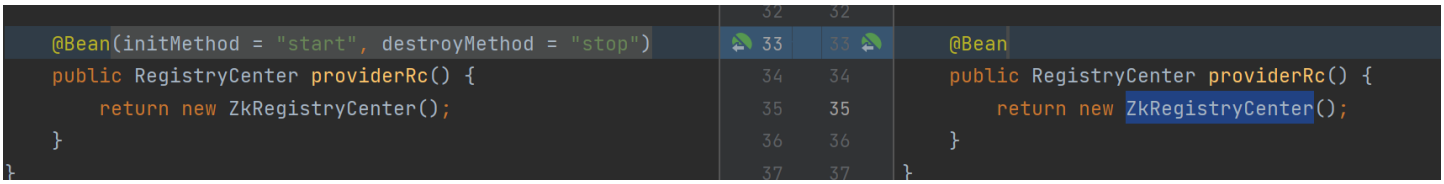
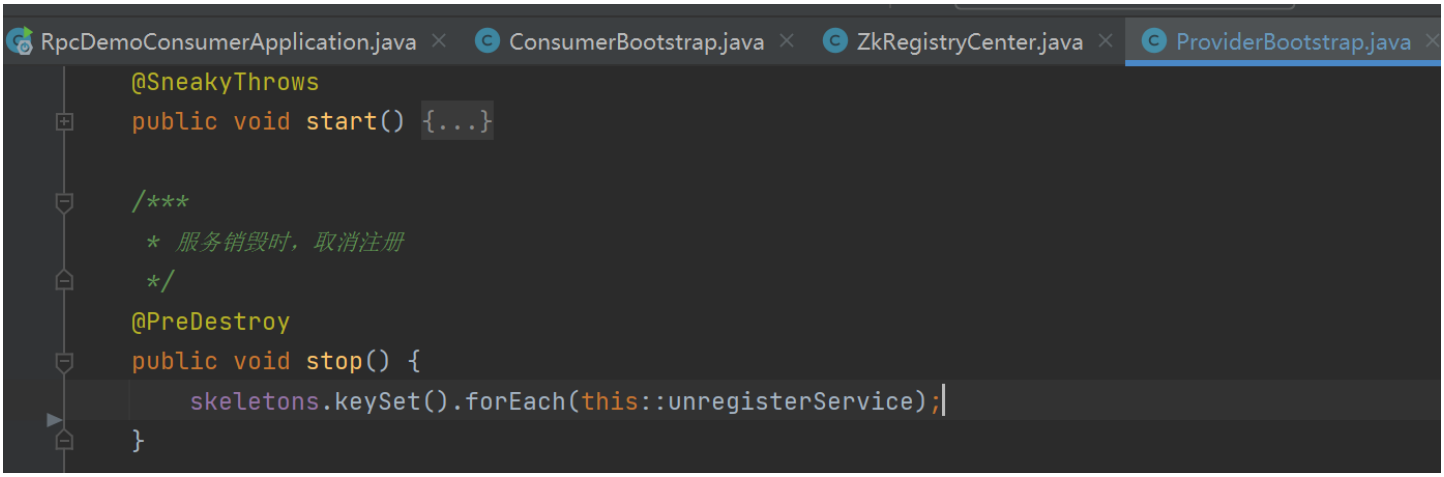
# 06 代码重构（20240320）

