2014

RSF 用户手册

远程服务调用框架

第 20 版 适用于 RSF 2.1.1 版本

赵磊
慧聪网-技术中心-架构研发部
2014/07/28

1. 发行版本记录

1.1. RSF 2.1.1 版本发布

2014-7-28 发布了 RSF 2.1.1 版本。

新功能：

Telnet 命令在独立的线程中执行，在服务端线程池满时也可以执行命令

Telnet 新加看 cpu 负载、内存、网络连接数的命令。见 22.5.Telnet 监视工具

主要修复了 BUG：

修改了 DefaultFuture 类潜在的内存溢出的 Bug

新的调整：

调整 Mina ReceiveBufferSize、SendBufferSize 的大小，适应传输图片等较大数据

1.2. RSF 1.3.5 版本发布

2014-5-13 发布了 RSF 1.3.5 版本。

主要修复了 BUG：

修改了 DefaultFuture 类潜在的内存溢出的 Bug

新的调整：

调整 Mina ReceiveBufferSize、SendBufferSize 的大小，适应传输图片等较大数据

1.3. RSF 2.1.0 版本发布

2014-3-13 发布了 RSF 2.1.0 版本。

主要是为了支持灰度发布 rsf 所做的调整。

新功能：

1. 实现了分组功能，达到组内通讯，组件隔离的目的；
2. 实现了采集客户端调用服务的情况，定时定量的向监控中心报告的功能；见：5.4.RSF 监控中心
3. <rsf:service>、<rsf:client>添加 groupName 属性，用于实现同一个接口不同实现的支持；
4. 实现了通过权重来人工干预选择服务提供者节点的功能。

新的调整：

简化了 rsf version 的获取方式。

1.4. RSF 1.3.4 版本发布

2014-3-13 发布了 RSF 1.3.4 版本。
新的调整：

尝试解决 rsf 频繁调用时，线程全部挂起的问题。

1.5. RSF 1.3.3 版本发布

2014-3-11 发布了 RSF 1.3.3 版本。
新的调整：

简化了 rsf version 的获取方式。

1.6. RSF 2.0.0 版本发布

2013-12-20 发布了 RSF 2.0.0 版本。
新功能：

1. 通过使用 Thrift、zookeeper 实现跨语言的服务调用，理论支持 Thrift、zookeeper 支持的开发语言。
2. 开发了 Rsf2.0(C#)、Rsf2.0(PHP)、Rsf2.0(Java)，简化 Thrift、zookeeper 的使用。
3. Telnet 命令，支持查看 thrift 的服务、线程池信息。
4. <rsf:include/>标签，用于分层管理 RSF 配置文件。

1.7. RSF 1.3.2 版本发布

2013-12-20 发布了 RSF 1.3.2 版本。
主要修复了 BUG：

1. 客户端向服务端发送的对象未实现 Serializable 接口时，RSF 给出友好提示。
2. 在高并发时由于 Mina 发出了不真实连接已断开的消息 ,导致 RSF 关闭了连接 ,添加连接检测 ,识别误报。
3. 在高并发时客户端的第一次通信可能创建了重复的连接，已修复。

主要调整：

1. 定时任务 RecoveryHandler 做接口级回声测试失败时，日志不再输出异常，改为输出“警告”。
2. RSF Server 绑定端口失败,端口 xxxx 被占用时，日志不再输出 Error, 改为输出“警告”
3. 客户端等待服务端响应,await()被意外打断时，日志不输出堆栈，再次进入等待。

1.8. RSF 1.3.1 版本发布

2013-09-23 发布了 RSF 1.3.1 版本。

主要修复了 BUG：

1、解决了特定条件下客户端无法找到服务提供者的问题。

2、解决了服务端使用“java 编码”方式配置时，无法定时注册问题。

1.9. RSF 1.3.0 版本发布

2013-08-06 发布了 RSF 1.3.0 版本。

新功能：

1、支持加密通信。

2、支持从 spring 容器获取 bean 做为服务端接口的实现类。

3、支持 Telnet 命令，可以查看 RSF 运行情况。

4、支持异步调用。

5、新的工具类、更规范的 API。

主要修改有：

1、调优了线程池，并可以通过 xml 配置线程池参数。

2、增强了回声测试，用于判断接口的方法是否存在。

3、调整了客户端 channel 的维护机制，解决无法找到服务提供者的问题。

主要修复了 BUG：

1、改进了 Telnet 被心跳断开的问题。

1.10. RSF 1.2.4 版本发布

2013-04-11 发布了 RSF 1.2.4 版本.

主要修改有：

完善了通过 java 编码方式开发服务端与客户端。

1.11. RSF 1.2.3 版本发布

2013-04-08 发布了 RSF 1.2.3 版本。

主要修改有：

RsfListener 类添加了 setConfigLoader()方法。

1.12. RSF 1.2.2 版本发布

2013-04-01 发布了 RSF 1.2.2 版本。

主要修复了 BUG：

1、解决了在同一台 Windows 操作系统上，启动多个 rsf 服务端会绑定到相同端口的问题（Linux 不存在此问

题。

2、解决了在 web.xml 中配置 RsfListener 启动器时，用逗号隔开的多个文件路径不能回车换行的问题。

3、RSF 启动时如果连接注册中心失败则会抛出异常，导致容器的后续启动失败。RsfListener 启动器 catch 了异常，保证容器可继续启动。RSF 会定时尝试连接注册中心，向注册中心再次发布服务。

1.13. RSF 1.2.1 版本发布

2013-02-22 发布了 RSF 1.2.1 版本。

主要修复了 BUG：

1、在 Linux 操作系统中，重启 RSF 服务端时需要间隔 30 秒用来等释放端口,否则服务端端口(63638)被占用无法启动。

1.14. RSF 1.2.0 版本发布

2012-10-30 发布了 RSF 1.2.0 版本。

新功能：

1、支持服务端动态端口，解决了在一台物理服务服务器同时运行多个应用时端口冲突的问题。

主要修改：

1、调整了运行时输出的日志级别与日志内容。

2、调整了线程名称，使用 JVM 工具查看线程时，线程名称相比 1.1.0 更规范。

1.15. RSF 1.1.0 版本发布

2012-07-19 发布了 RSF 1.1.0 版本。

新功能：

1、支持 XML 配置方式，并成为主要配置方式。

2、支持与“服务注册中心”的三点通信。

3、服务端可向“服务注册中心”发布服务。

4、客户端可通过“服务名”发现服务。

5、支持软负载（可配置多种型式）。

6、支持失败转移。

7、支持 Mock。

8、确定了线程模型，并可以配置两种线程池和线程数量。

9、支持超时时间设置

10、整理了运行时输出的日志级别与日志内容。

11、完成了广泛的性能测试，稳定性测试。

12、提供了 com.hc360.rsf.config.RsfListener 监听器用于在 web 项目中启动 RSF。

13、支持回声测试。
14、支持检测 RSF 版本号，可以在服务注册中心查看版本号。
15、支持检测重复加载 RSF jar 包，并在日志中输出提示信息。
16、编写发布了《RSF 用户手册》，方便开发者使用 RSF。
17、修复了若干 bug。

1.16. RSF 1.0.0 版本发布

2012-06-15 发布了 RSF 1.0.0 版本。
新功能：
1、确定了“rsf 通信协议”，为以后跨版本、跨平台通信奠定基础。
2、提供 Java 编码方式 API，可完成点对点通信(方法调用)。
3、使用基于 Java NIO 的非阻塞 IO，长连接。
4、支持心跳功能，心跳失败的通知。
5、支持两种序列化方式。
6、支持基于长连接的数据推送、回调（限于 Java 编码方式 API，试用版）。

2. 目录

3. RSF 快速入门

3.1. 快速运行一个 DEMO

如果你是第一次接触 RSF，阅读《RSF 用户手册》是必不可少的。本手册内容较长，还是要花些时间的，还好内容都不复杂。如果 你不打算全面掌握 RSF 的知识 ,而只是想快速的运行起来一个 RSF 的 demo 体验一下的话，只需要阅读核心的章节就好了。从阅读到复制代码并运行，大约花费你 20 分钟。

建议阅读核心章节	内容
RSF 体系结构	了解决客户端、服务端、注册中心 3 个角色
RSF 基本情况说明	了解要使用哪些 jar 包
RSF 同步调用示例	按步操作就可以运行一个 demo

3.2. 从 6 个维度认识 RSF

维度	支持	说明	使用频度
IO 模型	同步调用		最常使用
	异步调用	有返回值 无返回值	
配置方式	XML 配置		最常使用
	Java 编码配置	一般用于临时情况	
安全	非加密通信		最常使用
	加密通信	依赖 HCC、HAS	
服务发现	3 点通信（客户端、服务端、注册中心）	通过注册中心发现服务	最常使用
	2 点通信（客户端、服务端）	无发现能力 ,要求事先已知服务的地址	
跨语言	只运行于 Java 语言平台（rsf 协议）		最常使用
	跨语言（Thrift 协议）		
注册中心	db 注册中心	无跨语言要求时	最常使用
	zookeeper 注册中心	有跨语言要求时	

注：点击文本中的超连接，可以阅读相应的知识点。

4. RSF 概述

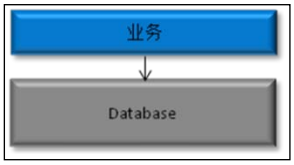
4.1. 服务化架构概述

大型互联网系统一般采用分布式服务化架构。各个系统之间的交互越来越多，把核心业务抽取出来作为独立的服务，逐渐形成稳定的领域服务层，对上层应用提供核心业务接口调用服务。上层应用通过组合调用服务，就可以上线新的应用，能快速的响应多变的市场需求。RSF（远程服务调用框架）是实现分布式服务化架构的基石。

4.2. 服务化架构的演进史

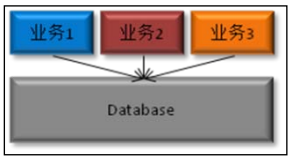
- 单一应用架构

当网站流量很小时，只需一个应用，将所有功能都部署在一起，可以支撑起小规模的应用。后期压力逐渐加大，也可采用负载均衡。



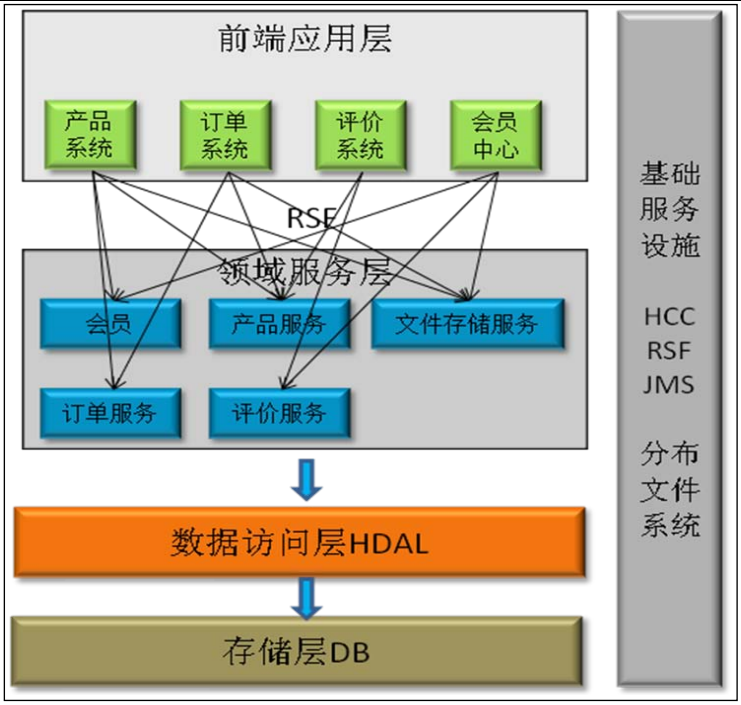
- 垂直应用架构

当访问量逐渐增大，单一应用依靠增加机器带来的加速越来越小，将应用垂直拆分为互不相干的几个应用，以提升效率。应用间无交互。



- 分布式服务化架构

当垂直应用越来越多，应用之间的交互不可避免，将核心业务抽取出来，作为独立的服务，形成领域服务层，供上层应用调用。



4.3. 远程方法调用

RSF（远程服务调用框架）核心功能是远程方法调用。RSF 是一个 jar 包，被各个系统使用，实现各个系统之间的通信。RSF 是实现系统与系统间通信（远程方法调用）的框架。并可通过 HRC 管理全局的服务。

RSF 采用 Java 语言开发，适用于 Java 语言开发的项目，在 2.0 版本支持跨语言通信（基于 Thrift）。

RSF 的主要特点是：在高并发大访问量环境下性能优秀；发布服务、调用远程服务简单、容易学习和使用；可实现服务的综合治理。

暴露服务、调用服务：基于 TCP 协议，使用 Java NIO(非阻塞 IO)实现高性能的网络通信。客户端与服务端依赖同一个服务接口(interface)，服务端实现服务接口(interface)并暴露服务供客户端调用，客户端生成服务接口(interface)的远程代理对象，客户端使用代理对象就像调用本地方法一样调用远程方法。

软负载均衡：当服务端应用部署了多个节点，客户端调用时可自动实现软负载均衡。服务端应用部署节点可以热增加或热减少，客户端会选择可用的服务端节点发起调用。

RSF 是解决系统(服务端)与系统(服务端)之间通信问题的框架，不是解决浏览器与系统(服务端)之间通信的框架，适合工作在同机房的局域网内。

4.4. RSF 通信协议

RSF 架构使用 rsf 通信协议进行通信，rsf 通信协议是基于 TCP 协议的应用层协议。

1字节	1字节	1字节	1字节	8个字节	4个字节
0-7位	8-15位	16-23位	24-31位	32-95位	96-127位
协议头标识：高位	协议头标识：低位	16：request/response标志 17：双向通信标志 18：心跳事件标志 19：握手标志 20：加密标志 21-23：序列化方式标志	24：保留 25：保留 26：保留 27-31：错误状态标志	long型的请求ID	int型的数据长度

第 0-7 位：11011010　　固定值
第 8-15 位：　10111011　　固定值
第 16 位：10000000　　request/response 标志
第 17 位：01000000　　双向通信标志 1：双向　　0：单向
第 18 位：00100000　　心跳事件标志 1：心跳　　0：请求
第 19 位：00010000　握手标志　1：握手 0：非握手
第 20 位：　00001000　加密标志 1：加密 0：非加密
第 20-23 位：00000111　8 种序列化实现方法（已使用了 1，2，4）
第 24 位：　10000000　　保留
第 25 位：　01000000　　保留
第 26 位：　00100000　保留
第 27-31 位：　00011111 状态标志 (成功、超时、服务未找到、服务端异常，客户端异常) 32 种
第 32-95 位：8 字节　　long 型的请求 ID
第 96-127 位：4 字节　　int 型的数据长度
注：本协议头由 com.hc360.rsf.rpc.protocol.codec.ExchangeCodec 类实现，具体实现请看源码。

rsf 通信协议保证了：RSF 可以**跨版本通信**，不同版本 RSF 之间可以正常通信。

rsf 通信协议奠定了：RSF 可以**跨语言通信**的基础，按照 rsf 协议规范，并使用某种中间描述语言实现序列化，可以实现跨平台的通信。

注：由于其它开发语言实现一套 rsf 协议难度较大，RSF2.0(java、php、c#)未使用 rsf 通信协议进行跨语言。而是基于现成的 Thrift、zookeeper 实现的，降低了开发难度和开发周期。何时能使用 rsf 协议实现跨语言还未知。

5. 对服务化架构的支撑

5.1. 服务治理

说到远程方法调用功能，已经有很多技术或框架都可以实现，但它们都缺少服务治理功能，无法担当起**服务化**的大任。HRC(服务注册中心)是 RSF 重要组成部分。

服务治理的应用场景：

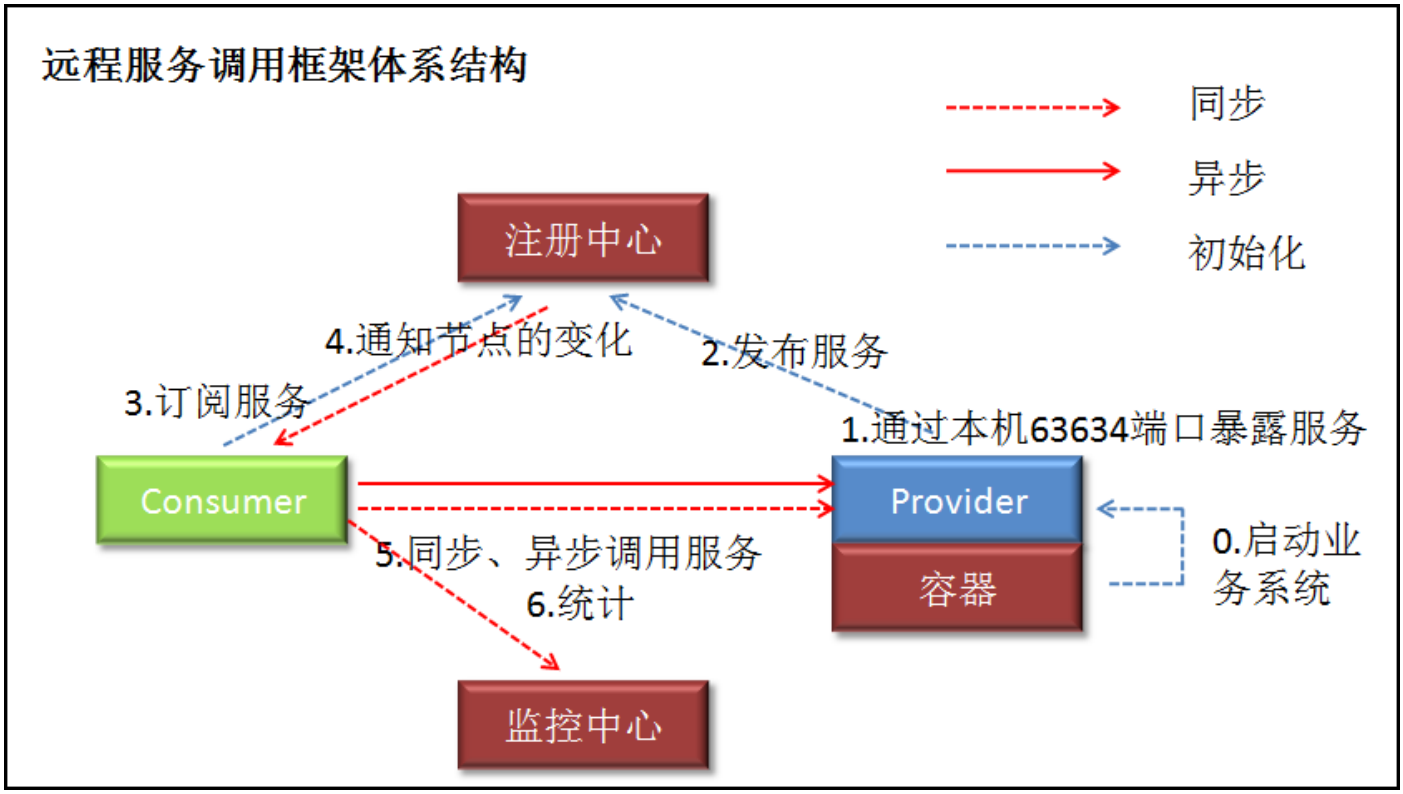
(1) 服务列表：随着业务的发展，对外暴露的服务越来越多，谁能说清楚哪个系统提供了哪些服务，每个服务在调用时接受什么样的参数，返回什么样的结果？通过服务注册中心可以查看服务列表，其中包括的信息有：服务接口总体功能描述（接受的参数与返回结果）、服务发布人、服务发布人部门、系统标识等。未来还将增加 服务所在层、访问口令、权重等。

(2) 服务分层管理：规模继续扩大，应用开始分层，比如核心数据层，业务集成层等。不允许从低层向高层依赖。

(3) 调用统计：统计每天的服务调用量、响应时间、失败次数，作为容量规划的参考指标。可以动态调整权重，在线上，将某台机器的权重一直加大，并在加大的过程中记录响应时间的变化，直到响应时间到达阀值，记录此时的访问量，再以此访问量乘以机器数反推总容量。

(4) 调用链跟踪：系统分布式后，断点调试变的困难，通过埋点日志记录调用流程，可便于跟踪调试，可画出系统间依赖关系图，可找出循环调用。

5.2. 体系结构



角色说明：

- **Provider:** 暴露服务的服务提供者(服务端)。
- **Consumer:** 调用远程服务的服务消费方（客户端）。
- **注册中心:** 发布服务、发现注册、管理服务。
- **容器:** 业务系统运行的容器。

调用关系说明：

- 0. 业务系统启动，加载、运行 RSF 配置文件。
- 1. 服务提供者，通过本机 63634 端口暴露服务。
- 2. 服务提供者，向注册中心发布自己提供的服务。
- 3. 服务消费者在第一次调用时，向注册中心下载(订阅)自己所需的服务的信息，以后每 1 分钟下载一次。
- 4. 注册中心主动向客户端通知服务端节点的变化情况。
- 5. 服务消费者，从服务提供者地址列表中，基于软负载均衡算法，选一个服务提供者发起调用。

特性说明：

- 注册中心负责服务地址的注册与查找，相当于目录服务。
- 注册中心，服务提供者，服务消费者三者之间均为长连接。
- 注册中心通过长连接感知服务提供者的存在，服务提供者某个节点死机，将记录下来供服务消费者定时来下载。
- 注册中心短时间停机，不影响已运行的提供者和消费者，消费者在本地内存缓存了提供者列表。
- 注册中心是可选的，服务消费者可以直连服务提供者进行调用。
- 服务提供者多个节点全部死掉后，服务消费者将无法调用。
- 服务提供者无状态，可动态增减节点，服务消费者每 1 分钟向注册中心取一次。

5.3. RSF 服务注册中心

服务注册中心(HRC)是 Web 应用，可以用浏览器访问，是众多服务的管理者，是“服务”的总控制中心。主要用途，参见 RSF 体系结构。

5.3.1. 访问生产环境的注册中心

域名是：register.org.hc360.com，所有业务系统通过这个域名访问生产环境的注册中心。
先登录中转机(需要账号)
用浏览器访问地址：http://register.org.hc360.com/register/login.htm (需要账号)

5.3.2. 访问测试环境的注册中心

地址是： http://register.org.hc360.com/register/login.htm
用户名：MMT 管理员
密　码：1

#70 环境

```
192.168.44.15 register.org.hc360.com    #注册管理中心
192.168.44.15 configure.org.hc360.com    #配置管理中心
192.168.44.22 sso.hc360.com
```

#75 环境

```
192.168.44.16 register.org.hc360.com    #注册管理中心
192.168.44.16 configure.org.hc360.com    #配置管理中心
192.168.44.23 sso.hc360.com
```

#81 环境

```
192.168.44.38 register.org.hc360.com    #注册管理中心
192.168.44.38 configure.org.hc360.com    #配置管理中心
192.168.44.21 sso.hc360.com
```

