

# JSF客户端用户手册

工具

被章耽添加，被章耽最后更新于十一月 04, 2016

- [说明](#)
- [功能列表](#)
  - [配置方式](#)
  - [服务注册与订阅](#)
  - [集群策略](#)
  - [负载均衡](#)
  - [服务依赖检查](#)
  - [直连调用](#)
  - [异步调用](#)
  - [异步回调](#)
  - [泛化调用](#)
  - [inivm调用](#)
  - [callback调用](#)
  - [多协议支持](#)
  - [非守护启动](#)
  - [延迟启动](#)
  - [延迟连接](#)
  - [粘滞连接](#)
  - [参数校验](#)
  - [请求上下文](#)
  - [隐式传参](#)
  - [Token调用](#)
  - [指定发布IP](#)
  - [指定注册IP](#)
  - [指定端口](#)
  - [方法限制](#)
  - [内置Filter](#)
  - [自定义Filter](#)
  - [连接监听](#)
  - [服务端动态分组](#)
  - [服务质量监控](#)
  - [Telnet运维](#)
  - [服务列表备份](#)
  - [代理类生成方式](#)
  - [序列化方式](#)
  - [线程名称和数量](#)
  - [线程池类型](#)
  - [事件分发类型](#)
  - [Meek调用（1.0.x版本）](#)
  - [Mock调用（1.2.x版本）](#)
  - [服务降级（1.0.x版本）](#)
  - [数据包大小限制](#)
  - [调用压缩](#)
  - [并发控制](#)
  - [灰度部署](#)
  - [结果缓存](#)
  - [黑白名单](#)
  - [路由配置](#)
- [功能成熟度](#)

## 说明

名词解释：

注册中心（Registry Server）：提供服务信息的注册订阅，配置管理，配置下发等功能。

注册中心寻址服务（Index Server）：简单的http的注册中心寻址。

注册中心管理端（jsf.jd.com）：管理端，用户可以登录管理自己的服务。

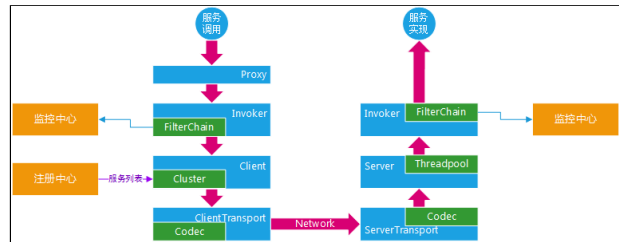
服务（Interface）：就是一个接口。

服务提供者（Provider）：启动一个监听端口，并发布服务。

服务调用者（Consumer）：连接Provider启动的端口，并调用服务。

监控中心（Monitor Server）：收集服务质量的监控中心服务。

JSF客户端（jsf.jar）：使用了jsf.jar的程序，不管是服务提供者和服务调用者。



## 功能列表

### 配置方式

配置项参考：[JSF配置参考手册](#)

目前支持三种配置方式：

名称	配置	说明
Spring+XML（推荐）	基于spring的schema校验做了扩展。推荐使用这种方式。 只需要在spring的配置文件中加入头，就可以方便的发布和调用服务。 无任何代码入侵，可扩展性强，修改配置只需要修改文件即可。	代码参见： <a href="#">JSF入门指南</a> <a href="#">JSF入门指南#2.DEMO下载</a>
Spring+annotation（不推荐）	使用注解方式，代码入侵，开发比较方便。 但是上线后修改必须重新打jar包，可扩展非常差。	代码参见： <a href="#">JSF入门指南#2.DEMO下载</a>
API方式	普通的javabean方式，代码入侵。 适用于需要动态发布和调用服务的场景，例如网关程序	代码参见： <a href="#">API方式使用说明</a> <a href="#">JSF入门指南#2.DEMO下载</a>

### 服务注册与订阅

JSF可配置多个注册中心（不过暂未开放）。如果用户未指定任何注册中心，JSF会默认生成一个注册中心。

```
<jsf:registry id="jsfRegistry" index="i.jsf.jd.com" />

<jsf:provider registry="jsfRegistry" /> <!-- 发注册中心jsfRegistry -->
<jsf:provider /> <!-- 未配置"registry" 则默认发全部jsf:registry 即注册中心1,2 -->
```

在启动Provider，会自动去指定注册中心注册一条Provider信息，同时订阅接口配置。

（不推荐）可以通过如下配置实现不注册和不订阅。

```
<jsf:provider register="false" subscribe="false" />
```

在启动Consumer的时候，会自动去注册中心注册一条Consumer信息，同时订阅服务列表，订阅接口配置。

（不推荐）可以通过如下配置实现不注册和不订阅。

```
<jsf:consumer register="false" subscribe="false" />
```

### 集群策略

Consumer调用时，可以设置调用集群策略，配置如下：

```
<jsf:consumer cluster="failover" />
```

目前支持如下几种集群策略：

名称	配置	说明
失败重试（默认）	failover	1. 非业务失败则自动切换到下一台重新发起调用，增强用户体验。 2. 重试可能会导致调用时间更长（例如2s超时，重试2次，耗时最长可能会是6秒） 3. 重试次数目前默认不重试，可以通过配置设置。 <jsf:consumer retries="2" /> 表示整个接口下方法都失败后重试2次。 <jsf:method method="hello" retries="1" /> 表示hello这个方法失败后重试1次。
失败忽略	failfast	1. 只发起一次调用，失败立即报错。约等于不重试的failover
广播调用	broadcast	1. 广播效果，逐个调用全部可用的Provider，失败的跳过。 2. 不支持异步调用，返回值也只有最后一个。
定点调用	pinpoint	1. 只适用于直连，因为需要提前知道Provider地址。参见 <a href="#">直连调用</a> 1.6.0开始也支持注册中心 2. 可以指定发送给那个Provider，该Provider不可用直接抛出异常 可在调用前设置地址： RpcContext.getContext().setAttachment(Constants.HIDDEN_KEY_PINPOINT, "jsf://127.0.0.1:10990") 3. 未指定发送Provider时等同于failover

### 负载均衡

在Consumer调用，会对服务列表进行软负载均衡，配置如下：

```
<jsf:consumer loadbalance="random" />
```

目前支持如下几种策略。

名称	配置	版本	优点	缺点
随机（默认）	random	1.0.0	1. 按权重的进行随机，可以方便的调整 2. 调用量越大分布越均匀	
轮询	roundrobin	1.0.0	1. 轮循，按公约后的权重设置轮循比率	如果某个Provider较慢，可能会积压请求
最少并发优先	leastactive	1.0.0	1. 在调用前后增加计数器，并发最小的Provider认为是最快的Provider 并发大的Provider认为是慢的Provider。 2. 并发数相同再比较权重。 3. 这样快的Provider收到更多请求。	快速抛异常的Provider有可能被误认为是响应快的Provider。  目前此问题已解决，Provider抛异常会降低选中概率，最低到10%，等Provider恢复并调用成功后，选中概率会恢复到100%。
一致性hash	consistenthash	1.2.0	1. 服务端节点没变化的情况下，同样的请求（根据第一个参数值进行hash）会指向同一台机器 2. 基于虚拟节点，如果一台挂了也没事，会	可能调用分布不均匀。

			半推到其它提供者	
本机IP调用优先	localpref	1.5.1	1. 本机IP上的Provider优先调用 2. 匹配到多个使用随机调用本机IP，未匹配也随机调用远程机器	计算相同IP稍微增加点耗时。

服务依赖检查

Consumer启动时候会去连Provider，默认Consumer是不强依赖与Provider的。

也就是说即使没有服务提供者，服务调用者也照常启动（建长连接失败可能会打印些异常）。只不过是调用时报错。

如果Consumer是强依赖于Provider的，可以通过如下配置进行检查。

<jsf:consumer check="true" /> <!-- 默认 check="false" -->

这样的话，如果没有可用的Provider，Consumer就启动失败，上线时就能提前暴露出问题所在。

直连调用

Consumer调用的时候可以不通过注册中心，直接指定Provider地址进行远程调用。

这样就无法从注册中心拿到新的服务列表地址，可扩展性差。

✔ 可用于开发和测试环境，不推荐线上生产环境使用。

参考:

<!-- 格式为 协议://ip:port/?key=value, 多个用英文逗号或英文分号隔开-->  
<jsf:consumer url="jsf://127.0.0.1:11090,jsf://127.0.0.1:11091" />

也可以强制调一个别的别名（适用于动态分组等场景）。

例如：服务提供者别名是yyy，动态追加一个分组xxx，老的xxx需要调用这个分组时得带上原始别名。

<!-- 格式为 协议://ip:port/?key=value, 多个用英文逗号或英文分号隔开-->  
<jsf:consumer alias="xxx" url="jsf://127.0.0.1:11090?alias=yyy" />