1. ZK注册中心ZkRegistryCenter的的启动与销毁

重构前，初始化和销毁方法依赖于bean的生命周期。



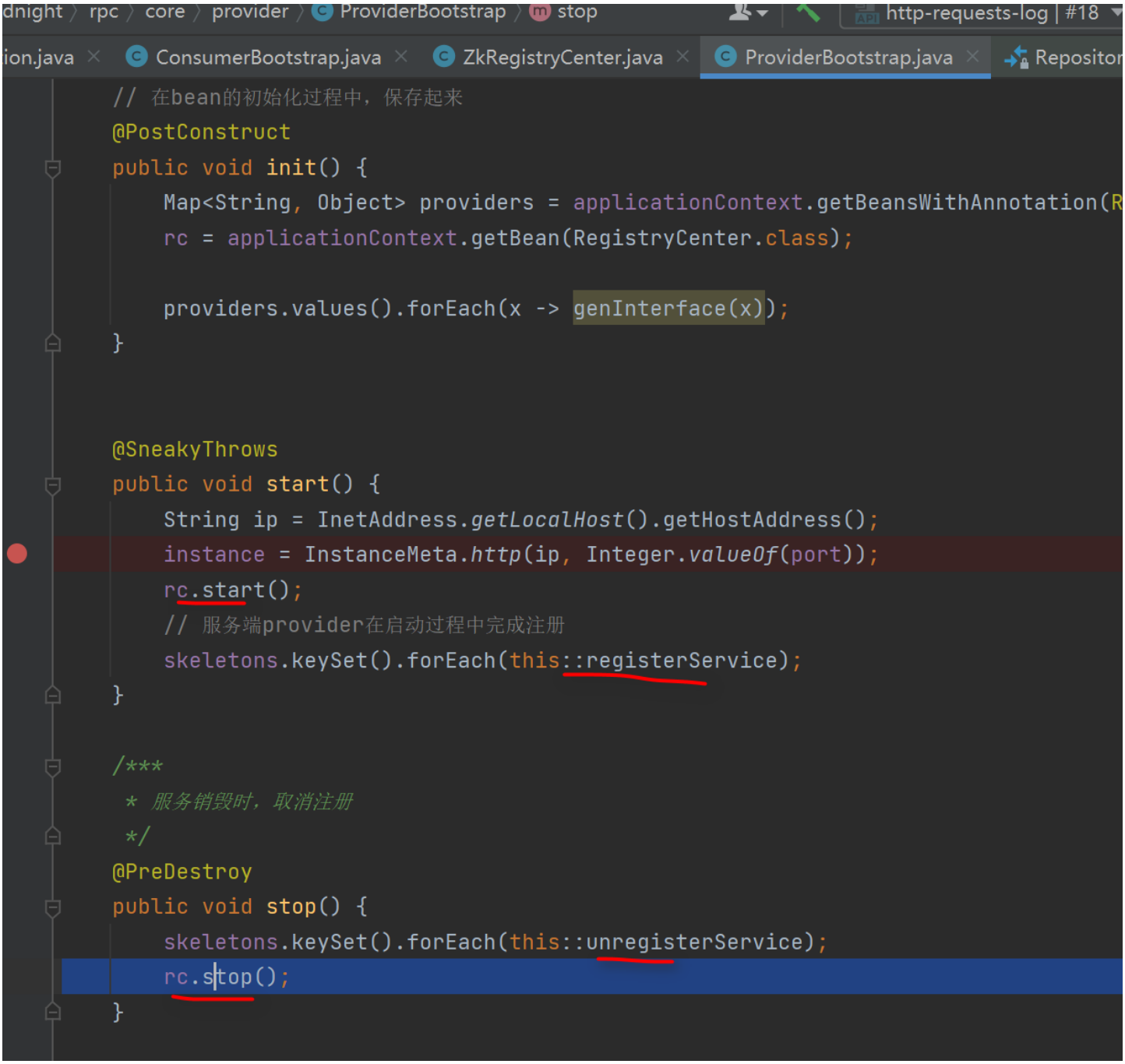

这样带来的问题是当整个服务停止时，销毁方法先执行，客户端和zk服务端已经断开连接，而服务取消注册的逻辑再执行时，就不会成功。




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。

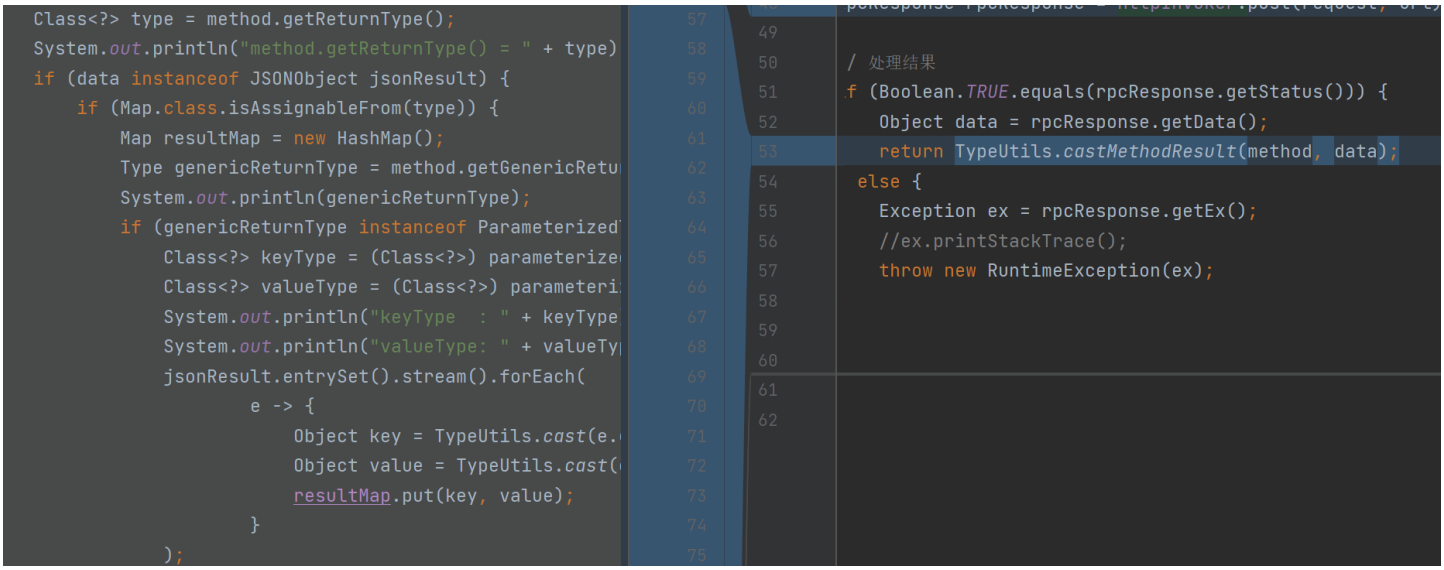
```java
@Override
public void unregister(ServiceMeta service, InstanceMeta instance) {
