2018

gRpc微服务框架及分布式事务

Shiming.liu@dianrong.com

目录 contents

01 saluki概述

saluki是以gRpc做服务底层的服务化框架，主旨在提高gRpc的易用性、扩展性的微服务框架

02 saluki特性

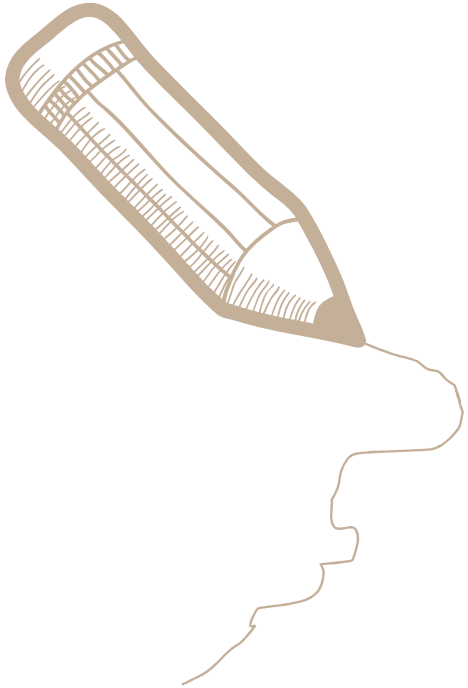
服务注册、发现、软负载均衡、路由的服务治理功能，降低开发复杂度

03 dts概述

dts是以两阶段提交，但是又不同于两阶段的分布式事务组件，目的在微服务时代下解决分布式事务一个利器

04 dts特性

基于两阶段，却没有两阶段低劣的性能，易于使用



PART 01

Saluki概述

Saluki summary

saluki是以gRpc做服务底层的服务化框架，主旨在提高gRpc的易用性、扩展性的微服务框架

saluki研发背景

多种语言问题： 全程网络是一家创业公司，初始以php快速开发业务，而后陆续又有java的开发人员加入，而当初我加入的时候，存在php、java、nodejs、c++等开发语言，如何用最少的代价把服务化推上线是主要目标

