Dubbo的高级用法

API的方式发布和引用服务

以API 配置的方式来配置你的 Dubbo 应用

通过API编码方式组装配置，启动Dubbo，发布及订阅服务。此方式可以支持动态创建ReferenceConfig/ServiceConfig，结合泛化调用可以满足API Gateway或测试平台的需要。

- 服务提供者

  通过ServiceConfig暴露服务接口，发布服务接口到注册中心。

```
public class ProviderApi {
    public static void main(String[] args) throws IOException {
        new EmbeddedZooKeeper(2181, false).start();
        // 服务实现
        UserService demoService = new UserServiceImpl();

        // 当前应用配置
        ApplicationConfig application = new ApplicationConfig();
        application.setName("demo-provider");

        // 连接注册中心配置
        RegistryConfig registry = new RegistryConfig();
        registry.setAddress("zookeeper://127.0.0.1:2181");

        // 服务提供者协议配置
        ProtocolConfig protocol = new ProtocolConfig();
        protocol.setName("dubbo");
        protocol.setPort(12345);
        protocol.setThreads(200);

        // 注意：ServiceConfig为重对象，内部封装了与注册中心的连接，以及开启服务端口
        // 服务提供者暴露服务配置
        ServiceConfig<UserService> service = new ServiceConfig<UserService>(); // 此实例很重，封装了与注册中心的连接，请自行缓存，否则可能造成内存和连接泄漏
        service.setApplication(application);
        service.setRegistry(registry); // 多个注册中心可以用setRegistries()
        service.setProtocol(protocol); // 多个协议可以用setProtocols()
        service.setInterface(UserService.class);
        service.setRef(demoService);
        service.setVersion("1.0.0");

        // 暴露及注册服务
        service.export();
```

```
        // 挂起等待(防止进程退出)
        System.in.read();
    }
}
```

- 服务消费者

  通过ReferenceConfig引用远程服务，从注册中心订阅服务接口。

```
@Test
    public void api() {
        ApplicationConfig applicationConfig = new ApplicationConfig();
        applicationConfig.setName("dubbo_consumer");

        RegistryConfig registryConfig = new RegistryConfig();
        registryConfig.setAddress("zookeeper://127.0.0.1:2181");

        ReferenceConfig<UserService> referenceConfig = new ReferenceConfig<>();
        referenceConfig.setApplication(applicationConfig);
        referenceConfig.setRegistry(registryConfig);
        referenceConfig.setInterface(UserService.class);
        UserService userService = referenceConfig.get();
        System.out.println(userService.queryUser("jack"));
    }
```

- bootstrap 服务发布

```
public class BootstrapApi {
    public static void main(String[] args) {
        new EmbeddedZooKeeper(2181, false).start();
        ConfigCenterConfig configCenter = new ConfigCenterConfig();
        configCenter.setAddress("zookeeper://127.0.0.1:2181");

        // 服务提供者协议配置
        ProtocolConfig protocol = new ProtocolConfig();
        protocol.setName("dubbo");
        protocol.setPort(12345);
        protocol.setThreads(200);

        // 注意：ServiceConfig为重对象，内部封装了与注册中心的连接，以及开启服务端口
        // 服务提供者暴露服务配置
        ServiceConfig<UserService> demoServiceConfig = new ServiceConfig<>();
        demoServiceConfig.setInterface(UserService.class);
        demoServiceConfig.setRef(new UserServiceImpl());
        demoServiceConfig.setVersion("1.0.0");

        // 第二个服务配置
        ServiceConfig<MockService> fooServiceConfig = new ServiceConfig<>();
        fooServiceConfig.setInterface(MockService.class);
        fooServiceConfig.setRef(new MockServiceImpl());
        fooServiceConfig.setVersion("1.0.0");
```

```
        // 通过DubboBootstrap简化配置组装，控制启动过程
        DubboBootstrap.getInstance()
                .application("demo-provider") // 应用配置
                .registry(new RegistryConfig("zookeeper://127.0.0.1:2181")) // 注册中心
配置
                .protocol(protocol) // 全局默认协议配置
                .service(demoServiceConfig) // 添加ServiceConfig
                .service(fooServiceConfig)
                .start()    // 启动Dubbo
                .await();   // 挂起等待(防止进程退出)
    }
}
```