异步调用

异步调用指的Consumer发起调用后，不是等待拿到结果，而是返回null同时可以拿到一个Future，过一会再拿Future去取。

适用于可以一次性并行发起多个请求的环境（例如某些集成页面，后台同时调用多个系统），总耗时等于最长时间的调用。

例如：一个页面要同时调用A(耗时600ms)，B(耗时500ms)，C(耗时100ms) 三个服务，如果采取线性调用，则需要1200ms；采取异步调用则只需要600ms。

使用方式为：在调用端配置加入如下代码：

<jsf:consumer async="true" /> <!-- 所有方法异步 默认 async="false" />

<jsf:consumer >  
  <jsf:method name="xxx" async="true" /> <!-- 指定方法异步 默认 async="false" />  
</jsf:consumer>

在调用的时候采用如下代码：

String result1 = service.echoStr("aaaa"); // 发起一次调用 此时返回null  
ResponseFuture<String> future1 = RpcContext.getContext().getFuture(); // 得到第一次调用的Future  
String result2 = service.echoStr("bbbb"); // 发起第二次调用 此时返回null  
ResponseFuture<String> future2 = RpcContext.getContext().getFuture(); // 得到第二次调用的Future  
String result3 = service.echoStr("ccc"); // 发起第三次调用 此时返回null  
ResponseFuture<String> future3 = RpcContext.getContext().getFuture(); // 得到第三次调用的Future  
// 依次拿到结果，一定要get，否则会造成结果未消费导致内存溢出。  
try {  
  result1 = future1.get(); // 可能会抛异常，请捕获  
} catch (Throwable e) {  
}  
try {  
  result2 = future2.get(); // 可能会抛异常，请捕获  
} catch (Throwable e) {  
}  
try {  
  result3 = future3.get(); // 可能会抛异常，请捕获  
} catch (Throwable e) {  
}

如果future1.get()的时候时间较长，则会阻塞等待结果，不过没关系，如果future1时间最长，那么future2和3能直接拿到结果，无需等待了。

New! 新增加函数式方法

ResponseFuture<String> future4 = ResponseFuture.build(new ResponseFuture.AsyncCall() {  
  public void invoke() throws Throwable {  
    service.get(1); // 业务调用代码  
  }


```

    }
});
try {
    String result4 = future4.get();
} catch (Throwable e) {
}

```

## 异步回调

异步调用的时候，大多数情况，我们是知道什么时候拿Future是get结果的，比如上面的例子，我们知道发完请求就可以拿结果。但是有的时候，我们希望异步调用有结果回来的时候，再触发下调用者的一些逻辑。

 异步回调的到时候，最好是限制发送的频率，要不然发送太快，一直不停的发，会导致内存溢出等。

做法如下：

1. 需要实现一个Listener实现com.jd.jsf.msg.ResponseListener接口，实现handleResult和catchException方法。

com.jd.testsaf.listener.TestResponseListener

```

package com.jd.testjsf.listener;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import com.jd.jsf.msg.ResponseListener;
public class TestResponseListener implements ResponseListener {
    private final static Logger LOGGER = LoggerFactory.getLogger(TestResponseListener.class);

    public void handleResult(Object result) {
        LOGGER.info("interface listener get result :{}", result);
    }
    public void catchException(Throwable e) {
        LOGGER.info("interface listener catch exceprion :{} {}", e.getClass().getCanonicalName(),
e.getMessage());
    }
}

```

2. 在接口级注入Listener。Listener是否单例可以自己通过bean控制。

```

<bean id="responseListener" class="com.jd.testjsf.listener.TestResponseListener"/>
<jsf:consumer onreturn="responseListener" async="true" /> <!-- 异步 且有回调监听器 -->

```

或者方法级注入。

```

<jsf:consumer async="true" >
    <jsf:method name="get" onreturn="responseListener" /> <!-- get方法有回调监听器 -->
</jsf:consumer>

```

3. 调用的时候和异步的一样，返回值为null，等到有结果返回会调用注入的responseListener

```

String result1 = service.echoStr("aaaa"); // 发起一次调用 此时返回null
// 无需再getFuture，等有结果会自动回调ResponseListener里的方法
//Future<String> future1 = RpcContext.getContext().getFuture();

```

4. 如果想每次请求都采用新的ResponseListener，而不像如果采用spring配置的方式那样，方法全局一个的话，可以采用以下方式。注意只支持API的方式，无法通过配置注入。

此特性需要1.6.1以上版本


```

RpcContext.getContext().getAttachments().put(Constants.INTERNAL_KEY_ONRETURN, new ResponseListener() {
    private int t = times;

    @Override
    public void handleResult(Object result) {
        System.out.println("method--"+t+"次:result:"+result);
    }
    @Override
    public void catchException(Throwable e) {
        System.out.println("method exception--"+e.getMessage());
    }
}); //在方法调用前注入responseListener实例

service.echoStr("test method response listener");

```

 此操作会分发到调用端对应的回调线程池上执行，线程池大小和队列可以通过<jsf:parameter /> 配置。

如果要自定义回调线程池的大小，可以使用如下参数：

```

<jsf:parameter key="callback.pool.coresize" value="20" />
<jsf:parameter key="callback.pool.maxsize" value="200" />
<jsf:parameter key="callback.pool.queue" value="1024" />

```

## 泛化调用

在网关等场景下，调用者是拿不到服务端的class或者jar包，那此时发起调用，改如何处理？

JSF支持泛化调用，只需要指定接口、方法、参数类型、参数值，就可以完成调用。

参见 [Generic调用说明](#)

injvm调用

在Consumer进行调用的时候，如果在同一JVM内有Provider，则优先调用本地JVM内的Provider，不走远程调用。

injvm调用不会影响统计结果。

可以通过如下配置关闭此特性。

```
<jsf:consumer injvm="false" /> <!-- 默认injvm="true" -->
```

callback调用

JSF协议为长连接，好处是可以进行服务端到调用端的callback调用。

使用方法如下：

- 1、方法上增加一个参数，类型为com.jd.jsf.transport.Callback<T>，需要指定回调的对象。

```
public interface HelloService {
    public void callBackString(String request, Callback<String> resultListener);
}
```

- 2、调用端调用的时候，直接注入一个Callback实现对象。

```
Callback callback = new Callback<String>() {
    public void notify(String result) {
        logger.info("notify callback string : {}", result);
    }
};
service.callBackString("hello", callback ); // 每次使用同一个callback，服务端也得到同一个。（推荐）
// service.callBackString("hello", new Callback<>() ); // 每次new一个callback，服务端每次拿到一个新的callback（不推荐）
```

如果Consumer调用的时候使用的时候同一个Callback对象，那么Provider端也只会生成一个Callback对象。

如果Consumer调用的时候使用的时候每次new一个Callback对象，那么Provider端每次得到一个新的Callback对象。客户端同一个接口类有限制，最多存在2000个Callback。

- 3、服务端实例类里面可以使用callback对象回调。

```
public class HelloServiceImpl implements HelloService {
    public void callBackString(String request, Callback<String> resultListener) {
        resultListener.notify("callback from server");
    }
}
```

也可以将Callback对象缓存起来，一直使用。（有点像注册机制）

如果连接已断开，Callback回调时会失败，抛出com.jd.jsf.error.CallbackStubException异常。如果已缓存可以删除。

多协议支持

目前jsf支持多种协议，只需要通过配置就可以完成发布和调用。

支持的协议如下：

协议	配置	连接方式	说明	备注
jsf协议	jsf（同时支持tcp+http）	tcp长连接	自定义协议，默认采用msgpack序列化	兼容dubbo协议调用 <a href="#">JSF与SAF的兼容与比较</a>
http协议	jsf（同时支持tcp+http）	http短连接		<a href="#">使用HTTP调用JSF接口</a>
dubbo协议	dubbo	tcp长连接	自定义协议，默认采用hessian序列化	请使用saf发布dubbo接口
rest协议	rest	http短连接	通过resteasy发布webservice	<a href="#">使用JSF发布REST协议接口</a>
webservises	webservice/jaxws	http短连接	通过cxf发布webservice	<a href="#">使用JSF发布Webservice协议接口</a>

不建议使用JSF发布dubbo协议的服务端，因为JSF和SAF可以在项目中共存。参见[JSF与SAF的兼容](#)

Provider通过配置指定协议。

```
<jsf:server id="myjsf" protocol="jsf" />
<jsf:server id="myrest" protocol="rest" />
<jsf:provider server="myjsf,myrest" /> <!-- 一个接口同时发两种协议 -->
```