开发人员问题： 如何指定强约定下的开发约束？让开发人员的代码更为友好及可维护性，并且提高开发效率，测试效率让开发人员更为便捷的开发业务

运维人员问题： 没有运维人员帮忙部署系统，开发就是运维，如何让运维工作不会降低开发人员的开发效率？容器化及容器化编排是我们的另一个目标

saluki特性

跨多种语言

底层使用gRpc，而gRpc的跨语言能力是我们所需要的

接口契约

改变gRpc的以stub方式作为客户端和服务端的契约不同，我们期望以接口作为服务契约

Req、Resp标准化

改变gRpc的入参及出参，使用标准的Java Bean作为请求参数及返回参数

服务治理能力

具有服务治理能力

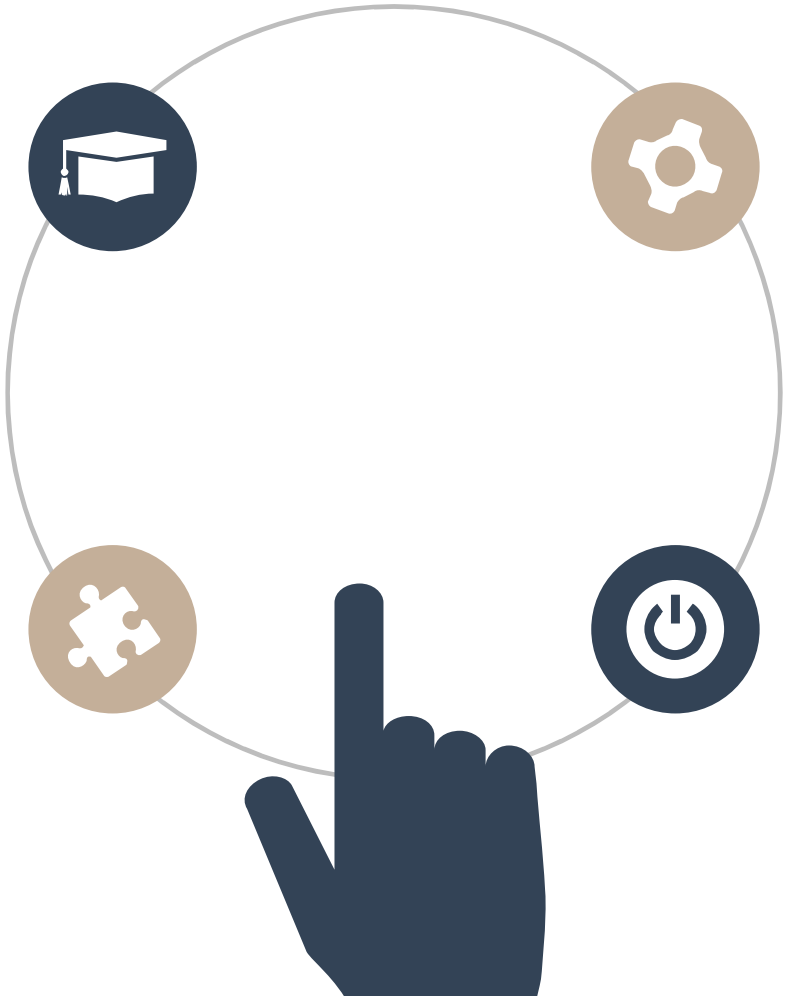
saluki技术特性

接口契约及Req、Resp标准化

开发自己的Gradle Maven插件，与gRpc的插件处于同等地位，插件生成标准的Java interface及标准的Java Bean作为客户端及服务端的契约方式