    String servicePath = "/" + service.toPath();
    try {
        // 服务路径不存在直接返回
        if (client.checkExists().forPath(servicePath) == null) {
            return;
        }

        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。




2. 抽取类型转换逻辑到工具类TypeUtils中。

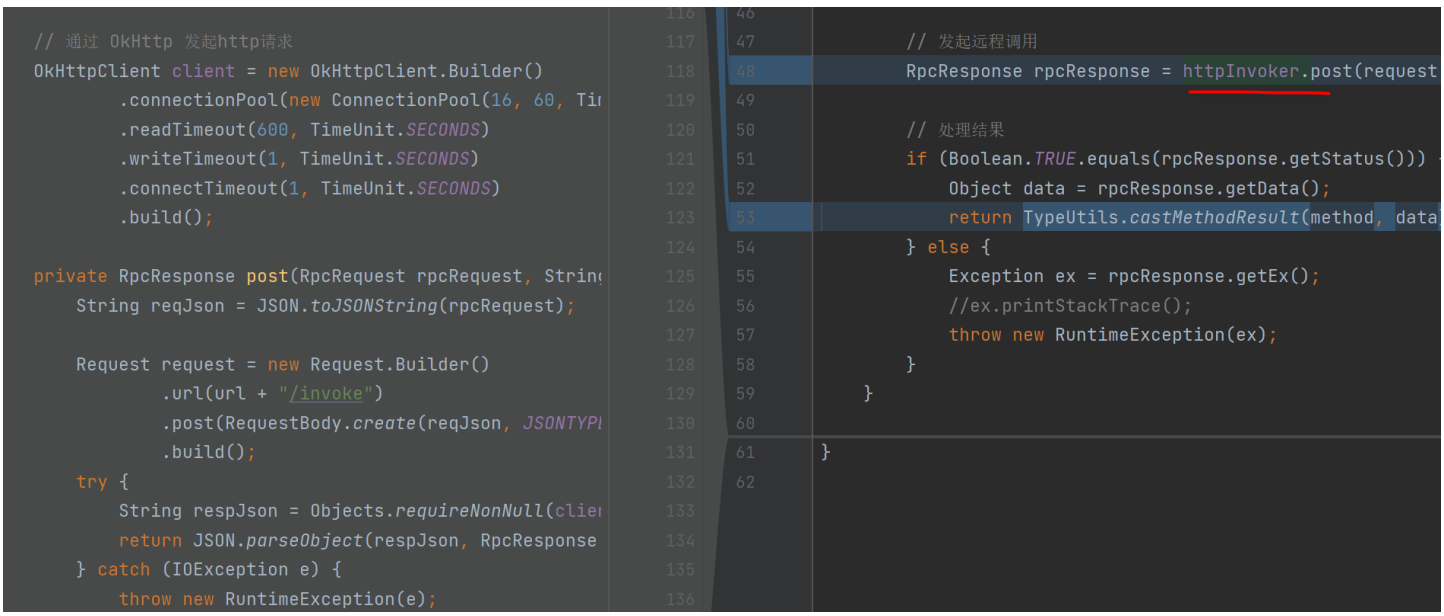
这样带来的好处是业务功能和非业务功能逻辑分离，提高代码可读性和可复用性。