Consumer通过配置指定协议。

```
<jsf:consumer protocol="jsf" />
```

非守护启动

服务发布的时候，默认服务端的线程是守护线程，那么Provider在export之后，需要手动的hold住主线程，要不然会程序会退出。

如果希望启动的是非守护线程，可以通过如下配置实现：

```
<jsf:provider daemon="false" /> <!-- 默认true -->
```

## 延迟启动

服务发布的时候，spring在加载到jsf.provider的时候默认会马上发布服务。

如果需要数据预热等需求，需要延迟一段时间再发布服务，可以通过如下配置实现：

```
<jsf:provider delay="10000" /> <!-- 表示延迟十秒发布，默认0 -->
```

在Spring集成的发布方式下，还支持一种spring加载完毕后再发布服务，配置如下：

```
<jsf:provider delay="-1" /> <!-- 表示Spring加载完毕后再发布服务 -->
```

## 延迟连接

在调用端拿到服务列表的时候，回去建立长连接，默认为初始化的时候创建，

如果希望第一次调用的时候创建，可以加入如下配置：

```
<jsf:consumer lazy="true" /> <!-- 默认 lazy="false" -->
```

长连接会在第一次调用的时候再进行创建。

## 粘滞连接

调用端拿到服务列表建立连接后，在进行调用的时候，每次都会根据负载均衡算法（例如随机，轮询）等对可用连接进行重新选择。

如果希望每次都调一个服务端，直到出异常为止，再调下一个，可以加入如下配置：

```
<jsf:consumer sticky="true" /> <!-- 默认 sticky="false" -->
```

## 参数校验

JSF在Consumer调用之前（推荐）可以进行参数校验操作，不通过的请求不能发给Provider。

同时也可以在Provider调业务代码之前进行参数校验操作，不通过的请求不调业务代码。

参见 [使用JSF的参数校验（Validation）功能](#)

## 请求上下文

在调用过程中，会有一个RpcContext记录一些请求的相关信息。

 RpcContext是基于ThreadLocal的，每次调用都会重新清空。所以Consumer每次调用前都要设置，Provider每次要接到请求就先取值。  
在调用链中，例如A→B→C，那么B机器在B调C之前，RpcContext拿到的是A→B的信息，B调C之后，RpcContext拿到的B→C的信息。

例如Provider可以获取获取远程地址信息（调用端IP）：

```
InetSocketAddress address = RpcContext.getContext().getRemoteAddress();  
String ip = RpcContext.getContext().getRemoteHostName();
```

异步调用的时候可以获取Future对象信息：

```
Future future = RpcContext.getContext().getFuture();
```

可以获取当前服务的所处角色：

```
boolean isProvider = RpcContext.getContext().isProviderSide();  
boolean isConsumer = RpcContext.getContext().isConsumerSide();
```

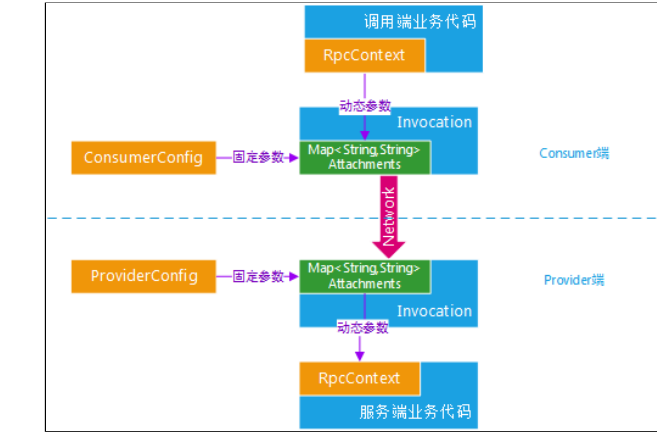
Provider可以获取Consumer的自动部署appId信息（**Provider1.5.2+**，**Consumer1.2.0+**且接入自动部署，其实读取**Consumer**启动脚本里的 **-Ddeploy.app.id=4556 -Ddeploy.instance.id=12345 -Ddeploy.app.name=jsf-web**）。

```
String appId = (String) RpcContext.getContext().getAttachment(Constants.HIDDEN_KEY_APPID); // 自动部署应用Id  
String appName = (String) RpcContext.getContext().getAttachment(Constants.HIDDEN_KEY_APPNAME); // 自动部署应用名称  
String appInsId = (String) RpcContext.getContext().getAttachment(Constants.HIDDEN_KEY_APPINSID); // 自动部署应用实例Id
```

另外：也可以自定义设置和获取一些隐藏的参数信息，参见下面的[隐式传参](#)

## 隐式传参

在Consumer调用的时候，除了发送方法需要的参数以为，还支持发送一个隐藏的Map<String,String>到服务端。



Consumer目前可以通过XML配置（API配置）指定每次调用都发送的固定的隐藏参数，或者在代码中每次调用都重新设置的动态的远程参数。

Consumer端：

1.spring配置方式传参如下：此类隐藏参数在每次调用的时候都会传给Provider。

```
<jsf:consumer id="myHello1" interface="com.jd.jsf.test.HelloService" alias="ZG1" protocol="jsf">
  <jsf:parameter key="user" value="zhanggeng" />          <!-- 接口级参数 -->
  <jsf:parameter key="passwd" value="11112222" hide="true" /> <!-- 接口级参数，标记为hide表示只能在filter取到，
  服务端业务代码取不到 -->
  <jsf:method name="echo">
    <jsf:parameter key="aaa" value="aaaaaaa" hide="true" /> <!-- 方法级参数 -->
    <jsf:parameter key="bbb" value="bbbbbbb" />          <!-- 方法级参数，标记为hide表示只能在filter取到，
    服务端业务代码取不到 -->
  </jsf:method>
</jsf:consumer>
```

2. 调用端代码中动态设置。

```
普通方式

RpcContext.getContext().setAttachment("user", "zhanggeng");
RpcContext.getContext().setAttachment(".passwd", "11112222"); // ".开头的对应上面的hide=true
xxxService.yyy();// 再开始调用远程方法
// 重要:下一次调用要重新设置，之前的属性会被删除
RpcContext.getContext().setAttachment("user", "zhanggeng");
RpcContext.getContext().setAttachment(".passwd", "11112222"); // ".开头的对应上面的hide=true
xxxService.zzz();// 再开始调用远程方法
```

```
session方式（版本1.5.2+）

try {
  // 当前线程设置一次即可
  RpcContext.getContext().setSessionAttribute("user", "zhanggeng");
  RpcContext.getContext().setSessionAttribute(".passwd", "11112222"); // ".开头的对应上面的hide=true
  xxxService.yyy();// 再开始调用远程方法
  xxxService.zzz();// 再开始调用远程方法
} finally {
  RpcContext.getContext().clearSession(); // 重要:需要主动清理
}
```

Provider端：

1. filter中直接获取，包括标记为hidden的参数。通过Rpccontext无法获取。

```
String consumerToken = (String) invocation.getAttachment(".passwd");

2. 服务端业务代码中直接获取

String user = RpcContext.getContext().getAttachment("user");
```


```
session方式（版本1.5.2+）

String user = RpcContext.getContext().getSessionAttribute("user");
```

3. Provider也可以自己在spring配置中传参，也会带到每个请求中（此项设置意义不大）。

```
<jsf:provider id="helloServiceExport" interface="com.jd.testjsf.HelloService" ref="helloService" server="jsf"
alias="JSF_0.0.1">
  <jsf:parameter key="yourname" value="zhangxx" /> <!-- 每次都能在代码拿到 -->
</jsf:provider>
```

其它：

 调用链中的隐式传参

注意：在调用链例如A->B->C，A和B都要隐式传参的时候，由于是同一个线程，会出现数据污染。例如A发参数P1给B，B收到请求拿到P1同时要发参数P2给C，那么C会直接拿到P1,P2。  
这种情况，就要求B收到P1，然后设置P2调用C之前，要求自己清空上下文数据  
(RpcContext.getContext().clearAttachments();)

Token调用

token认证基于隐式传参实现，Provider和Consumer需要指定同一个token才能进行调用。

Provider配置如下：加入隐藏key: token。

```
<jsf:provider id="helloServiceExport" interface="com.jd.testjsf.HelloService" ref="helloService" server="jsf"
alias="JSF_0.0.1">
  <jsf:parameter key="token" value="1qaz2wsx" hide="true" />
</jsf:provider>
```

Consumer配置如下：同样加入隐藏key: token

```
<jsf:consumer id="helloService" interface="com.jd.testjsf.HelloService" alias="JSF_0.0.1" protocol="jsf" >
  <jsf:parameter key="token" value="1qaz2wsx" hide="true" />
</jsf:consumer>
```

