个rpc消息分发前，都会调用路由函数进行路由计算。容器会提供与该rpc相关的玩家会话对象（当中包含了该玩家当前的状态）和rpc的描述消息（包含了rpc的具体信息），通过这两个对象，即可做出路由的决策。路由的结果是目标mail box的id，然后传递给底下的mail station层即可。

最上面的是代理层，其主要作用是隐藏底层rpc调用的细节。Pomelo会根据远程接口生成代理对象，上层代码调用远程对象就像调用本地对象一样。但这里对远程代理对象有两个约定的规则，即第一个参数必须是相关玩家的session对象，如果没有这么一个对象可以填充null，在路由函数中需做特殊处理。还有就是最后一个参数是rpc调用结果的回调函数，调用的错误或是结果全部通过该回调函数返回，且这个参数不能省略。而在远程服务的提供端，方法的声明与代理端的声明相比，除了不需要第一个session参数，其余的参数是一样的。

### rpc请求流程

对于发送rpc请求，rpc客户端采用了一种懒加载的机制，其主要实现思路是客户端与服务端的连接并不是在服务器启动后就创建，而是当客户端第一次向服务端发起rpc请求时才真正建立连接。当客户端与相应的服务端建立连接后，以后有从该客户端到对应服务端的请求就无需再建立连接，消息可以直接发送。消息的发送过程类似前面介绍的handler-filter链处理模式，同样在rpc请求过程开发者可以添加before和after filter对消息进行相应的处理，现在pomelo内建的rpc filter包括rpcLog和toobusy。

## RPC服务端

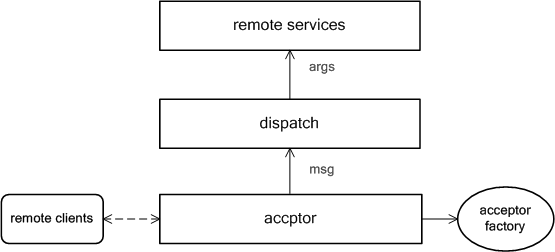
rpc服务端主要负责接收客户端的rpc请求后将相应的消息转给客户端请求的rpc服务中，同时将rpc服务处理完成的消息返回给rpc客户端。

### remote组件

remote组件在启动时会创建一个rpc server，同时加载系统中所有的rpc服务；remote组件在关闭时会停止rpc server的所有服务。

### RPC server

对于rpc server，其整体架构图如下所示：



最底下的是acceptor层，主要负责网络监听，消息的接收和解析。Acceptor层与mail box层相对应，可以看成是网络协议栈上同一层上的两端，即从mail box层传入的消息与acceptor层上传出的消息应该是同样的内容。所以这两端的实例必须一致，使用同样的底层传输协议，对传输的数据使用同样格式进行封装。在客户端替换了mail box的实现，则在服务提供端也必须替换成对应的acceptor实现。同mail box一样，pomelo提供基于socket.io的acceptor和基于原生socket的acceptor。

往上是dispatch层。该层主要完成的工作是根据rpc描述消息将请求分发给上层的远程服务。

最上层的是远程服务层，即提供远程服务业务逻辑的地方，由pomelo框架自动加载remote代码来完成。

## 总结

在本部分，详细介绍了rpc客户端和服务端的通信机制，包括对mail box、mail station、acceptor、gateway的功能进行了阐述，同时分析了pomelo中proxy组件和remote组件的相关功能。

# Protobuf压缩

* 上面我们使用了dictionary的方式对聊天应用中的路由信息进行了压缩，减少了很多通信中的额外开销。在这里，我们将使用pomelo提供的protobuf实现完成通信消息的基于protobuf的压缩。protobuf是google提出的数据交换格式，关于protobuf的更多信息请参阅[这里](https://code.google.com/p/protobuf/)。
* 原始的protobuf，首先需要定义一个.proto文件，然后调用protoc进行编译，根据不同的宿主语言，生成源码，然后将生成的源码应用到具体使用protobuf的应用中。这种使用方式比较笨重，因为涉及到了静态编译，应用程序无法在运行时动态地使用，一旦数据格式有变，就需要修改proto，编译，重新生成源码。
* pomelo的protobuf实现，借助了javascript的动态性，使得应用程序可以在运行时解析proto文件，不需要进行proto文件的编译。pomelo的实现中，为了更方便地解析proto文件，使用了json格式，与原生的proto文件语法是相通的，但是是不相同的。用户定义好客户端以及服务端的通信所需要的信息格式的proto文件，服务端的proto配置放在config/serverProtos.json中，客户端的proto配置放在config/clientProtos.json。如果在其配置文件里，配置了所有类型的proto信息，那么在通信过程中，将会全部使用二进制的方式对消息进行编码; 如果没有定义某一类消息相应的proto，pomelo还是会使用初始的json格式对消息进行编码。

#### chat中使用

下面将pomelo-protobuf应用到我们的聊天应用中，具体的代码在分支tutorial-protobuf中，使用下面命令切换分支：

$ git checkout tutorial-protobuf

* 首先提取所有的数据格式，分为客户端使用的数据格式以及服务器端使用的数据格式，如下：

// clientProtos.json

{

"chat.chatHandler.send": {

"required string rid": 1,

"required string content": 2,

"required string from": 3,

"required string target": 4

},

"connector.entryHandler.enter": {

"required string username": 1,

"required string rid": 2

},

"gate.gateHandler.queryEntry": {

"required string uid": 1

}

}

// serverProtos.json

{

"onChat": {

"required string msg": 1,

"required string from": 2,

"required string target": 3

},

"onLeave": {

"required string user": 1

},

"onAdd": {

"required string user": 1

}

}

* 然后将这两个配置文件分别命名为clientProtos.json和serverProtos.json中，并将这两个配置文件都放到config目录下;
* 在我们的程序中开启protobuf，在app.js的配置中，增加protobuf使用，在配置connector的时候，加入useProtobuf:

app.configure('production|development', 'connector', function() {

app.set('connectorConfig', {

connector: pomelo.connectors.hybridconnector,

heartbeat: 3,

useDict: true,

useProtobuf: true //enable useProtobuf

});

});

app.configure('production|development', 'gate', function(){

app.set('connectorConfig', {

connector : pomelo.connectors.hybridconnector,

useDict: true,

useProtobuf: true //enable useProtobuf

});

});

这样，我们对我们的聊天应用进行了protobuf的压缩。当然，我们这里仅仅是为了示例，实际上，对于onAdd以及onLeave这样的，数据包本身就很小，而且又是字符串，对其使用proto压缩的效果不大，完全没必要进行使用proto压缩，而且使用protobuf压缩会造成编解码的效率开销，得不偿失。实际运用中，还是需要根据实际情况进行合理的选择，更多时候我们是在消息的压缩率和编解码的开销中达到一个平衡。

对于proto文件里面没有配置的通信数据类型，pomelo依然会使用原始的基于json的数据通信格式。

#### 小结

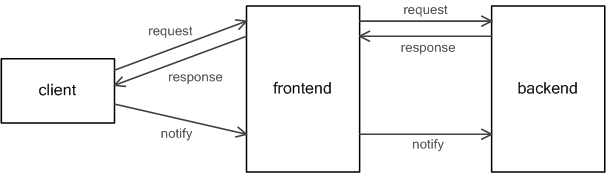
到这里为止，我们已经实现了一个功能基本完善的聊天应用了，我们使用了pomelo提供的filter机制，基于dict的route压缩和基于protobuf的消息压缩。下面将给聊天应用增加一些纯属“画蛇添足”的一些功能，目的是为了继续展示pomelo的特性。下一步，[给聊天应用增加一个rpc调用](https://github.com/NetEase/pomelo/wiki/%E5%A2%9E%E5%8A%A0rpc%E8%B0%83%E7%94%A8)。

# 与客户端通信

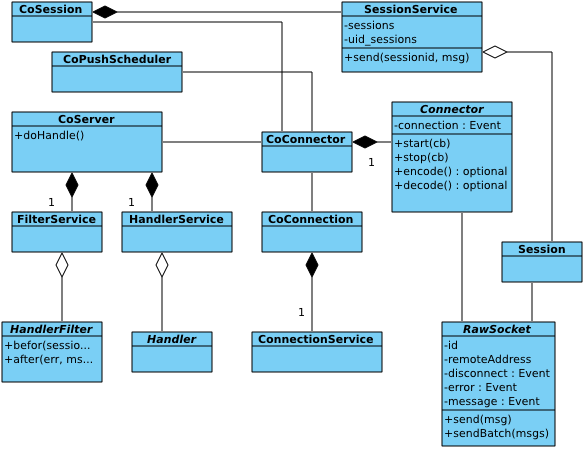
在这部分，主要介绍pomelo是如何与客户端通信以及前端服务器是如何处理用户请求的。处理客户端的请求和响应是pomelo的核心之一，它涉及到了很多组件，包括session组件，server组件，connection组件，connector组件，proxy组件，remote组件等。在本部分，我们仅仅介绍与前端服务器相关的组件，以及他们的作用，对于rpc以及后端服务器以来的backendSession以及channel，这里不做深入介绍，将在其他部分进行介绍。

对于前端服务器来说，session组件是sessionService的包装组件，用来维护用户的session信息；connection组件是connectionService的包装组件，是用来做连接统计的；connector组件会开启监听接口，承受客户端的连接，这里对connector组件底层使用的具体的connector不做太多关注，只关心其抽象行为。

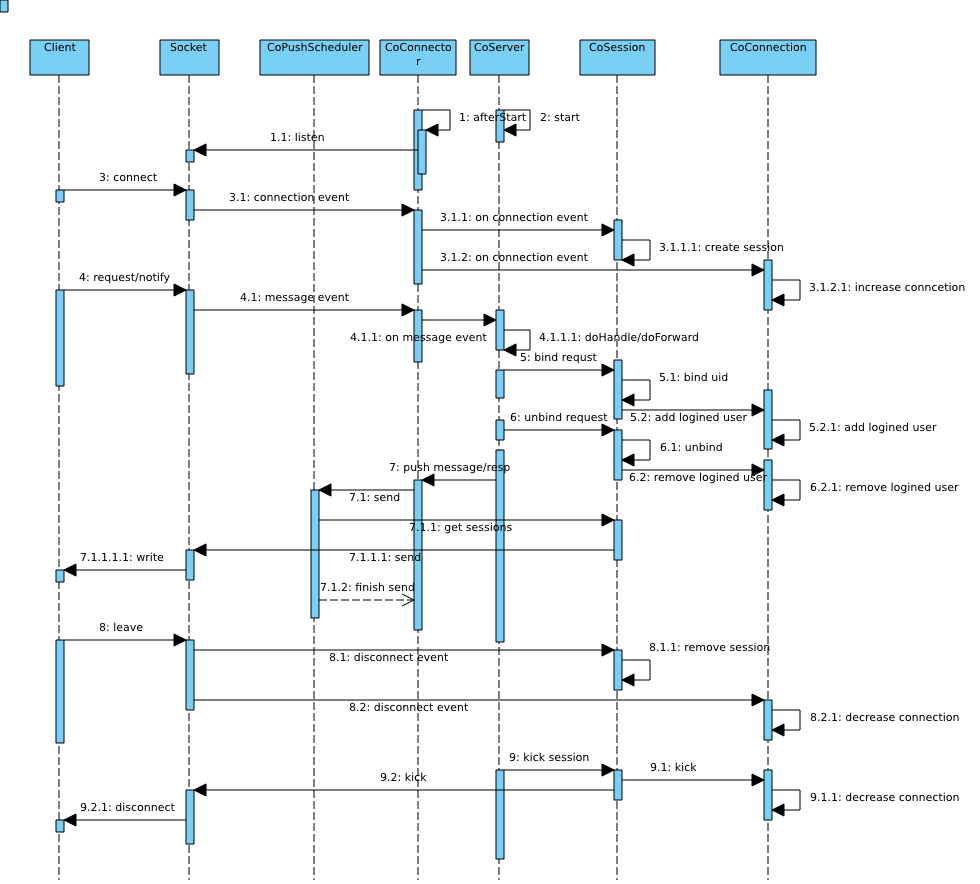
对于server组件来说，会维护服务器的Handler和HandlerFilter，当用户的请求到达前端服务器时，如果前端服务器定义了相应的Handler，那么前端服务器会使用filter-handler链对其进行处理，然后将处理后的结果返回；如果对请求路由检查发现请求是发向到后端服务器的，那么前端服务器会根据用户配置的router（也可能是默认的），计算出要发往的后端服务器id，然后发起rpc调用，后端服务器在接收到rpc调用时，从其中取出请求路由以及请求参数，发起filter-handler链对请求进行处理，完成调用，并将响应发给前端服务器，前端服务器再将响应发送到客户端，整个处理流程如下如所示:



下面的类图粗略地展示了这些类之间的关系:



下面通过类时序图的方式，选取典型的用例行为，来介绍框架的控制流程:



#### 初始化

* 在CoConnector的afterStart回调中，会开启socket的监听，端口使用服务器配置中的clientPort，然后就可以接受用户的连接了，并绑定了connctor的connection事件。
* 在CoServer的start回调里，会扫描当前服务器应该加载的Handler和HandlerFilter，并完成Handler和Filter的加载。此时已经做好了接收客户端连接的准备。

#### 客户端连接

当客户端连接到前端服务器时，会触发connector的监听事件，在事件的处理中，会通过CoConnection增加连接信息，用来做统计。会对连接返回的用来数据通信的socket绑定message，close，error，disconnect等事件，然后创建session，session由CoSession包装的SessionService维护。每一个session都会维护与其相关的socket。此时，客户端已经完成了与服务器端的连接。

