

06 代码重构（20240320）

1. ZK注册中心ZkRegistryCenter的启动与销毁

重构前，初始化和销毁方法依赖于bean的生命周期。

<pre>@Bean(initMethod = "start", destroyMethod = "stop") public RegistryCenter providerRc() { return new ZkRegistryCenter(); }</pre>	<pre>32 32 33 33 34 34 35 35 36 36 37 37</pre>	<pre>@Bean public RegistryCenter providerRc() { return new ZkRegistryCenter(); }</pre>
--	--	--

这样带来的问题是当整个服务停止时，销毁方法先执行，客户端和zk服务端已经断开连接，而服务取消注册的逻辑再执行时，就不会成功。

```
RpcDemoConsumerApplication.java × ConsumerBootstrap.java × ZkRegistryCenter.java × ProviderBootstrap.java ×
@SneakyThrows
public void start() {...}

/**
 * 服务销毁时，取消注册
 */
@PreDestroy
public void stop() {
    skeletons.keySet().forEach(this::unregisterService);
}
```

由于取消注册使用的是quietly()方式，出错了也不会报错，最终就是服务没有取消成功，消费者还可能调用到。

```
@Override
public void unregister(ServiceMeta service, InstanceMeta instance) {
    String servicePath = "/" + service.toPath();
    try {
        // 服务路径不存在直接返回
        if (client.checkExists().forPath(servicePath) == null) {
            return;
        }

        String instancePath = servicePath + "/" + instance.toPath();
        log.info("==> unregister to zk :" + instancePath);
        // quietly删除: 没有实例也不要报错
        client.delete().quietly().forPath(instancePath);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

重构后，注册中心先启动成功，在进行服务注册。服务销毁时，先取消注册，在关闭连接。

```
dnight > rpc > core > provider > ProviderBootstrap > stop
ConsumerBootstrap.java x ZkRegistryCenter.java x ProviderBootstrap.java x Repositor

// 在bean的初始化过程中, 保存起来
@PostConstruct
public void init() {
    Map<String, Object> providers = applicationContext.getBeansWithAnnotation(R
    rc = applicationContext.getBean(RegistryCenter.class);

    providers.values().forEach(x -> genInterface(x));
}

@sneakyThrows
public void start() {
    String ip = InetAddress.getLocalHost().getHostAddress();
    instance = InstanceMeta.http(ip, Integer.valueOf(port));
    rc.start();
    // 服务端provider在启动过程中完成注册
    skeletons.keySet().forEach(this::registerService);
}