双方一致才能完成调用，否则抛出异常。

API方式：（API方式为com.jd.jsf.util.Constants#HIDDEN\_KEY\_TOKEN）就是 .token 前面带点

```
providerConfig.setParameter(com.jd.jsf.gd.util.Constants.HIDDEN_KEY_TOKEN, "123456");
consumerConfig.setParameter(com.jd.jsf.gd.util.Constants.HIDDEN_KEY_TOKEN, "123456");
```

JSF调SAF：JSF调SAF的token服务参见：[JSF与SAF的兼容与比较#JSF调用SAF（dubbo）](#)

指定发布IP

如果服务端存在多网卡，且都能连注册中心的情况，就可能出现绑定的IP不固定的情况，此时需要手动指定发布的IP地址。

```
<jsf:server id="jsf" protocol="jsf" host="10.12.113.111" /> <!--默认连注册中心取，没有注册中心从网卡里取一个 -->
```

指定注册IP

 目前显示虚拟机出现 机器IP 和 机器物理网卡IP 不对应的情况，这是需要指定一个虚拟IP。  
即绑定的地址为 192.168.22.22:22000，而注册到注册中心为 172.17.11.11:22000。

例如虚拟机IP为172.17.11.11（其它机器通过此IP访问），该虚拟机对应的宿主物理机IP为192.168.22.22。可是在虚拟机上并没有IP为17.17.11.11的网卡信息，反而有一个192.168.22.22的网卡信息。  
此时发布jsf服务，是必须绑定到网卡上的，所以绑定的地址为 192.168.22.22:22000（绑定172.17.11.11会出现异常，因为没有网卡）。  
那么如果告诉注册中心此Provider地址为192.168.22.22:22000，其它机器又无法访问，这时候怎么办呢？

可以通过如下配置，指定注册ip，注册ip和绑定ip可以不同。

```
<jsf:server id="jsf" protocol="jsf" host="192.168.22.22" virtualhost="172.17.11.11"/>
```

如果不想每个都设置，可以保存到一个文件，例如 echo "172.17.11.11" > /etc/jsfvirtualhost，然后每次读取这个文件

```
<jsf:server id="jsf" protocol="jsf" virtualhostfile="/etc/jsfvirtualhost"/>
```

指定端口

jsf发布服务端时候可以指定发布的端口。

```
<jsf:server id="jsf" protocol="jsf" port="11090" /> <!--默认22000 -->
```

如果端口被占用，则自动加1进行重试，直到发布成功或无可用端口。

port可以配置为"-1"表示随机端口，随机端口的节点如果一段时间无心跳会被删除。（注：目前为了端口范围可控也已经控制为从22000开始叠加）。

方法限制

jsf支持一个接口下只发布部分方法（不推荐）

 不推荐这样发布，如果不让调用最好从接口里删除该方法

配置接口下的include白名单和exclude黑名单即可，多个方法用逗号隔开。

```
<jsf:provider id="helloServiceExport" interface="com.jd.testjsf.HelloService" ref="helloService" server="jsf"
alias="JSF_0.0.1"
include="*" exclude="get"> <!-- include代码 -->
```

<http://pcloud.jd.com/pages/viewpage.action?pagelId=10671257>



```
</jsf:provider>
```

内置Filter

目前Provider和Consumer内置了一些过滤器，符合条件会自动加载，无需手动开启，顺序和列表如下。

范围	过滤器	中文名	别名	可独立关闭	备注
provider/consumer	List<AbstractFilter>	全部内置过滤器	*	是	强制关闭全部

范围	过滤器	中文名	别名	可独立关闭	备注
provider/consumer	List<AbstractFilter>	全部内置过滤器	default	是	强制关闭全部
provider	com.jd.jsf.gd.filter.SystemTimeCheckFilter	系统时间检查过滤器	systemTimeCheck	是	linux下检查系统时间变化，防止定时器失效
provider	com.jd.jsf.gd.filter.ExceptionFilter	异常过滤器	exception	是(1.5.2+)	关闭后抛出未知异常，调用端可能无法反序列化
provider	com.jd.jsf.gd.filter.ProviderContextFilter	服务端上下文过滤器		否	
provider	com.jd.jsf.gd.filter.ProviderGenericFilter	服务端泛化调用过滤器	providerGeneric	是	关闭后无法支持泛化调用
provider	com.jd.jsf.gd.filter.ProviderHttpGWFilter	服务端网关调用过滤器	providerHttpGW	是	关闭后HTTP网关调用异常返回有问题
provider	com.jd.jsf.gd.filter.TokenFilter	token检查过滤器	token	是	配置了token才加载，关闭后无法支持自带token认证
provider	com.jd.jsf.gd.filter.ProviderMethodCheckFilter	方法调用验证器	providerMethodCheck	是	关闭后无法支持按方法名限制调用
provider	com.jd.jsf.gd.filter.ProviderTimeoutFilter	服务端超时打印过滤器	providerTimeout	是	关闭后无法提示服务端超时信息
provider	com.jd.jsf.gd.filter.ValidationFilter	参数校验过滤器	validation	是	配置了validation="true"才加载
provider	com.jd.jsf.gd.filter.CacheFilter	结果缓存过滤器	cache	是	配置了cacheref才加载
provider	com.jd.jsf.gd.filter.ProviderConcurrentsFilter	服务端并发控制器过滤器	providerConcurrents	是	配置了concurrents>=0才加载
provider	T extends com.jd.jsf.gd.filter.AbstractFilter	自定义过滤器			
provider	T extends com.jd.jsf.gd.filter.AbstractFilter	自定义全局过滤器			
consumer	com.jd.jsf.gd.filter.SystemTimeCheckFilter	系统时间检查过滤器	systemTimeCheck	是	linux下检查系统时间变化，防止定时器失效
consumer	com.jd.jsf.gd.filter.ExceptionFilter	异常过滤器	exception	是(1.5.2+)	关闭后抛出未知异常，调用端可能无法反序列化

		器			法反序列化
consumer	com.jd.jsf.gd.filter.ConsumerGenericFilter	调用端泛化调用过滤器	consumerGeneric	是	配置了generic="true"才加载

范围	过滤器	中文名	别名	可独立关闭	备注
consumer	com.jd.jsf.gd.filter.ConsumerContextFilter	调用端上下文过滤器		否	
consumer	com.jd.jsf.gd.filter.CacheFilter	结果缓存过滤器	cache	是	配置了cacheref才加载
consumer	com.jd.jsf.gd.filter.MockFilter	Mock调用过滤器（服务降级）	mock	是	
consumer	com.jd.jsf.gd.filter.ConsumerInvokeLimitFilter	按APP限制调用次数过滤器	consumerInvokeLimit	是	集成自动部署获取AppId才加载
consumer	com.jd.jsf.gd.filter.ValidationFilter	参数校验过滤器	validation	是	配置了validation="true"才加载
consumer	com.jd.jsf.gd.filter.ConsumerConcurrentsFilter	调用端并发控制器过滤器	consumerConcurrents	是	配置了concurrents>=0才加载
consumer	com.jd.jsf.gd.filter.ConsumerMonitorFilter	调用端监控过滤器	consumerMonitor	否	
consumer	T extends com.jd.jsf.gd.filter.AbstractFilter	自定义过滤器			
consumer	T extends com.jd.jsf.gd.filter.AbstractFilter	自定义全局过滤器			

可以通过如下配置关闭（jsf-1.2.0+）内置过滤器。通过"-xxx"关闭内置的别名为xxx的过滤器。

配置参考：

```
<!-- 例如不开启全部内置过滤器-->
<jsf:provider id="helloServiceExport" interface="com.jd.testjsf.HelloService"
    filter="-default" ref="helloService" server="jsf" alias="ZG110_0.0.1">
</jsf:provider>
<!-- 例如不开启调用端并发过滤器和结果缓存过滤器-->
<jsf:consumer id="helloService2" interface="com.jd.testjsf.HelloService" alias="ZG110_0.0.1"
    filter="-consumerConcurrents,-cache" protocol="jsf" >
</jsf:consumer>
```

自定义Filter

JSF在调用端调用前和服务端都会有一个FilterChain，里面已经内置了一些Filter，运行用户扩展一些自定义Filter。

使用方式如下：

- 1、继承com.jd.jsf.filter.AbstractFilter，实现具体的逻辑。

com.jd.saf.gd.filter.EchoFilter

```
package com.jd.jsf.filter;
import java.util.Map;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import com.jd.jsf.msg.RequestMessage;
import com.jd.jsf.msg.ResponseMessage;
public class EchoFilter extends AbstractFilter {
    private final static Logger LOGGER = LoggerFactory.getLogger(EchoFilter.class);
    @Override
    public ResponseMessage invoke(RequestMessage request) {
        Map<String, Object> configContext = super.getConfigContext(); // 可以拿到一些配置里的信息
        LOGGER.info("before invoke.."); // 在getNext().invoke(request)前加的代码, 将在远程方法调用前执行
        ResponseMessage response = null;
        if ("true".equals(configContext.get("mock"))) {
            response = getNext().invoke(request); // 调用链自动往下层执行
        } else {
            response = MessageBuilder.buildResponse(request); // 也可以自己构造返回对象
            response.setResponse("xxx"); // 返回结果
            // response.setException(); // 返回异常
        }
        LOGGER.info("after invoke.."); // 在getNext().invoke(request)后加的代码, 将在远程方法调用后执行
        return response;
    }
}
```