```
Class<?> type = method.getReturnType();
System.out.println("method.getReturnType() = " + type)
if (data instanceof JSONObject jsonResult) {
    if (Map.class.isAssignableFrom(type)) {
        Map resultMap = new HashMap();
        Type genericReturnType = method.getGenericRetu
        System.out.println(genericReturnType);
        if (genericReturnType instanceof Parameterized
            Class<?> keyType = (Class<?>) parameterize
            Class<?> valueType = (Class<?>) parameteri
            System.out.println("keyType   : " + keyType
            System.out.println("valueType: " + valueTy
            jsonResult.entrySet().stream().forEach(
                    e -> {
                        Object key = TypeUtils.cast(e.
                        Object value = TypeUtils.cast(
                        resultMap.put(key, value);
                    }
            );
```

```
/ 处理结果
f (Boolean.TRUE.equals(rpcResponse.getStatus())) {
    Object data = rpcResponse.getData();
    return TypeUtils.castMethodResult(method, data);
 else {
    Exception ex = rpcResponse.getEx();
    //ex.printStackTrace();
    throw new RuntimeException(ex);
```

## 3. 网络客户端封装为接口



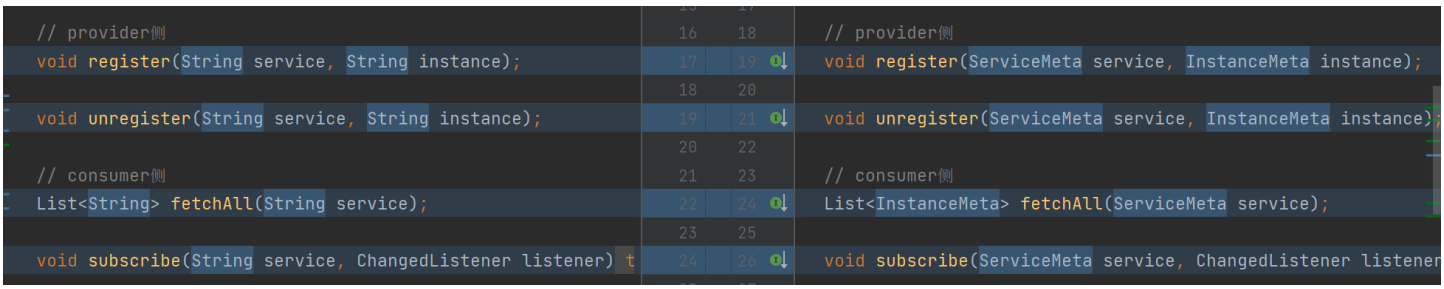
```
// 通过 OkHttp 发起http请求
OkHttpClient client = new OkHttpClient.Builder()
        .connectionPool(new ConnectionPool(16, 60, Ti
        .readTimeout(600, TimeUnit.SECONDS)
        .writeTimeout(1, TimeUnit.SECONDS)
        .connectTimeout(1, TimeUnit.SECONDS)
        .build();

private RpcResponse post(RpcRequest rpcRequest, Strin
    String reqJson = JSON.toJSONString(rpcRequest);

    Request request = new Request.Builder()
            .url(url + "/invoke")
            .post(RequestBody.create(reqJson, JSONTYP
            .build();
    try {
        String respJson = Objects.requireNonNull(clie
        return JSON.parseObject(respJson, RpcResponse
    } catch (IOException e) {
        throw new RuntimeException(e);
```

