RPC

RPC协议实际上就是约定了**远程调用过程中**

1.数据的格式

2.数据如何传输

RPC协议是应用层之上的协议，可扩展性很强

Dubbo用法示例

version版本号

在provider里面有两个不同的实现类

```
@Service(version = "async")
public class AsyncSiteServiceImpl implements SiteService {

    @Override
    public String siteName(String name) { return "async:" + name; }

    @Override
    public CompletableFuture<String> siteNameAsync(String name) {
        System.out.println("异步调用: " + name);
        return CompletableFuture.supplyAsync(() -> {
            return siteName(name);
        });
    }
}
```

```
@Service(version = "default")
public class DefaultSiteServiceImpl implements SiteService {
    @Override
    public String siteName(String name) {
        return "default:"+name;

//        URL url = RpcContext.getContext().getUrl();
//        return String.format("%s: %s, Hello, %s", url.getProto
    }
}
```

在consumer里面可以指定不同的提供者版本，获得不同的方法

```java
public class DefaultDubboConsumer {

    //@Reference(id = "siteService",version = "default",url = "dubbo://127.0.0.1:20881,
    @Reference(version = "default", loadbalance = "roundrobin")
    private SiteService siteService;

    public static void main(String[] args) {

        ConfigurableApplicationContext context = SpringApplication.run(DefaultDubboCons
//          SiteService siteService = (SiteService) context.getBean("siteService");
        SiteService siteService = (SiteService) context.getBean(SiteService.class);
        String name = siteService.siteName("q-face");
        System.out.println(name);

    }
}
```

在consumer里面可以指定实现的版本
利用多态调用不同的方法

指定protocol协议

```
#dubbo.protocol.name=dubbo
#dubbo.protocol.port=20880
#
## @Path
#dubbo.protocol.name=rest
#dubbo.protocol.port=8083
```

```
dubbo.protocols.protocol1.id=rest
dubbo.protocols.protocol1.name=rest
dubbo.protocols.protocol1.port=8083
dubbo.protocols.protocol1.host=0.0.0.0

dubbo.protocols.protocol2.id=dubbo1
dubbo.protocols.protocol2.name=dubbo
dubbo.protocols.protocol2.port=20882
dubbo.protocols.protocol2.host=0.0.0.0

dubbo.protocols.protocol3.id=dubbo2
dubbo.protocols.protocol3.name=dubbo
dubbo.protocols.protocol3.port=20883
dubbo.protocols.protocol3.host=0.0.0.0
```

可以指明需要用到的是哪一种协议

```
//@Service(version = "default",loadbalance = "roundrobin")
@Service(version = "default", protocol = "protocol1")
public class DefaultSiteServiceImpl implements SiteService {
    @Override
    public String siteName(String name) {
        return "default:"+name;

//        URL url = RpcContext.getContext().getUrl();
//        return String.format("%s: %s, Hello, %s", url.getProtocol(),
    }
}
```

使用rest协议调用服务

创建使用rest的类

```
/**
 * @author Thor
 * @公众号 Java架构栈
 */
@Service(version = "rest", protocol = "protocol1")
@Path("site")
public class RestSiteService implements SiteService {
    @Override
    @GET
    @Path("name")
    @Produces({ContentType.APPLICATION_JSON_UTF_8, ContentType.TEXT_PLAIN_UTF_8})
    public String siteName(@QueryParam("name") String name) { return "rest:" + name; }
}
```

之后就可以用这个服务，调用这个协议

```
@Service(version = "rest", protocol = "protocol1")
@Path("site")
public class RestSiteService implements SiteService {
    @Override
    @GET
    @Path("name")
    @Produces({ContentType.APPLICATION_JSON_UTF_8, ContentType.TEXT_PLAIN_UTF_8})
    public String siteName(@QueryParam("name") String name) { return "rest:" + name; }
}
```