- bootstrap 服务发现

```
public class BootstrapApi {
    public static void main(String[] args) {
        // 引用远程服务
        ReferenceConfig<UserService> demoServiceReference = new
ReferenceConfig<UserService>();
        demoServiceReference.setInterface(UserService.class);
        demoServiceReference.setVersion("1.0.0");

        ReferenceConfig<MockService> fooServiceReference = new
ReferenceConfig<MockService>();
        fooServiceReference.setInterface(MockService.class);
        fooServiceReference.setVersion("1.0.0");

        // 通过DubboBootstrap简化配置组装，控制启动过程
        DubboBootstrap bootstrap = DubboBootstrap.getInstance();
        bootstrap.application("demo-consumer") // 应用配置
                .registry(new RegistryConfig("zookeeper://127.0.0.1:2181")) // 注册中心
配置
                .reference(demoServiceReference) // 添加ReferenceConfig
                .reference(fooServiceReference)
                .start();    // 启动Dubbo

        // 和本地bean一样使用demoService
        // 通过Interface获取远程服务接口代理，不需要依赖ReferenceConfig对象
        UserService demoService =
DubboBootstrap.getInstance().getCache().get(UserService.class);
        System.out.println(demoService.queryUser("jack"));

        MockService fooService =
DubboBootstrap.getInstance().getCache().get(MockService.class);
        System.out.println(fooService.queryArea("1"));
    }
}
```

dubbo高级用法

启动是检查

通过spring配置文件

关闭某个服务的启动时检查 (没有提供者时报错):

如果不关闭该属性，当该服务没有发布时如果消费者引用了该服务则会报错。

```
<dubbo:reference interface="com.foo.BarService" check="false" />
```

关闭所有服务的启动时检查 (没有提供者时报错):

```
<dubbo:consumer check="false" />
```

关闭注册中心启动时检查 (注册订阅失败时报错):

```
<dubbo:registry check="false" />
```

通过 dubbo.properties

```
dubbo.reference.com.foo.BarService.check=false
dubbo.consumer.check=false
dubbo.registry.check=false
```

通过 -D 参数

```
java -Ddubbo.reference.com.foo.BarService.check=false
java -Ddubbo.consumer.check=false
java -Ddubbo.registry.check=false
```

直连提供者

Dubbo 中点对点的直连方式

在开发及测试环境下，经常需要绕过注册中心，只测试指定服务提供者，这时候可能需要点对点直连，点对点直连方式，将以服务接口为单位，忽略注册中心的提供者列表，A 接口配置点对点，不影响 B 接口从注册中心获取列表。

配置:

```
@DubboReference(check = false, url = "dubbo://localhost:20880")
```

服务分组

使用服务分组区分服务接口的不同实现

当一个接口有多种实现时，可以用 group 区分。

配置:

服务端

```
@DubboService(group = "groupImpl1")
public class GroupImpl1 implements Group {
    @Override
    public String doSomething(String s) {
        System.out.println("===========GroupImpl1.doSomething");
        return "GroupImpl1.doSomething";
    }
}
```

```
@DubboService(group = "groupImpl2")
public class GroupImpl2 implements Group {
    @Override
    public String doSomething(String s) {
        System.out.println("===========GroupImpl2.doSomething");
        return "GroupImpl2.doSomething";
    }
}
```

消费端

```
@DubboReference(check = false,group = "groupImpl1")
Group group;
```

分组聚合

通过分组对结果进行聚合并返回聚合后的结果

通过分组对结果进行聚合并返回聚合后的结果，比如菜单服务，用group区分同一接口的多种实现，现在消费方需从每种group中调用一次并返回结果，对结果进行合并之后返回，这样就可以实现聚合菜单项。

生产者配置：

```
@DubboService(group = "groupImpl1")
public class GroupImpl1 implements Group {
    @Override
    public String doSomething(String s) {
        System.out.println("===========GroupImpl1.doSomething");
        return "GroupImpl1.doSomething";
    }
}

@DubboService(group = "groupImpl2")
public class GroupImpl2 implements Group {
    @Override
    public String doSomething(String s) {
        System.out.println("===========GroupImpl2.doSomething");
        return "GroupImpl2.doSomething";
    }
}
```