#113 环境

```
192.168.44.45 register.org.hc360.com    #注册管理中心
192.168.44.45 configure.org.hc360.com    #配置管理中心
192.168.44.36 sso.hc360.com
```

所有业务系统通过这个域名访问服务注册中心：register.org.hc360.com
注：以上配置信息于 2013-02-25 更新，当你使用时如已发生变化请以实际配置为准。

5.3.3. 服务注册中心的操作界面

一、登录到服务注册中心后，在左侧菜单—>服务注册管理，可查看全部已经注册的服务信息，如下图。

系统名：服务提供者所属的系统。

服务名：接口的全限定名。

发布者：服务的发布人，用于找到负责人。

所属部门：服务的发布人的部门，为跨部门应用而准备。

服务显示名：服务的简名，只为查看。

Jar 版本号：Rsf 的版本号。

接口描述：接口的说明文档，显示的弹窗中，这是重点。

节点信息：一个应用布署了多个节点，一个服务接口就会有多个提供者，它们之间只是 IP,端口不同。

节点信息--IP：某个节点的 IP。

节点信息—端口：某个节点的端口。

节点信息—状态：某个节点的状态。

节点信息—操作：停用、启用某个节点的服务接口。停止消息会被推送给客户端，客户端将不再向这个节点的服务接口发起请求。

二、查看某个接口的详细描述： 在列表中点击“View”按钮，弹出窗口。

可以复制到 IDE 中，参与编译。

服务接口描述

```
package com.hc360.org.infrastructure.chat.rsf;

import com.hc360.b2b.exception.MmtException;

/**
 * 商务中心--洽谈会管理
 *
 * @author huguoqing
 * @version 4.0 Jul 2, 2012
 * @since 4.0
 */
public interface MmtServcieManager {

   /**
    * 当前用户报名（初审通过，初审未通过，复审通过，复审未通过）数据统计,默认显示全部
    * @param username   用户ID
    * @param checkstate 审核状态：0-初审未通过 1-初审通过 2-拒审 4-复审通过
    * @param page 第几页
    * @param pageSize 每页显示多少条
    * @return
    * @throws MmtException
    */
```

5.4. RSF 监控中心（试用版）

RSF 监控中心是 rsf 服务调用情况的总监控中心。是实现服务治理的重要组成部分。

主要作用：调用统计，统计每天的服务调用量、响应时间、失败次数，作为容量规划的参考指标

访问生产环境的 RSF 监控中心：

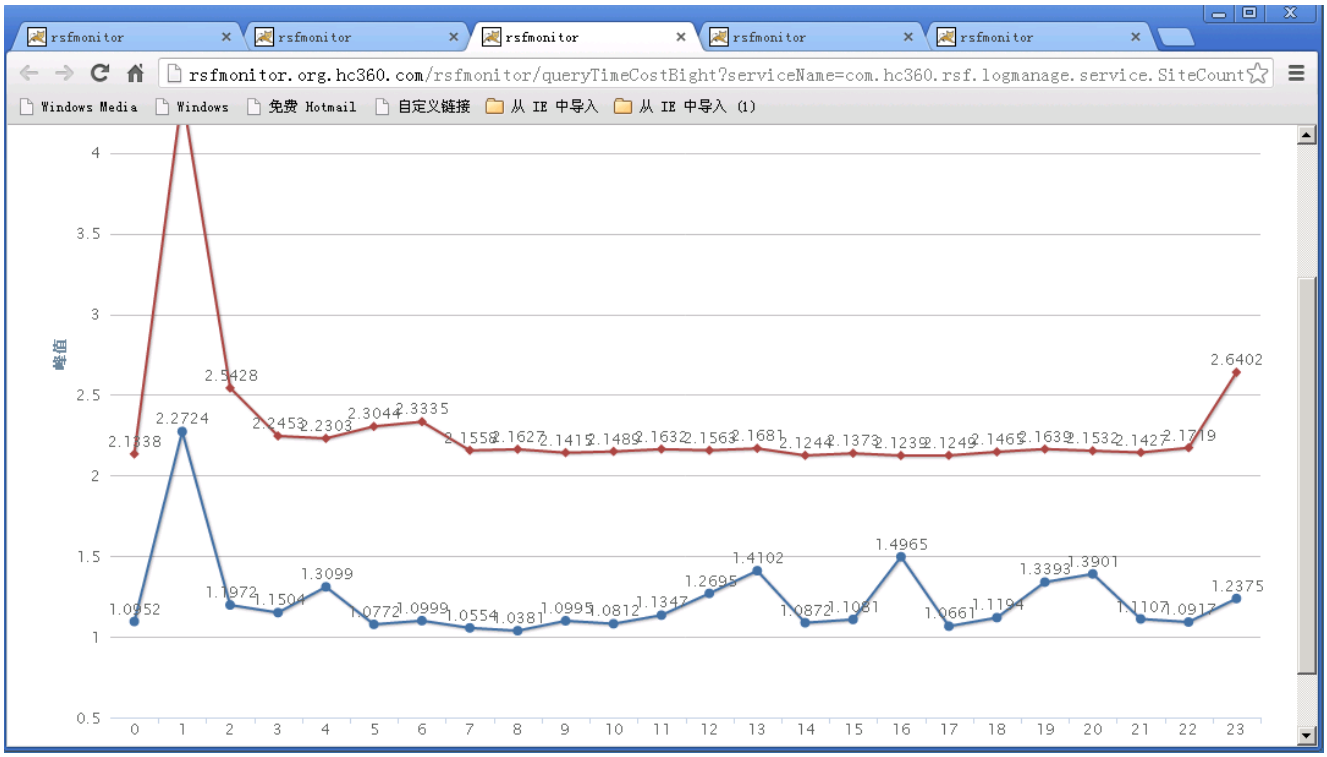
先登录中转机(需要账号)

用浏览器访问地址：http://rsfmonitor.org.hc360.com/rsfmonitor/

访问测试环境的 RSF 监控中心：

先配置正确的测试环境的 host

用浏览器访问地址：http://rsfmonitor.org.hc360.com/rsfmonitor/



RSF2.1.0 对监控中心支持

客户端：RSF2.1.0

服务端：RSF 监控中心，rsf:// rsfmonitor.org.hc360.com:63639

从 RSF2.1.0 起，加入到向监控中心异步报告的能力。客户端每积攒 10000 条报告或已达到 1 分钟，就向监控中心报告“调用情况详情记录”，含每一次的调用目标、耗时等等。

监控中心每天新建一张原始数据表（MySQL 数据库）记录这些原始信息，晚上 N 时使用 SQL 分析出图表，把结果记录到结果表。原始数据表保留 30 天。

6. RSF 基本情况说明

6.1. 如何取得 RSF 的 jar 包

1. 在公司内部，使用浏览器访问公司内的nexus仓库：http://192.168.34.253/nexus/index.html#welcome
2. 搜索 rsf 字样，出现结果列表，下载 RSF 最新版本的 RSF jar 包。

6.2. RSF 默认使用哪些端口

服务端本地暴露服务，默认使用 63634 端口。也可使用动态端口，请看 RSF 服务端动态端口
服务端向注册中心发布服务，注册中心默认使用 63638 端口。（可以通过 rsf 配置文件进行修改）。
客户端使用随机端口。

6.3. RSF 依赖的第三方 Jar 包

了解 RSF 依赖的第三方 jar 包很重要。RSF 有 3 种使用场景：

1、 常规通信
2、 加密通信
3、 跨语言通信

以上 3 种场景依赖的 jar 包不同，具体看下面的表格。99%的通信都是常规通信，如果你不知道加密通信、跨语言通信这事儿，那就属于常规通信。

1、RSF 常规通信时

常规情况 RSF 依赖以下 jar 包。

jar 包	功能说明
mina-core-2.0.7.jar	NIO 通信
slf4j-api-1.5.10.jar	日志门面 slf4j
slf4j-log4j12-1.5.10.jar	日志门面 slf4j -log4j 桥
log4j-1.2.15.jar	log4j 日志包

2、RSF 加密通信时

RSF 加密通信时，依赖以下 jar 包

jar 包	功能说明
hasclient-1.5.jar	的加解密工具类（作者：谢幕）
common_codec-1.4.jar	Apache 的加解密算法。(不要使用 1.3 版本)
configure_client_1.4.0.jar	通过配置中心下载公钥私钥

3、RSF 跨语言通信时

RSF 跨语言通信时，依赖以下 jar 包

jar 包	功能说明
thrift-0.9.1.exe	使用 thrift-0.9.1.exe 读取 IDL 文件，生成各个语言(java、php、c#)的代码，放入你的项目参与运行。 登录 http://192.168.34.253/nexus 仓库，搜索 thrift-0.9.1.exe，进行下载
libthrift-0.9.1.jar	java 版的 RSF，依赖 libthrift-0.9.1.jar。 其它语言的 RSF，请参看其它语言的 RSF 的用户手册： 《RSF 用户手册-PHP 版本》 《RSF 用户手册-C#版》
zookeeper-3.4.5.jar	zookeeper 客户端包

注意 thrift 要使用 0.9.0 或 0.9.1 版本，其它版本未测试过。

7. 同步调用和异步调用

RSF 支持：同步调用、异步调用-无返回值、异步调用-有返回值，3 种调用方式。

7.1. 同步调用

客户端调用远程方法后，阻塞等待，直到服务端返回结果或超时，客户端程序才会继续向下执行。
同步调用是最常用的，这符合我们的编程习惯，让程序顺序执行，执行完第一步，才能执行第二步。99%的情况你应该使用它。

同步调用有 3 种结果：

a. 取得想要的返回值，99.99%应该是这个结果。
b. 服务端业务代码出错，客户端收到一个明确的通知(异常形式抛出)--“服务端出错了”。
c．超时，客户端发起调用后，就再也没有消息了，无出错，无反回值，什么都没有，太可怕了，客户端不能无限的等待下去，超时是唯一的出路。

问：接口的方法返回值是 void(无返回值)，也会阻塞吗？
答：会，实际通信层也有成功状态的数据包返回。

7.2. 异步调用-无返回值

客户端调用远程方法后，不阻塞。服务端也不会向客户端发送返回值。

这时的程序已不是顺序执行了，客户端调用远程方法后，不阻塞，继续执行后面的代码。同时这个调用的数据包开始通过网络调用远程的方法。服务端也不会向客户端返回任务结果信息。客户端也不知道服务端执行成功与否。

异步调用（无返回值）后的结果：无结果，客户端像什么也没发生一样。

问：接口的方法有返回值，客户端会收到吗？
答：不会，就不应该设计有返回值的方法。实际通信层也没有数据包返回。

应用场景：异步记录日志。性能优秀，RSF 表示毫无压力，也不影响客户端业务的执行速度。可以实现把各个业务节点的日志集中记录到一个“日志服务器”上。

7.3. 异步调用-有返回值

客户端通过 RSF 的配置文件，注册一个回调函数，客户端调用远程方法后，不阻塞，断续执行后续业务代码。客户端通过回调用函数接收服务端的返回值。服务端处理完成后，使用 CallBackHelper 工具类，向客户端推送数据，客户端的回调函数会接收到这个值，回调函数运行在一新的线程中，不保留当时业务线程上下文。

7.4. 成熟度

调用方式	IO	成熟度
同步调用	同步	成熟可靠
异步调用-无返回值	异步	成熟可靠
异步调用-有返回值	异步	不成熟，不建议使用。目前本功能是测试版本，只建议少量使用。 在复杂网络环境下，可能接收不到返回值。例如：客户端发起了调用，并等待回调函数接收返回值时， 网络连接断开了，一会网络又恢复了，服务端无法向客户端推送返回值 ，因为有“推送能力”的连接已不存在了，现在是一条新的连接，服务端不知道你(客户端)想要接收推送的结果。 计划在以后版本做出改进。

8. RSF 同步调用示例

本章是整篇用户手册中最重要的一章，“RSF 同步调用示例”是众多示例是最重要的示例。“RSF 同步调用”在 RSF 众多功能中占比约 20%，但你 80%的时候都是在使用它。

可以这样说：你 80%的时候，是在使用 RSF 的这 20%的(核心)功能。

概览表：

维度	IO 模型	配置方式	安全	服务发现	跨语言	注册中心
值	同步调用	XML	非加密通信	3 点通信 通过注册中心 发现服务	只运行于 Java 语言平台	db 注册中心

8.1. 服务端

8.1.1. 编写接口

服务接口的定义要经过深思熟虑，有大局意识，包名、接口名、方法名、参数都要从长远的全局角度思考，做到见名知意、便于分类管理，保持易用性很关键，返回值一定要清晰。服务在未来会被很多新生应用（客户端）调用，一旦上线再想修改就因难了，一定要深思。

未来肯定会有一批服务接口，成为慧聪网的核心基础服务接口，所有上层业务都依赖他。让你的名字出现在这里吧！

动作执行者：服务提供方（服务端）。

服务提供者需要根据业务特点，开发一个 Java 接口类。这个接口是 RSF 调用的核心，一切对本服务的调用都通过这个接口完成，接口的全限定名就是服务的唯一标识，若有多个种类的业务就写多个接口。

接口类上应该有丰富的注释说明文字，说明本接口的业务功能。接口方法上要有丰富的注释说明文字，说明本方法处理的业务、方法接受的参数、返回值。尽量的写的详细，这些文字会被调用者看到（通过**注册中心**查看）。

服务调用者会使用这个接口，可以通过**注册中心**取得文字版的接口源码，放在服务调用者的项目中参与编译。

如果接口中的方法返回值是自定义 POJO 对象，服务调用者要有和服务提供者一样的 POJO 对象的源码文件，并参与编译。定义 POJO 对象必须有默认的构造方法。

所有通过网络传递的对象（实参、返回值）必须实现 Serializable 接口，对象的成员如果是对象也必须实现 Serializable 接口。自

注意：服务接口尽量不要写重载方法，RSF 支持写重载方法，但做异步回调时有些麻烦，要多配置一个<rsf:clientMethod>标签。不写重载方法就方便了。

示例接口如下：

```
package com.hc360.xxx;
public interface UserService {
    public UserBean getUserInfo(String userName) throws Exception;
    public String addUser(UserBean user) throws Exception;
    public String getData(String id) throws Exception;
```

```
    public void getPush(int number) throws Exception;
}
```

```
package com.hc360.xxx;
import java.io.Serializable;
public class UserBean implements Serializable{
    private static final long serialVersionUID = 1L;
    private String name;
    private int age;
    private Object obj;
    public String toString(){
        return "[name:"+name+",age:"+age+"]";
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public Object getObj() {
        return obj;
    }
    public void setObj(Object obj) {
        this.obj = obj;
    }
}
```

8.1.2. 实现接口

动作执行者：服务提供方（服务端）。

服务提供者根据业务要求实现服务接口。

注意：UserServiceImpl 必须有默认的构造方法。POJO 对象必须有默认的构造方法。用于通过反射创建实例。

```
package com.hc360.xxx;
import com.hc360.rsf.config.callback.AddressTool;
public class UserServiceImpl implements UserService {
    public UserBean getUserInfo(String userName) throws Exception{
        UserBean user=new UserBean();
        user.setAge(5);
        user.setName(userName);
        System.out.println("我是服务端,"+AddressTool.toStringInfo());
        return user;
    }

    public String addUser(UserBean user) throws Exception{
        return "success";
    }

    public String getData(String id) throws Exception{
```

```
        return id;
    }

    public void getPush(int number) throws Exception{
        System.out.println("服务端收到调用,number="+number);
    }
}
```

8.1.3. 编写服务端的配置文件

动作执行者：服务提供方（服务端）。

编写服务端配置文件 rsf_server.xml

Registry 标签：配置服务注册中心的地址，用于找到服务注册中心。

Service 标签：配置一个服务接口，在本地暴露服务，并发布服务到服务注册中心。

Document 标签:请把接口的原代码粘贴在这里，并附上丰富的说明，用于在注册中心查看。

rsf_server.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rsf xmlns="http://code.hc360.com/schema/rsf"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rsf="http://code.hc360.com/schema/rsf">

    <rsf:registry host="register.org.hc360.com"></rsf:registry>
    <rsf:service displayName="用户信息服务" owner="赵磊" department="MMT开发部"
interfaceClass="com.hc360.xxx.UserService" class="com.hc360.xxx.UserServiceImpl" portalId="系统英文名称">
        <rsf:document><![CDATA[
            //以下信息是对服务接口的说明
            //把接口的原代码粘贴到这里，记得带着包名
        ]]></rsf:document>
    </rsf:service>
</rsf>
```

8.1.4. 启动服务端

加载以上配置文件，实现在服务端本地 63634 端口暴露服务，并把服务发布到注册中心。

通过 mian()方法调用 ConfigLoader 工具类启动。

```
package com.hc360.xxx;
import com.hc360.rsf.config.ConfigLoader;
public class MainServer {
    public static void main(String[] args) {
        String xmlPath = "classpath:com/hc360/xxx/rsf_server.xml";
        ConfigLoader configLoader = new ConfigLoader(xmlPath);
        configLoader.start();
        //以上代码在客户端或服务端系统启动时执行一次就够了。
    }
```

```
}
```

8.2. 客户端

8.2.1. 编写客户端的配置文件

动作执行者：服务调用方（客户端）。

编写客户端配置文件 rsf_client.xml

Refistry 标签：配置服务注册中心地址，用于找到服务注册中心。

Client 标签：配置服务接口。通过注册中心可以获得服务接口的源代码，复制下来放在本地项目中参与编译。客户端与服务端依赖同一个 Java 接口。

rsf_client.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rsf xmlns="http://code.hc360.com/schema/rsf"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rsf="http://code.hc360.com/schema/rsf">

    <rsf:registry host="register.org.hc360.com"></rsf:registry>
    <rsf:client id="clientUserServiceImpl" displayName="调用用户信息服务" owner="张三"
department="MMT开发部"
    interfaceClass="com.hc360.xxx.UserService" url="" portalId="系统英文名称">
    </rsf:client>
</rsf>
```

8.2.2. 启动客户端

加载以上配置文件，实现在客户端向注册中心下载服务提供者列表，并生成一个远程服务接口的本地代理，通过这个代理调用远程服务接口。

通过 mian()方法调用 ConfigLoader 工具类启动。

```
package com.hc360.xxx;
import com.hc360.rsf.config.ConfigLoader;
public class MainClent {
    public static void main(String[] args) throws Exception {
        String xmlPath = "classpath:com/hc360/xxx/rsf_client.xml";
        ConfigLoader configLoader = new ConfigLoader(xmlPath);
        configLoader.start();
        //以上代码在客户端或服务端系统启动时执行一次就够了
        //以下代码只有客户端需要执行，getServiceProxyBean()返回的远程服务接口的本地代理对象已缓存，
多次执行返回的都是同一个对象。
        UserService userService= (UserService)
configLoader.getServiceProxyBean("clientUserServiceImpl");//配置文件中的id
        // 像调用本地方法一样调用远程方法。
```

```
        UserBean user=userService.getUserInfo("zhangShan");
        System.out.println(user);
        ConfigLoader.destroy();
    }
}
```

运行服务端的 main 方法。

运行客户端的 main 方法。

一切都顺利的话，你应该看到调用成功了。

9. 如何启动与停止 RSF

启动与停止 RSF 的方法，请见下表。

请根据项目类型，选择适合的方式启动 RSF。

因为 RSF 打开了很多系统资源（端口、线程），请尽量正确关闭 RSF，可执行关闭流程，释放资源。

kill 进程被认为是不正确的停止方法。

应用类型	启动 RSF	关闭 RSF	特点
非 Web 项目	**ConfigLoader** *new ConfigLoader（“path”）*	明确调用 ConfigLoader.destroy() 方法	基本启动器，可适用于任意扩展
Web 项目	**RsfListener** *容器启动时 Listener 被执行*	容器关闭时，Listener 被执行	在 web.xml 中配置一个 Listener
Spring 项目	**RsfSpringLoader** *spring 启动时执行*	spring 关闭时，执行 destroy 方法	在 Spring 配置文件中配置一个启动器，RSF 可以使用 Spring 容器中的 bean 做为服务接口的实现类。

9.1. Spring 项目使用 RsfSpringLoader 启动 RSF

9.1.1. RsfSpringLoader 支持的配置文件路径

一、绝对路径：file:D:\\config\\rsf.xml

二、类路径：classpath:com/hc360/rsf/config/xml/rsf.xml 或 com/hc360/rsf/config/xml/rsf.xml

三、Web 应用根路径：/WEB-INF/rsf.xml

9.1.2. 使用 RsfSpringLoader 启动 RSF

如果你的项目使用了 Spring 构架，并由 Spring 容器管理 bean,这些 bean 已注入必要的依赖或带有事务。同时 RSF 服务接口的实现类，就是 Spring 容器中的 bean，RSF 需要使用这些 bean，可以做如下配置。

本工具用于持有 Spring 的 ApplicationContext 容器,可以使用 RsfSpringLoader.getBean('xxxx')的静态方法得到 spring 容器中的 bean 对象。

在 Spring 的配置文件中加入以下内容。

```
<bean class="com.hc360.rsf.config.RsfSpringLoader" lazy-init="false" destroy-method="destroy">
    <property name="rsfConfigLocations">
        <list>
            <value>classpath:com/hc360/rsf/config/spring_loader/rsf_*.xml</value>
            <value>com/hc360/rsf/config/spring_loader/rsf_server.xml</value>
        </list>
```

```
        </property>
        </bean>
```

RsfSpringLoader 内部也是使用 ConfigLoader 完成的启动。ConfigLoader 类是在 com.hc360.rsf.config. RsfSpringLoader 完成实例化的，全局只有一个实例。可以通过 RsfSpringLoader. getConfigLoader()静态方法取得，一般客户端需要取得。

```
ConfigLoader configLoader = RsfSpringLoader. getConfigLoader();
```

<property name="rsfConfigLocations">标签和子标签用于配置 RSF 配置文件的路径，支持*号。是可选项，如果未配置，将不加载 rsf 的配置文件，不执行启动动作。

9.1.3. 使用 RsfSpringLoader 停止 RSF

上面配置了 destroy-method="destroy"，当 spring 容器退出时，执行 destroy 方法。

9.2. 非 Web 项目使用 ConfigLoader 启动 RSF

9.2.1. ConfigLoader 支持的配置文件路径

```
一、绝对路径：file:D:\\config\\rsf.xml
二、类路径：classpath:com/hc360/rsf/config/xml/rsf.xml
```

9.2.2. ConfigLoader 加载多个配置文件

```
ConfigLoader configLoader = new ConfigLoader(new String[]{xmlPath1,xmlPath2});
```

加载的配置文件的内容是 xmlPath1,xmlPath2 两个路径表示配置文件内容的总和。无作用域概念，例如只想配置一个注册中心，只在一个配置文件中配置就可以。

9.2.3. 使用 ConfigLoader 启动 RSF

```
String xmlPath = "classpath:com/hc360/配置文件路径/rsf.xml";
ConfigLoader configLoader = new ConfigLoader(xmlPath);
configLoader.start();
//以上代码在客户端或服务端系统启动时执行一次就够了

//以下代码只有客户端需要执行，可执行多次，getServiceProxyBean()返回的远程服务接口的本地代理对象已缓存，多次执行返回的都是同一个对象。
UserService userService= (UserService) configLoader.getServiceProxyBean("clientUserServiceImpl");//配置文件中的id
// 像调用本地方法一样调用远程方法。
UserBean user=userService.getUserInfo("");
```

```
System.out.println(user);
```

