
Dromara soul document

Dromara

2021 年 04 月 16 日

1	什么是 Soul	1
2	功能	2
3	架构图	3
4	团队介绍	4
4.1	团队成员（排名不分先后）	4
5	设计文档	5
5.1	数据库设计	5
5.2	配置流程介绍	6
5.2.1	说明	6
5.2.2	使用	6
5.2.3	作用	7
5.3	数据同步设计	7
5.3.1	说明	7
5.3.2	前言	7
5.3.3	原理分析	8
5.3.4	Zookeeper 同步	9
5.3.5	Websocket 同步	9
5.3.6	Http 长轮询	11
5.3.7	仓库地址	13
5.3.8	最后	13
5.4	元数据概念设计	13
5.4.1	说明	13
5.4.2	技术方案	14
5.4.3	元数据存储	15
6	Admin 使用文档	16
6.1	字典管理	16
6.1.1	说明	16

6.1.2	表设计	16
6.2	插件处理详解	17
6.2.1	说明	17
	表设计	17
	使用教程	17
6.3	选择器规则详解	19
6.3.1	说明	19
6.3.2	大体理解	20
6.3.3	选择器	20
6.3.4	规则	21
6.3.5	条件详解	22
7	用户文档	24
7.1	环境搭建	24
7.1.1	说明	24
7.1.2	启动 Soul-Admin	24
	远程下载	24
	docker 构建	25
	本地构建	25
7.1.3	启动 Soul-Bootstrap	26
	远程下载	26
	docker 构建	26
	本地构建	26
7.1.4	搭建自己的网关（推荐）	26
7.2	Http 用户	27
7.2.1	说明	27
7.2.2	引入网关对 http 的代理插件	28
7.2.3	Http 请求接入网关（springMvc 体系用户）	28
	Soul-Client 接入方式。（此方式针对 SpringMvc, SpringBoot 用户）	28
7.2.4	Http 请求接入网关（其他语言，非 springMvc 体系）	30
7.2.5	用户请求	30
7.3	Dubbo 接入 soul 网关	31
7.3.1	说明	31
7.3.2	引入网关对 dubbo 支持的插件	31
7.3.3	dubbo 服务接入网关，可以参考：soul-examples-dubbo	33
7.3.4	dubbo 插件设置	34
7.3.5	接口注册到网关	35
7.3.6	dubbo 用户请求以及参数说明	35
7.3.7	服务治理	36
7.3.8	大白话讲解如果通过 http -> 网关-> dubbo provider	38
7.4	SpringCloud 接入 Soul 网关	38
7.4.1	说明	38
7.4.2	引入网关 springCloud 的插件支持	39
7.4.3	SpringCloud 服务接入网关	40
7.4.4	插件设置	42

7.4.5	用户请求	42
7.5	Sofa 接入网关	43
7.5.1	说明	43
7.5.2	引入网关对 sofa 支持的插件	43
7.5.3	sofa 服务接入网关, 可以参考: soul-examples-sofa	44
7.5.4	sofa 插件设置	45
7.5.5	接口注册到网关	45
7.5.6	sofa 用户请求以及参数说明	45
7.6	使用不同的数据同步策略	46
7.6.1	说明	46
7.6.2	websocket 同步 (默认方式, 推荐)	46
7.6.3	zookeeper 同步	47
7.6.4	http 长轮询同步	48
7.6.5	nacos 同步	48
8	注册中心	50
8.1	注册中心设计	50
8.1.1	说明	50
8.1.2	Client	50
8.1.3	Server	53
8.1.4	Http 注册	54
8.1.5	Zookeeper 注册	54
8.1.6	Etcd 注册	55
8.1.7	Consul 注册	55
8.1.8	Nacos 注册	56
8.1.9	SPI 扩展	56
8.2	注册中心接入配置	57
8.2.1	说明	57
8.2.2	HTTP 方式注册	57
	Soul-Admin 配置	57
	Soul-Client 配置	58
8.2.3	Zookeeper 方式注册	58
	Soul-Admin 配置	58
	Soul-Client 配置	59
8.2.4	Etcd 方式注册	59
	Soul-Admin 配置	59
	Soul-Client 配置	60
8.2.5	Consul 方式注册	60
	Soul-Admin 配置	60
	Soul-Client 配置	62
8.2.6	Nacos 方式注册	63
	Soul-Admin 配置	63
	Soul-Client 配置	63
9	快速开始	65

9.1	Dubbo 快速开始	65
9.1.1	环境准备	65
9.1.2	运行 soul-examples-dubbo 项目	65
9.1.3	dubbo 插件设置	67
9.1.4	测试	67
9.2	Http 快速开始	69
9.2.1	环境准备	69
9.2.2	运行 soul-examples-http 项目	70
9.2.3	开启 divide 插件来处理 http 请求	71
9.2.4	测试 Http 请求	71
9.3	Grpc 快速开始	71
9.3.1	环境准备	72
9.3.2	运行 soul-examples-grpc 项目	72
9.3.3	Grpc 插件设置	73
9.3.4	测试	73
9.4	SpringCloud 快速开始	73
9.4.1	环境准备	74
9.4.2	运行 soul-examples-springcloud、soul-examples-eureka 项目	74
9.4.3	开启 springCloud 插件	77
9.4.4	测试 Http 请求	77
9.5	Sofa 快速开始	77
9.5.1	环境准备	77
9.5.2	运行 soul-examples-sofa 项目	78
9.5.3	sofa 插件设置	82
9.5.4	测试	82
9.6	Tars 快速开始	84
9.6.1	环境准备	84
9.6.2	运行 soul-examples-tars 项目	84
9.6.3	tars 插件设置	86
9.6.4	测试	86
10	插件集合	88
10.1	Divide 插件	88
10.1.1	说明	88
10.1.2	插件设置	88
10.1.3	插件讲解	89
10.2	Dubbo 插件	89
10.2.1	说明	89
10.2.2	插件设置	90
10.2.3	元数据	90
10.3	SpringCloud 插件	90
10.3.1	说明	90
10.3.2	引入网关 springCloud 的插件支持	90
10.3.3	插件设置	91
10.3.4	详解	91

10.4	Sofa 插件	91
10.4.1	说明	91
10.4.2	插件设置	92
10.4.3	元数据	92
10.5	限流插件	92
10.5.1	说明	92
10.5.2	技术方案	92
	采用 redis 令牌桶算法进行限流。	92
	采用 redis 漏桶算法进行限流。	93
	基于 redis 实现的滑动窗口算法	94
10.5.3	插件设置	94
10.5.4	插件使用	95
10.6	Hystrix 插件	95
10.6.1	说明	95
10.6.2	插件设置	96
10.6.3	插件使用	96
10.7	Sentinel 插件	96
10.7.1	说明	96
10.7.2	插件设置	97
10.7.3	插件使用	97
10.8	Resilience4j 插件	97
10.8.1	说明	97
10.8.2	插件设置	98
10.8.3	插件使用	98
10.9	Monitor 插件	99
10.9.1	说明	99
10.9.2	技术方案	99
10.9.3	插件设置	100
10.9.4	插件使用	100
10.9.5	metrics 信息	100
10.9.6	收集 metrics	101
10.9.7	面板展示	101
10.10	Waf 插件	103
10.10.1	说明	103
10.10.2	插件设置	103
10.10.3	插件使用	103
10.10.4	场景	104
10.11	Sign 插件	104
10.11.1	说明	104
10.11.2	插件设置	104
10.11.3	插件使用	104
10.11.4	新增 AK/SK	105
10.11.5	网关技术实现	105
10.11.6	鉴权使用指南	105
10.11.7	请求网关	106

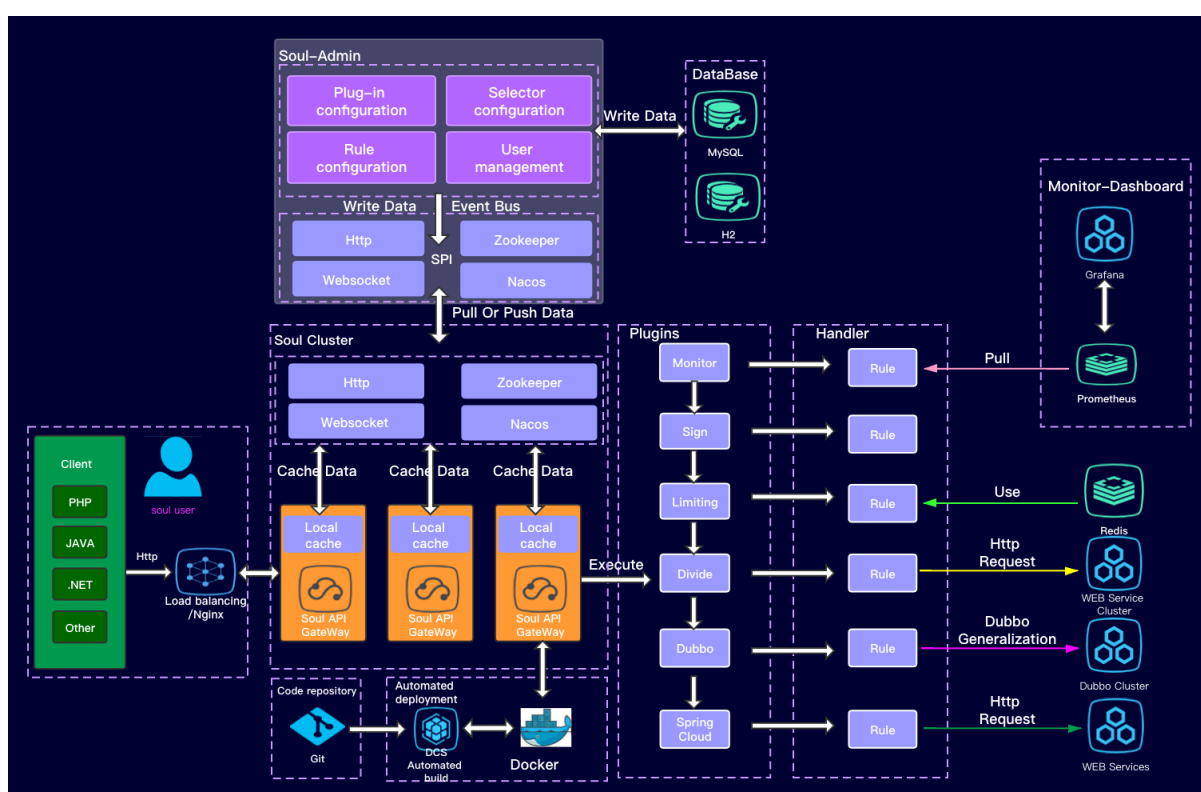
10.11.8	如果认证不通过会返回 code 为 401 message 可能会有变动。	106
10.11.9	签名认证算法扩展	106
10.12	Rewrite 插件	106
10.12.1	说明	106
10.12.2	插件设置	107
10.12.3	场景	107
10.13	Websocket 支持	107
10.13.1	说明	107
10.13.2	插件设置	107
10.13.3	请求路径	108
10.14	Context Path 插件	109
10.14.1	说明	109
10.14.2	插件设置	110
10.14.3	场景	110
10.15	重定向插件	110
10.15.1	说明	110
10.15.2	插件设置	110
10.15.3	Maven 依赖	111
10.15.4	场景	111
	重定向	111
	网关自身接口转发	111
11	开发者文档	113
11.1	自定义 Filter	113
11.1.1	说明	113
11.1.2	跨域支持	113
11.1.3	网关过滤 springboot 健康检查	114
11.1.4	继承 org.dromara.soul.web.filter.AbstractWebFilter	115
11.2	插件扩展	115
11.2.1	说明	115
11.2.2	单一职责插件	116
11.2.3	匹配流量处理插件	117
11.2.4	订阅你的插件数据, 进行自定义的处理	119
11.3	文件上传下载	121
11.3.1	说明	121
11.3.2	文件上传	121
11.3.3	文件下载	121
11.4	正确获取 Ip 与 host	121
11.4.1	说明	121
11.4.2	默认实现	121
11.4.3	扩展实现	122
11.5	自定义网关返回数据格式	122
11.5.1	说明	122
11.5.2	默认实现	122
11.5.3	扩展	123

11.6	自定义 sign 插件检验算法	124
11.6.1	说明	124
11.6.2	扩展	124
11.7	多语言 http 客户端	124
11.7.1	说明	124
11.7.2	自定义开发	125
11.8	线程模型	125
11.8.1	说明	125
11.8.2	io 与 work 线程	125
11.8.3	业务线程	125
11.8.4	切换类型	126
11.9	Soul 性能优化	126
11.9.1	说明	126
11.9.2	本身消耗	126
11.9.3	底层 netty 调优	126
12	社区贡献	128
12.1	Soul Contributor	128
12.1.1	提交 issue	128
12.1.2	开发流程	128
12.2	Soul Committer	130
12.2.1	提交者提名	130
12.2.2	提交者责任	130
12.2.3	日常工作	130
12.3	Soul Code Conduct	131
12.3.1	开发理念	131
12.3.2	代码提交行为规范	131
12.3.3	编码规范	131
12.3.4	单元测试规范	132
13	文档下载	134
13.1	PDF	134

这是一个异步的，高性能的，跨语言的，响应式的 API 网关。

- 支持各种语言 (http 协议), 支持 dubbo, springcloud 协议。
- 插件化设计思想, 插件热插拔, 易扩展。
- 灵活的流量筛选, 能满足各种流量控制。
- 内置丰富的插件支持, 鉴权, 限流, 熔断, 防火墙等等。
- 流量配置动态化, 性能极高, 网关消耗在 1~2ms。
- 支持集群部署, 支持 A/B Test, 蓝绿发布。

架构图



4.1 团队成员（排名不分先后）

名字	github	角色	所在公司
肖宇	yu199195	VP	京东
张永伦	tuohai666	PMC	京东
邓力铭	dengliming	PMC	某创业公司
汤煜冬	tydhot	PMC	perfma
张磊	SaberSola	PMC	哈罗
黄晓峰	huangxfchn	committer	shein
丁剑明	nuo-promise	committer	某创业公司
冯振兵	fengzhenbing	committer	某创业公司
杨泽	HoldDie	committer	IBM

5.1 数据库设计

- 插件采用数据库设计，来存储插件，选择器，规则配置数据，以及对应关系。
- 数据库表 UML 类图：

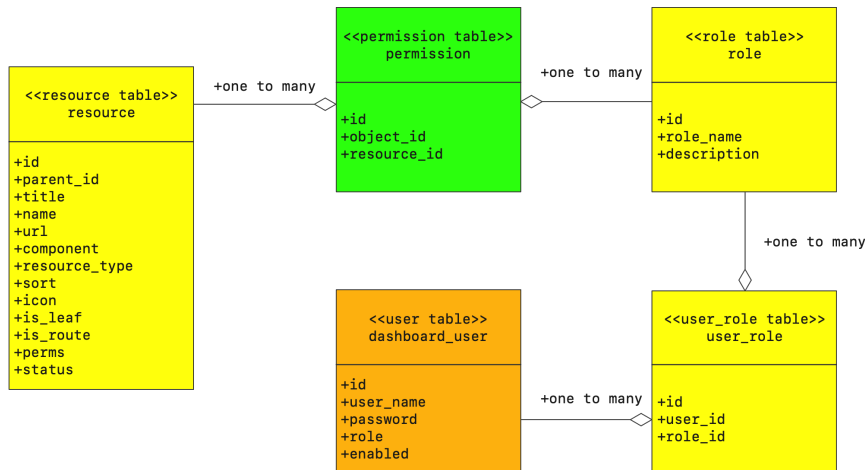


- 设计详解：
 - 一个插件对应多个选择器，一个选择器对应多个规则。
 - 一个选择器对应多个匹配条件，一个规则对应多个匹配条件。
 - 每个规则在对应插件下，不同的处理表现为 handle 字段，handle 字段就是一个 json 字符串。

具体的可以在 admin 使用过程中进行查看。

- 资源权限设计用来存储用户名称、角色、资源数据以及对应关系

- 数据库 UML 类图：



- 设计详情:

- 一个用户对应多个角色, 一个角色对应多个资源。

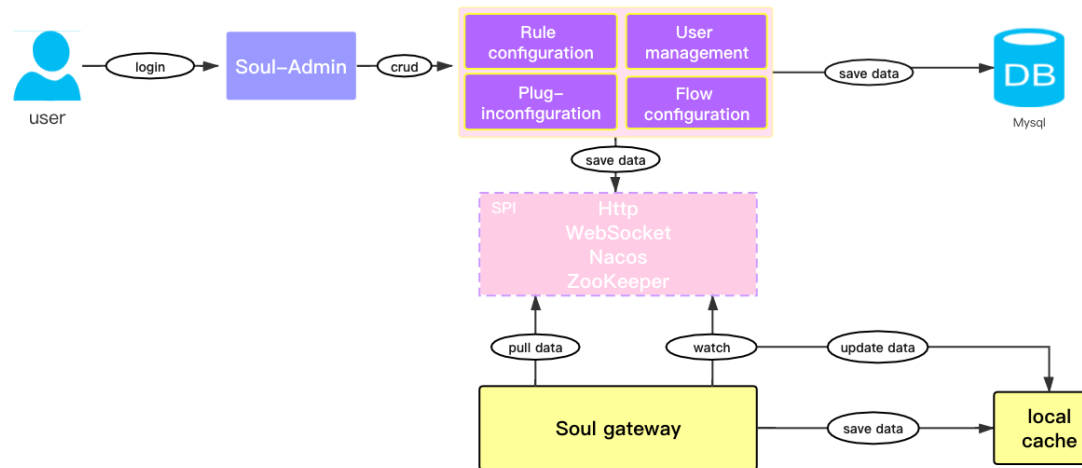
5.2 配置流程介绍

5.2.1 说明

- 本篇是对 admin 后台操作数据以后, 同步到网关的流程介绍。

5.2.2 使用

- 用户可以在 soul-admin 后台任意修改数据, 并马上同步到网关的 jvm 内存中。
- 同步 soul 的插件数据, 选择器, 规则数据, 元数据, 签名数据等等。
- 所有插件的选择器, 规则都是动态配置, 立即生效, 不需要重启服务。



- 下面是数据流程图：

5.2.3 作用

- 用户所有的配置都可以动态的更新，任何修改不需要重启服务。
- 使用了本地缓存，在高并发的时候，提供高效的性能。

5.3 数据同步设计

5.3.1 说明

本篇主要讲解数据库同步的三种方式，以及原理。

5.3.2 前言

网关是流量请求的入口，在微服务架构中承担了非常重要的角色，网关高可用的重要性不言而喻。在使用网关的过程中，为了满足业务诉求，经常需要变更配置，比如流控规则、路由规则等等。因此，网关动态配置是保障网关高可用的重要因素。那么，Soul 网关又是如何支持动态配置的呢？

使用过 Soul 的同学都知道，Soul 的插件全都是热插拔的，并且所有插件的选择器、规则都是动态配置，立即生效，不需要重启服务。但是我们在使用 Soul 网关过程中，用户也反馈了不少问题

- 依赖 zookeeper，这让使用 etcd、consul、nacos 注册中心的用户很是困扰
- 依赖 redis、influxdb，我还没有使用限流插件、监控插件，为什么需要这些

因此，我们对 Soul 进行了局部重构，历时两个月的版本迭代，我们发布了 2.0 版本

- 数据同步方式移除了对 zookeeper 的强依赖，新增 http 长轮询以及 websocket
- 限流插件与监控插件实现真正的动态配置，由之前的 yml 配置，改为 admin 后台用户动态配置

配置同步为什么不使用配置中心呢？

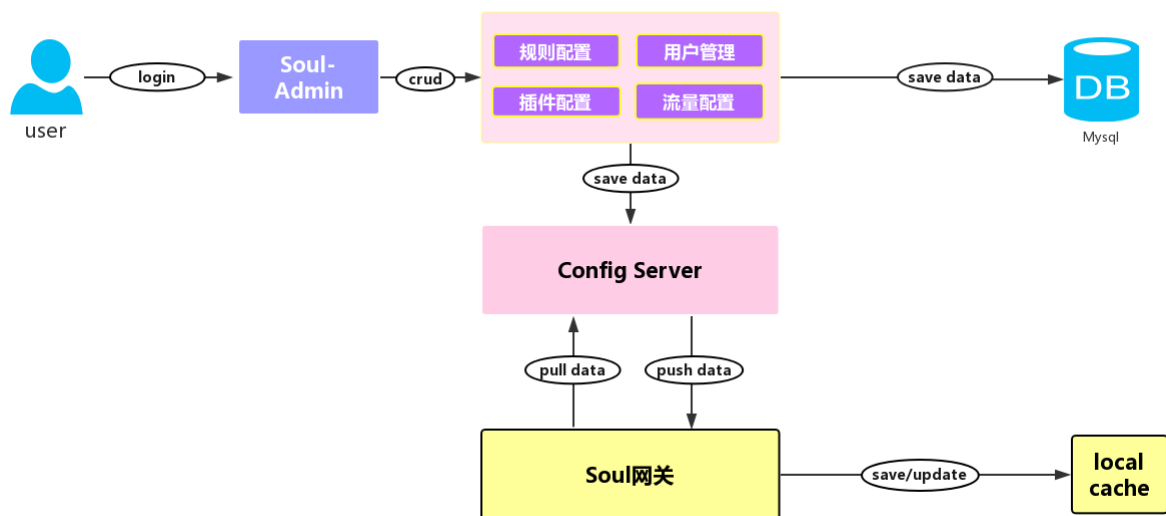
首先，引入配置中心，会增加很多额外的成本，不光是运维，而且会让 Soul 变得很重；另外，使用配置中心，数据格式不可控，不便于 soul-admin 进行配置管理。

动态配置更新？每次我查数据库，或者 *redis* 不就行了吗？拿到的就是最新的，哪里那么多事情呢？

soul 作为网关，为了提供更高的响应速度，所有的配置都缓存在 JVM 的 Hashmap 中，每次请求都走的本地缓存，速度非常快。所以本文也可以理解为分布式环境中，内存同步的三种方式。

5.3.3 原理分析

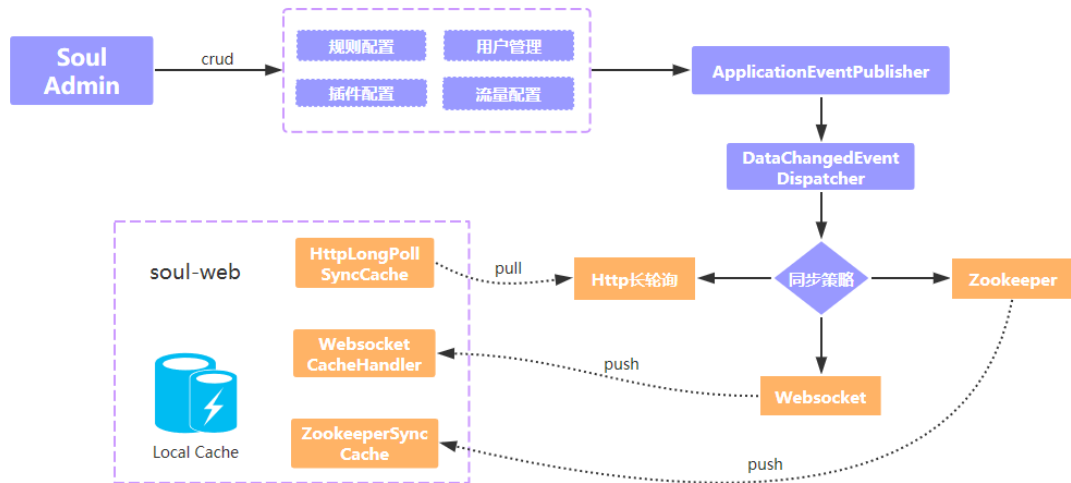
先来张高清无码图，下图展示了 Soul 数据同步的流程，Soul 网关在启动时，会从配置服务同步配置数据，并且支持推拉模式获取配置变更信息，并且更新本地缓存。而管理员在管理后台，变更用户、规则、插件、流量配置，通过推拉模式将变更信息同步给 Soul 网关，具体是 push 模式，还是 pull 模式取决于配置。关于配置同步模块，其实是一个简版的配置中心。



在 1.x 版本中，配置服务依赖 zookeeper 实现，管理后台将变更信息 push 给网关。而 2.x 版本支持 websocket、http、zookeeper，通过 `soul.sync.strategy` 指定对应的同步策略，默认使用 websocket 同步策略，可以做到秒级数据同步。但是，有一点需要注意的是，soul-web 和 soul-admin 必须使用相同的同步机制。

如下图所示，soul-admin 在用户发生配置变更之后，会通过 `EventPublisher` 发出配置变更通知，由 `EventDispatcher` 处理该变更通知，然后根据配置的同步策略（http、websocket、zookeeper），将配置发送给对应的事件处理器

- 如果是 websocket 同步策略，则将变更后的数据主动推送给 soul-web，并且在网关层，会有对应的 `WebsocketCacheHandler` 处理器来处理 admin 的数据推送
- 如果是 zookeeper 同步策略，将变更数据更新到 zookeeper，而 `ZookeeperSyncCache` 会监听到 zookeeper 的数据变更，并予以处理
- 如果是 http 同步策略，soul-web 主动发起长轮询请求，默认有 90s 超时时间，如果 soul-admin 没有数据变更，则会阻塞 http 请求，如果有数据发生变更则响应变更的数据信息，如果超过 60s 仍然没有数据变更则响应空数据，网关层接到响应后，继续发起 http 请求，反复同样的请求



5.3.4 Zookeeper 同步

基于 zookeeper 的同步原理很简单，主要是依赖 zookeeper 的 watch 机制，soul-web 会监听配置的节点，soul-admin 在启动的时候，会将数据全量写入 zookeeper，后续数据发生变更时，会增量更新 zookeeper 的节点，与此同时，soul-web 会监听配置信息的节点，一旦有信息变更时，会更新本地缓存。

soul 将配置信息写到 zookeeper 节点，是通过精细设计的。

5.3.5 Websocket 同步

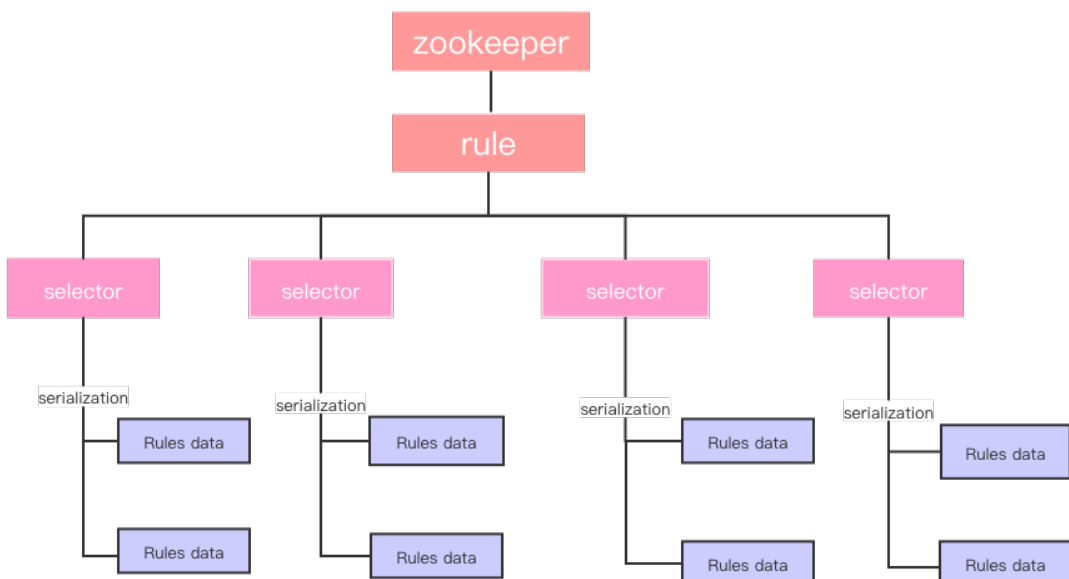
websocket 和 zookeeper 机制有点类似，将网关与 admin 建立好 websocket 连接时，admin 会推送一次全量数据，后续如果配置数据发生变更，则将增量数据通过 websocket 主动推送给 soul-web。使用 websocket 同步的时候，特别要注意断线重连，也叫保持心跳。soul 使用 java-websocket 这个第三方库来进行 websocket 连接。

```

public class WebsocketSyncCache extends WebsocketCacheHandler {
    /**
     * The Client.
     */
    private WebSocketClient client;

    public WebsocketSyncCache(final SoulConfig.WebsocketConfig websocketConfig) {
        ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(1,
            SoulThreadFactory.create("websocket-connect", true));
        client = new WebSocketClient(new URI(websocketConfig.getUrl())) {
            @Override
            public void onOpen(final ServerHandshake serverHandshake) {
                //....
            }
            @Override

```



```

        public void onMessage(final String result) {
            //....
        }
    };
    //进行连接
    client.connectBlocking();
    //使用调度线程池进行断线重连, 30 秒进行一次
    executor.scheduleAtFixedRate(() -> {
        if (client != null && client.isClosed()) {
            client.reconnectBlocking();
        }
    }, 10, 30, TimeUnit.SECONDS);
}

```

5.3.6 Http 长轮询

zookeeper、websocket 数据同步的机制比较简单，而 http 同步会相对复杂一些。Soul 借鉴了 Apollo、Nacos 的设计思想，取其精华，自己实现了 http 长轮询数据同步功能。注意，这里并非传统的 ajax 长轮询！

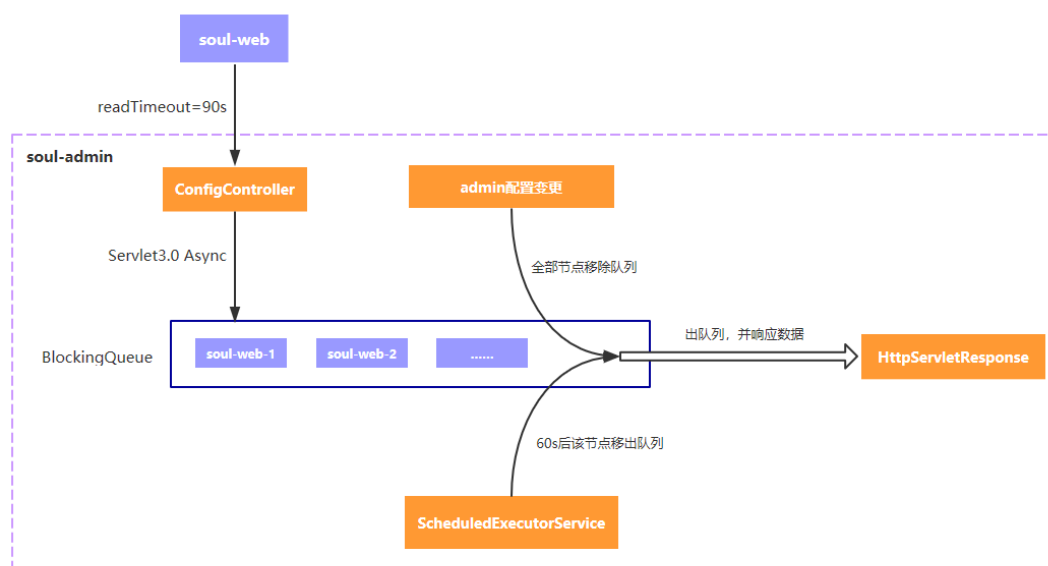


图 2: http 长轮询

http 长轮询机制如上所示，soul-web 网关请求 admin 的配置服务，读取超时时间为 90s，意味着网关层请求配置服务最多会等待 90s，这样便于 admin 配置服务及时响应变更数据，从而实现准实时推送。

http 请求到达 soul-admin 之后，并非立马响应数据，而是利用 Servlet3.0 的异步机制，异步响应数据。首先，将长轮询请求任务 LongPollingClient 扔到 BlockingQueue 中，并且开启调度任务，60s 后执行，这样做的目的是 60s 后将该长轮询请求移除队列，即便是这段时间内没有发生配置数据变更。因为即便是没有配置变更，也得让网关知道，总不能让其干等吧，而且网关请求配置服务时，也有 90s 的超时时间。

```

public void doLongPolling(final HttpServletRequest request, final
    HttpServletResponse response) {
    // 因为 soul-web 可能未收到某个配置变更的通知, 因此 MD5 值可能不一致, 则立即响应
    List<ConfigGroupEnum> changedGroup = compareMD5(request);
    String clientIp = getRemoteIp(request);
    if (CollectionUtils.isEmpty(changedGroup)) {
        this.generateResponse(response, changedGroup);
        return;
    }

    // Servlet3.0 异步响应 http 请求
    final AsyncContext asyncContext = request.startAsync();
    asyncContext.setTimeout(0L);
    scheduler.execute(new LongPollingClient(asyncContext, clientIp, 60));
}

class LongPollingClient implements Runnable {
    LongPollingClient(final AsyncContext ac, final String ip, final long
        timeoutTime) {
        // 省略.....
    }
    @Override
    public void run() {
        // 加入定时任务, 如果 60s 之内没有配置变更, 则 60s 后执行, 响应 http 请求
        this.asyncTimeoutFuture = scheduler.schedule(() -> {
            // clients 是阻塞队列, 保存了来自 soul-web 的请求信息
            clients.remove(LongPollingClient.this);
            List<ConfigGroupEnum> changedGroups =
                HttpLongPollingDataChangedListener.compareMD5((HttpServletRequest) asyncContext.
                    getRequest());
            sendResponse(changedGroups);
        }, timeoutTime, TimeUnit.MILLISECONDS);
        //
        clients.add(this);
    }
}

