## 自定义监控



redis-stat提供了一种简易的方式实现对Redis实例的监控，但多数场景下可能需要进行定制，比如鉴权方式、统计指标、告警等等，为实现更灵活的控制，可利用类似的方式对INFO信息进行解析，以实现自有Redis实例的统一监控。

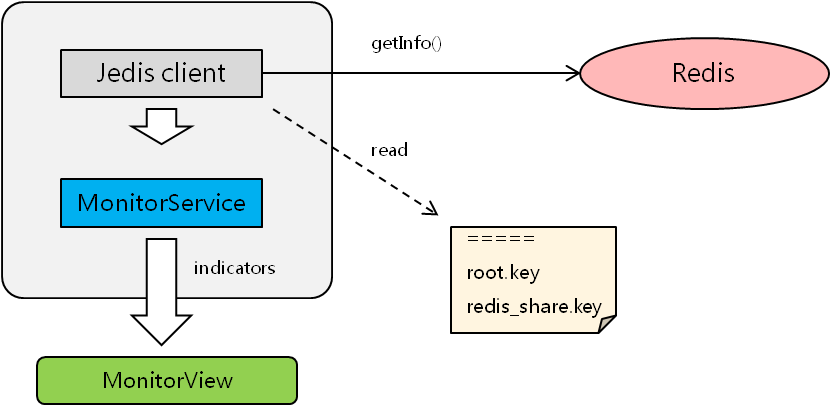


图 9‑4 自定义监控架构图

### 前端页面

一，Bootstrap3 后台管理系统模板

Redis自定义监控系统的前端页面使用的是开源的模板start bootstrap Admin2，它是基于Bootstrap3开发的，是一个轻量级的后台管理系统模板，它有强大的功能组件和UI组件，基本能满足后台管理系统的需求，我们在此基础上进行二次开发。读者可以从start bootstrap Admin2的官网下载最新的源码。

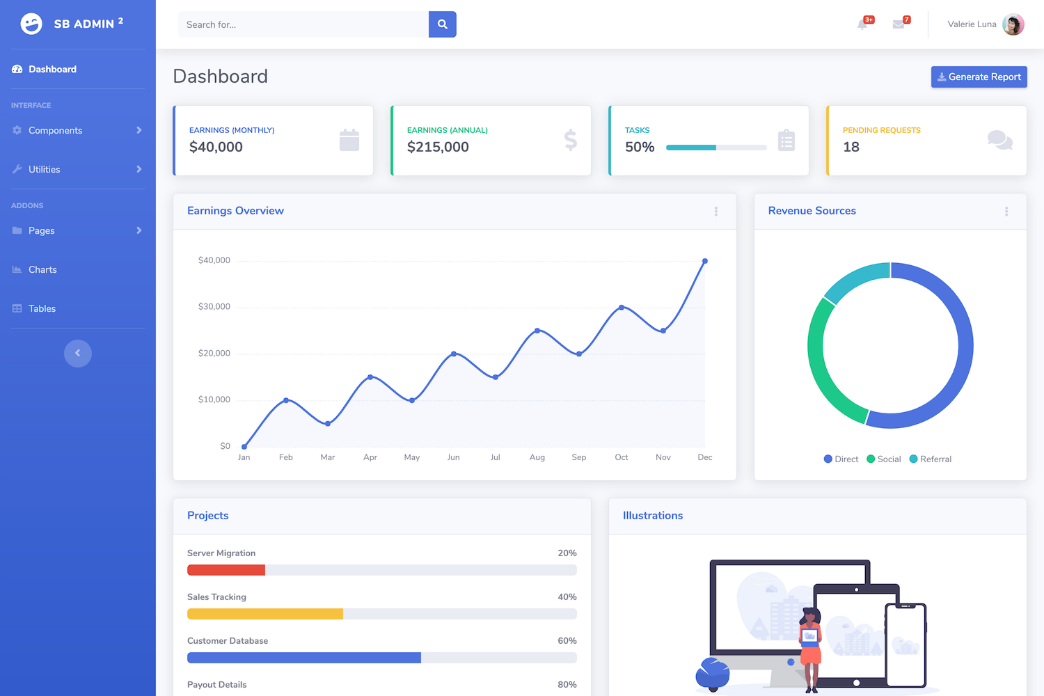


图 9‑5 start bootstrap Admin2模板用有强大的UI组件

二，ECharts

Redis自定义监控系统使用的图表是ECharts。ECharts是开源的商业及数据图表库，它是一个纯Javascript的图表库，可以流畅的运行在PC和移动设备上，兼容当前绝大部分浏览器(IE8/9/10/11,Chrome,Firefox,Safari等)，底层依赖轻量级的Canvas类库ZRender,提供直观，生动，可交互，可高度定制化定制的数据可视化图表。ECharts中加入了更丰富的交互功能以及更多的可视化效果，并对移动端做了深度的优化。