9.2.4. 使用 ConfigLoader 停止 RSF

可以调用静态方法

```
ConfigLoader.destroy()
```

完成停止工作，本方法是 RSF 1.3.0 版本新添加的。

RSF 1.2.1(含)以前的版本，一直使用 AbstractConfig.destroy()方法停 RSF，在 RSF 1.3.0 版本为了保持兼容性依然保留这个方法。建议大家使用 ConfigLoader.destroy()方法.使用 ConfigLoader 启动，使用 ConfigLoader 停止，API 更友好一些。其实 ConfigLoader.destroy()方法也是调用了 AbstractConfig.destroy()方法，它们执行效果相同。

9.3. Web 项目使用 RsfListener 启动 RSF

9.3.1. RsfListener 支持的配置文件路径

一、绝对路径：file:D:\\config\\rsf.xml
二、类路径：classpath:com/hc360/rsf/config/xml/rsf.xml
三、Web 应用根路径：/WEB-INF/rsf.xml

9.3.2. 使用 RsfListener 启动 RSF

如果是在 WEB 应用中使用 RSF，想要在容器启动时加载 RSF 的配置文件，可以通过 ServletContextListener 来完成。RSF 提供了 ServletContextListener 的实现类 com.hc360.rsf.config.RsfListener。可以通过在 web.xml 中配置一个 listener 来实现启动工作。

例如：

```
<context-param>
<param-name>rsfConfigFilePaths</param-name>
<param-value>classpath:com/xxx/rsf.xml,file:D:\\rsf.xml,/WEB-INF/rsf.xml</param-value>
</context-param>
<listener>
    <listener-class>com.hc360.rsf.config.RsfListener</listener-class>
</listener>
```

说明：通过 context-param 参数指明 rsf.xml 文件的位置,如果未指定默认值是：classpath:rsf.xml。其中 param-name 的值是 rsfConfigFilePaths，不可以修改。

9.3.3. 如何通过 RsfListener 拿到 ConfigLoader 类的实例

RsfListener 监听器内部也是使用 ConfigLoader 完成的启动。ConfigLoader 类是在 com.hc360.rsf.config.RsfListener 完成实例化的，全局只有一个实例。可以通过 RsfListener. getConfigLoader()

静态方法取得，一般客户端需要取得。

```
ConfigLoader configLoader =RsfListener. getConfigLoader();
UserService userService= (UserService) configLoader.getServiceProxyBean("clientUserServiceImpl");//配置文件中的id
userService.xxx();//发起业务调用
```

9.3.4. 使用 RsfListener 停止 RSF

如果你配置了 com.hc360.rsf.config.RsfListener 监听器，在容器关闭时会自己执行监听器中的退出方法，释放资源。Kill 进程除外。其实 RsfListener 类就是调用 ConfigLoader.destroy()方法完成的停止工作。

9.3.5. 使用 RsfServlet 启动与停止 RSF

如果你不喜欢使用 Listener 监听器，RSF 还提示了 Servlet 来完成启动与停止的工作，RsfServlet 的特性与 RsfListener 一样，接收的参数一样，提供的方法一样，可以互相替换。

```
<!-- servlet启动器 -->
<context-param>
    <param-name>rsfConfigFilePaths</param-name>
    <param-value>
        /WEB-INF/conf/rsf_server.xml,
        classpath:com/hc360/rsf/config/spring/rsf_server.xml,
    </param-value>
</context-param>
<servlet>
    <servlet-name>rsfStart</servlet-name>
    <servlet-class>com.hc360.rsf.config.RsfServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

10. 使用 XML 配置 RSF

10.1. 完整配置文件示例

父配置文件示例

```
<?xml version="1.0" encoding="UTF-8"?>
<rsf xmlns="http://code.hc360.com/schema/rsf"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rsf="http://code.hc360.com/schema/rsf">

    <!-- db注册中心 -->
    <rsf:registry id="reg1" type="db" host="register.org.hc360.com" ></rsf:registry>

    <!-- zookeeper注册中心 -->
    <rsf:registry id="reg2" type="zookeeper"
address="192.168.44.112:2181,192.168.44.113:2181,192.168.44.114:2181"></rsf:registry>

    <!-- rsf协议-->
    <rsf:protocol id="protocol_rsf"  name="rsf" port="63634" ></rsf:protocol>

    <!-- thrift协议-->
    <rsf:protocol id="protocol_thrift"  name="thrift" ></rsf:protocol>

    <!-- 目录扫描，引入多个子配置文件-->
    <rsf:include path="/WEB-INF/conf/rsf_client_*.xml"/>
    <rsf:include path="/WEB-INF/conf/rsf_server_*.xml"/>
</rsf>
```

子配置文件示例　rsf_client_*.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rsf xmlns="http://code.hc360.com/schema/rsf"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rsf="http://code.hc360.com/schema/rsf">

    <rsf:service
        portalId="detail"
        displayName="服务的中文名"
        owner="服务端开发人员姓名"
        department="服务端开发人员部门"
        interfaceClass="com.hc360.rsf.api.bean.data1.UserService"
        class="com.hc360.rsf.api.bean.data1.UserServiceImpl">
        <rsf:document><![CDATA[ //接口的源代码copy到这里 ]]></rsf:document>
    </rsf:service>
</rsf>
```

子配置文件示例　rsf_server_*.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rsf xmlns="http://code.hc360.com/schema/rsf"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rsf="http://code.hc360.com/schema/rsf">

    <rsf:client
        portalId="order"
```

```
        id="clientUserServiceImpl"
        displayName="客户端业务描述"
        owner="客户端开发人员姓名"
        department="客户端开发人员部门"
        interfaceClass="com.hc360.rsf.api.bean.data1.UserService">
    </rsf:client>
</rsf>
```

10.2. 命名空间

xmlns:rsf=http://code.hc360.com/schema/rsf，当把 rsf 命名空间做为默认命名空间时，标签的前缀可以不写，例如 <rsf:registry>写成<registry>。如果 RSF 框架的配置文件写在 Spring 配置文件中，默认命名空间就是 spring 的默认命名空间，RSF 框架的配置标签一定要带前缀写成<rsf:registry>。

示例：

```
<rsf xmlns="http://code.hc360.com/schema/rsf"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rsf="http://code.hc360.com/schema/rsf">
</rsf>
```

10.3. <rsf:include>标签

用于在 RSF 配置文件中引入子配置文件。实现对配置文件的分层管理，全局配置放在父文件中，让配置文件层次更清晰。这是 RSF2.0 新加入的功能，虽然姗姗来迟，但为了支持扫描/xml/**/rsf_*.xml 这种路径，做了大量工作。不管怎样我来了。

示例：

```
<rsf:include path="classpath:com/hc360/rsf/api/xml/include/rsf_*.xml"/>

<rsf:include path="com/hc360/rsf/api/xml/**/rsf_*.xml"/>

<rsf:include path="file:E:/workspace3.6/rsf/api/xml/**/rsf_*.xml"/>

<rsf:include path="/WEB-INF/conf/rsf_*.xml"/>
```

标签属性列表：

标签	属性	类型	是否必填	缺省值	描述	版本
<rsf: include>	path	string	必选	无	被包含配置文件的路径。 不支持 ./ ../ 这种形式的相对路径 支持 file: 前缀 支持 classpath: 前缀 支持无前缀，按 classpath:处理 支持/WEB-INF/ 前缀（仅限 web 项目）	2.0

					支持路径中写 *, **, ?	

“?” ：匹配一个字符，如“config?.xml”将匹配“config1.xml” ；

“*” :匹配多个字符串 ,如“cn/*/config.xml”将匹配“com/hc360/config.xml” ,但不匹配“cn/config.xml” ；而“cn/config-*.xml”将匹配“cn/config-dao.xml” ；

“**” ：匹配路径中的零个或多个目录，如“cn/**/config.xml”将匹配“cn/config.xml” ，也匹配“com/hc360/spring/config.xml” ;而“com/hc360/config-**.xml”将匹配“com/hc360/config-dao.xml” ，即把“**”当做两个“*”处理。

建议：

配置文件要分层管理，把全局配置如<rsf:registry>标签、<rsf: protocol>标签放在父配置文件中，把众多的<rsf:service>标签、<rsf:client>标签放在子配置文件中。并且支持目录扫描，让配置文件层次更清晰。

10.4. <rsf:registry>标签

注册中心配置标签，用于服务端发布服务、客户端下载服务时指明注册中心所在的位置。

一个注册中心对应一个<rsf:registry>标签。一般情况全局只需要一个<rsf:registry>标签就够了。

不配置本标签，就不会向注册中心注册服务。

示例：

```
<rsf:registry id="" host="register.org.hc360.com"></rsf:registry>
```

标签	属性	类型	是否必填		缺省值	描述	版本
<rsf:registry>	id	string	可选		无	Bean 的 id，不可重复	1.1
<rsf:registry>	host	string	必填	type=”db”时必填	无	db 注册中心 IP、域名	1.1
<rsf:registry>	port	int	可选		63638	db 注册中心端口	1.1
<rsf:registry>	timeout	int	可选		5000	与 db 注册中心通信超时时间 ms ,对 zookeeper 注册中心无效	1.1
<rsf:registry>	type	string	可选 如果想使用 zookeeper 注册中心，就必填。		db	注册中心类型， zookeeper：新注册中心 db：数据库注册中心（老）	2.0
<rsf:registry>	address	string	必填 type=”zookeeper”时必填		无	zookeeper 注册中心地址，支持填写多个地址，格式：ip:port, ip:port, ip:port	2.0

db 注册中心（数据库）：Java 语言默认使用 db 注册中心，一切没有变化，就当没有 zookeeper 注册中心。

zookeeper 注册中心：RSF2.0 支持跨语言调用时，使用 zookeeper 作为注册中心，用于支持多语言。

建议：

2.0 版本	**全局只能配置一个 db 注册中心，配置多个 db 注册中心时将无情的抛出异常。** **全局只能配置一个 zookeeper 注册中心，配置多个 zookeeper 注册中心时将无情的抛出异常。** **全局最多只能配置 1 个 db 注册中心和 1 个 zookeeper 注册中心。**
1.3.x 版本	全局可配置多个 db 注册中心，但一般只需要一个 db 注册中心就够了。 全局可配置多个 db 注册中心，正常情况 ip、port 应不相同，你非要无赖配置了多个一样的 db 注册中心（有人这么干了），将被温柔的过滤掉（为了照顾已上线的系统就没有抛出异常）。 全局只能配置一个 zookeeper 注册中心，配置多个 zookeeper 注册中心时将无情的抛出异常（没有历史包袱）。 建议全局配置可配置 1 个 db 注册中心和 1 个 zookeeper 注册中心。

10.5. <rsf:protocol>标签

协议配置标签。用于配置使用的通信协、线程池。只对服务端有效。
可以不配置本标签，RSF 将使用默认值。

示例：

```
<rsf:protocol id="protocol_rsf"  name="rsf" port="63634" ></rsf:protocol>
```

标签	属性	类型	是否必填	缺省值	描述	版本
<rsf: protocol>	id	string	可选	无	Bean 的 id ,用于被引用	1.1
<rsf: protocol>	host	string	可选	本机 IP	服务暴露在这个 IP，只对 rsf 协议有效	1.1
<rsf: protocol>	port	int 或 string	可选，建议不设置此值，程序会自动使用默认值。	63634-63600 如果 63634 被占用，按顺序尝试其它端口，最终选择一个可用的端口	服务暴露在这个端口。可以设置一个端口，也可以设置多个(一段)端口,用于解决端口冲突。 只对 rsf 协议有效 ,thrift 协议使用的端口在 service 标签中配置	1.1 1.2 有扩展 ,添加了动态端口
<rsf: protocol>	name	string	可选	rsf 协议	传输协议类型名称，目前支持两种协议 ,必需 2 选 1。 rsf：协议（自从 v1.1) thrift：协议(自从 v2.0)	1.1 2.0 有扩展

<rsf: protocol>	payload	int	可选	88388608(=8M)	请求数据包大小限制，单位：字节。设置为小于等于 0 时，采用默认值 。	1.1
<rsf: protocol>	~~threads~~	int	可选	200	服务端线程池大小(只对 fixed 线程池有用)(弃用并保留)	1.1
<rsf: protocol>	threadpool	string	可选	服务端 :fixed，客户端：cached	服务端线程池类型，可选：fixed/cached/mixed，字符不区分大小写。设置其它字符将报异常。	1.1
<rsf: protocol>	corePoolSize	int	可选	线程池类型不同，默认值也不同，请参见“线程模型”一章。keepaliver 的单位是 ms	线程池的基本大小	1.3.0
<rsf: protocol>	maximumPoolSize	int	可选		线程池最大大小	1.3.0
<rsf: protocol>	queueSize	int	可选		线程池任务队列大小	1.3.0
<rsf: protocol>	keepalive	int	可选		线程活动保持时间	1.3.0

建议：

RSF2.0 全局可有一个<rsf: protocol name=”rsf”>标签,用于启用 rsf 协议。

RSF2.0 全局可有一个<rsf: protocol name=”thrift”>标签,用于启用 thrift 协议。

RSF1.3.x 全局最多只能有一个<rsf: protocol>标签

10.6. <rsf:service>标签

服务端配置标签。

一个服务接口对应一个本标签，可以有多个<rsf:service>标签。

示例：

```
    <rsf:service
        portalId="detail"
        displayName="服务的中文名"
        owner="服务端开发人员姓名"
        department="服务端开发人员部门"
        interfaceClass="com.hc360.rsf.api.bean.data1.UserService"
        class="com.hc360.rsf.api.bean.data1.UserServiceImpl">
        <rsf:document><![CDATA[ //接口的源代码copy到这里 ]]></rsf:document>
    </rsf:service>
```

标签	属性	类型	是否必填	缺省值	描述	版本
<rsf:service>	id	string	可选	无	Bean 的 id，默认使用 class 的值	1.1
<rsf:service>	displayName	string	必填	无	服务的中文名称，用于显示在注册中心	1.1
<rsf:service>	owner	string	必填	无	服务的创建人（负责人）	1.1
<rsf:service>	department	string	必填	无	服务创建人所属部门	1.1
<rsf:service>	interfaceClass	Class	必填	无	接口类型	1.1
<rsf:service>	springBean	Object	与 class 选填一个	无	从 spring 容器中通过 id 取出 bean,这个 bean 实现了服务接口。 必须使用 RsfSpringLoader 启动器 ,本参数才能工作。 springBean=" id" 按 ID 取。 springBean=" auto" 按接口名取。	1.3
<rsf:service>	class	Object	与 springBean 选填一个	无	接口实现类全限定名	1.1
<rsf:service>	registries	string	可选	rsf 协议默认只向 db 注册中心注册。 thrift 协议默认只向 zookeeper 注册中心注册。	向指明向哪个注册中心注册、下载，值为<rsf:registry>的 id 属性值。向多个注册中心注册、下载时，多个注册中心的 id 用逗号分隔。 rsf 协议默认只向 db 注册中心注册。 thrift 协议默认只向 zookeeper 注册中心注册。 rsf 协议可向 db 注册中心注册 rsf 协议可向 zookeeper 注册中心注册 thrift 协议可向 zookeeper 注册中心注册 thrift 协议不可向 db 注册中心注册	1.1 2.0 有扩展
<rsf:service>	register	boolean	可选	是	是否向 "服务注册中心" 发布本接口。无论是否都会在本地暴露服务。点对占调用不受此参数影响。	1.1
<rsf:service>	weight	int	可选	100	权重（目前不支持）	1.1
<rsf:service>	portalId	string	必填		系统名称，一般写英文名称。	1.1
<rsf:service>	security	boolean	可选		True 为加密通信。控制级别为接口级，不能精确到方法级。	1.3
<rsf:service>	protocolId	string	可选	rsf 协	使用什么协议暴露服务，只可以选用一	2.0

			若使用thrift协议则必选	议	种协议。值是<rsf:protocol/>标签的 ID。 使用<rsf:protocol/>标签配置 rsf 协议或 thrift 协议，再引用<rsf:protocol/>标签的 ID。	
<rsf:service>	thriftPort	int	可选	随机值	使用 thrift 协议时，一个接口占用一个端口，本属性用于指明占用的端口。 如果没有指明 默认使用 63400-63600 之间的随机值做为端口号。	2.0
<rsf:service>	zookeeperPath	string	可选	空值	向 zookeeper 注册中心注册服务时使用的 path,其它语言（php \c++\.net）向 zookeeper 注册中心注册的服务名不一定是接口名，可以使用本属性，注册任意服务名。服务名要全局唯一。 若空，默认使用接口的全限定名。	2.0
<rsf:service>	encode	string	可选	空值	用于说明入参、返回值中的中文的编码，只供在注册中心显示使用，不对程序运行产生影响	2.0

10.7. <rsf:document>标签

说明文档配置标签。

是<rsf:service>的子标签。一个<rsf:service>标签必需有一个<rsf:document>子标签。

本标签是必选项，<rsf:service>必需有一个<rsf:document>子标签。

示例：

```
<rsf:service key=value >
    <rsf:document><![CDATA[本服务接口的说明文档，内容应尽量全面。包括服务接口总体业务说明、每个方法的业务说明，方法接收参数说明，方法返回值说明。最好直接把接口类的源代码copy到这里(包含包名等全部信息)。 ]]></rsf:document>
</rsf:service>
```

标签	属性	类型	是否必填	缺省值	描述	版本
<rsf:document>	id	CDATA	必填	无	本服务接口的说明文档，内容应尽量全面。包括服务接口总体业务说明、每个方法的业务说明，方法接收参数说明，方法返回值说明。最好直接把接口类的源代码 copy 到这里(包含包名等全部信息)。	1.1

10.8. <rsf:client>标签

客户端配置标签。

一个服务接口的引用对应一个标签，可以有多个<rsf:client>标签。

示例：

```
<rsf:client
    portalId="order"
    id="clientUserServiceImpl"
    displayName="客户端业务描述"
    owner="客户端开发人员姓名"
    department="客户端开发人员部门"
    interfaceClass="com.hc360.rsf.api.bean.data1.UserService">
</rsf:client>
```

标签	属性	类型	是否必填	缺省值	描述	版本
<rsf:client>	id	string	可选	无	Bean 的 id，用于接口的代理对象	1.1
<rsf:client>	displayName	string	必填	用于统计、管理，不会对程序运行产生影响	客户端业务描述 ,说明你调用这个服务做什么	1.1
<rsf:client>	owner	string	必填		客户端开发人员姓名	1.1
<rsf:client>	department	string	必填		客户端开发人员部门	1.1
<rsf:client>	interfaceClass	Class	必填		接口类型	1.1
<rsf:client>	timeout	int	可选	3000ms	远程服务调用超时时间(毫秒),设置的值 小于等于 0 将使用默认值。	1.1
<rsf:client>	registries	string	可选	rsf 协议默认只向 db 注册中心下载。 thrift 协议默认只向 zookeeper 注册中心下载。	向指明向哪个注册中心注册、下载，值为<rsf:registry>的 id 属性值。向多个注册中心注册、下载时，多个注册中心的 id 用逗号分隔。 rsf 协议默认只向 db 注册中心下载。 thrift 协议默认只向 zookeeper 注册中心下载。 rsf 协议可向 db 注册中心下载 rsf 协议可向 zookeeper 注册中心下载 thrift 协议可向 zookeeper 注册中心下载 thrift 协议不可向 db 注册中心下载	1.1 2.0 有扩展
<rsf:client>	loadbalance	string	可选	random	负载均衡策略，可选值：	1.1

					random,roundrobin,leastactive，分别表示：随机，轮循，最少活跃调用。如果配置错误使用默认值。	
<rsf:client>	mock	token	可选	无	mock 对象，必须实现服务接口。在开发阶段，可能无法连接到服务端，可以写一个 mock 对象模拟服务端，方便开发。当 mock 值为空时，才会真正向服务端发出网络请求。 目前 mock 只支持同步调用。	1.1
<rsf:client>	url	string	可选	无	点对点直连 URL 的配置。 本值为空，才向注册中心订阅服务列表。 本值不空，将直接向这个地址请求。示例如下： 127.0.0.1：指明 ip,使用默认协议、端口 127.0.0.1:63634：指明 ip,端口，使用默认协议 rsf://127.0.0.1:63634：指明 ip、端口、协议 thrift://127.0.0.1:9008：指明 ip、端口、协议	1.1
<rsf:client>	portalId	string	必填		系统名称，一般写英文名称。	1.1
<rsf:client>	security	Boolean	可选		True 为加密通信。控制级别为接口级，不能精确到方法级。	1.3
<rsf:client>	protocolId	string	可选	rsf 协议	使用什么协议暴露服务 rsf：协议 (自从 v1.1) thrift：协议(自从 v2.0 增加)	2.0
<rsf:client>	zookeeperPath	string	可选	无值	向 zookeeper 注册中心下载服务时使用的 path, 其它语言（php \c++\.net）向 zookeeper 注册中心注册的服务名不一定是接口名，客户端要指明“服务名”，用于向注册中心查找服务。 若为空，默认使用接口的全限定名。	2.0

10.9. <rsf:clientMethod>标签

客户端接口的方法配置标签，是<rsf:client>的子标签。一个方法对应一个标签，可以有多个<rsf:clientMethod>

标签。

用于说明：哪一个方法是异步执行的、是可以接受回调的，是可以接受推送数据的（通过回调函数接受）。

```
<rsf:client>
    <rsf:clientMethod name="getUserInfo" parameterTypes="java.lang.String" callback="回调函数" timeout="5000" async="false"></rsf:clientMethod>
</rsf:client>
```

标签	属性	类型	是否必填	缺省值	描述	版本
<rsf:clientMethod>	id	string	可选	无	Bean 的 id	1.1
<rsf:clientMethod>	name	string	必填	无	方法名	1.1
<rsf:clientMethod>	parameterTypes	string	可选	无	参数的类型。用于区分重载方法，如果没有重载方法，本参数可以为空。	1.1
<rsf:clientMethod>	callback	token	可选	无	回调函数，想异步接受数据时，才需要配置。值是类的全限定名，这个类必须实现 RSF 框架提供的 CallBack 接口。	1.1
<rsf:clientMethod>	timeout	int	可选	父元素的值	超时时间	1.1
<rsf:clientMethod>	async	boolean	可选	否	本方法是否异步执行，需要异步执行时才配置。异步执行时回调函数是可选的，有则回调，无则不回调。	1.3

11. RSF 异步调用示例

概览表：

维度	IO 模型	配置方式	安全	服务发现	跨语言	注册中心
值	异步调用	XML	非加密通信	3 点通信 通过注册中心 发现服务	只运行于 Java 语言平台	db 注册中心

11.1. 异步调用--无返回值

示例代码

rsf_server.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rsf xmlns="http://code.hc360.com/schema/rsf"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rsf="http://code.hc360.com/schema/rsf">

    <rsf:service id="" displayName="用户测试服务" owner="赵磊" department="MMT开发部"
interfaceClass="com.hc360.rsf.api.bean.data1.UserService"
    ref="" class="com.hc360.rsf.api.bean.data1.UserServiceImpl" portalId="测试"
security="false" register="false">
        <rsf:document><![CDATA[
            //记得带着包名
        ]]></rsf:document>
    </rsf:service>
</rsf>
```

rsf_client.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rsf xmlns="http://code.hc360.com/schema/rsf"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rsf="http://code.hc360.com/schema/rsf">