服务治理能力

基于Consul提供服务注册、服务发现、服务路由、服务熔断、服务隔离、软负载均衡、服务文档化测试等服务治理能力

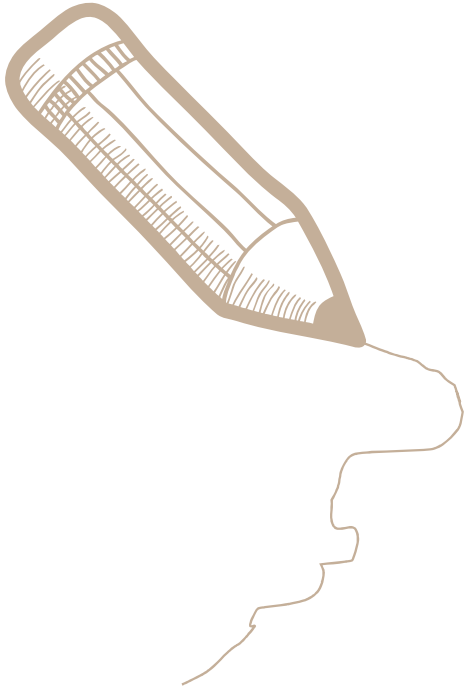


Docker容器化

基于Spring Boot提供saluki的starter插件，提供了FatJar能力，为容器化打好基础

GateWay

提供统一入口网关，兼容http1.0的协议，在网关层统一做协议处理



PART 02

详细内容

Detail Contents

KOPPT一个做PPT的神器KOPPT一个做PPT的神器KOPPT一个做PPT的神器

Saluki-plugin目标

标准java Interface

标准的Request、Response

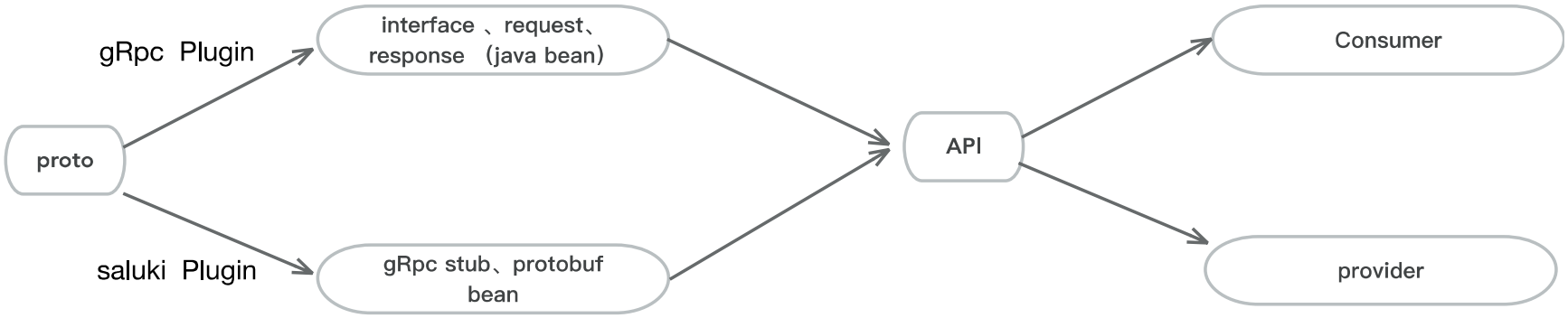
Java Bean**与**Protobuf Bean

相互映射

基于annotation来借助反射的功能来实现互相拷贝，借助saluki-serializer来具体实现

Maven、Gradle

业务开发流程



01 定义Proto

02 使用Saluki-Plugin,gRpc-Plugin

03 生成API（自动）

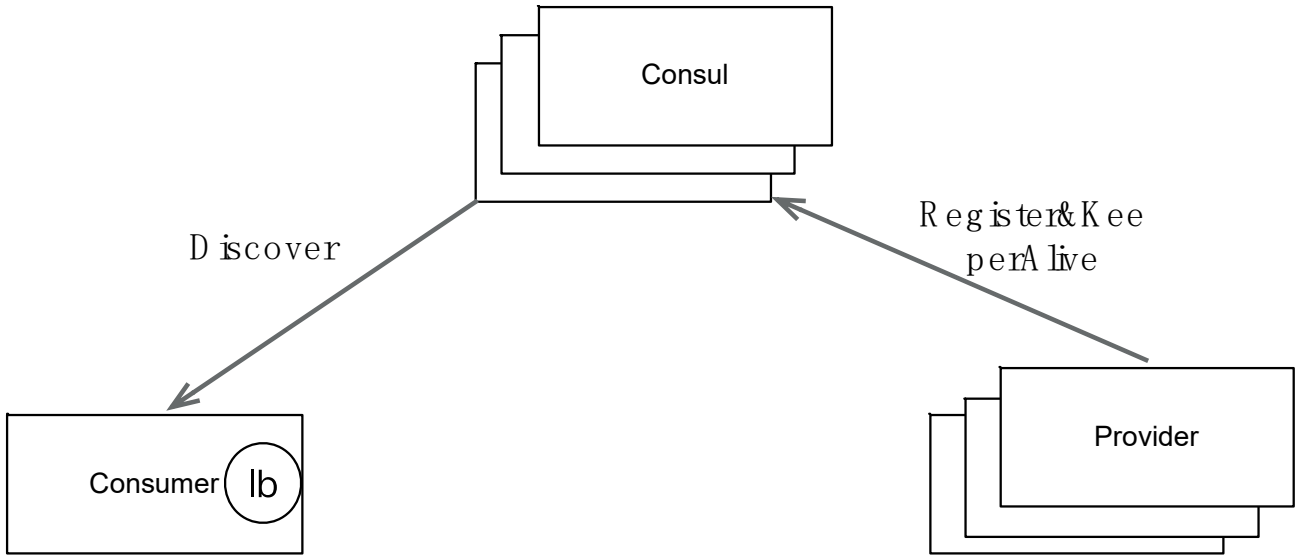
04 客户端服务端相应开发

Saluki目标

服务注册及发现

服务治理

熔断、隔离、重试、路由、文档



注册、发现、路由（gRpc扩展点）

gRpc ServerServiceDefinition内涵注册功能，按照服务名、方法名完成gRpc内部注册后在注册到Consul的注册中心

gRpc NameResolver提供了服务发现的扩展点

grpcLoadBalancer提供了负载均衡的扩展点， SubchannelPicker提供了动态路由的扩展点