```

如果这段时间内, 管理员变更了配置数据, 此时, 会挨个移除队列中的长轮询请求, 并响应数据, 告知是哪个 Group 的数据发生了变更 (我们将插件、规则、流量配置、用户配置数据分成不同的组)。网关收到响应信息之后, 只知道是哪个 Group 发生了配置变更, 还需要再次请求该 Group 的配置数据。有人会问, 为什么不是直接将变更的数据写出? 我们在开发的时候, 也深入讨论过该问题, 因为 http 长轮询机制只能保证准实时, 如果在网关层处理不及时, 或者管理员频繁更新配置, 很有可能便错过了某个配置变更的推送, 安全起见, 我们只告知某个 Group 信息发生了变更。

```

// soul-admin 发生了配置变更, 挨个将队列中的请求移除, 并予以响应
class DataChangeTask implements Runnable {
    DataChangeTask(final ConfigGroupEnum groupKey) {
        this.groupKey = groupKey;
    }
}

```

```
}
@Override
public void run() {
    try {
        for (Iterator<LongPollingClient> iter = clients.iterator(); iter.
hasNext(); ) {
            LongPollingClient client = iter.next();
            iter.remove();
            client.sendResponse(Collections.singletonList(groupKey));
        }
    } catch (Throwable e) {
        LOGGER.error("data change error.", e);
    }
}
}
```

当 soul-web 网关层接收到 http 响应信息之后，拉取变更信息（如果有变更的话），然后再次请求 soul-admin 的配置服务，如此反复循环。

5.3.7 仓库地址

github: <https://github.com/Dromara/soul>

gitee: <https://gitee.com/dromara/soul>

项目主页上还有视频教程，有需要的朋友可以去观看。

5.3.8 最后

此文介绍了 soul 作为一个高可用的微服务网关，为了优化响应速度，在对配置、规则、选择器数据进行本地缓存的三种方式，学了此文，我相信你对现在比较流行的配置中心有了一定的了解，看他们的代码也许会变得容易，我相信你也可以自己写一个分布式配置中心出来。3.0 版本已经在规划中，肯定会给大家带来惊喜。

5.4 元数据概念设计

5.4.1 说明

- 本篇主要讲解在 soul 网关中元数据的概念，设计，以及如何对接。

5.4.2 技术方案

- 在数据库中，新增了一张表，然后通过数据同步的方案，会把这张表的数据同步到网关 JVM 内存。
- 表结构如下：

```
CREATE TABLE IF NOT EXISTS `meta_data` (
  `id` varchar(128) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL
  COMMENT 'id',
  `app_name` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL
  COMMENT '应用名称',
  `path` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL
  COMMENT '路径，不能重复',
  `path_desc` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT
  NULL COMMENT '路径描述',
  `rpc_type` varchar(64) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL
  COMMENT 'rpc 类型',
  `service_name` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NULL
  DEFAULT NULL COMMENT '服务名称',
  `method_name` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NULL
  DEFAULT NULL COMMENT '方法名称',
  `parameter_types` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci
  NULL DEFAULT NULL COMMENT '参数类型 多个参数类型 逗号隔开',
  `rpc_ext` varchar(1024) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NULL
  DEFAULT NULL COMMENT 'rpc 的扩展信息，json 格式',
  `date_created` datetime(0) NOT NULL COMMENT '创建时间',
  `date_updated` datetime(0) NOT NULL ON UPDATE CURRENT_TIMESTAMP(0) COMMENT '更新时
  间',
  `enabled` tinyint(4) NOT NULL DEFAULT 0 COMMENT '启用状态',
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_unicode_ci ROW_FORMAT =
Dynamic;
```

- 元数据设计，目前最主要的是对 dubbo 的泛化调用上进行使用。
- 我重点讲一下 path 字段，在请求网关的时候，会根据你的 path 字段来匹配到一条数据，然后进行后续的流程。
- 重点讲一下 rpc_ext 字段，如果是 dubbo 类型的服务接口，如果服务接口设置了 group 和 version 字段的时候，会存在这个字段。
- dubbo 类型字段结构是如下，那么存储的就是 json 格式的字符串。

```
public static class RpcExt {
  private String group;
  private String version;
  private String loadbalance;
  private Integer retries;
  private Integer timeout;
}
```

5.4.3 元数据存储

- 每个 dubbo 接口方法，对应一条元数据。
- springcloud 协议，只会存储一条数据，path 为 /contextPath/**。
- http 服务，则不会有任何数据。

6.1 字典管理

6.1.1 说明

- 字典管理主要用来维护和管理公用数据字典

6.1.2 表设计

- sql

```
CREATE TABLE IF NOT EXISTS `soul_dict` (  
  `id` varchar(128) COLLATE utf8mb4_unicode_ci NOT NULL COMMENT '主键 id',  
  `type` varchar(100) COLLATE utf8mb4_unicode_ci NOT NULL COMMENT '类型',  
  `dict_code` varchar(100) COLLATE utf8mb4_unicode_ci NOT NULL COMMENT '字典编码',  
  `dict_name` varchar(100) COLLATE utf8mb4_unicode_ci NOT NULL COMMENT '字典名称',  
  `dict_value` varchar(100) COLLATE utf8mb4_unicode_ci DEFAULT NULL COMMENT '字典值',  
  `desc` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL COMMENT '字典描述或备注',  
  `sort` int(4) NOT NULL COMMENT '排序',  
  `enabled` tinyint(4) DEFAULT NULL COMMENT '是否开启',  
  `date_created` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',  
  `date_updated` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '更新时间',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

- 目前使用场景是插件处理配置 data_type=3 下拉框时使用

假如 sentinel 插件处理字段中的 degradeRuleGrade

那么新增规则时，编辑 degradeRuleGrade 字段时会自动从字典表查出 type=degradeRuleGrade 的所有字典作为下拉选项。

6.2 插件处理详解

6.2.1 说明

- 在 soul-admin 后台，每个插件都用 handle（json 格式）字段来表示不同的处理，而插件处理就是用来管理编辑 json 里面的自定义处理字段。
- 该功能主要是用来支持插件处理模板化配置的

表设计

- sql

```
CREATE TABLE IF NOT EXISTS `plugin_handle` (
  `id` varchar(128) NOT NULL,
  `plugin_id` varchar(128) NOT NULL COMMENT '插件 id',
  `field` varchar(100) NOT NULL COMMENT '字段',
  `label` varchar(100) DEFAULT NULL COMMENT '标签',
  `data_type` smallint(6) NOT NULL DEFAULT '1' COMMENT '数据类型 1 数字 2 字符串 3 下拉框',
  `type` smallint(6) NULL COMMENT '类型,1 表示选择器, 2 表示规则',
  `sort` int(4) NULL COMMENT '排序',
  `ext_obj` varchar(1024) DEFAULT NULL COMMENT '额外配置 (json 格式数据)',
  `date_created` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
  `date_updated` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '更新时间',
  PRIMARY KEY (`id`),
  UNIQUE KEY `plugin_id_field_type` (`plugin_id`,`field`,`type`)
) ENGINE=InnoDB;
```

使用教程

比如开发 springCloud 插件时规则表需要存一些配置到 handle 字段，配置对应的实体类如下：

```
public class SpringCloudRuleHandle implements RuleHandle {

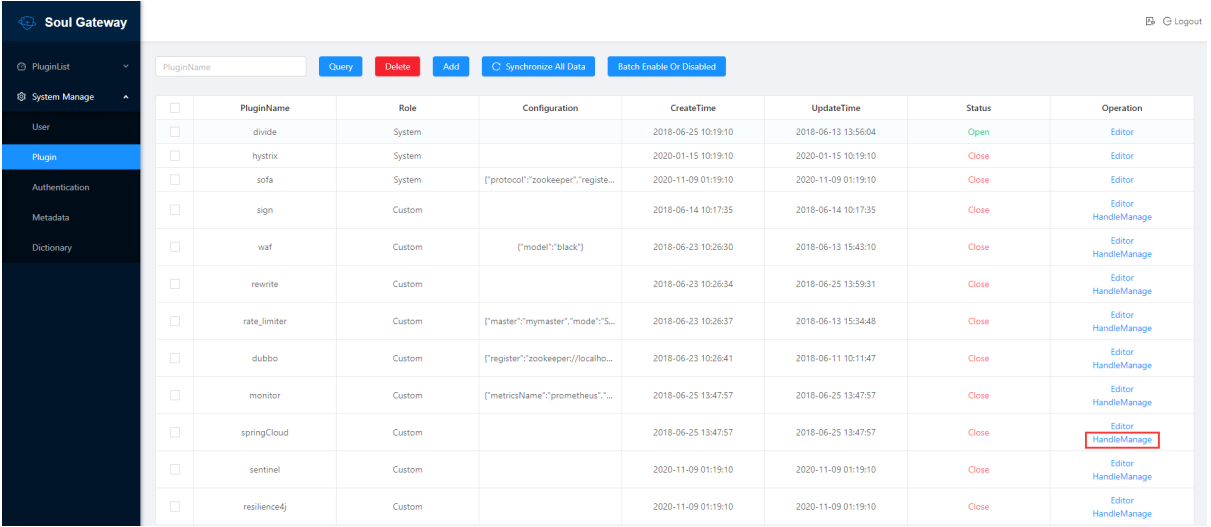
    /**
     * this remote uri path.
     */
    private String path;

    /**
     * timeout is required.
     */
    private long timeout = Constants.TIME_OUT;

}
```

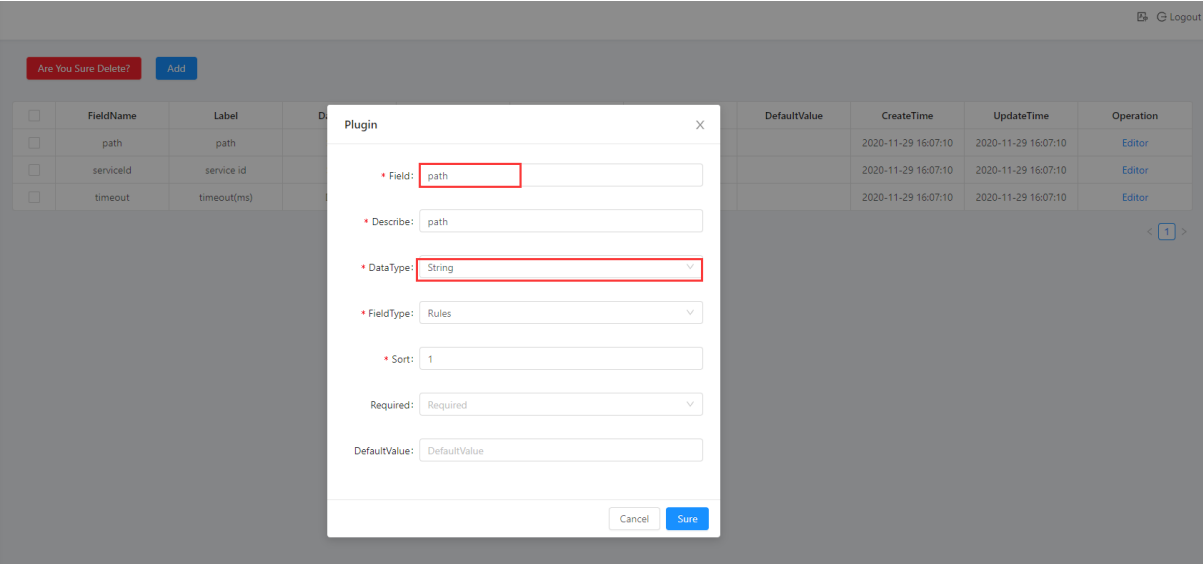
第一步、我们可以直接在插件管理界面 <http://localhost:9095/#/system/plugin> 点击编辑插件

处理



PluginName	Role	Configuration	CreateTime	UpdateTime	Status	Operation
divide	System		2018-06-25 10:19:10	2018-06-13 13:56:04	Open	Editor
hystrix	System		2020-01-15 10:19:10	2020-01-15 10:19:10	Close	Editor
sofa	System	["protocol":"zookeeper","regist...	2020-11-09 01:19:10	2020-11-09 01:19:10	Close	Editor
sign	Custom		2018-06-14 10:17:35	2018-06-14 10:17:35	Close	Editor HandleManage
waf	Custom	["model":"black"]	2018-06-23 10:26:30	2018-06-13 15:43:10	Close	Editor HandleManage
rewrite	Custom		2018-06-23 10:26:34	2018-06-25 13:59:31	Close	Editor HandleManage
rate_limiter	Custom	["master":"mymaster","mode":"S...	2018-06-23 10:26:37	2018-06-13 15:34:48	Close	Editor HandleManage
dubbo	Custom	["register":"zookeeper/localho...	2018-06-23 10:26:41	2018-06-11 10:11:47	Close	Editor HandleManage
monitor	Custom	["metricsName":"prometheus","...	2018-06-25 13:47:57	2018-06-25 13:47:57	Close	Editor HandleManage
springCloud	Custom		2018-06-25 13:47:57	2018-06-25 13:47:57	Close	Editor HandleManage
sentinel	Custom		2020-11-09 01:19:10	2020-11-09 01:19:10	Close	Editor HandleManage
resilience4j	Custom		2020-11-09 01:19:10	2020-11-09 01:19:10	Close	Editor HandleManage

第二步、新增一个字符串类型字段 `path` 和一个数字类型的 `timeout`



Are You Sure Delete? Add

Field	Label	DefaultValue	CreateTime	UpdateTime	Operation
path	path		2020-11-29 16:07:10	2020-11-29 16:07:10	Editor
serviceId	service id		2020-11-29 16:07:10	2020-11-29 16:07:10	Editor
timeout	timeout(ms)		2020-11-29 16:07:10	2020-11-29 16:07:10	Editor

Plugin

* Field: path

* Describe: path

* DataType: String

* FieldType: Rules

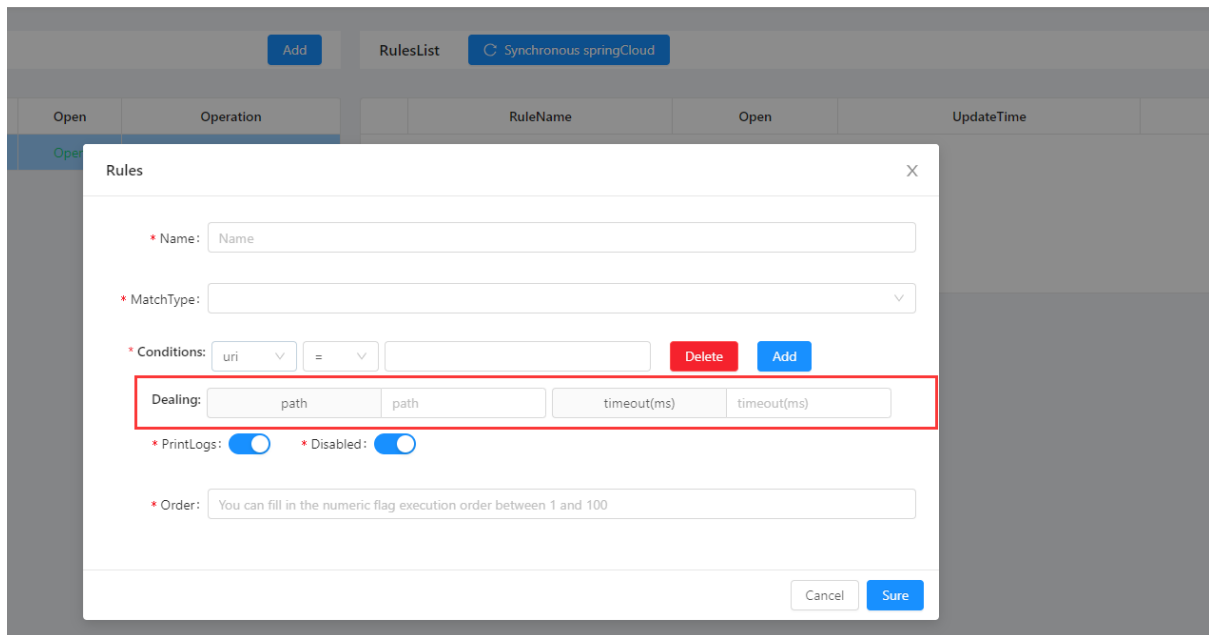
* Sort: 1

Required: Required

DefaultValue: DefaultValue

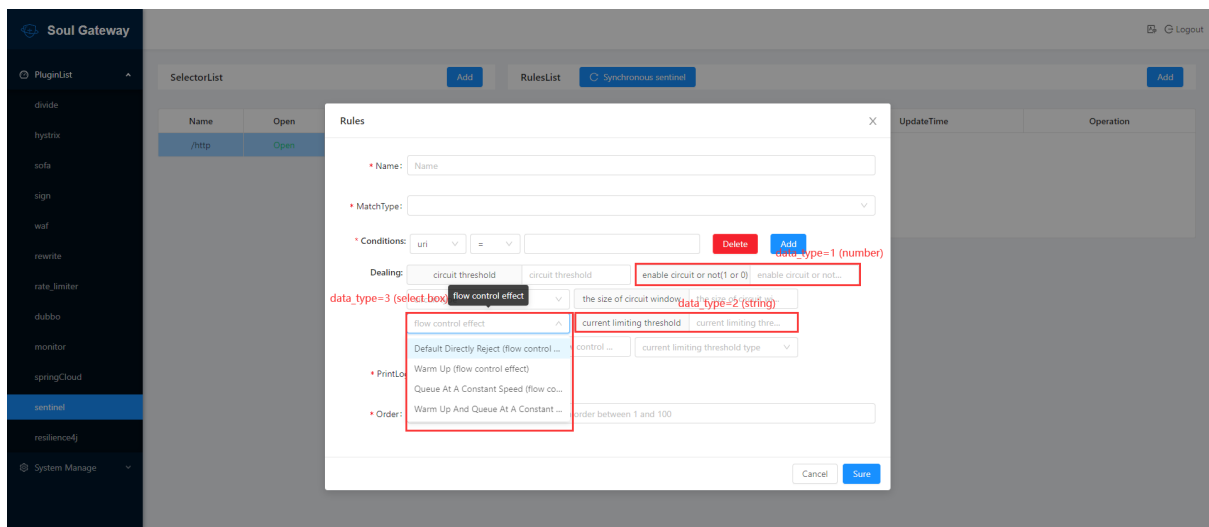
Cancel Sure

第三步、在插件规则配置页面新增规则时就可以直接输入 `path`、`timeout` 然后提交保存到 `handle` 字段了



注意：如果配置了 `data_type` 为 3 选择框，则规则新增页面里输入框下拉选择是通过 `field` 字段去 `*字典表 (soul_dict)` `<soul-dict.md>‘__*’` 查出所有可选项出来展示选择。

- 比如 sentinel 插件包含多种数据类型的字段，如下图：



6.3 选择器规则详解

6.3.1 说明

- 选择器和规则是 soul 网关中最灵魂的东西。掌握好它，你可以对任何流量进行管理。
- 本篇主要详解 soul 网关中，选择器与规则的概念，以及如何使用。

6.3.2 大体理解

- 一个插件有多个选择器，一个选择器对应多种规则。选择器相当于是对流量的一级筛选，规则就是最终的筛选。
- 我们想象一下，在一个插件里面，我们是不是希望根据我们的配置，达到满足条件的流量，我们插件才去执行它？
- 选择器和规则就是为了让流量在满足特定的条件下，才去执行我们想要的，这种规则我们首先要明白。
- 数据结构可以参考之前的 [数据库设计](#)

6.3.3 选择器

选择器

* 名称:

* 类型:

自定义流量

▼

* 匹配方式:

and

▼

* 条件:

uri

▼

match

▼

删除

新增

* 继续后续选择器:

☒

* 打印日志:

☒

* 是否开启:

☒

* http配置:

localhost

http://

1.1.1.1:8080

50

删除

新增

* 执行顺序:

取消

确定

- 选择器详解：
 - 名称：为你的选择器起一个容易分辨的名字
 - 类型：custom flow 是自定义流量。full flow 是全流量。自定义流量就是请求会走你下面的匹配方式与条件。全流量则不走。
 - 匹配方式：and 或者 or 是指下面多个条件是按照 and 还是 or 的方式来组合。

- 条件:

- * uri: 是指你根据 uri 的方式来筛选流量, match 的方式支持模糊匹配 (/**)
- * header: 是指根据请求头里面的字段来筛选流量。
- * query: 是指根据 uri 的查询条件来进行筛选流量。
- * ip: 是指根据你请求的真实 ip, 来筛选流量。
- * host: 是指根据你请求的真实 host, 来筛选流量。
- * post: 建议不要使用。
- * 条件匹配:
 - match: 模糊匹配, 建议和 uri 条件搭配, 支持 restful 风格的匹配。(/test/**)
 - =: 前后值相等, 才能匹配。
 - regEx: 正则匹配, 表示前面一个值去匹配后面的正则表达式。
 - like: 字符串模糊匹配。

- 是否开启: 打开才会生效

- 打印日志: 打开的时候, 当匹配上的时候, 会打印匹配日志。

- 执行顺序: 当多个选择器的时候, 执行顺序小的优先执行。

- 上述图片中表示: 当请求的 uri 前缀是 /test, 并且 header 头上 module 字段值为 test 的时候, 会转发到 1.1.1.1:8080 这个服务。

- 选择器建议: 可以 uri 条件, match 前缀 (/contextPath), 进行第一道流量筛选。

6.3.4 规则

规则

×

* 名称: /http/order/save

* 匹配方式: and

* 条件: uri = /http/order/save

删除

新增

http负载: 负载策略 random 重试次数 0

* 打印日志: ☒

* 是否开启: ☒

* 执行顺序: 1

取消

确定

- 当流量经过选择器匹配成功之后, 会进入规则来进行最终的流量匹配。

- 规则是对流量最终执行逻辑的确认。
- 规则详解：
 - 名称：为你的规则起一个容易分辨的名字
 - 匹配方式：and 或者 or 是指下面多个条件是按照 and 还是 or。
 - 条件：
 - * uri：是指你根据 uri 的方式来筛选流量，match 的方式支持模糊匹配 (/**)
 - * header：是指根据请求头里面的字段来筛选流量。
 - * query：是指根据 uri 的查询条件来进行筛选流量。
 - * ip：是指根据你请求的真实 ip，来筛选流量。
 - * host：是指根据你请求的真实 host，来筛选流量。
 - * post：建议不要使用。
 - * 条件匹配：
 - match：模糊匹配，建议和 uri 条件搭配，支持 restful 风格的匹配。(/test/**)
 - =：前后值相等，才能匹配。
 - regEx：正则匹配，表示前面一个值去匹配后面的正则表达式。
 - like：字符串模糊匹配。
 - 是否开启：打开才会生效。
 - 打印日志：打开的时候，当匹配上的时候，会打印匹配日志。
 - 执行顺序：当多个规则的时候，执行顺序小的优先执行。
 - 处理：每个插件的规则处理不一样，具体的差有具体的处理，具体请查看每个对应插件的处理。
- 上图表示：当 uri 等于 /http/order/save 的时候该规则被匹配，就会执行该规则中，负载策略是 random。
- 联合选择器，我们来表述一下：当一个请求的 uri 为 /http/order/save，会通过 random 的方式，转发到 1.1.1.1:8080。
- 规则建议：可以 uri 条件，match 最真实的 uri 路径，进行流量的最终筛选。

6.3.5 条件详解

- uri 匹配（推荐）
 - uri 匹配是根据你请求路径中的 uri 来进行匹配，在接入网关的时候，前端几乎不用做任何更改。
 - 当使用 match 方式匹配时候，同 springmvc 模糊匹配原理相同。
 - 在选择器中，推荐使用 uri 中的前缀来进行匹配，而在规则中，则使用具体路径来进行匹配。
 - 该匹配方式的时候，在匹配字段名称可以任意填写，匹配字段值需要正确填写。

- header 匹配
 - header 是根据你的 http 请求头中的字段值来匹配。
- query 匹配
 - 这个是根据你的 uri 中的查询参数来进行匹配, 比如 /test?a=1&&b=2, 那么可以选择该匹配方式。
 - 上述就可以新增一个条件, 选取 query 方式, a = 1。
- ip 匹配
 - 这个是根据 http 调用方的 ip 来进行匹配。
 - 尤其是在 waf 插件里面, 如果发现一个 ip 地址有攻击, 可以新增一条匹配条件, 填上该 ip, 拒绝该 ip 的访问。
 - 如果在 soul 前面使用了 nginx 代理, 为了获取正确的 ip, 你可能要参考 [parsing-ip-and-host](#)
- host 匹配
 - 这个是根据 http 调用方的 host 来进行匹配。
 - 尤其是在 waf 插件里面, 如果发现一个 host 地址有攻击, 可以新增一条匹配条件, 填上该 host, 拒绝该 host 的访问。
 - 如果在 soul 前面使用了 nginx 代理, 为了获取正确的 host, 你可能要参考 [parsing-ip-and-host](#)
- post 匹配
 - 不推荐使用。

7.1 环境搭建

7.1.1 说明

- soul 2.2.0 以后都是基于插件化可插拔的思想，本文是说明如何基于 soul 搭建属于你自己网关。
- 请确保你的机器安装了 JDK 1.8+，Mysql 5.0+。

7.1.2 启动 Soul-Admin

远程下载

- 2.3.0 下载 soul-admin-bin-2.3.0-RELEASE.tar.gz
- 解压缩 soul-admin-bin-2.3.0-RELEASE.tar.gz。进入 bin 目录。
- 使用 h2 来存储后台数据

```
> windwos : start.bat --spring.profiles.active = h2  
> linux : ./start.sh --spring.profiles.active = h2
```

- 使用 mysql 来存储后台数据。进入 /conf 目录，修改 application.yaml 中 mysql 的配置。

```
> windwos : start.bat  
> linux : ./start.sh
```


docker 构建

```
> docker pull dromara/soul-admin
> docker network create soul
```

- 使用 h2 来存储后台数据

```
> docker run -d -p 9095:9095 --net soul dromara/soul-admin
```

- 使用 mysql 来存储后台数据。

```
docker run -e "SPRING_PROFILES_ACTIVE=mysql" -d -p 9095:9095 --net soul dromara/soul-admin
```

如果你想覆盖环境变量，你可以这样操作。

```
docker run -e "SPRING_PROFILES_ACTIVE=mysql" -e "spring.datasource.url=jdbc:mysql://192.168.1.9:3306/soul?useUnicode=true&characterEncoding=utf-8&useSSL=false" -e "spring.datasource.password=123456" -d -p 9095:9095 --net soul dromara/soul-admin
```

另外一种方式，可以挂载你本地磁盘其他目录

把你的 application.yml 配置放到 xxx 目录，然后执行以下语句。

```
docker run -v D:\tmp\conf:/opt/soul-admin/conf/ -d -p 9095:9095 --net soul dromara/soul-admin
```

本地构建

- 下载代码

```
> git clone https://github.com/dromara/soul.git
> cd soul
```

- 编译代码

```
> mvn clean install -Dmaven.javadoc.skip=true -B -Drat.skip=true -Djacoco.skip=true -DskipITs -DskipTests
```

- 启动 SoulAdminBootstrap。
 - 如果使用 h2 来存储，设置变量 `--spring.profiles.active = h2`
 - 如果使用 mysql 来存储，修改 application.yaml 中的 mysql 配置。

访问 <http://localhost:9095> 用户名密码为: admin/123456

7.1.3 启动 Soul-Bootstrap

远程下载

- 2.3.0 下载 soul-bootstrap-bin-2.3.0-RELEASE.tar.gz
- 解压缩 soul-bootstrap-bin-2.3.0-RELEASE.tar.gz。进入 bin 目录。

```
> windwos : start.bat  
> linux : ./start.sh
```

docker 构建

```
> docker network create soul  
> docker pull dromara/soul-bootstrap  
> docker run -d -p 9195:9195 --net soul dromara/soul-bootstrap
```

本地构建

- 下载代码

```
> git clone https://github.com/dromara/soul.git  
> cd soul
```

- 编译代码

```
> mvn clean install -Dmaven.javadoc.skip=true -B -Drat.skip=true -Djacoco.skip=true  
-DskipITs -DskipTests
```

- 启动 SoulBootstrap。

7.1.4 搭建自己的网关（推荐）

- 首先你新建一个空的 springboot 项目，可以参考 soul-bootstrap. 也可以在 spring 官网:[<https://spring.io/quickstart>]
- 引入如下 jar 包：

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-webflux</artifactId>  
  <version>2.2.2.RELEASE</version>  
</dependency>  
  
<dependency>  
  <groupId>org.springframework.boot</groupId>
```

```

        <artifactId>spring-boot-starter-actuator</artifactId>
        <version>2.2.2.RELEASE</version>
</dependency>

<!--soul gateway start-->
<dependency>
    <groupId>org.dromara</groupId>
    <artifactId>soul-spring-boot-starter-gateway</artifactId>
    <version>${last.version}</version>
</dependency>

<!--soul data sync start use websocket-->
<dependency>
    <groupId>org.dromara</groupId>
    <artifactId>soul-spring-boot-starter-sync-data-websocket</artifactId>
    <version>${last.version}</version>
</dependency>

```

- 在你的 application.yaml 文件中加上如下配置：

```

spring:
  main:
    allow-bean-definition-overriding: true

management:
  health:
    defaults:
      enabled: false
soul :
  sync:
    websocket :
      urls: ws://localhost:9095/websocket //设置成你的 soul-admin 地址

```

- 你的项目环境搭建完成，启动你的项目。

7.2 Http 用户

7.2.1 说明

- 本文旨在帮助 http 用户。
- soul 网关使用 divide 插件来处理 http 请求。请求在 soul-admin 后台开启它。
- 接入前，请正确的启动 soul-admin，以及 搭建环境 OK。

7.2.2 引入网关对 http 的代理插件

- 在网关的 pom.xml 文件中增加如下依赖：

```
<!--if you use http proxy start this-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-divide</artifactId>
  <version>${last.version}</version>
</dependency>

<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-httpclient</artifactId>
  <version>${last.version}</version>
</dependency>
```

- 当然是要重新启动网关。

7.2.3 Http 请求接入网关（springMvc 体系用户）

- 首先要确保在 soul-admin 后台 divide 插件是否开启。

Soul-Client 接入方式。（此方式针对 SpringMvc, SpringBoot 用户）

- SpringBoot 用户
 - 在你的真实服务的 pom.xml 新增如下依赖：

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-client-springmvc</artifactId>
  <version>${last.version}</version>
</dependency>
```

- 注册中心详细接入配置请参考：[注册中心接入](#)。

- SpringMvc 用户
 - 在你的真实服务的 pom.xml 新增如下依赖：

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-client-springmvc</artifactId>
  <version>${last.version}</version>
</dependency>
```