    <rsf:client id="clientUserServiceImpl" displayName="调用用户测试服务" owner="张三"
department="MMT开发部"
    interfaceClass="com.hc360.rsf.api.bean.data1.UserService" portalId="测试"
url="127.0.0.1">
        <rsf:clientMethod name="getUserInfo" parameterTypes="java.lang.String"
async="true"></rsf:clientMethod>
    </rsf:client>
</rsf>
```

服务端代码

```
package com.hc360.rsf.api.push_async_void;
import com.hc360.rsf.config.ConfigLoader;
public class Server {

    /**
```

```
     * 加载rsf_server.xml配置文件
     *
     * 在本地暴露服务
     *
     * 向注册中心注册服务
     *
     * @param args
     */
    public static void main(String[] args) {
        String xmlPath = "classpath:rsf_server.xml";
        ConfigLoader configLoader = new ConfigLoader(xmlPath,Server.class);
        configLoader.start();
    }
}
```

客户端代码

```
package com.hc360.rsf.api.push_async_void;
import com.hc360.rsf.api.bean.data1.UserBean;
import com.hc360.rsf.api.bean.data1.UserService;
import com.hc360.rsf.config.ConfigLoader;
public class Client {

    /**
     * 加载rsf_server.xml配置文件
     *
     * 从注册下载服务提供者列表
     *
     * 发起调用
     *
     * @param args
     */
    public static void main(String[] args) {
        String xmlPath = "classpath:rsf_client.xml";
        ConfigLoader configLoader = new ConfigLoader(xmlPath,Client.class);
        configLoader.start();
        UserService userService= (UserService)
configLoader.getServiceProxyBean("clientUserServiceImpl");//配置文件中的id
        for(int i=1;i<=10;i++){
            try {
                UserBean user=userService.getUserInfo(""+i);//非阻塞调用
                System.out.println("结果: "+user);//无返值，结果为null

                if(i%100==0){
                    System.out.println("完成了"+(i)+"次");
                }
            } catch (Exception e) {
                e.printStackTrace();
            }finally{
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        ConfigLoader.destroy();
    }
}
```

运行服务端的 main 方法。
运行客户端的 main 方法。
一切都顺利的话，你应该看到调用成功了。

11.2. 异步调用--有返回值

示例代码
rsf_server.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rsf xmlns="http://code.hc360.com/schema/rsf"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rsf="http://code.hc360.com/schema/rsf">

    <rsf:service id="" displayName="测试服务" owner="赵磊" department="MMT开发部"
interfaceClass="com.hc360.rsf.api.bean.data2.PushService"
    class="com.hc360.rsf.api.bean.data2.PushServiceImpl" portalId="测试">
        <rsf:document><![CDATA[ 这是文档 ]]></rsf:document>
    </rsf:service>
</rsf>
```

rsf_client.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rsf xmlns="http://code.hc360.com/schema/rsf"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rsf="http://code.hc360.com/schema/rsf">

    <rsf:client id="pushServiceImpl" displayName="调用测试服务" owner="张三" department="MMT
开发部"
    interfaceClass="com.hc360.rsf.api.bean.data2.PushService" url="127.0.0.1" portalId="
测试">
        <rsf:clientMethod name="getUserInfo"
callback="com.hc360.rsf.api.bean.data2.CallBack_getUserInfo"
async="true"></rsf:clientMethod>
        <rsf:clientMethod name="findData" parameterTypes="java.lang.String"
callback="com.hc360.rsf.api.bean.data2.CallBack_findData"
async="true"></rsf:clientMethod>
        <rsf:clientMethod name="findData" parameterTypes="java.lang.String,int"
callback="com.hc360.rsf.api.bean.data2.CallBack_findData"
async="true"></rsf:clientMethod>
    </rsf:client>
</rsf>
```

服务端

```
package com.hc360.rsf.api.push_async_callback;
import com.hc360.rsf.config.ConfigLoader;

/**
 * 异步通信测试
 * 两节点之间通信测试，Client、Server之间通信测试
 *
```

```
 * 模拟服务提供者 （服务端）
 *
 * @author zhaolei 2012-6-27
 */
public class Server {

    /**
     * 加载rsf_server.xml配置文件
     *
     * 在本地暴露服务
     *
     * 向注册中心注册服务
     *
     * @param args
     */
    public static void main(String[] args) {
        String xmlPath = "classpath:rsf_server.xml";
        ConfigLoader configLoader = new ConfigLoader(xmlPath,Server.class);
        configLoader.start();
    }
}
```

客户端

```
package com.hc360.rsf.api.push_async_callback;

import com.hc360.rsf.api.bean.data2.PushBean;
import com.hc360.rsf.api.bean.data2.PushService;
import com.hc360.rsf.config.ConfigLoader;

/**
 * 异步通信测试
 * 两节点之间通信测试，Client、Server之间通信测试
 *
 * 模拟服务调用者（客户端）
 *
 * @author zhaolei 2012-6-27
 */
public class Client {

    /**
     * 加载rsf_server.xml配置文件
     *
     * 从注册下载服务提供者列表
     *
     * 发起调用
     *
     * @param args
     */
    public static void main(String[] args) {
        String xmlPath = "classpath:rsf_client.xml";
        ConfigLoader configLoader = new ConfigLoader(xmlPath,Client.class);
        configLoader.start();

        PushService
pushService=(PushService)configLoader.getServiceProxyBean("pushServiceImpl");
        for(int i=0;i<1;i++){
```

```
            try{
                PushBean result=pushService.getUserInfo(i+"");
                System.out.println("userService.getUserInfo()调用结束,结果:"+result);

                String result2=pushService.findData(i+"");
                System.out.println("userService.findData()调用结束,结果:"+result2);

                result2=pushService.findData(i+"",100);
                System.out.println("userService.findData()调用结束,结果:"+result2);

            }catch(Exception e){
                e.printStackTrace();
            }
        }
        //ConfigLoader.destroy();
    }
}
```

传输的对象

```
package com.hc360.rsf.api.bean.data2;
import java.io.Serializable;
public class PushBean implements Serializable{
    /***/
    private static final long serialVersionUID = 1L;
    private String name;
    private int age;

    public String toString(){
        return "[name:"+name+",age:"+age+"]";
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

服务接口

```
package com.hc360.rsf.api.bean.data2;

/**
 * 测试接口
 *
 * @author zhaolei 2012-4-25
 */
public interface PushService {
    public PushBean getUserInfo(String userName);

    public String addUser(PushBean user);
```

```
    public String findData(String id);

    public String findData(String id,int age);

    public void getPush(int number);
}
```

服务接口的实现类

```
package com.hc360.rsf.api.bean.data2;

import com.hc360.rsf.config.callback.CallBackHelper;
import com.hc360.rsf.config.callback.PushResult;

/**
 * 测试接口的实现类
 * @author zhaolei 2012-4-25
 */
public class PushServiceImpl implements PushService {

    public PushBean getUserInfo(String userName) {
        PushBean user=new PushBean();
        user.setAge(5);
        user.setName(userName);

        //使用工具关联key,并推送数据
        CallBackHelper.put("key");
        //取IP ,prot
        //List<CallBackWrap> list=CallBackHelper.get("key");

        //向客户端推送数据
        //以下代码可执行多次
        //以下代码可放在其它地方执行
        //服务端发起的推送是阻塞的，可以判断推送是否成功
        PushResult[] rs=CallBackHelper.send("key","getUserInfo-DataDataDataData");
        if(rs!=null){
            for(PushResult obj:rs){
                System.out.println("推送结果:"+obj);
            }
        }
        return user;
    }

    public String addUser(PushBean user){
        return "ok";
    }

    public String findData(String id){
        //使用工具关联key,并推送数据
        CallBackHelper.put("key_findData_1");
        //取IP ,prot
        //List<CallBackWrap> list=CallBackHelper.get("key_findData");

        //向客户端推送数据
        //以下代码可执行多次
        //以下代码可放在其它地方执行
        //服务端发起的推送是阻塞的，可以判断推送是否成功
        PushResult[] rs=CallBackHelper.send("key_findData_1","findData-DataDataData_1");
```

```
        if(rs!=null){
            for(PushResult obj:rs){
                System.out.println("推送结果:"+obj);
            }
        }
        return "123";//客户端发起异步调用时，无法收到这个值，返回值无用
    }

    public String findData(String id,int age){
        //使用工具关联key,并推送数据
        CallBackHelper.put("key_findData_2");
        //取IP ,prot
        //List<CallBackWrap> list=CallBackHelper.get("key_findData");

        //向客户端推送数据
        //以下代码可执行多次
        //以下代码可放在其它地方执行
        //服务端发起的推送是阻塞的，可以判断推送是否成功
        PushResult[] rs=CallBackHelper.send("key_findData_2","findData-DataDataData_2");
        if(rs!=null){
            for(PushResult obj:rs){
                System.out.println("推送结果:"+obj);
            }
        }
        return "123";
    }

    public void getPush(int number){

    }
}
```

回调函数1

```
package com.hc360.rsf.api.bean.data2;
import java.io.Serializable;
import com.hc360.rsf.config.callback.CallBack;

/**
 * 回调函数
 * @author zhaolei 2012-6-8
 */
public class CallBack_getUserInfo implements CallBack {

    public Object call(Serializable data) {
        System.out.println("客户端:收到服务端推送来的数据:"+data+",方法名：getUserInfo");
        return null;
    }
}
```

回调函数2

```
package com.hc360.rsf.api.bean.data2;
import java.io.Serializable;
import com.hc360.rsf.config.callback.CallBack;

/**
 * 回调函数
 * @author zhaolei 2012-6-8
```

```
 */
public class CallBack_findData implements CallBack {
    public Object call(Serializable data) {
        System.out.println("客户端:收到服务端推送来的数据:"+data+",方法名：findData");
        return null;
    }
}
```

运行服务端的 main 方法。

运行客户端的 main 方法。

一切都顺利的话，你应该看到调用成功了。

12. 两点通信（脱离注册中心）

概览表：

维度	IO 模型	配置方式	安全	服务发现	跨语言	注册中心
值	同步调用	XML	非加密通信	2 点通信 无发现能力	只运行于 Java 语言平台	db 注册中心

12.1. 三点通信与两点通信的区别

3 点通信：角色有客户端、服务端、注册中心，客户端通过注册中心发现服务端。
2 点通信：角色有客户端、服务端。客户端事先明确知道服务端的地址(ip:prot)。不依赖注册中心。

12.2. 如何实现两点通信

客户端与服务端通信可以不依赖服务注册中心，三点间的通信变成点对点通信。当客户端明确知道服务端的 IP、端口时，可以直接调用。客户端在配置文件中使用 url 属性指明服务端的位置，这时将没有软负载能力。客户端也服务端也不需要配置服务注册中心了。

修改客户端 RSF 配置文件，client 标签上使用 url 属性，指明服务端的地址(ip:prot)。

<rsf:client>	url	string	点对点直连 URL 的配置。 本值为空，才向注册中心订阅服务列表。 本值不空，将直接向这个地址请求。示例如下： 127.0.0.1：指明 ip,使用默认协议、端口 127.0.0.1:63634：指明 ip,端口，使用默认协议 rsf://127.0.0.1:63634：指明 ip、端口、协议

注：url 只支持一个服务端地址。

13. 使用 java 编码方式配置 RSF

概览表：

维度	IO 模型	配置方式	安全	服务发现	跨语言	注册中心
值	同步调用	Java 编码配置方式	非加密通信	3 点通信 通过注册中心发现服务	只运行于 Java 语言平台	db 注册中心

RSF 支持通过 java 编码方式开发服务端与客户端（不使用 XML 配置文件），但要知道这只是补充方式。常规方式还是通过 xml 文件来配置 RSF 的服务端与客户端。不希望大家广泛的使用 java 编码方式开发服务端与客户端，可以用在测试代码上。

支持 3 种调用方式：同步，异步无返回值 ，异步有返回值，下面只给了同步调用的未作。

13.1. 服务端示例

```
public static void main(String[] args) {
    UserService userService=new UserServiceImpl();

    //协议
    ProtocolConfig protocol=new ProtocolConfig();
    protocol.setPort(63634);
    protocol.setName("rsf");

    //注册中心
    RegistryConfig register=new RegistryConfig();
    register.setHost("register.org.hc360.com");
    register.setPort(63638);

    ServiceConfig<UserService> server=new ServiceConfig<UserService>();
    server.setDisplayName("测试服务");
    server.setDepartment("MMT开发部");
    server.setOwner("赵磊");
    server.setDucment("服务说明:提供测试功能");
    server.setPortalId("测试");
    server.setInterfaceClass(UserService.class);
    server.setRef(userService);
    server.setProtocol(protocol);//协议
    server.setRegistry(register);//注册中心地址
```

```
        server.export();//暴露
        server.registerService();//向服务注册中心  注册本服务
    }
```

13.2. 客户端示例

```
public static void main(String[] args) throws InterruptedException {
        //注册中心
        RegistryConfig register=new RegistryConfig();
        register.setHost("register.org.hc360.com");
        register.setPort(63638);

        ClientConfig<UserService> client=new ClientConfig<UserService>();
        client.setCharset("GBK");
        client.setDisplayName("测试客户端");
        client.setDepartment("MMT开发部");
        client.setOwner("赵磊");
        //client.setUrl("rsf://127.0.0.1:63634");//不直连
        client.setRegistry(register);//走注册中心
        client.setInterfaceClass(UserService.class);
        UserService userService=client.getInstance();//重点,取得接口的代理

        long t7=System.nanoTime();
        for(int i=0;i<count;i++){
            try{
                long t1=System.nanoTime();
                UserBean result=userService.getUserInfo(i+"");
                long t2=System.nanoTime();
                System.out.println("调用结束,用时:"+df.format((t2-t1)/Constants.TIME_C)+" "+result);
            }catch(Exception e){
                e.printStackTrace();
            }finally{
                if(t>0){
                    Thread.currentThread().sleep(t);
                }
            }
        }
        long t8=System.nanoTime();
```

```
        System.out.println("总用时:"+df.format((t8-t7)/Constants.TIME_C)+",总计:"+count+"次,平
均:"+df.format((t8-t7)/Constants.TIME_C/count));
        AbstractConfig.destroy();
}
```

14. RSF 日志配置

RSF 框架采用 SLF4J 做为日志门面，日志实现可以根据属主项目环境要求选用 log4j、JDK Log、Apache commons-log 等。

14.1. 日志级别

1、Debug 级：每发出（收到）一次请求，创建（闭关）一个连接，都记录详细的日志。
2、Info：级：比 Debug 级粗一些，记录主要调用事件的日志。
3、Warn 级：记录可以恢复的错误，不需要人工处理。
4、Error 级：记录需要人工介入的错误。

14.2. 日志配置建议

下面以 Log4j 为例，说明日志配置建议，可便于你跟踪调试程序。

14.2.1. 生产环境下建议采用如下配置

```
log4j.logger.com.hc360.rsf =warn
```

可以查看到 RSF 运行时的错误信息。

14.2.2. 开发环境下建议采用如下配置

```
log4j.logger.com.hc360.rsf =info
```

可查看到 RSF 运行的主要日志信息。
可得到类似下图的日志信息：

```
1  RSF Server 启动完成,工作在127.0.0.1:63634.
2  RSF ServerConfig 发布服务:com.hc360.rsf.config.p2p.push.PushService,显示名称：测试服务
3  关闭重复连接 Close mina channel (0x00000005: nio socket, client, /192.168.34.154:56569 => /192.168.34.154:63634)
4
5  调用完成,耗时:4.421280000ms,/192.168.34.154:56492->/192.168.34.154:63634
6  调用完成,耗时:6.711240000ms,/192.168.34.154:56492->/192.168.34.154:63634
7
8  执行退出程序,释放资源。
9  Close mina channel (0x00000003: nio socket, client, /192.168.34.154:56492 => /192.168.34.154:63634)
10 关闭 Mina Client
11 关闭 Mina Server,地址:/127.0.0.1:63634
```

RSF1.1.0 版本以后的调用耗时信息更丰富了，包含：数据包大小、总耗时、业务耗时、网络耗时、IP，日志输出示例如下：

```
调用完成,数据包:75Byte,总耗时:6.258ms,业务耗时:4.041ms,网络耗时:2.217ms,目标:com.hc360.rsf.config.p2p.requestresponse.UserService
```

```
getUserInfo(),/192.168.34.154:57272->/192.168.34.154:63634
```

上面日志显的“数据包:75Byte” 是纯业务数据大小，不包含 16Byte 的 rsf 通信协议头。75+16=91Byte 是总的网络传输量。

14.2.3. 调试环境下建议采用如下配置

```
log4j.logger.com.hc360.rsf =debug
```

可查看到 RSF 运行的全部日志信息，信息量大。

15. 服务端接口与客户端接口的兼容性

思考：服务调用者（客户端）和服务提供者（服务端）共同依赖同一个 Java 接口文件，这个接口发生变化后会不会导致所以使用这个接口的应用重新发布？答案是不会。

请看以下兼容性测试结果，接口可以添加新方法，传输对象可以添加新成员变量，兼容性良好。

15.1. 接口兼容性

两端接口的包名：必须相同
两端接口的类名：必须相同
两端接口的方法名：必须相同
两端接口的方法的参数：必须相同
两端接口方法的数量：可以不同，服务提供方在服务上线后可单方面添加接口中的方法，对原有客户端无影响。

15.2. 传输对象的兼容性

两端对象：必须实现 Serializable 接口
两端对象的包名：必须相同
两端对象的类名：必须相同
两端对象的 serialVersionUID：java 序列化要求相同，Hessian 序列化要求可以不同，RSF 默认使用 Hessian 序列化。
两端对象的成员属性名：可以不同，只有共有的属性才可正常传递值。
两端对象的成员属性数据类型：必须相同
两端对象的成员属性的数量：可以不同，只有共有的属性才可正常传递值。

15.3. 在接口中声明异常

服务端
服务接口中的方法关于声明异常注意以下几点。
1、可不声明抛出异常。（这样把旧接口改造成 RSF 服务时不会强制你抛出异常）
2、可声明抛出 Exception
3、不可以声明抛出其它类型的异常，如自定义异常。因为客户端无法 catch 到你声明的自定义异常

客户端
客户端调用服务端，假设服务端是不可信的，假设发生异常是常态的。
不论服务接口的方法上是否声明抛出了异常，抛出了什么类型的异常，客户端都应 try catch Exception,也只应 try

catch Exception 这一种异常类型，这样是最安全的。RSF 可能会抛出 TimeoutException,或其它类型你不知道的异常。

16. RSF 的线程模型

16.1. 线程池

了解 RSF 的线程模型，非常重要，由其是在处理高并发的业务时。RSF 主要是在服务端使用了线程池，当服务端接收到一个请求时会放在一个独立的线程中处理。

rsf 协议与 thrift 协议的线程池是独立的，如果服务端同时使用这两个协议，会同时存在两个线程池。互不影响。

	服务端线程池	客户端线程池	业务线程
rsf 协议	有独立的线程池，可配置参数。	有独立的线程池，客户端在接收服务端数据时，会使用线程池。	使用容器(如 tomcat)的线程池
thrift 协议	有独立的线程池，可配置参数。	无独立的线程池。	使用容器(如 tomcat)的线程池

16.2. RSF 各种线程池默认值

RSF 1.3.x 的线程池默认参数

类型	corePoolSize	queueSize	maximumPoolSize	keepalive	rsf 协议默认线程池
缓存线程池 Cached	0	0	不限	60*1000ms	
固定线程池 Fixed	200	300	200	0 ms	
混合线程池 Mixed	100	100	400	60*1000ms	是

RSF 2.0.0 的线程池默认参数

类型	corePoolSize	queueSize	maximumPoolSize	keepalive	rsf 协议默认线程池	thrift 协议默认线程池
缓存线程池 Cached	0	0	不限	60*1000ms		
固定线程池 Fixed	200	300	200	0 ms		
混合线程池 Mixed	0	0	400	60*1000ms	是	是

RSF2.0.0 调整了线程池的默认参数，上表红色的值是变化后的，新的参数更合理，更稳键。

如果你不满意默认参数，可以通过配置文件来自定义线程池的参数。

注意：以上 3 个线程，没有本质的区别，只是初始化参数不同。如何 3 个线程池都使用相同的参数，他们的表现将是一样的。

16.3. 如何配置线程池

服务端线程池：参数可以通过<rsf: protocol>标签配置。

客户端线程池：保有 RSF 的 Java 客户端有线程池，不可配置。

16.4. 达到上限的处理

服务端：当请求达到线程池上限后，请求被拒绝，抛出 RejectedExecutionException 异常，这是对服务端的保护。

客户端：无。

17. RSF 的健壮性

17.1. 失败转移

RSF 框架采用长连接，做失败转移的难度大于短连接，短连接可以在创建连接失败后转移，但长连接不行，所以 RSF 失败转移是在请求发起前做的。如果因网络问题发出调用失败，则换用其它节点再调用，再不成功再换，直到都失败或一个成功。如果调用发出后发生了失败（业务异常、超时），将不再重试。

总结：网络失败会重试，业务失败不重试(试了也白试)。

当服务端部署了多个节点时，客户端在请求发出前，就可以知道哪些服务端是可用的，哪些服务端是不可使用的。只选择可用的服务端发起请求（软负载均衡）。当服务端又恢复后，会再次被请求。

当服务提供者(服务端)因某种原因不能提供服务了，客户端是如何保证不再向这个出问题的服务提供者(服务端)发出请求的？

请看以下表格：

前提条件	结论
两点之间正常通信时，正常退出服务端	客户端可以立即感知
两点之间正常通信时，kill 服务端进程	客户端可以立即感知
两点之间正常通信时，禁用服务端、客户端网卡	客户端可以立即感知
两点之间正常通信时，拔断网线	客户端**不可以感知**

通过以上表格，可以知道，客户端判定服务端连接是否可用，大多时候是可靠的，除了“拔断网线”这种情况。针对“拔断网线”这种情况我们将采用心跳来监测。请看下一节。

以上说的都是在客户端的真实的数据发向服务端之前做的工作，那客户端的真实的数据已经发向服务端之后，并失败了，再做失败转移是否可以呢？答案是否定的。请看以下几种请情：

1、 发生网络超时失败：假如服务端操作数据库耗时 5 秒，RSF 网络请求超时默认是 3 秒，客户端在发出请求 3 秒后收到超时结果，但 5 秒后服务端数据入库成功。做为客户端的失败转移，如果再次发出重试的请求，就是导致数据库数据有两份。所以失败转移要在客户端的真实的数据发向服务端之前做。

2、 服务异常失败：假如服务端数据库表字段的必填项为空导致失败，客户端知道失败后，再做失败转移再试一次，也白试。

3、 只有可以 100%判定是网络原因，数据未能从客户端到达服务端，才可以安全的做失败转移。所以失败转移在客户端的真实的数据发向服务端之前做。

如果因服务器连接不上，请求未能发出，客户端将自动换另一台负载均衡的服务重试。

如果请求已经发出并到达了服务端，发生失败（超时，服务端业务异常）后将不会重试，向上抛给业务层。请求只会被发出一次。

17.2. 心跳

客户端、服务端、服务注册中心三者间有心跳，默认每 3 秒心跳一次，两次心跳失败后（6 秒）认定连接不可

用。当两点间实然网线断开后，最多 6 秒可以感知。如果在这 6 秒时间内有请求，一定会失败。

18. RSF 的性能

18.1. 性能测试

结论：RSF 一次通信平均耗时约 2 毫秒。

服务端机器配置:CPU:4 核心至强,内存:8G.