2. 通过spring配置或者api方式注入到Provider或Consumer中。

⚠ 过滤器不能是单例。因为过滤器有next对象, 如果是单例, 会被覆盖。请通过scope="prototype" 设置

Provider端配置参考

```
<bean id="echoFilter" class="com.jd.jsf.filter.EchoFilter" scope="prototype"/> <!-- 过滤器不能是单例 -->
<jsf:provider id="helloServiceExport" interface="com.jd.testjsf.HelloService" ref="helloService" server="jsf"
alias="JSF_0.0.1"
filter="echoFilter"> <!-- 多个filter用逗号隔开 -->
</jsf:provider>
```

Consumer端配置参考

```
<bean id="echoFilter" class="com.jd.jsf.filter.EchoFilter" scope="prototype"/> <!-- 过滤器不能是单例 -->
<jsf:consumer id="helloService" interface="com.jd.testjsf.HelloService" protocol="jsf" alias="JSF_0.0.1"
filter="echoFilter"> <!-- 多个filter用逗号隔开 -->
</jsf:consumer>
```

**New!** 1.2.0版本开始, 增加一种全局过滤器配置方式

```
<!-- 全部provider和consumer走此逻辑 -->
<jsf:filter id="globalfilter" class="com.jd.testjsf.customfilter.EchoFilter" />

<!-- 指定provider和consumer走此逻辑 -->
<jsf:filter id="globalfilter" class="com.jd.testjsf.customfilter.EchoFilter" providers=""
consumers="helloService2" />

<!-- 全局filter 从1.6.1版本开始支持ref属性指定filter bean 在ref和class属性都指定的情况下已ref指定的bean为准-->
<jsf:filter id="globalfilter" ref="echoFilter"/>
```

**New!** 1.6.0版本增加了通过注解的方式加载自定义filter（适用于通用框架, 不建议普通使用者使用）

jsf 客户端会加载META-INF/jsf/ 和 META-INF/services/ 下的配置文件

具体步骤:

- 自定义filter的实现如上文档所述, 不同的地方在配置这块, 不需要在xml中配置。在自定义filter上加上注解。

```
/**
 * Extensible value: filter的标识 . order 定义filter 加载filter的先后顺序
 *
 * *AutoActive providerSide 为true 则代表provider都会走这个filter, consumerSide 为false代表调用端不会走这个filter
 */
@Extensible(value = "testext", order = 2)
@AutoActive(providerSide = true, consumerSide = false)
public class ExtensionFilter extends AbstractFilter {
    private static final Logger logger = LoggerFactory.getLogger(ExtensionFilter.class);

    @Override
    public ResponseMessage invoke(RequestMessage request) {
        logger.info("extension filter invoked...");
        return getNext().invoke(request);
    }
}
```

- 在工程配置的/META-INF/jsf 或/META-INF/services 目录下配置文件名为com.jd.jsf.gd.filter.Filter内容示例如下

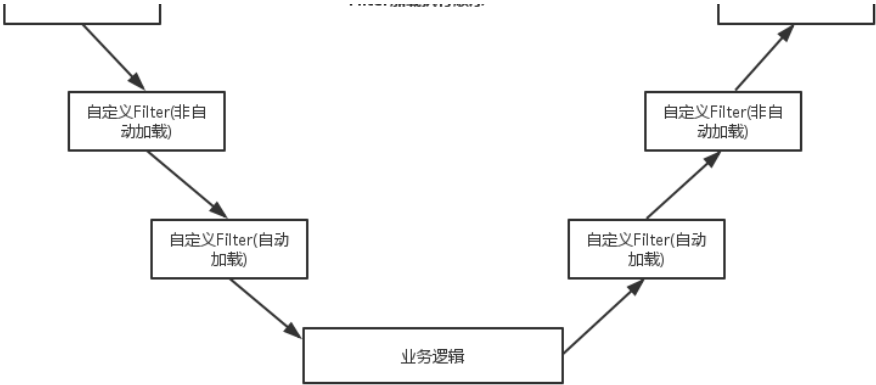
```
com.jd.jsf.gd.config.ext.ExtensionFilter # test auto active on provider side
```

配置这两步后就可以自动加载自定义filter了。

内置Filter

Filter加载执行顺序

内置Filter



连接监听

Provider的Server监听一个端口，当Consumer连接和断开的时候，可以触发注入的连接监听器。

⚠ 此操作会分发到服务端端口对应的业务线程池上执行，建议执行一些快速的操作，不要执行复杂的业务逻辑。

配置如下：

1、写一个实现类，继承com.jd.jsf.msg.ConnectListener接口

```
com.jd.testsaf.listener.TestConnectListener

package com.jd.testjsf.listener;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import com.jd.jsf.msg.ConnectListener;
import io.netty.channel.ChannelHandlerContext;
public class TestConnectListener implements ConnectListener {
    private final static Logger LOGGER = LoggerFactory.getLogger(TestConnectListener.class);
    public void connected(ChannelHandlerContext ctx) { // 建立连接时
        LOGGER.info("conneted.... from {}", ctx.channel().remoteAddress());
    }
    public void disconnected(ChannelHandlerContext ctx) { // 断开连接时
        LOGGER.info("disconneted.... from {}", ctx.channel().remoteAddress());
    }
}
```

2、注入到Provider的配置中

```
<bean id="listener1" class="com.jd.testjsf.listener.TestConnectListener" />
<jsf:server id="jsf" protocol="jsf" onconnect="listener1" /> <!-- 多个用逗号隔开 -->
```

服务端动态分组

管理端操作，Provider启动后可以在管理端进行重新分组。

- 1、服务信息 - 服务管理 - 接口点进去 - Provider列表
- 2、勾选Provider端，选择动态分组按钮，填入新分组。

SAF服务管理平台 首页 关于我们 联系方式

服务信息

服务管理

实例管理

com.jd.testsaf>HelloService

负责人ERP账户:

主机: 请输入主机IP 别名: 版本: 所有版本 查询 刷新 返回

添加服务 上下线 批量删除 清除死亡节点 导入 导出 动态分组 取消分组 黑白名单 刷新注册中心缓存

序号	别名	协议	主机	端口	SAF版本	状态	操作
1	SAFchang:0.0.1	dubbo	10.12.125.26	12345	109	●	查看 服务详情 方法调用
2	SAFchang:0.0.1	dubbo	10.12.107.83	12345	109	●	查看 服务详情 方法调用

追加模式是指：老alias和新的alias都会被调到

替换模式是指：老alias不再被调，新alias被调到

- 3、系统会自动复制选择的Provider（IP端口等除了别名都一样）到新的alias。
- 4、注册中心加载到变化后，会将alias下新的Provider列表推送给Consumer，包含这些动态分组Provider的原始alias和新的alias。
- 5、Consumer调用到动态分组时，会自动处理。

调用端切换分组

管理端操作，下发配置到调用端

1、服务信息 - 服务管理 - 接口点进去 - Consumer列表下面配置下发按钮。



- 2、选择一个Consumer，填写新的alias点下发。
- 3、等待注册中心加载数据后下方给Consumer。
- 4、Consumer得到新配置后，重新生成代理类。此时会调用新的分组。

服务质量监控

目前只收集Provider数据。  
Provider默认开启监控服务端，按每分钟每方法像MonitorServer发送监控数据和异常数据，发送失败数据丢弃。  
可以在管理端（服务信息 - 服务管理 - 属性配置按钮点进去）进行设置是否打开、设置收集机器的白名单。  
也可以在配置中强行关闭此功能。不推荐

```
<jsf:provider id="helloServiceExport" interface="com.jd.testjsf.HelloService" ref="helloService" server="jsf"
alias="JSF_0.0.1">
  <jsf:parameter key="monitor" value="false" hide="true" /> <!-- 强行关闭monitor 不推荐 -->
</jsf:provider>
```

Telnet运维

服务端发布接口后，监听一个端口。默认运行直接telnet到该端口进行操作。  
可以通过配置关闭此功能。注意：关闭后，部分依赖telnet的功能（例如管理端查看服务列表，invoke等）将不能使用。

```
<jsf:server telnet="false" />
```