```
        // 发起远程调用
        RpcResponse rpcResponse = httpInvoker.post(request

        // 处理结果
        if (Boolean.TRUE.equals(rpcResponse.getStatus())) {
            Object data = rpcResponse.getData();
            return TypeUtils.castMethodResult(method, data
        } else {
            Exception ex = rpcResponse.getEx();
            //ex.printStackTrace();
            throw new RuntimeException(ex);
        }
    }
}
```

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时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。

```java
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



```
// provider侧
void register(String service, String instance);

void unregister(String service, String instance);

// consumer侧
List<String> fetchAll(String service);

void subscribe(String service, ChangedListener listener) t
```

```
// provider侧
void register(ServiceMeta service, InstanceMeta instance);

void unregister(ServiceMeta service, InstanceMeta instance);

// consumer侧
List<InstanceMeta> fetchAll(ServiceMeta service);

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。

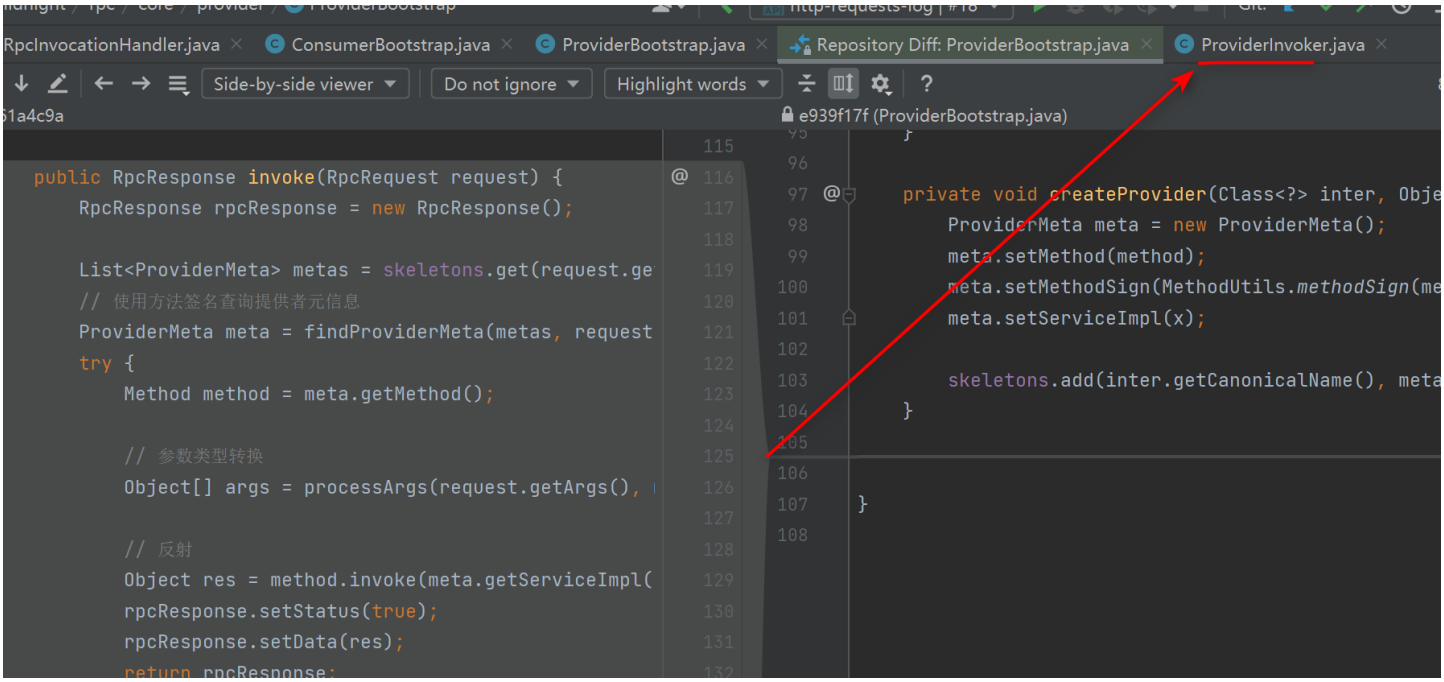
```java
/**
 * 描述服务元数据
 */
@Builder
@Data
public class ServiceMeta {
    private String app;
    private String namespace;
    private String env;
    private String name;
//    private String version;

    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。

```java
/**
 * 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