spider开发指南

本部分将介绍使用spider RPC开发分布式应用的客户端和服务端。

spider RPC中间件基于J2SE 8开发，因此需要确保服务器上安装了JDK 8及以上版本，除了在可信模式下依赖于轻量级的redis外，不依赖于任何额外需要独立安装和配置的依赖程序。

spider RPC中间件的核心设计初衷是像调用本地服务一样调用远程服务，能够灵活的在运行时确定目标服务在哪台服务器，且高效的管理上百台的大规模服务器集群。

#### 依赖jar包引入

spider包坐标如下：

<dependency>

<groupId>com.ld.net.spider</groupId>

<artifactId>com.ld.net.spider</artifactId>

<version>1.0.0-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>com.ld.net.spider</groupId>

<artifactId>com.ld.net.spider.ext</artifactId>

<version>1.0.0-SNAPSHOT</version>

</dependency>

#### 定义服务接口

开发spider服务的第一步是定义spider服务接口，spider服务以java interface的方式进行定义。

##### POJO

本示例涉及POJO定义如下：

**package** com.medsoft.spider.api;

**public** **class** Demo {

**private** **int** id;

**private** String name;

**private** **double** bonus;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **double** getBonus() {

**return** bonus;

}

**public** **void** setBonus(**double** bonus) {

**this**.bonus = bonus;

}

}

package com.medsoft.spider.api;

import java.util.List;

import java.util.Map;

import com.ld.net.spider.meta.SpiderBizHead;

public class DemoQuery extends SpiderBizHead {

private Map<String,String> param;

private List<Demo> result;

public List<Demo> getResult() {

return result;

}

public void setResult(List<Demo> result) {

this.result = result;

}

public Map<String,String> getParam() {

return param;

}

public void setParam(Map<String,String> param) {

this.param = param;

}

public void addParam(String key,String value) {

this.param.put(key, value);

}

}

##### 接口

**package** com.medsoft.spider.api;

**import** java.util.List;

**import** com.ld.net.spider.annotation.Service;

**import** com.ld.net.spider.annotation.ServiceModule;

@ServiceModule

**public** **interface** DemoDrpcpService {

@Service(desc = "Drpcp修改", serviceId = "99000011")

**public** DemoResp opDrpcpService(DemoReq req);

@Service(desc = "Drpcp list查询", serviceId = "99000012")

**public** List<DemoResp> queryDrpcpService(DemoReq req);

@Service(desc = "Drpcp分页查询", serviceId = "99000002")

**public** DemoQuery queryDrpcpService1(DemoQuery req);

@Service(desc = "调用其他MS", serviceId = "99000003")

**public** String callAS(DemoQuery req);

}

这种方式使用通用的注解进行定义。用于新系统的开发。

@ ServiceModule标识接口DemoDrpcpService为spider服务模块，@Service定义了具体的服务。这两个注解的具体定义以及属性含义参见《金融中间件概要设计-第四节 服务标识》。

需要注意的是，spider运行时要求所有参数必须通过单个DTO进行传递，否则在启动时会出错而终止，这更多的出于服务管理的需要而非技术的限制或性能的下降。

#### 服务端开发

spider服务的实现和标准的j2ee开发一样，只要定义java类实现服务接口即可，没有任何额外的侵入性。

**package** com.medsoft.spider.server;

**import** java.util.ArrayList;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** com.ld.net.base.utils.JsonUtils;

**import** com.ld.net.spider.exception.SpiderException;

**import** com.medsoft.spider.api.DemoDrpcpService;

**import** com.medsoft.spider.api.DemoQuery;

**import** com.medsoft.spider.api.DemoReq;

**import** com.medsoft.spider.api.DemoResp;

**import** com.medsoft.spider.other.api.OtherService;

@Service

**public** **class** DemoDrpcpServiceImpl **implements** DemoDrpcpService {

@Autowired

**private** OtherService otherService;

@Override

**public** DemoResp opDrpcpService(DemoReq req) {

System.***out***.println("接收到spider调用请求: " + JsonUtils.*toJson*(req));

**return** **new** DemoResp();

}

@Override

**public** DemoQuery queryDrpcpService1(DemoQuery req) {

System.***out***.println("接收到spider调用请求: " + JsonUtils.*toJson*(req));

**return** **new** DemoQuery();

}

@Override

**public** String callAS(DemoQuery req) {

System.***out***.println("调用其他系统的AS");

**try** {

**return** otherService.doAs(req);

} **catch** (SpiderException e) {

**throw** e;

}

}

@Override

**public** List<DemoResp> queryDrpcpService(DemoReq req) {

List<DemoResp> list = **new** ArrayList<DemoResp>();