ECharts是由来自百度EFE数据可视化团队开发的，ECharts是基于JavaScript的数据图表库，在编程的灵活性和图表的丰富性方面非常强大，优点很多。

* ECharts是一款独立的Web版数据可视化工具，界面友好，提供强大的互动性操作。
* 对图形参数的修改十分简单，直观，便于初学者使用。
* 丰富的可视化图表，具有高度互动性，这得益于其完善的文档和简单的JavaScript API，相比Matplotlib的图表，更加现代和绚丽。
* 深度的交互式数据探索

提供了图例、视觉映射、数据区域缩放、tooltip、数据刷选等开箱即用的交互组件，可以对数据进行多维度数据筛取、视图缩放、展示细节等交互操作。

* 移动端优化

ECharts 提供了常规的折线图，柱状图，散点图，饼图，K线图，用于统计的盒形图，用于地理数据可视化的地图，热力图，线图，用于关系数据可视化的关系图，treemap，多维数据可视化的平行坐标，还有用于BI的漏斗图，仪表盘，并且支持图与图之间的混搭。读者可以从ECharts的官网下载最新的源码。

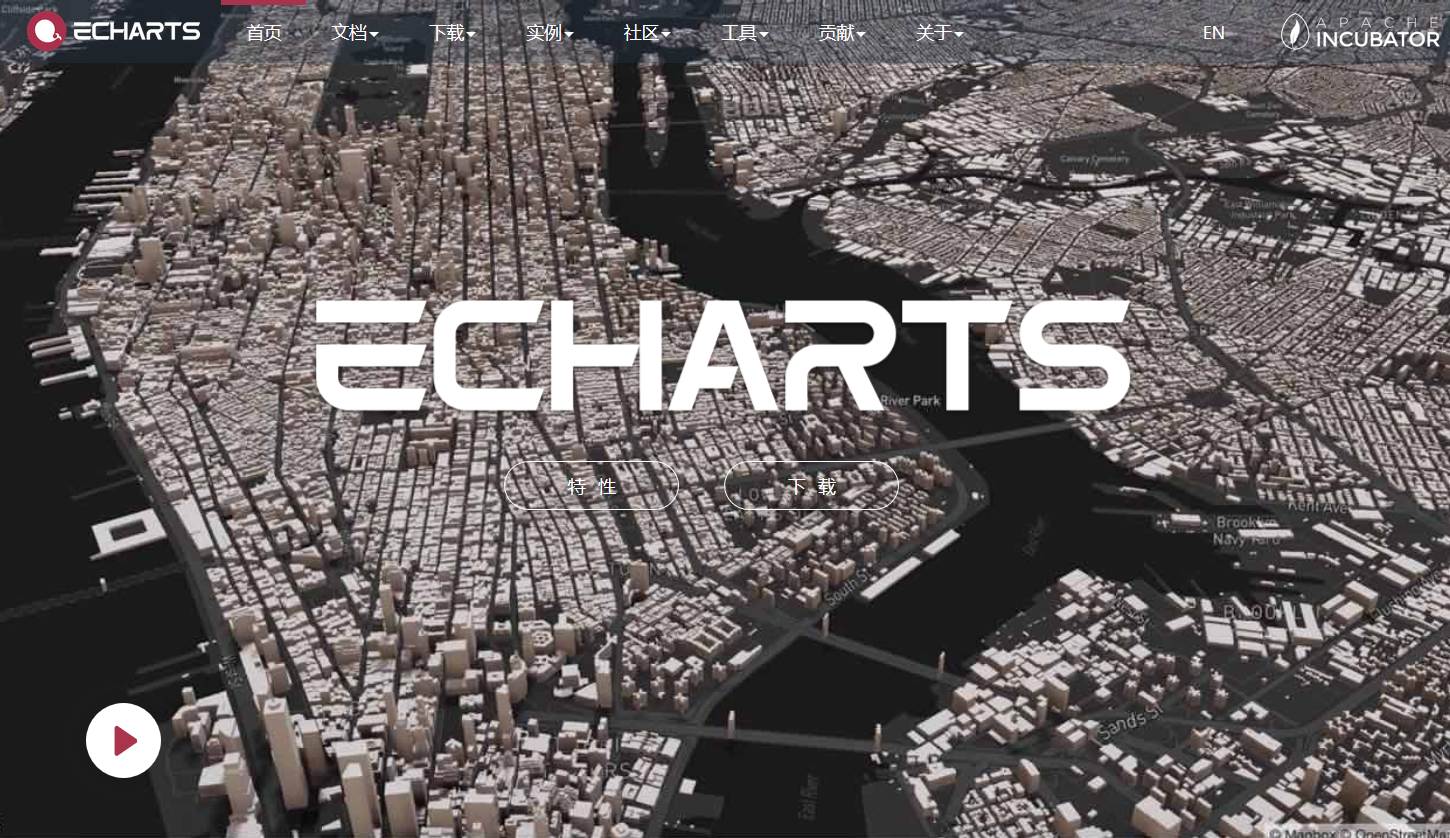


图 9‑6 ECharts官网

Echarts的文档齐全，在文档中对各种图形的案例描述很详细，从官网文档入手是再好不过了，读者可以参考它的官网例子和API参考文档。



图 9‑7 ECharts官网例子

首先从ECharts官网下载需要的版本，根据开发者功能和体积上的需求，ECharts官网提供了不同打包的下载，如果体积上没有要求，可以直接下载完整版本。本书下载的是ECharts完整版本，版本是4.1.0，文件名是echarts.min.js，可以去ECharts官网下载。