#### 客户端请求

* 当客户端连接完成后，客户端就会发起请求，请求会触发socket的message事件，在此事件处理中，首先会对message事件所携带的包进行解包,然后将请求交给CoServer处理。如果请求的是前端服务器的Handler，那么CoServer的doHandle中将会发起其filter-handler链，完成请求的处理，最常见的这种请求就是用户登录请求。如果请求的路由不是前端服务器的，那么CoServer的doFoward将会发起sys rpc给相应的后端rpc。当发起sys rpc调用时，由于同类型的后端服务器一般都有很多，故需要做一个路由选择。这个路由选择策略用户可以配置，通过app.route调用，如果用户不配置的话，pomelo会使用一个默认的路由配置。后端服务器接受到请求后，会执行其CoServer的doHandle，跟前端服务器一样，会使用filter-handler链，对用户的请求进行处理，然后将响应返回给前端服务器，并由前端服务器将响应发送到客户端。
* 前端服务器会调用connector的send函数将响应或者推送的消息发送给客户端，send调用不会直接将要发送的消息通过socket直接发送给客户端，而是将发送任务调度给CoPushScheduler，CoPushScheduler可以实现具体的发送策略。pomelo中提供了两种方式的pushScheduler，direct会立即将用户的响应发送给用户，buffer则会缓冲发送任务，并按时冲刷，pomelo默认使用的是direct的方式，如果想使用buffer的方式，可以通过如下的调用启用:

app.set('pushSchedulerConfig', {scheduler: pomelo.pushSchedulers.buffer, flushInterval: 20});

这里，flushInterval是定时冲刷的周期，我们也可以自己定制实现相应的scheduler，并配置到应用程序中。

* 当服务器的请求处理逻辑需要给客户端推送消息时，会通过用户的uid或者session id从SessionService里获得到对应的Session，session中会维护与客户端用来数据通信的socket，然后将要推送的数据通过session维护的socket连接发送到客户端。

#### 绑定/解绑用户

* 一般来说，当session连接完成后，都会有用户登录的请求，从而完成session与具体用户的绑定。一般在CoServer的处理中，当会将相应的用户绑定到session上，此时会调用CoSession包装的SessionService的操作bind完成对应的用户绑定操作。此外，还会调用CoConnection的addLoginedUser来增加用户，维护统计信息。
* 对于用户注销的请求，一般在CoServer的处理中，会完成session与uid的解绑，此时会调用CoSession包装的SessionService的操作unbind完成对应用户的解绑操作。此外，还会调用CoConnection的removeLoginedUser来减少登录用户，维护统计信息。

#### 客户端断开

当客户端断开连接时，connector监听的socket上会激发disconnect事件，在具体的事件处理中，会从SessionService中删除掉对应的Session，释放掉session维护的连接，还会调用ConnectionService上decreaseConnectionCount，维护统计信息。

上面选取了客户端与服务器交互的几个典型行为，说明了整个客户端请求中的控制流程。这里仅仅涉及到了前端服务器，对于后端服务器的具体处理，这里仅仅提到了会发起rpc调用，而没有具体地深入介绍。

## Pomelo中的请求处理链

在pomelo中，HandlerFilter分为beforeFilter和afterFilter，对于beforeFilter来说,其方法签名为

before(msg, session, next);

其中msg是请求，session表示当前请求的session，在前端服务器的话是FrontendSession，在后端服务器的话是BackendSession，next是用来组成请求链的，是用来指定下一步调用的。如果在具体的filter上没有错误的话，那么就直接调用next(), 否则，则调用next(err, resp),向后面传递具体的处理错误以及响应。在filter的具体实现中，在逻辑处理完后，必须调用next，否则将打断整个处理链。如果有任意一个beforeFilter的next调用中传递了err的话，此处理链将会立即被中断，直接会转入错误处理。在next传递err的时候，可以携带一个resp参数，作为对客户端错误的响应，即next(err, resp)。

具体Handler的签名一般为：

<handler\_Name>(msg, session, next);

msg是经过beforeFilter链处理过的msg，session是经过beforeFilter链处理后的session，next是下一步处理。如果需要给客户端响应的话，没有错误的话，使用next(null, resp),否则可以使用next(err, resp),向后面传递错误信息。这里，resp是给客户端的响应，一般来说客户端的响应都是在Handler的具体逻辑中生成。在具体Handler的实现中，也必须调用next。其next语义与前面的beforeFilter中的next语义一致。

对于afterFilter来说，其方法签名为:

after(err, msg, session, resp, next);

afterFilter是做一些清理操作的，在执行afterFilter链的时候，具体的响应已经发送给了客户端，也就是说在afterFilter如果对resp做更改的话，将对客户端响应没有任何影响。同样，这里的next参数，也是指定了下一步的处理,其签名是next(err),不过与上面beforeFilter和Handler不同的是，由于在afterFilter中常做的是一些清理操作，而且此时具体的响应resp已经发送到了客户端，所以afterFilter中，处理链对err将不再敏感，无论是否有err，整个afterFilter链都会执行完毕。

ErrorHandler，是当在处理请求时产生异常时进行的处理，具体的签名为:

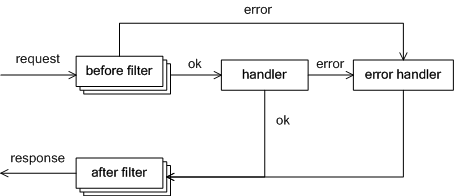
<errorHandler\_Name>(err, msg, resp, session, cb);

在beforeFilter或者Handler中，如果处理产生错误，那么将会转向错误处理，ErrorHandler就是用来进行错误处理的，具体的参数意义跟上面的一样，其中resp是由前面产生错误的next(err, resp)调用传递来的，cb的签名为cb(err, resp),cb会将resp发送给客户端。因此在ErrorHandler里面，是需要调用cb(err, resp), 否则，客户端将得不到服务器端的响应。在errorHandler中可以根据传入的resp以及err信息，重新生成要发送给客户端的resp。通过如下方式设置全局的ErrorHandler，

var errorHandler = require('<path');

app.set('errorHandler', errorHandler);

如果用户没有配置全局的errorHandler的话，默认的errorHandler会向客户端返回由beforeFilters或者Handler产生的resp。整个请求处理链的大致流程如下：



## pomelo内建filter

pomelo内建了常见的一些filter，用户可以通过如下的方式启用:

app.filter(pomelo.filters.<filterName>(<args>));

下面介绍一下这几个fitler:

#### serial

这个filter是用来对用户请求做串行化的，可以使得用户的请求只有在第一个请求被处理完后，才会处理第二个请求。serial中使用了一个taskManager，当用户请求到来时，在beforeFilter中，将用户的请求放到taskManager中，taskManager中维护着一个task队列。在对应的afterFilter中，如果taskManager还有未处理的请求，将会处理其请求，即在一个请求的afterFilter里启动在taskManager中还没处理的下一个请求，这样就实现了请求的序列化。

#### timeout

这个filter是用来对服务端处理超时进行警告的，在beforeFilter中会启动一个定时器，在afterFilter中清除。如果在其定时器时间内，afterFilter被调用，定时器将会被清除，因此不会出现超时警告。如果定时器超时时，afterFilter还没有执行到，则会引发超时警告,并记录日志。默认的处理超时是3秒，可以在加载timeout的时候作为参数传入。

#### time

这个filter使用来记录服务器处理时间的，在beforeFilter中会记录一下当前的时间戳，在afterFilter中再次获取当前的时间戳，然后两个时间戳相减，得到整个处理时间，然后记录日志。

#### toobusy

这个filter中，一旦检测到node.js中事件循环的请求等待队列过长，超过一个阀值时，就会触发toobusy。一旦触发了toobusy，那么toobusy的filter中将终止此请求处理链，并在next调用中，传递错误参数。

## 总结

在本部分，介绍了前端服务器与客户端的通信的相关内容，讲述了相关的类关系，以及典型用例行为的控制流程。对于请求响应链中的before filter，handler，after fitler，error handler等做了较为详细的分析，最后简单分析了一下pomelo内建提供的一些filter。

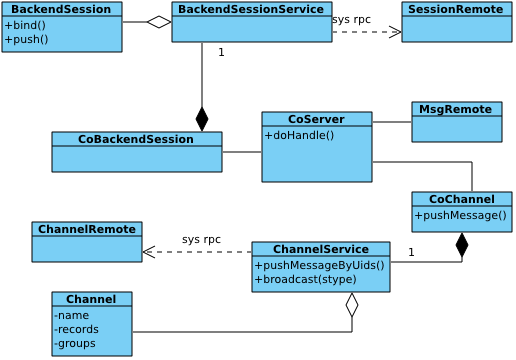
# 后端服务器

在这部分，我们继续讨论与用户请求相关的内容。后端服务器中是用来处理用户请求的具体逻辑的地方，当前端服务器接收到来自客户端的请求时，通过分析请求的路由，并做简单的校验表明路由是合法的，那么前端服务器就会根据路由策略配置，选择某一后端服务器，发起rpc调用。后端服务器的所有调用请求均来自前端服务器的rpc调用。

当后端服务器发起filter-handler链对前端服务器分派过来的请求进行处理时，如果仅仅需要给用户端响应，那么仅仅通过rpc的回调返回具体的响应即可。但是，很多情况下，具体的请求处理逻辑需要给其他用户推送消息。比如，在一个聊天应用中，当有一个用户发起聊天请求时，其聊天的所有内容都需要推送给同一房间的其他用户。当然，消息推送逻辑并不仅仅在后端服务器中使用，前端服务器也可能会有类似的场景。

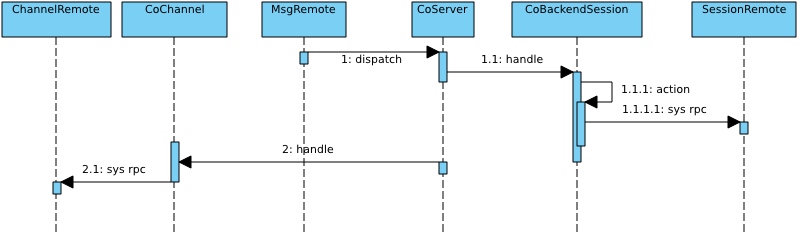
CoBackendSession组件和CoChannel组件一般是用在后端服务器中的，它们一起来完成给特定的用户推送消息。我们知道，BackendSession可以看作前端原始session在后端服务器的一个代理，CoBackendSession包装的BackendSessionService就是用来创建并管理后端的BackendSession，并可以通过相应的bind以及push调用，可以给前端原始的session绑定uid，以及设置一些属性。

CoChannel包装的ChannelService中维护了Channel的信息，每一个Channel可以看作是一系列绑定用户的uid集合，通过Channel的相应调用即可向客户端推送消息。以下是对后端服务器来说，相应的类关系图:



* 后端服务器的所有请求都是从前端服务器的rpc请求中获得的，也就是说后端服务器的CoServer组件的请求是MsgRemote派发的。
* 当前端服务器发出rpc请求时，会携带用来创建BackendSession的信息。在后端服务器中，会创建对应的session信息，这个session就是backendSession，对backendSession所做的任何更改不会影响原始的前端服务器中的session。当遇到用户的登录请求时，可能需要给原始的session绑定uid，并且设定一些自定义属性。以聊天为例，后端服务器处理登录请求时，就需要给session绑定uid，并且给其设置属性room\_id等。这些可以通过使用BackendSession的bind以及push操作。在后端服务器求处理链上的所有session参数，其类型均是BackendSession，对其的直接修改不会直接反映到原始的前端服务器的session上。
* 有时候，需要对用户进行分组，以便更好地推送消息。还以聊天为例，一个聊天室的成员应分为一组，当有人说话时，直接将消息推送到这一组即可。pomelo中的Channel就是应用这种场景的，每一个Channel中维护一个uid列表，当调用Channel的pushMessge方法时，会给所有的在这个Channel中的用户推送消息。
* ChannelService还提供了pushMessageByUids方法，使得推送消息的时候，不用通过Channel，直接传入一个用户列表即可，这样使得消息推送更加灵活。同时，ChannelService还提供了broadcast方法，可以针对某一类型的前端服务器，给其所维护的所有已经绑定uid的session广播消息。
* 以上的对BackendSession以及Channel操作，无论是给session绑定id，还是通过Channel发送消息,还是通过ChannelService进行广播，实际上都涉及到与客户端的通信，由于后端服务器是无法与客户端进行通信的，这些操作实际上都是对前端服务器的rpc调用。因为在前端服务器发起rpc调用给后端服务器派发请求的时候，已经携带了前端服务器id等信息，在BackendSession中会维护此session所在的前端服务器的id，因此，此时后端服务器向前端服务器发起rpc调用时，不再需要路由计算，直接使用相应的frontendId即可。

以下是一些典型用例行为的时序图：



* 上面的图中展示了后端服务器中的调用流程，从MsgRemote获得请求，然后分派给CoServer，Server会发起Filter-Handler链对用户请求进行处理，在Filter-Handler链中的session参数，均为BackendSession。当调用了session的bind，push，kick等操作时，CoBackendSession会向对应的前端服务器发起rpc调用，这个rpc调用由SessionRemote提供服务，完成对应session的bind，push，unbind,kick等操作。
* 如果在Handler-Filter链中处理时需要给用户推送或者广播消息，就可以使用Channel了。可以通过Channel的pushMessage给一个Channel推送消息，也可以使用ChannelService的pushMessageByUids。这些操作实际上也是对前端服务器的rpc调用,为这些操作提供rpc服务的是ChannelRemote。