DemoResp e1 = **new** DemoResp();

DemoResp e2 = **new** DemoResp();

list.add(e1);

list.add(e2);

**return** list;

}

}

如上所示，实现非常简单，没有任何spider侵入性，故不再重复讲解。

需要注意的是，上述红色字体的OtherService，该服务为标准Spring Bean服务或者微服务MS或其它子系统提供的公用业务服务，比如查询账户信息，视具体而定。

#### 客户端开发

##### 依赖注入

客户端调用spider服务在开发上和调用标准的本地spring bean服务完全相同，只要设置目标服务自动注入就可以直接使用了，如下所示：

@Controller

**public** **class** IndexAction {

@Autowired

**private** DemoDrpcpService demoDrpcpService;

@RequestMapping(value="/to\_pnp2\_cnp1.html",method=RequestMethod.***GET***)

**public** @ResponseBody String indexV(Model model,HttpServletRequest request){

DemoQuery req = **new** DemoQuery();

DemoReq param = **new** DemoReq();

req.setAppVersion("v1.1");

req.setSystemId("02");

req.setCompanyId("100001");

param.setId(8888);

param.setName("zhangsan");

**return** JsonUtils.*toJson*(demoDrpcpService.queryDrpcpService1(req));

}

}

##### 调用功能号

除了上述通过注入接口方式调用spider服务外，spider运行时还支持直接通过功能号的方式调用，如下所示：

DemoReq param = **new** DemoReq();

param.setId(8888);

param.setName("zhangsan");

param.setCompanyId(companyId);

param.setSystemId(systemId);

ServiceDefinition service = ServiceDefinitionContainer.*getService*("99000011");

**try** {

**return** (DemoResp) service.getMethod().invoke(BeanManagerHelper.*getBean*(service.getClz()), param);

} **catch** (IllegalAccessException | IllegalArgumentException

| InvocationTargetException e) {

e.printStackTrace();

}

两种方式的调用效果相同。

#### spider过滤器开发DEMO

TODO。

#### 接口包与实现包的部署

使用Spider开发的分布式系统最简单的情况下通常具有如下的部署结构：



在复杂的生产部署中，其部署结构可能会异常复杂，如下所示：



每个节点的含义及其业务、技术角色可见《金融中间件架构设计》相关章节。

本示例中，我们以简单的客户端/服务端为例介绍包的部署与配置文件的设置。

spider客户端只需要放置定义服务接口的jar或class即可。

spider服务端需要同时放置定义服务接口的jar或class，和服务实现。一般来说，这两者都打包在jar中而非分散的class会比较合理。

#### Spider配置

Spider包含一个配置文件，名称为spider.xml，可以修改，具体位置可参见《spider概要设计-配置文件一节》，目前支持三种配置文件指定方式，在标准的maven工程下，建议将其放在src/main/resources下，作为标准配置文件的一部分。

##### 客户端spider.xml示例

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<spider>

<nodeName value=*"ANB"* cloud=*"false"* role=*"production"*

serviceCenter=*"0.0.0.0"* appVersion=*""* charset=*"GBK"* />

<plugins>

<plugin pluginId=*"spider.localService"* serviceTimeout=*"10000"*

zlibCompress=*"false"* encrypt=*"false"* anonymous=*"true"*

serviceProxyPackage=*"com.medsoft.spider.api"*>

<server enable=*"false"* port=*"7070"* reliable=*"false"*

threadCount=*"200"* serviceExportPackage=*""* />

</plugin>

<plugin pluginId=*"spider.channel"*>

<cluster clusterName=*"CNB-1"* connectionSize=*"1"*>

<workNode address=*"127.0.0.1"* port=*"18021"* />

</cluster>

</plugin>

<plugin pluginId=*"spider.filter"*>

</plugin>

</plugins>

<routeItems>

<routeItem serviceId=*"99\*"* clusterName=*"CNB-1"* />

<!-- <routeItem serviceId="\*" appVersion="" subSystemId=""

systemId="" companyId="" clusterName="spider-server" /> -->

</routeItems>

</spider>

上述配置中，红色部分是与客户端直接相关的配置。

详细含义以及默认值参见《金融中间件概要设计-第五章节 配置文件》。

##### 服务端spider.xml示例

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<spider>

<nodeName value=*"MSNP-1"* cloud=*"false"* role=*"production"*

serviceCenter=*"0.0.0.0"* appVersion=*""* serviceDefineType=*"spider"* needLdPackAdapter=*"false"* charset=*"GBK"*/>