图9‑8下载ECharts

本例使用ECharts绘制简单的图表，新建 echarts\_demo目录，在目录下新建文件夹jslib,把下载的echarts.min.js放入到此目录中。本案例中的脚本结构如下。

echarts\_demo01

└─ jslib

└─ echarts.min.js

│─ echarts\_demo.html

然后在echarts\_demo01目录下新建echarts\_demo.html页面。

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title></title>

<!-- 引入 echarts.js -->

<script src="jslib/echarts.min.js"></script>

</head>

<body>

<!-- 为ECharts准备一个具备大小（宽高）的Dom -->

<div id="main" style="width: 600px;height:400px;"></div>

<script type="text/javascript">

// 基于准备好的dom，初始化echarts实例

var myChart = echarts.init(document.getElementById('main'));

// 指定图表的配置项和数据

option = {

xAxis: {

type: 'category',

data: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']

},

yAxis: {

type: 'value'

},

series: [{

data: [820, 932, 901, 934, 1290, 1330, 1320],

type: 'line'

}]

};

// 使用刚指定的配置项和数据显示图表。

myChart.setOption(option);

</script>

</body>

</html>

使用浏览器打开echarts\_demo.html页面，显示效果如下图所示。

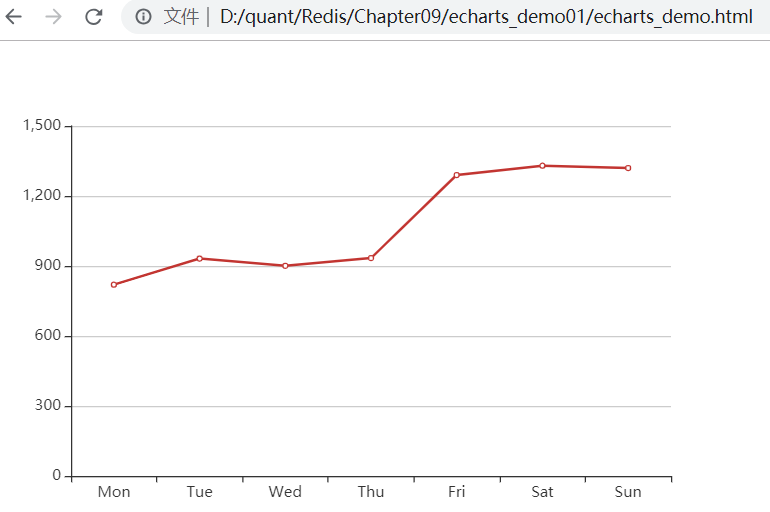


图 9‑9曲线图

本案例中，我们使用ECharts绘制简单的曲线图，有以下知识点需要注意。

1，引入ECharts

ECharts是一个纯JavaScript的图标库，需要像普通的JavaScript库一样用script标签引用。

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<!-- 引入 ECharts 文件 -->

<script src="jslib/echarts.min.js"></script>

</head>

</html>

2, 绘制简单图表

在绘图前需要为 ECharts 准备一个具备高宽的 DOM 容器。

<body>

<!-- 为 ECharts 准备一个具备大小（宽高）的 DOM -->

<div id="main" style="width: 600px;height:400px;"></div>

</body>

然后就可以通过 echarts.init 方法初始化一个 echarts 实例并通过 setOption 方法生成一个简单的曲线图。

<script type="text/javascript">

// 基于准备好的dom，初始化echarts实例

var myChart = echarts.init(document.getElementById('main'));

// 指定图表的配置项和数据

option = {

xAxis: {

type: 'category',

data: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']

},

yAxis: {

type: 'value'

},

series: [{

data: [820, 932, 901, 934, 1290, 1330, 1320],

type: 'line'

}]

};

// 使用刚指定的配置项和数据显示图表。

myChart.setOption(option);

</script>

本案例中的图表的配置项参数可以参考ECharts官网的文档中的配置项手册，其他图形的配置项也是如此。

本案例使用EChart绘制了一个简单的曲线图，没有使用任何图片，显示的柱状图是通过Echarts的JavaScript函数绘制的，图形中的数据和互动也都是通过JavaScript函数实现的，这些JavaScript函数实现细节都封装在EChart图表库里。

