项目申请书

项目名称: 为规则引擎LiteFlow提供基于Redis规则存储适配

项目主导师: 铂赛东 bryan31@dromara.org

申请人: 陈贤文

日期: 2023年5月9日

邮箱: mcxw@hust.edu.cn

项目申请书

1.项目背景

2.技术方法及可行性

3.项目实现细节梳理

3.1 Redis数据格式对象设计

需求分析

监听规则的逻辑

设计思路

3.2 Mockito技术进行Mock测试

单元测试

Mock Redis客户端对象

测试规则解析器

测试规则监听逻辑

测试规则执行

压力测试

4.规划

1.项目研发第一阶段

2.项目研发第二阶段

3.期望

1.项目背景

LiteFlow作为一款规则引擎框架，目前提供了多种规则持久化的适配，如多种数据库，多种注册中心等等，而此次的项目是为LiteFlow增加基于Redis的规则存储适配器，即利用Redis来存储规则，主要有以下几点内容

1.读取redis的规则

2.组装规则和解析规则

3.建立对redis规则变化的监听，触发刷新规则的逻辑。

难点1

对redis数据格式对象的设计

难点2

如何利用redis自身的特性通过流行的redis客户端完成key变化的监听，并获取内容

技术要求

需要新增一个插件模块，用于redis的规则存储设计用于redis规则存储的配置项解析和拼装规则解析器设计监听规则的逻辑需要新增一个单元测试工程，用于各种情况的单元测试使用mockito技术进行mock测试设计和进行压力测试

2.技术方法及可行性

redis

曾学习过redis基础用法，对redis规则较为熟悉，后续可以加强存储设计方向学习

mockito

理解 Junit 框架中的单元测试。理解 Junit 框架中的单元测试。后续会多加mockito技术的学习

3.项目实现细节梳理

3.1 Redis数据格式对象设计

需求分析

为了实现Redis规则存储插件，我们需要设计一个Redis数据格式对象，以便将规则存储在Redis服务器中。该对象应满足以下需求：

1. 可以将规则保存为JSON格式，并将其存储在Redis服务器中。
2. 可以从Redis服务器中检索规则并将其解析为可执行的代码。
3. 可以订阅Redis通道以监听规则变更通知。

主要代码区间

该对象包含以下主要代码区间：

- 配置项设计

为了启用Redis规则存储插件，我们需要在系统配置文件中添加以下配置项：

```
redis.host=127.0.0.1 # redis host地址
redis.port=6379 # redis 端口号
redis.password=123456 # redis 密码，如果不需要密码可不配置
redis.rule.prefix=test # 规则前缀
```

- 规则解析器

为了解析和拼装规则，我们需要开发一个规则解析器。解析器将执行以下操作：

1. 从Redis服务器中获取规则的JSON表示形式。
2. 解析JSON格式的规则，根据规则类型进行处理。
3. 将规则编译成可执行的代码。
4. 执行规则并返回结果。

监听规则的逻辑

我们需要设计监听规则的逻辑，以便在规则变化时自动更新规则。这可以通过以下步骤来完成：

1. 在系统启动时，连接到Redis服务器。
2. 订阅规则变更通知频道。
3. 当接收到通知时，重新加载规则。

设计思路

基于以上需求和主要代码区间，我们可以设计一个Redis数据格式对象，它包含以下方法：

`saveRule(rule: Rule): void`

将给定的规则存储在Redis服务器中，并使用配置文件中定义的规则前缀作为键名。

`getRule(ruleName: string): Rule`

从Redis服务器中检索具有给定名称的规则，并返回一个Rule对象。

`subscribe(channel: string, callback: (channel: string, message: string) ⇒ void): void`

订阅给定名称的Redis通道，并在接收到消息时调用给定的回调函数。

`unsubscribe(channel: string): void`

取消订阅给定名称的Redis通道。

`publish(channel: string, message: string): void`

向给定名称的Redis通道发布消息。

`parseRule(jsonRule: string): Rule`

将给定的JSON格式的规则解析为Rule对象。

`compileRule(rule: Rule): void`

将给定的规则编译成可执行的代码。

`executeRule(rule: Rule, data: any): any`

执行给定的规则，并针对给定的输入数据返回结果。

设计一个Redis数据格式对象需要考虑到各种需求和主要代码区间。该对象应提供一系列方法，以便在插件中存储、检索、解析和执行规则。它还应提供订阅/取消订阅Redis通道的功能，以便在规则变化时及时更新规则。

3.2 Mockito技术进行Mock测试

单元测试

在进行单元测试时，我们将使用Mockito技术进行mock测试。Mock测试是一种模拟对象的测试方法，它可以在不依赖外部系统（例如Redis服务器）的情况下进行测试。以下是我们将使用的测试工具：

- JUnit：用于编写单元测试用例。
- Mockito：用于创建模拟对象和设置期望行为。

Mock Redis客户端对象

为了模拟Redis客户端对象，我们将使用Mockito创建一个模拟对象。这样，我们就可以在不实际连接到Redis服务器的情况下测试插件代码。以下是我们将使用的代码：

```
// 创建模拟Redis客户端对象
Jedis redisClient = Mockito.mock(Jedis.class);

// 设置模拟对象的期望行为
when(redisClient.get("test:rule1")).thenReturn("{\"name\": \"rule1\", \"type\":
\"javascript\", \"code\": \"function(ruleData) { return ruleData + 'bar'; }\"}");
```