### 注意事项

* BackendSession是前端服务器中的session在后端服务器中的代理，当后端服务器需要给前端的原始session绑定uid或者设置自定义属性时，需要使用调用bind和push，解绑uid绑定使用unbind。如果仅仅调用了BackendSession的set/get，而没有调用push的话，那么对BackendSession的属性的修改，只在后端服务器的处理链中后面部分有效，而不对其他任何地方的Session产生影响。比如，内建的Filter timeout，在before filter中，开启一个定时器，并把定时器id作为一个属性set到BackendSession中，这个定时器id属性将会在处理请求链的后面部分可以被访问，因此，在after filter中，就通过取得定时器id进行了定时器的清理工作。这种对BackendSession修改仅仅在后端服务器里有效，不会对前端的原始session造成任何影响。
* 对于前端服务器维护的Session信息，可以认为，一个客户端连接就对应一个Session，Session可以看作与客户端连接一一对应。当用户登录的时候，会使用uid绑定对应的session，也可以理解为这个用户通过哪个session进行了登录。在sessionService里有选项singleSession，如果设置为true的话，那就表示一个uid只允许一个session登录，当有新的session建立登录的时候，以前的登录会被踢掉。否则，是允许一个uid绑定多个session的，也就是说一个uid允许维持多个连接。这在实际中是很有意义的，比如，用户的客户端可能有多个设备，那么这样的话，多个设备就可以同时在线。
* 关于Channel，Channel中维护着一组uid，每一个uid会对应多个session，每个session由sessionid以及serverid来指定其前端的连接信息，一个uid可以加入多个Channel中。Channel是后端服务器本地的，也就是说两个后端服务器A和B不会共享Channel信息，当出现跨服务器访问Channel的时候，会出现Channel找不到的错误。当确实需要进行共享Channel信息时，可以考虑使用pomelo提供的global-channel插件，那里使用了redis来维护Channel信息，而不再把Channel信息放在服务器本地，后端服务器通过redis即可查询Channel中的uid信息，然后就可以发起调用了。

### 总结

对客户端请求的处理是pomelo较为复杂的部分，它由pomelo的多个组件共同完成，前端服务器上的CoConnector会加载connector并开启请求监听，当有客户端连接的时候，其对应的连接事件会触发，从而会新的连接创建并维护session，这些操作由CoSession完成。当用户请求具体的服务的时候，前端服务器的CoServer会完成相应的服务器路由，后端服务器的Remote接收到请求后完成请求派发，后端服务器的CoServer会启动Filter-Handler链对请求进行处理，当处理过程中需要给session设置自定义属性以及绑定uid时，可以通过CoBackendSession来完成，当需要给客户端推送消息的时候，可以使用CoChannel提供的功能。当用户的请求通过了Filter-Handler链处理后，对应的响应会通过rpc调用的回调，再次返回到前端服务器的rpc发起者CoServer，然后CoConnector会将后端的响应或者后端推送的消息调度给CoPushScheduler，由CoPushScheduler实现具体的消息发布调度。当可以发布消息的时候，CoPushScheduler会通过CoSession获得到客户端连接的socket，然后通过socket将消息发送出去,完成整个消息处理流程。如果是用户的notify，将不会发送响应。

# 命令行文档

## 命令行文档

Pomelo框架包含一个类似Shell的命令行工具，该工具能够帮助开发者更便捷、更有效率地进行应用开发。该工具包括的命令支持绝大多数的应用开发操作，例如创建项目、启动应用、停止应用、关闭应用等。用户可以通过pomelo --help命令查询相关命令及其使用说明。

## 命令行安装

使用npm(node包管理工具)全局安装pomelo及命令行工具。

npm install pomelo -g

## 命令介绍

* init：创建一个新项目，该项目中包含创建pomelo应用的基本文件及pomelo应用的简单示例。
* start：启动应用及服务器。
* add: 动态增加服务器。
* list: 列出当前应用开启的所有服务器的信息，包括服务器Id、服务器类型、pid、堆使用情况、启动时长。
* stop：关闭应用及服务器或者停止指定的服务器。
* kill：强制关闭应用及服务器。
* --version：列出当前使用pomelo的版本信息。
* --help：列出所有pomelo支持的命令及使用说明。

## 命令使用说明

pomelo init

根据给出的路径或文件名创建新项目，支持相对路径和绝对路径。默认情况下为当前路径，项目名称为当前文件夹名称。

pomelo start [development|production] [--daemon]

应用启动有development和production两种模式，默认以development模式启动。用户可以通过--daemon参数让应用在后台运行，此时服务器的运行日志可以在game-server/logs/目录下查看，应用默认在前台运行。用户可以在/nameofypurproject/game-server/config/servers.json中不同的服务器中添加不同参数。例如：

{"connector":[{"id":"connector-server-1", "host":"127.0.0.1", "port":4050, "wsPort":3050, "args":"--debug=[port] --trace"}]}

如果需要在某台服务器上进行debug,可以在对应的服务器配置中加入--debug=[port]参数，例如要在8080端口开启debug：--debug=8080。其它支持的参数类型可以参考nodejs及v8的命令行参数。该命令需在项目的game-server目录下使用。

应用运行daemon模式运行需要系统安装forever模块，安装命令：npm install forever -g

pomelo add host=[host] port=[port] id=[id] serverType=[serverType]

根据用户给定的配置信息，动态的添加服务器。添加的参数必须包括服务器ip地址(host),服务器端口号(port)，服务器标识(id)，服务器类型(serverType)，其它参数可以根据需要添加。该命令需在master服务器的项目的根目录下使用。

pomelo list

当应用启动后，列出所有服务器信息。该命令需在项目的根目录下使用。

pomelo stop [id]

停止当前应用，优雅地关闭应用。和kill命令不同，这种关闭首先会切断客户端与服务器的连接，然后逐一关闭所有服务器；或者在加入服务器标识(id)，动态停止特定的服务器。该命令需在项目的根目录下使用。

pomelo kill [--force]

强制关闭应用。在本地进行应用开发过程中，如果遇到kill之后还有服务器进程没有关闭的情况，可以使用pomelo kill --force强制关闭所有服务器进程。该操作可能产生数据丢失等不好的影响，所以不推荐在线上使用该命令。该命令需在项目的根目录下使用。

pomelo --version

列出当前使用pomelo的版本信息。该命令可以全局使用。

pomelo --help

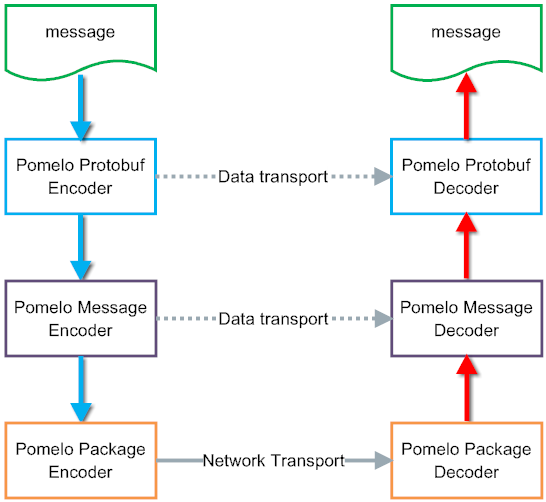
列出所有pomelo支持的命令及使用说明。该命令可以全局使用。

# 协议格式

pomelo核心提供了两种connector，sioconnector和hybridconnector。其中sioconnector基于socket.io，使用json作为其通信格式，hybridconnector则用于tcp/websocket的通信，它底层使用的是二进制协议。虽然在sioconnector中，socket.io的实现很好，对于超时、握手等都做了处理，并且使用json作为通信格式，方便了协议的定制和修改，但同时也带来了较多的通讯冗余数据。hybridconnector则是使用了二进制版本通讯协议，同时提供了route字典压缩和protobuf压缩，提高带宽利用率，以满足诸如移动环境的需求，同时上层接口仍保持json格式的接口，对以前版本之前的代码不产生任何影响，保留兼容性。在本部分，主要介绍hybridconector实现的具体的通信协议。

pomelo的二进制协议包含两层编码：package和message。message层主要实现route压缩和protobuf压缩，message层的编码结果将传递给package层。package层主要实现pomelo应用基于二进制协议的握手过程，心跳和数据传输编码，package层的编码结果可以通过tcp，websocket等协议以二进制数据的形式进行传输。message层编码可选，也可替换成其他二进制编码格式，都不影响package层编码和发送。

Pomelo协议层的结构如下图所示：

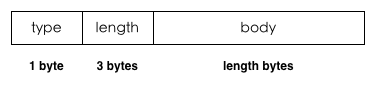


## pomelo package

package协议主要用来封装在面向连接的二进制流的通讯协议（如：tcp）上的pomelo数据包。package分为控制包和数据包两种类型。前者用来实现pomelo应用层面的控制流程，包括客户端和服务器的握手，心跳和服务器主动断开连接的通知等控制信息。后者则是用来在客户端和服务器之间传输应用数据。

#### package格式

package分为header和body两部分。header描述package包的类型和包的长度，body则是需要传输的数据内容。具体格式如下：



* type - package类型，1个byte，取值如下。
  + 0x01: 客户端到服务器的握手请求以及服务器到客户端的握手响应
  + 0x02: 客户端到服务器的握手ack
  + 0x03: 心跳包
  + 0x04: 数据包
  + 0x05: 服务器主动断开连接通知
* length - body内容长度，3个byte的大端整数，因此最大的包长度为2^24个byte。
* body - 二进制的传输内容。

各个package类型的具体描述和控制流程如下。

#### 握手

握手流程主要提供一个机会，让客户端和服务器在连接建立后，进行一些初始化的数据交换。交换的数据分为系统和用户两部分。系统部分为pomelo框架所需信息，用户部分则是用户可以在具体应用中自定义的内容。

握手的内容为utf-8编码的json字符串（不压缩），通过body字段传输。

握手请求：

{

"sys": {

"version": "1.1.1",

"type": "js-websocket"

},

"user": {

// any customized request data

}

}

* sys.version - 客户端的版本号。每个客户端SDK的每一个版本都有一个固定的版本号。在握手阶段客户端将该版本号上传给服务器，服务器可以由此来判断当前客户端是否合适与服务器通讯。
* sys.type - 客户端的类型。可以通过客户端类型和版本号一起来确定客户端是否合适。

握手响应：

{

"code": 200, // response code

"sys": {

"heartbeat": 3, // heartbeat interval in second

"dict": {}, // route dictionary

"protos": {} // protobuf definition data

},

"user": {

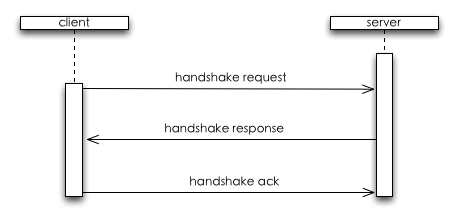
// any customized response data

}

}

* code - 握手响应的状态码。目前的取值：200代表成功，500为处理用户自定义握手流程时失败，501为客户端版本号不符合要求。
* sys.heartbeat - 可选，心跳时间间隔，单位为秒，没指定表示不需要心跳。
* dict - 可选，route字段压缩的映射表，没指定表示没有字典压缩。
* protos - 可选，protobuf压缩的数据定义，没有表示protobuf压缩。
* user - 可选，用户自定义的握手数据，没有表示没有用户自定义的握手数据。

握手的流程如下：

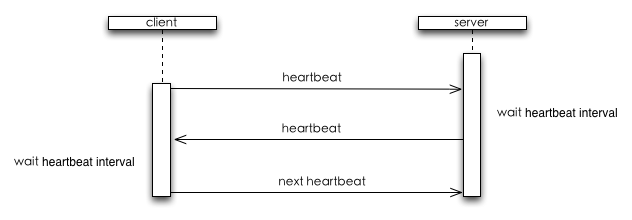


当底层连接建立后，客户端向服务器发起握手请求，并附带必要的数据。服务器检验握手数据后，返回握手响应。如果握手成功，客户端向服务器发送一个握手ack，握手阶段至此成功结束。

#### 心跳

心跳包的length字段为0，body为空。

心跳的流程如下：



服务器可以配置心跳时间间隔。当握手结束后，客户端发起第一个心跳。服务器和客户端收到心跳包后，延迟心跳间隔的时间后再向对方发送一个心跳包。

心跳超时时间为2倍的心跳间隔时间。服务器检测到心跳超时并不会主动断开客户端的连接。客户端检测到心跳超时，可以根据策略选择是否要主动断开连接。

#### 数据

数据包用来在客户端和服务器之间传输数据所用。数据包的body是由上层传下来的任意二进制数据，package层不会对body内容做任何处理。

#### 服务器主动断开

当服务器主动断开客户端连接时（如：踢掉某个在线玩家），会先向客户端发送一个控制消息，然后再断开连接。客户端可以通过这个消息来判断是否是服务器主动断开连接的。

## pomelo message

message协议的主要作用是封装消息头，包括route和消息类型两部分，不同的消息类型有着不同的消息头，在消息头里面可能要打入message id(即requestId)和route信息。由于可能会有route压缩，而且对于服务端push的消息，message id为空，对于客户端请求的响应，route为空，因此message的头格式比较复杂。

消息头分为三部分，flag，message id，route。如下图所示：

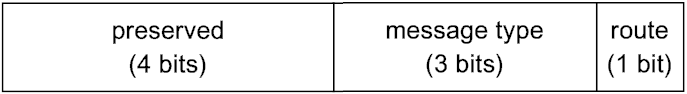


从上图可以看出，pomelo消息头是可变的，会根据具体的消息类型和内容而改变。其中：

* flag位是必须的，占用一个byte，它决定了后面的消息类型和内容的格式;
* message id和route则是可选的。其中message id采用[varints 128变长编码](https://developers.google.com/protocol-buffers/docs/encoding#varints)方式，根据值的大小，长度在0～5byte之间。route则根据消息类型以及内容的大小，长度在0～255byte之间。

### 标志位flag

flag占用message头的第一个byte，其内容如下

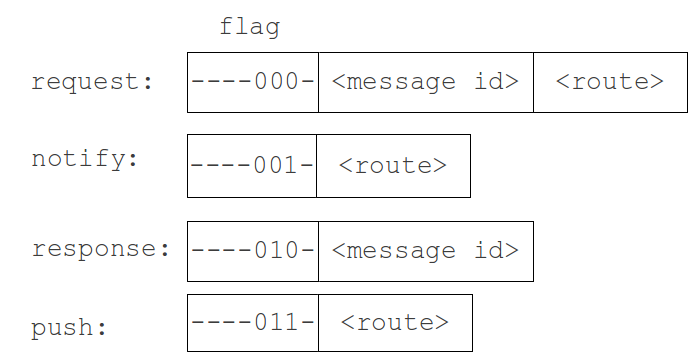


现在只用到了其中的4个bit，这四个bit包括两部分，占用3个bit的message type字段和占用1个bit的route标识，其中：

* message type用来标识消息类型,范围为0～7，现在消息共有四类，request，notify，response，push，值的范围是0～3。不同的消息类型有着不同的消息内容，下面会有详细分析。
* 最后一位的route表示route是否压缩，影响route字段的长度。 这两部分之间相互独立，互不影响。