- 在你的 bean 定义的 xml 文件中新增如下：

```

<bean id="springMvcClientBeanPostProcessor" class="org.dromara.soul.
client.springmvc.init.SpringMvcClientBeanPostProcessor">
    <constructor-arg ref="soulRegisterCenterConfig"/>
</bean>

<bean id="soulRegisterCenterConfig" class="org.dromara.soul.register.
common.config.SoulRegisterCenterConfig">
    <property name="registerType" value="http"/>
    <property name="serverList" value="http://localhost:9095"/>
    <property name="props">
        <map>
            <entry key="contextPath" value="/你的 contextPath"/>
            <entry key="appName" value=" 你的名字"/>
            <entry key="port" value=" 你的端口"/>
            <entry key="isFull" value="false"/>
        </map>
    </property>
</bean>

```

- 在你的 controller 的接口上加上 @SoulSpringMvcClient 注解。
 - 你可以把注解加到 Controller 类上面，里面的 path 属性则为前缀，如果含有 /** 代表你的整个接口需要被网关代理。
 - 举例子 (1)：代表 /test/payment, /test/findById 都会被网关代理。

```

@RestController
@RequestMapping("/test")
@SoulSpringMvcClient(path = "/test/**")
public class HttpTestController {

    @PostMapping("/payment")
    public UserDTO post(@RequestBody final UserDTO userDTO) {
        return userDTO;
    }

    @GetMapping("/findById")
    public UserDTO findById(@RequestParam("userId") final String userId) {
        UserDTO userDTO = new UserDTO();
        userDTO.setUserId(userId);
        userDTO.setUserName("hello world");
        return userDTO;
    }
}

```

- 举例子 (2)：代表 /order/save, 会被网关代理，而/order/findById 则不会。

```

@RestController
@RequestMapping("/order")

```

```

@SoulSpringMvcClient(path = "/order")
public class OrderController {

    @PostMapping("/save")
    @SoulSpringMvcClient(path = "/save")
    public OrderDTO save(@RequestBody final OrderDTO orderDTO) {
        orderDTO.setName("hello world save order");
        return orderDTO;
    }

    @GetMapping("/findById")
    public OrderDTO findById(@RequestParam("id") final String id) {
        OrderDTO orderDTO = new OrderDTO();
        orderDTO.setId(id);
        orderDTO.setName("hello world findById");
        return orderDTO;
    }
}

```

- 启动你的项目，你的接口接入到了网关。

7.2.4 Http 请求接入网关（其他语言，非 springMvc 体系）

- 首先在 soul-admin 找到 divide 插件，进行选择器，和规则的添加，进行流量的匹配筛选。
- 如果不懂怎么配置，请看选择，规则介绍 [选择器规则介绍](#)。
- 您也可以自定义开发属于你的 http-client，参考 [多语言 Http 客户端开发](#)。

7.2.5 用户请求

- 说白了，你之前怎么请求就怎么请求，没有很大的变动，变动的地方有 2 点。
- 第一点，你之前请求的域名是你自己的服务，现在要换成网关的域名（这个你听的懂？）
- 第二点，soul 网关需要有一个路由前缀，这个路由前缀就是你接入项目进行配置 contextPath，如果熟的话，可以自由在 soul-admin 中的 divide 插件进行自由更改。

```

# 比如你有一个 order 服务 它有一个接口，请求路径 http://localhost:8080/test/save

# 现在就需要换成：http://localhost:9195/order/test/save

# 其中 localhost:9195 为网关的 ip 端口，默认端口是 9195，/order 是你接入网关配置的 contextPath

# 其他参数，请求方式不变。

# 我讲到这里还不懂？ 请加群问吧

```

- 然后你就可以进行访问了，如此的方便与简单。

7.3 Dubbo 接入 soul 网关

7.3.1 说明

- 此篇文章是 dubbo 用户使用 dubbo 插件支持，以及自己的 dubbo 服务接入 soul 网关的教程。
- 支持 alibaba dubbo (< 2.7.x) 以及 apache dubbo (>=2.7.x)。
- 接入前，请正确的启动 soul-admin，以及搭建环境 Ok。

7.3.2 引入网关对 dubbo 支持的插件

- 在网关的 pom.xml 文件中增加如下依赖：
 - alibaba dubbo 用户, dubbo 版本换成你的，注册中心的 jar 包换成你的，以下是参考。

```

<!--soul alibaba dubbo plugin start-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-alibaba-dubbo</artifactId>
  <version>${last.version}</version>
</dependency>
<!-- soul alibaba dubbo plugin end-->
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>dubbo</artifactId>
  <version>2.6.5</version>
</dependency>
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-client</artifactId>
  <version>4.0.1</version>
</dependency>
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-framework</artifactId>
  <version>4.0.1</version>
</dependency>
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-recipes</artifactId>
  <version>4.0.1</version>
</dependency>

```

- apache dubbo 用户, dubbo 版本换成你的，使用什么注册中心换成你的，以下是参考，使用什么注册中心，就引入啥。

```

<!--soul apache dubbo plugin start-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-apache-dubbo</artifactId>
  <version>${last.version}</version>
</dependency>
<!--soul apache dubbo plugin end-->

<dependency>
  <groupId>org.apache.dubbo</groupId>
  <artifactId>dubbo</artifactId>
  <version>2.7.5</version>
</dependency>
<!-- Dubbo Nacos registry dependency start -->
<dependency>
  <groupId>org.apache.dubbo</groupId>
  <artifactId>dubbo-registry-nacos</artifactId>
  <version>2.7.5</version>
</dependency>
<dependency>
  <groupId>com.alibaba.nacos</groupId>
  <artifactId>nacos-client</artifactId>
  <version>1.1.4</version>
</dependency>
<!-- Dubbo Nacos registry dependency end-->

<!-- Dubbo zookeeper registry dependency start-->
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-client</artifactId>
  <version>4.0.1</version>
</dependency>
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-framework</artifactId>
  <version>4.0.1</version>
</dependency>
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-recipes</artifactId>
  <version>4.0.1</version>
</dependency>
<!-- Dubbo zookeeper registry dependency end -->

```

- 重启网关服务。

7.3.3 dubbo 服务接入网关，可以参考：soul-examples-dubbo

- alibaba dubbo 用户

- springboot

- * 引入以下依赖

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-client-alibaba-dubbo</artifactId>
  <version>${last.version}</version>
</dependency>
```

- * 注册中心详细接入配置请参考：[注册中心接入](#)。

- spring

- * 引入以下依赖：

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-client-alibaba-dubbo</artifactId>
  <version>${last.version}</version>
</dependency>
```

- * 在你的 bean 定义的 xml 文件中新增如下：

```
<bean id="alibabaDubboServiceBeanPostProcessor" class="org.dromara.soul.
client.alibaba.dubbo.AlibabaDubboServiceBeanPostProcessor">
  <constructor-arg ref="soulRegisterCenterConfig"/>
</bean>

<bean id="soulRegisterCenterConfig" class="org.dromara.soul.register.
common.config.SoulRegisterCenterConfig">
  <property name="registerType" value="http"/>
  <property name="serverList" value="http://localhost:9095"/>
  <property name="props">
    <map>
      <entry key="contextPath" value="/你的 contextPath"/>
      <entry key="appName" value=" 你的名字"/>
      <entry key="ifFull" value="false"/>
    </map>
  </property>
</bean>
```

- apache dubbo 用户

- springboot

- * 引入以下依赖

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-client-apache-dubbo</artifactId>
  <version>${last.version}</version>
</dependency>
```

* 注册中心详细接入配置请参考：[注册中心配置](#)。

- spring

* 引入以下依赖：

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-client-apache-dubbo</artifactId>
  <version>${last.version}</version>
</dependency>
```

* 在你的 bean 定义的 xml 文件中新增如下：

```
<bean id="apacheDubboServiceBeanPostProcessor" class="org.dromara.
soul.client.apache.dubbo.ApacheDubboServiceBeanPostProcessor">
  <constructor-arg ref="soulRegisterCenterConfig"/>
</bean>

<bean id="soulRegisterCenterConfig" class="org.dromara.soul.register.
common.config.SoulRegisterCenterConfig">
  <property name="registerType" value="http"/>
  <property name="serverList" value="http://localhost:9095"/>
  <property name="props">
    <map>
      <entry key="contextPath" value="/你的 contextPath"/>
      <entry key="appName" value=" 你的名字"/>
      <entry key="ifFull" value="false"/>
    </map>
  </property>
</bean>
```

7.3.4 dubbo 插件设置

- 首先在 soul-admin 插件管理中，把 dubbo 插件设置为开启。
- 其次在 dubbo 插件中配置你的注册地址，或者其他注册中心的地址。

```
{"register":"zookeeper://localhost:2181"}    or {"register":"nacos://localhost:8848"
"}
```

7.3.5 接口注册到网关

- 你 dubbo 服务实现类的，方法上加上 @SoulDubboClient 注解，表示该接口方法注册到网关。
- 启动你的提供者，输出日志 `dubbo client register success` 大功告成，你的 dubbo 接口已经发布到 soul 网关. 如果还有不懂的，可以参考 `soul-test-dubbo` 项目。

7.3.6 dubbo 用户请求以及参数说明

- 说白了，就是通过 http 的方式来请求你的 dubbo 服务
- soul 网关需要有一个路由前缀，这个路由前缀就是你接入项目进行配置 `contextPath`

```
# 比如你有一个 order 服务 它有一个接口，它的注册路径 /order/test/save

# 现在就是通过 post 方式请求网关: http://localhost:9195/order/test/save

# 其中 localhost:9195 为网关的 ip 端口，默认端口是 9195，/order 是你 dubbo 接入网关配置的 contextPath
```

- 参数传递：
 - 通过 http post 方式访问网关，通过 body，json 类型传递。
 - 更多参数类型传递，可以参考 `soul-examples-dubbo` 中的接口定义，以及参数传递方式。
- 单个 java bean 参数类型（默认）
- 多参数类型支持，在网关的 yaml 配置中新增如下配置：

```
soul :
  dubbo :
    parameter: multi
```

- 自定义实现多参数支持：
 - 在你搭建的网关项目中，新增一个类 A，实现 `org.dromara.soul.web.dubbo.DubboParamResolveService`。

```
public interface DubboParamResolveService {

    /**
     * Build parameter pair.
     * this is Resolve http body to get dubbo param.
     *
     * @param body          the body
     * @param parameterTypes the parameter types
     * @return the pair
     */
    Pair<String[], Object[]> buildParameter(String body, String parameterTypes);
}
```

- body 为 http 中 body 传的 json 字符串。
- parameterTypes: 匹配到的方法参数类型列表, 如果有多个, 则使用, 分割。
- Pair 中, left 为参数类型, right 为参数值, 这是 dubbo 泛化调用的标准
- 把你的类注册成 Spring 的 bean, 覆盖默认的实现。

```
@Bean
public DubboParamResolveService A() {
    return new A();
}
```

7.3.7 服务治理

- 标签路由
 - 请求时在 header 中添加 Dubbo_Tag_Route, 并设置对应的值, 之后当前请求就会路由到指定 tag 的 provider, 只对当前请求有效;
- 服务提供者直连
 - 设置 @SoulDubboClient 注解中的 url 属性;
 - 修改 Admin 控制台修改元数据内的 url 属性;
 - 对所有请求有效;
- 参数验证和自定义异常
 - 指定 validation="soulValidation";
 - 在接口中抛出 SoulException 时, 异常信息会返回, 需要注意的是显式抛出 SoulException;

```
@Service(validation = "soulValidation")
public class TestServiceImpl implements TestService {

    @Override
    @SoulDubboClient(path = "/test", desc = "test method")
    public String test(@Valid HelloServiceRequest name) throws SoulException {
        if (true){
            throw new SoulException("Param binding error.");
        }
        return "Hello " + name.getName();
    }
}
```

- 请求参数

```

public class HelloServiceRequest implements Serializable {

    private static final long serialVersionUID = -5968745817846710197L;

    @NotEmpty(message = "name cannot be empty")
    private String name;

    @NotNull(message = "age cannot be null")
    private Integer age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}

```

- 发送请求

```

{
  "name": ""
}

```

- 返回

```

{
  "code": 500,
  "message": "Internal Server Error",
  "data": "name cannot be empty,age cannot be null"
}

```

- 当按照要求传递请求参数时，会返回自定义异常的信息

```

{
  "code": 500,
  "message": "Internal Server Error",
  "data": "Param binding error."
}

```

7.3.8 大白话讲解如果通过 http -> 网关-> dubbo provider

- 说白了，就是把 http 请求，转成 dubbo 协议，内部使用 dubbo 泛化来进行调用。
- 首先你要回想下，你的 dubbo 服务在接入网关的时候，是不是加了个 @SoulDubboClient 注解，里面是不是有个 path 字段来指定你请求的路径？
- 你是不是还在 yml 中配置了一个 contextPath？
- 如果您还记得，那我们就开始。
- 假如你有一个这样的方法，contextPath 配置的是 /dubbo

```
@Override
@SoulDubboClient(path = "/insert", desc = "插入一条数据")
public DubboTest insert(final DubboTest dubboTest) {
    return dubboTest;
}
```

- 那么我们请求的路径为：http://localhost:9195/dubbo/insert，再说一下，localhost:9195 是网关的域名，如果你更改了，这里也要改。
- 那么请求参数呢？DubboTest 是一个 javabean 对象，有 2 个字段，id 与 name，那么我们通过 body 中传递这个对象的 json 数据就好。

```
{"id": "1234", "name": "XIAO5y"}
```

- 如果你的接口中，没有参数，那么 body 传值为：

```
{}
```

- 如果你的接口有很多个参数？往上看一点，有介绍。

7.4 SpringCloud 接入 Soul 网关

7.4.1 说明

- 此篇文章是教你如何将 springCloud 接口，快速接入到 soul 网关。
- 请在 soul-admin 后台将 springCloud 插件设置为开启。
- 接入前，请正确的启动 soul-admin 以及搭建环境 Ok。

7.4.2 引入网关 springCloud 的插件支持

- 在网关的 pom.xml 文件中引入如下依赖。

```
<!--soul springCloud plugin start-->
<dependency>
    <groupId>org.dromara</groupId>
    <artifactId>soul-spring-boot-starter-plugin-springcloud</artifactId>
    <version>${last.version}</version>
</dependency>

<dependency>
    <groupId>org.dromara</groupId>
    <artifactId>soul-spring-boot-starter-plugin-httpclient</artifactId>
    <version>${last.version}</version>
</dependency>
<!--soul springCloud plugin end-->

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-commons</artifactId>
    <version>2.2.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
    <version>2.2.0.RELEASE</version>
</dependency>
```

- 如果你使用 eureka 作为 springCloud 的注册中心
 - 新增如下依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>2.2.0.RELEASE</version>
</dependency>
```

- 在网关的 yml 文件中新增如下配置：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/ # 你的 eureka 地址
  instance:
    prefer-ip-address: true
```

- 如果你使用 nacos 作为 springCloud 的注册中心

- 新增如下依赖:

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
  <version>2.1.0.RELEASE</version>
</dependency>
```

- 在网关的 yml 文件中新增如下配置:

```
spring:
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848 # 你的 nacos 地址
```

- 重启你的网关服务。

7.4.3 SpringCloud 服务接入网关

- 在你提供服务的项目中, 引入如下依赖:

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-client-springcloud</artifactId>
  <version>${last.version}</version>
</dependency>
```

- 注册中心详细接入配置请参考: [注册中心接入](#)
- 在你的 controller 的接口上加上 @SoulSpringCloudClient 注解
- 你可以把注解加到 Controller 类上面, 里面的 path 属性则为前缀, 如果含有 /** 代表你的整个接口需要被网关代理
 - 举例 (1): 代表 /test/payment, /test/findByUserId 都会被网关代理。

```
@RestController
@RequestMapping("/test")
@SoulSpringCloudClient(path = "/test/**")
public class HttpTestController {

    @PostMapping("/payment")
    public UserDTO post(@RequestBody final UserDTO userDTO) {
        return userDTO;
    }

    @GetMapping("/findByUserId")
    public UserDTO findByUserId(@RequestParam("userId") final String userId) {
        UserDTO userDTO = new UserDTO();
    }
}
```



```

        userDTO.setUserId(userId);
        userDTO.setUserName("hello world");
        return userDTO;
    }
}

```

- 举例子 (2): 代表 /order/save, 会被网关代理, 而/order/findById 则不会。

```

@RestController
@RequestMapping("/order")
@SoulSpringCloudClient(path = "/order")
public class OrderController {

    @PostMapping("/save")
    @SoulSpringMvcClient(path = "/save")
    public OrderDTO save(@RequestBody final OrderDTO orderDTO) {
        orderDTO.setName("hello world save order");
        return orderDTO;
    }

    @GetMapping("/findById")
    public OrderDTO findById(@RequestParam("id") final String id) {
        OrderDTO orderDTO = new OrderDTO();
        orderDTO.setId(id);
        orderDTO.setName("hello world findById");
        return orderDTO;
    }
}

```

- 举例子 (3): isFull: true 代表 /sb-demo7-api/**, 整个服务会被网关代理

```

soul:
  client:
    registerType: http
    serverLists: http://localhost:9095
    props:
      contextPath: /http
      appName: http
      isFull: true
# registerType : 服务注册类型, 支持 http/zookeeper
# serverList: 为 http 注册类型时, 填写 Soul-Admin 项目的地址, 注意加上 http://, 多个地址用英文逗号分隔
#               为 zookeeper 注册类型时, 填写 zookeeper 地址, 多个地址用英文分隔
# contextPath: 为你的这个 mvc 项目在 soul 网关的路由前缀, 这个你应该懂意思把? 比如/order , /product 等等, 网关会根据你的这个前缀来进行路由。
# appName: 你的应用名称, 不配置的话, 会默认取 dubbo 配置中 application 中的名称
# isFull: 设置 true 代表代理你的整个服务, false 表示代理你其中某几个 controller

```

```

@RestController
@RequestMapping("/order")
public class OrderController {

    @PostMapping("/save")
    @SoulSpringMvcClient(path = "/save")
    public OrderDTO save(@RequestBody final OrderDTO orderDTO) {
        orderDTO.setName("hello world save order");
        return orderDTO;
    }

    @GetMapping("/findById")
    public OrderDTO findById(@RequestParam("id") final String id) {
        OrderDTO orderDTO = new OrderDTO();
        orderDTO.setId(id);
        orderDTO.setName("hello world findById");
        return orderDTO;
    }
}

```

- 启动你的服务，如果输出以下日志：http client register success，证明你的接口已经被注册到 soul 网关。

7.4.4 插件设置

- 在 soul-admin 插件管理中，把 springCloud 插件设置为开启。

7.4.5 用户请求

- 说白了，你之前怎么请求就怎么请求，没有很大的变动，变动的地方有 2 点。
- 第一点，你之前请求的域名是你自己的服务，现在要换成网关的域名（这个你听得懂？）
- 第二点，soul 网关需要有一个路由前缀，这个路由前缀就是你接入项目进行配置 contextPath，如果熟的话，可以自由在 soul-admin 中的 springCloud 插件进行自由更改。

```

# 比如你有一个 order 服务 它有一个接口，请求路径 http://localhost:8080/test/save

# 现在就需要换成：http://localhost:9195/order/test/save

# 其中 localhost:9195 为网关的 ip 端口，默认端口是 9195，/order 是你接入网关配置的 contextPath

# 其他参数，请求方式不变。

# 我讲到这里还不懂？ 请加群问吧

```

- 然后你就可以进行访问了，如此的方便与简单。

7.5 Sofa 接入网关

7.5.1 说明

- 此篇文章是 sofa 用户使用 sofa 插件支持，以及自己的 sofa 服务接入 soul 网关的教程。
- 接入前，请正确的启动 soul-admin 以及搭建环境 Ok。

7.5.2 引入网关对 sofa 支持的插件

- 在网关的 pom.xml 文件中增加如下依赖：
- sofa 版本换成你的，注册中心的 jar 包换成你的，以下是参考。

```
<dependency>
  <groupId>com.alipay.sofa</groupId>
  <artifactId>sofa-rpc-all</artifactId>
  <version>5.7.6</version>
</dependency>
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-client</artifactId>
  <version>4.0.1</version>
</dependency>
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-framework</artifactId>
  <version>4.0.1</version>
</dependency>
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-recipes</artifactId>
  <version>4.0.1</version>
</dependency>
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-sofa</artifactId>
  <version>${last.version}</version>
</dependency>
```

- 重启网关服务。

7.5.3 sofa 服务接入网关，可以参考：soul-examples-sofa

- springboot
 - 引入以下依赖

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-client-sofa</artifactId>
  <version>${soul.version}</version>
</dependency>
```

- 注册中心详细接入配置请参考：[注册中心接入](#)

- spring
 - 引入以下依赖：

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-client-sofa</artifactId>
  <version>${project.version}</version>
</dependency>
```

- 在你的 bean 定义的 xml 文件中新增如下：

```
<bean id="sofaServiceBeanPostProcessor" class="org.dromara.soul.client.sofa.
SofaServiceBeanPostProcessor">
  <constructor-arg ref="soulRegisterCenterConfig"/>
</bean>
<bean id="soulRegisterCenterConfig" class="org.dromara.soul.register.common.config.
SoulRegisterCenterConfig">
  <property name="registerType" value="http"/>
  <property name="serverList" value="http://localhost:9095"/>
  <property name="props">
    <map>
      <entry key="contextPath" value="/你的 contextPath"/>
      <entry key="appName" value=" 你的名字"/>
      <entry key="ifFull" value="false"/>
    </map>
  </property>
</bean>
```

7.5.4 sofa 插件设置

- 首先在 soul-admin 插件管理中，把 sofa 插件设置为开启。
- 其次在 sofa 插件中配置你的注册地址或者其他注册中心的地址。

```
{"protocol":"zookeeper","register":"127.0.0.1:2181"}
```

7.5.5 接口注册到网关

- 你 sofa 服务实现类的，方法上加上 @SoulSofaClient 注解，表示该接口方法注册到网关。
- 启动你的提供者，输出日志 sofa client register success 大功告成，你的 sofa 接口已经发布到 soul 网关. 如果还有不懂的，可以参考 soul-test-sofa 项目。

7.5.6 sofa 用户请求以及参数说明

- 说白了，就是通过 http 的方式来请求你的 sofa 服务
- soul 网关需要有一个路由前缀，这个路由前缀就是你接入项目进行配置 contextPath

```
# 比如你有一个 order 服务 它有一个接口，它的注册路径 /order/test/save

# 现在就是通过 post 方式请求网关: http://localhost:9195/order/test/save

# 其中 localhost:9195 为网关的 ip 端口，默认端口是 9195 ， /order 是你 sofa 接入网关配置的 contextPath
```

- 参数传递：
 - 通过 http post 方式访问网关，通过 body，json 类型传递。
 - 更多参数类型传递，可以参考 [soul-examples-sofa](#) 中的接口定义，以及参数传递方式。
- 单个 java bean 参数类型（默认）
- 自定义实现多参数支持：
 - 在你搭建的网关项目中，新增一个类 A，实现 org.dromara.soul.plugin.api.sofa.SofaParamResolveService。

```
public interface SofaParamResolveService {

    /**
     * Build parameter pair.
     * this is Resolve http body to get sofa param.
     *
     * @param body          the body
     * @param parameterTypes the parameter types
     * @return the pair
     */
}
```

```
Pair<String[], Object[]> buildParameter(String body, String parameterTypes);
}
```

- body 为 http 中 body 传的 json 字符串。
- parameterTypes: 匹配到的方法参数类型列表, 如果有多个, 则使用, 分割。
- Pair 中, left 为参数类型, right 为参数值, 这是 sofa 泛化调用的标准。
- 把你的类注册成 Spring 的 bean, 覆盖默认的实现。

```
@Bean
public SofaParamResolveService A() {
    return new A();
}
```

7.6 使用不同的数据同步策略

7.6.1 说明

- 数据同步是指将 soul-admin 配置的数据, 同步到 soul 集群中的 JVM 内存里面, 是网关高性能的关键。
- 实现原理, 请看: [数据同步](#)。
- 文中所说的网关, 是指你搭建的网关环境, 请看: [搭建环境](#)。

7.6.2 websocket 同步（默认方式, 推荐）

- 网关配置（记得重启）
 - 首先在 pom.xml 文件中引入以下依赖：

```
<!--soul data sync start use websocket-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-sync-data-websocket</artifactId>
  <version>${last.version}</version>
</dependency>
```

- 在 springboot 的 yml 文件中进行如下配置:

```
soul :
  sync:
    websocket :
      urls: ws://localhost:9095/websocket
#urls: 是指 soul-admin 的地址, 如果有多个, 请使用 (,) 分割.
```

- soul-admin 配置, 或在 soul-admin 启动参数中设置 `--soul.sync.websocket=''`, 然后重启服务。

```
soul:
  sync:
    websocket:
```

- 当建立连接以后会全量获取一次数据, 以后的数据都是增量的更新与新增, 性能好。
- 支持断线重连 (默认 30 秒)。

7.6.3 zookeeper 同步

- 网关配置 (记得重启)
 - 首先在 pom.xml 文件中引入以下依赖:

```
<!--soul data sync start use zookeeper-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-sync-data-zookeeper</artifactId>
  <version>${last.version}</version>
</dependency>
```

- 在 springboot 的 yml 文件中进行如下配置:

```
```yaml
soul :
 sync:
 zookeeper:
 url: localhost:2181
 sessionTimeout: 5000
 connectionTimeout: 2000
 #url: 配置成你的 zk 地址, 集群环境请使用 (,) 分隔
```
```

- soul-admin 配置, 或在 soul-admin 启动参数中设置 `--soul.sync.zookeeper.url='你的地址'`, 然后重启服务。

```
soul:
  sync:
    zookeeper:
      url: localhost:2181
      sessionTimeout: 5000
      connectionTimeout: 2000
```

- 使用 zookeeper 同步机制也是非常好的, 时效性也高, 我们生产环境使用的就是这个, 但是也要处理 zk 环境不稳定, 集群脑裂等问题。

7.6.4 http 长轮询同步

- 网关配置（记得重启）

– 首先在 pom.xml 文件中引入以下依赖：

```
<!--soul data sync start use http-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-sync-data-http</artifactId>
  <version>${last.version}</version>
</dependency>
```

– 在 springboot 的 yml 文件中进行如下配置：

```
soul :
  sync:
    http:
      url: http://localhost:9095
#url: 配置成你的 soul-admin 的 ip 与端口地址, 多个 admin 集群环境请使用 (,) 分隔。
```

– soul-admin 配置, 或在 soul-admin 启动参数中设置 `--soul.sync.http=''`, 然后重启服务。

```
soul:
  sync:
    http:
```

- http 长轮询使得网关很轻量，时效性略低。
- 其根据分组 key 来拉取，如果数据量过大，过多，会有一定的影响。什么意思呢？就是一个组下面的一个小地方更改，会拉取整个的组数据。
- 在 soul-admin 集群时候，可能会有 bug。

7.6.5 nacos 同步

- 网关配置（记得重启）

– 首先在 pom.xml 文件中引入以下依赖：

```
<!--soul data sync start use nacos-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-sync-data-nacos</artifactId>
  <version>${last.version}</version>
</dependency>
```

– 在 springboot 的 yml 文件中进行如下配置：


```
soul :  
  sync:  
    nacos:  
      url: localhost:8848  
      namespace: 1c10d748-af86-43b9-8265-75f487d20c6c  
      acm:  
        enabled: false  
        endpoint: acm.aliyun.com  
        namespace:  
        accessKey:  
        secretKey:
```

#url: 配置成你的 nacos 地址, 集群环境请使用 (,) 分隔。

其他参数配置, 请参考 nacos 官网。

- soul-admin 配置, 或在 soul-admin 启动参数中使用 -- 的方式一个一个传值

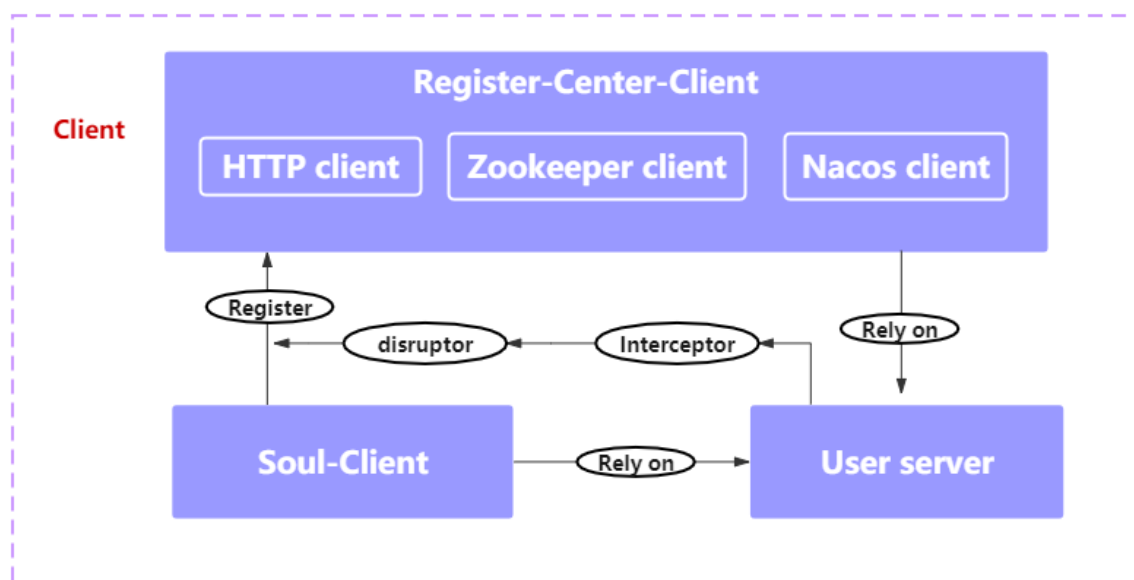
```
soul:  
  sync:  
    nacos:  
      url: localhost:8848  
      namespace: 1c10d748-af86-43b9-8265-75f487d20c6c  
      acm:  
        enabled: false  
        endpoint: acm.aliyun.com  
        namespace:  
        accessKey:  
        secretKey:
```