本案例中绘制的柱状图虽然在显示上是静止的，但在浏览器中是使用ECharts的JavaScript函数绘制的，支持各种动态互动功能，读者可以好好体会一下。

### Websocket与消息推送

Redis自定义监控系统后端获得的Redis监控信息是通过WebSocket 技术推送到前台页面的，WebSocket是HTML5开始提供的一种浏览器与服务器间进行全双工通讯的网络技术。依靠这种技术可以实现客户端和服务器端的长连接，双向实时通信，如下图所示。

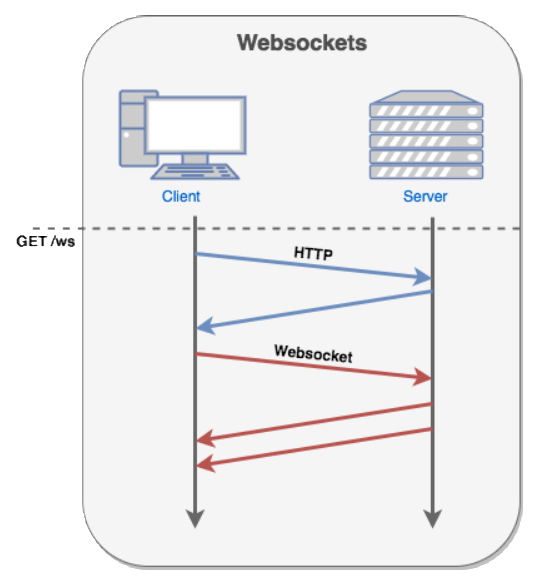


图 9‑10 WebSocket消息推送

1， WebSocket客户端

在客户端，没有必要为 WebSockets 使用 JavaScript 库。实现 WebSockets 的 Web 浏览器将通过 WebSockets 对象公开所有必需的客户端功能（主要指支持 Html5 的浏览器）。

以下 API 用于创建 WebSocket 对象。

var Socket = new WebSocket(url, [protocol] );

以上代码中的第一个参数 url, 指定连接的 URL。第二个参数 protocol 是可选的，指定了可接受的子协议。

当创建了 WebSocket 对象后，可以使用 WebSocket 对象的相关事件。

表格 9‑1 WebSocket对象的相关事件

|  |  |  |
| --- | --- | --- |
| 事件 | 事件处理程序 | 描述 |
| open | Socket.onopen | 连接建立时触发 |
| message | Socket.onmessage | 客户端接收服务端数据时触发 |
| error | Socket.onerror | 通信发生错误时触发 |
| close | Socket.onclose | 连接关闭时触发 |

监控页面的WebSocket核心代码。

<script type="text/javascript">

var websocket = null;

//判断当前浏览器是否支持WebSocket

if ('WebSocket' in window) {

websocket = new WebSocket("ws://localhost:8080/monitor/websocket");

//console.log('websocket=' + websocket);

} else {

alert('当前浏览器 Not support websocket')

}

//连接发生错误的回调方法

websocket.onerror = function() {

setMessageInnerHTML("WebSocket连接发生错误");

};

//连接成功建立的回调方法

websocket.onopen = function() {

setMessageInnerHTML("WebSocket连接成功");

send();

}

//接收到消息的回调方法

websocket.onmessage = function(event) {

jsonObj = JSON.parse(event.data);

//console.log(jsonObj);

renderBoard(jsonObj);

renderChat(jsonObj);

}

function trim(x) {

return x.replace(/^\s+|\s+$/gm, '');

}

//连接关闭的回调方法

websocket.onclose = function() {

setMessageInnerHTML("WebSocket连接关闭");

}

//监听窗口关闭事件，当窗口关闭时，主动去关闭websocket连接，防止连接还没断开就关闭窗口，server端会抛异常。

window.onbeforeunload = function() {

closeWebSocket();

}

//将消息显示在网页上

function setMessageInnerHTML(innerHTML) {

document.getElementById('message').innerHTML += innerHTML + '<br/>';

}

//关闭WebSocket连接

function closeWebSocket() {

websocket.close();

}

//发送消息

function send() {

var message = Math.floor(100000 \* Math.random());

websocket.send(message);

}

</script>

Websocket与Java后台交互将返回以下格式的数据：jsonObj

{

"Stats\_total\_commands\_processed": "416",

"CPU\_used\_cpu\_sys": "0.30",

"CPU\_used\_cpu\_user": "0.27",

"Clients\_connected\_clients": "2",

"Memory\_used\_memory": "0.68",

"Memory\_used\_memory\_rss": "0.64",

"time": "2019-10-07 17:10:27",

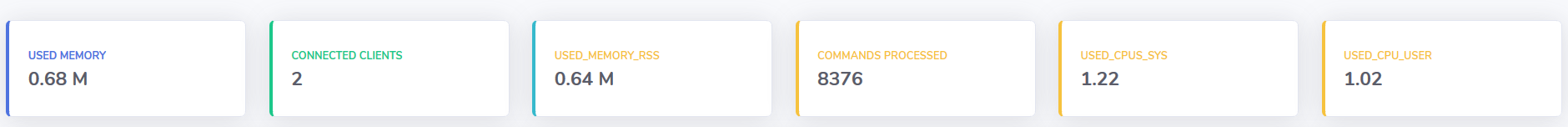
"db0\_keys": "2"