<plugins>

<plugin pluginId=*"spider.localService"* serviceTimeout=*"10000"*

zlibCompress=*"false"* encrypt=*"false"* anonymous=*"true"*

serviceProxyPackage=*""*>

<server enable=*"true"* port=*"18051"* reliable=*"false"*

threadCount=*"200"* serviceExportPackage=*"com.medsoft.spider.api"* />

</plugin>

<plugin pluginId=*"spider.channel"*>

</plugin>

<plugin pluginId=*"spider.filter"*>

</plugin>

</plugins>

<routeItems>

<routeItem serviceId=*"\*"* clusterName=*"spider.localService"* />

<!-- <routeItem serviceId="\*" appVersion="" subSystemId=""

systemId="" companyId="" clusterName="spider-server" /> -->

</routeItems>

</spider>

上述配置中，红色部分是与客户端直接相关的配置。

详细含义以及默认值参见《金融中间件概要设计-第五章节 配置文件》。

#### 服务端配置注意事项

spider启动时会进行下列自检操作：

1. 在/tmp/spider/下检查是否存在${nodeName}.pid文件，如果存在，则说明本服务器上存在同名已启动spider实例，不允许启动；
2. 运行服务器模式时，检查本节点spider.xml中定义的服务器端口号是否已经被占用，如果已经被占用，则提示端口号已经被占用，不允许启动；

#### 运行方式

当前版本spider支持运行于标准java应用程序和web容器下两种模式，他们提供完全相同的核心服务。在运行于标准java程序模式下时，spider提供的RESTFUL管理控制台和API将不可用。

##### 标准java程序启动Demo

package com.ld.net.spring;

import java.io.IOException;

import java.io.InputStream;

import java.util.Properties;

import org.apache.log4j.PropertyConfigurator;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.util.StringUtils;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

/\*\*

\* @author zhjh256@163.com

\* {@link} http://www.cnblogs.com/zhjh256

\*/

public class StartServer {

static final Logger logger = LoggerFactory.getLogger(StartServer.class);

public static void main(String[] args) {

PropertyConfigurator.configure("classpath:log4j.properties");

InputStream in = StartServer.class.getClassLoader().getResourceAsStream("server.properties");

Properties p = new Properties();

String[] springFiles = null;

try {

p.load(in);

springFiles = p.get("springFiles").toString().split(",");

} catch (NullPointerException | IOException e) {

logger.warn("classpath下找不到配置文件server.properties或者配置文件中没有属性springFiles,根据默认规则加载配置文件!");

e.printStackTrace();

}

if (springFiles == null) {

String path = StartServer.class.getResource("").toString();

springFiles = new String[] {"classpath\*:spider-base-service.xml","spring-\*.xml"}; // spring-\*.xml为其它要加载的启动文件

}

logger.info("启动配置文件:" +StringUtils.arrayToCommaDelimitedString(springFiles));

ApplicationContext atx = new ClassPathXmlApplicationContext(springFile);

while(true) {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

##### Web容器模式

web.xml配置。客户端和服务端的web.xml配置要求相同，如不需要启用web监控，只需要包含如下即可：

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath\*:spider-base-service.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

如需启用web监控，则还需要包含如下：

<servlet>

<servlet-name>springMVC</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath\*:spider-base-web.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

#### 启动运行

上述步骤均完成后，就可以启动spider服务端和客户端进行验证了。Spider服务端和客户端的启动顺序无关紧要，spider运行时会每隔指定间隔自动进行检测并尝试建立断开和尚未建立的连接。

### 性能优化

为了支持上千的并发连接，请确保linux内核参数somaxconn大于2000。

Spider中间件运行时会自动根据服务器配置调节线程池大小，一般来说无需进行特定的优化。

#### 可信模式下特殊优化

Spider在可信模式下使用redis作为存储机制，请确保服务器可用于redis的内存不低于4GB、同时设置AOF每秒1次以平衡性能和高可靠性，在HA模式下确保本节点和远程redis节点间的网络延迟不高于1ms。

### 异常处理

对于spider异常号011，超时，意味着请求有可能执行成功、有可能执行失败，因为在请求达到spider服务端到请求执行完成返回给客户端这段时间，服务器和客户端之间的网络链路完全有可能断了，即使心跳检测优化为自动重连时候使用和原来一样的端口号，也有可能这段时间完全有可能这个端口被分配给其他应用了，这会导致程序异常复杂，且服务端不会认为这是一条相同的通道。所以业务应保证可重复执行。