8.1 注册中心设计

8.1.1 说明

- 本篇主要讲解注册中心原理

8.1.2 Client



配置中声明使用的注册中心客户端类型，如 HTTP/Zookeeper

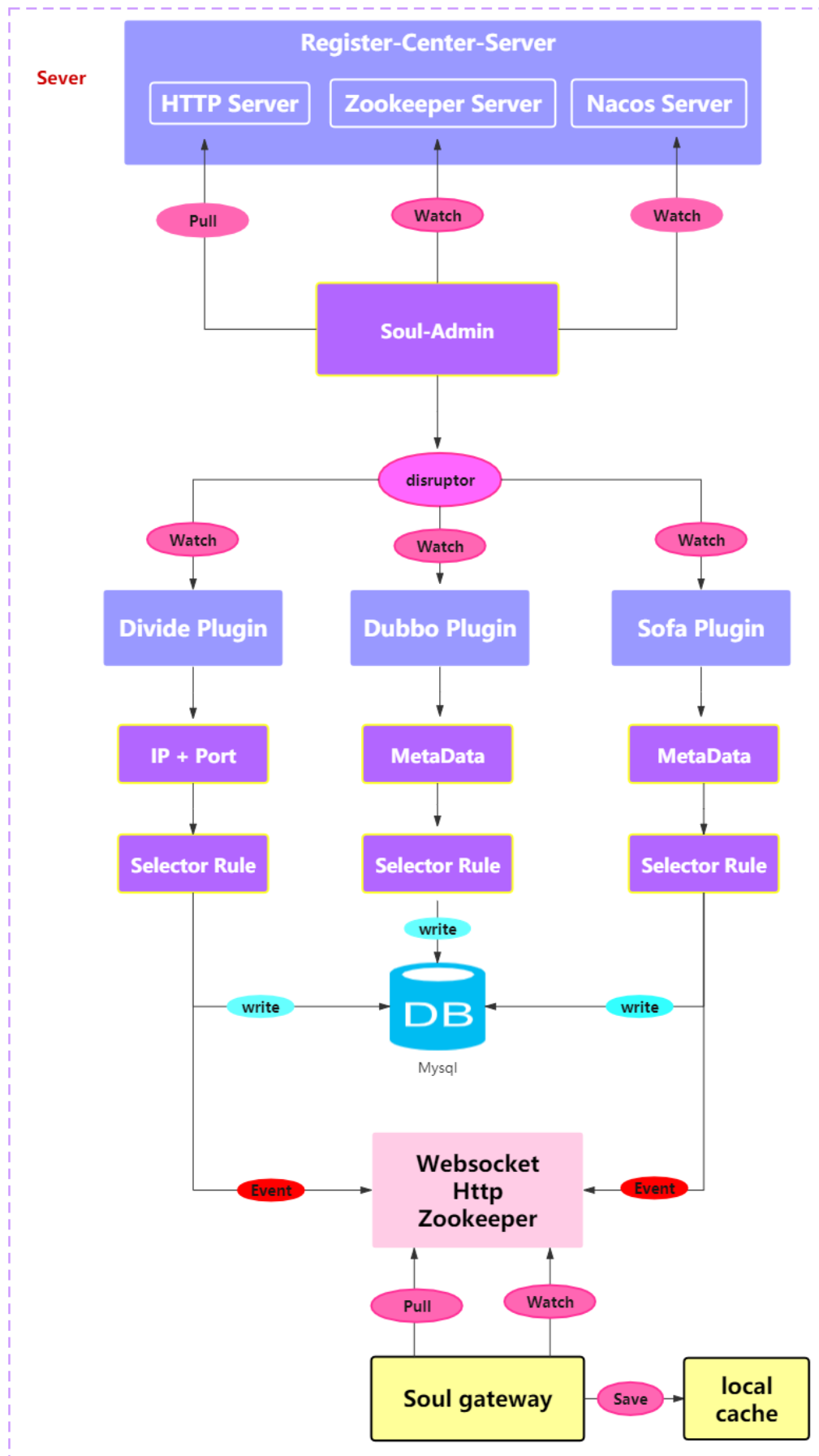
应用程序启动时使用 SPI 方式加载并初始化对应注册中心客户端

通过实现 Spring Bean 相关的后处理器接口，在其中获取需要进行注册的服务接口信息，将获取的信息放入 Disruptor 中

注册中心客户端从 Disruptor 中读取数据，并将接口信息注册到 Soul-Admin

Disruptor 在其中起数据与操作解耦的作用，利于扩展

8.1.3 Server



在 Soul-Admin 配置中声明使用的注册中心服务端类型，如 HTTP/Zookeeper

Soul-Admin 启动时，加载配置的类型，加载并初始化对应的注册中心服务端

注册中心服务端收到 Soul-Client 注册的接口信息后，将其放入 Disruptor 中，然后会触发注册处理逻辑，将服务接口信息更新并发布同步事件

Disruptor 在其中起到数据与操作解耦，利于扩展；同时比较注册请求过多，导致注册异常，有数据缓冲作用

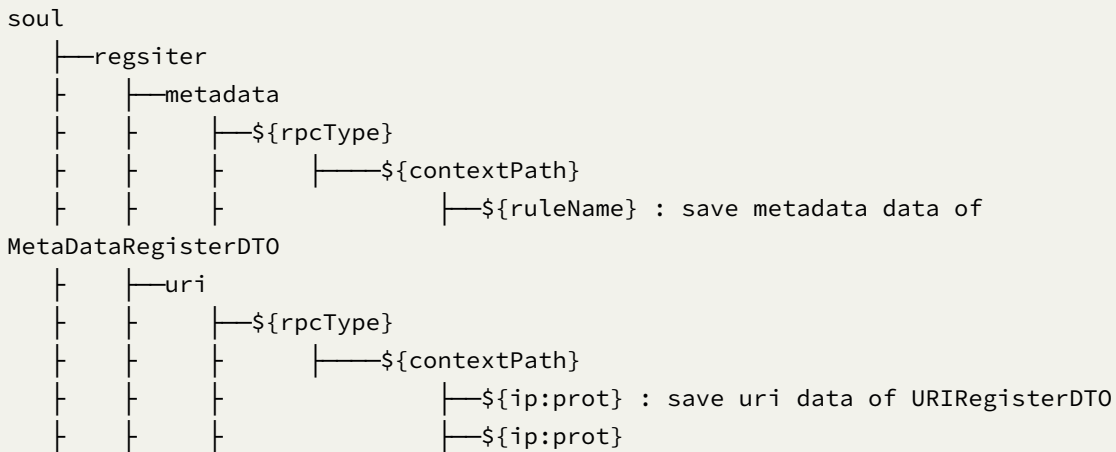
8.1.4 Http 注册

HTTP 服务注册原理较为简单，在 Soul-Client 启动后，会调用 Soul-Admin 的相关服务注册接口，上传数据进行注册

Soul-Admin web 服务接口收到请求后进行数据更新和数据同步事件发布

8.1.5 Zookeeper 注册

Zookeeper 存储结构如下：



Soul-Client 启动时，将服务接口信息（MetadataRegisterDTO/URIRegisterDTO）写到如上的 zookeeper 节点中。

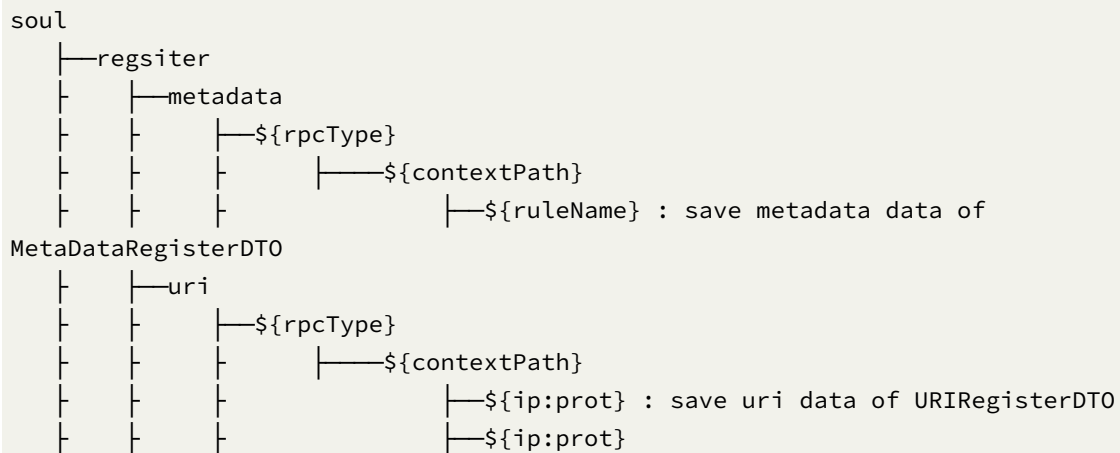
Soul-Admin 使用 Zookeeper 的 Watch 机制，对数据的更新和删除等事件进行监听，数据变更后触发对应的注册处理逻辑。

在收到 MetadataRegisterDTO 节点变更后，触发 selector 和 rule 的数据变更和数据同步事件发布。

收到 URIRegisterDTO 节点变更后，触发 selector 的 upstream 的更新和数据同步事件发布。

8.1.6 Etcd 注册

Etcd 的键值存储结构如下：



Soul-Client 启动时，将服务接口信息（MetaDataRegisterDTO/URIRegisterDTO）以 Ephemeral 方式写到如上的 Etcd 节点中。

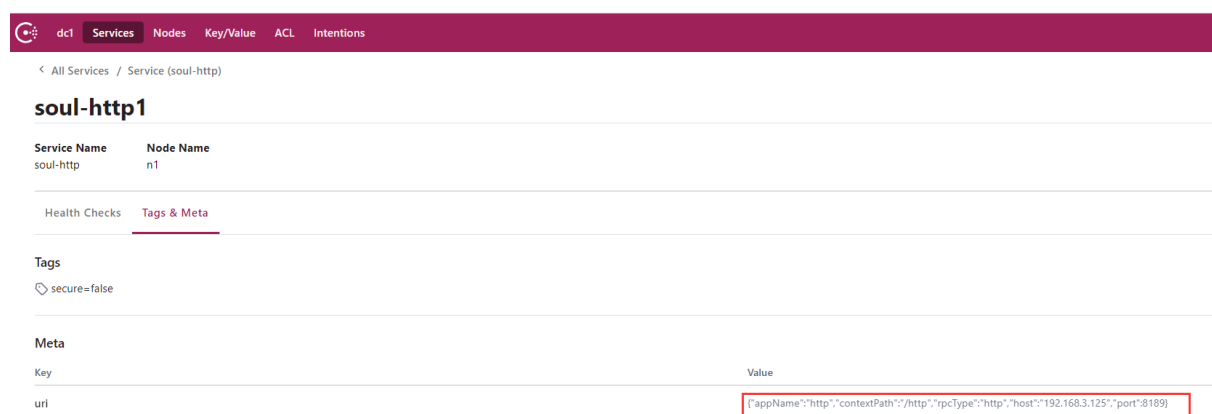
Soul-Admin 使用 Etcd 的 Watch 机制，对数据的更新和删除等事件进行监听，数据变更后触发对应的注册处理逻辑。

在收到 MetaDataRegisterDTO 节点变更后，触发 selector 和 rule 的数据变更和数据同步事件发布。

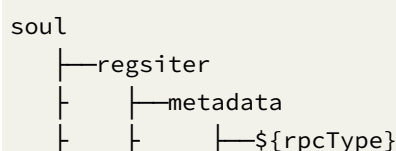
收到 URIRegisterDTO 节点变更后，触发 selector 的 upstream 的更新和数据同步事件发布。

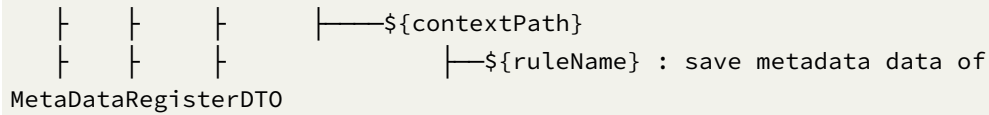
8.1.7 Consul 注册

Consul 的 Metadata 和 URI 分两部分存储，URIRegisterDTO 随着服务注册记录在服务的 metadata 里，服务下线时随着服务节点一起消失。



Consul 的 MetaDataRegisterDTO 存在 Key/Value 里，键值存储结构如下：





Soul-Client 启动时,将服务接口信息(MetadataRegisterDTO/URIRegisterDTO)分别放在 ServiceInstance 的 Metadata (URIRegisterDTO) 和 KeyValue (MetadataRegisterDTO), 按照上述方式进行存储。

Soul-Admin 通过监听 Catalog 和 KeyValue 的 index 的变化, 来感知数据的更新和删除, 数据变更后触发对应的注册处理逻辑。

在收到 MetadataRegisterDTO 节点变更后, 触发 selector 和 rule 的数据变更和数据同步事件发布。

收到 URIRegisterDTO 节点变更后, 触发 selector 的 upstream 的更新和数据同步事件发布。

8.1.8 Nacos 注册

Nacos 分为两部分: URI 和 Metadata。

URI 使用实例注册方式, 在服务异常的情况下, 相关 URI 数据节点会自动进行删除, 并发送事件到订阅端, 订阅端进行相关的下线处理。

Metadata 使用配置注册方式, 没有相关上下线操作, 当有 URI 实例注册时, 会相应的发布 Metadata 配置, 订阅端监听数据变化, 进行更新处理。

URI 实例注册命令规则如下:

```
soul.register.service.${rpcType}
```

初始监听所有的 RpcType 节点, 其下的 {contextPath} 实例会对应注册到其下, 根据 IP 和 Port 进行区分, 并携带其对应的 contextPath 信息。

URI 实例上下线之后, 触发 selector 的 upstream 的更新和数据同步事件发布。

URI 实例上线时, 会发布对应的 Metadata 数据, 其节点名称命令规则如下:

```
soul.register.service.${rpcType}.${contextPath}
```

订阅端会对所有的 Metadata 配置继续监听, 当初次订阅和配置更新后, 触发 selector 和 rule 的数据变更和数据同步事件发布。

8.1.9 SPI 扩展

SPI 名称	详细说明
SoulClientRegisterRepository	Soul 网关客户端接入注册服务资源

已知实现类	详细说明
HttpClientRegisterRepository	基于 Http 请求的实现
ZookeeperClientRegisterRepository	基于 Zookeeper 注册的实现
EtcdClientRegisterRepository	基于 etcd 注册的实现
ConsulClientRegisterRepository	基于 consul 注册的实现
NacosClientRegisterRepository	基于 Nacos 注册的实现

SPI 名称	详细说明
SoulServerRegisterRepository	Soul 网关客户端注册的后台服务资源

已知实现类	详细说明
SoulHttpRegistryController	使用 Http 服务接口来处理客户端注册请求
ZookeeperServerRegisterRepository	使用 Zookeeper 来处理客户端注册节点
EtcdServerRegisterRepository	使用 etcd 来处理客户端注册节点
ConsulServerRegisterRepository	使用 consul 来处理客户端注册节点
NacosServerRegisterRepository	使用 Nacos 来处理客户端注册节点

8.2 注册中心接入配置

title: 注册中心接入配置 keywords: soul description: 注册中心接入配置—

8.2.1 说明

说明然后使用不同的注册方式，快速接入。

8.2.2 HTTP 方式注册

Soul-Admin 配置

在 application.yml 配置注册中心为 HTTP 即可，如下：

```
soul:
  register:
    registerType: http
  props:
    checked: true # 是否开启检测
    zombieCheckTimes: 5 # 失败几次后剔除服务
    scheduledTime: 10 # 定时检测间隔时间 (秒)
```

Soul-Client 配置

在 application.yml 中配置注册方式为 HTTP，并填写 Soul-Admin 服务地址列表，如下：

```
soul:
  client:
    registerType: http
    serverLists: http://localhost:9095
    props:
      contextPath: /http
      appName: http
      port: 8188
      isFull: false
# registerType : 服务注册类型, 填写 http
# serverList: 为 http 注册类型时, 填写 Soul-Admin 项目的地址, 注意加上 http://, 多个地址用英文逗号分隔
# port: 你本项目的启动端口, 目前 springmvc/tars/grpc 需要进行填写
# contextPath: 为你的这个 mvc 项目在 soul 网关的路由前缀, 这个你应该懂意思把? 比如/order , /product 等等, 网关会根据你的这个前缀来进行路由.
# appName: 你的应用名称, 不配置的话, 会默认取 `spring.application.name` 的值
# isFull: 设置 true 代表代理你的整个服务, false 表示代理你其中某几个 controller; 目前适用于 springmvc/springcloud
```

8.2.3 Zookeeper 方式注册

Soul-Admin 配置

- 首先在 pom.xml 文件中加入相关的依赖（默认已经引入）：

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-register-server-zookeeper</artifactId>
  <version>${project.version}</version>
</dependency>
```

- 在 application.yml 配置注册中心为 Zookeeper，填写相关 zookeeper 服务地址和参数，如下：

```
soul:
  register:
    registerType: zookeeper
    serverLists : localhost:2181
    props:
      sessionTimeout: 5000
      connectionTimeout: 2000
```

Soul-Client 配置

- 首先在 pom.xml 文件中加入相关的依赖（默认已经引入）：

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-register-client-zookeeper</artifactId>
  <version>${project.version}</version>
</dependency>
```

- 在 application.yml 中配置注册方式为 Zookeeper，并填写 Zookeeper 服务地址和相关参数，如下：

```
soul:
  client:
    registerType: zookeeper
    serverLists: localhost:2181
    props:
      contextPath: /http
      appName: http
      port: 8188
      isFull: false
# registerType : 服务注册类型, 填写 zookeeper
# serverList: 为 zookeeper 注册类型时, 填写 zookeeper 地址, 多个地址用英文分隔
# port: 你本项目的启动端口, 目前 springmvc/tars/grpc 需要进行填写
# contextPath: 为你的这个 mvc 项目在 soul 网关的路由前缀, 这个你应该懂意思把? 比如/order , /product 等等, 网关会根据你的这个前缀来进行路由.
# appName: 你的应用名称, 不配置的话, 会默认取 `spring.application.name` 的值
# isFull: 设置 true 代表代理你的整个服务, false 表示代理你其中某几个 controller; 目前适用于 springmvc/springcloud
```

8.2.4 Etcd 方式注册

Soul-Admin 配置

- 首先在 pom.xml 文件中加入相关的依赖（默认已经引入）：

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-register-server-etcd</artifactId>
  <version>${project.version}</version>
</dependency>
```

- 在 application.yml 配置注册中心为 etcd, 填写相关 etcd 服务地址和参数，如下：

```
soul:
  register:
    registerType: etcd
    serverLists : http://localhost:2379
```

```
props:
  etcdTimeout: 5000
  etcdTTL: 5
```

Soul-Client 配置

- 首先在 pom.xml 文件中加入相关的依赖（默认已经引入）：

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-register-client-etcd</artifactId>
  <version>${project.version}</version>
</dependency>
```

- 在 application.yml 中配置注册方式为 etcd, 并填写 etcd 服务地址和相关参数, 如下：

```
soul:
  client:
    registerType: etcd
    serverLists: http://localhost:2379
    props:
      contextPath: /http
      appName: http
      port: 8188
      isFull: false
# registerType : 服务注册类型, 填写 etcd
# serverList: 为 etcd 注册类型时, 填写 etcd 地址, 多个地址用英文分隔
# port: 你本项目的启动端口, 目前 springmvc/tars/grpc 需要进行填写
# contextPath: 为你的这个 mvc 项目在 soul 网关的路由前缀, 这个你应该懂意思把? 比如/order , /
# product 等等, 网关会根据你的这个前缀来进行路由.
# appName: 你的应用名称, 不配置的话, 会默认取 `spring.application.name` 的值
# isFull: 设置 true 代表代理你的整个服务, false 表示代理你其中某几个 controller; 目前适用于
springmvc/springcloud
```

8.2.5 Consul 方式注册

Soul-Admin 配置

- 首先在 pom.xml 文件中加入相关的依赖：

```
<!--soul-register-server-consul 默认已经引入-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-register-server-consul</artifactId>
  <version>${project.version}</version>
</dependency>
```

<!--spring-cloud-starter-consul-discovery 需要用户自行引入, 建议选用 2.2.6.RELEASE 版本, 其他版本不保证正常工作-->

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-consul-discovery</artifactId>
  <version>2.2.6.RELEASE</version>
</dependency>
```

- 在 application.yml 配置注册中心为 consul, 额外还需要配置 spring.cloud.consul, 如下:

```
soul:
  register:
    registerType: consul
  props:
    delay: 1
    wait-time: 55

spring:
  cloud:
    consul:
      discovery:
        instance-id: soul-admin-1
        service-name: soul-admin
        tags-as-metadata: false
      host: localhost
      port: 8500
```

registerType : 服务注册类型, 填写 consul

delay: 对 Metadata 的监控每次轮询的间隔时长, 单位为秒, 默认 1 秒

wait-time: 对 Metadata 的监控单次请求的等待时间 (长轮询机制), 单位为秒, 默认 55 秒

instance-id: consul 服务必填, consul 需要通过 instance-id 找到具体服务

service-name 服务注册到 consul 时所在的组名, 不配置的话, 会默认取 `spring.application.name` 的值

host: 为 consul 注册类型时, 填写 consul 地址, 默认 localhost

port: 为 consul 注册类型时, 填写 consul 端口, 默认是 8500

tags-as-metadata: false, 必填, 如果不填默认为 true, 则无法读取 metadata 里的 URI 信息导致 selector 的 upstream 数据更新失败。

Soul-Client 配置

注意，**consul** 注册中心目前和 **SpringCloud** 服务不兼容，会和 **Eureka/Nacos** 注册中心冲突

- 首先在 pom.xml 文件中加入相关的依赖（需要自行引入）：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-consul-discovery</artifactId>
  <version>2.2.6.RELEASE</version>
</dependency>
```

- 在 application.yml 中配置注册方式为 consul, 额外还需要配置 spring.cloud.consul, 如下：

```
soul:
  client:
    registerType: consul
    props:
      contextPath: /http
      appName: http
      port: 8188
      isFull: false

spring:
  cloud:
    consul:
      discovery:
        instance-id: soul-http-1
        service-name: soul-http
        host: localhost
        port: 8500
# registerType : 服务注册类型, 填写 consul
# soul.client.props.port: 你本项目的启动端口, 目前 springmvc/tars/grpc 需要进行填写
# contextPath: 为你的这个 mvc 项目在 soul 网关的路由前缀, 这个你应该懂意思把? 比如/order , /
# product 等等, 网关会根据你的这个前缀来进行路由.
# appName: 你的应用名称, 不配置的话, 会默认取 `spring.application.name` 的值
# isFull: 设置 true 代表代理你的整个服务, false 表示代理你其中某几个 controller; 目前适用于
# springmvc
# instance-id: consul 服务必填, consul 需要通过 instance-id 找到具体服务
# service-name 服务注册到 consul 时所在的组名, 不配置的话, 会默认取 `spring.application.
# name` 的值
# host: 为 consul 注册类型时, 填写 consul 地址, 默认 localhost
# spring.cloud.consul.port: 为 consul 注册类型时, 填写 consul 端口, 默认是 8500
```

8.2.6 Nacos 方式注册

Soul-Admin 配置

- 首先在 pom.xml 文件中加入相关的依赖（默认已经引入）：

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-register-server-nacos</artifactId>
  <version>${project.version}</version>
</dependency>
```

- 在 application.yml 配置注册中心为 nacos, 填写相关 nacos 服务地址和参数, 还有 Nacos 的命名空间（需要和 Soul-Client 保持一致），如下：

```
soul:
  register:
    registerType: nacos
    serverLists : localhost:8848
    props:
      nacosNameSpace: SoulRegisterCenter
```

Soul-Client 配置

- 首先在 pom.xml 文件中加入相关的依赖（默认已经引入）：

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-register-client-nacos</artifactId>
  <version>${project.version}</version>
</dependency>
```

- 在 application.yml 中配置注册方式为 nacos, 并填写 nacos 服务地址和相关参数, 需要命名空间（需要和 Soul-Admin 端保持一致），IP（可不填，则自动获取本机 ip）和端口，如下：

```
soul:
  client:
    registerType: nacos
    serverLists: localhost:8848
    props:
      contextPath: /http
      appName: http
      port: 8188
      isFull: false
      nacosNameSpace: SoulRegisterCenter
# registerType : 服务注册类型, 填写 etcd
# serverList: 为 etcd 注册类型时, 填写 etcd 地址, 多个地址用英文分隔
# port: 你本项目的启动端口, 目前 springmvc/tars/grpc 需要进行填写
```

```
# contextPath: 为你的这个 mvc 项目在 soul 网关的路由前缀, 这个你应该懂意思把? 比如/order , /product 等等, 网关会根据你的这个前缀来进行路由.  
# appName: 你的应用名称, 不配置的话, 会默认取 `spring.application.name` 的值  
# isFull: 设置 true 代表代理你的整个服务, false 表示代理你其中某几个 controller; 目前适用于 springmvc/springcloud  
# nacosNameSpace: nacos 的命名空间
```


9.1 Dubbo 快速开始

本文档将演示了如何快速使用 Dubbo 接入 Soul 网关。您可以直接在工程下找到本文档的示例代码。

9.1.1 环境准备

请参考配置网关环境并启动 soul-admin 和 soul-bootstrap, 另外如果你的 dubbo 如果使用 zookeeper 需提前下载启动。

9.1.2 运行 soul-examples-dubbo 项目

下载 soul-examples-dubbo, 调整 spring-dubbo.xml 的注册地址为你本地, 如:

```
<dubbo:registry address="zookeeper://localhost:2181"/>
```

运行 TestApacheDubboApplicationmain 方法启动 dubbo 项目。成功启动会有如下日志:

```
2021-02-06 20:58:01.807 INFO 3724 --- [pool-2-thread-1] o.d.s.client.common.utils.
RegisterUtils : dubbo client register success: {"appName":"dubbo","contextPath":"/
dubbo","path":"/dubbo/insert","pathDesc":"Insert a row of data","rpcType":"dubbo",
"serviceName":"org.dromara.soul.examples.dubbo.api.service.DubboTestService",
"methodName":"insert","ruleName":"/dubbo/insert","parameterTypes":"org.dromara.
soul.examples.dubbo.api.entity.DubboTest","rpcExt":{"group":"","version":"","
","loadbalance":"random","retries":2,"timeout":10000,"url":"",""},
"enabled":true}
2021-02-06 20:58:01.821 INFO 3724 --- [pool-2-thread-1] o.d.s.client.common.utils.
RegisterUtils : dubbo client register success: {"appName":"dubbo","contextPath":"/
dubbo","path":"/dubbo/findAll","pathDesc":"Get all data","rpcType":"dubbo",
"serviceName":"org.dromara.soul.examples.dubbo.api.service.DubboTestService",
"methodName":"findAll","ruleName":"/dubbo/findAll","parameterTypes":"","rpcExt":{"
"group":"","version":"","","loadbalance":"random","retries":2,"timeout\
":10000,"url":"",""},"enabled":true}
```

```

2021-02-06 20:58:01.833 INFO 3724 --- [pool-2-thread-1] o.d.s.client.common.utils.
RegisterUtils : dubbo client register success: {"appName":"dubbo","contextPath":"/
dubbo","path":"/dubbo/findById","pathDesc":"Query by Id","rpcType":"dubbo",
"serviceName":"org.dromara.soul.examples.dubbo.api.service.DubboTestService",
"methodName":"findById","ruleName":"/dubbo/findById","parameterTypes":"java.lang.
String","rpcExt":{"group":"","version":"","loadbalance":"random","retries\
":2,"timeout":10000,"url":"","enabled":true}
2021-02-06 20:58:01.844 INFO 3724 --- [pool-2-thread-1] o.d.s.client.common.utils.
RegisterUtils : dubbo client register success: {"appName":"dubbo","contextPath":"/
dubbo","path":"/dubbo/findById","pathDesc":"","rpcType":"dubbo","serviceName":
"org.dromara.soul.examples.dubbo.api.service.DubboMultiParamService","methodName":
"findById","ruleName":"/dubbo/findById","parameterTypes":"java.util.List",
"rpcExt":{"group":"","version":"","loadbalance":"random","retries\
":2,"timeout":10000,"url":"","enabled":true}
2021-02-06 20:58:01.855 INFO 3724 --- [pool-2-thread-1] o.d.s.client.common.utils.
RegisterUtils : dubbo client register success: {"appName":"dubbo","contextPath":"/
dubbo","path":"/dubbo/findByIdsAndName","pathDesc":"","rpcType":"dubbo",
"serviceName":"org.dromara.soul.examples.dubbo.api.service.DubboMultiParamService",
"methodName":"findByIdsAndName","ruleName":"/dubbo/findByIdsAndName",
"parameterTypes":"java.util.List,java.lang.String","rpcExt":{"group":"","
version":"","loadbalance":"random","retries":2,"timeout":10000,"url\
":"","enabled":true}
2021-02-06 20:58:01.866 INFO 3724 --- [pool-2-thread-1] o.d.s.client.common.utils.
RegisterUtils : dubbo client register success: {"appName":"dubbo","contextPath":"/
dubbo","path":"/dubbo/batchSave","pathDesc":"","rpcType":"dubbo","serviceName":
"org.dromara.soul.examples.dubbo.api.service.DubboMultiParamService","methodName":
"batchSave","ruleName":"/dubbo/batchSave","parameterTypes":"java.util.List","rpcExt
":{"group":"","version":"","loadbalance":"random","retries":2,"
timeout":10000,"url":"","enabled":true}
2021-02-06 20:58:01.876 INFO 3724 --- [pool-2-thread-1] o.d.s.client.common.utils.
RegisterUtils : dubbo client register success: {"appName":"dubbo","contextPath":"/
dubbo","path":"/dubbo/findByIdsAndName","pathDesc":"","rpcType":"dubbo",
"serviceName":"org.dromara.soul.examples.dubbo.api.service.DubboMultiParamService",
"methodName":"findByIdsAndName","ruleName":"/dubbo/findByIdsAndName",
"parameterTypes":"[Ljava.lang.Integer;java.lang.String","rpcExt":{"group":"","
version":"","loadbalance":"random","retries":2,"timeout":10000,"url\
":"","enabled":true}
2021-02-06 20:58:01.889 INFO 3724 --- [pool-2-thread-1] o.d.s.client.common.utils.
RegisterUtils : dubbo client register success: {"appName":"dubbo","contextPath":"/
dubbo","path":"/dubbo/saveComplexBeanTestAndName","pathDesc":"","rpcType":"dubbo",
"serviceName":"org.dromara.soul.examples.dubbo.api.service.DubboMultiParamService",
"methodName":"saveComplexBeanTestAndName","ruleName":"/dubbo/
saveComplexBeanTestAndName","parameterTypes":"org.dromara.soul.examples.dubbo.api.
entity.ComplexBeanTest,java.lang.String","rpcExt":{"group":"","
version":"","loadbalance":"random","retries":2,"timeout":10000,"url\
":"","enabled":true}
2021-02-06 20:58:01.901 INFO 3724 --- [pool-2-thread-1] o.d.s.client.common.utils.
RegisterUtils : dubbo client register success: {"appName":"dubbo","contextPath":"/
dubbo","path":"/dubbo/batchSaveAndNameAndId","pathDesc":"","rpcType":"dubbo",
"serviceName":"org.dromara.soul.examples.dubbo.api.service.DubboMultiParamService",
"methodName":"batchSaveAndNameAndId","ruleName":"/dubbo/batchSaveAndNameAndId",
"parameterTypes":"java.util.List,java.lang.String,java.lang.String","rpcExt":{"
group":"","version":"","loadbalance":"random","retries":2,"timeout\
":10000,"url":"","enabled":true}

```

```

2021-02-06 20:58:01.911 INFO 3724 --- [pool-2-thread-1] o.d.s.client.common.utils.
RegisterUtils : dubbo client register success: {"appName":"dubbo","contextPath":"/
dubbo","path":"/dubbo/saveComplexBeanTest","pathDesc":"","rpcType":"dubbo",
"serviceName":"org.dromara.soul.examples.dubbo.api.service.DubboMultiParamService",
"methodName":"saveComplexBeanTest","ruleName":"/dubbo/saveComplexBeanTest",
"parameterTypes":"org.dromara.soul.examples.dubbo.api.entity.ComplexBeanTest",
"rpcExt":{"group":"","version":"","loadbalance":"random","retries\
":2,"timeout":10000,"url":"","enabled":true}
2021-02-06 20:58:01.922 INFO 3724 --- [pool-2-thread-1] o.d.s.client.common.utils.
RegisterUtils : dubbo client register success: {"appName":"dubbo","contextPath":"/
dubbo","path":"/dubbo/findByStringArray","pathDesc":"","rpcType":"dubbo",
"serviceName":"org.dromara.soul.examples.dubbo.api.service.DubboMultiParamService",
"methodName":"findByStringArray","ruleName":"/dubbo/findByStringArray",
"parameterTypes":["Ljava.lang.String;","rpcExt":{"group":"","version":"","loadbalance\
":"random","retries":2,"timeout":10000,"url":"","enabled
":true}