隔离、熔断、降级

Hystrix

- 以服务名作为Group组名，以服务名+方法名作为CommandKey

- 开启Hystrix的熔断服务、隔离服务、并按照合适的标志来开启降级服务

路由规则

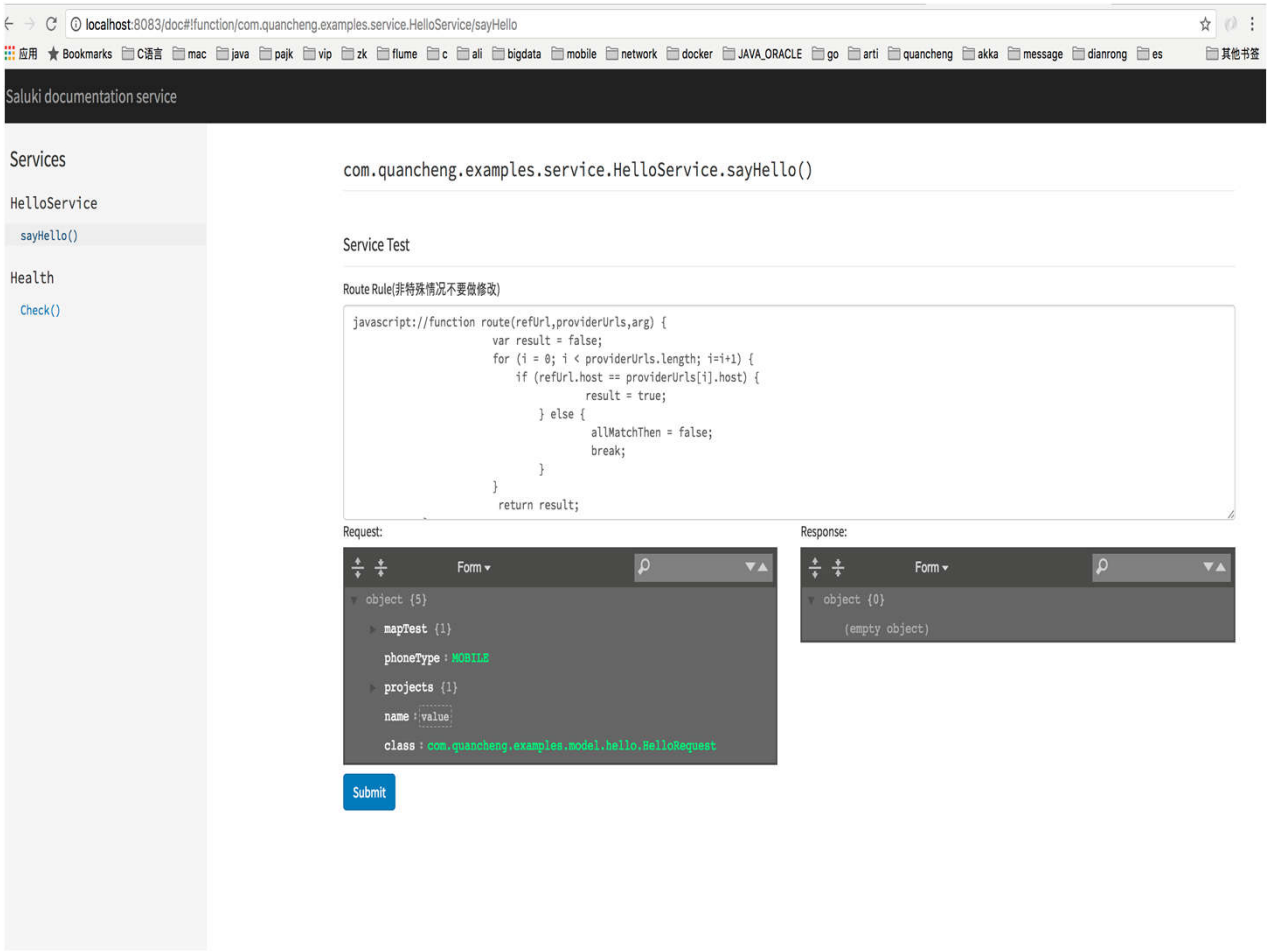
存放在注册中心

- 条件路由
- 脚本路由

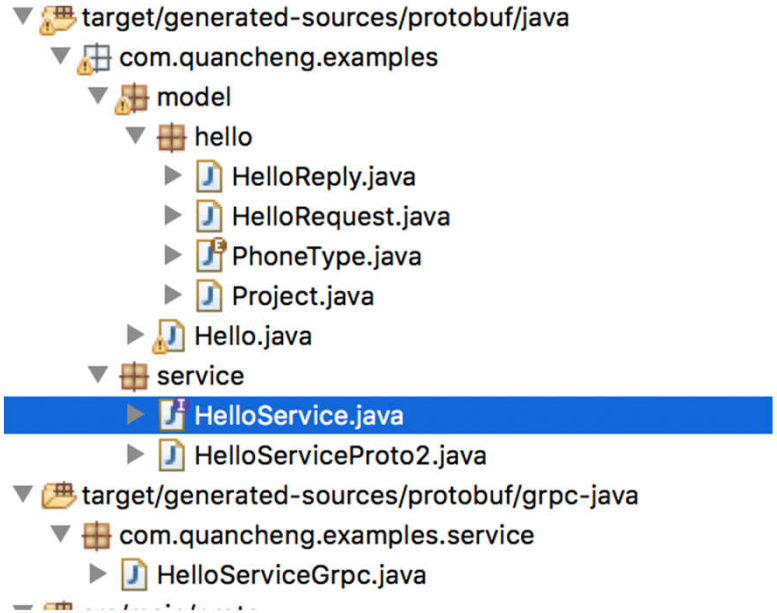
文档及测试

展现注册的服务及相应的方法及入参出参

直接输入参数可以直接在web端测试



实际开发流程



01 **定义Proto协议**

```
syntax = "proto3";
option java_package = "com.quancheng.examples.service";
option java_outer_classname = "HelloServiceProto2";
package com.quancheng.examples.service;
import "example/hello.proto";
service HelloService {
   rpc sayHello
(com.quancheng.examples.model.HelloRequest) returns
(com.quancheng.examples.model.HelloReply) {}
```

实际开发流程

```
@SalukiService
public class HelloServiceImpl implements HelloService {

  @Override
  public HelloReply sayHello(HelloRequest request) {
    HelloReply reply = new HelloReply();
    reply.setMessage(request.getName());