测试规则解析器

我们将编写单元测试来测试规则解析器是否正确解析规则。以下是我们将使用的代码：

```
@Test
public void testParseRule() {
    // 创建模拟规则JSON字符串
    String json = "{\"name\": \"rule1\", \"type\": \"javascript\", \"code\":
\"function(ruleData) { return ruleData + 'bar'; }\"}";

    // 解析规则
    Rule rule = redisDataFormat.parseRule(json);

    // 验证规则的名称
    assertEquals("rule1", rule.getName());

    // 验证规则的类型
    assertEquals(RuleType.JAVASCRIPT, rule.getType());

    // 验证规则的代码
    assertEquals("function(ruleData) { return ruleData + 'bar'; }", rule.getCode());
}
```

测试规则监听逻辑

我们将编写单元测试来测试规则监听逻辑是否能够正确地更新规则。以下是我们将使用的代码：

```
@Test
public void testSubscribe() {
    // 创建模拟Redis客户端对象
    Jedis redisClient = Mockito.mock(Jedis.class);

    // 创建模拟回调函数
    Consumer<String> callback = Mockito.mock(Consumer.class);

    // 创建Redis数据格式对象
    RedisDataFormat redisDataFormat = new RedisDataFormat(redisClient, "test");

    // 订阅规则变更通知
    redisDataFormat.subscribe("test:rules", callback);

    // 发布规则变更通知
    redisDataFormat.publish("test:rules", "rule1");
```

```
    // 验证回调函数是否被调用
    Mockito.verify(callback).accept("rule1");
}
```

测试规则执行

我们将编写单元测试来测试规则执行是否按预期执行。以下是我们将使用的代码：

```
@Test
public void testExecuteRule() {
    // 创建模拟规则
    Rule rule = new Rule("rule1", RuleType.JAVASCRIPT, "function(ruleData) { return
ruleData + 'bar'; }");

    // 执行规则
    Object result = redisDataFormat.executeRule(rule, "foo");

    // 验证结果是否正确
    assertEquals("foobar", result);
}
```

压力测试

为了进行压力测试，我们将使用以下工具：

- JMeter：用于模拟高负载条件下的插件行为。
- Redis Data Format插件：用于在JMeter中集成并测试插件。

测试方案

我们将模拟100个并发用户对插件进行10分钟的访问。每个用户将发送100个请求，并等待0.5秒钟的响应时间。以下是我们将使用的测试方案：

1. 创建一个线程组，其中包含100个并发用户。每个用户将发送100个请求。
2. 将每个请求设置为执行函数规则操作。
3. 在执行前，使用Redis Data Format插件加载所有规则。
4. 在测试计划中添加一个定时器，以便每个用户等待

4.规划

1.项目研发第一阶段

（07 月 01 日 - 08 月 15 日）：

了解LiteFlow的基本使用，对基于Redis的规则存储适配器，即利用Redis来存储规则进行熟悉

2.项目研发第二阶段

（08 月 16 日 - 09 月 30 日）：

对redis数据格式对象的设计，了解其需求和主要代码区间。

- 设计用于redis规则存储的配置项
- 解析和拼装规则解析器

- 设计监听规则的逻辑

单元测试工程

为了确保插件的正确性，我们需要创建一个单元测试工程，用于对各种情况进行测试。测试工程应包括以下测试：

1. 测试规则解析器是否正确解析规则。
2. 测试规则监听逻辑是否能够正确地更新规则。
3. 测试规则执行是否按预期执行。

Mockito技术进行Mock测试

我们将使用Mockito技术进行Mock测试。Mock测试是一种模拟对象的测试方法，它可以

在不依赖外部系统（例如Redis服务器）的情况下进行测试。Mock测试将执行以下步骤：

1. 使用Mockito创建模拟Redis客户端对象。
2. 在测试中，使用模拟对象代替实际的Redis客户端对象。
3. 运行测试以验证插件是否按预期运行。

压力测试设计

为了确保插件的高性能和可伸缩性，我们需要设计和执行压力测试。压力测试应模拟高负载条件下的插件行为，并记录以下数据：

1. 每秒处理请求数。
2. 平均响应时间。
3. CPU和内存使用率。
4. Redis服务器的吞吐量。

3.期望

我将采取以下技术实现细节：

- 设计一个Redis数据格式对象，用于将规则存储在Redis服务器中。
- 实现配置项设计，以便启用Redis规则存储插件。
- 解析和拼装规则解析器，以支持多种规则类型。
- 设计监听规则的逻辑，以便在规则变化时自动更新规则。
- 编写单元测试工程，在各种情况下测试插件代码的正确性。
- 使用Mockito技术进行mock测试，以确保插件的正确性和稳定性。
- 设计和执行压力测试，以验证插件的高性能和可伸缩性。
- 利用Redis发布/订阅功能和键空间通知机制对key变化进行监听，并获取内容。

以上技术实现细节将帮助我们实现一个高效、可靠和易于维护的Redis规则存储插件模块, 为规则引擎LiteFlow提供基于Redis规则存储适配。