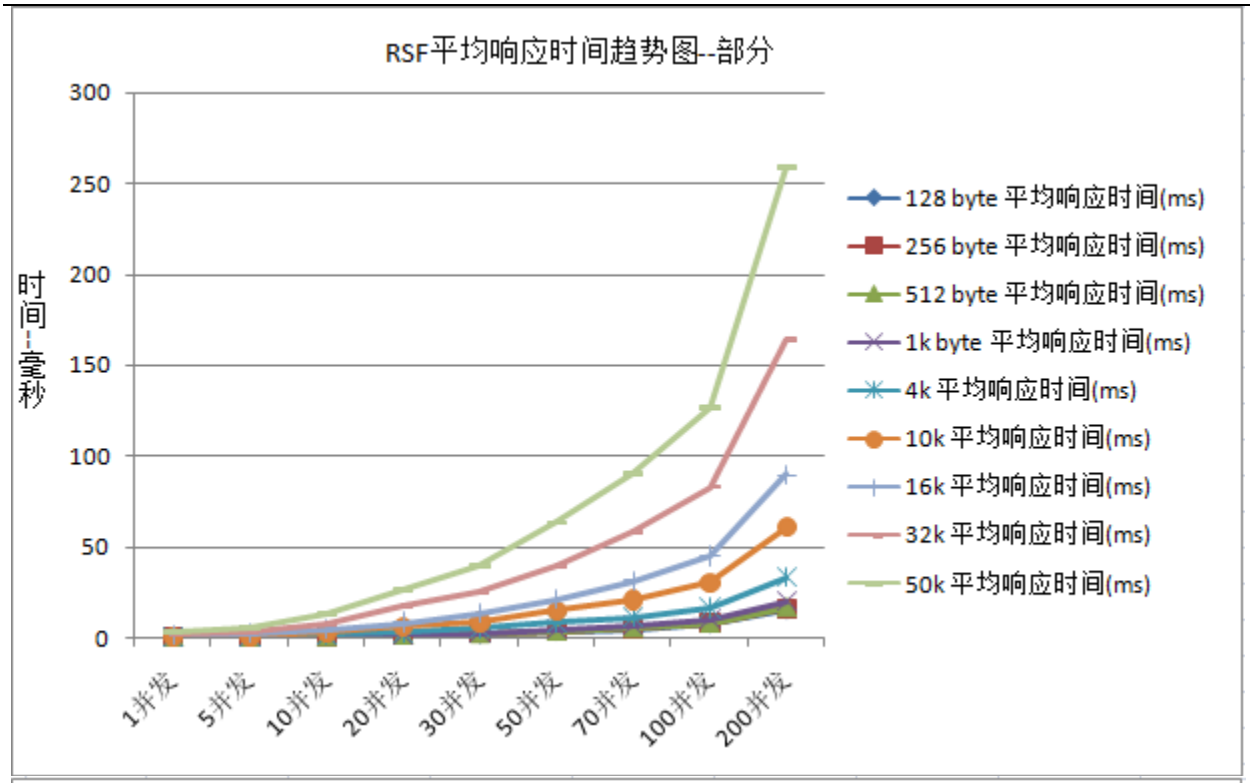
客户端机器配置: CPU:2 棵双核 AMD 皓龙 2212 频率 2g,内存:8G.

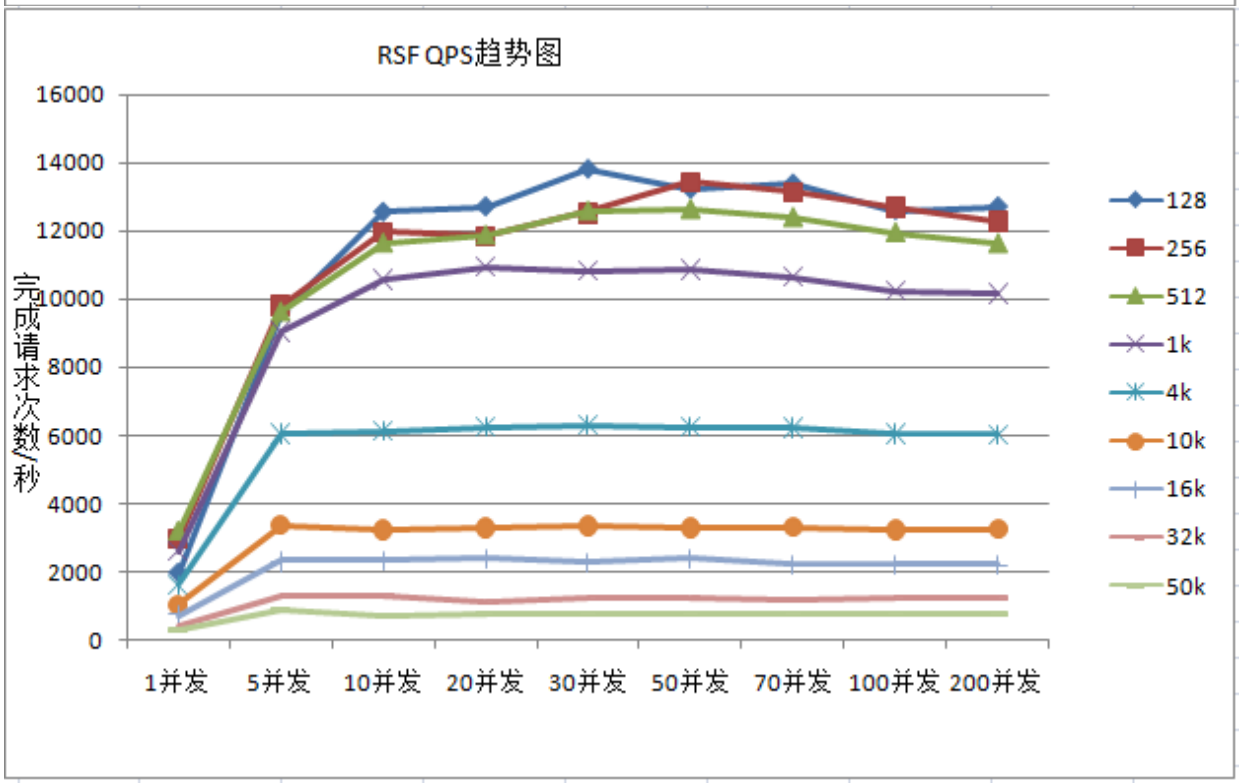
网络环境:1000Mb 局域网.

测试方法: 模拟 N 个并发,每并发请求 10000 次,每次发送与接收的数据大小见下表中的”对象大小”。

测试结果：

数据大	测试项目	1并发	5并发	10并发	20并发	30并发	50并发	70并发	100并发	200并发
128	128 byte 平均响应时间(ms)	0.5	0.52	0.76	1.63	2.17	3.77	5.16	7.92	15.7
	128 byte QPS	1966	9607	12552	12710	13812	13223	13384	12606	12713
	128 byte 网络吞吐量(MB/s)	0.24	1.17	1.6	1.49	1.68	1.61	1.65	1.53	1.55
256	256 byte 平均响应时间(ms)	0.33	0.5	0.83	1.68	2.59	3.71	5.3	7.86	16.2
	256 byte QPS	2958	9829	11954	11840	12532	13439	13154	12705	12290
	256 byte 网络吞吐量(MB/s)	0.7	2.39	2.91	2.89	2.81	3.28	3.2	3.1	3
512	512 byte 平均响应时间(ms)	0.31	0.51	0.85	1.67	2.37	3.94	5.6	8.34	17.1
	512 byte QPS	3186	9637	11661	11891	12600	12650	12413	11971	11646
	512 byte 网络吞吐量(MB/s)	1.55	4.7	5.69	5.8	6.15	6.17	6.06	5.84	5.6
1k	1k byte 平均响应时间(ms)	0.38	0.55	0.94	1.82	2.77	4.61	6.56	9.76	19.7
	1k byte QPS	2603	9026	10553	10918	10795	10836	10649	10230	10130
	1k byte 网络吞吐量(MB/s)	2.54	8.81	10.3	10.66	10.54	10.58	10.39	9.99	9.89
4k	4k 平均响应时间(ms)	0.62	0.82	1.62	3.19	4.77	8.01	11.26	16.54	33.2
	4k QPS	1618	6029	6118	6251	6282	6231	6210	6032	6018
	4k 网络吞吐量(MB/s)	6.32	23.55	23.89	24.42	24.54	24.34	24.25	23.56	23.5
10k	10k 平均响应时间(ms)	0.96	1.47	3.08	6.05	8.91	15.11	21.13	30.78	61
	10k QPS	1048	3374	3232	3298	3363	3305	3311	3247	3276
	10k 网络吞吐量(MB/s)	10.07	32.95	31.56	32.2	32.84	32.28	32.33	31.71	31.99
16k	16k 平均响应时间(ms)	1.36	2.17	4.27	8.38	13.1	21.09	31.45	45.18	90
	16k QPS	733	2301	2332	2381	2288	2369	2224	2212	2206
	16k 网络吞吐量(MB/s)	11.45	35.95	36.44	37.21	35.7	37.02	34.7	34.57	34.4
32k	32k 平均响应时间(ms)	2.34	3.91	7.94	17.68	25.05	40.09	58.7	83.11	164
	32k QPS	427	1276	1256	1130	1197	1246	1191	1202	1216
	32k 网络吞吐量(MB/s)	13.35	39.87	39.27	35.33	37.4	38.95	37.23	37.59	38
50k	50k 平均响应时间(ms)	3.41	5.66	13.66	26.59	39.45	63.56	90.5	127	259
	50k QPS	292	881	731	751	760	785	773	786	771
	50k 网络吞吐量(MB/s)	14.29	43.05	35.71	36.71	37.12	38.35	37.7	38.4	37.6





18.2. 与 WebService 对比

原来业务使用 WebService 通信，后改为 RSF 通信，是同一个业务点--商务中心的“我报名的采购会”业务，页面准备齐全数据需要 5 次通信。

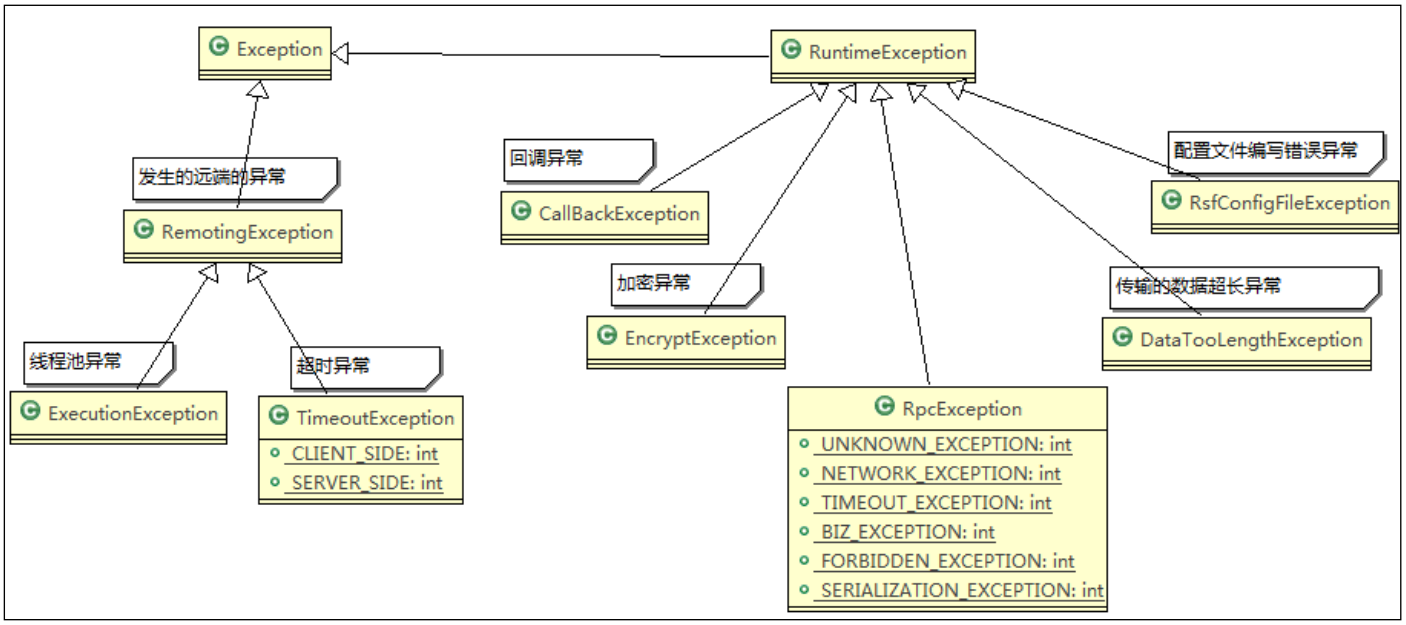
原：WebService 一次通信耗时 15.2ms 数据量小于 3K。15*5=75ms

现：RSF 一次通信耗时 2.2ms　数据量小于 3K。2.2*5=11ms

结论：节约了 64ms

19. RSF 常见异常

19.1. 异常体系图



19.2. 自动检测重复的 RSF jar 包

如果你的项目的 class path 中有多个 RSF 框架的 Jar 包存在，日志会输出以下 ERROR 信息。请关注日志，以免因项目中同时使用多个版本的 RSF 的 Jar 包，新版的功能比旧版多，你认为使用的是新版本，实际加载的是旧版本，而导致 Bug 很难查找。

```
2012-06-27 15:16:23 [ERROR]-[com.hc360.rsf.common.Version] RSF框架jar包重复。发现重复
        RSF框架jar包重复。发现重复的类 com/hc360/rsf/common/Version.class在2个jar包中
        RSF框架jar包重复。发现重复的类 com/hc360/rsf/common/Version.class在2个jar包中
        RSF框架jar包重复。发现重复的类 com/hc360/rsf/common/Version.class在2个jar包中
        RSF框架jar包重复。发现重复的类 com/hc360/rsf/common/Version.class在2个jar包中
```

19.3. 无法连接到远端主机异常

当目标主机 IP、端口错误或目标主机未提供服务时，将产生以下异常。最常见的情况是无法找到服务注册中心。

```
        at com.hc360.rsf.config.p2p.both.Server.main(Server.java:27)
2012-07-24 15:53:21 [ERROR]-[com.hc360.rsf.remoting.transport.mina.MinaClient] 无法连接到远端主机--192.168.119.216:63638,URL=
Caused by: java.net.ConnectException: Connection refused: no further information
        at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)
        at sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:527)
        at org.apache.mina.transport.socket.nio.NioSocketConnector.finishConnect(NioSocketConnector.java:224)
        at org.apache.mina.transport.socket.nio.NioSocketConnector.finishConnect(NioSocketConnector.java:46)
        at org.apache.mina.core.polling.AbstractPollingIoConnector.processConnections(AbstractPollingIoConnector.java:439)
        at org.apache.mina.core.polling.AbstractPollingIoConnector.access$700(AbstractPollingIoConnector.java:64)
        at org.apache.mina.core.polling.AbstractPollingIoConnector$Connector.run(AbstractPollingIoConnector.java:508)
        at org.apache.mina.util.NamePreservingRunnable.run(NamePreservingRunnable.java:64)
        at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:651)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:676)
        at java.lang.Thread.run(Thread.java:595)
```

19.4. 在客户端显示服务端异常信息

客户端调用服务端接口，但服务端发生了某种异常，客户端如何知道？请看如下图片，RSF 把服务端异常带回到客户端，供客户端开发人员查看，帮你划清问题的界线。

```
2012-07-24 15:51:31 [DEBUG]-[com.hc360.rsf.remoting.transport.mina.MinaHandlerDelegate] 消息到达事件，side=client
com.hc360.rsf.rpc.RpcException: 调用com.hc360.rsf.config.p2p.requestresponse.UserService getUserInfo()时异常
        at com.hc360.rsf.rpc.protocol.RsfInvokerClientP2p.invoke(RsfInvokerClientP2p.java:64)
        at com.hc360.rsf.rpc.proxy.jdk.JdkInvocationHandler.invoke(JdkInvocationHandler.java:78)
        at $Proxy0.getUserInfo(Unknown Source)
        at com.hc360.rsf.config.p2p.both.Client.main(Client.java:36)
Caused by: com.hc360.rsf.remoting.RemotingException: 此异常信息是服务端异常，显示给调用者查看：(0x00000002: nio sock
java.lang.NullPointerException
        at com.hc360.rsf.config.p2p.requestresponse.UserServiceImpl.getUserInfo(UserServiceImpl.java:13)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:592)
        at com.hc360.rsf.rpc.protocol.RsfInvokerServer.invoke(RsfInvokerServer.java:63)
        at com.hc360.rsf.remoting.transport.mina.MinaHandlerDelegate.received(MinaHandlerDelegate.java:174)
        at com.hc360.rsf.remoting.transport.dispather.ChannelEventRunnable.run(ChannelEventRunnable.java:76)
        at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:651)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:676)
        at java.lang.Thread.run(Thread.java:595)
        ---------服务端的异常信息结束----------
        at com.hc360.rsf.remoting.exchange.support.DefaultFuture.returnFromResponse(DefaultFuture.java:184)
```

19.5. 未序列化异常

通过网络传输的对象，没有实现 Serializable 接口。这个经常被忘记，也是常见异常。也可能是服务端的返回结果对象未序列化，日志都会反映出来。

```
        at com.hc360.rsf.config.p2p.both.Client.main(Client.java:42)
Caused by: com.hc360.rsf.rpc.RpcException: 序列化异常,请检查被传输对象和对象的成员是否实现了Serializable接口:
        at com.hc360.rsf.rpc.protocol.codec.RsfCodec.encodeRequestData(RsfCodec.java:71)
        at com.hc360.rsf.rpc.protocol.codec.ExchangeCodec.encodeRequest(ExchangeCodec.java:279)
        at com.hc360.rsf.rpc.protocol.codec.ExchangeCodec.encode(ExchangeCodec.java:92)
        at com.hc360.rsf.rpc.protocol.codec.mina.MinaCodecAdapter$InternalEncoder.encode(MinaCodecAdapter.java:91)
        at org.apache.mina.filter.codec.ProtocolCodecFilter.filterWrite(ProtocolCodecFilter.java:322)
        ... 13 more
```

19.6. 请求超时异常

客户端向服务发起请求，服务端未在 3000ms 内返回结果，将产生以下异常。超时时间可以配置。

```
com.hc360.rsf.rpc.RpcException: 调用com.hc360.rsf.config.p2p.requestresponse.UserService addUser()时异常
        at com.hc360.rsf.rpc.protocol.RsfInvokerClientP2p.invoke(RsfInvokerClientP2p.java:64)
        at com.hc360.rsf.rpc.proxy.jdk.JdkInvocationHandler.invoke(JdkInvocationHandler.java:78)
        at $Proxy0.addUser(Unknown Source)
        at com.hc360.rsf.config.p2p.both.Client.main(Client.java:12)
Caused by: com.hc360.rsf.remoting.TimeoutException: 调用发起端等待响应超时,timeout=3000 ms
        at com.hc360.rsf.remoting.exchange.support.DefaultFuture.get(DefaultFuture.java:150)
        at com.hc360.rsf.remoting.exchange.support.DefaultFuture.get(DefaultFuture.java:111)
        at com.hc360.rsf.remoting.transport.mina.MinaChannel.request(MinaChannel.java:128)
        at com.hc360.rsf.rpc.protocol.RsfInvokerClientP2p.invoke(RsfInvokerClientP2p.java:61)
        ... 3 more
```

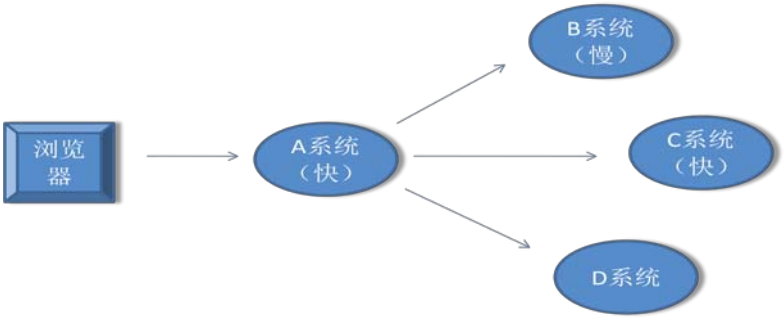
19.7. XML 配置文件编写不正确

RSF 的 xml 配置文件检查很全面，当编写不正确时，如没有必须的属性等等，在 RSF 加载 XML 文件时会报类似下图的异常信息。

```
Exception in thread "main" com.hc360.rsf.config.RsfConfigFileException:
RSF配置文件编写不正确,rsf:service元素,不能找到服务接口：com.hc360.rsf.config.p2p.requestresponse.UserService
        at com.hc360.rsf.config.ConfigLoader.paresServiceConfig(ConfigLoader.java:552)
        at com.hc360.rsf.config.ConfigLoader.paresElement(ConfigLoader.java:345)
        at com.hc360.rsf.config.ConfigLoader.pares(ConfigLoader.java:326)
        at com.hc360.rsf.config.ConfigLoader.loadXML(ConfigLoader.java:288)
        at com.hc360.rsf.config.ConfigLoader.load(ConfigLoader.java:87)
        at com.hc360.rsf.config.ConfigLoader.<init>(ConfigLoader.java:52)
        at com.RR4xmlTest.main(RR4xmlTest.java:16)
```

20. 超时时间

20.1. 要求超时时间尽量短的场景



同步调用：

浏览器请求 A 系统某个业务，A 系统需要调用一个外部 B 系统取数据，但 B 系统很慢。假如支持 A 系统运行的 Tomcat 线程池大小是 200，由于 B 系统慢响应（或不响应）会导致 A 系统的 Tomcat 的 200 个线程全被挂住（等待 B 系统响应）。这时 A 系统的 CPU、内存都有很多空闲，但已无法接受第 201 个用户的请求。这就是系统间同步调用的特点，如何你使用 WebService,RMI, HttpURLConnection 发启的调用都有这样的特点。