```

9.1.3 dubbo 插件设置

- 首先在 soul-admin 插件管理中，把 dubbo 插件设置为开启。
- 其次在 dubbo 插件中配置你的注册地址，或者其他注册中心的地址。

9.1.4 测试

soul-examples-dubbo 项目成功启动之后会自动把加 @SoulDubboClient 注解的接口方法注册到网关。

打开插件管理->dubbo 可以看到插件规则配置列表

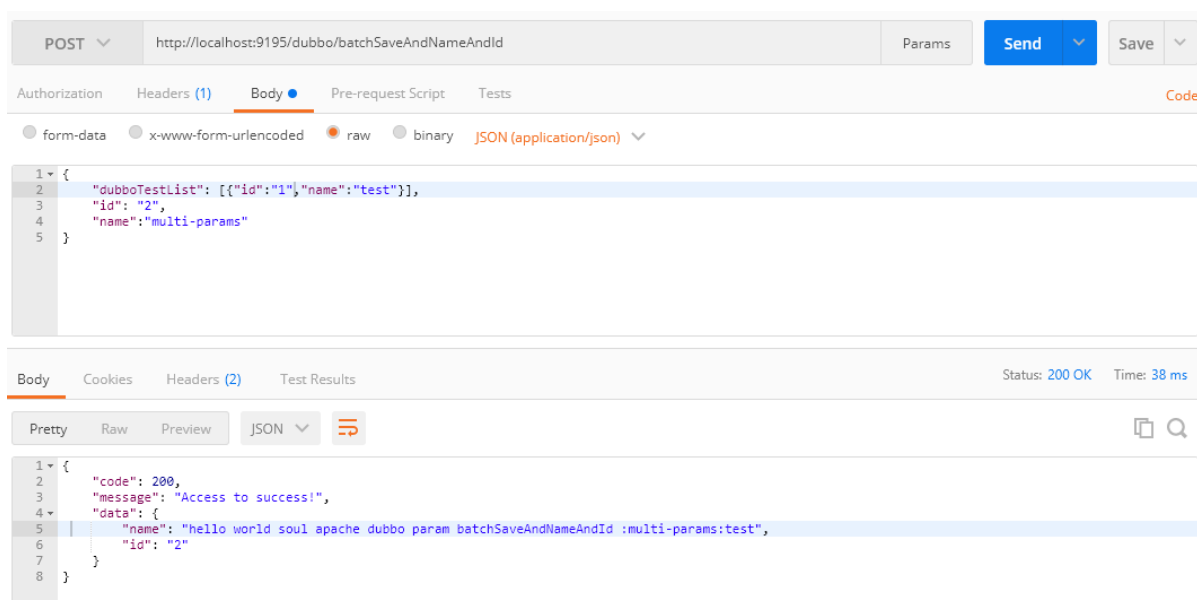
SelectorList			RulesList			
			C Synchronous dubbo			
Name	Open	Operation	RuleName	Open	UpdateTime	Operation
/dubbo	Open	Modify Delete	/dubbo/insert	Open	2021-02-06 20:58:01	Modify Delete
			/dubbo/findAll	Open	2021-02-06 20:58:01	Modify Delete
			/dubbo/findById	Open	2021-02-06 20:58:01	Modify Delete
			/dubbo/findByIdList	Open	2021-02-06 20:58:01	Modify Delete
			/dubbo/findByIdsAndName	Open	2021-02-06 20:58:01	Modify Delete
			/dubbo/batchSave	Open	2021-02-06 20:58:01	Modify Delete
			/dubbo/findByIdsAndName	Open	2021-02-06 20:58:01	Modify Delete
			/dubbo/saveComplexBeanTestAndName	Open	2021-02-06 20:58:01	Modify Delete
			/dubbo/batchSaveAndNameAndId	Open	2021-02-06 20:58:01	Modify Delete
			/dubbo/saveComplexBeanTest	Open	2021-02-06 20:58:01	Modify Delete
			/dubbo/findByStringArray	Open	2021-02-06 20:58:01	Modify Delete

下面使用 postman 模拟 http 的方式来请求你的 dubbo 服务



复杂多参数示例：对应接口实现类为 `org.dromara.soul.examples.apache.dubbo.service.impl.DubboMultiParamServiceImpl#batchSaveAndNameAndId`

```
@Override
@SoulDubboClient(path = "/batchSaveAndNameAndId")
public DubboTest batchSaveAndNameAndId(List<DubboTest> dubboTestList, String id,
String name) {
    DubboTest test = new DubboTest();
    test.setId(id);
    test.setName("hello world soul apache dubbo param batchSaveAndNameAndId :" +
name + ":" + dubboTestList.stream().map(DubboTest::getName).collect(Collectors.
joining("-")));
    return test;
}
```



当你的参数不匹配时会报如下异常：

```
2021-02-07 22:24:04.015 ERROR 14860 --- [20888-thread-3] o.d.soul.web.handler.
GlobalErrorHandler : [e47b2a2a] Resolved [SoulException: org.apache.dubbo.
remoting.RemotingException: java.lang.IllegalArgumentException: args.length !=
types.length
```

```

java.lang.IllegalArgumentException: args.length != types.length
    at org.apache.dubbo.common.utils.PojoUtils.realize(PojoUtils.java:91)
    at org.apache.dubbo.rpc.filter.GenericFilter.invoke(GenericFilter.java:82)
    at org.apache.dubbo.rpc.protocol.ProtocolFilterWrapper$1.
invoke(ProtocolFilterWrapper.java:81)
    at org.apache.dubbo.rpc.filter.ClassLoaderFilter.invoke(ClassLoaderFilter.
java:38)
    at org.apache.dubbo.rpc.protocol.ProtocolFilterWrapper$1.
invoke(ProtocolFilterWrapper.java:81)
    at org.apache.dubbo.rpc.filter.EchoFilter.invoke(EchoFilter.java:41)
    at org.apache.dubbo.rpc.protocol.ProtocolFilterWrapper$1.
invoke(ProtocolFilterWrapper.java:81)
    at org.apache.dubbo.rpc.protocol.dubbo.DubboProtocol$1.reply(DubboProtocol.
java:150)
    at org.apache.dubbo.remoting.exchange.support.header.HeaderExchangeHandler.
handleRequest(HeaderExchangeHandler.java:100)
    at org.apache.dubbo.remoting.exchange.support.header.HeaderExchangeHandler.
received(HeaderExchangeHandler.java:175)
    at org.apache.dubbo.remoting.transport.DecodeHandler.received(DecodeHandler.
java:51)
    at org.apache.dubbo.remoting.transport.dispatcher.ChannelEventRunnable.
run(ChannelEventRunnable.java:57)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.
java:1149)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.
java:624)
    at java.lang.Thread.run(Thread.java:748)
] for HTTP POST /dubbo/batchSaveAndNameAndId

```

9.2 Http 快速开始

本文档将演示了如何快速使用 Http 请求接入 Soul 网关。您可以直接在工程下找到本文档的示例代码。

9.2.1 环境准备

请参考[配置网关环境](#)并启动 soul-admin。

引入网关对 http 的代理插件

- 在网关的 pom.xml 文件中增加如下依赖：

```

<!--if you use http proxy start this-->
<dependency>
    <groupId>org.dromara</groupId>
    <artifactId>soul-spring-boot-starter-plugin-divide</artifactId>
    <version>${last.version}</version>
</dependency>

```

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-httpclient</artifactId>
  <version>${last.version}</version>
</dependency>
```

启动 soul-bootstrap 项目。

9.2.2 运行 soul-examples-http 项目

下载 [soul-examples-http](#)

运行 `org.dromara.soul.examples.http.SoulTestHttpApplicationmain` 方法启动项目。

成功启动会有如下日志：

```
2021-02-10 00:57:07.561 INFO 3700 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : http client register success: {"appName":"http","context":"/http",
"path":"/http/test/**","pathDesc":"","rpcType":"http","host":"192.168.50.13","port
":8188,"ruleName":"/http/test/**","enabled":true,"registerMetaData":false}
2021-02-10 00:57:07.577 INFO 3700 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : http client register success: {"appName":"http","context":"/http",
"path":"/http/order/save","pathDesc":"Save order","rpcType":"http","host":"192.168.
50.13","port":8188,"ruleName":"/http/order/save","enabled":true,"registerMetaData
":false}
2021-02-10 00:57:07.587 INFO 3700 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : http client register success: {"appName":"http","context":"/http",
"path":"/http/order/path/**/name","pathDesc":"","rpcType":"http","host":"192.168.
50.13","port":8188,"ruleName":"/http/order/path/**/name","enabled":true,
"registerMetaData":false}
2021-02-10 00:57:07.596 INFO 3700 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : http client register success: {"appName":"http","context":"/http",
"path":"/http/order/findById","pathDesc":"Find by id","rpcType":"http","host":"192.
168.50.13","port":8188,"ruleName":"/http/order/findById","enabled":true,
"registerMetaData":false}
2021-02-10 00:57:07.606 INFO 3700 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : http client register success: {"appName":"http","context":"/http",
"path":"/http/order/path/**","pathDesc":"","rpcType":"http","host":"192.168.50.13",
"port":8188,"ruleName":"/http/order/path/**","enabled":true,"registerMetaData
":false}
2021-02-10 00:57:08.023 INFO 3700 --- [ main] o.s.b.web.embedded.netty.
NettyWebServer : Netty started on port(s): 8188
2021-02-10 00:57:08.026 INFO 3700 --- [ main] o.d.s.e.http.
SoulTestHttpApplication : Started SoulTestHttpApplication in 2.555 seconds (JVM
running for 3.411)
```

9.2.3 开启 divide 插件来处理 http 请求

- 在 soul-admin 插件管理中，把 divide 插件设置为开启。

9.2.4 测试 Http 请求

soul-examples-http 项目成功启动之后会自动把加 @SoulSpringMvcClient 注解的接口方法注册到网关。

打开插件管理->divide 可以看到插件规则配置列表

SelectorList

Add

RulesList

Synchronous divide

Add

Name	Open	Operation
/http	Open	Modify Delete

<

1

>

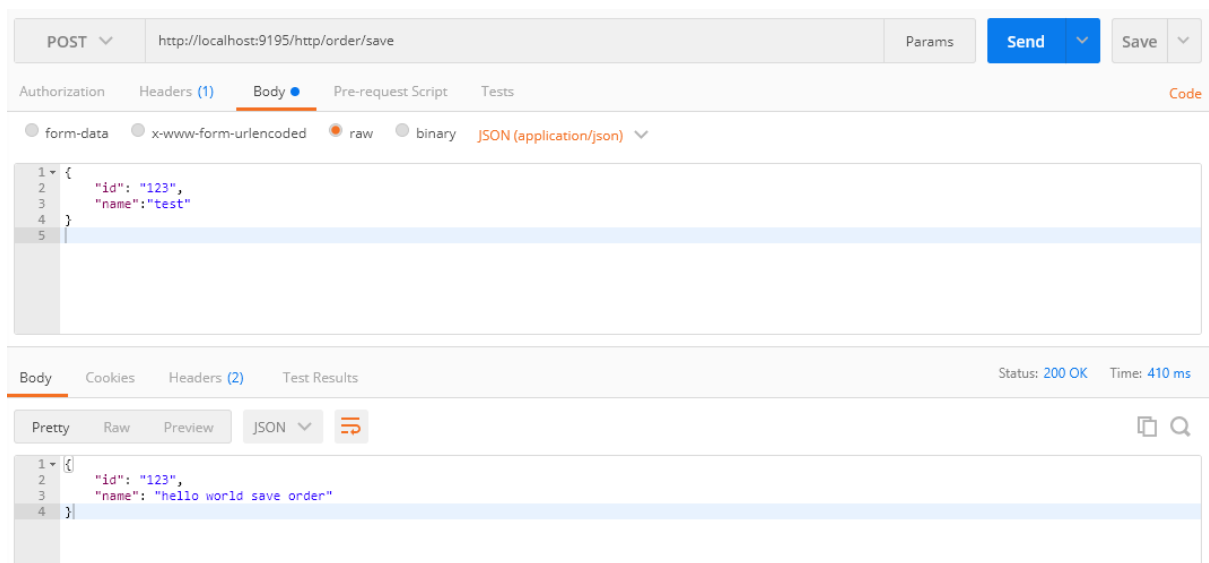
	RuleName	Open	UpdateTime	Operation
+	/http/test/**	Open	2021-02-10 00:57:07	Modify Delete
+	/http/order/save	Open	2021-02-10 00:57:07	Modify Delete
+	/http/order/path/**/name	Open	2021-02-10 00:57:07	Modify Delete
+	/http/order/findById	Open	2021-02-10 00:57:07	Modify Delete
+	/http/order/path/**	Open	2021-02-10 00:57:07	Modify Delete

<

1

>

下面使用 postman 模拟 http 的方式来请求你的 http 服务



9.3 Grpc 快速开始

本文档将演示了如何快速使用 Grpc 接入 Soul 网关。您可以直接在工程下找到本文档的示例代码。

9.3.1 环境准备

请参考配置网关环境并启动 soul-admin 和 soul-bootstrap。

注：soul-bootstrap 需要引入 grpc 依赖

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-grpc</artifactId>
  <version>${project.version}</version>
</dependency>
```

9.3.2 运行 soul-examples-grpc 项目

下载soul-examples-grpc

在 soul-examples-grpc 下执行以下命令生成 java 代码

```
mvn protobuf:compile //编译消息对象
mvn protobuf:compile-custom //依赖消息对象，生成接口服务
```

运行 org.dromara.soul.examples.grpc.SoulTestGrpcApplicationmain 方法启动项目。

成功启动会有如下日志：

```
2021-02-10 01:57:02.154 INFO 76 --- [          main] o.d.s.e.grpc.
SoulTestGrpcApplication : Started SoulTestGrpcApplication in 2.088 seconds (JVM
running for 3.232)
2021-02-10 01:57:02.380 INFO 76 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : grpc client register success: {"appName":"127.0.0.1:8080",
"contextPath":"/grpc","path":"/grpc/echo","pathDesc":"","rpcType":"grpc",
"serviceName":"echo.EchoService","methodName":"echo","ruleName":"/grpc/echo",
"parameterTypes":"echo.EchoRequest,io.grpc.stub.StreamObserver","rpcExt":{"\
"timeout\":-1},"enabled":true}
```

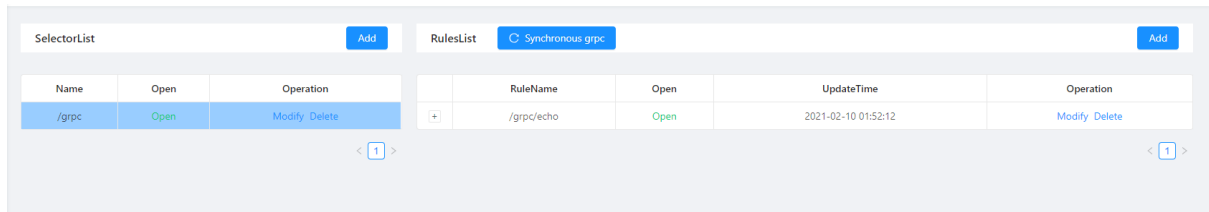

9.3.3 Grpc 插件设置

- 在 soul-admin 插件管理中，把 grpc 插件设置为开启。

9.3.4 测试

soul-examples-grpc 项目成功启动之后会自动把加 @SoulGrpcClient 注解的接口方法注册到网关。

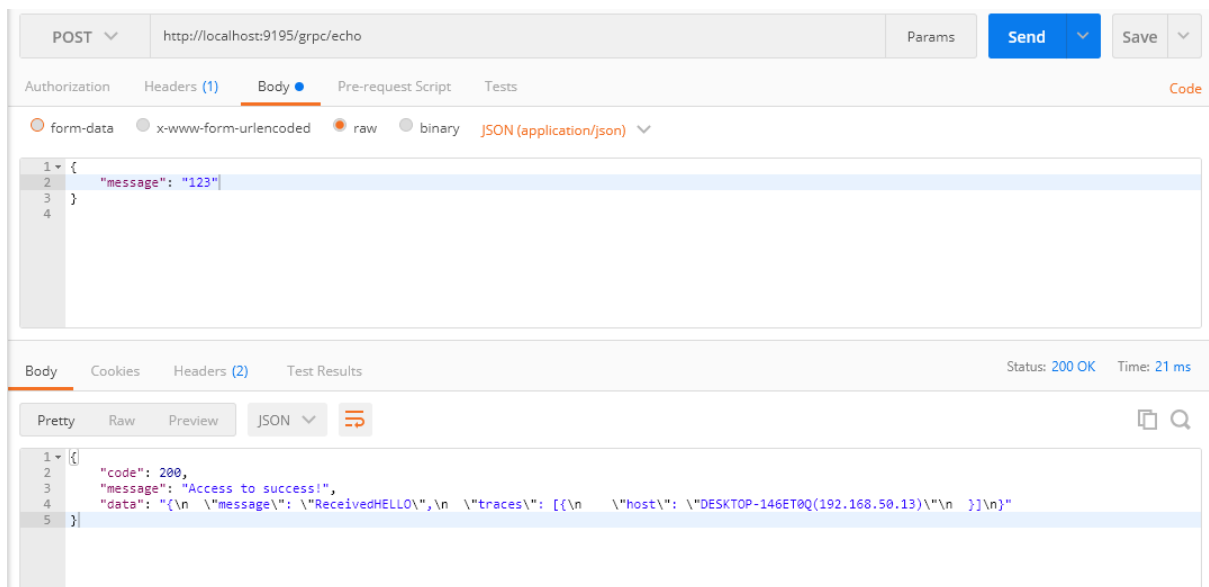
打开插件管理->grpc 可以看到插件规则配置列表



The screenshot shows the 'Plugin Management' interface in Soul Admin. It has two tabs: 'SelectorList' and 'RulesList'. The 'RulesList' tab is active, showing a table of rules. The table has columns: Name, Open, UpdateTime, and Operation. There is one rule listed: '/grpc/echo' with 'Open' status and 'UpdateTime' of '2021-02-10 01:52:12'. The 'Operation' column has links for 'Modify' and 'Delete'.

Name	Open	UpdateTime	Operation
/grpc/echo	Open	2021-02-10 01:52:12	Modify Delete

下面使用 postman 模拟 http 的方式来请求你的 grpc 服务



9.4 SpringCloud 快速开始

本文档将演示了如何快速使用 SpringCloud 方式接入 Soul 网关。您可以直接在工程下找到本文档的示例代码。

9.4.1 环境准备

请参考配置网关环境并启动 soul-admin。

- 在网关的 pom.xml 文件中增加如下依赖：

```
<!--soul springCloud plugin start-->
dependency>
    <groupId>org.dromara</groupId>
    <artifactId>soul-spring-boot-starter-plugin-springcloud</artifactId>
    <version>${project.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-commons</artifactId>
    <version>2.2.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
    <version>2.2.0.RELEASE</version>
</dependency>

<!-- 如果使用 eureka 作为注册中心需要引入 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>2.2.0.RELEASE</version>
</dependency>

<!--soul springCloud plugin start end-->
```

启动 soul-bootstrap 项目。

9.4.2 运行 soul-examples-springcloud、soul-examples-eureka 项目

示例项目中我们使用 eureka 作为 springCloud 的注册中心

下载soul-examples-eureka、soul-examples-springcloud

1、先启动 eureka 服务

运行 org.dromara.soul.examples.eureka.EurekaServerApplicationmain 方法启动项目。

2、先启动 spring cloud 服务

运行 org.dromara.soul.examples.springcloud.SoulTestSpringCloudApplicationmain 方法启动项目。

成功启动会有如下日志：

```

2021-02-10 14:03:51.301 INFO 2860 --- [          main] o.s.s.concurrent.
ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2021-02-10 14:03:51.669 INFO 2860 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : springCloud client register success: {"appName":"springCloud-test
","context":"/springcloud","path":"/springcloud/order/save","pathDesc":"","rpcType
":"springCloud","ruleName":"/springcloud/order/save","enabled":true}
2021-02-10 14:03:51.676 INFO 2860 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : springCloud client register success: {"appName":"springCloud-test
","context":"/springcloud","path":"/springcloud/order/path/**","pathDesc":"","
","rpcType":"springCloud","ruleName":"/springcloud/order/path/**","enabled":true}
2021-02-10 14:03:51.682 INFO 2860 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : springCloud client register success: {"appName":"springCloud-test
","context":"/springcloud","path":"/springcloud/order/findById","pathDesc":"","
","rpcType":"springCloud","ruleName":"/springcloud/order/findById","enabled":true}
2021-02-10 14:03:51.688 INFO 2860 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : springCloud client register success: {"appName":"springCloud-test
","context":"/springcloud","path":"/springcloud/order/path/**/name","pathDesc":"","
","rpcType":"springCloud","ruleName":"/springcloud/order/path/**/name","enabled
":true}
2021-02-10 14:03:51.692 INFO 2860 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : springCloud client register success: {"appName":"springCloud-test
","context":"/springcloud","path":"/springcloud/test/**","pathDesc":"","rpcType":
"springCloud","ruleName":"/springcloud/test/**","enabled":true}
2021-02-10 14:03:52.806 WARN 2860 --- [          main]
ockingLoadBalancerClientRibbonWarnLogger : You already have
RibbonLoadBalancerClient on your classpath. It will be used by default. As Spring
Cloud Ribbon is in maintenance mode. We recommend switching to
BlockingLoadBalancerClient instead. In order to use it, set the value of `spring.
cloud.loadbalancer.ribbon.enabled` to `false` or remove spring-cloud-starter-
netflix-ribbon from your project.
2021-02-10 14:03:52.848 WARN 2860 --- [          main] igation
$LoadBalancerCaffeineWarnLogger : Spring Cloud LoadBalancer is currently working
with default default cache. You can switch to using Caffeine cache, by adding it to
the classpath.
2021-02-10 14:03:52.921 INFO 2860 --- [          main] o.s.c.n.eureka.
InstanceInfoFactory : Setting initial instance status as: STARTING
2021-02-10 14:03:52.949 INFO 2860 --- [          main] com.netflix.discovery.
DiscoveryClient : Initializing Eureka in region us-east-1
2021-02-10 14:03:53.006 INFO 2860 --- [          main] c.n.d.provider.
DiscoveryJerseyProvider : Using JSON encoding codec LegacyJacksonJson
2021-02-10 14:03:53.006 INFO 2860 --- [          main] c.n.d.provider.
DiscoveryJerseyProvider : Using JSON decoding codec LegacyJacksonJson
2021-02-10 14:03:53.110 INFO 2860 --- [          main] c.n.d.provider.
DiscoveryJerseyProvider : Using XML encoding codec XStreamXml
2021-02-10 14:03:53.110 INFO 2860 --- [          main] c.n.d.provider.
DiscoveryJerseyProvider : Using XML decoding codec XStreamXml
2021-02-10 14:03:53.263 INFO 2860 --- [          main] c.n.d.s.r.aws.
ConfigClusterResolver : Resolving eureka endpoints via configuration

```

```

2021-02-10 14:03:53.546 INFO 2860 --- [main] com.netflix.discovery.
DiscoveryClient : Disable delta property : false
2021-02-10 14:03:53.546 INFO 2860 --- [main] com.netflix.discovery.
DiscoveryClient : Single vip registry refresh property : null
2021-02-10 14:03:53.547 INFO 2860 --- [main] com.netflix.discovery.
DiscoveryClient : Force full registry fetch : false
2021-02-10 14:03:53.547 INFO 2860 --- [main] com.netflix.discovery.
DiscoveryClient : Application is null : false
2021-02-10 14:03:53.547 INFO 2860 --- [main] com.netflix.discovery.
DiscoveryClient : Registered Applications size is zero : true
2021-02-10 14:03:53.547 INFO 2860 --- [main] com.netflix.discovery.
DiscoveryClient : Application version is -1: true
2021-02-10 14:03:53.547 INFO 2860 --- [main] com.netflix.discovery.
DiscoveryClient : Getting all instance registry info from the eureka server
2021-02-10 14:03:53.754 INFO 2860 --- [main] com.netflix.discovery.
DiscoveryClient : The response status is 200
2021-02-10 14:03:53.756 INFO 2860 --- [main] com.netflix.discovery.
DiscoveryClient : Starting heartbeat executor: renew interval is: 30
2021-02-10 14:03:53.758 INFO 2860 --- [main] c.n.discovery.
InstanceInfoReplicator : InstanceInfoReplicator onDemand update allowed rate
per min is 4
2021-02-10 14:03:53.761 INFO 2860 --- [main] com.netflix.discovery.
DiscoveryClient : Discovery Client initialized at timestamp 1612937033760 with
initial instances count: 0
2021-02-10 14:03:53.762 INFO 2860 --- [main] o.s.c.n.e.s.
EurekaServiceRegistry : Registering application SPRINGCLOUD-TEST with eureka
with status UP
2021-02-10 14:03:53.763 INFO 2860 --- [main] com.netflix.discovery.
DiscoveryClient : Saw local status change event StatusChangeEvent
[timestamp=1612937033763, current=UP, previous=STARTING]
2021-02-10 14:03:53.765 INFO 2860 --- [nfoReplicator-0] com.netflix.discovery.
DiscoveryClient : DiscoveryClient_SPRINGCLOUD-TEST/host.docker.
internal:springCloud-test:8884: registering service...
2021-02-10 14:03:53.805 INFO 2860 --- [main] o.s.b.w.embedded.tomcat.
TomcatWebServer : Tomcat started on port(s): 8884 (http) with context path ''
2021-02-10 14:03:53.807 INFO 2860 --- [main] .s.c.n.e.s.
EurekaAutoServiceRegistration : Updating port to 8884
2021-02-10 14:03:53.837 INFO 2860 --- [nfoReplicator-0] com.netflix.discovery.
DiscoveryClient : DiscoveryClient_SPRINGCLOUD-TEST/host.docker.
internal:springCloud-test:8884 - registration status: 204
2021-02-10 14:03:54.231 INFO 2860 --- [main] o.d.s.e.s.
SoulTestSpringCloudApplication : Started SoulTestSpringCloudApplication in 6.338
seconds (JVM running for 7.361)

```

9.4.3 开启 springCloud 插件

- 在 soul-admin 插件管理中，把 springCloud 插件设置为开启。

9.4.4 测试 Http 请求

soul-examples-springcloud 项目成功启动之后会自动把加 @SoulSpringCloudClient 注解的接口方法注册到网关。

打开插件管理->springCloud 可以看到插件规则配置列表

SelectorList

Add

Name	Open	Operation
/springcloud	Open	Modify Delete

<

1

>

RulesList

Synchronous springCloud

Add

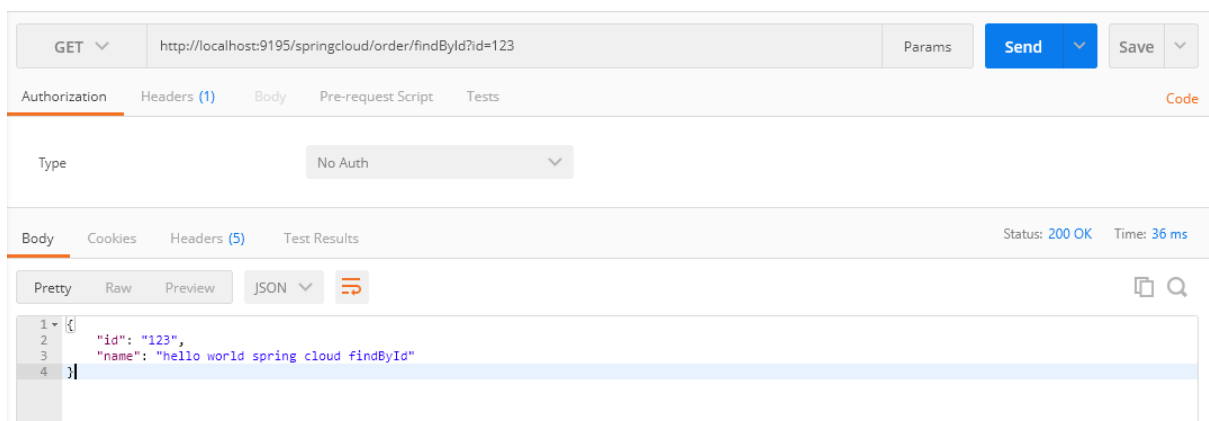
	RuleName	Open	UpdateTime	Operation
+	/springcloud/order/save	Open	2021-02-10 14:00:04	Modify Delete
+	/springcloud/order/path/**/name	Open	2021-02-10 14:00:04	Modify Delete
+	/springcloud/order/findById	Open	2021-02-10 14:00:04	Modify Delete
+	/springcloud/order/path/**	Open	2021-02-10 14:00:04	Modify Delete
+	/springcloud/test/**	Open	2021-02-10 14:00:04	Modify Delete

<

1

>

下面使用 postman 模拟 http 的方式来请求你的 SpringCloud 服务



9.5 Sofa 快速开始

本文档将演示了如何快速使用 Sofa 接入 Soul 网关。您可以直接在工程下找到本文档的示例代码。

9.5.1 环境准备

请参考配置网关环境并启动 soul-admin 和 soul-bootstrap，另外使用 zookeeper 需提前下载启动。

注：启动 soul-bootstrap 之前需要引入 sofa 依赖

```
<!-- soul sofa plugin starter-->
<dependency>
  <groupId>com.alipay.sofa</groupId>
```

```

        <artifactId>sofa-rpc-all</artifactId>
        <version>${sofa.rpc.version}</version>
    </dependency>
    <dependency>
        <groupId>org.dromara</groupId>
        <artifactId>soul-spring-boot-starter-plugin-sofa</artifactId>
        <version>${project.version}</version>
    </dependency>

    <!-- zookeeper -->
    <dependency>
        <groupId>org.apache.curator</groupId>
        <artifactId>curator-client</artifactId>
        <version>4.0.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.curator</groupId>
        <artifactId>curator-framework</artifactId>
        <version>4.0.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.curator</groupId>
        <artifactId>curator-recipes</artifactId>
        <version>4.0.1</version>
    </dependency>
    <!-- soul sofa plugin end-->

```

9.5.2 运行 soul-examples-sofa 项目

下载 [soul-examples-dubbo](#)，调整 `application.yml` 的 zk 注册地址为你本地，如：

```

com:
  alipay:
    sofa:
      rpc:
        registry-address: zookeeper://127.0.0.1:2181

```

运行 `org.dromara.soul.examples.sofa.service.TestSofaApplicationmain` 方法启动 sofa 服务。

成功启动会有如下日志：

```

2021-02-10 02:31:45.599 INFO 2156 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : sofa client register success: {"appName":"sofa","contextPath":"/
sofa","path":"/sofa/insert","pathDesc":"Insert a row of data","rpcType":"sofa",
"serviceName":"org.dromara.soul.examples.sofa.api.service.SofaSingleParamService",
"methodName":"insert","ruleName":"/sofa/insert","parameterTypes":"org.dromara.soul.
examples.sofa.api.entity.SofaSimpleTypeBean","rpcExt":{"loadbalance":"hash",
"retries":3,"timeout":-1},"enabled":true}