命令输入telnet 10.12.11.11 22000 回车；  
然后看到"jsf>" 后，支持如下命令执行：

命令	含义	起始版本	备注
help	help 查看支持的命令列表	1.0.0	
	help ls 查看ls的使用说明	1.0.0	
version	查看当前jsf的版本	1.0.0	
ls	ls 查看全部发布的接口信息	1.0.0	
	ls -l 查看接口明细	1.0.0	
	ls com.jd.testjsf.HelloService 查看接口下方法	1.0.0	
	ls -l com.jd.testjsf.HelloService 查看接口下方法明细（含参数）	1.0.0	
ps	ps 查看全部建立的长连接，	1.0.0	
	ps 22000 表示查看22000端口下长连接明细	1.0.0	
	ps -c 22000 表示查看22000端口下长连接数量	1.6.0	
conf	conf -p Show provider configs	1.0.0	
	conf -c Show consumer configs	1.0.0	
	conf -r Show jsf context	1.0.0	
	conf -s Show all interface settings	1.0.0	
	conf -g Show global settings	1.0.0	
	conf -a Show all above	1.0.0	
sudo	后面传入超级管理员密码（存在注册中心中），可以开启高级功能。	1.0.0	
invoke	直接发起调用，传入接口方法参数值进行调用	1.0.0	1.本机127.0.0.1或者没走注册中心可直接访问 2.其它需要超级权限，或者需要密码 如果方法带token
	invoke -p passwd com.jd.testjsf.HelloService.echoStr("111zg222")	1.0.0	
	invoke -t 1qaz2wsx com.jd.testjsf.HelloService.echoStr("111zg222")	1.0.0	
tp	tp 查看回调线程池和系统上下文信息	1.0.0	
	tp 22000 查看22000端口下的线程池信息	1.0.0	
	tp 22000 1000 10 查看22000端口下的线程池信息，每隔1秒，连续10次	1.6.1	
debug	开启debug模式。"debug on"打开，"debug off"关闭	1.5.0	
info	info com.jd.testjsf.HelloService 查看接口描述	1.2.0	
method	method com.jd.testjsf.HelloService.put 查看方法描述	1.2.0	
exit	退出	1.0.0	
jvm	查看jvm的内存和线程信息	1.0.0	
reset	重置部分功能，例如rest sechudle	1.2.0	

在JSF管理端-服务管理-接口点进去-Provider列表- 右边有个详情按钮，里面可以手动输入命令。

服务列表备份

Consumer在注册中心订阅了服务列表后，会将服务列表信息备份到本地文件，防止注册中心全部死掉，也能保证有一份最近的服务列表。

默认备份路径为 \$(user.home)/jsf文件夹下。

可以通过配置进行修改。

```
<jsf:registry id="jsfRegistry" protocol="jsfRegistry" index="i.jsf.jd.com" file="/home/admin/jsfbak" />
```

代理类生成方式

支持JDK和javassist（默认）两种代理类生成方式。

主要作用就是在调用端拦截下业务代码的本地调用，转为调用远程服务端。

可通过配置进行设置。

```
<jsf:provider proxy="jdk" />
<jsf:consumer proxy="javassist" />
```

序列化方式

目前JSF默认使用msgpack进行序列化。msgpack较hessian速度更快，数据包更小，同样支持多语言，而且支持更多数据结构（例如Set）。参见[MsgPack和Hessian序列化的区别](#)

自定义的bean最好实现 java.io.Serializable 接口，最好有空的构造函数。

不想序列化的字段加上 transient 修饰符即可。

注：修饰符为static、transient或者加了@Ignore注解的字段，都将不序列化。

**i** Msgpack序列化保证字段的顺序，不保证字段的名字对应。所以在单边增加字段的时候（例如服务端先升级，调用端先不升级，或者反之）时，新加的字段需要放在字段最后面（不用文件最后面），且不能调整字段的顺序。父类的字段不能变，对方没有的字段值为null

其它还支持 hessian、json 和 java （Provider端根据请求自动判断，Consumer端配置即可）：

**w** jsf协议兼容msgpack和hessian，无需配置  
<jsf:consumer protocol="jsf" serialization="hessian" /><!-- 调用端jsf协议使用hessian序列化 -->  
若发布为dubbo协议，则必须指定使用hessian进行序列化。  
<jsf:consumer protocol="dubbo" serialization="hessian" /><!-- 调用端dubbo协议必须使用hessian序列化 -->

1.6.0开始 增加 protobuf支持(不支持无参数或者无返回值方法,不支持参数和返回值是null)。

线程名称和数量

	线程池	名称	默认	可配置	说明
服务端	Boss线程池(IO)	JSF-SEV-BOSS-	max(4,cpu/2)	否	同一种协议的 Server 共用一个 Boss 线程池
	Worker线程池(IO)	JSF-SEV-WORKER-	max(8,cpu+1)	jsf.server的iothreads，同一种协议以第一个配置为准	同一种协议的 Server 共用一个 Worker 线程，
	业务线程池	JSF-BZ-	20-200	配置参考下面的 <a href="#">线程池类型</a>	一个端口一个业务线程池，如果一个端口下多个服务，那么这些服务共用这一个业务线程池。
调用端	Worker线程池(IO)	JSF-CLI-WORKER-	max(6,cpu+1)	不走注册中心可配置，可在第一个jsf:consumer中配置iothreads属性	全局的调用端共用一个 Woker 线程池
	重连断开Provider节点线程	JSF-CLI-RC-	1	否	一个consumer一个
	给已连接Provider发心跳线程	JSF-CLI-HB-	1	否	一个consumer一个
	回调线程池	JSF-CLI-CB-	20-200	<a href="#">JSF配置参考手册- &lt;jsf:parameter&gt;</a>	全局共享，有callback时才会初始化
注册中心	注册中心心跳+重试线程	JSF-jsfRegistry-HB&Retry	1	否	走注册中心才加载
	注册中心定时检查数据线程	JSF-jsfRegistry-Check	1	否	走注册中心才加载
	注册中心回调处理线程	JSF-jsfRegistry-CB	1	否	走注册中心才加载
其它	发送Monitor数据线程	JSF-MNTR-SEND	1	否	开启监控才初始化
	切分Monitor数据线程	JSF-MNTR-SLICE	1	否	开启监控才初始化
	检查超时Future线程	JSF-Future-Checker	1	否	
	检查超时Callback Future线程	JSF-Future-Checker-CB	1	否	

线程池类型

一个端口（即一个jsf:server）后有一个业务线程池。

目前业务线程池支持固定（fixed）和伸缩的（cached）两种线程池类型。通过配置threadpool线程池类型，通过threads配置最大线程数。

业务线程池同时支持队列，可通过queueype配置线程池队列类型，通过queues参数配置队列大小。

参考配置：

	threadpool 线程池类型	初始 线程 数	threads 最大线 程数	queues 队列大 小	说明	优点	缺点
伸缩有队列线程池	cached	20	100	256	任务来了先丢到队列中，队列满了才会增加线程，直到线程满，得不到执行线程抛异常	节约线程资源，空闲一分钟自动回收，需要时重建； 队列带来一定的并发缓冲功能	队列带来一定的执行延迟
伸缩无队列线程池（默认）	cached	20	200	0	任务来了直接分配线程，直到线程池满，得不到执行线程抛异常	节约线程资源，空闲一分钟自动回收，需要时重建；	并发突然变大无缓冲
固定有队列线程池	fixed	100	100	256	线程数量固定，没有拿到线程丢到队列里，得不到执行线程抛异常	没有线程伸缩带来的性能问题； 队列带来一定的并发缓冲功能	队列带来一定的执行延迟
固定无队列线程池	fixed	200	200	0	线程数量固定，得不到执行线程抛异常	没有线程伸缩带来的性能问题	并发突然变大无缓冲

默认为伸缩线程池**cached**，1分钟回收，线程数最小**20**，最大**200**，队列为**0**的普通队列。

```
<jsf:server id="jsf" protocol="jsf" threads="200" threadpool="cached" queueype="normal" queues="0" />
```

参考的固定线程池配置

```
<jsf:server id="jsf" protocol="jsf" threads="400" threadpool="fixed" />
```