跟本的解决方法是让 B 系统变快，但 B 系统是另一个部门或另一个公司开发的，你无力改变。

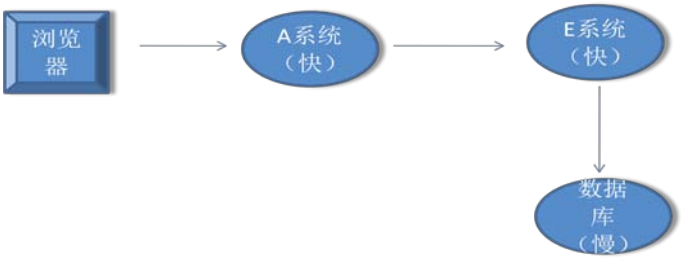
RSF 有超时功能的同步调用：

RSF 的调用，本质是异步的，通过异步模拟同步调用，使它具有了设定超时时间的能力。客户端向服务端发起调用时，客户端的业务线程（主线程）通知 Java Nio 的一个线程发出网络通信实现调用，客户端的业务线程（主线程）处于“等待”状态，等待服务端有返回结果后执行后续逻辑。如等待 3 秒后无返回结果，进入超时逻辑。

RSF 为你提供了另一种选择—自动超时。达到超时时间后，自动断开对 B 系统（慢系统）的请求，使 A 系统可以接受第 201 个用户的请求，很可能这第 201 个用户并不需要 B 系统，而是需要 C 系统（快）。

这种场景，要求设置更短的超时时间，可以早一点解放 A 系统。

20.2. 要求超时时间尽量长的场景



A 系统调用 E 系统向数据库（慢）写数据，由于数据库响应很慢（但还有响应），写一条数据要花 4 秒种才能完成，但这时 A 系统的这次请求因 3 秒超时而失败，但又过 1 秒钟后，数据库操作成功了。

A 系统再次重试后，会导致主键冲突或多写了一条数据。这个场景，要求设置更长的超时时间，来保证数据安全。

20.3. 由你来抉择超时时间

前面两个场景，对超时时间的设置要求是矛盾的。要理解原理，由你来跟根据业务特点抉择多长的超时时间合适。可以在配置文件中设置超时时间，可独立控制到方法级。

不必纠结，用默认值吧

不要被前面的抉择吓坏，默认值是 3 秒，可以适用于 99%的系统场景，用默认值吧。

21. RSF 服务端动态端口

21.1. 问题

这是在 RSF1.2 版中新加入的功能，为了解决一台物理服务器同时运行多个 RSF 服务端时端口冲突的问题。

在部署 web 应用时，我们把“商务中心”项目打成一个 war 包，部署在一台物理服务器中。服务器上一个 WebSphere 中同时运行着 2 个“商务中心”逻辑节点，也就是一个 war 包文件在一台物理服务器上部署 2 次 “商务中心”逻辑节点都需要使用操作系统的某个端口(默认是 63634)，这时就发生了端口冲突。

21.2. 动态端口

RSF 1.2 版新加了服务端动态端口功能，默认值是 63634-63600。如果 63634 被占用，会按降序尝试其它端口，最终选择一个可用的端口。

21.3. 服务端动态端口适用场景

适用于三点间通信场景下(客户端、服务端、服务注册中心)，我们 99%的应用符合这个场景。

客户端是通过“服务名”向“服务注册中心”查询而找到服务提供者的。只要“服务名”唯一不变就可以找到服务提供者，并完成调用，服务端端口变化没有影响。

不适用于对点对点通信场景(客户端、服务端)。点对点通信要求客户端明确知道服务端的 IP、端口，服务端端口动态是不可以变化的。

21.4. 配置方法

XML 配置文件标签：

```
<rsf:protocol  port="" ></rsf:protocol>
```

使用 RSF1.2 及以上版本，不写<rsf:protocol/>标签或不写属性 port=""，RSF 都将使用默认值(63634-63600)，就已具备服务端动态端口的能力。

配置<rsf:protocol port="63634-63600" ></rsf:protocol>,服务端启动时，如果第一个端口被占用，会按顺序尝试其它端口，最终选择一个可用的端口。

支持 port="63634" 设置单一端口，这是对旧 API 的兼容，无动态端口能力。

支持 port="63634-63600" 设置一段端口,两端包含，降序

支持 port="63600-63634" 设置一段端口,两端包含，升序

支持 port="63634,63634-63600,63631" 混合设置，会保证顺序，但不会排除重复端口

22. RSF 其它高级特性

22.1. Mock 模拟服务端

当服务调用者（客户端）在开发阶段，需要调用服务提供者（服务端）时，但服务提供者（服务端）还没有开发完成，无法与你联调。这时客户端可以通过 Mock 在本地模拟远程服务接口的实现，返回模拟值，方便独立开发。

客户端业务开发人员自己实现服务接口，并修改 rsf.xml 配置文件指明即可。

```
<rsf:client  其它属性略...     mock="服务接口的本地实现类">
```

注意 Mock 实现类，必须有默认构造方法。

22.2. 回声测试

使用 com.hc360.rsf.rpc.EchoService 接口，可以测试客户端与服务端之间的网络通信是否正常。这是接口级别的测试。任何一个远程服务接口的本地代理，都可以被强制转换成 com.hc360.rsf.rpc. EchoService 类型，就可以使用$echo 方法测试了。服务端会原样返回你发送的字条符串。

示例：

```
UserService userService= (UserService)
configLoader.getServiceProxyBean("clientUserServiceImpl");//配置文件中的id
EchoService echo=(EchoService)userService;// 强制转换类型
Object rs=echo.$echo("回声测试");
```

1.3 新特性：EchoService 接口里添加 $echoInterface 方法，用于判断服务端是否存在指定的接口，如存在返回 true,反之，false.

22.3. 依赖 RSF 的业务系统的启动顺序

各个业务系统使用 RSF 框架后，可能即充当服务提供者提供服务，又充当服务调用者调用服务，各个系统可能出现三角依赖关系，这时候谁应该先启动呢？答案是**谁先启动都可以**。服务提供者启动后会立即向注册中心发布提供的服务，服务调用者启动时什么也不做，当要发出第一次请求时，才向注册中心下载服务提供者列表，完成调用。依靠这种延后的策略来解决三角依赖。

22.4. RSF 不能做什么

1、RSF 不可以使用方法上的实参的引用作为返回值

客户端使用接口上的方法上的实参的引用作为返回值，是不行的，你的目的将无法达到。因为 RSF 无法跨 JVM

修改实参引用。

2、RSF 不适合传输大文件

由于 RSF 的数据是“整块”传输的，所以想传输大文件时，对内存是一个严峻的考验。

RSF 在服务端做了限制，当接收到的数据大于 8MB 时会抛出 DataTooLengthExceptione，当然可以在 RSF 服务端的配置文件中修改这个值。

RSF 设计之初是为“方法调用”而设计的，适合高并、小数据量的调用场景。不是为了传输大文件设计的，请用户有一个正确的认识。

你自己只要稍加变通，RSF 就可以传输大文件了。比如要传输一个 100M 文件，你可以把它分成 100 份，每份 1M，使用 RSF 传输 100 次，就可以了。只不过分割与拼合的工作要由你自己来实现。

22.5. Telnet 监视工具

从 RSF 1.3.0 版本开始，支持通过 telnet 连接到 rsf 服务端，查看 rsf 信息。

登录：输入命令 `telnet 127.0.0.1 63634` （请根据实际情况修改 IP 地址和端口）。

telne 命令列表

telnet 命令	功能说明	支持此命令的 RSF 版本
help	查看帮助信息	1.3.0 以上（含）
version	查看 RSF 版本	1.3.0 以上（含）
list	查看服务列表	1.3.0 以上（含）
threadpool	查看线程池信息	1.3.0 以上（含）
jvm	查看 jvm 信息，如内存信息	1.3.0 以上（含）
stat	查看统计信息，如网络吞吐量	1.3.0 以上（含）
set charset=GBK 或 UTF-8	设置编码， 用于解决 Telnet 命令的回显结果是乱码问题	1.3.0 以上（含）
uptime	执行 Linux 命令 uptime 不支持 window 操作系统	2.1.1 以上（含）
free	执行 Linux 命令 free –m 不支持 window 操作系统	2.1.1 以上（含）
netstat	执行 Linux 命令 netstat -ant | grep 服务端的端口 不支持 window 操作系统	2.1.1 以上（含）
向上键	历史命令—上一条命令	1.1.0 以上（含）
向下键	历史命令—下一条命令	1.1.0 以上（含）
Ctrl+c	退出	1.1.0 以上（含）

22.6. 通过注册中心查看使用的 RSF 版本号

登录 HRC 注册中心，可以查看全部业务系统使用的 RSF 的版本号。当出现错误并找不到原因时关注版本号也很重要，可以这个问题在新版本已解决，但你还在使用旧版本。

RSF 是如何读取版本号的？

1. 首先查找 Jar 包中的 META-INF\MANIFEST.MF 规范中的版本号，每次 RSF 发布新版本都会相应修改 Jar 包中的 MANIFEST.MF 文件中的内容，示例：Implementation-Version: 1.1.0 。
2. 如果 MANIFEST.MF 规范中没有版本号,则查找 RSF 的 jar 包文件名中包含的版本号，如 RSF-1.0.12.jar。
3. 如果 Jar 包文件名中不包含版本号，则使用 RSF 中 Version 类中默认值。

22.7. 通过注册中心来查看客户端的信息

- 目的：

用于 rsf 日常维护而开发的功能，查看客户端可用服务提供者列表来确定 rsf 是否出问题，同时可以用于分组功能的验证。

- 操作流程：
  1. 通过 http://register.org.hc360.com/register/login.htm 登录到注册中心；
  2. 点击左侧列表中的“查看客户端信息”展示所有连接到注册中心的所有客户端；

  3. 点击列表中的“查看客户端信息”。

- 说明：
  1. 由于重启，网络不稳等原因，经常会导致操作流程 2 中的客户端列表出现大量的无效连接，请按照提示，尽量查看靠前的数据；
  2. 注册中心会定时清理前一天的客户端信息，所以只能显示当天的客户端信息。

22.8. 通过注册中心来修改服务提供者的权重

- 目的：

用于人为的干预服务提供者的选择，可以调节客户端调用服务的分布。

- 操作界面：

- 说明：
  1. 权重的取值范围是：0-100；
  2. 修改后马上生效，可以在“查看客户端信息”的功能里查看验证。

22.9. 通过 groupName 属性来区分统一接口不同实现

- 出现的原因：
  由于存在同一个接口在不同的系统上有不同的实现的现象，原来的 rsf 是区分不了的，所以有可能会调到错误的实现上，因此 groupName 属性出现了。
- 操作方式：
  针对同一系统不同实现的情况，只需做两件事：
  1. 服务端：在想要区分的<rsf:service>添加 groupName 属性，并赋不同的值；
  2. 客户端：在<rsf:client>添加 groupName 属性，并赋予想要调用服务的 groupName
- 说明：
  1. 如果服务端添加了 groupName，但客户端没有，则调不通；
  2. 如果客户端添加了 groupName，但服务端没有，则调不通；
  3. 如果服务端与客户端都添加了 groupName，但是客户端的 groupName 与服务端的 groupName 对不上，则调不通；
  4. 如果服务端与客户端都添加了 groupName，且客户端的 groupName 与某个服务端的 groupName 对应，则能调通；
  5. 如果服务端与客户端都没有添加 groupName，则能调通，但有可能调到错误的实现上。

22.10. 采集服务调用情况

- 监控中心的域名和端口：
  rsfmonitor.org.hc360.com:63639
- 采集频度：
  两种方式：
  1. 定时汇报：1 分钟汇报一次
  2. 超过阀值汇报：超过 10000 次调用汇报一次
- 是否需要开发者配置：

不需要开发者做任何配置，一旦客户端系统发起了调用，采集任务自动开启，并开始缓存服务调用情况。

23. RSF 的工具类

RSF 提供了一些静态的工具方法，可以让你在业务代码中调用，用于控制 RSF 虽在。

23.1. ConfigLoader 启动关闭 RSF

方法	功能说明
ConfigLoader(String filePath)	构造方法，加载 RSF 配置文件
ConfigLoader(String[] configFilePaths)	构造方法，加载 RSF 配置文件
ConfigLoader(InputStream input)	构造方法,通过流加载 RSF 配置文件
destroy()	关闭 RSF，释放 RSF 所有打开的资源（线程、端口）。
start()	启动 RSF
Object getServiceProxyBean(String beanId)	按 ID 取得"远程服务接口"的"本地代理"
Object[] getServiceProxyBean(Class clazz)	按类型取得"远程服务接口"的"本地代理"
<T> T getServiceProxyBeanT(Class<T> clazz)	按类型取得"远程服务接口"的"本地代理"

23.2. AbstractConfig

方法	功能说明
destroy()	关闭 RSF，释放 RSF 所有打开的资源。（淘汰，但保留）。

23.3. RsfListener 在 Web 项目中启动 RSF 的监听器。

方法	功能说明
ConfigLoader getConfigLoader()	取得 ConfigLoader
setConfigLoader(ConfigLoader configLoader)	设置 ConfigLoader

23.4. RsfSpringLoader 在 spring 项目中启动 RSF 的启动类。

方法	功能说明
ConfigLoader getConfigLoader()	取得 ConfigLoader
Object getBean(String name)	
Map getBean(Class clazz)	
String[] getBeanNamesForType(Class clazz)	
ApplicationContext getApplicationContext()	

23.5. AddressTool 获取通信双方的 IP 地址、端口。

服务端想取得通信双方的 IP 地址、端口时，可以使用 AddressTool 工具类。切忌一定要在服务端的服务接口实现类的业务方法体内使用本工具，否则只能返回 null。

客户端想取得通信双方的 IP 地址、端口时，可以使用 AddressTool 工具类。切忌一定要在客户端的服务接口代理类的业务方法调用**完成后**使用本工具，因为调用完成才能取出地址，否则可能返回“上一次通信时的地址”或 null。

方法	功能说明
static String toStringInfo()	返回值 :本端地址 :192.168.34.154:52460,远端地址 :192.168.34.154:63634
static String getLocalIp()	取得本端 IP
static int getLocalPort()	取得本端 port
static String getRemoteIp()	取得远端 IP
static int getRemotePort()	取得远端 port

23.6. CallBackHelper 服务端向客户端推送数据的工具

（测试版本，网络连接断开再重建后无法推送）

方法	功能说明
static void put(String key)	把当前的 CallBack 与一个 key 关联
static PushResult[] send(String key, Serializable data)	通过 key 找出 CallBack，并通过这些 CallBack 回推数据。可能找到多个 CallBack。返回的 PushResult 对象是推送结果。
PushResult send(String ip, int port, Serializable data)	通过 ip,prot 找出 1 个 CallBack，并通过这个 CallBack 推数据。返回的 PushResult 对象是推送结果。

PushResult send(String key, String ip, int port, Serializable data)	通过 key,ip,prot 找出 1 个 CallBack，并通过这个 CallBack 推数据。返回的 PushResult 对象是推送结果。
List<CallBackWrap> get(String key)	通过 key 找出全部关联的 CallBack

24. RSF 加密通信

账户安全访问服务，是慧聪网自主研发的基于认证、加密解密的安全网络访问服务。核心技术涉及 HRSF(远程服务调用框架)、HCC（配置管理中心）、HAS（账户安全申请服务）。

24.1. 证书种类说明

证书名称	证书用途	证书持有者	生命周期
COMMONKEY	通用密钥，各个系统都使用同一个 COMMONKEY。	业务系统、HCC、HAS 保存在硬盘的目录中	持久
PRIVATEKEY	系统名称+口令 经过加密后成为 PRIVATEKEY。 每个系统有自己的 PRIVATEKEY。	业务系统 保存在硬盘的目录中	持久
口令	业务系统在获取公钥、私钥时需要验证口令。 每个系统有自己的口令。	业务系统、HCC （有） 保存在数据库中	持久
公钥	非对称加解密算法使用的公钥。 每个系统有自己的公钥。	HCC 保存在数据库中	持久
私钥	非对称加解密算法使用的私钥。 每个系统有自己的私钥。	HCC 保存在数据库中	持久
sessionKey	执行三次握手时 HRSF 生成的一次性的密码，使用私钥加密传输，对方使用公钥解密出密码，完成三次握手，建立安全的网络通道。使用本通道传输的数据，都使用这个密码加密、解密。 每个安全的网络通道有自己的 sessionKey。	HRSF 保存在内存中	临时，网络连接断开后失效

24.2. 涉及系统的职责

1、HAS（账户安全申请服务）服务端

慧聪自主研发账户安全服务器证书申请服务，是一个 WEB 应用，可以通过浏览器访问。账户安全访问服务负责：

- HAS 管理员的管理与权限分配。
- 公钥、私钥、口令 的生成，并存储于 HCC。
- 下载 PRIVATEKEY。
- 列表查看。

2、HAS 客户端

加密解决算法工具类 *AuthHelper*，是一个 jar 包，提供静态工具方法，读取 COMMONKEY、PRIVATEKEY、口令 、公钥、私钥、sessionKey 完成各个环节的加密、解密。

3、HCC（配置管理中心）服务端

慧聪自主研发配置文件管理服务。在账户安全访问服务中，负责：

- 存储口令、公钥、公钥。
- 管理系统信任列表。
- 管理公私、私钥下载的 IP 白名单。

4、HCC（配置管理中心）客户端

作为一个 jar 包被各个业务系统使用，负责：

- 下载信任列表，供 HRSF 使用。
- 下载公钥、公钥。并在各业务系统内存中缓存各个版本公钥、公钥，供 HAS 客户端使用。

5、HRSF(远程服务调用框架)

慧聪自主研发网络通信框架。在账户安全访问服务中，负责：

- 使用 HAS 客户端，完成三次握手建立安全网络通道。
- 使用 HAS 客户端，对传输的数据加密、解密。

24.3. 依赖的第三方 jar 包

实现加密通信，一定要使用 1.3.0 及以上版本的 RSF。

RSF 的加密通信功能依赖了以下第三方 JAR 包： hasclient-1.5.jar、common_codec-1.4.jar、configure_client_1.4.0.jar

也可阅读 RSF 依赖的第三方 Jar 包

24.4. RSF 加密通信示例

概览表：

维度	IO 模型	配置方式	安全	服务发现	跨语言	注册中心
值	同步调用	XML	加密通信	3 点通信 通过注册中心 发现服务	只运行于 Java 语言平台	db 注册中心

场景：A 与 B 系统之间要使用加密通信--执行步骤		
顺序	**角色**	**操作**
1	HAS 管理员	为 A 系统申请证书
2	开发环境：开发人员 测试、正式环境：运维人员	为 A 系统所在的服务器安装证书
3	开发人员	A 系统的 RSF 启用加密通信
4	HAS 管理员	为 B 系统申请证书
5	开发环境：开发人员 测试、正式环境：运维人员	为 B 系统所在的服务器安装证书

6	开发人员	B 系统的 RSF 启用加密通信

24.4.1. 申请证书

操作人：HAS 管理员
目的：为 xx 系统申请一套证书
警告：非 HAS 管理员不要尝试操作，以免造成证书升级，导致重新部署
操作步骤：

1、确认系统名称（重要）

系统名称需要手工输入到 HAS 系统 ,要提前确认正确的系统名称。例如 hfbdeposit 系统的 portalId 是 10069，应该使用英文的系统名称 hfbdeposit，不能使用 10069，所以 xx=hfbdeposit。

系统名称要与 HCC（配置中心）中的系统名称一致，全小写字母。可能这个系统名称在 HCC 已存在，那只能使用 HCC 的系统名称。

可能这个系统名称在 HCC 不存在，HAS 会在 HCC 中创建这个系统名称。

2、登录 HAS

登录生产环境 HAS：http://has.org.hc360.com/hasservice

登录测试环境 HAS：http://has.org.hc360.com:8080/hasservice

3、HAS 系统管理员，为 xx 系统生成密钥。 （必选）

登录后，在管理页面输入系统名称，下载生成的 PRIVATEKEY，发送给业务开发人员。

4、HAS 系统管理员，为 xx 系统"激活"密钥。 （必选）

5、HAS 系统管理员，在配置管理中心，添加“系统信任列表”。（必选）（参考配置中心用户手册）

6、HAS 系统管理员，在配置管理中心，添加“下载密钥白名单”。（可选）（参考配置中心用户手册）

7、HAS 系统管理员，在配置管理中心，添加“下载配置文件白名单”。（可选）（参考配置中心用户手册）

24.4.2. 安装证书

操作人：xx 系统的开发人员、系统运维人员
目的：为 xx 系统安装证书，可以进行加密通信。
操作步骤：

1、开发人员联系 HAS 系统管理员，申请为 xx 系统生成一套证书，并取得 PRIVATEKEY。

3、COMMONKEY 的安放，根据你使用的操作系统，选择一个进行安放：

Linux 系统放在：/accountsecret/secretkey/ 目录下，权限为 web 服务器安装用户可读权限

Windows 系统放在：tomcat 所在磁盘如，E:\accountsecret\secretkey\目录下

4、PRIVATEKEY 的安放：

Linux 系统放在：/accountsecret/privatekey/xx/ 目录下，权限为 web 服务器安装用户可读权限

Windows 系统放在：tomcat 所在磁盘如，E:\accountsecret\privatekey\xx\ 目录下

xx 表示系统名称，与 accountsysid.properties 文件中的 sysid=xx 一致。

24.4.3. 开启 RSF 加密通信

操作人：xx 系统的开发人员

建立 accountsysid.properties 文件

开发人员在工程的 src 目录建立 accountsysid.properties 文件。

文件内容为：sysid=xx。

xx 表示系统名称，xx 这个系统名称必需要与配置中心中的系统名称一致。本例是：sysid=hfbdeposit

修改 RSF 配置文件，启用加密通信

在确保**非加密通信**可正常工作的前提下，只需要修改一处 RSF 的配置文件，就可以启用加密通信。

安全通信的服务端的 rsf 配置文件，<rsf:service>标签，需要添加属性 security="true"

安全通信的客户端的 rsf 配置文件，<rsf:client>标签，需要添加属性 security="true"