### 消息类型

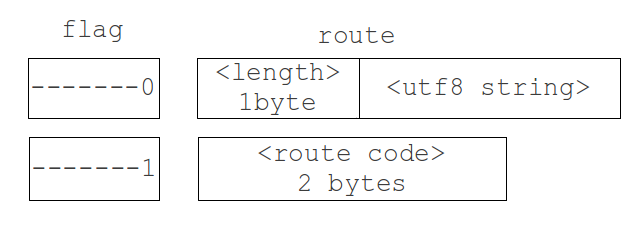
不同类型的消息，对应不同消息头，消息类型通过flag字段的第2-4位来确定，其对应关系以及相应的消息头如下图：



上面的 **-** 表示不影响消息类型的bit位。

### route压缩标志位

route主要分为压缩和未压缩两种，由flag的最后一位（route压缩标志位）指定，当flag中的route标志为0时，表示未压缩的route，为1则表示是压缩route。route通过系统生成和用户自定义的字典进行压缩，具体内容见[pomelo压缩协议](https://github.com/NetEase/pomelo/wiki/%E6%B6%88%E6%81%AF%E5%8E%8B%E7%BC%A9)。route字段的编码会依赖flag的这一位，其格式如下图:



上图是不同的flag标志对应的route字段的内容：

* flag的最后一位为1时，后面跟的是一个uInt16表示的route字典编号，需要通过查询字典来获取route;
* flag最后一位为0是，后面route则由一个uInt8的byte，用来表示route的字节长度。之后是通过utf8编码后的route字符串，其长度就是前面一位byte的uInt8的值，因此route的长度最大支持256B。

### 总结

在本部分，介绍了pomelo提供的hybridconnector的线上协议，包括package层和message层。当用户使用hybridconnector的时候，可以根据这里提供的协议信息，在客户端可以依据此协议完成与服务端的通信。

# 应用程序配置

在这部分，我们来介绍如何配置框架。我们知道在pomelo中，可以配置各个组件的选项，加载配置文件，开启pomelo的特性等等。这一切配置都是在game-server/app.js中进行的。实际上，在pomelo的应用中有两个app.js,一个在game-server目录下，一个在web-server目录下。其中game-server下的app.js是整个游戏服务器的入口和配置点，而web-server下的app.js则是web服务器入口。在这里，我们仅仅介绍如何在game-server/app.js中配置框架以及pomelo框架使用的服务器配置文件的格式。

### app.js文件

app.js是运行pomelo项目的入口，在app.js中，首先会创建一个app的实例，这个app作为整个框架的配置上下文来使用,用户可以通过这个上下文，设置一些全局变量，加载配置信息等等操作。app.js中的一般代码如下：

var pomelo = require('pomelo');

var app = pomelo.createApp();

// some configuration

app.configure(<env>, <serverType>, function() {

});

app.configure(....);

app.set(...);

app.route(...);

// ...

// start app

app.start();

首先会创建一个app实例，然后是一些通过app这个上下文对框架的一些配置以及一些初始化操作，最后启动应用。这里我们将主要关注对框架的配置部分。

### 使用app.configure调用来配置

服务器的配置主要由configure()方法完成，完整的app.configure配置参数如下：

app.configure([env], [serverType], [function]);

前两个参数是可选的， 以下是参数说明：

* env: 运行环境， 可设成development, production或development|production
* serverType: 服务器类型，设置了这个参数只会对当前参数类型服务器做初始化，不设置则对所有服务器执行初始化function
* function: 具体的初始化操作， 内部可以写任何对框架的配置操作逻辑。

以下是一些配置实例：

#### 实例一

app.configure(function(){

// do some configuration

});

这种配置将对所有模式(development/production)下的所有服务器生效，它等价于在app.js中，不调用configure，直接执行相关的配置，代码示例如下：

app.configure(function() {

doSomeConfiguration();

});

// <==>

doSomeConfiguration(); // equivalent to above `app.configure`

#### 实例二

app.configure('development', function(){

// do some configuration just for development env only.

});

这种配置则只针对development模式下所有服务器生效，同样在这里可以填入任何配置。

#### 实例三

app.configure('development', 'chat', function(){

// do some configuration just for development env and chat server only.

});

这种配置则针对development模式下的chat服务器生效，这里同样可以填入任何配置。

#### 配置内容

在configure中用户可以根据应用的不同需求在不同的服务器中进行相关配置，例如在全局设置mysql参数：

app.configure('development|production', function(){

app.loadConfig('mysql', app.getBase() + '/config/mysql.json');

});

另外也可以选择在具体的服务器中进行应用的配置，例如可以做一些初始化操作：

var initArea = function(){

//area init

};

app.configure('development|production', 'area', function(){

initArea();

});

而更多地，可以在configure中，针对不同的服务器，不同的环境，对框架进行不同的配置。这些配置包括设置一个上下文变量供应用使用，开启一些功能选项，配置加载一个自定义的component，针对不同的服务器，配置filter等等配置操作，如下所示:

app.configure('development|production', 'chat', function() {

app.route('chat', routeUtil.chat);

});

app.enable('systemMonitor');

app.configure('development|production','gate', function() {

app.set('connectorConfig', {

connector: pomelo.connectors.hybridconnector,

heartbeat: 3

});

}); // configure connector for gate server

下面就对这些框架配置作些介绍。

### 上下文变量存取

上下文对象app提供了设置和获取应用变量的方法，其签名为：

app.set(name, value, [isAttach]);

app.get(name);

* 如上，对于set来说，有三个参数，分别是变量名，变量的值以及一个可选的参数isAttach。如果isAttach设置为true的话，那么表示将变量attach到app对象上，作为app的一个属性，以后对此变量的访问，可以直接通过app.name进行访问，这个参数默认为false。
* 对于get调用，就是一个很简单的通过变量名获取变量值。

示例代码如下:

app.set('server',server);

var server = app.get('server');

app.set('service', service, true);

var service = app.service;

在pomelo中框架中，可以通过app.set给pomelo的组件配置相应的选项，也可以通过app.get获得到pomelo框架加载的服务，如backendSessionService，channelService等等，示例代码如下:

app.set('connectorConfig', {

// ...

}); // set opts for connector component

var backendSessionService = app.get('backendSessionService'); // get backendSessionService instance

如果用户需要自己设置一些自己的自定义变量，也可以通过app这个上下文实现获取和设置。

### 开启和关闭功能选项

应用的功能选项配置可以通过enable和disable来打开和关闭。另外，用户可以通过enabled和disabled对相应的状态进行判断，如果该状态存在则返回true,反之返回false。例如要打开或者关闭应用的rpc debug log并查看其状态是否存在, 示例代码如下：

app.enable('rpcDebugLog');

app.enabled('rpcDebugLog'); // return true

app.disable('rpcDebugLog');

app.disabled('rpcDebugLog'); //return true

在pomelo框架中，当需要做更详细的监控管理的时候，可以打开systemMonitor选项，打开systemMonitor选项会使得默认加载额外的admin-module，示例代码如下：

app.enable('systemMonitor'); // enable system monitor

同样，用户可以设置自己应用的一些功能选项，并通过enable，disable，enabled，disabled来进行开启关闭以及检查。

### 加载配置文件

用户可以通过loadConfig加载配置文件，加载后文件中的参数将直接挂载到app对象上。例如需要加载mysql.json文件，示例代码如下：

{

"development":

{

"host":"127.0.0.1",

"port":"3306",

"database":"pomelo"

}

}

然后，加载完成后，就可以直接通过app对象，访问具体的配置参数，示例代码如下：

app.loadConfig('mysql', path.resolve('./config/mysql.json'));

var host = app.get('mysql').host; //返回 127.0.0.1

当然，用户可以使用loadConfig的调用加载任何json格式的配置文件，用于其他的目的，并能通过app进行访问。需要注意的是所有的json配置文件中都要指定具体的模式，也就是development或者production。

### 加载component

pomelo的功能由其component提供，pomelo会默认根据服务器的类型加载不同的内建组件，另外用户可以根据应用需求自定义组件。组件的加载主要是使用load方法，示例代码如下：

app.load(HelloWorldComponent, [opts]); //opts is optional

### 加载plugin

pomelo还可以加载自定义的插件，一个插件由多个component和一组对应用的事件进行响应的事件处理组成，加载插件使用app.use, 示例代码如下：

// app.use(<plugin>, <plugin options>);

var statusPlugin = require('pomelo-status-plugin');

app.use(statusPlugin, {

status:{

host: '127.0.0.1',

port: 6379

}

});

### 配置router

router主要负责请求路由信息的维护，路由计算，路由结果缓存等工作，并能根据需要切换路由策略，更新路由信息等。用户可以自定义不同服务器的不同路由规则，然后进行配置即可。以下示例为chat服务器配置路由规则：

//routeUtil.js

app.route('chat', routeUtil.chat);

在routerUtil中可以具体的定义不同服务器的路由规则，例如：

routeUtil.chat = function(session, msg, app, callback) {

var chatServers = app.getServersByType('chat');

if (!chatServers) {

callback(new Error('can not find chat servers.'));

return;

}

var server = dispatcher.dispatch(session.rid, chatServers);

callback(null, server.id);

};

在路由函数中，通过最后的回调函数中返回服务器的id即可，这里使用dispatcher对session.rid进行hash处理从而完成服务器选择。用户可以根据自己的实际需求进行配置相应的router。

### 配置filter

当一个客户端请求到达服务器后，经过filter链和handler处理，最后生成响应返回给客户端。handler是业务逻辑实现的地方，filter则是执行业务前进行预处理和业务处理后清理的地方。为了开发者方便，系统内建提供了一些filter，例如：serialFilter,timeFilter，timeOutFilter，另外用户可以根据应用的需要自定义filter。配置filter的调用示例如下：

app.filter(pomelo.filters.serial()); // use builtin filter: serial filter

app.filter(FooFilter); // use FooFilter as a before & after filter

app.before(beforeFilter); // use beforeFilter as a before filter

app.after(afterFilter); // use afterFilter as an after filter

用户可以自定义自己的filter，然后通过app.filter调用，将其配置进框架。如果仅仅是before filter，那么就调用app.before，如果是after filter，就掉用app.after，如果既定义了before filter又定义了after filter，那么就可以使用app.filter调用了。

### 配置admin-module

pomelo 提供了监控管理框架，可以给其配置不同的admin module，具体的配置使用调用app.registerAdmin，示例代码如下：

app.registerAdmin(require('../modules/watchdog'), {app: app, master: true});

用户也可以自定义自己的module，然后通过registerAdmin调用，加载到框架。

### 服务器配置文件格式

pomelo的配置文件都在game-server/config目录下，其中有两个很重要的服务器配置文件servers.json和master.json。 所有的配置文件中，都分为development和production两种配置模式，以对应具体的是开发调试环境还是具体的产品环境，其配置字段如下:

master.json中:

* id: master 服务器的服务器id，是一个字符串;
* host: master 服务器的host，可以是一个ip或域名;
* port: master服务器开的端口，默认是3005;
* args: 可选的，在这里配置的参数选项，是用来给node/v8使用的，如你可以配置"args": "--debug=5858",这样的话，就可以开启调试。

servers.json中:

* id: 应用服务器的服务器id，是一个字符串;
* host: 应用服务器的host，可以是一个ip或域名;
* port: 接受rpc请求时使用的端口号，对于后端服务器来说必须的，对于前端服务器来说，如果仅仅像gate服务器那样，并不维护具体的客户端连接，仅仅给客户端返回可以连接的前端服务器的地址信息的话，port则是可以省略的。一般来说，port不应该省略;
* frontend: 是一个boolean值，对于前端服务器，配置为true，如果省略的话，则默认为false。后端服务器不配置;
* clientPort: 这是前端服务器接受客户端连接启用的端口，对于后端服务器来说，不需要配置，对于前端服务器，则是必需的。
* max-connections: 可选的，用来说明前端服务器最多承受的连接数，如果超过了这个连接数，则后来的连接将会会connector拒绝;
* args: 可选的，同master中的语义一样。

### 总结

在这部分，介绍了在app.js如何配置整个框架并在最后给出了服务器配置文件的格式。通过application的configure等调用，可以给不同的服务器完成不同的配置，比如，配置router，配置filter，为特定类型的服务器加载自定义的component等等。同样，在这里，也可以做一些初始化的加载操作，比如，当应用需要数据库时，可以加载mysql的配置文件，并将配置信息设置到app上下文中，这样在应用中，就可以通过app直接获取到对应的配置信息。

# 增加admin module

一个pomelo的应用，一般是由一个服务器群来支持，对于这些应用服务器的管理以及监控就显得尤为重要。比如监控这些应用服务器的进程状态，系统状态，杀死某个服务器等等。

对服务器的监控和管理有三个主体：master，monitor，client。服务器的管理和监控由master服务器加载的master component和普通的应用服务器加载的monitor component，还有服务器管理客户端共同完成，下面的叙述中将不加区分地使用monitor与应用服务器，master与master服务器。

master负责收集所有服务器的信息，下发对服务器的操作指令。monitor负责上报服务器状态，并对master的命令作出反应。client是第三方监视的客户端，它注册到master上，通过给master发请求获得服务器群信息，或者给master发指令，管理操作应用服务器群。pomelo中内建实现并使用了console和watchdog这两个admin module，它们是pomelo核心的一部分,这里不再详述。