    return reply;
  }
}
```

02 服务端实现

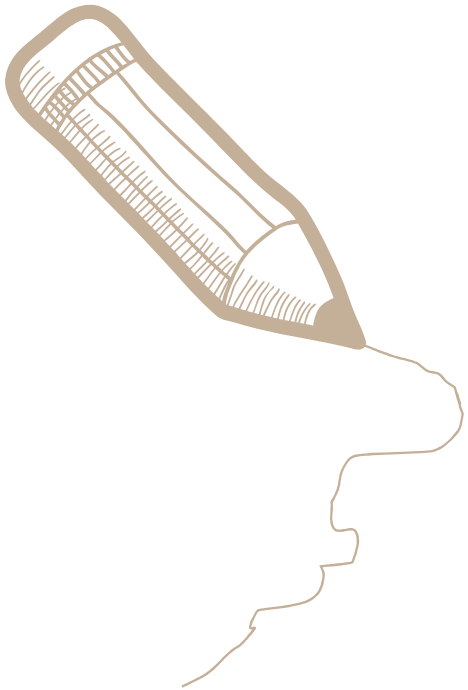
实际开发流程

```
@RestController
@RequestMapping("/proxy")
public class ProxyServiceController {

  @SalukiReferenc
  private HelloService helloService;

  @RequestMapping("/hello")
  public HelloReply hello() {
    return call(name);
  }
  private HelloReply call(final String name) {
    HelloRequest request = new HelloRequest();
    request.setName(name);
    HelloReply reply = helloService.sayHello(request);
    return reply;
  }
}
```

02 客户端引用



PART 03

Dts分布式事务

Dts概述

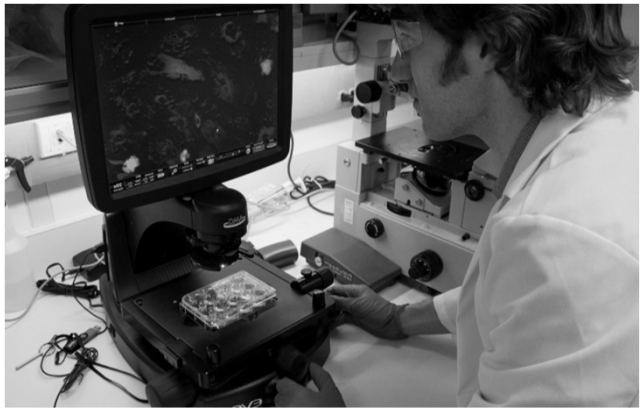
分布式事务

dts是一个分布式事务处理组件，易用性是Dts的主要目标，不修改源代码能够将分布式事务处理起来

支持saluki及Spring cloud

原则上支持所有的服务化框架，易扩展

基于两阶段提交



Dts主要优点

- 可以把业务快速加入分布式事务
- 已有服务无需修改代码就可以支持事务
- 不改动用户数据库表结构
- 可以与其他中间件快速集成
- 很好的满足原子性、一致性、隔离性（Read Uncommited）、持久性