}

返回的数据指标含义如下

|  |  |
| --- | --- |
| 指标 | 内容 |
| Stats\_total\_commands\_processed | 运行以来执行过的命令的总数量 |
| CPU\_used\_cpu\_sys | 内核空间占用CPU百分比 |
| CPU\_used\_cpu\_user | 用户空间占用CPU百分比 |
| Clients\_connected\_clients | 连接客户端数量 |
| Memory\_used\_memory | 使用总内存 |
| Memory\_used\_memory\_rss | 使用物理内存 |
| time | 当前的系统时间戳 |
| db0\_keys |  |

然后使用renderBoard(jsonObj)函数渲染页面上显示的网格，如下图所示



renderBoard()函数的核心代码如下：

<script type="text/javascript">

function renderBoard(jsonObj){

//console.log(jsonObj);

Memory\_used\_memory = jsonObj.Memory\_used\_memory;

Clients\_connected\_clients = jsonObj.Clients\_connected\_clients ;

Memory\_used\_memory\_rss = jsonObj.Memory\_used\_memory\_rss;

Stats\_total\_commands\_processed = jsonObj.Stats\_total\_commands\_processed;

CPU\_used\_cpu\_sys = jsonObj.CPU\_used\_cpu\_sys;

CPU\_used\_cpu\_user = jsonObj.CPU\_used\_cpu\_user;

document.getElementById('Memory\_used\_memory').innerHTML = Memory\_used\_memory + " M";

document.getElementById('Clients\_connected\_clients').innerHTML = Clients\_connected\_clients ;

document.getElementById('Memory\_used\_memory\_rss').innerHTML = Memory\_used\_memory\_rss + " M" ;

document.getElementById('Stats\_total\_commands\_processed').innerHTML = Stats\_total\_commands\_processed ;

document.getElementById('CPU\_used\_cpu\_sys').innerHTML = CPU\_used\_cpu\_sys ;

document.getElementById('CPU\_used\_cpu\_user').innerHTML = CPU\_used\_cpu\_user ;

}

</script>

还使用了 renderChat(jsonObj)函数渲染页面上显示的动态曲线图形，如下图所示

图 9‑11 网格

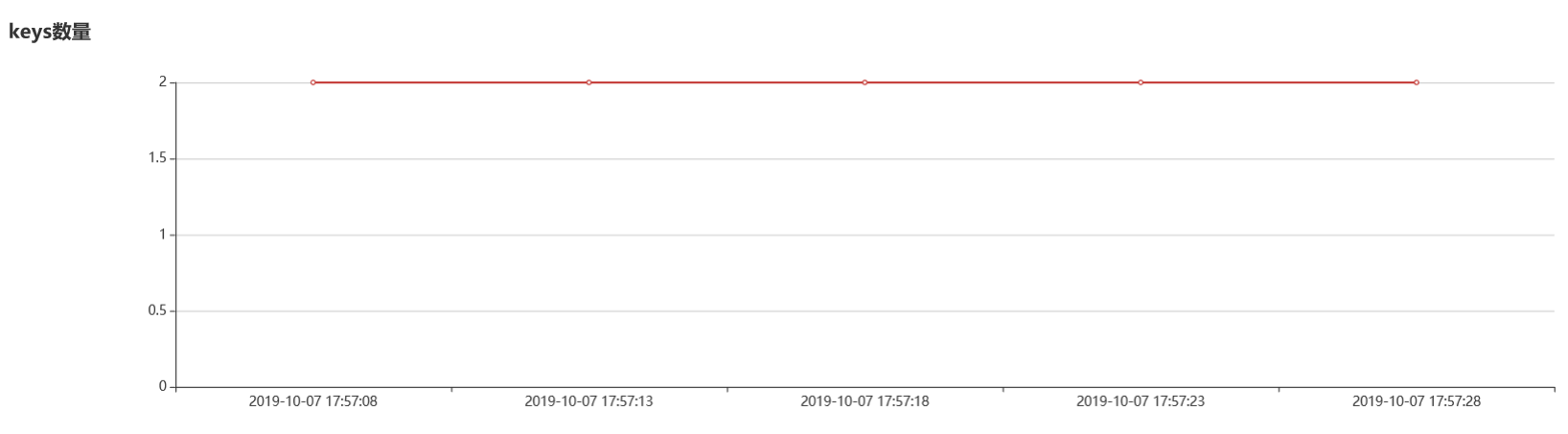


图 9‑12 曲线图

使用Echarts 绘制动态曲线图形的核心代码如下。

首先，在绘图前需要为 ECharts 准备一个具备高宽的 DOM 容器。

<script src="https://cdn.bootcss.com/echarts/3.7.2/echarts.min.js"></script>

<div id="main" style="width: 100%; height: 400px;"></div>

然后，通过 echarts.init() 方法初始化一个 echarts 实例并通过 setOption 方法生成一个动态的的曲线图，为了达到动态曲线图的效果，曲线图最多只显示20个元素，交互数据时删除数组中最后一个元素，达到动态效果。