由于对于具体的应用来说，需要监控和管理的信息也是各不相同的，因此，pomelo并没有实现固定的监控模块，而是提供了一个可插拔的监控框架机制，用户只需要定义一个监控模块所需要的回调方法，并完成相应的配置即可。

一组相关的供不同主体调用的回调函数构成一个admin module，一个admin module中一般包括四个回调方法，monitorHandler，masterHandler，clientHandler, start。其中monitorHandler是monitor收到master的请求或者通知时由monitor回调，masterHandler是master收到monitor的请求或者通知时回调，clientHandler是master收到client的请求或通知时回调的, start是当admin module加载完成后，用来执行一些初始化监控时调用。

为了演示admin module的用法，我们将给聊天应用增加一个监控模块，我们让monitor每隔5秒钟向master上报一下自己的当前时间。当然，上报时间没有太多的实际意义，不过为了保持示例的简单化，选择上报时间还是可取的。实际使用中，可以上报任何信息，使用方式都是与上报时间的方式是一样的，这里使用上报时间仅仅是为了使得示例尽可能简单，更容易抓住如何使用admin module。

### chat中使用

下面我们将给我们的聊天应用增加一个监控管理模块，具体的代码在分支tutorial-admin-module上，使用如下命令切换分支：

$ git checkout tutorial-admin-module

* 首先，我们在app目录下建立文件modules/timeReport.js, 在其中定义monitorHandler，masterHandler和clientHandler，代码如下：

module.exports = function(opts) {

return new Module(opts);

}

var moduleId = "timeReport";

module.exports.moduleId = moduleId;

var Module = function(opts) {

this.app = opts.app;

this.type = opts.type || 'pull';

this.interval = opts.interval || 5;

}

Module.prototype.monitorHandler = function(agent, msg, cb) {

console.log(this.app.getServerId() + ' ' + msg);

var serverId = agent.id;

var time = new Date().toString();

agent.notify(moduleId, {serverId: serverId, time: time});

};

Module.prototype.masterHandler = function(agent, msg) {

if (!msg) {

var testMsg = 'testMsg';

agent.notifyAll(moduleId, testMsg);

return;

}

console.log(msg);

var timeData = agent.get(moduleId);

if (!timeData) {

timeData = {};

agent.set(moduleId, timeData);

}

timeData[msg.serverId] = msg.time;

};

Module.prototype.clientHandler = function(agent, msg, cb) {

cb(null, agent.get(moduleId));

}

* 这里我们没有定义start回调，因为我们这里用不到。在定义完上面的admin module后，需要将其注册到我们的应用中，使用Application.registerAdmin调用，在app.js中增加如下代码：

var timeReport = require('./app/modules/timeReport');

app.registerAdmin(timeReport, {app: app});

这里registerAdmin可以接收两个或三个参数，如果是三个参数的话，第一个必须是字符串来指定moduleId。如果是两个参数的话，moduleId将使用第一个参数，也就是module的工厂函数的moduleId属性。这里由于我们给timeReport定义了moduleId属性，因此我们就省略掉了第一个moduleId参数了。最后一个参数是配置选项，可以配置监控数据获取是pull还是push方式，以及获取周期。在我们这个例子中，由于注册时没有传入任何关于type和interval的配置，将使用默认值，也就是使用拉模式，每隔5秒获取一次数据。

### 一些说明

* 在导出一个module的时候，一般需要指定一个moduleId，在这里，我们指定的moduleId是timeReport。当然我们如果这里不指定moduleId的话，在调用Application.registerAdmin的时候再指定moduleId也是可以的。
* 一个module有两个属性很重要，type和interval，type指出的是数据所采用的方式，有两种pull和push。pull方式是让master定时给monitor发请求，monitor给其上报信息。push的方式则是monitor定时上报自己的信息。interval就是这个信息上报的时间周期了。我们例子中使用的是方式通过opts传入，如果opts中没有配置的话，默认使用pull方式，上报周期为5秒，而实际上，我们就是使用了这样的两个参数值，即使用pull方式，让master主动拉数据，每5秒拉一次。
* 还有一个要注意的地方是masterHandler的实现，可能会让人感到迷惑。实际上，由于使用pull的方式，masterHandler会在两种情况下被回调，一种是每隔5秒产生的一次拉数据事件，一种是monitor向master上报信息。这两种情况，可以通过参数msg区分。
  + 如果是定时器产生的周期性的拉数据事件导致的回调，此时msg参数是undefined，因此此时只是简单的调用notifyAll，参数moduleId使用来区分到底是哪个监控模块；testMsg参数在这里仅仅用来示例如何传参,在monitorHandler中也仅仅把其打印到console上而已，实际应用中，可以用其传递更有意义的参数；
  + 如果是monitor在收到master的通知后，上报自己的时间信息的话，此时msg将会是一个对象，这个时候，master将这个时间值打印到console，并缓存其值，当然这个值没什么意义，仅仅是为了示例。因此这段代码通过对msg的判断区分了这两种情况。
  + 实际应用中，也经常使用判断msg来区分两种情况的方式。考虑另一种情况，假如使用的不是pull方式，而是push方式的话，那么monitor将会遇到两种情况，与master类似，一种是定时器的周期事件，一种是master给其发了通知或请求，此时也可以通过判断msg进行两种情况的区分，只不过此时将会在monitorHandler中进行判断了。关于这种使用push方式并在monitorHandler中通过判断msg的值进行区分两种情况的实现方式，读者可以自行尝试。
* monitorHandler的实现中，当收到master的通知后，取出了master传来的参数，这里的参数就是testMsg，实际应用中可以使用更复杂的更有实际意义的参数。然后通过对参数进行分析，执行相应的逻辑。这里的逻辑很简单，就是获取自己当前的时间，然后通知给master。
* clientHandler是当有第三方监控客户端给master发请求时，由master进行回调的。为了保持简单，我们这里不再对client做过多的介绍,在开发指南部分会有详细的介绍。

### 小结

在这部分里，我们使用了pomelo提供的监控管理框架完成了monitor向master上报其本地时间的功能。实际上，通过定制自己的admin module可以实现上报任何我们需要的上报的数据。比如，在实际应用中，connector服务器可以向master报告登录到其服务器上的用户信息，monitor可以向master上报其进程相关的信息等等。在pomelo-admin中还实现了另外几个admin module，这些admin module可以通过对Application调用app.enable('systemMonitor')完成开启，这里不再详述，可以直接阅读相关代码。到此为止，我们基本上就介绍完了pomelo的所有基本功能，下面会有一个简单的[总结还有一些没有涉及到的内容](https://github.com/NetEase/pomelo/wiki/%E6%80%BB%E7%BB%93)。

# 增加filter

在实际的应用中，我们往往需要在逻辑服务器处理请求之前需要对用户请求做一些前置处理，而当请求被处理后，又需要做一些善后处理，由于这是一种很常见的情形，pomelo对其进行了抽象，也就是filter。在pomelo中，filter分为before filter和after filter。在一个请求到达Handler被处理之前，可以经过多个before Filter组成的filter链进行一些前置处理，比如对请求进行排队，超时处理。当请求被Handler处理完成后，又可以通过一个after filter链进行一些善后处理。这里需要注意的是在after filter中一般只做一些清理处理，而不应该再去修改到客户端的响应内容，因为此时，对客户端的响应内容已经发给了客户端。

本例是一个聊天应用，在聊天室里，当有人说脏话时，往往需要进行屏蔽。我们在这里就以加一个脏话屏蔽的filter来示范如何使用pomelo的filter。具体的代码请切换到tutorial-abuse-filter分支，使用如下命令：

$ git checkout tutorial-abuse-filter

### Filter结构

Filter是一个对象，定义一个Filter的大致代码如下:

var Filter = function (<args>) {

// ....

};

Filter.prototype.before = function(msg, session, next) {

// ...

}

Filter.prototype.after = function(err, msg, session, resp, next) {

// ...

}

如果定义了before，那么就可以作为一个before filter使用，如果定义了after，就是一个after filter。

对于before filter来说，其有两个参数msg和session，这里msg可能是用户请求原始内容，也可能是经过了前面filter链处理后的内容。如果在后端服务器上，session在这里是BackendSession，如果在前端服务器上，则是FrontendSession，用户对其的直接修改都只会在整个请求处理链的后面处理过程中有效，而不会对前端的session有任何影响，更不会影响原始的session信息了。当然如果确实有需要修改session的话，比如绑定uid的话，可以通过BackendSessionService的相关调用达到目的。

在after中，err是当前面有错误的错误信息，resp是对客户端的相应内容。当定义好Filter后，通过application的filter，before或after调用将其挂到对应的逻辑服务器处理的处理链上。这是只是一个简单教程，不做深入探讨。

### 定义我们自己的Filter

我们这里需要的一个脏话过滤Filter，为了简单期间我们只对fuck进行过滤。在before filter里，如果用户发言里有fuck字眼，那么就将其替换为\*\*\*\*,并在其session里增加一个标记。在after filter里，我们检查session的这个标记，如果是脏话，那么就将这个用户的名字记录下来，同样为了简单起见，我们将其记录的方式仅仅是打到console中。我们的abuseFilter 代码如下:

// abuseFilter.js

module.exports = function() {

return new Filter();

}

var Filter = function() {

};

Filter.prototype.before = function (msg, session, next) {

if (msg.content.indexOf('fuck') !== -1) {

session.\_\_abuse\_\_ = true;

msg.content = msg.content.replace('fuck', '\*\*\*\*');

}

next();

};

Filter.prototype.after = function (err, msg, session, resp, next) {

if (session.\_\_abuse\_\_) {

var user\_info = session.uid.split('\*');

console.log('abuse:' + user\_info[0] + " at room " + user\_info[1]);

}

next(err);

};

在定义完filter后，我们需要把其配置到chat服务器中，在app.js中增加代码如下：

// app.js

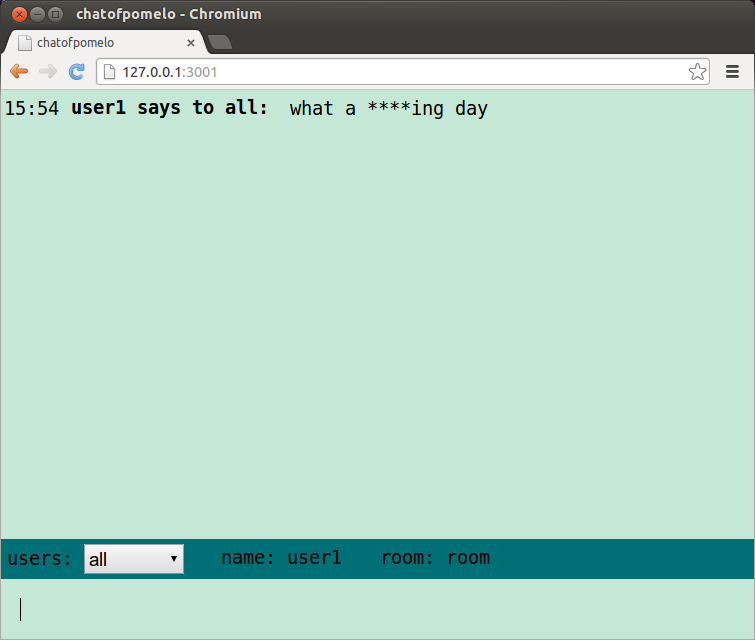
var abuseFilter = require('./app/servers/chat/filter/abuseFilter');

app.configure('production|development', 'chat', function() {

app.filter(abuseFilter());

}

好了，让我们按照前面所讲的部分，重新运行我们的chat应用，在聊天内容里面输入what a fucking day，看看是不是已经被\*\*\*\*替换了，效果图如下：



### 一些说明

* 需要指出的是，这里使用filter来做脏话替换，可能不是很合理，但仅仅为了示例filter的使用，还是可以的。一般情况下，在before filter里，可以做一些请求排队，超时处理, 而在after filter里做一些清理记录的处理。比如，为了统计Handler的处理请求时间，可以在before filter里给session记录一个时间戳，在after filter里取出刚才的时间戳，跟当前时间做运算，就能得到Handler处理请求的时间。pomelo内置提供了几个filter，有toobusy，timeout等，这里不再深入。
* 一个filter里可以只定义before，可以只定义after，也可以两者都定义。application中与filter相关的调用为filter，after和before。如果一个filter既定义了before又定义了after，那么就可以调用filter，这样，application就会将其after和before都加载进去，否则，就只能调用after或者before了。

### 小结

在这部分，我们使用了pomelo 提供的filter机制实现了我们聊天应用的脏话过滤。当然我们的实现非常得简陋而且并不一定很合理，但是仅仅是为了说明Filter的使用方式，还是可行的。下一步我们来使用[基于dict的route压缩](https://github.com/NetEase/pomelo/wiki/%E8%AF%95%E8%AF%95route%E5%8E%8B%E7%BC%A9)来继续完善我们的聊天应用。

# 增加rpc调用

在多进程应用中，进程间通讯是不可或缺的。在pomelo中，借助javascript的语言特性，实现了对开发者非常友好的一个rpc框架。下面，我们将在我们的chat应用中，实践一个rpc调用。为了保持简单，而又能说明问题，我们“画蛇添足”地实现一个时间服务器，当gate服务器接受到用户的查询请求时，我们让gate服务器向时间服务器请求当前的时间，并将其打印在console上。当然这个例子没有啥实际意义，你也可以认为它有意义，因为一个统一的时间服务器可以提供统一的时间信息。这里仅仅是为了示例rpc调用的使用方式，并向用户展示pomelo中rpc调用的方便性。

实际上在我们的聊天应用中，已经有了rpc调用的实现，那就是当有用户连接connector或者离开时，connector会向chat发起rpc，chat会根据相应的用户离开和加入，对应操作其Channel信息。由于在前面我们并没有很详细地对其进行分析，所以干脆重新实现一个全新的rpc，同时，我们也可以展示如何在pomelo应用中增加一个服务器类型。

### chat中增加rpc

下面将给我们的聊天应用增加一个rpc调用和一个time类型的服务器，具体的代码在分支tutorial-rpc 上，使用如下命令切换分支:

$ git checkout tutorial-rpc

首先，我们在servers下建立时间服务器类型time，建立服务名称timeHandler,获取时间方法getCurrentTime(arg1, arg2, cb)，其中arg1, arg2没有实际意义，纯属于示例的目的，在servers/time/remote/timeRemote.js 里面，定义方法：

// timeRemote.js

module.exports.getCurrentTime = function (arg1, arg2, cb) {

console.log("timeRemote - arg1: " + arg1+ "; " + "arg2: " + arg2);

var d = new Date();

var hour = d.getHours();

var min = d.getMinutes();

var sec = d.getSeconds();

cb( hour, min, sec);

};

这里，首先将客户端传来的两个参数打印到console上，然后获取到当前的时间，然后取出其时分秒信息，将其发到客户端。 客户端的调用：

// gateHandler.js

var routeParam = Math.floor(Math.random()\*10);

app.rpc.time.timeRemote.getCurrentTime(routeParam, arg1, arg2, function(hour, min, sec) {

// ...