消费者配置：

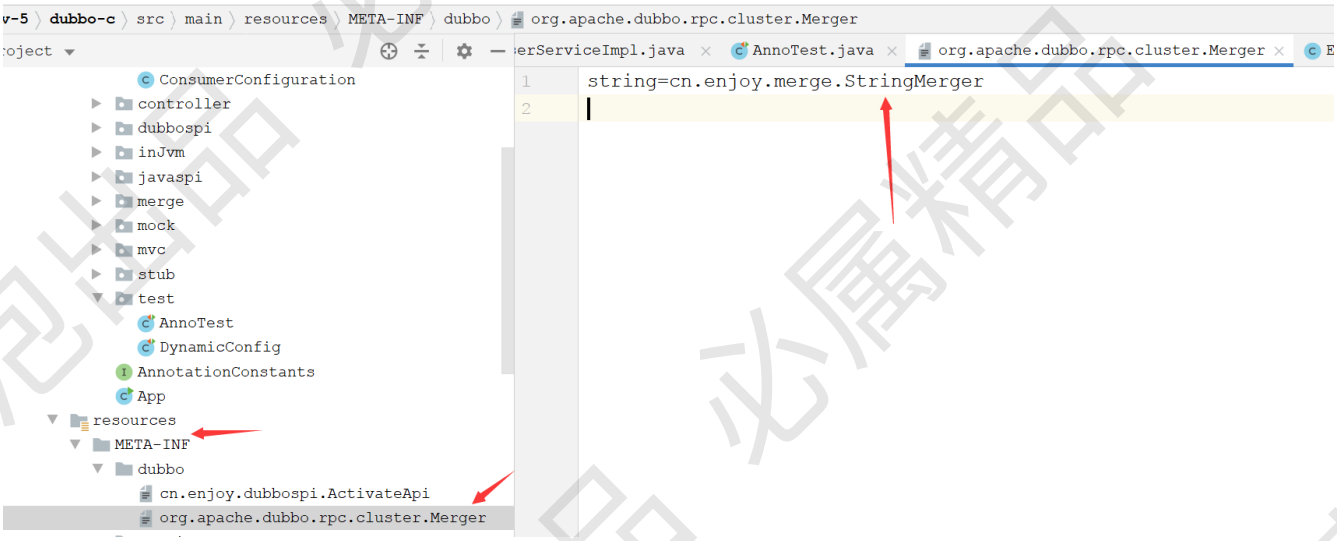
```
@DubboReference(check = false,group = "*",parameters = {"merger","true"})
Group group;
```

SPI文件配置

在resources下创建META-INF文件夹并在其下面创建dubbo文件夹，然后在dubbo文件夹下面创建org.apache.dubbo.rpc.cluster.Merger文件，在该文件下写好Merger的实现类，如：

```
string=cn.enjoy.merge.StringMerger
```

截图如下：



StringMerger类：

```
public class StringMerger implements Merger<String> {

    //定义了所有group实现类的返回值的合并规则
    @Override
    public String merge(String... strings) {
        if(strings.length == 0) {
            return null;
        }
        StringBuilder builder = new StringBuilder();
        for (String string : strings) {
            if(string != null) {
                builder.append(string).append("-");
            }
        }
        return builder.toString();
    }
}
```

多版本

在 Dubbo 中为同一个服务配置多个版本

当一个接口实现，出现不兼容升级时，可以用版本号过渡，版本号不同的服务相互间不引用。

可以按照以下的步骤进行版本迁移：

1. 在低压力时间段，先升级一半提供者为新版本
2. 再将所有消费者升级为新版本
3. 然后将剩下的一半提供者升级为新版本

生产者配置：

```
@DubboService(version = "1.0.0")
public class VersionServiceImpl implements VersionService {
    @Override
    public String version(String s) {
        System.out.println("========VersionServiceImpl.1.0.0");
        return "========VersionServiceImpl.1.0.0";
    }
}

@DubboService(version = "1.0.1")
public class VersionServiceImpl1 implements VersionService {
    @Override
    public String version(String s) {
        System.out.println("========VersionServiceImpl1.1.0.1");
        return "========VersionServiceImpl1.1.0.1";
    }
}
```

消费者配置：

指定不同的版本调用

```
@DubboReference(check = false,version = "1.0.0")
VersionService versionService;
```

参数验证

在 Dubbo 中进行参数验证

参数验证功能是基于 JSR303 实现的，用户只需标识 JSR303 标准的验证 annotation，并通过声明 filter 来实现验证。

Maven 依赖

```
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>1.0.0.GA</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.2.0.Final</version>
</dependency>
```

参数验证类:

```
public class ValidationParamter implements Serializable {
    private static final long serialVersionUID = 32544321432L;
    @NotNull
    @Size(
        min = 2,
        max = 20
    )
    private String name;
    @Min(18L)
    @Max(100L)
    private int age;
    @Past
    private Date loginDate;
    @Future
    private Date expiryDate;
}
```