<script type="text/javascript">

var time\_arr = [];

var data\_arr =[];

function initChat(){

// 基于准备好的dom，初始化echarts实例

var myChart = echarts.init(document.getElementById('main'));

// 指定图表的配置项和数据

option = {

title : {

text : 'keys数量'

},

tooltip : {},

xAxis: {

type: 'category',

data: time\_arr

},

yAxis: {

type: 'value'

},

series: [{

data: data\_arr,

type: 'line',

smooth: true

}]

};

// 使用刚指定的配置项和数据显示图表。

myChart.setOption(option);

}

function renderChat(jsonObj){

keys = jsonObj.db0\_keys;

time = jsonObj.time;

//console.log(keys, time);

// 使曲线图中最多显示20个点

if (time\_arr.length < 5) {

time\_arr.push(time);

data\_arr.push(keys);

}else{

// 删除数组中的第一个元素

time\_arr.splice(0, 1);

data\_arr.splice(0, 1);

time\_arr[time\_arr.length] = time;

data\_arr[data\_arr.length] = keys;

}

//console.log(time\_arr.length);

initChat();

}

initChat();

</script>

### 控制器

创建控制器WebSocketController.java，调用监控类MonitorService获得Redis的性能指标。

package com.dxtd.monitor.action;

import java.io.IOException;

import java.util.concurrent.CopyOnWriteArraySet;

import javax.websocket.OnClose;

import javax.websocket.OnError;

import javax.websocket.OnMessage;

import javax.websocket.OnOpen;

import javax.websocket.Session;

import javax.websocket.server.ServerEndpoint;

import com.dxtd.monitor.service.RedisServiceHandler;

@ServerEndpoint("/websocket")

public class WebSocketController {

// 静态变量，用来记录当前在线连接数。应该把它设计成线程安全的。

private static int onlineCount = 0;

// concurrent包的线程安全Set，用来存放每个客户端对应的MyWebSocket对象。若要实现服务端与单一客户端通信的话，可以使用Map来存放，其中Key可以为用户标识

private static CopyOnWriteArraySet<WebSocketController> webSocketSet = new CopyOnWriteArraySet<WebSocketController>();

// 与某个客户端的连接会话，需要通过它来给客户端发送数据

private Session session;

/\*\*

\* 连接建立成功调用的方法

\*

\* @param session 可选的参数。session为与某个客户端的连接会话，需要通过它来给客户端发送数据

\*/

@OnOpen

public void onOpen(Session session) {

this.session = session;

webSocketSet.add(this); // 加入set中

addOnlineCount(); // 在线数加1

System.out.println("有新连接加入！当前在线人数为" + getOnlineCount());

}

/\*\*

\* 连接关闭调用的方法

\*/

@OnClose

public void onClose() {

webSocketSet.remove(this); // 从set中删除

subOnlineCount(); // 在线数减1

System.out.println("有一连接关闭！当前在线人数为" + getOnlineCount());

if (getOnlineCount() == 0) {

closeQueryThread();

}

}

private Thread queryThread = null;

public void closeQueryThread() {

if (null != queryThread) {

queryThread.interrupt();

}

}

/\*\*

\* 收到客户端消息后调用的方法

\*

\* @param message 客户端发送过来的消息

\* @param session 可选的参数

\*/

@OnMessage

public void onMessage(String message, Session session) {

System.out.println("来自客户端的消息:" + message);

RedisServiceHandler handler = new RedisServiceHandler(this);

queryThread = new Thread(handler);

queryThread.start();

}

/\*\*

\* 发生错误时调用

\*

\* @param session

\* @param error

\*/

@OnError

public void onError(Session session, Throwable error) {

System.out.println("发生错误");

error.printStackTrace();

}

/\*\*

\* 这个方法与上面几个方法不一样。没有用注解，是根据自己需要添加的方法。

\*

\* @param message

\* @throws IOException

\*/

public void sendMessage(String message) throws IOException {

if (this.session.isOpen())

this.session.getBasicRemote().sendText(message);

}

public static synchronized int getOnlineCount() {

return onlineCount;

}

public static synchronized void addOnlineCount() {

WebSocketController.onlineCount++;

}

public static synchronized void subOnlineCount() {

WebSocketController.onlineCount--;