根据这个

```
dubbo.protocols.protocol1.id=rest
dubbo.protocols.protocol1.name=rest
dubbo.protocols.protocol1.port=8083
dubbo.protocols.protocol1.host=0.0.0.0
```

rest协议在8083端口

所以可以发送 `localhost:8083/site/name?name=haohao`

消费者通过url直连指定的服务提供者

- 配置文件中声明三个dubbo协议

```
dubbo.protocols.protocol1.id=dubbo1
dubbo.protocols.protocol1.name=dubbo
dubbo.protocols.protocol1.port=20881
dubbo.protocols.protocol1.host=0.0.0.0

dubbo.protocols.protocol2.id=dubbo2
dubbo.protocols.protocol2.name=dubbo
dubbo.protocols.protocol2.port=20882
dubbo.protocols.protocol2.host=0.0.0.0

dubbo.protocols.protocol3.id=dubbo3
dubbo.protocols.protocol3.name=dubbo
dubbo.protocols.protocol3.port=20883
dubbo.protocols.protocol3.host=0.0.0.0
```

- 服务提供者暴露服务，未指定协议，则会暴露三个服务，每个协议对应一个服务

```
/**
 * @author Thor
 * @公众号 Java架构栈
 */
@Service(version = "default")
public class DefaultSiteServiceImpl implements SiteService {
    @Override
    public String siteName(String name) {
        return "default:"+name;
    }
}
```

- 消费者端通过url指定某一个服务

```
@Reference(id = "siteService",version = "default",url =
"dubbo://127.0.0.1:20881/com.qf.site.SiteService:default")
    private SiteService siteService;
```

服务超时

服务提供者和**服务消费者**都可以配置服务超时时间（默认时间为1秒，当然可以指定超时时间）

- 服务提供者的超时时间：执行该服务的超时时间。如果超时，则会打印超时日志（warn），但服务会正常执行完。

```java
/**
 * @author Thor
 * @公众号 Java架构栈
 */
@Service(version = "timeout", timeout = 4000)
public class TimeoutSiteServiceImpl implements SiteService {
    @Override
    public String siteName(String name) {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("serving...");
        return "timeout site service:"+name;
    }
}
```

集群容错

dubbo为集群调用提供了容错方案：

- failover：（默认，推荐）

当出现失败时，会进行重试，默认重试2次，一共三次调用。但是会出现幂等性问题。

虽然会出现幂等性问题，但是依然推荐使用这种容错机制，在业务层面解决幂等性问题：

- 方案一：把数据的业务id作为数据库的联合主键，此时业务id不能重复。
- 方案二（推荐）：使用分布式锁来解决重复消费问题。

- failfast：当出现失败时。立即报错，不进行重试。
- failsafe：失败不报错，记入日志。
- failback：失败就失败，开启定时任务 定时重发。
- forking：并行访问多个服务器，获取某一个结果既视为成功。

结论：如果使用dubbo，不推荐把重试关掉，而是在非幂等性操作的场景下，服务提供者方要做幂等性的解决方案（保证）。

服务降级

服务消费者通过Mock指定服务超时后执行的策略：

```java
/**
 * @author Thor
 * @公众号 Java架构栈
 */
@EnableAutoConfiguration
public class MockDubboConsumer {

    @Reference(version = "timeout", timeout = 1000, mock = "fail:return timeout")
    private SiteService siteService;


    public static void main(String[] args) {
        ConfigurableApplicationContext context = SpringApplication.run(MockDubboConsumer.class);
        SiteService siteService = (SiteService) context.getBean(SiteService.class);
        String name = siteService.siteName("q-face");
        System.out.println(name);
    }
}
```

java

- `mock=force:return+null` 表示消费方对该服务的方法调用都直接返回 null 值，不发起远程调用。用来屏蔽不重要服务不可用时对调用方的影响。
- 还可以改为 `mock=fail:return+null` 表示消费方对该服务的方法调用在失败后，再返回 null 值，不抛异常。用来容忍不重要服务不稳定时对调用方的影响。