生产者:

```
@DubboService
public class ValidationServiceImpl implements ValidationService {
    @Override
    public void save(ValidationParamter validationParamter) {
        System.out.println("========ValidationServiceImpl.save");
    }

    @Override
    public void update(ValidationParamter validationParamter) {
        System.out.println("========ValidationServiceImpl.update");
    }

    @Override
    public void delete(long l, String s) {
        System.out.println("========ValidationServiceImpl.delete");
    }
}
```

消费者:

```
@DubboReference(check = false,validation = "true")
ValidationService validationService;

@Test
public void validation() {
    ValidationParamter paramter = new ValidationParamter();
    paramter.setName("Jack");
    paramter.setAge(98);
    paramter.setLoginDate(new Date(System.currentTimeMillis() - 10000000));
    paramter.setExpiryDate(new Date(System.currentTimeMillis() + 10000000));
    validationService.save(paramter);
}
```

使用泛化调用

实现一个通用的服务测试框架，可通过 `GenericService` 调用所有服务实现

泛化接口调用方式主要用于客户端没有 API 接口及模型类元的情况，参数及返回值中的所有 POJO 均用 `Map` 表示，通常用于框架集成，比如：实现一个通用的服务测试框架，可通过 `GenericService` 调用所有服务实现。

消费者：

```
@Test
public void usegeneric() {
    ApplicationConfig applicationConfig = new ApplicationConfig();
    applicationConfig.setName("dubbo_consumer");

    RegistryConfig registryConfig = new RegistryConfig();
    registryConfig.setAddress("zookeeper://192.168.67.139:2184");

    ReferenceConfig<GenericService> referenceConfig = new ReferenceConfig<>();
    referenceConfig.setApplication(applicationConfig);
    referenceConfig.setInterface("com.xiangxue.jack.service.UserService");
    //这个是使用泛化调用
    referenceConfig.setGeneric(true);
    GenericService genericService = referenceConfig.get();

    Object result = genericService.$invoke("queryUser", new String[]{"java.lang.String"},
new Object[]{"Jack"});
    System.out.println(result);
}
```

本地调用

在 Dubbo 中进行本地调用

本地调用使用了 injvm 协议，是一个伪协议，它不开启端口，不发起远程调用，只在 JVM 内直接关联，但执行 Dubbo 的 Filter 链。

本地调用，调用的就是本地工程的接口实例

示例：

```
@DubboReference(check = false,injvm = true)
StudentService studentService;

@Test
public void inJvm() throws InterruptedException {
    System.out.println(studentService.find("xx"));
}
```

参数回调

通过参数回调从服务器端调用客户端逻辑

参数回调方式与调用本地 callback 或 listener 相同，只需要在 Spring 的配置文件中声明哪个参数是 callback 类型即可。Dubbo 将基于长连接生成反向代理，这样就可以从服务器端调用客户端逻辑。

服务接口示例：

```
public interface CallbackService {
    void addListener(String var1, CallbackListener var2);
}
```

CallbackListener.java

```
public interface CallbackListener {
    void changed(String msg);
}
```

生产者：

```
@DubboService(methods = {@Method(name = "addListener", arguments = {@Argument(index = 1,
callback = true)})})
public class CallbackServiceImpl implements CallbackService {
    @Override
    public void addListener(String s, CallbackListener callbackListener) {
        //这里就是回调客户端的方法
        callbackListener.changed(getChanged(s));
    }

    private String getChanged(String key) {
        return "Changed: " + new SimpleDateFormat("yyyy-MM-dd:mm:ss").format(new Date());
    }
}
```