Dts的整体架构

01

Dts Server

事务协调者

- ✓ 分配事务Id
- ✓ 处理分支注册
- ✓ 处理事务提交/回滚

02

Dts Client

事务发起者

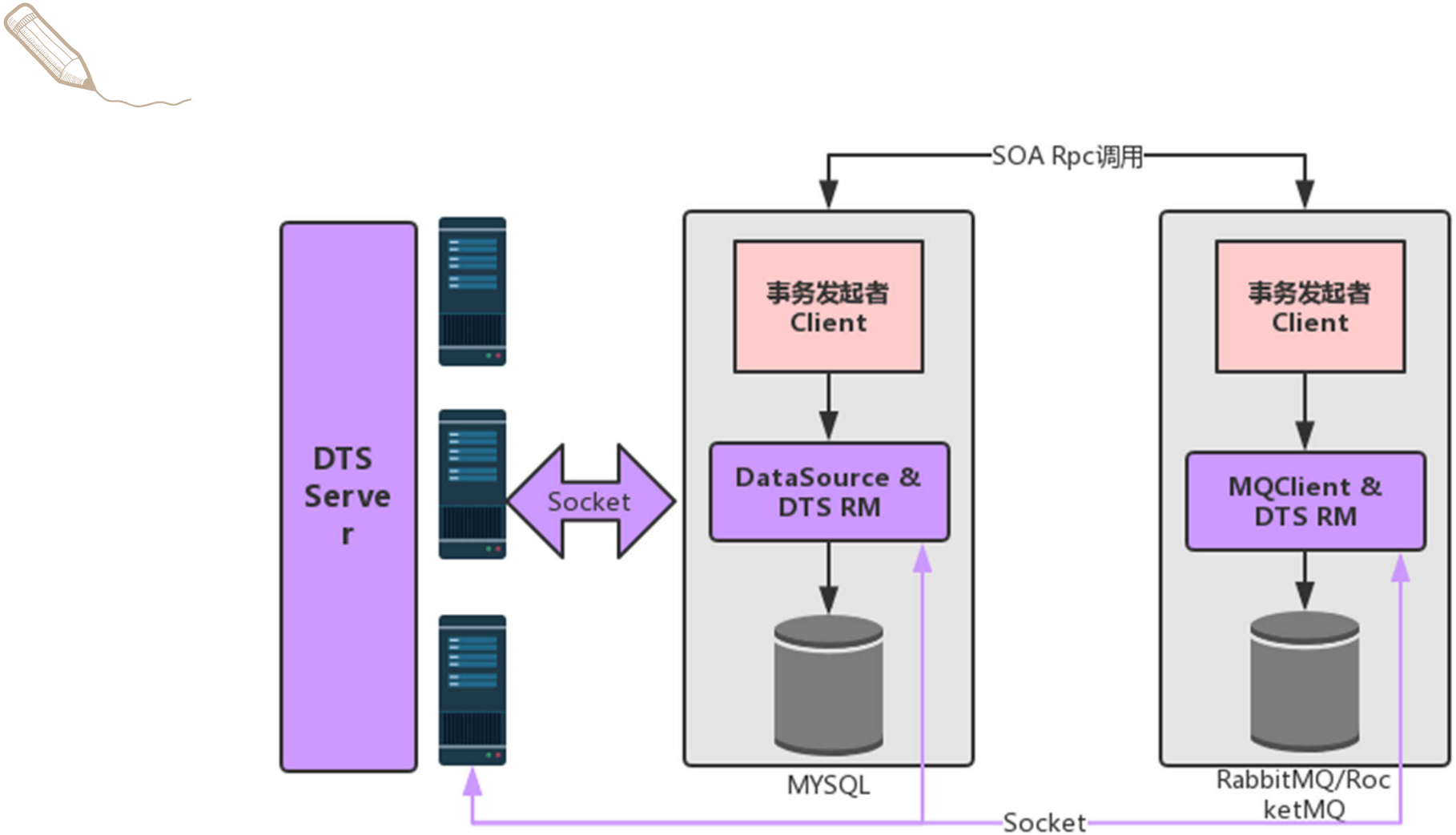
- ✓ 发起事务，界定事务边界
- ✓ 提交必须由Client发起
- ✓ 回滚必须由Client发起
- ✓ 从Zookeeper中获取Dts Server 列表