/**
 * 服务销毁时, 取消注册
 */
@PreDestroy
public void stop() {
    skeletons.keySet().forEach(this::unregisterService);
    rc.stop();
}
```

2. 抽取类型转换逻辑到工具类TypeUtils中。

这样带来的好处是业务功能和非业务功能逻辑分离, 提高代码可读性和可复用性。

Class<?> type = method.getReturnType();	57	48	rpcResponse = rpcResponse; httpInvoker.post(request, obj
System.out.println("method.getReturnType() = " + type)	58	49	
if (data instanceof JSONObject jsonResult) {	59	50	/ 处理结果
if (Map.class.isAssignableFrom(type)) {	60	51	if (Boolean.TRUE.equals(rpcResponse.getStatus())) {
Map resultMap = new HashMap();	61	52	Object data = rpcResponse.getData();
Type genericReturnType = method.getGenericRetu	62	53	return TypeUtils.castMethodResult(method, data);
System.out.println(genericReturnType);	63	54	else {
if (genericReturnType instanceof Parameterized	64	55	Exception ex = rpcResponse.getEx();
Class<?> keyType = (Class<?>) parameterize	65	56	//ex.printStackTrace();
Class<?> valueType = (Class<?>) parameteri	66	57	throw new RuntimeException(ex);
System.out.println("keyType : " + keyType	67	58	
System.out.println("valueType: " + valueTy	68	59	
jsonResult.entrySet().stream().forEach(69	60	
e -> {	70	61	
Object key = TypeUtils.cast(e.	71	62	
Object value = TypeUtils.cast(72		
resultMap.put(key, value);	73		
}	74		
);	75		

3. 网络客户端封装为接口

// 通过 OkHttp 发起http请求	116	46	
OkHttpClient client = new OkHttpClient.Builder()	117	47	// 发起远程调用
.connectionPool(new ConnectionPool(16, 60, Tir	118	48	RpcResponse rpcResponse = httpInvoker.post(request
.readTimeout(600, TimeUnit.SECONDS)	119	49	
.writeTimeout(1, TimeUnit.SECONDS)	120	50	// 处理结果
.connectTimeout(1, TimeUnit.SECONDS)	121	51	if (Boolean.TRUE.equals(rpcResponse.getStatus()))
.build();	122	52	Object data = rpcResponse.getData();
	123	53	return TypeUtils.castMethodResult(method, data
	124	54	} else {
private RpcResponse post(RpcRequest rpcRequest, String	125	55	Exception ex = rpcResponse.getEx();
String reqJson = JSON.toJSONString(rpcRequest);	126	56	//ex.printStackTrace();
	127	57	throw new RuntimeException(ex);
Request request = new Request.Builder()	128	58	}
.url(url + "/invoke")	129	59	}
.post(RequestBody.create(reqJson, JSONTYPI	130	60	
.build());	131	61	
try {	132	62	
String respJson = Objects.requireNonNull(clie	133		
return JSON.parseObject(respJson, RpcResponse	134		
} catch (IOException e) {	135		
throw new RuntimeException(e);	136		

定义HttpInvoker接口，添加当前使用的方式OkHttp作为客户端。这样基于接口设计带来的好处是想使用其他客户端时就很方便替换，不需要改变上层逻辑，添加新的实现类即可。

```

public class OkHttpInvoker implements HttpInvoker {
    private final static MediaType JSON_TYPE = MediaType.get("application/json; charset=utf-8");
    private OkHttpClient client;

    public OkHttpInvoker() {
        // 通过 OkHttp 发起http请求
        client = new OkHttpClient.Builder()
            .connectionPool(new ConnectionPool(16, 60, TimeUnit.SECONDS))
            .readTimeout(600, TimeUnit.SECONDS)
            .writeTimeout(1, TimeUnit.SECONDS)
            .connectTimeout(1, TimeUnit.SECONDS)
            .build();
    }

    @Override
    public RpcResponse<?> post(RpcRequest rpcRequest, String url) {
        String reqJson = JSON.toJSONString(rpcRequest);
    }
}

```

4. 封装ServiceMeta和InstanceMeta

16	18	// provider側
17	19	void register(String service, String instance);
18	20	
19	21	void unregister(String service, String instance);
20	22	
21	23	// consumer側
22	24	List<String> fetchAll(String service);
23	25	
24	26	void subscribe(String service, ChangedListener listener);

16	18	// provider側
17	19	void register(ServiceMeta service, InstanceMeta instance);
18	20	
19	21	void unregister(ServiceMeta service, InstanceMeta instance);
20	22	
21	23	// consumer側
22	24	List<InstanceMeta> fetchAll(ServiceMeta service);
23	25	
24	26	void subscribe(ServiceMeta service, ChangedListener listener);

使用ServiceMeta代替String类型，来表示一个服务的元数据，表达含义更加丰富，支持更多非功能性场景。