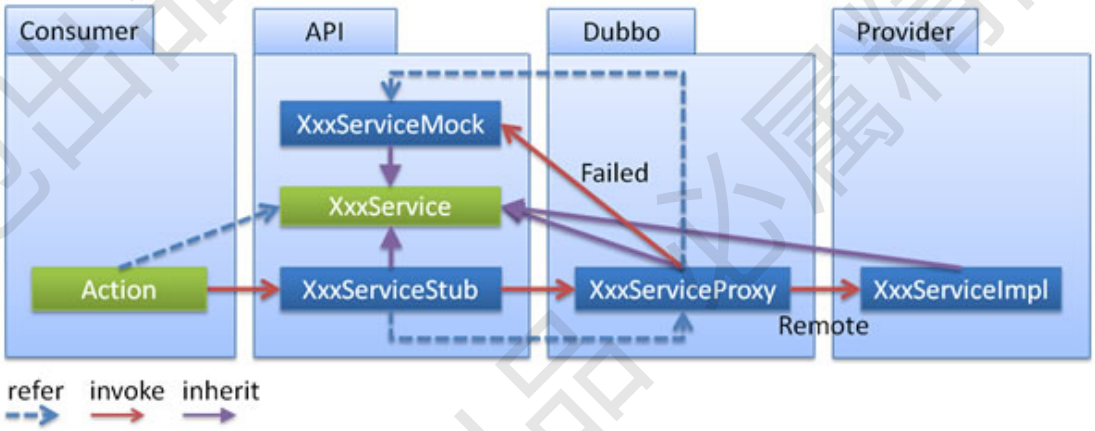
消费者：

```
@DubboReference(check = false)
CallbackService callbackService;

callbackService.addListener("jack", new CallbackListener() {
            public void changed(String arg0) {
                System.out.println("======================callback result: "
                        + arg0);
            }
});
```

本地存根

在 Dubbo 中利用本地存根在客户端执行部分逻辑

远程服务后，客户端通常只剩下接口，而实现全在服务器端，但提供方有些时候想在客户端也执行部分逻辑，比如：做 ThreadLocal 缓存，提前验证参数，调用失败后伪造容错数据等等，此时就需要在 API 中带上 Stub，客户端生成 Proxy 实例，会把 Proxy 通过构造函数传给 Stub 1，然后把 Stub 暴露给用户，Stub 可以决定要不要去调 Proxy。



消费者：

```
@DubboReference(check = false,stub = "cn.enjoy.stub.LocalStubProxy")
StubService stubService;
```

```
 * 接管了studService的调用
 *
 * 只有这个类才会决定要不要远程调用
 */
public class LocalStubProxy implements StubService {

    private StubService stubService;

    //这个必须要，传stubService实例本身
    public LocalStubProxy(StubService stubService) {
        this.stubService = stubService;
    }

    @Override
    public String stub(String s) {
        //代码在客户端执行，你可以在客户端做ThreadLocal本地缓存，或者校验参数之类工作的
```

```
        try {
            //用目标对象掉对应的方法   远程调用
            return stubService.stub(s);
        }catch (Exception e) {
            //用来降级
            System.out.println("降级数据");
        }
        //掉完后又执行代码
        return null;
    }
}
```

本地伪装

如何在 Dubbo 中利用本地伪装实现服务降级

本地伪装通常用于服务降级，比如某验权服务，当服务提供方全部挂掉后，客户端不抛出异常，而是通过 Mock 数据返回授权失败。

消费者：

```
    //这两种方式会走rpc远程调用  fail -- 会走远程服务
    @DubboReference(check = false,mock = "true")
//    @DubboReference(check = false,mock = "cn.enjoy.mock.LocalMockService")

    //不走服务直接降级  force -- 是不会走远程服务的，强制降级..这种方式是用dubbo-admin去配置它，服务治理的方式
//    @DubboReference(check = false,mock = "force:return jack")
    MockService mockService;
```

```
 * MockServiceMock
 *
 * 接口名+"Mock"
 */
public class MockServiceMock implements MockService {
    @Override
    public String mock(String s) {
        System.out.println(this.getClass().getName() + "--mock");
        return s;
    }

    @Override
    public String queryArea(String s) {
        System.out.println(this.getClass().getName() + "--queryArea");
        return s;
    }

    @Override
    public String queryUser(String s) {
        System.out.println(this.getClass().getName() + "--queryUser");
        return s;
    }
}
```