03

Dts Resource

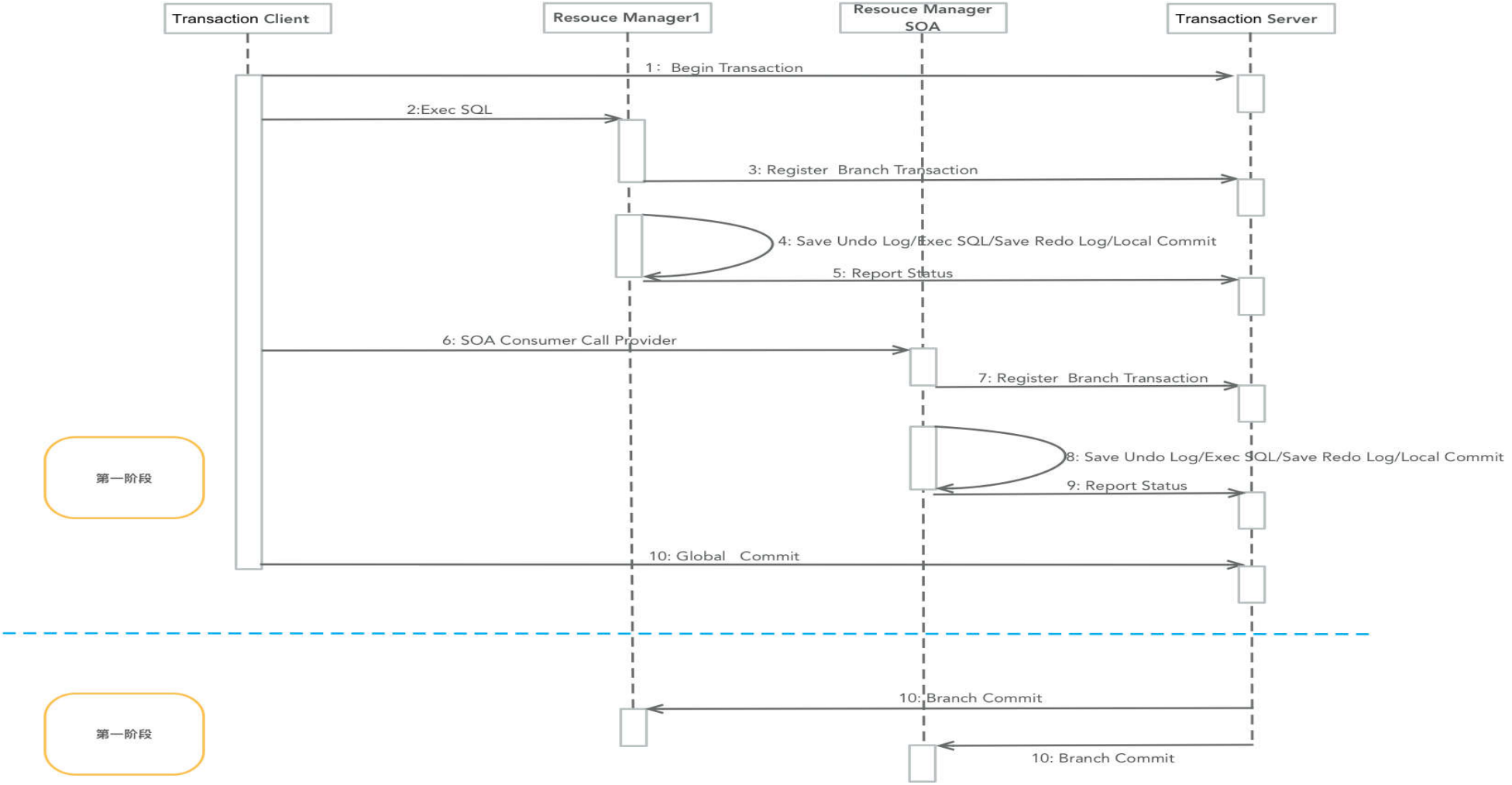
资源管理器

- ✓ 将业务的数据源（DataSource）适配成为资源管理器
- ✓ 将MQ的客户端适配成为资源管理器
- ✓ 向Dts Server注册自己的分支


SOA Rpc调用
DTS Server
Socket
事务发起者 Client
DataSource & DTS RM
MYSQL
事务发起者 Client
MQClient & DTS RM
RabbitMQ/RocketMQ
Socket