});

在客户端的rpc调用中，getCurrentTime的第一个参数是用来做路由计算的，arg1, arg2...为调用参数示例，这里的参数arg1, arg2实际上没有啥实际用途，仅仅是为了示例，我们在远程调用的服务端也仅仅是将其打印到console而已。当然实际的rpc调用的时候，就可以使用多个参数，从客户端给服务端传参数。最后的回调应与服务端最后的回调签名保持一致。对于routeParam，我们在这里不再使用session，而是使用一个0-10之内的随机整数，然后直接让其与服务器的个数做hash，得到一个具体的时间服务器。

当有多个time服务器的时候，我们还要为每一次对time服务的请求配置相应的路由，rpc路由函数的第一个参数为rpc调用时的第一个参数，对于本例来说，就是随机数routeParam。在pomelo中很多时候是使用session作为路由参数的，这里示例了一个不一样的路由参数，具体示例代码如下：

// app.js

var router = function(routeParam, msg, context, cb) {

var timeServers = app.getServersByType('time');

var id = timeServers[routeParam % timeServers.length].id;

cb(null, id);

}

app.route('time', router);

这样我们就定义了对time服务器的路由函数。路由函数的参数routeParam就是rpc调用时的第一个参数，msg中封装了rpc调用的详细信息，包括namespace，servertype，等等。context是rpc客户端的上下文，一般由全局的application充当,cb是一个回调，第一个参数是当有错误发生时的错误信息,第二个参数是具体的服务器id。

在服务器配置config/servers.json中增加time服务器如下：

"time":[

{"id": "time-server-1", "host":"127.0.0.1", "port" : 7000},

{"id": "time-server-2", "host":"127.0.0.1", "port" : 7001}

]

好了，这样就为聊天应用增加了一个时间服务器，时间服务器提供一个远程时间，当gate接收到查询请求时，会向time服务器发一个请求，time服务器会为其提供一个时间。gate服务器会向console打印出其得到的远程时间，time服务器会向console打印出gate发起rpc请求时提供的两个参数arg1,arg2,虽然这两个参数没有啥实际意义，但是还是演示了如何在远程调用中传参数。最后的两个回调函数，我们保持了回调函数的签名一致即可。回调函数可以有多个参数，说明我们的rpc调用实际上是可以返回多个结果的。

### 一些说明

* 你可能注意到了我们在time服务器的timeRemote.js和chat服务器的chatRemote.js中，定义远程调用方式的不同了。在timeRemote.js中，直接在module.exports上面挂载getCurrentTime，而在chatRemote.js中，则是另一种方式，示例如下:

// chatRemote.js

module.exports = function(app) {

return new ChatRemote(app);

};

// timeRemote.js

module.exports.getCurrentTime(arg1, arg2, cb) {

// ...

};

实际上这两种方式都是可以的，pomelo在加载具体的remote的时候，如果发现加载到的不是一个对象而是一个函数，那么将认为其是一个工厂方法，它将使用一个全局的上下文（一般是唯一的一个application实例）作为参数，调用这个函数，并使用其结果。chatRemote就是使用这种方式，最终加载到的remote对象实际上是一个ChatRemote对象; 而对于timeRemote来说，require调用返回的就是一个对象，这个对象有一个方法getCurrentTime，所以这个时候，就不需要进行一次函数调用了。

当remote需要当前的application实例的时候，往往可以使用第一种chatRemote的那种方式，而当remote跟app完全没关系时，也可以使用timeRemote的这种实现方式，这在pomelo中是没有差别的。

不仅仅是remote，对于handler也是一样，也就是说定义handler的时候也可以使用这两种方式中的一种，只不过到目前为止，我们这个例子中使用的都是类似于chatRemote中的那种实现方式。

* 当前端服务器接受客户端请求，将请求路由给后端服务器时，pomelo使用的是发起一个系统级远程调用的方式，这个时候会使用session作为rpc请求的路由参数，这也是我们看到的前面在给chat配置路由的时候，路由函数的第一个参数总是session。在time中，我们使用了一个随机整数作为路由参数，因此time的路由函数的第一个参数也就是这个随机整数了。实际上pomelo的rpc框架对于路由参数的使用是没有限制的，并不仅限于一直使用session。
* rpc调用的返回值是通过回调的形式获得的，回调函数也就是我们上面看到的rpc调用的最后一个参数。这个回调函数可以有多个参数，表示远程调用可以返回多个值，在我们这个例子中，返回了时分秒三个值。
* 0.8版本以后，当进行rpc调用的时候，可以跳过路由计算而直接将调用发送到一个具体的服务器或者广播到一类服务器的调用方式，代码示例如下：

// route

var routeParam = session;

app.rpc.area.playerRemote.leaveTeam(routeParam, args..., cb);

// to specified server 'area-server-1'

app.rpc.area.playerRemote.leaveTeam.toServer('area-server-1', args..., cb);

// broadcast to all the area servers

app.rpc.area.playerRemote.leaveTeam.toServer('\*', args..., cb);

### 小结

在这部分，我们给聊天应用增加了一个time服务器并实现了一个rpc调用，并对pomelo的rpc调用进行了一些说明。下一步，我们将[增加一个component](https://github.com/NetEase/pomelo/wiki/%E7%BB%99pomelo%E5%8A%A0%E4%B8%AA%E7%BB%84%E4%BB%B6)。

# 扩充服务器

当我们的应用只有很少人用的时候，往往只需要一台服务器就可以支撑。但是随着用户的增加，一台服务器可能就无法承受同一时刻巨大的访问量，这需要我们对服务器进行伸缩扩充。

多服务器版本的聊天应用在分支tutorial-multi-server上，你需要执行如下命令来切换到多服务器分支上：

$ git checkout tutorial-multi-server

### 配置修改

在pomelo中，对服务器的扩充非常简单，只需要修改一下配置文件，多添几台服务器配置就行了，对于我们的聊天例子来说，如下下面所示，在config/servers.json中的配置：

{

"development":{

"connector":[

{"id":"connector-server-1", "host":"127.0.0.1", "port":4050, "clientPort": 3050, "frontend": true},

{"id":"connector-server-2", "host":"127.0.0.1", "port":4051, "clientPort": 3051, "frontend": true},

{"id":"connector-server-3", "host":"127.0.0.1", "port":4052, "clientPort": 3052, "frontend": true}

],

"chat":[

{"id":"chat-server-1", "host":"127.0.0.1", "port":6050},

{"id":"chat-server-2", "host":"127.0.0.1", "port":6051},

{"id":"chat-server-3", "host":"127.0.0.1", "port":6052}

],

"gate":[

{"id": "gate-server-1", "host": "127.0.0.1", "clientPort": 3014, "frontend": true}

]

},

"production":{

"connector":[

{"id":"connector-server-1", "host":"127.0.0.1", "port":4050, "clientPort": 3050, "frontend": true},

{"id":"connector-server-2", "host":"127.0.0.1", "port":4051, "clientPort": 3051, "frontend": true},

{"id":"connector-server-3", "host":"127.0.0.1", "port":4052, "clientPort": 3052, "frontend": true}

],

"chat":[

{"id":"chat-server-1", "host":"127.0.0.1", "port":6050},

{"id":"chat-server-2", "host":"127.0.0.1", "port":6051},

{"id":"chat-server-3", "host":"127.0.0.1", "port":6052}

],

"gate":[

{"id": "gate-server-1", "host": "127.0.0.1", "clientPort": 3014, "frontend": true}

]

}

}

### router配置

与前面的每个服务器类型仅有一台服务器的例子相比，这里我们的connector和chat都具有多台服务器。因此需要考虑对用户请求的服务器分配问题。

对于gate服务器来说，在前面的例子中，由于只有一个connector服务器，所以直接返回仅有的一个服务器就行了，而这里有多台服务器，所以需要从中选择一个服务器的信息进行返回，这里我们增加了一个工具函数dispatch,它完成具体的分配运算，他使用用户的uid的crc32的校验码与connector服务器的个数取余，从而得到一个connector服务器，大致代码如下:

// util/dispatcher.js

module.exports.dispatch = function(key, list) {

var index = Math.abs(crc.crc32(key)) % list.length;

return list[index];

};

// gateHandler.js

handler.queryEntry = function(msg, session, next) {

// ...

// get all connectors

var connectors = this.app.getServersByType('connector');

// ...

var res = dispatcher.dispatch(uid, connectors); // select a connector from all the connectors

// do something with res

};

当客户端请求到来时，因为有多台chat服务器，需要选择由哪台chat服务器来服务，也就是前端服务器把这个客户端请求路由到哪个后端服务器上。配置路由使用application的route调用，这里我们也使用了前面提到的工具函数dispatch，使用同样的服务器分配策略，示例如下：

// app.js

// route definition for chat server

var chatRoute = function(session, msg, app, cb) {

var chatServers = app.getServersByType('chat');

if(!chatServers || chatServers.length === 0) {

cb(new Error('can not find chat servers.'));

return;

}

var res = dispatcher.dispatch(session.get('rid'), chatServers);

cb(null, res.id);

};

app.configure('production|development', function() {

app.route('chat', chatRoute);

});

其中chatRoute就是路由函数，他接受四个参数，返回一个其选择的后端服务器id，四个参数中，第一个是专门用作路由计算的参数，前端服务器路由请求给后端服务器发rpc调用时，会使用session作为计算路由的参数，但是当用户自定定义rpc的时候，用户完全可以自己定义这个参数的含义，当然也可以使用session。第二个参数msg描述了当前rpc调用的所有信息，包括调用的服务器类型，服务器名字，具体的调用方法等信息。第三个参数是一个上下文变量，一般情况下会由app来充当，第四个是一个获得到后端服务器id后的回调函数。

### 一些说明

* 通过修改服务器的配置文件，并增加具体的路由选择配置，就可以很轻易地实现服务器的扩充。 对于我们这个应用来说，如果我们还想继续扩充我们的服务器，那么只需在servers.json里面继续增加服务器的配置就行了。
* 如果我们一开始实现时就考虑以后的扩充，实现所有的路由选择函数的话，那么以后当需要扩充服务器的时候，只需要在servers.json里面增加相应的配置就行了。
* 关于客户端请求的路由，你可能会问，为什么我们在单服务器的那个例子里，没有给chat定义router。那是因为如果我们不定义router的话，pomelo会使用一个默认的router完成路由，因为只有一台chat服务器，那么pomelo总会把所有的对其的请求路由给这个服务器，所以，我们在前一个例子中，就省略掉了chat的路由配置。
* 实际应用中，我们一般都要自己实现router,而不使用pomelo默认的。使用多台服务器要考虑负载平衡，同时要尽量使得服务器的服务是无状态的。我们chat的例子中，定义的router就是使用了用户的rid的crc校验码作为键值对当前的所有chat服务器个数做了一个简单的hash运算，以使得所有的chat服务器的负载尽可能平衡。

### 小结

在这部分，我们实现了对服务器的扩充，它是如此的简单，只需要修改相应的服务器配置即可。下面我们将尝试使用pomelo的[filter机制](https://github.com/NetEase/pomelo/wiki/%E5%A2%9E%E5%8A%A0filter)继续完善我们的聊天应用。

# 消息压缩

在实际编程中，为了减少数据传输带宽的消耗，提高传输效率，pomelo提供了对消息的压缩，包括基于字典的对route的压缩和基于protobuf的对具体传输数据的压缩。

# route压缩

在实际编程中，网络带宽的有效数据负载率是一个值得考虑的问题。特别地，对于移动客户端来说，网络资源往往并不是很丰富，为了尽可能地节省网络资源，往往需要尽大可能地增加数据包的有效数据率。

#### route问题

在pomelo编程中，pomelo中的route是用来确定消息的分发路径，将其交给相应的服务器和服务处理的。route分为两类，由客户端发给服务端消息时使用的route和服务端向客户端广播时使用的route。

* 前一种route是由服务器自动生成的，其中的字段就代表了对应的方法在服务端的位置。如“area.playerHandler.attack”则表示在“area”类型的服务器上的“playerHandler”提供的“attack”方法，其格式为".."。 路由信息过长，使得有效消息数据负载率大大降低。例如，在聊天应用中，如果用户的发言仅仅是一个字符，结果不得不携带一个route,"chat.chatHandler.send",这样使得有效数据负载率大大降低。
* 后一种route是服务端想客户端推送消息时使用，是客户端的路由信息，如“onMove”，“onAttack”等，其格式一般为"on"这些字段是由用户自己定义的。虽然可以定义很短的路由，但是那样会造成可读性变差，不利于代码阅读。

一般来说，当应用固定后，具体路由就不会再变动，因此可以考虑通过一种简单替换的方式对路由信息进行压缩。

#### 基于dict的压缩

pomelo中实现了基于字典的route压缩，目前route压缩功能仅仅支持hybridconnector，sioconnector目前无法使用route压缩。其实现原理如下：