}

}

@ServerEndpoint("/websocket") 注解是一个类层次的注解，它的功能主要是将目前的类定义成一个websocket服务器端, 注解的值将被用于监听用户连接的终端访问URL地址,客户端可以通过这个URL来连接到WebSocket服务器端。本例创建的项目名称是monitor， 部署在端口为8080的Tomcat服务器上，所以本例中WebSocket对象的访问地址是“ws://localhost:8080/monitor/websocket”，对应的HTML5的WebSocket核心代码是如下内容。

var websocket = null;

//判断当前浏览器是否支持WebSocket

if ('WebSocket' in window) {

websocket = new WebSocket("ws://localhost:8080/monitor/websocket");

//console.log('websocket=' + websocket);

} else {

alert('当前浏览器 Not support websocket')

}

### 业务逻辑

创建监控类MonitorService，用来监控Redis状态，发送info命令，解析结果，并对结果进行转换处理。

package com.dxtd.monitor.service;

import java.util.HashMap;

import java.util.Map;

import javax.tools.Tool;

import com.dxtd.monitor.util.RedisUtil;

import com.dxtd.monitor.util.Tools;

import redis.clients.jedis.Jedis;

public class MonitorService {

//调用info 命令

public String getInfo() {

String infoContent = null;

Jedis jedis = null;

try {

jedis = RedisUtil.getJedis();

infoContent = jedis.info();

} catch (Exception e) {

e.printStackTrace();

} finally {

if (jedis != null) {

jedis.close();

}

}

return infoContent;

}

public Double getIntValue(Map<String, String> infoMap, String key) {

if (null == infoMap)

return 0.0;

Double value = Double.valueOf(infoMap.get(key));

return value;

}

public String getStringValue(Map<String, String> infoMap, String key) {

if (null == infoMap)

return "";

return infoMap.get(key).trim();

}

public Integer getKeys(String info, String key) {

Integer keysNumValue = null;

String[] strs = info.split("\n");

if (strs != null && strs.length > 0) {

for (int i = 0; i < strs.length; i++) {

String s = strs[i].trim();

if (s.indexOf(key) > -1) {

String[] str = s.split(",");

if (null != str) {

String[] dbs = str[0].split(":");

String[] dbKeys = dbs[1].split("=");

String keyStr = dbKeys[0];

keysNumValue = Integer.valueOf(dbKeys[1]);

break;

}

}else {

keysNumValue = 0;

}

}

}

return keysNumValue;

}

//解析结果

public Map parseInfo(String content) {

String[] lines = content.split("\n");

Map infoMap = new HashMap();

String part = null;

for (String line : lines) {

if (line.isEmpty()) {

continue;

}

if (line.startsWith("#")) {

part = line.replace("#", "").trim();

continue;

}

int index = line.indexOf(':');

if (index >= 0) {

infoMap.put(part + "\_" + line.substring(0, index), line.substring(index + 1));

}

}

return infoMap;

}

//数据转换处理

public void transData(String infoContent) {

Map<String, String> infoMap = parseInfo(infoContent);

// 内核空间占用CPU百分比

String ucs = getStringValue(infoMap, "CPU\_used\_cpu\_sys");

// 用户空间占用CPU百分比

String ucu = getStringValue(infoMap, "CPU\_used\_cpu\_user");

// 阻塞客户端数量

String cbc = getStringValue(infoMap, "Clients\_blocked\_clients");

// 连接客户端数量

String ccc = getStringValue(infoMap, "Clients\_connected\_clients");

// 使用总内存

String mum = getStringValue(infoMap, "Memory\_used\_memory");

// 使用物理内存

String mur = getStringValue(infoMap, "Memory\_used\_memory\_rss");

// 运行以来执行过的命令的总数量

String cmd = getStringValue(infoMap, "Stats\_total\_commands\_processed");

// 每秒过期key数量

String exp = getStringValue(infoMap, "Stats\_expired\_keys");

// 每秒淘汰key数量

String evt = getStringValue(infoMap, "Stats\_evicted\_keys");

// 每秒命中数量

String hit = getStringValue(infoMap, "Stats\_keyspace\_hits");

// 每秒丢失数量

String mis = getStringValue(infoMap, "Stats\_keyspace\_misses");

Integer db0keysNum = getKeys(infoContent, "db0\_keys");

long thisTs = System.currentTimeMillis();

System.out.println("ucs=" + ucs + ",ucu=" + ucu + ",cbc=" + cbc);

System.out.println("ccc=" + ccc + ",mum=" + mum + ",mur=" + mur);

System.out.println("cmd=" + cmd + ",exp=" + exp);

System.out.println("evt=" + evt + ",hit=" + hit + ",mis=" + mis);

System.out.println("db0keysNum=" + db0keysNum);

}

public Map<String, Object> getRedisInfo(String infoContent) {

Map<String, Object> map = new HashMap<String, Object>();

Map<String, String> infoMap = parseInfo(infoContent);

// 使用总内存

String mum = getStringValue(infoMap, "Memory\_used\_memory");

// 连接客户端数量

String ccc = getStringValue(infoMap, "Clients\_connected\_clients");