- **服务化框架充当了什么角色?**
  - ✓ **透传事务ID，当服务提供者获取到了事务Id后，将本地的资源管理器向Dts Server发起注册分支的请求**

Dts的数据流时序图


Transaction Client
Resouce Manager1
Resouce Manager SOA
Transaction Server
1： Begin Transaction
2:Exec SQL
3: Register Branch Transaction
4: Save Undo Log/Exec SQL/Save Redo Log/Local Commit
5: Report Status
6: SOA Consumer Call Provider
7: Register Branch Transaction
8: Save Undo Log/Exec SQL/Save Redo Log/Local Commit
9: Report Status
10: Global Commit
第一阶段
10: Branch Commit
10: Branch Commit
第一阶段

- **Dts如何支持回滚?**
  - ✓ **对Insert/Delete/Update SQL解析**
  - ✓ **构造查询SQL，查询并保存修改前数据(Undo)，用于回滚**
  - ✓ **执行用户SQL**
  - ✓ **查询并保存修改后数据(Redo)，用于回滚前脏数据校验**
  - ✓ **Undo/Redo Log与用户本地事务作为一个事务提交**

Dts于Xa事务不同之处

- XA依赖于数据库的XA接口
- XA在第一阶段没有提交本地事务，而事务中间件立即执行并可见
- XA在第二阶段提交各分支事务，而事务中间件清理各分支事务的Undo/Redo日志
- XA依赖于数据库的Rollback接口来回滚事务，而事务中间件通过Undo/Redo日志来实现分支事务的回滚

实际开发，（Client）

```
@DtsTransaction
public HelloReply callService() {
  HelloRequest request = new HelloRequest();
  request.setName("liushiming");
  HelloReply reply = helloService.dtsNormal(request);
  helloService.dtsException(request);
  return reply;
}
```

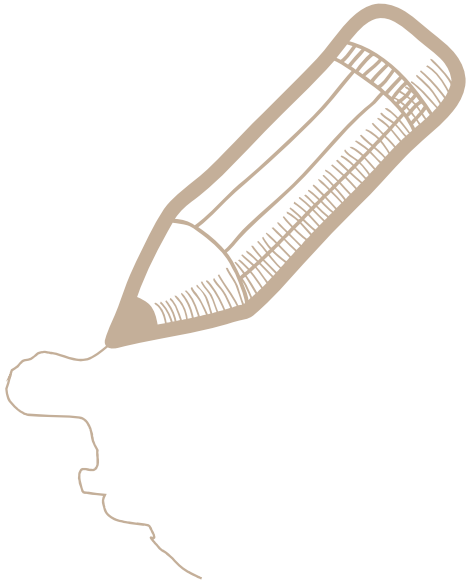
实际开发，（Resource）

```
@Bean
@Primary
public DataSource dataSource() {
  DruidDataSource datasource = new DruidDataSource();
  datasource.setUrl(this.dbUrl);
  datasource.setUsername(username);
  datasource.setPassword(password);
  datasource.setDriverClassName(driverClassName);
} catch (SQLException e) {
   logger.error("druid configuration initialization filter", e);
   }
  int startIndex = dbUrl.lastIndexOf("/");
  String databaseName = dbUrl.substring(startIndex + 1, dbUrl.length());
  datasource.setConnectionProperties(connectionProperties);
  return new DtsDataSource(datasource, databaseName);
}
```

实际开发，（Resource）

```
@Override
@Transactional
public HelloReply dtsNormal(HelloRequest hellorequest) {
  StudentDo studentDo = new StudentDo();
  studentDo.setName("liushiming");
  studentDao.save(studentDo);
  HelloReply reply = new HelloReply();
  reply.setMessage("update");
  return reply;
 }

 @Override
 public HelloReply dtsException(HelloRequest hellorequest) {
  throw new RuntimeException("rollback");
}
```



Thanks