```

```

2021-02-10 02:31:45.605 INFO 2156 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : sofa client register success: {"appName":"sofa","contextPath":"/
sofa","path":"/sofa/findById","pathDesc":"Find by Id","rpcType":"sofa","serviceName
":"org.dromara.soul.examples.sofa.api.service.SofaSingleParamService","methodName":
"findById","ruleName":"/sofa/findById","parameterTypes":"java.lang.String","rpcExt
":{"loadbalance\":"hash\","retries\":3,"timeout\":-1},"enabled":true}
2021-02-10 02:31:45.611 INFO 2156 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : sofa client register success: {"appName":"sofa","contextPath":"/
sofa","path":"/sofa/findAll","pathDesc":"Get all data","rpcType":"sofa",
"serviceName":"org.dromara.soul.examples.sofa.api.service.SofaSingleParamService",
"methodName":"findAll","ruleName":"/sofa/findAll","parameterTypes":"","rpcExt":{"
loadbalance\":"hash\","retries\":3,"timeout\":-1},"enabled":true}
2021-02-10 02:31:45.616 INFO 2156 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : sofa client register success: {"appName":"sofa","contextPath":"/
sofa","path":"/sofa/batchSaveNameAndId","pathDesc":"","rpcType":"sofa","serviceName
":"org.dromara.soul.examples.sofa.api.service.SofaMultiParamService","methodName":
"batchSaveNameAndId","ruleName":"/sofa/batchSaveNameAndId","parameterTypes":"java.
util.List,java.lang.String,java.lang.String#org.dromara.soul.examples.sofa.api.
entity.SofaSimpleTypeBean","rpcExt":{"loadbalance\":"hash\","retries\":3,
"timeout\":-1},"enabled":true}
2021-02-10 02:31:45.621 INFO 2156 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : sofa client register success: {"appName":"sofa","contextPath":"/
sofa","path":"/sofa/saveComplexBeanAndName","pathDesc":"","rpcType":"sofa",
"serviceName":"org.dromara.soul.examples.sofa.api.service.SofaMultiParamService",
"methodName":"saveComplexBeanAndName","ruleName":"/sofa/saveComplexBeanAndName",
"parameterTypes":"org.dromara.soul.examples.sofa.api.entity.SofaComplexTypeBean,
java.lang.String","rpcExt":{"loadbalance\":"hash\","retries\":3,"timeout\":-1}
","enabled":true}
2021-02-10 02:31:45.627 INFO 2156 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : sofa client register success: {"appName":"sofa","contextPath":"/
sofa","path":"/sofa/findByIdsAndName","pathDesc":"","rpcType":"sofa",
"serviceName":"org.dromara.soul.examples.sofa.api.service.SofaMultiParamService",
"methodName":"findByIdsAndName","ruleName":"/sofa/findByIdsAndName",
"parameterTypes":"[Ljava.lang.Integer;java.lang.String","rpcExt":{"loadbalance\
\":"hash\","retries\":3,"timeout\":-1},"enabled":true}
2021-02-10 02:31:45.632 INFO 2156 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : sofa client register success: {"appName":"sofa","contextPath":"/
sofa","path":"/sofa/findByStringArray","pathDesc":"","rpcType":"sofa","serviceName
":"org.dromara.soul.examples.sofa.api.service.SofaMultiParamService","methodName":
"findByStringArray","ruleName":"/sofa/findByStringArray","parameterTypes":"[Ljva.
lang.String;","rpcExt":{"loadbalance\":"hash\","retries\":3,"timeout\":-1},
"enabled":true}
2021-02-10 02:31:45.637 INFO 2156 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : sofa client register success: {"appName":"sofa","contextPath":"/
sofa","path":"/sofa/saveTwoList","pathDesc":"","rpcType":"sofa","serviceName":"org.
dromara.soul.examples.sofa.api.service.SofaMultiParamService","methodName":
"saveTwoList","ruleName":"/sofa/saveTwoList","parameterTypes":"java.util.List,java.
util.Map#org.dromara.soul.examples.sofa.api.entity.SofaComplexTypeBean","rpcExt":
{"loadbalance\":"hash\","retries\":3,"timeout\":-1},"enabled":true}

```

```

2021-02-10 02:31:45.642 INFO 2156 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : sofa client register success: {"appName":"sofa","contextPath":"/
sofa","path":"/sofa/batchSave","pathDesc":"","rpcType":"sofa","serviceName":"org.
dromara.soul.examples.sofa.api.service.SofaMultiParamService","methodName":
"batchSave","ruleName":"/sofa/batchSave","parameterTypes":"java.util.List#org.
dromara.soul.examples.sofa.api.entity.SofaSimpleTypeBean","rpcExt":{"loadbalance\
":"hash","\retries\":3,"\timeout\":-1},"enabled":true}
2021-02-10 02:31:45.647 INFO 2156 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : sofa client register success: {"appName":"sofa","contextPath":"/
sofa","path":"/sofa/findById","pathDesc":"","rpcType":"sofa","serviceName":
"org.dromara.soul.examples.sofa.api.service.SofaMultiParamService","methodName":
"findById","ruleName":"/sofa/findById","parameterTypes":"java.util.List",
"rpcExt":{"loadbalance\":"hash","\retries\":3,"\timeout\":-1},"enabled":true}
2021-02-10 02:31:45.653 INFO 2156 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : sofa client register success: {"appName":"sofa","contextPath":"/
sofa","path":"/sofa/saveComplexBean","pathDesc":"","rpcType":"sofa","serviceName":
"org.dromara.soul.examples.sofa.api.service.SofaMultiParamService","methodName":
"saveComplexBean","ruleName":"/sofa/saveComplexBean","parameterTypes":"org.dromara.
soul.examples.sofa.api.entity.SofaComplexTypeBean","rpcExt":{"loadbalance\":"
hash","\retries\":3,"\timeout\":-1},"enabled":true}
2021-02-10 02:31:45.660 INFO 2156 --- [pool-1-thread-1] o.d.s.client.common.utils.
RegisterUtils : sofa client register success: {"appName":"sofa","contextPath":"/
sofa","path":"/sofa/findByIdsAndName","pathDesc":"","rpcType":"sofa","serviceName":
"org.dromara.soul.examples.sofa.api.service.SofaMultiParamService","methodName":
"findByIdsAndName","ruleName":"/sofa/findByIdsAndName","parameterTypes":"java.util.
List,java.lang.String","rpcExt":{"loadbalance\":"hash","\retries\":3,"\timeout\
":-1},"enabled":true}
2021-02-10 02:31:46.055 INFO 2156 --- [ main] o.a.c.fimps.
CuratorFrameworkImpl : Starting
2021-02-10 02:31:46.059 INFO 2156 --- [ main] org.apache.zookeeper.
ZooKeeper : Client environment:zookeeper.version=3.4.6-1569965, built on
02/20/2014 09:09 GMT
2021-02-10 02:31:46.059 INFO 2156 --- [ main] org.apache.zookeeper.
ZooKeeper : Client environment:host.name=host.docker.internal
2021-02-10 02:31:46.059 INFO 2156 --- [ main] org.apache.zookeeper.
ZooKeeper : Client environment:java.version=1.8.0_211
2021-02-10 02:31:46.059 INFO 2156 --- [ main] org.apache.zookeeper.
ZooKeeper : Client environment:java.vendor=Oracle Corporation
2021-02-10 02:31:46.059 INFO 2156 --- [ main] org.apache.zookeeper.
ZooKeeper : Client environment:java.home=C:\Program Files\Java\jdk1.8.0_
211\jre
2021-02-10 02:31:46.059 INFO 2156 --- [ main] org.apache.zookeeper.
ZooKeeper : Client environment:java.class.path=C:\Program Files\Java\
jdk1.8.0_211\jre\lib\charsets.jar;C:\Program Files\Java\jdk1.8.0_211\jre\lib\
deploy.jar;C:\Program Files\Java\jdk1.8.0_211\jre\lib\ext\access-bridge-64.jar;C:\
Program Files\Java\jdk1.8.0_211\jre\lib\ext\cldrdata.jar;C:\Program Files\Java\
jdk1.8.0_211\jre\lib\ext\dnsns.jar;C:\Program Files\Java\jdk1.8.0_211\jre\lib\ext\
jaccess.jar;C:\Program Files\Java\jdk1.8.0_211\jre\lib\ext\jfxrt.jar;C:\Program
Files\Java\jdk1.8.0_211\jre\lib\ext\localedata.jar;C:\Program Files\Java\jdk1.8.0_
211\jre\lib\ext\nashorn.jar;C:\Program Files\Java\jdk1.8.0_211\jre\lib\ext\sunec.
95;C:\Program Files\Java\jdk1.8.0_211\jre\lib\ext\sunec_provider.jar;C:\Program 80
Files\Java\jdk1.8.0_211\jre\lib\ext\sunmscapi.jar;C:\Program Files\Java\jdk1.8.0_
211\jre\lib\ext\sunpkcs11.jar;C:\Program Files\Java\jdk1.8.0_211\jre\lib\ext\zipfs.
jar;C:\Program Files\Java\jdk1.8.0_211\jre\lib\javaws.jar;C:\Program Files\Java\

```



```

2021-02-10 02:31:46.060 INFO 2156 --- [          main] org.apache.zookeeper.
ZooKeeper          : Client environment:java.library.path=C:\Program Files\Java\
jdk1.8.0_211\bin;C:\Windows\Sun\Java\bin;C:\Windows\system32;C:\Windows;C:\Program
Files\Common Files\Oracle\Java\javapath;C:\ProgramData\Oracle\Java\javapath;C:\
Program Files (x86)\Common Files\Oracle\Java\javapath;C:\Windows\system32;C:\
Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\
Windows\System32\OpenSSH\;C:\Program Files\Java\jdk1.8.0_211\bin;C:\Program Files\
Java\jdk1.8.0_211\jre\bin;D:\SOFT\apache-maven-3.5.0\bin;C:\Program Files\Go\bin;
C:\Program Files\nodejs\;C:\Program Files\Python\Python38\;C:\Program Files\
OpenSSL-Win64\bin;C:\Program Files\Git\bin;D:\SOFT\protobuf-2.5.0\src;D:\SOFT\zlib-
1.2.8;c:\Program Files (x86)\Microsoft SQL Server\100\Tools\Binn\;c:\Program Files\
Microsoft SQL Server\100\Tools\Binn\;c:\Program Files\Microsoft SQL Server\100\DTS\
Binn\;C:\Program Files\Docker\Docker\resources\bin;C:\ProgramData\DockerDesktop\
version-bin;D:\SOFT\gradle-6.0-all\gradle-6.0\bin;C:\Program Files\mingw-w64\x86_
64-8.1.0-posix-seh-rt_v6-rev0\mingw64\bin;D:\SOFT\hugo_extended_0.55.5_Windows-
64bit;C:\Users\DLM\AppData\Local\Microsoft\WindowsApps;C:\Users\DLM\go\bin;C:\
Users\DLM\AppData\Roaming\npm;;C:\Program Files\Microsoft VS Code\bin;C:\Program
Files\nimble-cli\bin;.
2021-02-10 02:31:46.060 INFO 2156 --- [          main] org.apache.zookeeper.
ZooKeeper          : Client environment:java.io.tmpdir=C:\Users\DLM\AppData\Local\
Temp\
2021-02-10 02:31:46.060 INFO 2156 --- [          main] org.apache.zookeeper.
ZooKeeper          : Client environment:java.compiler=<NA>
2021-02-10 02:31:46.060 INFO 2156 --- [          main] org.apache.zookeeper.
ZooKeeper          : Client environment:os.name=Windows 10
2021-02-10 02:31:46.060 INFO 2156 --- [          main] org.apache.zookeeper.
ZooKeeper          : Client environment:os.arch=amd64
2021-02-10 02:31:46.060 INFO 2156 --- [          main] org.apache.zookeeper.
ZooKeeper          : Client environment:os.version=10.0
2021-02-10 02:31:46.060 INFO 2156 --- [          main] org.apache.zookeeper.
ZooKeeper          : Client environment:user.name=DLM
2021-02-10 02:31:46.060 INFO 2156 --- [          main] org.apache.zookeeper.
ZooKeeper          : Client environment:user.home=C:\Users\DLM
2021-02-10 02:31:46.060 INFO 2156 --- [          main] org.apache.zookeeper.
ZooKeeper          : Client environment:user.dir=D:\X\dml_github\soul
2021-02-10 02:31:46.061 INFO 2156 --- [          main] org.apache.zookeeper.
ZooKeeper          : Initiating client connection, connectString=127.0.0.1:21810
sessionTimeout=60000 watcher=org.apache.curator.ConnectionState@3e850122
2021-02-10 02:31:46.069 INFO 2156 --- [27.0.0.1:21810] org.apache.zookeeper.
ClientCnxn          : Opening socket connection to server 127.0.0.1/127.0.0.
1:21810. Will not attempt to authenticate using SASL (unknown error)
2021-02-10 02:31:46.071 INFO 2156 --- [27.0.0.1:21810] org.apache.zookeeper.
ClientCnxn          : Socket connection established to 127.0.0.1/127.0.0.1:21810,
initiating session
2021-02-10 02:31:46.078 INFO 2156 --- [27.0.0.1:21810] org.apache.zookeeper.
ClientCnxn          : Session establishment complete on server 127.0.0.1/127.0.0.
1:21810, sessionId = 0x10005b0d05e0001, negotiated timeout = 40000
2021-02-10 02:31:46.081 INFO 2156 --- [ain-EventThread] o.a.c.f.state.
ConnectionStateManager : State change: CONNECTED

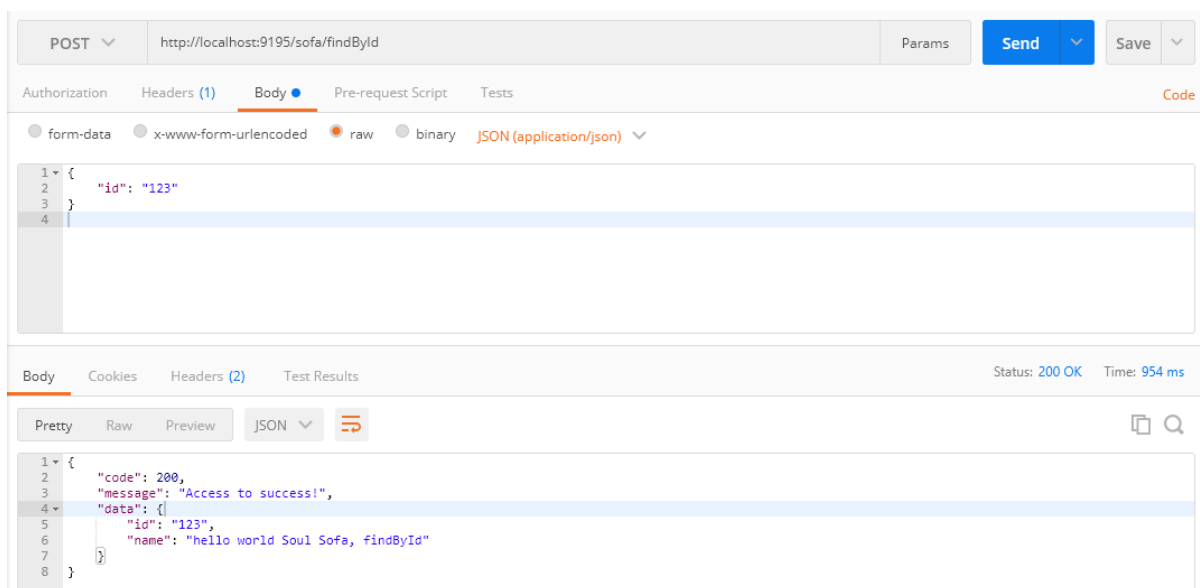
```

- 首先在 soul-admin 插件管理中，把 sofa 插件设置为开启。
- 其次在 sofa 插件中配置你的注册地址，或者其他注册中心的地址。

soul-examples-sofa 项目成功启动之后会自动把加 @SoulSofaClient 注解的接口方法注册到网关。

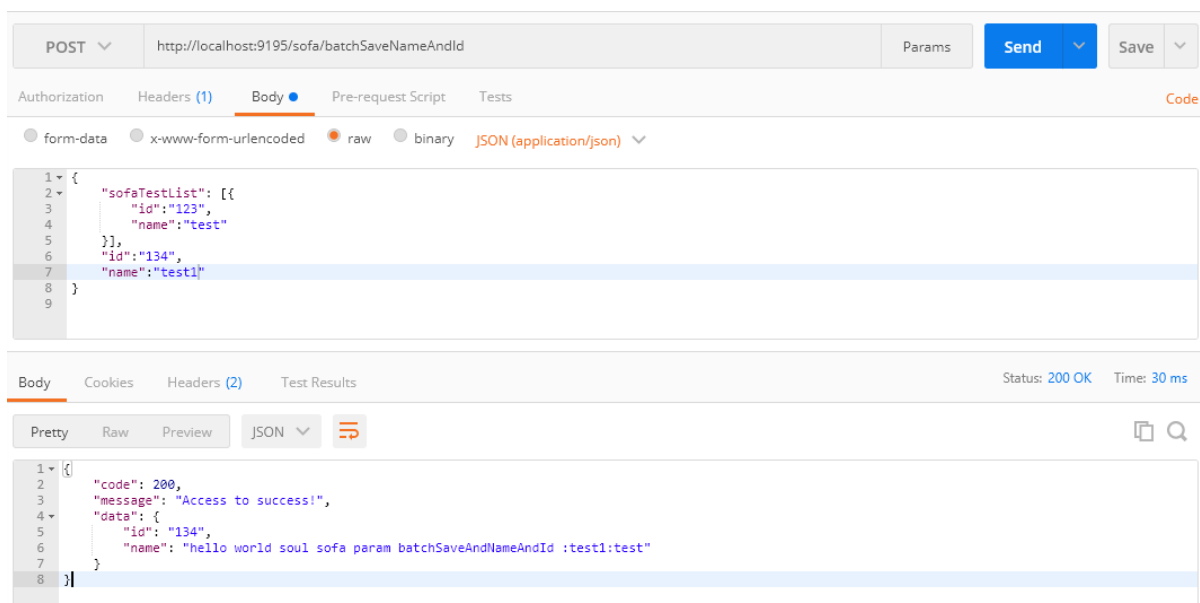
SelectorList			RulesList				
Name	Open	Operation					
/sofa	Open	Modify Delete					

9.5. Sofa 快速开始



复杂多参数示例：对应接口实现类为 `org.dromara.soul.examples.sofa.service.impl.SofaMultiParamServiceImpl#batchSaveNameAndId`

```
@Override
@SoulSofaClient(path = "/batchSaveNameAndId")
public SofaSimpleTypeBean batchSaveNameAndId(final List<SofaSimpleTypeBean>
sofaTestList, final String id, final String name) {
    SofaSimpleTypeBean simpleTypeBean = new SofaSimpleTypeBean();
    simpleTypeBean.setId(id);
    simpleTypeBean.setName("hello world soul sofa param batchSaveAndNameAndId :" +
name + ":" + sofaTestList.stream().map(SofaSimpleTypeBean::getName).
collect(Collectors.joining("-")));
    return simpleTypeBean;
}
```



9.6 Tars 快速开始

本文档将演示如何快速使用 Tars 接入 Soul 网关。您可以直接在工程下找到本文档的[示例代码](#)。

9.6.1 环境准备

请参考[配置网关环境](#)并启动 soul-admin 和 soul-bootstrap。

注：soul-bootstrap 需要引入 tars 依赖

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-tars</artifactId>
  <version>${project.version}</version>
</dependency>

<dependency>
  <groupId>com.tencent.tars</groupId>
  <artifactId>tars-client</artifactId>
  <version>1.7.2</version>
</dependency>
```

9.6.2 运行 soul-examples-tars 项目

下载[soul-examples-tars](#)

修改 application.yml 中的 host 为你本地 ip

修改配置 src/main/resources/SoulExampleServer.SoulExampleApp.config.conf:

- 建议弄清楚 config 的主要配置项含义, 参考开发指南
- config 中的 ip 要注意提供成本机的
- local=..., 表示开放的本机给 tarsnode 连接的端口, 如果没有 tarsnode, 可以掉这项配置
- locator: registry 服务的地址, 必须是有 ip port 的, 如果不需要 registry 来定位服务, 则不需要配置;
- node=tars.tarsnode.ServerObj@xxxx, 表示连接的 tarsnode 的地址, 如果本地没有 tarsnode, 这项配置可以去掉

更多 config 配置说明请参考[Tars 官方文档](#)

运行 org.dromara.soul.examples.tars.SoulTestTarsApplicationmain 方法启动项目。

注: 服务启动时需要在启动命令中指定配置文件地址 **-Dconfig=xxx/SoulExampleServer.SoulExampleApp.config.conf**

如果不加-Dconfig 参数配置会可能会如下抛异常:

```

com.qq.tars.server.config.ConfigurationException: error occurred on load server
config
    at com.qq.tars.server.config.ConfigurationManager.
loadServerConfig(ConfigurationManager.java:113)
    at com.qq.tars.server.config.ConfigurationManager.init(ConfigurationManager.
java:57)
    at com.qq.tars.server.core.Server.loadServerConfig(Server.java:90)
    at com.qq.tars.server.core.Server.<init>(Server.java:42)
    at com.qq.tars.server.core.Server.<clinit>(Server.java:38)
    at com.qq.tars.spring.bean.PropertiesListener.
onApplicationEvent(PropertiesListener.java:37)
    at com.qq.tars.spring.bean.PropertiesListener.
onApplicationEvent(PropertiesListener.java:31)
    at org.springframework.context.event.SimpleApplicationEventMulticaster.
doInvokeListener(SimpleApplicationEventMulticaster.java:172)
    at org.springframework.context.event.SimpleApplicationEventMulticaster.
invokeListener(SimpleApplicationEventMulticaster.java:165)
    at org.springframework.context.event.SimpleApplicationEventMulticaster.
multicastEvent(SimpleApplicationEventMulticaster.java:139)
    at org.springframework.context.event.SimpleApplicationEventMulticaster.
multicastEvent(SimpleApplicationEventMulticaster.java:127)
    at org.springframework.boot.context.event.EventPublishingRunListener.
environmentPrepared(EventPublishingRunListener.java:76)
    at org.springframework.boot.SpringApplicationRunListeners.
environmentPrepared(SpringApplicationRunListeners.java:53)
    at org.springframework.boot.SpringApplication.
prepareEnvironment(SpringApplication.java:345)
    at org.springframework.boot.SpringApplication.run(SpringApplication.java:308)
    at org.springframework.boot.SpringApplication.run(SpringApplication.java:1226)
    at org.springframework.boot.SpringApplication.run(SpringApplication.java:1215)
    at org.dromara.soul.examples.tars.SoulTestTarsApplication.
main(SoulTestTarsApplication.java:38)
Caused by: java.lang.NullPointerException
    at java.io.FileInputStream.<init>(FileInputStream.java:130)
    at java.io.FileInputStream.<init>(FileInputStream.java:93)
    at com.qq.tars.common.util.Config.parseFile(Config.java:211)
    at com.qq.tars.server.config.ConfigurationManager.
loadServerConfig(ConfigurationManager.java:63)
    ... 17 more
The exception occurred at load server config

```

成功启动会有如下日志:

```

[SERVER] server starting at tcp -h 127.0.0.1 -p 21715 -t 60000...
[SERVER] server started at tcp -h 127.0.0.1 -p 21715 -t 60000...
[SERVER] server starting at tcp -h 127.0.0.1 -p 21714 -t 3000...
[SERVER] server started at tcp -h 127.0.0.1 -p 21714 -t 3000...
[SERVER] The application started successfully.

```

```

The session manager service started...
[SERVER] server is ready...
2021-02-09 13:28:24.643 INFO 16016 --- [main] o.s.b.w.embedded.tomcat.
TomcatWebServer : Tomcat started on port(s): 55290 (http) with context path ''
2021-02-09 13:28:24.645 INFO 16016 --- [main] o.d.s.e.tars.
SoulTestTarsApplication : Started SoulTestTarsApplication in 4.232 seconds (JVM
running for 5.1)
2021-02-09 13:28:24.828 INFO 16016 --- [pool-2-thread-1] o.d.s.client.common.
utils.RegisterUtils : tars client register success: {"appName":"127.0.0.1:21715",
"contextPath":"/tars","path":"/tars/helloInt","pathDesc":"","rpcType":"tars",
"serviceName":"SoulExampleServer.SoulExampleApp.HelloObj","methodName":"helloInt",
"ruleName":"/tars/helloInt","parameterTypes":"int,java.lang.String","rpcExt":{"\
"methodInfo":[{"methodName":"helloInt","params":[{}],\returnType":"\
"java.lang.Integer"},{"methodName":"hello","params":[{}],\returnType":"\
"java.lang.String"}]},"enabled":true}
2021-02-09 13:28:24.837 INFO 16016 --- [pool-2-thread-1] o.d.s.client.common.
utils.RegisterUtils : tars client register success: {"appName":"127.0.0.1:21715",
"contextPath":"/tars","path":"/tars/hello","pathDesc":"","rpcType":"tars",
"serviceName":"SoulExampleServer.SoulExampleApp.HelloObj","methodName":"hello",
"ruleName":"/tars/hello","parameterTypes":"int,java.lang.String","rpcExt":{"\
"methodInfo":[{"methodName":"helloInt","params":[{}],\returnType":"\
"java.lang.Integer"},{"methodName":"hello","params":[{}],\returnType":"\
"java.lang.String"}]},"enabled":true}

```

9.6.3 tars 插件设置

- 在 soul-admin 插件管理中，把 tars 插件设置为开启。

9.6.4 测试

soul-examples-tars 项目成功启动之后会自动把加 @SoulTarsClient 注解的接口方法注册到网
关。

打开插件管理->tars 可以看到插件规则配置列表

SelectorList			RulesList			
			Synchronous tars			
Name	Open	Operation	RuleName	Open	UpdateTime	Operation
/tars	Open	Modify Delete	/tars/helloInt	Open	2021-02-09 13:15:27	Modify Delete
			/tars/hello	Open	2021-02-09 13:15:27	Modify Delete

下面使用 postman 模拟 http 的方式来请求你的 tars 服务

The screenshot displays a REST client interface with the following components:

- Request Section:**
 - Method: **POST**
 - URL: `http://localhost:9195/tars/hello`
 - Buttons: **Params**, **Send**, **Save**
 - Tabs: **Authorization**, **Headers (1)**, **Body** (selected), **Pre-request Script**, **Tests**
 - Body Type: **JSON (application/json)** (selected from a dropdown)
 - Request Body (JSON):

```
1 {  
2   "no": "123",  
3   "name": "test"  
4 }  
5
```
- Response Section:**
 - Tabs: **Body** (selected), **Cookies**, **Headers (2)**, **Test Results**
 - Status: **200 OK**, Time: **30 ms**
 - Response Format: **JSON** (selected from a dropdown)
 - Response Body (JSON):

```
1 {  
2   "code": 200,  
3   "message": "Access to success!",  
4   "data": "hello no=123, name=test, time=1612867753446"  
5 }
```

10.1 Divide 插件

10.1.1 说明

- divide 插件是网关处理 http 协议请求的核心处理插件。

10.1.2 插件设置

- 开启插件，soul-admin -> 插件管理 -> divide 设置为启用。
- divide 插件，配合如下 starter 一起才能生效，具体请看：[http 用户](#)。

```
<!--if you use http proxy start this-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-divide</artifactId>
  <version>${last.version}</version>
</dependency>

<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-httpclient</artifactId>
  <version>${last.version}</version>
</dependency>
```


10.1.3 插件讲解

- divide 插件是进行 http 正向代理的插件，所有 http 类型的请求，都是由该插件进行负载均衡的调用。
- 选择器和规则，请详细看：[选择器规则](#)。
- http 配置，是网关匹配到流量以后，真实调用的 http 配置，可以配置多个，设置负载均衡权重，具体的负载均衡策略，在规则中指定。
 - 配置详解：
 - * 第一个框：hostName，一般填写 localhost，该字段暂时没使用。
 - * 第二个框：http 协议，一般填写 http:// 或者 https://，不填写默认为 http://
 - * 第三个框：ip 与端口，这里填写你真实服务的 ip + 端口。
 - * 第四个框：负载均衡权重。
 - ip + port 检测
 - * 在 soul-admin 会有一个定时任务来扫描配置的 ip 端口，如果发现下线，则会删除该 ip + port
 - * 可以进行如下配置：

```
soul.upstream.check:true 默认为 true，设置为 false，不检测
soul.upstream.scheduledTime:10 定时检测时间间隔，默认 10 秒
```

10.2 Dubbo 插件

10.2.1 说明

- dubbo 插件是将 http 协议转换成 dubbo 协议的插件，也是网关实现 dubbo 泛化调用的关键。
- dubbo 插件需要配合元数据才能实现 dubbo 的调用，具体请看：[元数据](#)。
- apache dubbo 和 alibaba dubbo 用户，都是使用该同一插件。

```
<!--if you use dubbo start this-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-alibaba-dubbo</artifactId>
  <version>${last.version}</version>
</dependency>

<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-apache-dubbo</artifactId>
  <version>${last.version}</version>
</dependency>
```

10.2.2 插件设置

- 在 soul-admin -> 插件管理-> dubbo 设置为开启。
- 在 dubbo 插件的配置中，配置如下：配置 dubbo 的注册中心。

```
{"register":"zookeeper://localhost:2181"} or {"register":"nacos://localhost:8848"}
```

- 插件需要配合依赖 starter 进行使用, 具体请看: [dubbo 用户](#)。
- 选择器和规则，请详细看: [选择器规则](#)。

10.2.3 元数据

- 每一个 dubbo 接口方法，都会对应一条元数据，可以在 soul-admin -> 元数据管理，进行查看。
- 路径：就是你 http 请求的路径。
- rpc 扩展参数，对应为 dubbo 接口的一些配置，调整的化，请在这里修改，支持 json 格式，以下字段：

```
{"timeout":10000,"group":"","version":"","loadbalance":"","retries":1}
```

10.3 SpringCloud 插件

10.3.1 说明

- 该插件是用来将 http 协议转成 springCloud 协议的核心。

10.3.2 引入网关 springCloud 的插件支持

- 在网关的 pom.xml 文件中引入如下依赖。

```
<!--soul springCloud plugin start-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-springcloud</artifactId>
  <version>${last.version}</version>
</dependency>

<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-httpclient</artifactId>
  <version>${last.version}</version>
</dependency>
<!--soul springCloud plugin end-->
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-commons</artifactId>
  <version>2.2.0.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
  <version>2.2.0.RELEASE</version>
</dependency>
```

10.3.3 插件设置

- 在 soul-admin -> 插件管理 -> springCloud, 设置为开启。
- 插件需要配合依赖 starter 进行使用, 具体请看: [springCloud 用户](#)。
- 选择器和规则, 请详细看: [选择器规则](#)。

10.3.4 详解

- 应用名称: 就是你根据条件匹配以后, 需要调用的你的具体的应用名称。
- soul 会从 springCloud 的注册中心上面, 根据应用名称获取对应的服务真实 ip 地址, 发起 http 代理调用。

10.4 Sofa 插件

10.4.1 说明

- sofa 插件是将 http 协议转换成 sofa 协议的插件, 也是网关实现 sofa 泛化调用的关键。
- sofa 插件需要配合元数据才能实现 dubbo 的调用, 具体请看: [元数据](#)。

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-sofa</artifactId>
  <version>${last.version}</version>
</dependency>
```

10.4.2 插件设置

- 在 soul-admin -> 插件管理-> sofa 设置为开启。
- 在 sofa 插件的配置中，配置如下：配置 sofa 的注册中心。

```
{"protocol":"zookeeper","register":"127.0.0.1:2181"}
```

- 插件需要配合依赖 starter 进行使用，具体请看：[sofa 用户](#)。
- 选择器和规则，请详细看：[选择器规则](#)。

10.4.3 元数据

- 每一个 sofa 接口方法，都会对应一条元数据，可以在 soul-admin -> 元数据管理，进行查看。
- 路径：就是你 http 请求的路径。
- rpc 扩展参数，对应为 sofa 接口的一些配置，调整的话，请在这里修改，支持 json 格式，以下字段：

```
{"loadbalance":"hash","retries":3,"timeout":-1}
```

10.5 限流插件

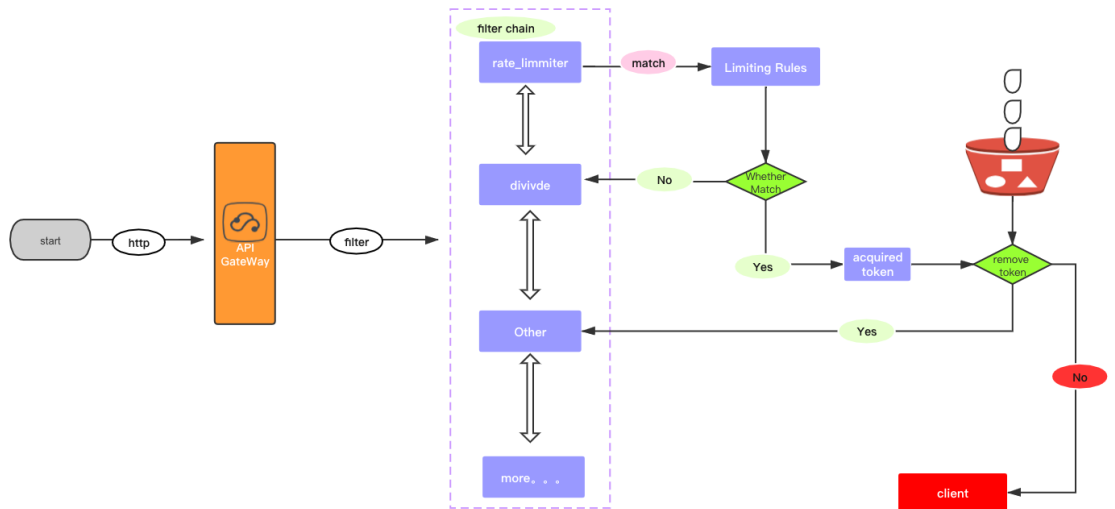
10.5.1 说明

- 限流插件，是网关对流量管控限制核心的实现。
- soul 网关提供了多种限流算法的实现，包括令牌桶算法、并发的令牌桶算法、漏桶算法、滑动时间窗口算法。
- soul 网关的限流算法实现都是基于 redis。
- 可以到接口级别，也可以到参数级别，具体怎么用，还得看你对流量配置。