当Provider的业务线程池满了，无可在线程池的时候，会返回一个异常给Consumer，告知Consumer该Provider线程池已耗尽。

1. 此时需要使用jstack等排除线程情况，找到真正的性能瓶颈。
2. 如果的确是压力太大撑不住，就进行Provider的水平扩容。

关于线程池的配置，其实没用一个统一或者可推荐的配置可以套用；  
如果是普通的接口，默认的就够用；  
如果是高并发或者特性需求的接口，请以压测结果为准。

事件分发类型

服务端有 boss线程，IO线程（work线程），业务线程；调用端有IO线程和业务线程

正常的调用包的header解析在IO线程，body的序列化和反序列化在业务线程执行。

心跳包直接在IO线程返回。

ResponseListener和Callback事件在callback线程池上执行。

Mock调用（1.0.x版本）

支持方法级的Mock调用。当启动Mock调用的时候：

对于Consumer来说，将不走远程调用，而是走本地模拟实现。可用于开发和测试环境下，不依赖服务端的调用。或者线上环境的服务降级操作。

对于Provider来说，将不走具体业务实现，而是直接走本地模拟实现，可用于服务降级，或者异常的屏蔽（例如数据库异常，先返回固定数据）。

使用方式如下：

1. 不管是Provider还是Consumer，使用Mock调用的时候，需要集成调用的接口实现一个模拟实现类。这个类和业务实现类的分开。

```
public class HelloServiceMock implements HelloService {
    public String get(Integer id) throws MyException {
        return "get from mock";
    }
    public boolean put(int id, String str) {
        return true;
    }
    public String echoStr(String str) {
        return "echo from mock";
    }
}
```

2. 配置的时候将此实现类注入到ProviderConfig或者ConsumerConfig的"mockRef"属性中。。
3. 配置接口级或者方法级的"mock"属性。

Provider端配置

```
<!-- 业务实现 -->
<bean id="helloServiceImpl" class="com.jd.testjsf.HelloServiceImpl"/>
<!-- 模拟实现 -->
<bean id="helloServiceMock" class="com.jd.testjsf.mock.HelloServiceMock"/>

<jsf:provider ref="helloServiceImpl" mockref="helloServiceMock" mock="true"> <!-- 默认mock="false",配置为true代表接口下方法mock为true -->
  <jsf:method name="echoStr" mock="false"/> <!-- 除了此方法其它方法走mock -->
</jsf:provider>
```

Consumer配置

```
<!-- 模拟实现 -->
<bean id="helloServiceMock" class="com.jd.testjsf.mock.HelloServiceMock"/>

<jsf:consumer mockref="helloServiceMock" mock="false"> <!-- 默认mock="false" -->
  <jsf:method name="echoStr" mock="true"/> <!-- 除了此方法其它方法不走mock -->
</jsf:consumer>
```

Mock调用（1.2.x版本）

从1.2.0版本开始，mock调用改为在管理端动态配置，使用的时候无需在本地生成实现类。  
在管理端的服务管理-接口列表-接口点进去-下面有个mock设置



然后选择方法，开启，设置下mock返回值（这个值是符合json格式的，字符串用双引号，list/数组用[]，map/对象用{}等）

mock设置

方法: echoStr 开关: 开启

主机: echoStr 别名:

MOCK值: "result for mock"

取消 保存

等到注册中心将这个配置发给客户端，客户端就会不发起远程调用，直接走配置的返回值。

服务降级（1.0.x版本）

- 依赖JSF调用端的Mock调用功能。
- 从管理端下发配置
- 1、首先得在Provider和Consumer中加入mock功能（mockref不能为空），同时设置mock为false先不启用。参见[Mock调用](#)
  - 2、再JSF管理端的 服务信息-服务管理-接口点进去 的页面。

可以勾选服务列表里的Provider，点击配置下发按钮，弹出mock选择是或者否。Provider收到配置下发后，会自动开启mock调用功能，从而实现服务端降级。



同样勾选服务列表里的Consumer，点击配置下发按钮，弹出mock选择是或者否。Consumer收到配置下发后，会自动开启mock调用功能，从而实现服务端降级。



数据包大小限制

默认数据包大小为8M，即服务的请求和返回值序列化后默认不超过8M，要不然数据接收方会丢弃此数据。  
数据较大，建议开启[调用压缩](#)，如果压缩后数据还大，可以自行定义数据包配置。



如果请求值数据较大，则服务提供者进行配置。

```
<jsf:server payload="16777216" /> <!-- 默认是8388608=8*1024*1024 -->
```

如果是返回值数据较大，则服务调用者进行配置。

注意：由于JSF是长连接复用的，请把特殊配置的consumer放到其它consumer之前加载，防止配置覆盖。

```
<jsf:consumer payload="16777216" /> <!-- 默认是8388608=8*1024*1024 -->
```

调用压缩

在Consumer发送请求和Provider返回响应的时候，都可以开启调用压缩。

目前支持多种算法：snappy, lzma,

1、如果Consumer端配置了压缩，且请求的数据大于2048B，那么请求数据将被压缩后再发给Provider。

```
<jsf:consumer compress="snappy" />
```

服务端针对压缩过的请求，返回响应的时候也会判断是否启动压缩。（但是：虽然客户端配了压缩，如果请求小于2048B，还是需要服务端配置压缩才触发）

2、如果Provider端配置了压缩，那么不管请求时是不是带压缩标识，返回响应的时候也都会检查数据大小，如果数据大小超过阈值，则将响应数据压缩后再发给Consumer。

```
<jsf:provider compress="snappy" />
```

最佳实践：请求数据大的客户端配置，响应数据大的服务端配置。

并发控制

Provider和Consumer都支持方法级别的并发控制（Mock调用不受控制）。

通过concurrents属性进行配置，方法级的配置可覆盖接口级的配置，接口级没配置则走默认。

参见：[JSF配置参考手册](#)，配置-1关闭并发过滤器，等于0表示开启过滤但是不限制

**Provider端：**

主要控制的是业务线程池的使用数量。

例如：一个Server业务线程池默认最大200，下面有2个接口，每个接口有5个方法，那么这10个方法的请求是共享这200个业务线程池的。

为了防止一个方法（例如出异常的时候）占用了全部线程池，则要求每个方法最多使用50个线程（不一定要均分，不是200/10=20这样子，也可以超过20）。

Provider端如果来请求时，超过并发大小，调用就会立即抛出RPC异常给客户端：

配置如下：

```
<jsf:server threads="200" /> <!-- 默认这个server最大200个线程 -->
<jsf:provider concurrents="50"> <!-- 接口下每个方法控制在50并发，即最大50个业务线程同时执行-->
  <jsf:method name="getHashSet" concurrents="30" /> <!-- 这个方法比较特别，控制在30并发，最大30个业务线程同时执行-->
</jsf:provider>
```

**Consumer端：**

主要控制调用者自己的业务线程同时发起的请求数。

Consumer如果来请求时，超过并发大小，则会等待执行直到超时为止：

```
<jsf:consumer concurrents="50"> <!-- 接口下每个方法控制在50并发，即最大50个业务线程在调用-->
  <jsf:method name="getHashSet" concurrents="30" /> <!-- 这个方法比较特别，控制在30并发，最大30个业务线程在调用-->
</jsf:consumer>
```

灰度部署

服务发布后，默认立即上线，调用端可以马上发起调用。

如果需要做到灰度上线，即上线后不立即提供服务，等验证后再在管理端手动上线，可以通过如下配置实现。

```
<jsf:provider dynamic="false" /><!-- 指定接口灰度部署 -->
```

结果缓存

在Consumer的cache在远程调用之前，或者Provider的cache逻辑在调用业务代码之前支持配置cache，用于调用结果的缓存。如果从缓存中拿到调用结果，则不往后执行。

此配置支持方法级的cache。

⚠ 请自行设置缓存过期策略，保证数据的准确性和实时性。  
请自行设置缓存清理策略，保证内存不会溢出。

Cache使用参考：

- 1、实现一个Cache实现类，实现接口com.jd.jsf.gd.filter.cache.Cache。主要实现buildKey， put， get三个方法。

com.jd.jsf.gd.filter.cache.Cache

```
public class LruCacheImpl implements Cache {

    /**
     * slf4j Logger for this class
     */
    private final static Logger LOGGER = LoggerFactory.getLogger(LruCacheImpl.class);

    private final Map<Object, Object> cache;

    public LruCacheImpl() {
        //最近访问的最前面 固定大小1024, 保证内存使用量
        this.cache = Collections.synchronizedMap(new LRUHashMap(1024));
    }

    /**
     * 可以加其它方法自行实现 销毁缓存 等操作
     * @param key
     */
    public void invalidateCache(Object key) {
        cache.remove(key);
    }

    /**
     * 通过调用参数获得唯一的key, 返回null以后将不从cache中load
     *
     * @param interfaceId
     *      接口名
     * @param methodName
     *      方法名
     * @param args
     *      方法参数
     * @return 关键字, 可以返回null
     */
    @Override
    public Object buildKey(String interfaceId, String methodName, Object[] args) {
        String key;
        if ("put".equals(methodName)) {
            key = String.valueOf(args[0]); // put方法取参数1作为值
        } else if ("echoStr".equals(methodName)) {
            key = RpcContext.getContext().getRemoteHostName(); // echoStr方法用ip做缓存
        } else {
            key = null; // 返回null以后将不从cache中load
        }
        return key;
    }

    /**
     * 放入缓存
     *
     * @param key
     *      方法参数得到的关键字
     * @param result
     *      缓存的调用结果
     */
    @Override
    public void put(Object key, Object result) {
        cache.put(key, result);
        LOGGER.info("put into cache {} : {}", key, result);
    }

    /**
     * 拿出缓存
     *
     * @param key
     *      方法参数得到的关键字
     * @return 缓存的调用结果
     */
    @Override
    public Object get(Object key) {
        Object value = cache.get(key);
        LOGGER.info("load from cache {} : {}", key, value);
        return value;
    }
}
```