注意：

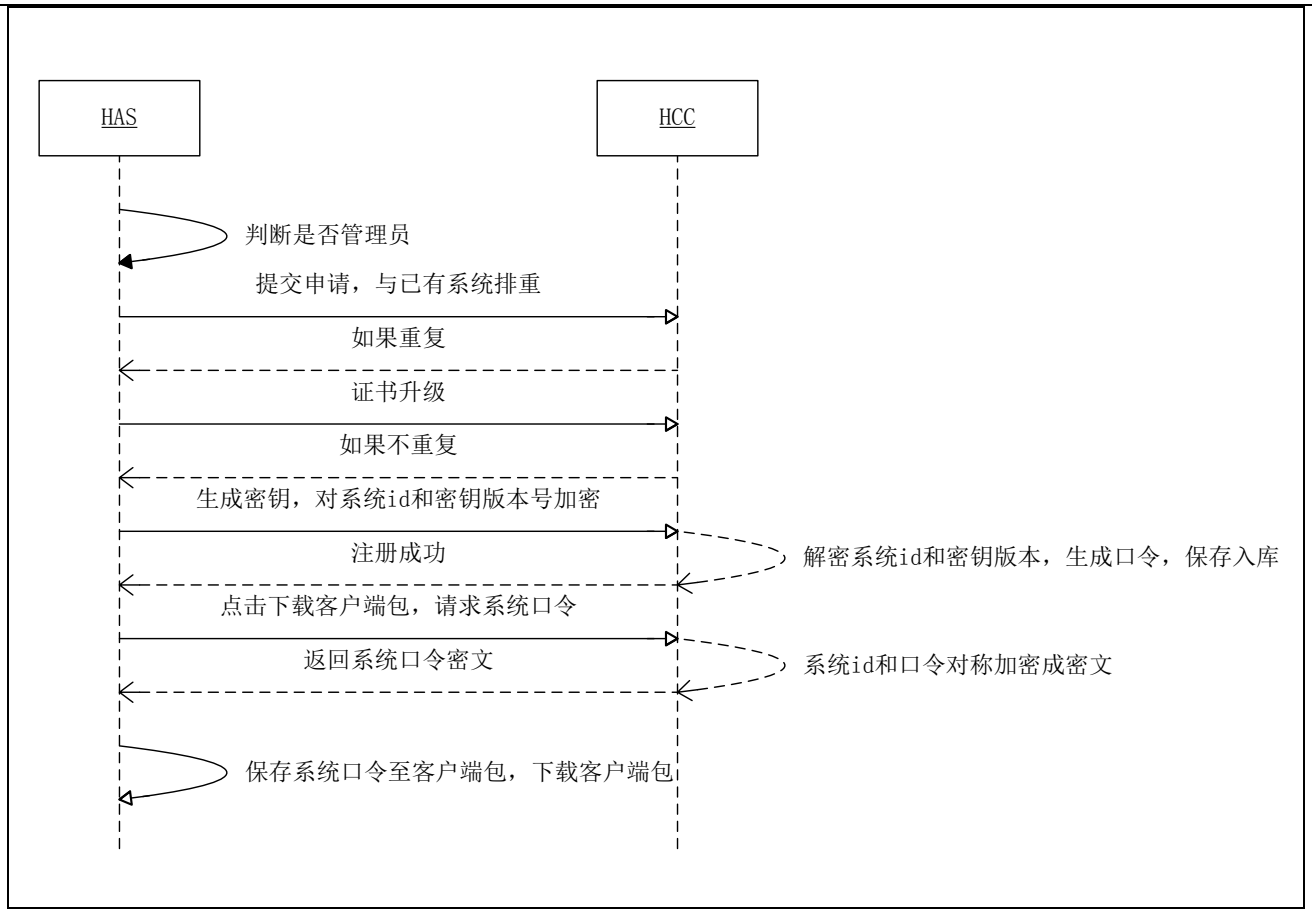
必须客户端与服务端同时启用加密，才能进行加密通信。不可以一端要求加密，另一端要求不加密，这样会导致抛出异常。

开启加密的控制级别是接口级，不能精确的方法级。一个接口中的多个方法，要么全开启加密，要么全关闭加密。

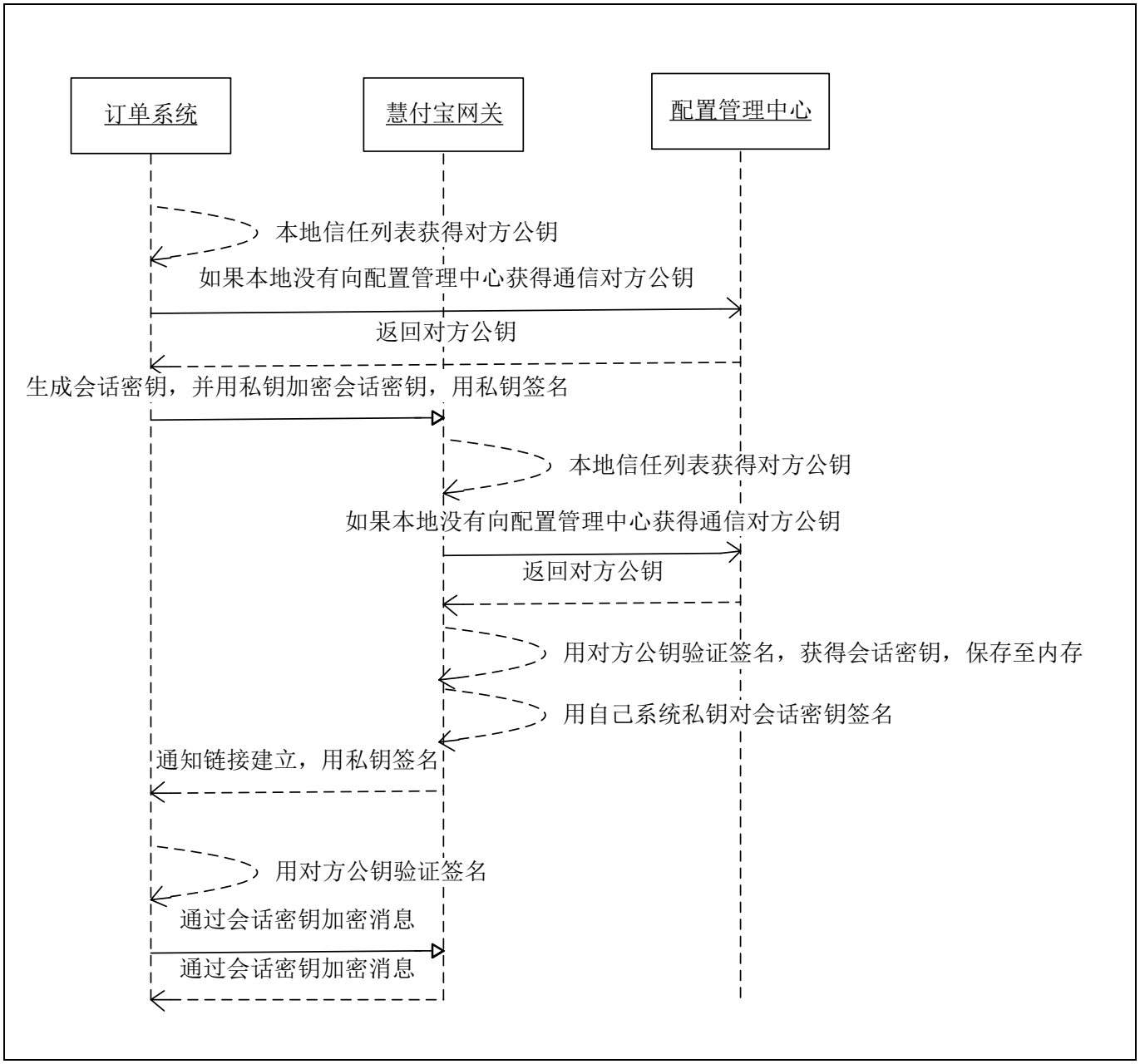
24.5. 加密通信原理

24.5.1. 证书申请时序图

申请证书时，HAS 与 HCC 有交互，过程如下图：


HAS
HCC
判断是否管理员
提交申请，与已有系统排重
如果重复
证书升级
如果不重复
生成密钥，对系统id和密钥版本号加密
注册成功
解密系统id和密钥版本，生成口令，保存入库
点击下载客户端包，请求系统口令
返回系统口令密文
系统id和口令对称加密成密文
保存系统口令至客户端包，下载客户端包

24.5.2. 三次握手时序图



24.5.3. 三次握手 RSF 通信流程设计

RSF 使用 HAS 项目提供 AuthHelper 工具类，完成三次握手。

Rsf 协议的第 19 位表示，是否是“三次握手”通信。

三次握手流程如下：

1、 RSF Client 端获得请求签名包 byte[] getRequestSignPackage(client 端系统 ID ，会话密钥)。

2、 *AuthHelper 工具使用 client 端系统 ID，向配置管理中心下载我方的公钥、私钥，并使用私钥对临时生成的会话密钥签章，返回签名包。*

3、 RSF 把请求签名包从客户端发向服务端

4、 RSF Server 端验证请求签名包，String authRequestSignPackage（client 端的系统 ID，请求签名包），返回“会话密钥”，保存会话密钥于内存。

5、 AuthHelper 工具使用 client 端系统 ID，向配置管理中心下载 client 方的公钥,验章，并返回会话密钥。

6、 RSF Server 端获得应答签名包 byte[] getResponseSignPackage(Server 端系统 ID，会话密钥)

7、 AuthHelper 工具使用 Server 端系统 ID，向配置管理中心下载我方的公钥、私钥并使用私钥对会话密钥签章，返回签名包。

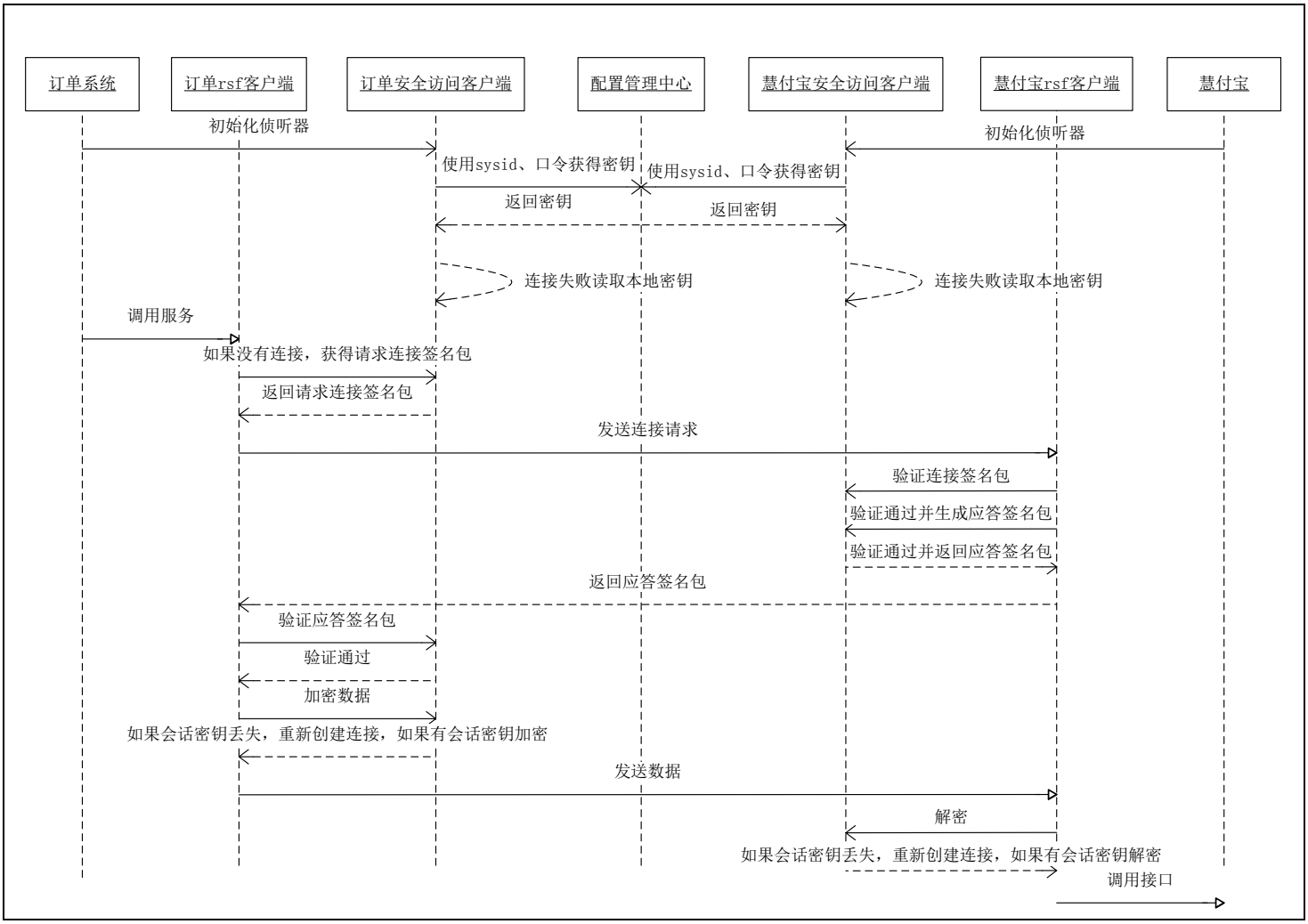
8、 RSF 把应答签名包从服务端发向客户端

9、 RSF Client 端验证应答签名包 authResponseSignPackage(Server 端系统 ID ,应答签名包) ,返回“会话密钥”。

10、 AuthHelper 工具使用 server 端系统 ID，向配置管理中心下载 server 方的公钥,验章，并返回会话密钥。

11、 RSF 验证返回的会话密钥是否是当初发送的会话密钥，保留会话密钥于内存，完成三次握手。

24.5.4. 业务系统通过 rsf 创建连接、加解密数据流程



24.5.5. 加密过程选用的算法

一、has 与 hcc 申请证书的加密方式

1、has 调用 hcc 接口添加系统，sysid 加密（加密算法 A），使用公共密钥文件

2、hcc 产生系统口令，使用公共密钥加密（加密算法 A），保存入库。

3、has 调用 hcc 接口保存系统私钥，sysid、version 加密（加密算法 A），使用公共密钥文件

4、hcc 接到系统私钥，sysid、version 解密（算法 A），私钥（不解密）、公钥（不解密）保存入库

二、hcc 客户端和 hcc 服务器建立连接的加密方式

1、hcc 客户端调用 has 客户端获得系统口令（算法 B）

2、hcc 客户端将系统 id（A 算法加密）和系统口令传给 hcc 服务端

3、hcc 服务端解密系统 id(A 算法)，系统口令（B 算法）和数据库中系统口令（A 算法）比较

4、hcc 服务端返回系统私钥、公钥（不用解密）

三、握手

1、hcc 生成会话密钥（明文）通过 has 客户端加密签名（算法 C），使用私钥密钥文件

2、hcc 将加密后会话密钥发送对方 hcc，进行解密和二次签名（算法 C），使用私钥密钥文件和公钥

3、对方 hcc 返回握手信息，调用 has 客户端解密（算法 C），使用公钥

四、通信

通信使用会话密钥加密（算法 D）

25. 跨语言通信

RSF 通过对 Thrift 的包装，实现跨语言的通信。目前包装了 C#、PHP、Java3 种语言，实现服务端功能与客户端功能，可以 3 种语言之间互相调用。

由于 RSF 使用了 Thrift、zookeeper，理论上 RSF 可支持任何 Thrift 支持的语言之进行通信。前提是未包装过的语言要自主去 zookeeper 注册中心上获取服务提供者信息 ,要了解 zookeeper 数据存储节点和数据存储格式。

25.1. 帮助文档和依赖包

开发语言	文档	开发人	依赖包
Java	RSF 用户手册第 18 版（本文档）	赵磊	thrift-0.9.1.exe libthrift-0.9.1.jar zookeeper-3.4.5.jar
C#	RSF2.0(C#)跨语言项目使用手册(第 1 版)	姚明臣	请看 RSF2.0(C#)文档
PHP	RFS2.0-PHP 用户手册	高志昌	请看 RFS2.0-PHP 文档

25.2. 名词解释

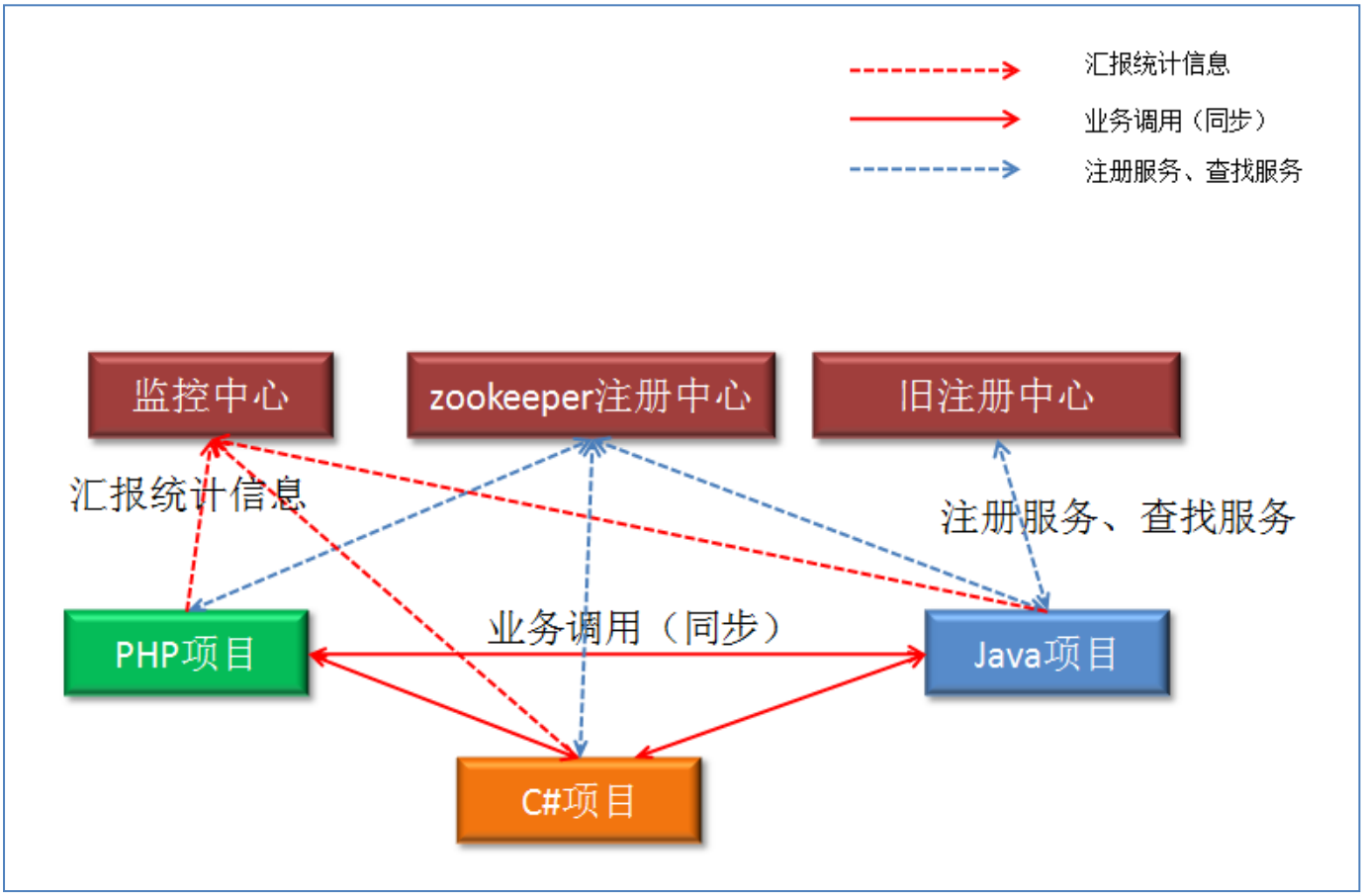
Thrift 是一个软件框架，用来进行可扩展且跨语言的服务的开发。它结合了功能强大的软件堆栈和代码生成引擎，以构建在 C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, and OCaml 这些编程语言间无缝结合的、高效的服务。

thrift 最初由 facebook 开发，07 年四月开放源码，08 年 5 月进入 apache 孵化器。

thrift 允许你定义一个简单的定义文件中的数据类型和服务接口。以作为输入文件，编译器生成代码用来方便地生成 RPC 客户端和服务器通信的无缝跨编程语言。

ZooKeeper 是 Hadoop 的正式子项目，它是一个针对大型分布式系统的可靠协调系统，提供的功能包括：配置维护、名字服务、分布式同步、组服务等。ZooKeeper 的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

25.3. 体系结构图



25.3.1. 远程服务调用框架服务端

核心功能：

1、实现使用 thrift 在本地暴露服务，可接受来自客户端的调用。

2、通过 zookeeper client 注册服务到 zookeeper 注册中心。

健壮性要求：

发生网络中断之后网络又恢复、注册中心重启情况，要保证“远程服务调用框架”的服务端可以自动重新注册服务。

25.3.2. 远程服务调用框架客户端

核心功能：

点对点通信：

客户端在已知服务端的 IP、端口的信息时，可进行点对点通信，不依赖注册中心。

三点通信：

客户端通过 zookeeper client 从 zookeeper 注册中心获取服务列表，含：协议类型（thrift、rsf、webservice）、IP、端口、（以后可能添加：权重、是否要密码、路由规则）。并通过 thrift 发起网络通信，调用服务端。

健壮性要求：

如果注册中心的服务列表发生了变化，就接收 zookeeper client 的通知后，再重新从 zookeeper 注册中心获取服务列表。

客户端特性：

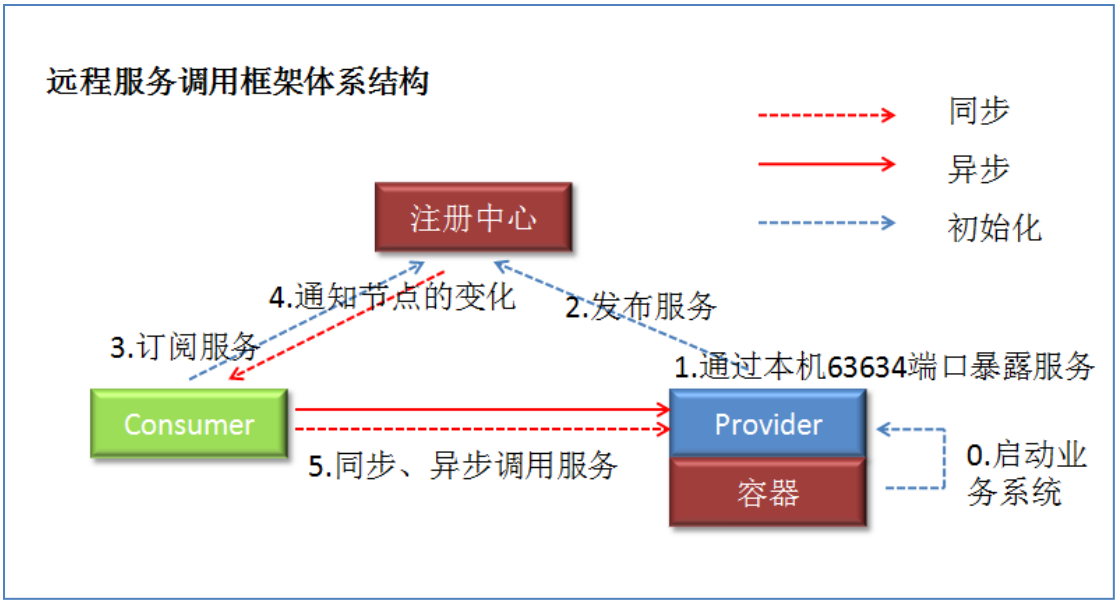
1、可设置请求超时时间，发出请求 N 秒内未接收到返回结果，应报超时错误。

2、负载均衡：在客户端实现软负载均衡，客户通过注册中心取得服务端务提供方地址列表，如果一个服务有 3 个节点提供，客户端应把多次请求均匀的分配到这 3 个节点上，实现软负载均衡。