10.5.2 技术方案

采用 **redis** 令牌桶算法进行限流。

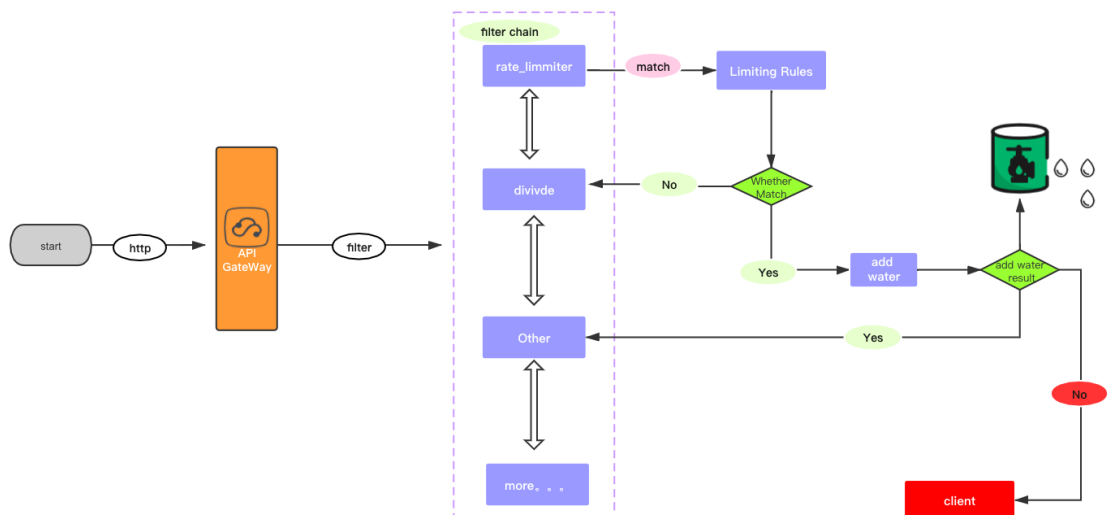
- 系统以恒定的速率产生令牌，然后将令牌放入令牌桶中。
- 令牌桶有一个容量，当令牌桶满了的时候，再向其中放入的令牌就会被丢弃。
- 每次一个请求过来，需要从令牌桶中获取一个令牌，如果有令牌，则提供服务；如果没有令牌，则拒绝服务。



• 流程图:

采用 **redis** 漏桶算法进行限流。

- 水（请求）先进入到漏桶里，漏桶以一定的速度出水，当水流入速度过大会直接溢出（拒绝服务）



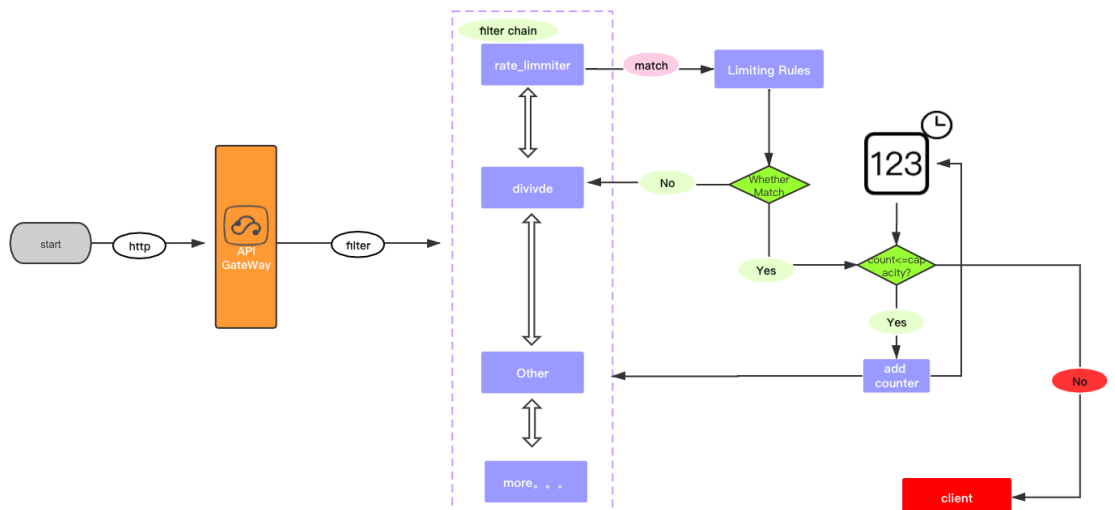
• 流程图:

基于 redis 实现的滑动窗口算法

- 滑动时间窗口通过维护一个单位时间内的计数值，每当一个请求通过时，就将计数值加 1，当计数值超过预先设定的阈值时，就拒绝单位时间内的其他请求。如果单位时间已经结束，则将计数器清零，开启下一轮的计数。



- 算法图：



- 流程图：

10.5.3 插件设置

- 在 soul-admin-> 插件管理-> rate_limiter 将其设置为开启。
- 在插件中，对 redis 进行配置。
- 目前支持 redis 的单机，哨兵，以及集群模式。
- 如果是哨兵，集群等多节点的，在 URL 中的配置，请对每个实例使用 ; 分割。 如 192.168.1.1:6379;192.168.1.2:6379。
- 如果用户无需使用，在 admin 后台把插件禁用。

10.5.4 插件使用

- 在网关的 pom.xml 文件中添加 rateLimiter 的支持。

```
<!-- soul ratelimiter plugin start-->
<dependency>
    <groupId>org.dromara</groupId>
    <artifactId>soul-spring-boot-starter-plugin-ratelimiter</artifactId>
    <version>${last.version}</version>
</dependency>
<!-- soul ratelimiter plugin end-->
```

- 选择器和规则，请详细看：[选择器规则](#)。
- 规则详细说明
 - 令牌桶算法/并发令牌桶算法

algorithmName（算法名）：tokenBucket/concurrent

replenishRate（速率）：是你允许用户每秒执行多少请求，而丢弃任何请求。这是令牌桶的填充速率。

burstCapacity（容量）：是允许用户在一秒钟内执行的最大请求数。这是令牌桶可以保存的令牌数。

- 漏桶算法

algorithmName（算法名）：leakyBucket

replenishRate（速率）：单位时间内执行请求的速率，漏桶中水滴漏出的速率。

burstCapacity（容量）：是允许用户在一秒钟内执行的最大请求数。这是桶中的水量。

- 滑动窗口算法

algorithmName（算法名）：slidingWindow

replenishRate（速率）：单位时间内执行请求的速率，用于计算时间窗口大小。

burstCapacity（容量）：时间窗口内（单位时间内）最大的请求数量。

10.6 Hystrix 插件

10.6.1 说明

- hystrix 插件是网关用来对流量进行熔断的核心实现。
- 使用信号量的方式来处理请求。

10.6.2 插件设置

- 在 soul-admin-> 插件管理-> hystrix, 设置为开启。
- 如果用户不使用, 则在 soul-admin 后台把此插件停用。

10.6.3 插件使用

- 在网关的 pom.xml 文件中添加 hystrix 的支持。

```
<!-- soul hystrix plugin start-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-hystrix</artifactId>
  <version>${last.version}</version>
</dependency>
<!-- soul hystrix plugin end-->
```

- 选择器和规则, 请详细看: [选择器规则](#)
- Hystrix 处理详解:
 - 跳闸最小请求数量: 最小的请求量, 至少要达到这个量才会触发熔断
 - 错误百分比阈值: 这段时间内, 发生异常的百分比。
 - 最大并发量: 最大的并发量
 - 跳闸休眠时间 (ms): 熔断以后恢复的时间。
 - 分组 Key: 一般设置为:contextPath
 - 命令 Key: 一般设置为具体的路径接口。
 - 失败降级 URL: 默认为 /fallback/hystrix。

10.7 Sentinel 插件

10.7.1 说明

- sentinel 插件是网关用来对流量进行限流与熔断的可选选择之一。
- sentinel 为网关熔断限流提供能力。

10.7.2 插件设置

- 在 soul-admin-> 插件管理-> sentinel, 设置为开启。
- 如果用户不使用, 则在 soul-admin 后台把此插件停用。

10.7.3 插件使用

- 在网关的 pom.xml 文件中添加 sentinel 的支持。

```
<!-- soul sentinel plugin start-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-sentinel</artifactId>
  <version>${last.version}</version>
</dependency>
<!-- soul sentinel plugin end-->
```

- 选择器和规则, 请详细看: [选择器规则](#)
- Sentinel 处理详解:
 - 是否开启流控 (1 或 0): 是否开启 sentinel 的流控。
 - 流控效果: 流控效果 (直接拒绝 / 排队等待 / 慢启动模式), 不支持按调用关系限流。
 - 限流阈值类型: 限流阈值类型, QPS 或线程数模式。
 - 是否开启熔断 (1 或 0): 是否开启 sentinel 熔断。
 - 熔断类型: 熔断策略, 支持秒级 RT/秒级异常比例/分钟级异常数。
 - 熔断阈值: 阈值。
 - 熔断窗口大小: 降级的时间, 单位为 s。
 - 熔断 URI: 熔断后的降级 uri。

10.8 Resilience4j 插件

10.8.1 说明

- resilience4j 插件是网关用来对流量进行限流与熔断的可选选择之一。
- resilience4j 为网关熔断限流提供能力。

10.8.2 插件设置

- 在 soul-admin-> 插件管理-> resilience4j, 设置为开启。
- 如果用户不使用, 则在 soul-admin 后台把此插件停用。

10.8.3 插件使用

- 在网关的 pom.xml 文件中添加 resilience4j 的支持。

```
<!-- soul resilience4j plugin start-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-resilience4j</artifactId>
  <version>${last.version}</version>
</dependency>
<!-- soul resilience4j plugin end-->
```

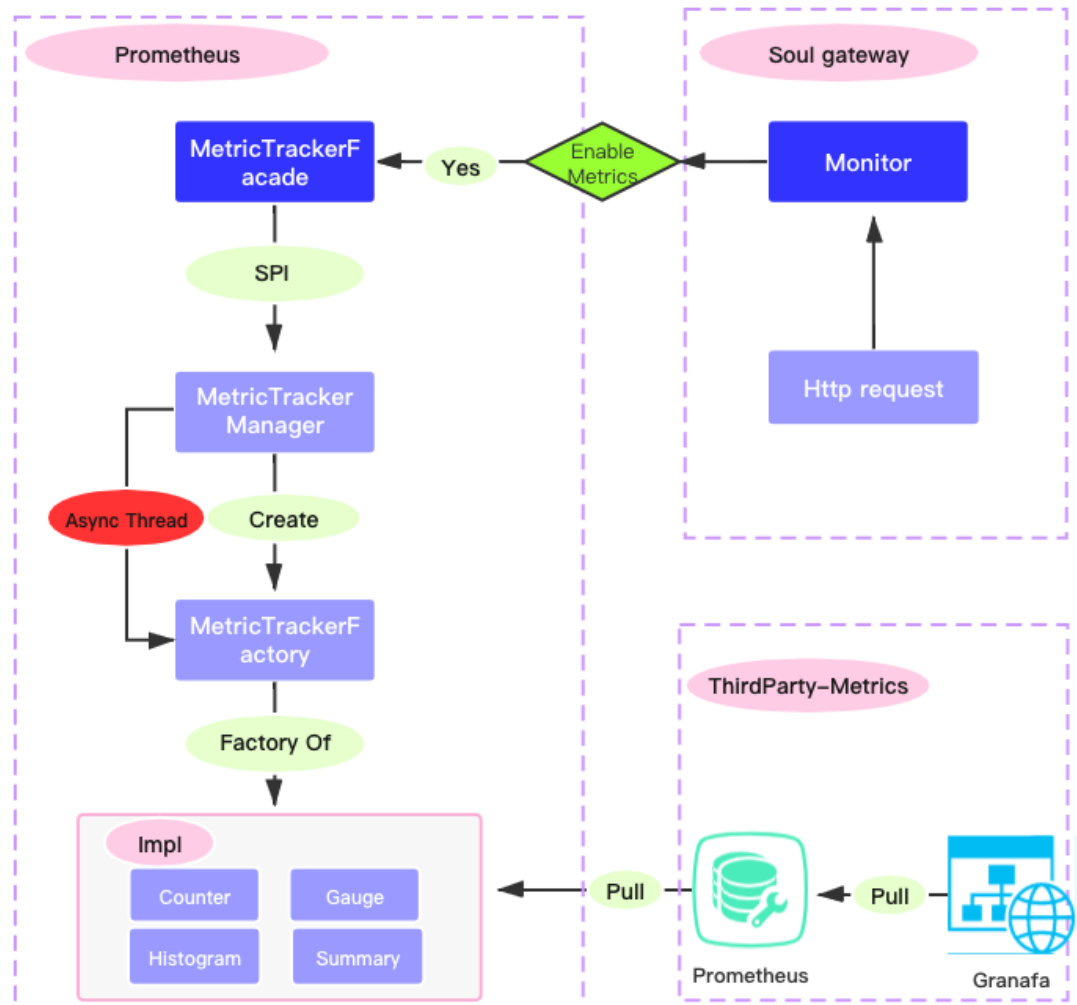
- 选择器和规则, 请详细看: [选择器规则](#)
- Resilience4j 处理详解:
 - timeoutDurationRate: 等待获取令牌的超时时间, 单位 ms, 默认值: 5000。
 - limitRefreshPeriod: 刷新令牌的时间间隔, 单位 ms, 默认值: 500。
 - limitForPeriod: 每次刷新令牌的数量, 默认值: 50。
 - circuitEnable: 是否开启熔断, 0: 关闭, 1: 开启, 默认值: 0。
 - timeoutDuration: 熔断超时时间, 单位 ms, 默认值: 30000。
 - fallbackUri: 降级处理的 uri。
 - slidingWindowSize: 滑动窗口大小, 默认值: 100。
 - slidingWindowType: 滑动窗口类型, 0: 基于计数, 1: 基于时间, 默认值: 0。
 - minimumNumberOfCalls: 开启熔断的最小请求数, 超过这个请求数才开启熔断统计, 默认值: 100。
 - waitIntervalFunctionInOpenState: 熔断器开启持续时间, 单位 ms, 默认值: 10。
 - permittedNumberOfCallsInHalfOpenState: 半开状态下的环形缓冲区大小, 必须达到此数量才会计算失败率, 默认值: 10。
 - failureRateThreshold: 错误率百分比, 达到这个阈值, 熔断器才会开启, 默认值 50。
 - automaticTransitionFromOpenToHalfOpenEnabled: 是否自动从 open 状态转换为 half-open 状态, ,true: 是, false: 否, 默认值: false。

10.9 Monitor 插件

10.9.1 说明

- monitor 插件是网关用来监控自身运行状态(JVM 相关), 请求的响应延迟, QPS、TPS 等相关 metrics。

10.9.2 技术方案



- 流程图
- 异步或者同步的方式，在 soul 网关里面进行 metrics 埋点。
- prometheus 服务端通过 http 请求来拉取 metrics，再使用 Grafana 展示。

10.9.3 插件设置

- 在 soul-admin-> 插件管理-> monitor , 设置为开启。
- 在 monitor 插件中新增以下配置

```
{"metricsName":"prometheus","host":"localhost","port":"9191","async":"true"}

# port : 为暴露给 prometheus 服务来拉取的端口
# host : 不填写则为 soul 网关的 host.
# async : "true" 为异步埋点, false 为同步埋点
```

- 如果用户不使用, 则在 soul-admin 后台把此插件停用。

10.9.4 插件使用

- 在网关的 pom.xml 文件中添加 monitor 的支持。

```
<!-- soul monitor plugin start-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-monitor</artifactId>
  <version>${last.version}</version>
</dependency>
<!-- soul monitor plugin end-->
```

- 选择器和规则, 请详细看: [选择器规则](#)。
 - 只有当匹配的 url, 才会进行 url 请求埋点。

10.9.5 metrics 信息

- 所有的 JVM, 线程, 内存, 等相关信息都会埋点, 可以在 Granfana 面板中, 新增一个 JVM 模块, 则会完全展示具体请看: https://github.com/prometheus/jmx_exporter
- 另外还有如下自定义的 metrics

名称	类型	标签名称	说明
request_total	Counter	无	收集 Soul 网关所有的请求
http_request_total	Counter	path,type	收集 monitor 插件匹配的请求

10.9.6 收集 metrics

用户需部署 Prometheus 服务来采集

- 选择对应环境的下载地址安装
- 修改配置文件: prometheus.yml

```
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped
  # from this config.
  - job_name: 'prometheus'
    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.
    static_configs:
      - targets: ['localhost:9190']
```

注: job_name 跟 monitor 插件配置的 metricsName 相对应

- 配置完成之后 window 下可以直接双击 prometheus.exe 启动即可, 默认启动端口为 9090, 可通过 <http://localhost:9090/> 验证是否成功

10.9.7 面板展示

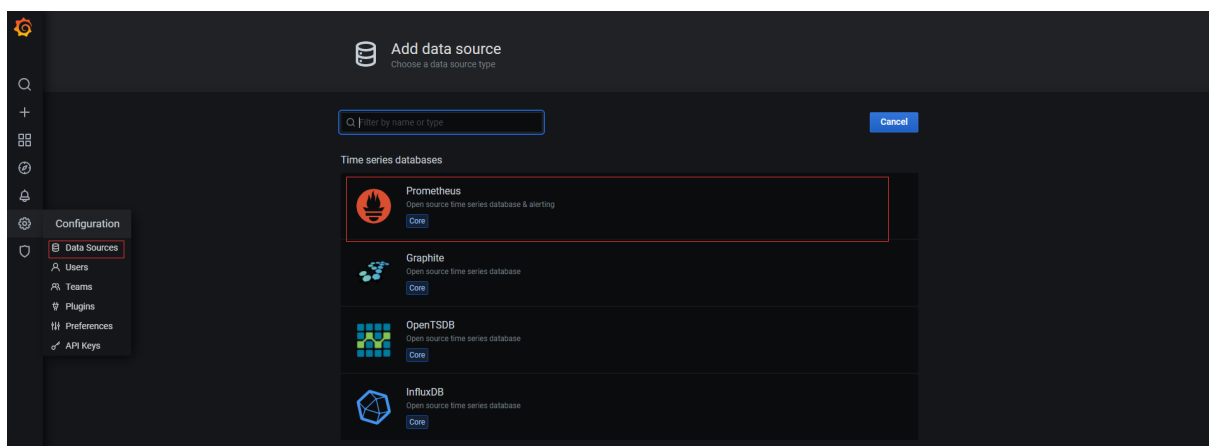
推荐使用 Grafana, 用户可以自定义查询来个性化显示面板盘。

下面介绍 Grafana 部署 (windows 版)

- 安装 Grafana

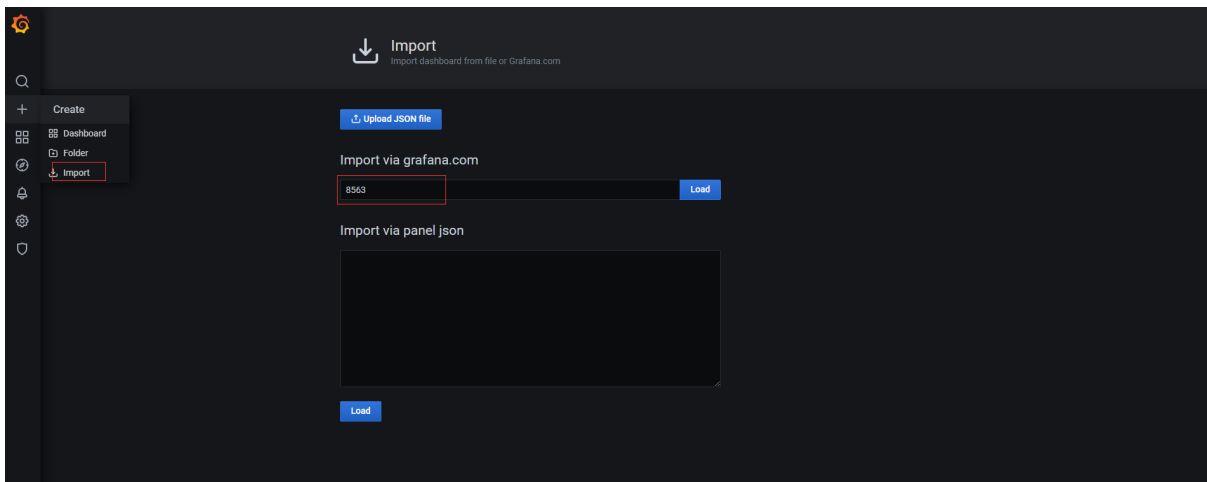
下载地址 解压进入 bin 目录然后双击 grafana-server.exe 运行访问 <http://localhost:3000/?orgId=1> admin/admin 验证是否成功

- 配置 Prometheus 数据源

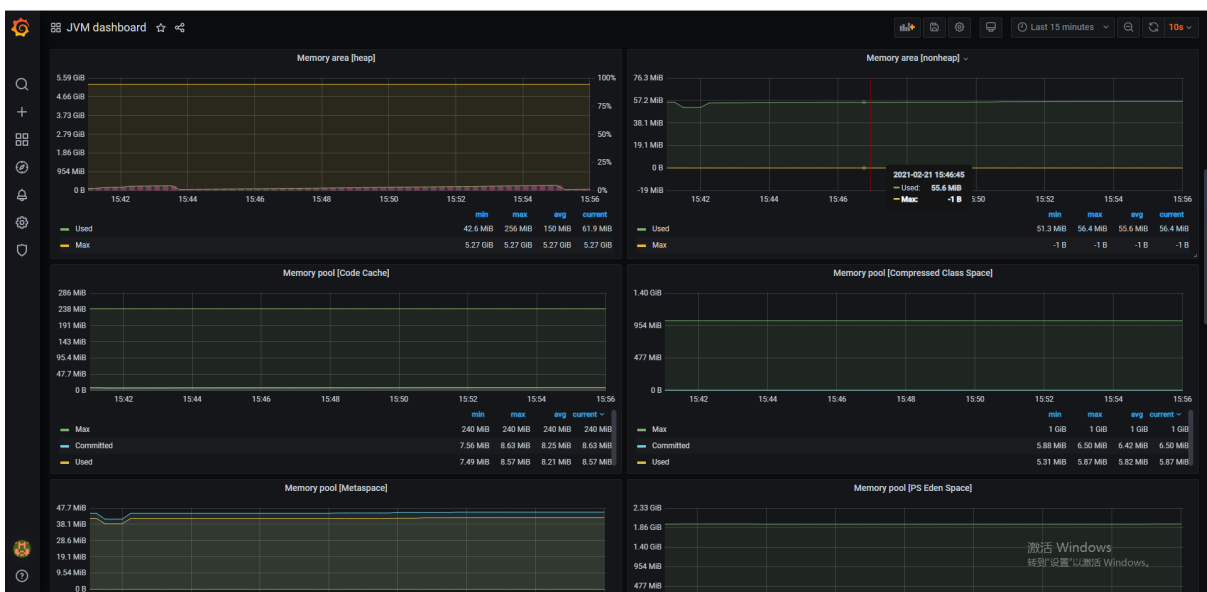


- 配置 JVM 面板

点击 Create - Import, 输入 dashboards 的 id (推荐 8563)



最终 JVM 监控面板效果如下：



- 配置自定义 metric 面板 request_total、http_request_total

点击 Create - Import, 输入 dashboards 的面板 json 配置

最终自定义 Http 请求监控面板效果如下：



10.10 Waf 插件

10.10.1 说明

- waf 插件，是网关的用来对流量实现防火墙功能的核心实现。

10.10.2 插件设置

- 在 soul-admin -> 插件管理 -> waf 设置为开启。
- 如果用户不想使用此功能，请在 admin 后台停用此插件。
- 插件编辑里面新增配置模式。

```
{"model": "black"}
```

默认为黑名单模式，设置值为 mixed 则为混合模式，下面会专门进行讲解

10.10.3 插件使用

- 在网关的 pom.xml 文件中添加 waf 的支持。

```
<!-- soul waf plugin start-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-waf</artifactId>
  <version>${last.version}</version>
</dependency>
<!-- soul waf plugin end-->
```

- 选择器和规则，请详细看：[选择器规则](#)

- 当 model 设置为 black 模式的时候，只有匹配的流量才会执行拒绝策略，不匹配的，直接会跳过。
- 当 model 设置为 mixed 模式的时候，所有的流量都会通过 waf 插件，针对不同的匹配流量，用户可以设置是拒绝，还是通过。

10.10.4 场景

- waf 插件也是 soul 的前置插件，主要用来拦截非法请求，或者异常请求，并且给与相关的拒绝策略。
- 当面对重放攻击时，你可以根据 ip 或者 host 来进行匹配，拦截掉非法的 ip 与 host，设置 reject 策略。
- 关于如何确定 ip 与 host 值，请看 [parsing-ip-and-host](#)

10.11 Sign 插件

10.11.1 说明

- sign 插件是 soul 网关自带的，用来对请求进行签名认证的插件。

10.11.2 插件设置

- 在 soul-admin-> 插件管理中-> sign 插件设置为开启。

10.11.3 插件使用

- 在网关的 pom.xml 文件中添加 sign 的支持。

```
<!-- soul sign plugin start-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-sign</artifactId>
  <version>${last.version}</version>
</dependency>
<!-- soul sign plugin end-->
```

- 选择器和规则，请详细看：[选择器规则](#)。
 - 只有匹配的请求，才会进行签名认证。

10.11.4 新增 AK/SK

- 在 soul-admin -> 认证管理中，点击新增，新增一条 AK/SK。

10.11.5 网关技术实现

- 采用 Ak/SK 鉴权技术方案。
- 采用鉴权插件，责任链的模式来完成。
- 当鉴权插件开启，并配置所有接口鉴权时候生效。

10.11.6 鉴权使用指南

- 第一步：AK/SK 由网关来进行分配，比如分配给你的 AK 为： 1TEST123456781 SK 为： 506EEB535CF740D7A755CB4B9F4A1536
- 第二步：确定好你要访问的网关路径比如 /api/service/abc
- 第三步：构造参数（以下是通用参数）

字段	值	描述
timestamp	当前时间戳 (String 类型)	当前时间的毫秒数（网关会过滤掉 5 分钟之前的请求）
path	/api/service/abc	就是你需要访问的接口路径 (根据你访问网关接口自己变更)
version	1.0.0	目前定位 1.0.0 写死，String 类型

对上述 2 个字段进行 key 的自然排序，然后进行字段与字段值拼接最后再拼接上 SK，代码示例。

第一步：首先构造一个 Map。

```
Map<String, String> map = Maps.newHashMapWithExpectedSize(2);
//timestamp 为毫秒数的字符串形式 String.valueOf(LocalDate.now().
toInstant(ZoneOffset.of("+8")).toEpochMilli())
map.put("timestamp","1571711067186"); //值应该为毫秒数的字符串形式
map.put("path", "/api/service/abc");
map.put("version", "1.0.0");
```

第二步：进行 Key 的自然排序，然后 Key, Value 值拼接最后再拼接分配给你的 Sk。

```
List<String> storedKeys = Arrays.stream(map.keySet()
    .toArray(new String[]{}))
    .sorted(Comparator.naturalOrder())
    .collect(Collectors.toList());
final String sign = storedKeys.stream()
    .map(key -> String.join("", key, params.get(key)))
    .collect(Collectors.joining()).trim()
    .concat("506EEB535CF740D7A755CB4B9F4A1536");
```

- 你得到的 sign 值应该为: path/api/service/abctimestamp1571711067186version1.0.0506EEB535CF740D7A755CB4B9F4A1536

第三步: 进行 Md5 加密后转成大写。

```
DigestUtils.md5DigestAsHex(sign.getBytes()).toUpperCase()
```

- 最后得到的值为: A021BF82BE342668B78CD9ADE593D683

10.11.7 请求网关

- 假如你访问的路径为: /api/service/abc。
- 访问地址: http: 网关的域名/api/service/abc。
- 设置 header 头, header 头参数为:

字段	值	描述
timestamp	1571711067186	上述你进行签名的时候使用的时间值
appKey	1TEST123456781	分配给你的 Ak 值
sign	A90E66763793BDBC817CF3B52AAAC041	上述得到的签名值
version	1.0.0	写死, 就为这个值

- 签名插件会默认过滤 5 分钟之后的请求

10.11.8 如果认证不通过会返回 code 为 401 message 可能会有变动。

```
"code":401,"message":"sign is not pass,Please check you sign algorithm!","data":null}
```

10.11.9 签名认证算法扩展

- 请参考开发者文档中的 [扩展签名算法](#)。

10.12 Rewrite 插件

10.12.1 说明

- soul 网关在对目标服务进行代理调用的时候, 还容许用户使用 rewrite 插件来重写请求路径

10.12.2 插件设置

- 在 soul-admin-> 插件管理-> rewrite, 设置为开启。
- 在网关的 pom.xml 文件中添加 rewrite 的支持。
- 如果用户不需要, 可以把插件禁用。

```
<!-- soul rewrite plugin start-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-rewrite</artifactId>
  <version>${last.version}</version>
</dependency>
<!-- soul rewrite plugin end-->
```

- 选择器和规则, 请详细看: [选择器规则](#)。
 - 只有匹配的请求, 才会进行重写。

10.12.3 场景

- 顾名思义, 重写插件就是对 uri 的重新定义。
- 当匹配到请求后, 设置自定义的路径, 那么自定义的路径就会覆盖之前的真实路径。
- 在调用的时候, 就会使用用户自定义的路径。

10.13 Websocket 支持

10.13.1 说明

- soul 网关是支持 websocket 的代理。
- websocket 支持中, 使用了 divide 插件。

10.13.2 插件设置

- 在 soul-admin-> 插件管理-> divide, 设置为开启。
- 在网关的 pom.xml 文件中新增依赖

```
<!--if you use http proxy start this-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-divide</artifactId>
  <version>${last.version}</version>
</dependency>
```

```
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-httpclient</artifactId>
  <version>${last.version}</version>
</dependency>
```

10.13.3 请求路径

- 使用 soul 代理 websocket 的时候，其请求路径为（例子）：`ws://localhost:9195/?module=ws&method=/websocket&rpcType=websocket`。

参数详解：

- localhost:8080 是 soul 启动的 ip 和端口。
- module（必填）：值是你用来匹配 selector 的关键
- method（参数）：你的 websocket 路径，同时也用做匹配 rule
- rpcType：websocket 必填，且必须为 websocket

- 在 divide 插件中选择器新增一条配置，如下

选择器

* 名称: websocket路由

* 类型: 自定义流量

* 匹配方式: and 这里填写module字段 和你的请求匹配

* 条件: 这里选择query module = ws 删除 新增

* 继续后续选择器: ☒ * 打印日志: ☒ * 是否开启: ☒

这里填写你的websocket服务的地址

* http配置: localhost http:// 127.0.0.1:8080 50 删除 新增

* 执行顺序: 1

取消 确定

- 在这一条选择器下新增一条规则：

规则

* 名称: websocket路由

* 匹配方式: and

这里要选择query

* 条件: query method = websocket 删除 新增

这里填写method 与你的url的保持一致, 值也一样

Hystrix处理:

跳闸最小请求数量 20 错误半比阈值 50

最大并发量 100 跳闸休眠时间(ms) 5000

分组Key groupKey 命令Key commandKey 超时时间(ms) 5000

http负载: 负载策略 random 重试次数 0

* 打印日志: ☒ * 是否开启: ☒

* 执行顺序: 1

取消 确定

- 总结, 这个时候注意看你的路径 `ws://localhost:9195/?module=ws&method=/websocket&rpcType=websocket`。

它就会被你新增的选择器规则匹配, 然后代理的真实 websocket 地址为: `127.0.0.1:8080/websocket`, 这样 soul 就进行的 websocket 的代理。

你就可以进行和 websocket 服务进行通信了, 就是这么简单。

- 最后再说一句, module, method 命名和值, 你完全可以自己来决定, 我的只是列子, 只要选择器和规则能够匹配就行。

10.14 Context Path 插件

10.14.1 说明

- soul 网关在对目标服务调用的时候, 还容许用户使用 `context_path` 插件来重写请求路径的 `contextPath`

10.14.2 插件设置

- 在 soul-admin-> 插件管理-> context_path 设置为开启。
- 在网关的 pom.xml 文件中添加 context_path 的支持。
- 如果用户不需要，可以把插件禁用。

```
<!-- soul context_path plugin start-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-context-path</artifactId>
  <version>${last.version}</version>
</dependency>
<!-- soul context_path plugin end-->
```

- 选择器和规则，请详细看：[选择器规则](#)。
- 只有匹配的请求，并且配置规则才会进行重写 contextPath。

10.14.3 场景

- 顾名思义，context_path 插件就是对 uri 的 contextPath 重新定义。
- 当匹配到请求后，设置自定义的 contextPath，那么就会根据请求的 Url 截取自定义的 contextPath 获取真正的 Url，例如请求路径为/soul/http/order，配置的 contextPath 为/soul/http，那么真正请求的 url 为/order。

10.15 重定向插件

10.15.1 说明

soul 网关在对目标服务进行代理调用的时候，还容许用户使用 redirect 插件来重定向请求。

10.15.2 插件设置

- 在 soul-admin-> 插件管理-> redirect，设置为开启。
- 在网关的 pom.xml 文件中添加 redirect 的支持。
- 如果用户不需要，可以把插件禁用。
- 选择器和规则，只有匹配的请求，才会进行转发和重定向，请详细看：[选择器规则](#)。

10.15.3 Maven 依赖

在 soul-bootstrap 工程的 pom.xml 文件中添加插件依赖。

```
<!-- soul redirect plugin start-->
<dependency>
  <groupId>org.dromara</groupId>
  <artifactId>soul-spring-boot-starter-plugin-redirect</artifactId>
  <version>${last.version}</version>
</dependency>
<!-- soul redirect plugin end-->
```

10.15.4 场景

顾名思义，redirect 插件就是对 uri 的重新转发和重定向。

重定向

- 我们在 Rule 配置自定义路径时，应该为一个可达的服务路径。
- 当匹配到请求后，根据自定义的路径，Soul 网关会进行 308 服务跳转。

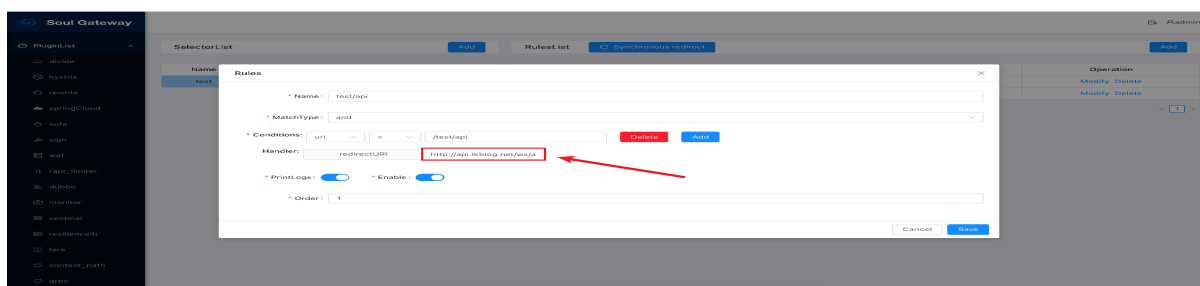


图 1: 重定向配置

网关自身接口转发

- 当满足匹配规则时，服务内部会使用 DispatcherHandler 内部接口转发。
- 要实现网关自身接口转发，我们需要在配置路径使用 / 作为前缀开始，具体配置如下图。

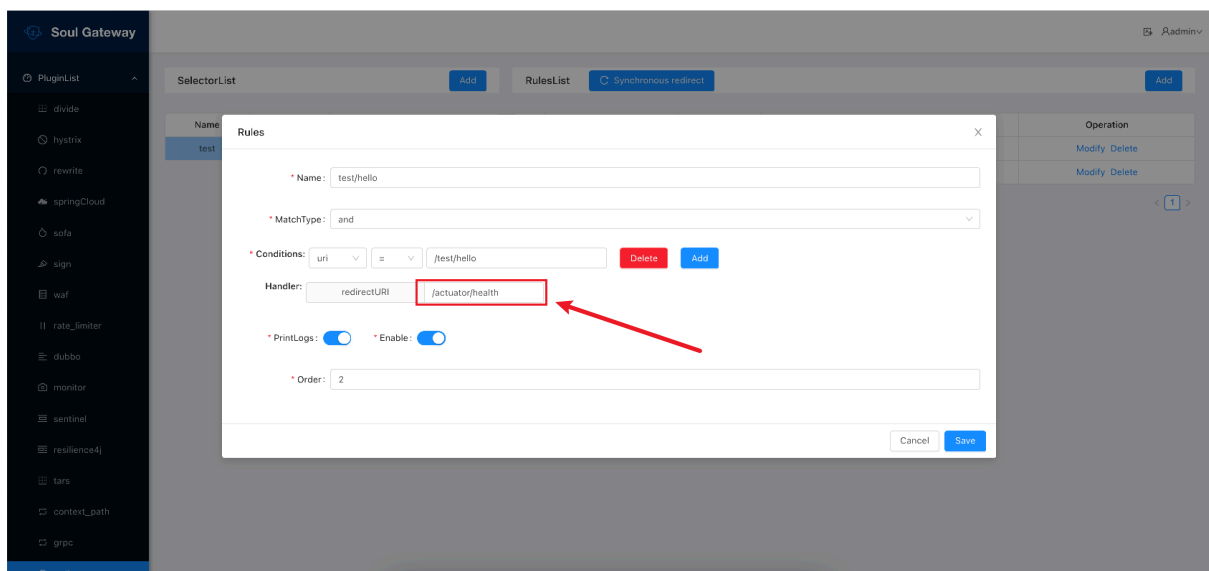


图 2: 自身接口转发

11.1 自定义 Filter

11.1.1 说明

- 本文是说明如何进行 `org.springframework.web.server.WebFliter` 的扩展。

11.1.2 跨域支持

- 新增 `org.dromara.soul.bootstrap.cors.CrossFilter` 实现 `WebFilter`。

```
public class CrossFilter implements WebFilter {

    private static final String ALLOWED_HEADERS = "x-requested-with, authorization, Content-Type, Authorization, credential, X-XSRF-TOKEN,token,username,client";

    private static final String ALLOWED_METHODS = "*";

    private static final String ALLOWED_ORIGIN = "*";

    private static final String ALLOWED_EXPOSE = "*";

    private static final String MAX_AGE = "18000";

    @Override
    @SuppressWarnings("all")
    public Mono<Void> filter(final ServerWebExchange exchange, final WebFilterChain chain) {
        ServerHttpRequest request = exchange.getRequest();
        if (CorsUtils.isCorsRequest(request)) {
            ServerHttpResponse response = exchange.getResponse();
            HttpHeaders headers = response.getHeaders();
            headers.add("Access-Control-Allow-Origin", ALLOWED_ORIGIN);
        }
    }
}
```

```

        headers.add("Access-Control-Allow-Methods", ALLOWED_METHODS);
        headers.add("Access-Control-Max-Age", MAX_AGE);
        headers.add("Access-Control-Allow-Headers", ALLOWED_HEADERS);
        headers.add("Access-Control-Expose-Headers", ALLOWED_EXPOSE);
        headers.add("Access-Control-Allow-Credentials", "true");
        if (request.getMethod() == HttpMethod.OPTIONS) {
            response.setStatus(HttpStatus.OK);
            return Mono.empty();
        }
    }
    return chain.filter(exchange);
}
}

```

- 将 CrossFilter 注册成为 spring 的 bean，完事。

11.1.3 网关过滤 springboot 健康检查

- 注意顺序，使用 @Order 注解

```

@Component
@Order(-99)
public final class HealthFilter implements WebFilter {

    private static final String[] FILTER_TAG = {"/actuator/health", "/health_check"};

    @Override
    public Mono<Void> filter(@Nullable final ServerWebExchange exchange, @Nullable
final WebFilterChain chain) {
        ServerHttpRequest request = Objects.requireNonNull(exchange).getRequest();
        String urlPath = request.getURI().getPath();
        for (String check : FILTER_TAG) {
            if (check.equals(urlPath)) {
                String result = JsonUtils.toJson(new Health.Builder().up().
build());
                DataBuffer dataBuffer = exchange.getResponse().bufferFactory().
wrap(result.getBytes());
                return exchange.getResponse().writeWith(Mono.just(dataBuffer));
            }
        }
        return Objects.requireNonNull(chain).filter(exchange);
    }
}

```

11.1.4 继承 `org.dromara.soul.web.filter.AbstractWebFilter`

- 新增一个类，继承它。
- 实现它的 2 个方法。

```
/**
 * this is Template Method ,children Implement your own filtering logic.
 *
 * @param exchange the current server exchange
 * @param chain provides a way to delegate to the next filter
 * @return {@code Mono<Boolean>} result: TRUE (is pass), and flow next filter; FALSE
 (is not pass) execute doDenyResponse(ServerWebExchange exchange)
 */
protected abstract Mono<Boolean> doFilter(ServerWebExchange exchange,
WebFilterChain chain);

/**
 * this is Template Method ,children Implement your own And response client.
 *
 * @param exchange the current server exchange.
 * @return {@code Mono<Void>} response msg.
 */
protected abstract Mono<Void> doDenyResponse(ServerWebExchange exchange);
```

- `doFilter` 方法返回 `Mono` 表示通过，反之则不通过，不通过的时候，会调用 `doDenyResponse` 输出相关信息到前端。

11.2 插件扩展

11.2.1 说明

- 插件是 soul 网关的核心执行者，每个插件在开启的情况下，都会对匹配的流量，进行自己的处理。
- 在 soul 网关里面，插件其实分为 2 类。
 - 一类是单一职责的调用链，不能对流量进行自定义的筛选。
 - 另一类，能对匹配的流量，执行自己的职责调用链。
- 用户可以参考 `soul-plugin` 模块，新增自己的插件处理，如果有好的公用插件，请把代码提交上来。

11.2.2 单一职责插件

- 引入如下依赖:

```
<dependency>
    <groupId>org.dromara</groupId>
    <artifactId>soul-plugin-api</artifactId>
    <version>${last.version}</version>
</dependency>
```

- 用户新增一个类 A, 直接实现 org.dromara.soul.plugin.api.SoulPlugin

```
public interface SoulPlugin {

    /**
     * Process the Web request and (optionally) delegate to the next
     * {@code WebFilter} through the given {@link SoulPluginChain}.
     *
     * @param exchange the current server exchange
     * @param chain    provides a way to delegate to the next filter
     * @return {@code Mono<Void>} to indicate when request processing is complete
     */
    Mono<Void> execute(ServerWebExchange exchange, SoulPluginChain chain);

    /**
     * return plugin order .
     * This attribute To determine the plugin execution order in the same type
     plugin.
     *
     * @return int order
     */
    int getOrder();

    /**
     * acquire plugin name.
     * this is plugin name define you must Provide the right name.
     * if you impl AbstractSoulPlugin this attribute not use.
     *
     * @return plugin name.
     */
    default String named() {
        return "";
    }

    /**
     * plugin is execute.
     * if return true this plugin can not execute.
     *
     * @param exchange the current server exchange
```

```

    * @return default false.
    */
    default Boolean skip(ServerWebExchange exchange) {
        return false;
    }
}

```

- 接口方法详细说明
 - execute() 方法为核心的执行方法，用户可以在里面自由的实现自己想要的功能。
 - getOrder() 指定插件的排序。
 - named() 指定插件的名称。
 - skip() 在特定的条件下，该插件是否被跳过。
- 注册成 Spring 的 bean，参考如下，或者直接在实现类上加 @Component 注解。

```

@Bean
public SoulPlugin a() {
    return new A();
}

```

11.2.3 匹配流量处理插件

- 引入如下依赖：

```

<dependency>
    <groupId>org.dromara</groupId>
    <artifactId>soul-plugin-base</artifactId>
    <version>${last.version}</version>
</dependency>

```

- 新增一个类 A，继承 org.dromara.soul.plugin.base.AbstractSoulPlugin
- 以下是参考：

```

/**
 * This is your custom plugin.
 * He is running in after before plugin, implement your own functionality.
 * extends AbstractSoulPlugin so you must user soul-admin And add related plug-in
 development.
 *
 * @author xiaoyu(Myth)
 */
public class CustomPlugin extends AbstractSoulPlugin {

    /**
     * return plugin order .

```

```

    * The same plugin he executes in the same order.
    *
    * @return int
    */
@Override
public int getOrder() {
    return 0;
}

/**
 * acquire plugin name.
 * return you custom plugin name.
 * It must be the same name as the plug-in you added in the admin background.
 *
 * @return plugin name.
 */
@Override
public String named() {
    return "soul";
}

/**
 * plugin is execute.
 * Do I need to skip.
 * if you need skip return true.
 *
 * @param exchange the current server exchange
 * @return default false.
 */
@Override
public Boolean skip(final ServerWebExchange exchange) {
    return false;
}

@Override
protected Mono<Void> doExecute(ServerWebExchange exchange, SoulPluginChain
chain, SelectorZkDTO selector, RuleZkDTO rule) {
    LOGGER.debug("..... function plugin start.....");
    /**
     * Processing after your selector matches the rule.
     * rule.getHandle() is you Customize the json string to be processed.
     * for this example.
     * Convert your custom json string pass to an entity class.
     */
    final String ruleHandle = rule.getHandle();

    final Test test = GsonUtils.getInstance().fromJson(ruleHandle, Test.class);

```

```

    /*
     * Then do your own business processing.
     * The last execution chain.execute(exchange).
     * Let it continue on the chain until the end.
     */
    System.out.println(test.toString());
    return chain.execute(exchange);
}
}

```

- 详细讲解:

- 继承该类的插件, 插件会进行选择器规则匹配, 那如何来设置呢?
- 首先在 soul-admin 后台-> 插件管理中, 新增一个插件, 注意名称与你自定义插件的 named() 方法要一致。
- 重新登陆 soul-admin 后台, 你会发现在插件列表就多了一个你刚新增的插件, 你就可以进行选择器规则匹配
- 在规则中, 有个 handler 字段, 是你自定义处理数据, 在 doExecute() 方法中, 通过 final String ruleHandle = rule.getHandle(); 获取, 然后进行你的操作。

- 注册成 Spring 的 bean, 参考如下或者直接在实现类上加 @Component 注解。

```

@Bean
public SoulPlugin a() {
    return new A();
}

```

11.2.4 订阅你的插件数据, 进行自定义的处理

- 新增一个类 A, 实现 org.dromara.soul.plugin.base.handler.PluginDataHandler

```

public interface PluginDataHandler {

    /**
     * Handler plugin.
     *
     * @param pluginData the plugin data
     */
    default void handlerPlugin(PluginData pluginData) {
    }

    /**
     * Remove plugin.
     *
     * @param pluginData the plugin data
     */
}

```

```
default void removePlugin(PluginData pluginData) {  
}  
  
/**  
 * Handler selector.  
 *  
 * @param selectorData the selector data  
 */  
default void handlerSelector(SelectorData selectorData) {  
}  
  
/**  
 * Remove selector.  
 *  
 * @param selectorData the selector data  
 */  
default void removeSelector(SelectorData selectorData) {  
}  
  
/**  
 * Handler rule.  
 *  
 * @param ruleData the rule data  
 */  
default void handlerRule(RuleData ruleData) {  
}  
  
/**  
 * Remove rule.  
 *  
 * @param ruleData the rule data  
 */  
default void removeRule(RuleData ruleData) {  
}  
  
/**  
 * Plugin named string.  
 *  
 * @return the string  
 */  
String pluginNamed();  
}
```

- 注意 `pluginNamed()` 要和你自定义的插件名称相同。
- 注册成 Spring 的 bean，参考如下或者直接在实现类上加 `@Component` 注解。


```
@Bean
public PluginDataHandler a() {
    return new A();
}
```

11.3 文件上传下载

11.3.1 说明

- 本文主要介绍 soul 的文件上传下载的支持。

11.3.2 文件上传

- 默认限制文件大小为 10M。
- 如果想修改，在启动服务的时候，使用`--file.size = 30`，为 int 类型。
- 你之前怎么上传文件，还是怎么上传。

11.3.3 文件下载

- soul 支持流的方式进行下载，你之前的接口怎么写的，还是怎么写，根本不需要变。

11.4 正确获取 Ip 与 host

11.4.1 说明

- 本文是说明，如果网关前面有一层 nginx 的时候，如何获取正确的 ip 与端口。
- 获取正确的之后，在插件以及选择器中，可以根据 ip，与 host 来进行匹配。

11.4.2 默认实现

- 在 soul 网关自带实现为: `org.dromara.soul.web.forwarded.RemoteAddressResolver`。
- 它需要你在 nginx 设置 X-Forwarded-For，以便来获取正确的 ip 与 host。

11.4.3 扩展实现

- 新增一个类 A，实现 `org.dromara.soul.plugin.api.RemoteAddressResolver`

```
public interface RemoteAddressResolver {

    /**
     * Resolve inet socket address.
     *
     * @param exchange the exchange
     * @return the inet socket address
     */
    default InetSocketAddress resolve(ServerWebExchange exchange) {
        return exchange.getRequest().getRemoteAddress();
    }

}
```

- 把你新增的实现类注册成为 spring 的 bean，如下

```
@Bean
public SignService a() {
    return new A
}
```

11.5 自定义网关返回数据格式

11.5.1 说明

- 本文是说明基于 soul 网关返回自定义的数据个数。
- 网关需要统一的返回格式，而每个公司都有自己定义的一套，所以需要对此进行扩展。

11.5.2 默认实现

- 默认的实现为 `org.dromara.soul.plugin.api.result.DefaultSoulResult`
- 返回的数据格式如下：

```
public class SoulDefaultEntity implements Serializable {

    private static final long serialVersionUID = -2792556188993845048L;

    private Integer code;

    private String message;
```

```
private Object data;

}
```

- 返回的 json 格式如下:

```
{
  "code": -100, //返回码,
  "message": " 您的参数错误, 请检查相关文档!", //提示字段
  "data": null // 具体的数据
}
```

11.5.3 扩展

- 新增一个类 A 实现 `org.dromara.soul.plugin.api.result.SoulResult`

```
public interface SoulResult<T> {

    /**
     * Success t.
     *
     * @param code    the code
     * @param message the message
     * @param object  the object
     * @return the t
     */
    T success(int code, String message, Object object);

    /**
     * Error t.
     *
     * @param code    the code
     * @param message the message
     * @param object  the object
     * @return the t
     */
    T error(int code, String message, Object object);
}
```

- 其他泛型 T 为你自定义的数据格式, 返回它就好
- 把你新增的实现类注册成为 spring 的 bean, 如下

```
@Bean
public SoulResult a() {
    return new A();
}
```

11.6 自定义 sign 插件检验算法

11.6.1 说明

- 用户可以自定义签名认证算法来实现验证。

11.6.2 扩展

- 默认的实现为 `org.dromara.soul.plugin.sign.service.DefaultSignService`。
- 新增一个类 A 实现 `org.dromara.soul.plugin.api.SignService`。

```
public interface SignService {  
  
    /**  
     * Sign verify pair.  
     *  
     * @param exchange the exchange  
     * @return the pair  
     */  
    Pair<Boolean, String> signVerify(ServerWebExchange exchange);  
}
```

- Pair 中返回 true，表示验证通过，为 false 的时候，会把 String 中的信息输出到前端。
- 把你新增的实现类注册成为 spring 的 bean，如下

```
@Bean  
public SignService a() {  
    return new A  
}
```

11.7 多语言 http 客户端

11.7.1 说明

- 本文主要讲解其他语言的 http 服务如何接入网关。
- 如何自定义开发 soul-http-client。

11.7.2 自定义开发

- 请求方式: POST
- 请求路径: `http://soul-admin/soul-client/springmvc-register soul-admin`, 表示为 admin 的 ip + port
- 请求参数: soul 网关默认的需要参数, 通过 body 里面传, json 类型。

```
{
  "appName": "xxx", //应用名称 必填
  "context": "/xxx", //请求前缀 必填
  "path": "xxx", //路径需要唯一 必填
  "pathDesc": "xxx", //路径描述
  "rpcType": "http", //rpc 类型 必填
  "host": "xxx", //服务 host 必填
  "port": xxx, //服务端口 必填
  "ruleName": "xxx", //可以同 path 一样 必填
  "enabled": "true", //是否开启
  "registerMetaData": "true" //是否需要注册元数据
}
```

11.8 线程模型

11.8.1 说明

- 本文主要介绍 soul 的线程模型, 以及各种场景的使用。

11.8.2 io 与 work 线程

- soul 内置依赖 spring-webflux 而其底层是使用的 netty, 这一块主要是使用的 netty 线程模型。

11.8.3 业务线程

- 默认使用调度线程来执行。
- 默认使用固定的线程池来执行, 其线程数为 $\text{cpu} * 2 + 1$ 。

11.8.4 切换类型

- `reactor.core.scheduler.Schedulers`。
- 可以使用 `-Dsoul.scheduler.type=fixed` 这个是默认。设置其他的值就会使用弹性线程池来执行 `Schedulers.elastic()`。
- 可以使用 `-Dsoul.work.threads = xx` 来指定线程数量，默认为 `cpu * 2 + 1`，最小为 16 个线程。

11.9 Soul 性能优化

11.9.1 说明

- 本文主要介绍如何对 soul 进行优化

11.9.2 本身消耗

- soul 本身所有的操作，都是基于 jvm 内存来匹配，本身消耗时间大概在 1-3 ms 左右。

11.9.3 底层 netty 调优

- soul 内置依赖 spring-webflux 而其底层是使用的 netty
- 我们可以自定义 netty 的相关参数来对 soul 进行优化, 以下是示例:

```
@Bean
public NettyReactiveWebServerFactory nettyReactiveWebServerFactory() {
    NettyReactiveWebServerFactory webServerFactory = new
NettyReactiveWebServerFactory();
    webServerFactory.addServerCustomizers(new EventLoopNettyCustomizer());
    return webServerFactory;
}

private static class EventLoopNettyCustomizer implements NettyServerCustomizer {

    @Override
    public HttpServer apply(final HttpServer httpServer) {
        return httpServer
            .tcpConfiguration(tcpServer -> tcpServer
                .runOn(LoopResources.create("soul-netty", 1, DEFAULT_IO_
WORKER_COUNT, true), false)
                .selectorOption(ChannelOption.SO_REUSEADDR, true)
                .selectorOption(ChannelOption.ALLOCATOR,
PooledByteBufAllocator.DEFAULT)
                .option(ChannelOption.TCP_NODELAY, true)
                .option(ChannelOption.ALLOCATOR, PooledByteBufAllocator.
DEFAULT));
    }
}
```

```
}  
}
```

- 这个类在 `soul-bootstrap` 中已经内置，在压测的时候，可以根据自己的需求来进行优化设置。
- 业务线程模型可以看[线程模型](#)

12.1 Soul Contributor

您可以报告 bug，提交一个新的功能增强建议或者直接对以上内容提交改进补丁。

12.1.1 提交 issue

- 在提交 issue 之前，请经过充分的搜索，确定该 issue 不是通过简单的检索即可以解决的问题。
- 查看issue 列表，确定该 issue 不是一个重复的问题。
- 新建一个 issue 并选择您的 issue 类型。
- 使用一个清晰并有描述性的标题来定义 issue。
- 根据模板填写必要信息。
- 在提交 issue 之后，对该 issue 分配合适的标签。如：bug, enhancement, discussion 等。
- 请对自己提交的 issue 保持关注，在讨论中进一步提供必要信息。

12.1.2 开发流程

Fork 分支到本地

- 从 soul 的 repo 上 fork 一个分支到您自己的 repo 来开始工作，并设置 upstream 为 soul 的 repo。

```
git remote add upstream https://github.com/dromara/soul.git
```

选择 issue

- 请在选择您要修改的 issue。如果是您新发现的问题或想提供 issue 中没有的功能增强，请先新建一个 issue 并设置正确的标签。
- 在选中相关的 issue 之后，请回复以表明您当前正在这个 issue 上工作。并在回复的时候为自己设置一个 deadline，添加至回复内容中。

创建分支

- 切换到 fork 的 master 分支，拉取最新代码，创建本次的分支。

```
git checkout master
git pull upstream master
git checkout -b issueNo
```

注意：PR 会按照 squash 的方式进行 merge，如果不创建新分支，本地和远程的提交记录将不能保持同步。

编码

- 请您在开发过程中遵循 soul 的 [开发规范](#)。并在准备提交 pull request 之前完成相应的检查。
- 将修改的代码 push 到 fork 库的分支上。

```
git add 修改代码
git commit -m 'commit log'
git push origin issueNo
```

提交 PR

- 发送一个 pull request 到 soul 的 master 分支。
- 接着导师做 CodeReview，然后他会与您讨论一些细节（包括设计，实现，性能等）。当导师对本次修改满意后，会将提交合并到当前开发版本的分支中。
- 最后，恭喜您已经成为了 Soul 的贡献者！

删除分支

- 在导师将 pull request 合并到 soul 的 master 分支中之后，您就可以将远程的分支（origin/issueNo）及与远程分支（origin/issueNo）关联的本地分支（issueNo）删除。

```
git checkout master
git branch -d issueNo
git push origin --delete issueNo
```

注意

为了让您的 id 显示在 contributor 列表中，别忘了以下设置：

```
git config --global user.name "username"
git config --global user.email "username@mail.com"
```

常见问题

- 每次 Pull Request(PR) 后，你需要执行以下操作，否则，之前 PR 的提交记录会和这次的提交记录混在一起，具体操作流程如下：

```
git checkout master
git fetch upstream
git reset --hard upstream/master
git push -f
```

12.2 Soul Committer

12.2.1 提交者提名

当你做了很多贡献以后，社区会进行提名。成为 committer 你会拥有

- soul 仓库写的权限
- idea 正版使用

12.2.2 提交者责任

- 开发新功能；
- 代码重构；
- 及时和可靠的评审 Pull Request；
- 思考和接纳新特性请求；
- 解答问题；
- 维护文档和代码示例；
- 改进流程和工具；
- 引导新的参与者融入社区。

12.2.3 日常工作

1. committer 需要每天查看社区待处理的 Pull Request 和 issue 列表，指定给合适的 committer，即 assignee。
2. assignee 在被分配 issue 后，需要进行如下判断：
 - 判断是否是长期 issue，如是，则标记为 pending。
 - 判断 issue 类型，如：bug, enhancement, discussion 等。
 - 判断 Milestone，并标记。

注意

无论是否是社区 issue，都必须有 assignee，直到 issue 完成。

12.3 Soul Code Conduct

12.3.1 开发理念

- 用心保持责任心和敬畏心，以工匠精神持续雕琢。
- 可读代码无歧义，通过阅读而非调试手段浮现代码意图。
- 整洁认同《重构》和《代码整洁之道》的理念，追求整洁优雅代码。
- 一致代码风格、命名以及使用方式保持完全一致。
- 精简极简代码，以最少的代码表达最正确的意思。高度复用，无重复代码和配置。及时删除无用代码。
- 抽象层次划分清晰，概念提炼合理。保持方法、类、包以及模块处于同一抽象层级。

12.3.2 代码提交行为规范

- 确保通过全部测试用例，确保执行 `./mvnw clean install` 可以编译和测试通过。
- 确保覆盖率不低于 master 分支。
- 确保使用 Checkstyle 检查代码，违反验证规则的需要有特殊理由。模板位置在 https://github.com/dromara/soul/blob/master/script/soul_checkstyle.xml，请使用 checkstyle 8.8 运行规则。
- 应尽量将设计精细化拆分；做到小幅度修改，多次提交，但应保证提交的完整性。
- 确保遵守编码规范。

12.3.3 编码规范

- 使用 linux 换行符。
- 缩进（包含空行）和上一行保持一致。
- 类声明后与下面的变量或方法之间需要空一行。
- 不应有无意义的空行。请提炼私有方法，代替方法体过长或代码段逻辑闭环而采用的空行间隔。
- 类、方法和变量的命名要做到顾名思义，避免使用缩写。
- 返回值变量使用 `result` 命名；循环中使用 `each` 命名循环变量；`map` 中使用 `entry` 代替 `each`。
- 配置文件使用 `Spinal Case` 命名（一种使用-分割单词的特殊 `Snake Case`）。
- 需要注释解释的代码尽量提成小方法，用方法名称解释。
- `equals` 和 `==` 条件表达式中，常量在左，变量在右；大于小于等条件表达式中，变量在左，常量在右。
- 除了构造器参与全局变量名称相同的赋值语句外，避免使用 `this` 修饰符。
- 除了用于继承的抽象类之外，尽量将类设计为 `final`。

- 嵌套循环尽量提成方法。
- 成员变量定义顺序以及参数传递顺序在各个类和方法中保持一致。
- 优先使用卫语句。
- 类和方法的访问权限控制为最小。
- 方法所用到的私有方法应紧跟该方法，如果有多个私有方法，书写私有方法应与私有方法在原方法的出现顺序相同。
- 方法入参和返回值不允许为 `null`。
- 优先使用三目运算符代替 `if else` 的返回和赋值语句。
- 优先考虑使用 `LinkedList`，只有在需要通过下标获取集合中元素值时再使用 `ArrayList`。
- `ArrayList`，`HashMap` 等可能产生扩容的集合类型必须指定集合初始大小，避免扩容。
- 日志与注释一律使用英文。
- 注释只能包含 `javadoc`，`todo` 和 `fixme`。
- 公开的类和方法必须有 `javadoc`，其他类和方法以及覆盖自父类的方法无需 `javadoc`。

12.3.4 单元测试规范

- 测试代码和生产代码需遵守相同代码规范。
- 单元测试需遵循 AIR（Automatic, Independent, Repeatable）设计理念。
 - 自动化（Automatic）：单元测试应全自动执行，而非交互式。禁止人工检查输出结果，不允许使用 `System.out`，`log` 等，必须使用断言进行验证。
 - 独立性（Independent）：禁止单元测试用例间的互相调用，禁止依赖执行的先后次序。每个单元测试均可独立运行。
 - 可重复执行（Repeatable）：单元测试不能受到外界环境的影响，可以重复执行。
- 单元测试需遵循 BCDE（Border, Correct, Design, Error）设计原则。
 - 边界值测试（Border）：通过循环边界、特殊数值、数据顺序等边界的输入，得到预期结果。
 - 正确性测试（Correct）：通过正确的输入，得到预期结果。
 - 合理性设计（Design）：与生产代码设计相结合，设计高质量的单元测试。
 - 容错性测试（Error）：通过非法数据、异常流程等错误的输入，得到预期结果。
- 如无特殊理由，测试需全覆盖。
- 每个测试用例需精确断言。
- 准备环境的代码和测试代码分离。
- 只有 `junit Assert`，`hamcrest CoreMatchers`，`Mockito` 相关可以使用 `static import`。
- 单数据断言，应使用 `assertTrue`，`assertFalse`，`assertNull` 和 `assertNotNull`。
- 多数据断言，应使用 `assertThat`。

- 精确断言，尽量不使用 `not`, `containsString` 断言。
- 测试用例的真实值应命名为 `actualXXX`，期望值应命名为 `expectedXXX`。
- 测试类和 `@Test` 标注的方法无需 `javadoc`。

13.1 PDF

- [English](#)
- [中文](#)