2、注入到Provider或者Consumer配置里

```
<bean id="cacheImpl" class="com.jd.testjsf.cache.LruCacheImpl" />
<jsf:provider>
  <jsf:method name="put" cacheref="cacheImpl" cache="true" /> <!-- 只有put方法走缓存 -->
</jsf:provider>

<jsf:consumer cacheref="cacheImpl" cache="true"> <!-- 全部方法走缓存 -->
</jsf:consumer>
```

3、在调用的时候，会先根据请求参数调用cache.buildKey方法；  
如果返回的key不为null，通过这个key去寻址调用结果；  
如果找到结果则返回，找不到则往后执行。

黑白名单

如果通过注册中心调用，就可以使用调用的黑白名单功能，粒度为接口+IP级。

在管理端 服务信息-服务管理-接口点进去。可以看到黑白名单的设置界面。



设置黑白名单后，注册中心会加载到最新的黑白名单设置。  
注册中心将服务列表+黑白名单计算后，推送给调用者。如果Consumer在黑名单中，将调用不到任何Provider。  
同时黑白名单也会通过注册中心下发到服务端，服务端在接到请求的时候也会进行校验。

路由配置

JSF的路由功能目前有IP路由和参数路由。  
IP路由：主要在管理端和注册中心完成，不需要将规则下发到调用端。  
在管理端配置路由功能后，注册中心读取到路由，计算出新的服务端列表，将新服务端列表下发到调用端。  
Consumer从注册中心拿到的就是经过运算后的注册中心列表。

```
{ "router.item.key": "172.17.39.117", "router.item.value": "172.17.39.116" }
```

参数路由：在管理端进行配置，注册中心加载配置，将规则下发给调用端。  
Consumer在选择Provider进行调用的时候，会实时根据路由规则进行计算。  
目前可以根据 方法名 和 参数值 (toString()) 进行路由，支持的运算有>,>=,<,<=,==,! =。

```
{ "router.item.key": "handleSimple.arg1==1", "router.item.value": "172.17.39.115" }
```




具体参见 exmample:<http://source.jd.com/app/jsf-example/tree/master/src/main/java/com/jd/testjsf/router>

功能成熟度

功能	子功能	成熟度	优点	缺点	建议	特定用户
配置方式	spring+xml	大量应用	无代码入侵，可扩展性强。		可用于生产环境	
	api	大量应用	适用于动态创建发布和调用的系统，例如网关系统。	代码入侵	可用于生产环境	
	spring+annotation	已测试	开发较方便	代码入侵，可扩展性差	可用于开发、测试环境， 不推荐线上环境	
集群策略	failover	大量	多用于读服务，失败重试，增加用户体验，配置灵活	失败重试会增加调用耗时	可用于生产环境	

		未应用	广播调用		广播调用	
	failfast	大量应用	多用于非幂等的写服务，失败立即报错，只一次调用	调用失败率增加	可用于生产环境	
	broadcast	已测试	广播调用可以一次调用全部Provider，多用于无返回值的通知场景	耗时较长，不保证全部调用成功	特定场景使用	JSF管理端
	pinpoint	已测试	指定调用	只支持直连，无法做到负载	特定场景使用	JSF管理端
负载均衡	random	大量应用	根据权重随机。请求量越大分布越均匀，可以灵活调整权重决定结果		可用于生产环境	
	roundrobin	大量应用	均匀请求	无法调整权重，如果某些机器慢，可能会积压请求	可用于生产环境	
	leastactive	少量应用	快速的Provider收到更多请求	抛异常时也认为是快速，需要慢慢恢复	可用于生产环境	
服务依赖检查		大量应用			可用于生产环境	
直连调用		大量应用	开发测试时定向调用，防止乱调。	无法动态的更新服务列表	可用于开发、测试环境， 不推荐线上环境	
异步调用		少量应用	无需多线程即可并行发起调用，减少调用耗时		可用于生产环境	
异步回调		已测试			可用于生产环境	
泛化调用		少量应用	网关场景方便调用，无需服务端接口	需要自己维护接口定义等信息	可用于生产环境	JOS
功能	子功能	成熟度	优点	缺点	建议	
injvm调用		已测试	无需发起rpc调用走本地。	服务端和调用端一起部署，这种设计就有问题	可用于开发、测试环境， 不推荐线上环境	
callback调用		已测试	服务端主动回调到调用端	长连接端口后要重新注册callback	可用于生产环境	JSF注册中心
多协议支持	jsf	大量应用	基于TCP的自定义协议、速度快、兼容dubbo协议	无法跨语言	可用于生产环境	
	dubbo	大量应用	基于TCP的自定义协议、速度快	无法跨语言	可用于生产环境	
	rest	少量应用	跨语言，基于HTTP	框架耗时比tcp慢2/3，未算业务时间	可用于生产环境	
	webservice	少量应用	跨语言，基于HTTP		可用于生产环境	
延迟启动		少量应用	给予数据准备时间		可用于生产环境	
延迟连接		少量应用	减少非必要的长连接		可用于生产环境	
粘滞连接		已测试	减少负载均衡算法时间		可用于生产环境	
参数校验		已测试	调用端增加校验减少错误数据给服务端带来的压力	增加耗时	可用于生产环境	
请求上下		少	方便的获取上下文信息		可用于生产	

文		量应用			环境	
隐式传参		少量应用	方便的传入隐藏参数		可用于生产环境	
Token调用		少量应用	简易有效的token认证	改动oken双方都得同时改	可用于生产环境	
指定发布IP		少量应用	解决了多网卡情况		可用于生产环境	
指定注册IP		少量应用	解决了虚拟机无网卡情况		可用于生产环境	
功能	子功能	成熟度	优点	缺点	建议	
指定端口		大量应用	解决了端口冲突情况		可用于生产环境	
方法限制 <small>NEW</small>		少量应用	方便的限制方法是否可以被调用	这种设计也有问题，不能调的方法应该从接口中删除	可用于开发、测试环境， 不推荐线上环境	
自定义filter		少量应用	自定义扩展一些拦截功能		可用于生产环境	
连接监听 <small>NEW</small>		已测试	监听连接数的变化	IO线程执行	可用于生产环境	JSF注册中心
服务动态分组 <small>NEW</small>		已测试			可用于生产环境	
调用分组映射 <small>NEW</small>		已测试			可用于生产环境	
服务质量监控		少量应用	可以定时收集数据到hbase，生成统计报表或者预警。	统计需要耗时，约占框架耗时的1%~2%	可用于生产环境	
Telnet运维		少量应用	方便的运维调试功能	有安全风险	可用于生产环境	JSF管理端
服务列表备份		大量应用	容灾的一部分		可用于生产环境	
代理类	javasssit	大量应用	比原生更加强大、更加快速的代理类		可用于生产环境	
	jdk	已测试	原生的代理类		可用于生产环境	
序列化	msgpack <small>NEW</small>	大量应用	跨语言，比hessian更快速	增加字段需要配置。	可用于生产环境	
	hessian	大量应用	目跨语言，兼容性强，支持增减字段	做了dubbo兼容；不支持set；速度较慢。	可用于生产环境	
	java	已测试		做了dubbo兼容；速度慢；不跨语言。	可用于开发、测试环境， 不推荐线上环境	
线程池类型	fixed	大量应用	固定线程池，初始化后不会照成性能问题		可用于生产环境	
	cached	大量应用	有最小和最大值，空闲一分钟自动删除线程，需要再重建		可用于生产环境	
功能	子功能	成熟	优点	缺点	建议	

		度				
事件分发 类型		稳定			可用于生产 环境	
mock调用 		已测试	用于开发测试使用，或者线上服务降级		可用于生产 环境	
服务降级		已测试	用户紧急情况处理	需要提前准备mockref类	可用于生产 环境	
调用压缩 		已测试	压缩后数据较小		可用于生产 环境	
并发控制		已测试	按方法级的并发控制，防止瞬间的大流量		可用于生产 环境	
灰度部署		少量应用	验证后再手动上线		可用于生产 环境	
结果缓存		已测试	减少发起的远程调用次数	需要自行实现cache失效和清理策略	可用于生产 环境	
黑白名单 		已测试	拦截恶意调用		可用于生产 环境	
路由配置						

无