* 对于系统生成的route，也就是服务端的路由信息，即格式为".."的路由信息，在系统启动时由CoDictionary组件进行服务端路由信息扫描，然后会对每一个route生成唯一的字典项，由一个无符号小整数标识。
* 对于用户自定义的route，也就是客户端的路由信息，即格式为"on"，则需要用户提供一个自定义的route列表，会根据这一个列表对每个用户自定义的route生成一个对应的字典项，即也就是一个无符号小整数。
* 在开启字典功能的状态下，使用hybridconnector的时候，当协议握手的时候，服务端会将整个字典的消息发送给客户端，这样客户端和服务端都会拥有相同的具体route无符号整数的对应关系。
* 当有消息传递时，其中的route在发送时会被替换为在字典项,而接收端会自动还原，这一过程对于用户而言是完全透明的。

#### 使用route压缩

目前仅仅在hybridconnector中实现了支持，启用此功能只需在app.js中配置开启即可，示例如下：

app.set('connectorConfig',

{

connector : pomelo.connectors.hybridconnector,

heartbeat : 3,

useDict : true // enable route compression

}

});

客户端的路由信息，也就是用户自定义的路由信息，需要用户自己通过配置文件配置，具体的配置文件为config/dictionary.json，在这个文件中加入一个字符串列表即可，示例如下:

[

"onDropItem",

"onAttack",

"onDied",

"onMove",

"onRevive",

"addEntities",

"onRemoveEntities",

"onPathCheckout"

]

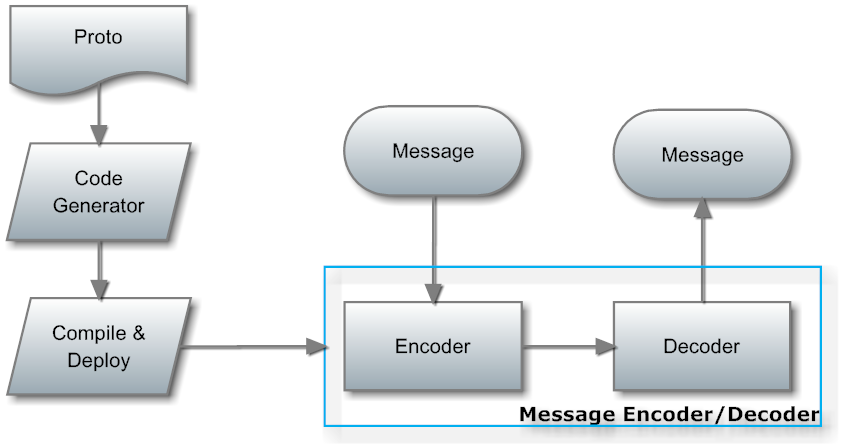
注意：对于没有加入到这个列表中的客户端路由信息，依然会使用原始的路由，而不会使用整数，pomelo在打包消息的时候会进行判断，如果字典项里有相应的路由信息，那么会使用字典项，如果没有的话，会使用原始的路由信息。这一切对用户是透明的，用户只需要配置好就行，不用关心其具体实现。

### 基于protobuf的传输数据压缩

在进行消息传输时，pomelo实现了基于protobuf的数据编码协议，与其他的编码协议如xml，json相比，protobuf有着更好的传输效率和压缩比率。在我们的lordofpomelo项目中，使用protobuf进行数据编码后的消息大小只有基于Json的编码的20%左右。

#### protobuf协议介绍

protobuf协议是由google制定的，主要用于其内部的rpc调用和文件编码。原生的protobuf包括两部分内容：基于二进制的数据编码协议和基于proto元数据的代码生成器。首先，需要根据每条消息来编写对应的proto文件，然后使用google提供的代码生成器，基于proto文件来生成相应的编码器和解码器，然后使用生成的编/解码器来进行编/解码操作，对应的流程如下图：

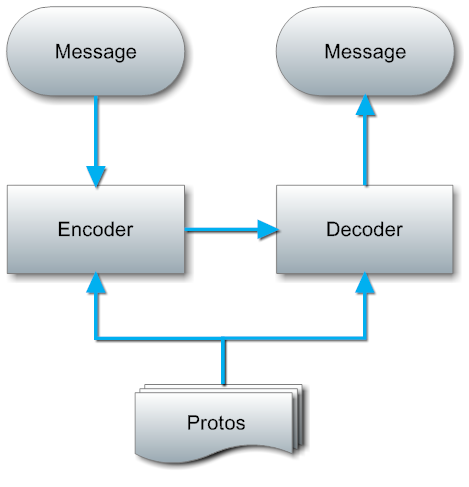


这种方式的优势是代码静态生成，运行时不需要proto文件信息，而且可以根据具体的信息内容对代码进行优化，编解码的时候不需要类型元信息，效率很高。但缺点也十分明显：使用复杂（涉及到代码生成，编译，部署），改动成本高昂（需要重新生成，编译代码，并对代码进行部署），需要生成大量新代码（每个消息都需要一个独立的编码/解码器）。 关于protobuf协议的更多内容，可以参见其官网[protobuf项目](https://code.google.com/p/protobuf/)。

#### pomelo中的protobuf

原生的带有代码生成器的protobuf过于重量级，缺乏灵活性，任何消息的修改都会是一个非常重量级的操作，而这个在pomelo中，由于pomelo是能够快速开发的，因此也必须要求代码不能有编译阶段，所有的类型信息应该在运行时进行评估。因此，我们没有采用生成代码的方式，而是根据proto文件的定义，对消息进行即时的解析。

在pomelo中，我们实现了一个通用的protobuf编/解码器，以及一个proto文件解析器。通过分析proto文件内容，实现了对消息的编码/解码。这样，当修改/添加消息类型时，只需要修改对应的proto文件就可以了。具体的运行流程如下图：



从上图可以看出，与原生的protobuf生成代码的方式相比，pomelo中的解决方案要更将灵活，轻量。不需要生成任何代码，在运行时通过proto文件中对消息的定义，实现对消息的动态编码/解码功能。

#### proto 文件定义

原生的protobuf中，每一个消息都与一个proto中的message定义对应，而在生成编码/解码器之后，这些message的定义不再被使用。而在pomelo中，因为我们需要proto的内容来动态的对消息进行编码/解码，因此需要维护一个完整的protos信息表, 我们将所有的proto定义放在一个json文件中，通过key来进行区分，在pomelo中，key就是消息的route。我们的proto文件使用了类似与原始proto的语法，不过是使用了json格式，示例如下：

"onMove" : {

"required uInt32 entityId" : 1,

"message Path": {

"required uInt32 x" : 1,

"required uInt32 y" : 2

},

"repeated Path path" : 2,

"required uInt32 speed" : 3

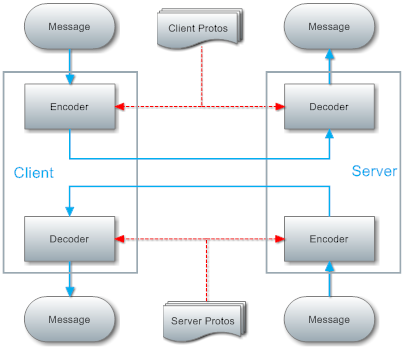
},

"onAttack" : {

"required uInt32 attacker" : 1,

"required uInt32 target" : 2

}

在pomelo中，对于同样route的消息，如'area.playerHander.attack'，在客户端和服务端的格式可能完全不同，这就意味着对于客户端的编码器和解码器对于同样route的消息需要不同的定义。因此，我们需要两套protos文件，server protos和client protos，具体的关系如下图： 

#### 使用protobuf

虽然protobuf的实现看上去十分复杂，但由于这一层对用户是完全透明的，使用会非常简单。用户只需要通过简单的两步定义就可以在原有的项目中开启protobuf功能。

* 首先，需要在connector组件上打开protobuf开关，在app.js中的配置如下：

app.set('connectorConfig',

{

connector : pomelo.connectors.hybridconnector,

heartbeat : 3,

useProtobuf : true

});

* 实际上需要加入的就是“useProtobuf：true”这一项。当设置这一标识后，pomelo会在客户端握手时将protos内容同步到客户端，并默认开启protobuf压缩功能。
* 在protobuf功能开启用，用户还需要加入protos定义来实现对具体消息的编码/解码。protos文件默认在/game-server/config目录下，包括两个文件：serverProtos.json和clientProtos.json，分别表示服务端->客户端消息的protos和 客户端->服务端消息的protos。只要在其中加入有效的proto定义，就可以开启对应消息的protobuf编码功能，CoProtobuf组件会自动加载这两个proto文件。
* 当然pomelo中的protobuf实现对原有项目是完全兼容的，你可以直接在老的项目中打开protobuf开关而不会引起任何问题。只是当proto定义是空的，默认所有的消息都不会经过protobuf压缩，而是采用默认的二进制编码进行传输。
* 当你想对某个消息进行protobuf编码时，只需要在对应的protos文件（serverProtos.json或clientProtos.json）中加入对应的protobuf项，pomelo在启动时就会自动识别并对消息进行压缩，而不会对其他未定义的消息产生任何影响。

### 总结

在这部分，介绍了pomelo中实现的对数据的压缩。通过对数据的压缩提高了带宽的有效数据利用率，使得在一些带宽以及流量敏感的环境中，pomelo能够更好地工作。目前pomelo的route以及基于protobuf的数据压缩仅仅支持hybridconnector，对于sioconnector，是使用json作为通信格式的，目前不支持对其进行压缩。

# 组件概述

我们知道，pomelo的应用程序执行的全部过程，就是对其相应组件的生命周期的管理，实际的所有逻辑功能均有pomelo组件提供。pomelo内建提供了十多个组件，这些组件适用于不同的服务器，提供不同的功能。有些组件提供的功能比较复杂，有些则比较简单。下面我们将以提供的功能为主线来阐述pomelo提供的内建组件。

### master组件

master组件仅仅由master服务器加载，它主要的功能包括启动所有的应用服务器、管理和监控所有的应用服务器和接受管理客户端的请求与响应。

在master组件的start方法里，会根据用户提供的服务器配置信息，启动用户配置的所有的具体应用服务器。

当master组件start结束后，他将开启一个socket监听端口，接受应用服务器和监控客户端的连接和注册，收集应用服务器上报的监控信息，给应用服务器推送一些消息，并对管理客户端发出的管理请求给予响应。管理客户端如[pomelo-cli](https://github.com/NetEase/pomelo/wiki/Pomelo-cli%E4%BD%BF%E7%94%A8)可能发出的请求包括查看某个服务器进程状态，增加一个服务器，停掉一个服务器等。以增加一个服务器为例，当管理客户端发出增加服务器请求时，会提供相应的服务器参数，如服务器类型，主机ip，开启的端口等。此时，master组件接受后，会启动相应的服务器，并将新增加的服务器信息广播通知给其他已经启动的服务器。

master组件无配置项。

### monitor组件

monitor组件由所有的包括master服务器在内的服务器都会加载，它的主要功能就是与master建立连接进行通信，从而对整个应用服务器群进行管理和监控。master服务器本身也会加载monitor服务器，因为master服务器也会收集其本身自己的监控信息。

可以认为monitor服务器与master服务器是对等组件，monitor会通过master接受一些命令，比如关闭整个服务器等。对于一些周期性监控的信息，pomelo提供了两种收集方式，即pull方式和push方式。pull方式要求master周期地去与monitor通信，拉取相应的监控信息；push方式，则是由monitor周期地主动地向master报告其监控信息。

monitor组件无配置项。

### connector组件

connector组件是一个重量级的组件，它会依赖于session组件，server组件,pushScheduler组件和connection组件。connector组件仅仅被前端服务器加载，它主要用来管理客户端的连接。connector组件会加载底层的connector，创建端口监听，绑定事件响应。当有客户端连接请求时，connector组件会请求session组件，获得当前连接的session，如果session组件中没有相应的session的话，session组件会为这个新连接创建新的session，并维护相应的连接；然后connector组件还会向connection组件上报连接信息，供统计使用；最后，将拿到的session以及客户端的请求，一起抛给server组件，由server组件进行请求处理。当server组件处理完请求后，又会通过connector组件将响应返回给客户端。在返回响应给客户端的时候，connector组件做了一个缓存选择，这个缓存实现依赖于pushScheduler组件，也就是说connector组件并不是直接将相应发给客户端，而是将响应给pushScheduler组件。pushScheduler组件根据相应调度策略，可能不缓存直接通过session组件维护的连接，将响应发出去，也可能进行缓存，并按时flush。这是可以配置的。

connector组件支持如下配置项:

* connector: 底层使用的通信connector，不配置的话，会默认使用sioconnector;
* useProtobuf: 目前仅仅支持connector配置使用hybridconnector的情况，配置其为true，将开启消息的protobuf功能；
* useDict： 目前仅仅支持connector配置使用hybridconnector的情况，配置其为true时，将会开启基于字典的路由消息压缩；
* useCrypto： 目前仅仅支持connector配置为hybridconnector的情况，配置其为true时，将会启用通信时的数字签名；
* encode/decode： 消息的编码解码方式，如果不配置的话，将会默认使用connector配置中，底层connector提供的相应的编码解码函数。
* transports：这个配置选项是用于sioconnector的，因为socket.io的通信方式可能会有多种，如websocket，xhr-polling等等。通过这个配置选项可以选择需要的方式。

配置connector组件，通过调用如下方式进行:

app.set('connectorConfig', opts);

### session组件

session组件跟connector相关，也是仅仅被前端服务器加载，为sessionService提供一个组件包装, 加载session组件后，会在app的上下文中增加sessionService，可以通过app.get('sessionService')获取。它主要用来维护客户端的连接信息，以及生成session并维护session。如果与经典TCP进行类比的话，那么session中维护的连接就可以粗略地认为就是TCP服务器端accept返回的socket句柄。一个连接与一个session对应，同时session组件还维护具体登录用户与session的绑定信息。一个用户可以有多个客户端登录，对应于多个session。当需要给客户端推送消息或者给客户端返回响应的话，必须通过session组件拿到具体的客户端连接来进行。

session组件支持如下配置项:

* singleSession： 如果这个配置项配置为true的话，那么将将不允许一个用户同时绑定到多个session，在绑定用户一次后，后面的绑定将会失败。

配置session组件，通过调用如下方式进行:

app.set('sessionConfig', opts);

### connection组件

connection组件是一个功能相对简单的组件，也是仅仅被前端服务器加载,为connectionService提供一个组件包装,他主要进行连接信息的统计,connector组件接收到客户端连接请求以及有客户端离线时，以及用户登录下线等等情况，都会向其汇报。

connection组件无配置项。

### server组件

server组件也是一个功能比较复杂的组件，它被除master外的服务器加载。server组件会加载并维护自身的Filter信息和Handler信息。server组件会从connector组件的回调里获得到相应的客户端请求或者通知，然后会使用自己的before filters对其消息进行过滤，再次调用自己的相应Handler进行请求的逻辑处理，然后将响应通过回调的方式发给connector处理。最后调用after filters进行一些清理处理。

当然，如果客户请求的服务本来就是前端服务器提供的话，会是上面的那种处理流程。如果客户请求的服务是后端服务器提供的服务的话，则将不是上面的那种处理流程，此时会出现sys rpc调用。前面那种前端服务器自己处理的情况具体调用为doHandle，而发起rpc调用的情况则为doForward。这两种处理流程的不同点是，对于自身的请求，调用自己的filter-handler链进行处理，对于不是前端服务器自己提供的服务，则是发起一个sys rpc，然后将rpc调用的结果作为响应，发给connector进行处理。关于这个rpc调用则是pomelo内建的msgRemote实现的。

对于后端服务器来说，其客户请求不是直接来源于真实的客户端，而是来源于前端服务器对其发起的sys rpc调用，这个rpc调用的实现就是pomelo内建的msgRemote，在msgRemote的实现里，会将来自前端服务器的sys rpc调用请求派发给后端服务器的server组件，然后后端服务器会启用filter-handler链对其进行处理，最后通过rpc调用的返回将具体的响应返回给前端服务器。

在前端服务器将客户端请求向后端服务器分派时，由于同类型的后端服务器往往有很多，因此需要一个路由策略router，一般情况下用户通过Application.route调用为后端服务器配置router。

server组件无配置项。

### pushScheduler组件

pushScheduler组件也是一个功能较为简单的组件，它仅仅被前端服务器加载，与connector组件的关系密切。当connector组件收到server组件的对客户端请求的响应后，connector并不直接将此响应返回给客户端，而是将这个给客户端发送响应的操作调度给scheduler组件。pushScheduler组件完成最后通过session组件拿到具体的客户端连接并将请求的响应发送给客户端的任务。因此，通过pushScheduler组件可以对发给用户的响应进行缓冲，从而提高通信效率。pomelo实现了两种调度策略，一种是不进行任何缓冲，直接将响应发送给客户端，一种是进行缓冲，并定时地将已缓冲的响应发送给对应的客户端。

pushScheduler配置项:

* scheduler： scheduler组件的具体调度策略配置，默认的是直接将响应发给客户端，同时pomelo还提供了有缓冲并且定时刷新的调度策略。用户也可以自定义自己的调度策略。

配置pushScheduler组件，通过调用如下:

app.set('pushSchedulerConfig', opts);

如果要启用使用缓冲的scheduler的话，可以在app.js中增加:

app.set('pushSchedulerConfig', {scheduler: pomelo.pushSchedulers.buffer, flushInterval: 20});

flushInterval是刷新周期，默认为20毫秒。

### proxy组件

proxy组件是一个重量级的组件，它被除master外的所有服务器加载。proxy组件会扫描具体应用服务器的目录，抽取其中的remote部分，由于javascript语言的动态性，可以很轻易地获得到remote中的关于远程调用的元信息，生成stub，并将这些调用都挂到app.rpc下面，当用户发起rpc调用时，proxy组件会查看其扫描到的stub信息，以此决定此远程调用是否合法。同时，proxy又会创建一个RpcClient，当发起远程调用时，负责与远端的remote进行通信，并得到远程调用的结果供调用者使用。当进行远程调用时，由于同类型的远程服务器可能有多个，所以这里同样需要配置相应的router。

proxy的配置项：

* cacheMsg, 配置cacheMsg为true的话，将开启rpc调用时的对消息的缓冲，而不是直接一旦有rpc请求就发出。
* interval, 与配置参数cacheMsg配合使用，设置flush缓存的周期
* mailBoxFactory, rpc底层实现需要的，用户可以定义自己的mailBoxFactory,我们将在rpc原理里面详述。

另外，可以开启rpc的调用日志，通过如下的调用:

app.enable('rpcDebugLog');

配置proxy使用：

app.set('proxyConfig', opts);

### remote组件

remote组件是与proxy组件对等的组件，它用来提供rpc调用服务。rpc组件完成对当前服务器的remote的加载，并开启监听端口，等待rpc客户端的连接及相应的rpc调用。当接收到具体的调用请求时，会根据调用请求中描述的调用请求信息，调用相应的remote中的相应方法。然后再将具体的处理结果返回给rpc客户端。rpc服务端还支持对调用请求的filter，也就是说跟server组件处理客户端请求一样，rpc服务端处理具体请求时也会使用filter-remote链进行处理。

remote组件配置项:

* cacheMsg, 与proxy组件的含义相同
* interval， 与proxy组件的含义相同
* acceptorFactory, rpc底层实现需要的,可以认为跟proxy配置中的mailBoxFactory是对等的，我们将在rpc原理里面详述。

跟proxy组件一样，用户可以开启rpcDebugLog来得到所有的rpc调用过程的日志。 配置remote组件使用：

app.set('remoteConfig', opts);

### dictionary组件

dictionary组件是一个可选组件，不会被默认加载，只有当connector组件的配置中开启了useDict的时候，此组件才会被加载。此组件会遍历所有handler的route字符串，还会从config/dictionary.json中读取客户端的route字符串，然后对这些字符串进行编码，给予每一个路由赋予一个唯一的小整数，实现route信息压缩，当客户端与前端服务器通信时需要路由信息时，将不会再使用很长的那个字符串，而仅仅使用一个小整数。

dictionary的配置项:

* dict, 客户端路由字符串文件的位置，默认使用的是config/dictionary.json 配置dictionary组件使用:

app.set('dictionaryConfig', opts);

### protobuf组件

protobuf组件也是一个可选组件，不会被默认加载，只有当connector组件的配置中开启了useProtobuf的时候，此组件才会被加载。此组件会加载对应的proto文件，并完成消息的基于protobuf的编解码。默认的proto文件的配置信息在config/serverProtos.json和config/clientProtos.json中。具体会在详细介绍pomelo-protobuf中详细介绍。

protobuf组件无配置项。

### channel组件

channel组件维护channel信息，可以被除了master之外的服务器加载。channel组件可以看作是channelService的组件包装,加载该组件后，会在app上下文中加入channelService，可以通过app.get('channelService')获取。可以认为一个channel就是一个用户的集合，每一个用户大致对应于前端服务器中的一个session，用户可以通过channel组件向一个channel里面的所有用户推送消息。当然，由于后端服务器并不与客户端直接相连，故后端服务器会发起一个sys rpc来表示向客户端推送消息，接受这个远程调用的是pomelo已经实现的ChannelRemote。

channel组件的配置项：

* broadcastFilter， broadcast的过滤函数。会在执行channel的broadcast的时候，在前端服务器上，在消息发送给每个session之前，进行一个过滤。其函数签名为

broadcastFilter(session, msg, filterParam)

其中filterParam参数由在channelService的broadcast调用时传入，如下:

channelService.broadcast(type, route, {filterParam: param}, cb);

可以通过如下方式对Channel组件进行配置：

app.set('channelConfig', opts)

### backendSession组件

BackendSession组件可以看作是BackendSessionService的组件包装，加载该组件后，会在app的上下文中加入backendSessionService，可以通过app.get('backendSessionService')调用获取。可以被除了master之外的服务器加载。它主要为后端服务器提供BackendSession信息，并通过远程过程调用完成一些比如对原始session绑定uid等操作。

backendSession组件无配置项。

# 给pomelo加个组件

pomelo的核心是由一系列松耦合的component组成，同时我们也可以实现我们自己的component来完成一些自己定制的功能。对于我们的chat应用，我们尝试给其增加一个component，目的是展示如何增加一个component，以及component的生命周期管理，而不会特别关注这个component的实际功能。我们现在就给其增加一个component HelloWorld，这个component仅仅在master服务器上加载运行,在master服务器的话，它将每隔一段时间在console上打印出来一个HelloWorld，具体的时间间隔由opts配置来指定。

### chat中应用

具体的代码在分支tutorial-component上，使用如下命令切换分支：

$ git checkout tutorial-component

* 首先，在app下建立components/HelloWorld.js文件, 大致代码如下：

// components/HelloWorld.js

module.exports = function(app, opts) {

return new HelloWorld(app, opts);

};

var DEFAULT\_INTERVAL = 3000;

var HelloWorld = function(app, opts) {

this.app = app;

this.interval = opts.interval | DEFAULT\_INTERVAL;

this.timerId = null;

};

HelloWorld.name = '\_\_HelloWorld\_\_';

HelloWorld.prototype.start = function(cb) {

console.log('Hello World Start');

var self = this;

this.timerId = setInterval(function() {

console.log(self.app.getServerId() + ": Hello World!");

}, this.interval);

process.nextTick(cb);

}

HelloWorld.prototype.afterStart = function (cb) {

console.log('Hello World afterStart');

process.nextTick(cb);

}

HelloWorld.prototype.stop = function(force, cb) {

console.log('Hello World stop');

clearInterval(this.timerId);

process.nextTick(cb);

}

我们看到每一个component一般都要定义start，afterStart，stop这些hook函数，供pomelo管理其生命周期时进行调用。对于component的启动，pomelo总是先调用其加载的每一个component提供的start函数，当全部调用完后，才会去调用其加载的每一个component的afterStart方法，这里总是按顺序调用的。在afterStart中，一些需要全局就绪的工作可以放在这里完成，因为调用afterStart的时候，所有component的start已经调用完毕。stop用于程序结束时对component进行清理时使用。虽然我们这个例子非常简单，但是足以说明如何在pomelo中定制自己的component，并使用。我们在HelloWorld的start里面启用了一个定时器，每隔一段时间就向console打印一个HelloWorld。然后在stop里关闭它。

* 然后，我们让master服务器来加载我们的这个component，具体代码如下：

// app.js

var helloWorld = require('./app/components/HelloWorld');

app.configure('production|development', 'master', function() {

app.load(helloWorld, {interval: 5000});

});

好了， 我们通过实现一个简单的component，明白了如何实现自己的定制component，当然这个component很简单，也没有啥实际意义，但是它是一个完整的component，有完整的生命周期管理，我们通过这个例子已经了解到了component的创建以及加载。

### 一些说明

* 这里定义的HelloWorld component中，往外导出的是一个工厂函数，而不是一个对象。当app加载component时，如果是一个工厂函数，那么app会将自己作为上下文信息以及后面的opts作为参数传给这个函数，使用这个函数的返回值作为component对象。同样，也可以直接给module.exports赋予一个对象，那样的话，就可以直接使用而不用调用工厂函数，不过这样的话丧失了使用app和具体配置参数， 不够灵活，因此，使用工厂方法的方式是一个好选择。
* 对于start和afterStart的执行，pomelo总是会先按顺序执行完所有component的start后，才会按顺序执行所有component的afterStart，因此可以在afterStart里执行一些需要其他component执行了start后才可以执行的逻辑。
* 实际上，pomelo应用的整个运行过程可以认为是管理其component的生命周期过程，pomelo的所有功能都是通过其内建的component来实现的。用户可以轻松地定制自己的component，然后将其load进去，这样就很轻松地实现了对pomelo的扩展。

### 小结

在这部分，我们给聊天应用“画蛇添足”般地添加了一个HelloWorld component，使得可以更好地理解如何定制自己的component，并加载它。pomelo的框架非常灵活，有很好的可扩展性，从我们的例子中，我们可以看出，可以很容易地对pomelo进行扩展。pomelo不仅可以扩展component，pomelo还提供了一个灵活的可扩展的服务器监控管理框架，[下一步](https://github.com/NetEase/pomelo/wiki/%E5%A2%9E%E5%8A%A0admin-module)，我们将给我们的聊天应用增加一个监控模块，让应用服务器自动向master上报自己的本地时间，以此来示例如何在pomelo中定制自己的监控管理模块。

# 风格与约定

pomelo 有自己的风格以及一些约定，下面对其进行了简单的总结：

* pomelo是一个框架，因此我们再编写的代码都是用来配置框架的,特别是要求我们自己定义一些框架所需的回调方法。
* 在pomelo中，无论是component，handler，filter，module，remote等等，它们在导出的时候，往往都会导出一个工厂函数，而不是直接导出对象，这样的话，就能够进行上下文的注入，同时在加载时可以传入一些配置参数。比起直接导出一个对象，更为灵活好用。在pomelo中也大量使用这种方式。
* pomelo中，很多情况下，框架会从特定的地方读取配置和代码，因此代码组织要遵循pomelo的规范。在game-server/app/servers目录下书写服务器的代码。每一个服务器代码都分到名为handler和remote的两个目录下，它们分别描述了这个服务器接受客户端请求和接受rpc请求的服务端逻辑。因此，在pomelo中此目录结构很重要。
* pomelo中命名风格基本上与常见的java中命名风格相同。用于创建对象的函数全部首字母大写;普通方法和对象的方法采用首单词小写字母开头，后面单词大写字母开头的方式;不使用下划线;常量使用全大写;单词间使用下划线隔开。这种风格是非常常见的命名风格。

# Pomelo 网址分析的

### <http://blog.csdn.net/zj510/article/category/1878449>

# pomelo的一些问题处理

## https://cnodejs.org/topic/51ca033873c638f370e9cce5