3、失败转移：当出现失败，可偿试其它服务器节点（有多个服务提供者时）。

25.3.3. 稳定性

客户端：C ，服务端：S，注册中心：R 。三者可以在复杂的网络环境下正常通信。



场景 1：

客户端：C ，服务端：S，注册中心：R ，三者正在正常工作中，R 死掉，C 使用本地的缓存服务列表可以与 S 正常通信（维持旧的通信）。即使 C 无法访问 R，也不影响旧有的通信。

场景 2：

客户端：C ，服务端：S，注册中心：R ，三者正在正常工作中，S1 死掉，R 应感知，并通知 C。C 将不要调用死掉的 S1，转而调用 S2。

场景 3：
客户端：C ，服务端：S，注册中心：R ，C、R 已正常启动，但 S 未启动，C 就发起了调用，C 向注册中心查询服务列表时结果为空，C 应给予提示“无法找到服务提供者”，并停止发起调用。

场景 4：(未实现)
客户端：C ，服务端：S，注册中心：R ，三者正在正常工作中，突然 R 与 S 之间网络断开，其它网络正常，R、S、C 都在继续工作。由于 R 会认定 S 已死掉并通知 C，其实 S 未死掉，由于 C 与 S 间网络正常，C 与 S 是可以正常通信的。此场景可以考虑让 C 与 S 正常通信。

25.4. RSF 跨语言通信示例

概览表：

维度	IO 模型	配置方式	安全	服务发现	跨语言	注册中心
值	同步调用	XML	非加密通信	3 点通信 通过 zookeeper 注册中心发现服务	可跨语言平台	zookeeper 注册中心

25.4.1. 依赖的包

thrift-0.9.1.exe 用于把 IDL 文件生成各种语言的代码
libthrift-0.9.1.jar
zookeeper-3.4.5.jar 用于与 zookeeper 通信
注意 thrift 要使用 0.9.0 或 0.9.1 版本，其它版本未测试过。
登录 http://192.168.34.253/nexus 仓库，搜索 thrift-0.9.1.exe，进行下载

25.4.2. 基本步骤

编写 IDL 文件
通过 IDL 文件生成各种语言的代码，并 copy 到你的项目中
编写 RSF 配置文件

25.4.3. Thrift 特别的地方

1、 thrift 生成的代码，接口是一个内部类，在 RSF 配置文件中引用内部类要使用$符号，例如：com.hc360.rsf.thrift.ThriftTestInterfaceA$Iface
2、 thrift 每个服务(接口)独占一个端口，一个应用可以有多个服务(接口)，就要占用多个端口。这与 rsf 协议不同，rsf 协议是一个应用占用一个端口，一个应用中可以有多个服务(接口)。

25.4.4. 编写 IDL 文件

编写 test.thrift 文件，文件名随意起，无影响。

```
namespace java com.hc360.rsf.thrift
namespace php com.hc360.rsf.php
namespace csharp com.hc360.rsf.csharp
namespace cpp com.cpp

struct FileInfo{
   1: string fileUrl,
   2: string fileName,
   3: binary fileContext,
   4: list<string> gmFormat,
   5: i32 operateResult,
   6: string businTyp,
}

service ThriftTestInterfaceA {
   string a1(1:string arg),
   list<FileInfo> a2(1:list<FileInfo> fileList),
}

service ThriftTestInterfaceB {
   string b5(1:string arg),
   bool b6(1:list<FileInfo> fileList),
   void b7(),
}
```

25.4.5. 通过 IDL 文件生成各种语言的代码

下面是生成 Java 代码，注意 thrift 要使用 0.9.0 或 0.9.1 版本，其它版本未测试过。

登录 http://192.168.34.253/nexus 仓库，搜索 thrift-0.9.1.exe，进行下载

```
thrift-0.9.1.exe -r --gen java test.thrift
```

其它语言请参考：

RSF2.0(C#)跨语言项目使用手册(第 1 版)

RFS2.0-PHP 用户手册

把生成的代码 copy 到你的项目中。

25.4.6. 编写 RSF 配置文件

rsf_server.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rsf xmlns="http://code.hc360.com/schema/rsf"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rsf="http://code.hc360.com/schema/rsf">

    <!-- db注册中心 -->
    <rsf:registry id="reg1" type="db" host="register.org.hc360.com"></rsf:registry>

    <!-- zookeeper注册中心 -->
    <rsf:registry id="reg2" type="zookeeper"
address="192.168.44.112:2181,192.168.44.113:2181,192.168.44.114:2181"></rsf:registry>

    <!-- rsf协议-->
    <rsf:protocol id="protocol_rsf"  name="rsf" port="63634" ></rsf:protocol>

    <!-- thrift协议   -->
    <rsf:protocol id="protocol_thrift"  name="thrift" ></rsf:protocol>

    <rsf:service id=""
    protocolId="protocol_thrift" thriftPort="9008"
    displayName="测试服务A" owner="赵磊" department="用户平台开发部"
    interfaceClass="com.hc360.rsf.api.thrift.ThriftTestInterfaceA$Iface"
    class="com.hc360.rsf.api.thrift.ThriftTestInterfaceA_Impl"
    portalId="测试系统">
        <rsf:document><![CDATA[ 接口说明，请详细说明本接口的业务功能]]></rsf:document>
    </rsf:service>

    <rsf:service id=""
    protocolId="protocol_thrift" thriftPort="9009"
    displayName="测试服务B" owner="赵磊" department="用户平台开发部"
    interfaceClass="com.hc360.rsf.api.thrift.ThriftTestInterfaceB$Iface"
    class="com.hc360.rsf.api.thrift.ThriftTestInterfaceB_Impl"
    portalId="测试系统">
        <rsf:document><![CDATA[ 接口说明，请详细说明本接口的业务功能]]></rsf:document>
    </rsf:service>

    <rsf:service id=""
    registries="reg1,reg2"
    protocolId="protocol_rsf"
    displayName="RSF测试服务" owner="赵磊" department="用户平台开发部"
    interfaceClass="com.hc360.rsf.api.bean.data1.UserService"
    class="com.hc360.rsf.api.bean.data1.UserServiceImpl"
    portalId="测试" security="false" register="true">
        <rsf:document><![CDATA[ //记得带着包名]]></rsf:document>
    </rsf:service>

</rsf>
```

rsf_client.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rsf xmlns="http://code.hc360.com/schema/rsf"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rsf="http://code.hc360.com/schema/rsf"
```

```
    >

    <!-- db注册中心 -->
    <rsf:registry id="reg1" type="db" host="register.org.hc360.com"></rsf:registry>

    <!-- zookeeper注册中心 -->
    <rsf:registry id="reg2" type="zookeeper"
address="192.168.44.112:2181,192.168.44.113:2181,192.168.44.114:2181"></rsf:registry>

    <!-- rsf协议-->
    <rsf:protocol id="protocol_rsf"  name="rsf" port="63634"></rsf:protocol>

    <!-- thrift协议-->
    <rsf:protocol id="protocol_thrift"  name="thrift" ></rsf:protocol>

    <!--
    url="thrift://127.0.0.1:9001"
    -->
    <!-- 调用Java的服务端  -->
    <rsf:client id="ThriftTestInterfaceA_Impl" displayName="调用用户测试服务" owner="张三"
department="MMT开发部"
    registries="reg1,reg2"
    interfaceClass="com.hc360.rsf.api.thrift.ThriftTestInterfaceA$Iface"
    protocolId="protocol_thrift"
    portalId="测试" timeout="3000">
    </rsf:client>

    <!-- 调用Java的服务端  -->
    <rsf:client id="ThriftTestInterfaceB_Impl" displayName="调用用户测试服务" owner="张三"
department="MMT开发部"
    registries="reg1,reg2"
    interfaceClass="com.hc360.rsf.api.thrift.ThriftTestInterfaceB$Iface"
    protocolId="protocol_thrift"
    portalId="测试" timeout="3000">
    </rsf:client>

</rsf>
```

服务端

```
package com.hc360.rsf.api.rsf20;

import com.hc360.rsf.config.ConfigLoader;

/**
 * thrift通信测试
 *
 * 三节点之间通信测试，Client、Server、注册中心之间通信测试
 * 测试前请保证“注册中心”可以访问
 *
 * 模拟服务提供者 （服务端）
 *
 * @author zhaolei 2013-9-24
 */
public class MainServer {

    /**
     * 加载rsf_server.xml配置文件
```

```
     *
     * 在本地暴露服务
     *
     * 向注册中心注册服务
     *
     * @param args
     */
    public static void main(String[] args) {
        String xmlPath = "classpath:rsf_server.xml";
        ConfigLoader configLoader = new ConfigLoader(xmlPath,MainServer.class);
        configLoader.start();
    }
}
```

客户端

```
package com.hc360.rsf.api.rsf20;
import com.hc360.rsf.api.thrift.ThriftTestInterfaceA;
import com.hc360.rsf.api.thrift.ThriftTestInterfaceB;
import com.hc360.rsf.config.ConfigLoader;

/**
 * 三节点之间通信测试，Client、Server、注册中心之间通信测试
 * 测试前请保证“注册中心”可以访问
 *
 * 模拟服务调用者（客户端）
 *
 * @author zhaolei 2012-6-27
 */
public class MainClient {

    /**
     * 加载rsf_server.xml配置文件
     *
     * 从注册下载服务提供者列表
     *
     * 发起调用
     *
     * @param args
     */
    public static void main(String[] args) {
        String xmlPath = "classpath:rsf_client.xml";
        ConfigLoader configLoader = new ConfigLoader(xmlPath,MainClient.class);
        configLoader.start();
        ThriftTestInterfaceA.Iface userServiceA= (ThriftTestInterfaceA.Iface)
configLoader.getServiceProxyBean("ThriftTestInterfaceA_Impl");//配置文件中的id
        ThriftTestInterfaceB.Iface userServiceB= (ThriftTestInterfaceB.Iface)
configLoader.getServiceProxyBean("ThriftTestInterfaceB_Impl");//配置文件中的id
        for(int i=1;i<=5;i++){
            try {
                System.out.println("调用完成，返回结果："+userServiceA.a1(""+i));
                System.out.println("调用完成，返回结果："+userServiceA.a2(null));

                System.out.println("调用完成，返回结果："+userServiceB.b5(""+i));
                System.out.println("调用完成，返回结果："+userServiceB.b6(null));

                userServiceB.b7();
                System.out.println("调用完成，返回结果：userServiceB.b7()");
```

```
            } catch (Exception e) {
                e.printStackTrace();
            }finally{
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        ConfigLoader.destroy();
    }
}
```

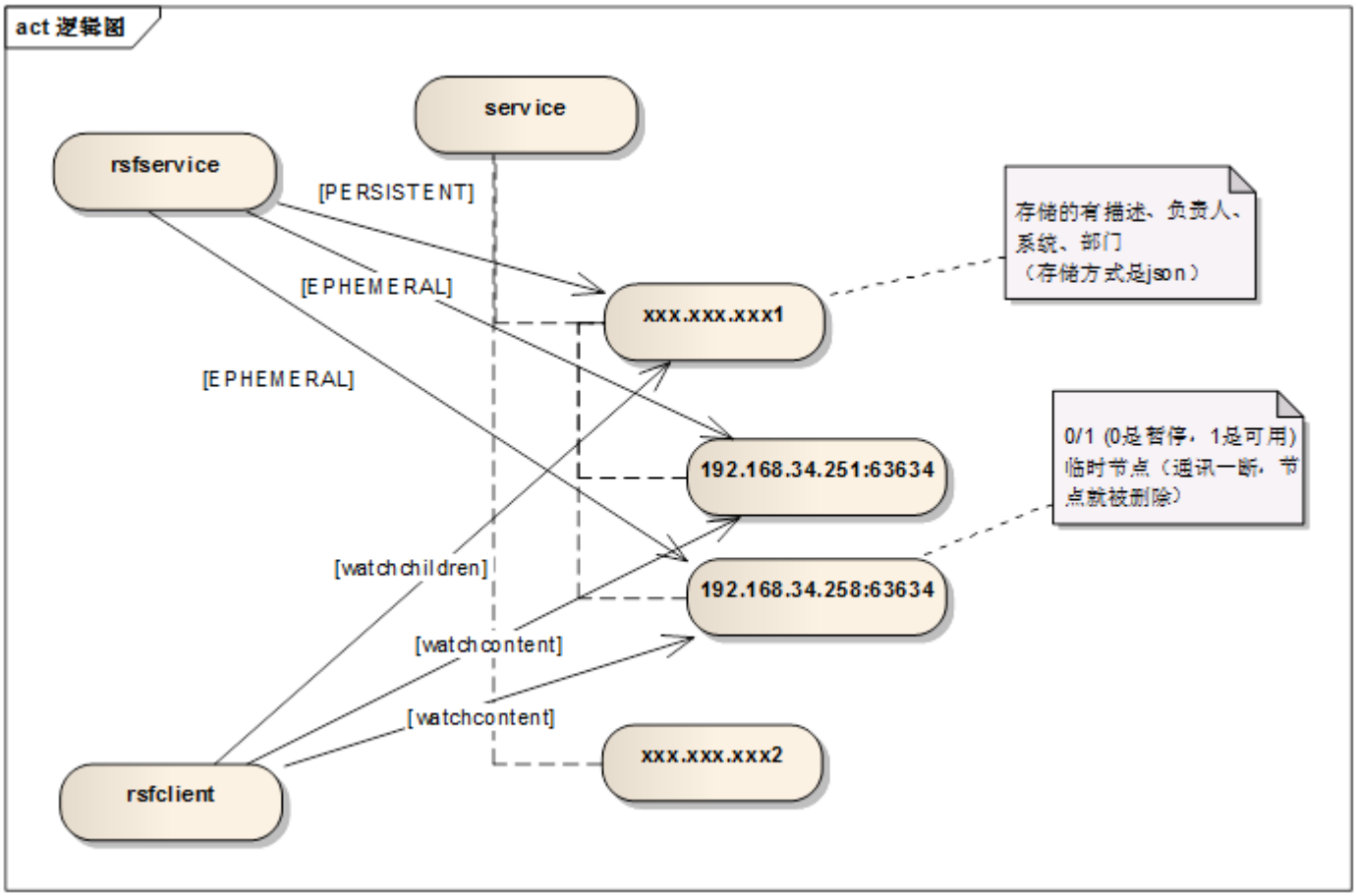
运行服务端的 main 方法。

运行客户端的 main 方法。

一切都顺利的话，你应该看到调用成功了。

25.5. zookeeper 注册中心数据存储格式

由于 RSF 使用了 Thrift、zookeeper，理论上 RSF 可支持任何 Thrift 支持的语言之进行通信。前提是未包装过的语言要自主去 zookeeper 注册中心上获取服务提供者信息 ,要了解 zookeeper 数据存储节点和数据存储格式。



Rsfservice 注册服务分为两种，服务目录是持久化目录，服务提供者节点目录是临时目录；
Rsfclient 监控服务变化也分为两种，对服务目录的 watcher 监控子目录是否有变化，对节点目录进行内容监控。
registerConsole 显示列表功能点只需遍历/service 的子目录；查询服务只需过滤/service 子目录；暂停/启用服务节点只需要对相应节点操作即可；至于清理不可用服务也非常方便，因为利用 zookeeper 临时目录的 session 一断就删目录的特性，只需删除没有子目录的服务目录即可。
好处：数据结构清晰，自己管自己的。Rsfservice 直接注册，无需数据处理。
坏处：由于临时目录的特性，可能出现暂停的服务节点由于重新注册而可用，引起其他用户的疑问，到时候不好解释，例如网络闪断。

25.5.1. 根结点命名

在 zookeeper 中保存的，服务提供信息，统一存放在一个父节点下，父节点名名称为 rsf_service

25.5.2. “服务”在 zookeeper 保存的数据格式

```
<?xml version="1.0" encoding="UTF-8"?>
<service>
        <property name="xxx">xxx</property >
        <property name="descibe"> <![CDATA[ 这是文档 ]]> </property >
</service>
```

key	数据类型	说明	
protocol	String	协议 thrift\rsf\webservice	
systemName	String	系统名称	
serviceName	String	全局唯一的服务名/path	
displayName	String	服务的中文名称	
owner	String	服务发布人	
department	String	服务发布人部门	
descibe	String	服务接口总体功能描述 使用 rsf 协议时，是 Java 接口文件源代码 使用 Thrift 协议时，是 idl 文件内容	
jarVersion	String	使用的 RSF 版本号如:RSF(PHP)1.0	
encode	String	中文参数与返回值的编码	
layer	String	服务所在层	未使用
version	String	服务接口版本	
weights	int	权重	
token	String	令牌	
url	String	URL	

25.5.3. “节点“在 zookeeper 保存的数据格式

```
<?xml version="1.0" encoding="UTF-8"?>
<node>
        <property name="xxx">xxx</property >
        <property name="xxx">xxx</property >
</node>
```

key	数据类型	说明	
ip	String	服务提供者的 ip	注意双网卡问题
port	int	服务提供者的端口	
stat	int	是否可用的标志位，状态: 0-激活/1-暂停/2-不可用/	（再次注册时会被覆盖）

25.5.4. serviceName 全局唯一的服务名命名规范。

在同一个 IDL 文件，不同语言的命名空间可以不相同，如下例：
namespace java com.hc360.my.xxx
namespace csharp com.hc360.my.xxx
namespace php com.hc360.my.xxx
namespace cpp tutorial

所以无法使用 namespace 做为全局唯一的服务名(Zookeeper 的 path)，需要按规范生成 serviceName，达到“全局唯一”的目的。
1、由服务提供者 制定 serviceName。
2、建议格式 com.hc360.rsf.系统名.服务名。

26. 灰度发布的支持

26.1. 背景

现有的 RSF 环境是一个所有 server（服务提供者），client（服务消耗者）构成的大环境，不利于线上测试验证。灰度发布是想通过构造几个业务可以互通的小环境，逐步实现局部上线，平缓过渡，提供微创新环境。

26.2. 目的

通过将指定的系统分组，以达到创建不同的 rsf 交互环境，便于公司微创新的实施。

26.3. 功能要求

1. 分组内的客户端只能调用分组内的服务端。
2. 如果开启分组里只有客户端，那客户端可以调用未分组或未开启分组的服务端；
3. 如果开启分组里只有服务端，那服务端的服务不能给任何客户端用；
4. 如果开启分组里既有服务端又有客户端，则客户端只能调用分组里的服务端，即使服务端的服务不可用，客户端也不能调用分组外的任何服务端。

26.4. 使用要求

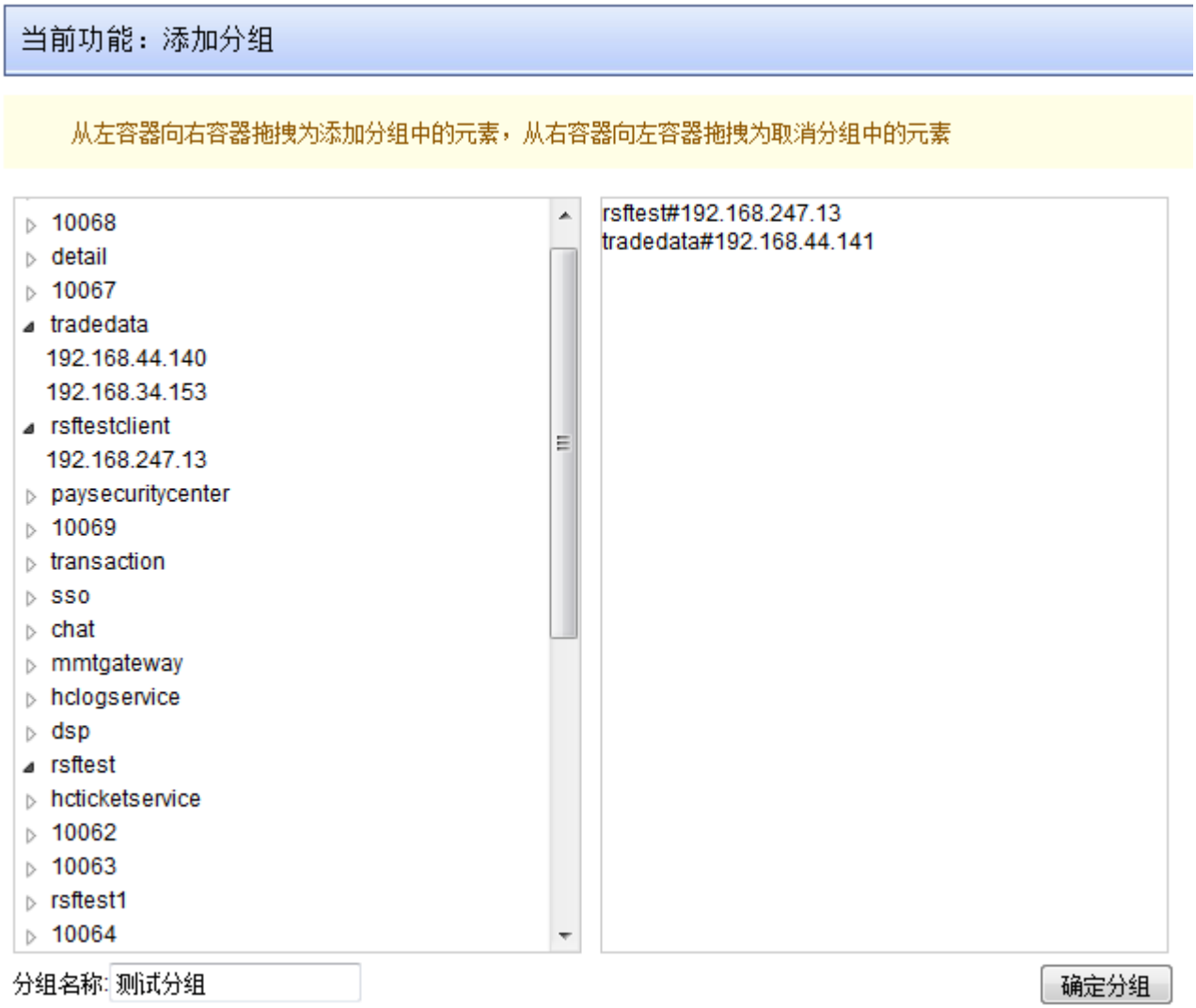
RSF 版本：2.1.0 以上

26.5. 原理

在配置中心进行设置分组信息，rsf 客户端得到服务所有的提供者信息及已开启的分组信息，进行对服务提供者的过滤，得到自己能用的服务提供者。

26.6. 操作流程

1. 登录到注册中心，地址是：http://register.org.hc360.com/register；
2. 选择左侧菜单中的“新添分组”,进入以下页面，从左侧容器里拖拽系统实例到右侧容器，并填写分组名；



3. 点击“确定分组 “，跳转到”分组列表“中

说明：这时只是把几个系统化成了一个逻辑单元，并没有真正的创建一个自己交互的环境；

点击“启动 “，建立一个自己交互的环境；

只有是“未启动“状态的分组才能删除。

26.7. 验证方式

可前后两次查看客户端信息来验证，看看是否有变化，是否与预期的一样。