```

/**
 * 描述服务元数据
 */
@Builder
@Data
public class ServiceMeta {
    private String app;
    private String namespace;
    private String env;
    private String name;
    // private String version;

    public String toPath() {
        return String.format("%s_%s_%s_%s", app, namespace, env, name);
    }
}

```

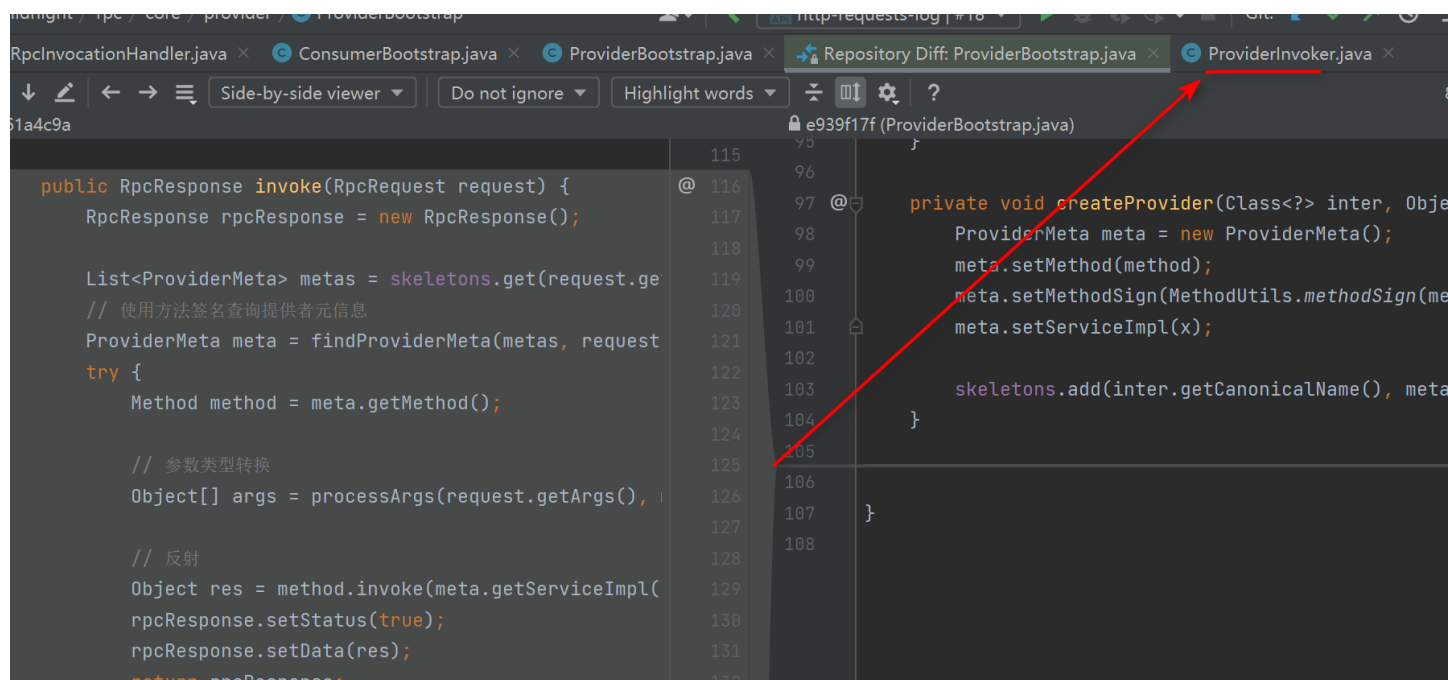
同理，使用InstanceMeta代替String类型，来表示一个实例的元数据，表达含义更加丰富，支持更多非功能性场景。

```
/**
 * 描述服务实例的云数据
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
public class InstanceMeta {
    private String schema;
    private String host;
    private Integer port;
    private String context;
    /**
     * online or offline
     */
    private boolean status;

    private Map<String, String> parameters;
```

5. 封装ProviderInvoker

将服务提供者的调用逻辑抽出独立的类，职责更单一。



此次，重构的要点

1. 业务功能和非功能性逻辑分开，提高可读性和可扩展性。
2. 使用保证类型替换String类型，丰富表达含义。
3. 工具类抽取，提高复用性。
4. 基于接口做设计，提高扩展性。
5. 类职责保持单一，解耦，类聚。

源码：<https://github.com/midnight2104/midnight-rpc/tree/lesson6>