// 使用物理内存

String mur = getStringValue(infoMap, "Memory\_used\_memory\_rss");

// 运行以来执行过的命令的总数量

String cmd = getStringValue(infoMap, "Stats\_total\_commands\_processed");

// 内核空间占用CPU百分比

String ucs = getStringValue(infoMap, "CPU\_used\_cpu\_sys");

// 用户空间占用CPU百分比

String ucu = getStringValue(infoMap, "CPU\_used\_cpu\_user");

Integer db0keysNum = getKeys(infoContent, "db0:keys");

map.put("Memory\_used\_memory", Tools.transByteToMBSize(Integer.valueOf(mum )) );

map.put("Clients\_connected\_clients", ccc);

map.put("Memory\_used\_memory\_rss", Tools.transByteToMBSize(Integer.valueOf(mur )) );

map.put("Stats\_total\_commands\_processed", cmd);

map.put("CPU\_used\_cpu\_sys", ucs);

map.put("CPU\_used\_cpu\_user", ucu);

map.put("db0\_keys", db0keysNum+"");

map.put("time", Tools.getCurrntTime());

return map;

}

}

建立RedisServiceHandler类实现一个线程，在这个线程的run()方法里，生成MonistorService类对象来监控Redis的状态，并对info命令的返回值进行处理，然后把返回结果返还给WebSocket的前台页面。

package com.dxtd.monitor.service;

import java.io.IOException;

import java.util.Map;

import com.alibaba.fastjson.JSON;

import com.dxtd.monitor.action.WebSocketController;

public class RedisServiceHandler implements Runnable {

private WebSocketController webSocket;

public RedisServiceHandler(WebSocketController webSocket){

this.webSocket = webSocket;

}

public void run() {

while(true){

try {

MonitorService monitor = new MonitorService();

String info = monitor.getInfo();

Map<String,Object> redisInfo = monitor.getRedisInfo(info);

//群发消息

webSocket.sendMessage(JSON.toJSONString(redisInfo) );

//每5秒发送一次消息，以便页面更新数据

Thread.sleep(1000 \* 5 );

} catch (IOException e) {

e.printStackTrace();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

### 常用工具类

RedisUtil类是连接Redis的工具类，封装了Jedis 线程池的业务逻辑。

package com.dxtd.monitor.util;

import redis.clients.jedis.Client;

import redis.clients.jedis.Jedis;

import redis.clients.jedis.JedisPool;

import redis.clients.jedis.JedisPoolConfig;

public class RedisUtil {

private static JedisPool jedisPool;

/\*\*

\* 建立连接池 真实环境，一般把配置参数缺抽取出来。

\*

\*/

private static void createJedisPool() {

JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();

//指定连接池中最大空闲连接数

jedisPoolConfig.setMaxIdle(10);

//链接池中创建的最大连接数

jedisPoolConfig.setMaxTotal(100);

//设置创建链接的超时时间

jedisPoolConfig.setMaxWaitMillis(2000);

//表示连接池在创建链接的时候会先测试一下链接是否可用，这样可以保证连接池中的链接都可用的。

jedisPoolConfig.setTestOnBorrow(true);

// 创建连接池

jedisPool = new JedisPool(jedisPoolConfig, "127.0.0.1", 6379);

}

/\*\*

\* 在多线程环境同步初始化

\*/

private static synchronized void poolInit() {

if (jedisPool == null)

createJedisPool();

}

/\*\*

\* 获取一个jedis 对象

\*

\* @return

\*/

static {

if (jedisPool == null)

poolInit();

}

/\*\*

\* 归还一个连接

\*

\* @param jedis

\*/

public static void returnRes(Jedis jedis) {

jedis.close();

}

public static Jedis getJedis() {

if (jedisPool == null)

poolInit();

Jedis jedis = jedisPool.getResource();

return jedis;

}

/\*\*

\* 获取redis 服务器信息

\*

\* @param String

\*/

public String getRedisInfo() {

if (jedisPool == null)

poolInit();

Jedis jedis = null;

try {

jedis = jedisPool.getResource();

Client client = jedis.getClient();

client.info();

String info = client.getBulkReply();

return info;

} finally {

// 返还到连接池

if(null != jedis)

jedis.close();

}

}

}

常用工具类Tools，来获得格式化后的时间和转化单位。

package com.dxtd.monitor.util;

import java.text.DecimalFormat;

import java.text.SimpleDateFormat;

import java.util.Date;

public class Tools {

//获得格式化后的时间

public static String getCurrntTime() {

SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

Date now = new Date();

return format.format(now);

}

// 转化单位，从byte到M

public static String transByteToMBSize(int size) {

int MB = 1024 \* 1024;// 定义MB的计算常量

DecimalFormat df = new DecimalFormat("0.00");// 格式化小数

String resultSize = "";

resultSize = df.format(size / (float) MB);

return resultSize;

}

}