粘滞连接

为有状态服务配置粘滞连接

粘滞连接用于有状态服务，尽可能让客户端总是向同一提供者发起调用，除非该提供者挂了，再连另一台。

粘滞连接将自动开启延迟连接，以减少长连接数。

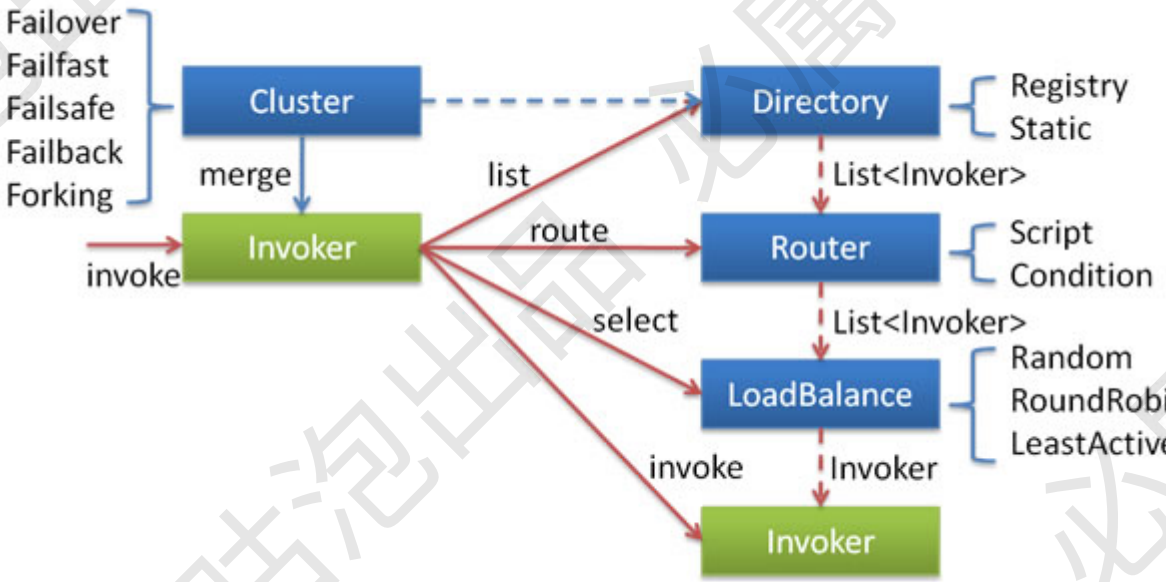
sticky=true

```
@DubboReference(check = false,protocol = "dubbo",retries = 3,timeout = 1000000000,cluster =
"failover",loadbalance = "random",sticky = true,methods = {@Method(name = "doKill",isReturn
= false)}/*,url = "dubbo://localhost:20880"*/)
UserService userService;
```

集群容错

集群调用失败时，Dubbo 提供的容错方案

在集群调用失败时，Dubbo 提供了多种容错方案，缺省为 failover 重试。



各节点关系：

- 这里的 `Invoker` 是 `Provider` 的一个可调用 `Service` 的抽象，`Invoker` 封装了 `Provider` 地址及 `Service` 接口信息
- `Directory` 代表多个 `Invoker`，可以把它看成 `List<Invoker>` ，但与 `List` 不同的是，它的值可能是动态变化的，比如注册中心推送变更
- `Cluster` 将 `Directory` 中的多个 `Invoker` 伪装成一个 `Invoker`，对上层透明，伪装过程包含了容错逻辑，调用失败后，重试另一个
- `Router` 负责从多个 `Invoker` 中按路由规则选出子集，比如读写分离，应用隔离等
- `LoadBalance` 负责从多个 `Invoker` 中选出具体的一个用于本次调用，选的过程包含了负载均衡算法，调用失败后，需要重选

集群容错模式

Failover Cluster

失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长延迟。可通过 `retries="2"` 来设置重试次数(不含第一次)。该配置为缺省配置

Failfast Cluster

快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

Failsafe Cluster

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 `forks="2"` 来设置最大并行数。

Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错。通常用于通知所有提供者更新缓存或日志等本地资源信息。

Available Cluster

调用目前可用的实例（只调用一个），如果当前没有可用的实例，则抛出异常。通常用于不需要负载均衡的场景。

配置：

```
@reference(cluster = "broadcast", parameters = {"broadcast.fail.percent", "20"})
```

负载均衡

Dubbo 提供的集群负载均衡策略

在集群负载均衡时，Dubbo 提供了多种均衡策略，缺省为 `random` 随机调用。

具体实现上，Dubbo 提供的是客户端负载均衡，即由 Consumer 通过负载均衡算法得出需要将请求提交到哪个 Provider 实例。

负载均衡策略

目前 Dubbo 内置了如下负载均衡算法，用户可直接配置使用：

算法	特性	备注
RandomLoadBalance	加权随机	默认算法，默认权重相同
RoundRobinLoadBalance	加权轮询	借鉴于 Nginx 的平滑加权轮询算法，默认权重相同，
LeastActiveLoadBalance	最少活跃优先 + 加权随机	背后是能者多劳的思想
ShortestResponseLoadBalance	最短响应优先 + 加权随机	更加关注响应速度
ConsistentHashLoadBalance	一致性 Hash	确定的入参，确定的提供者，适用于有状态请求

配置：

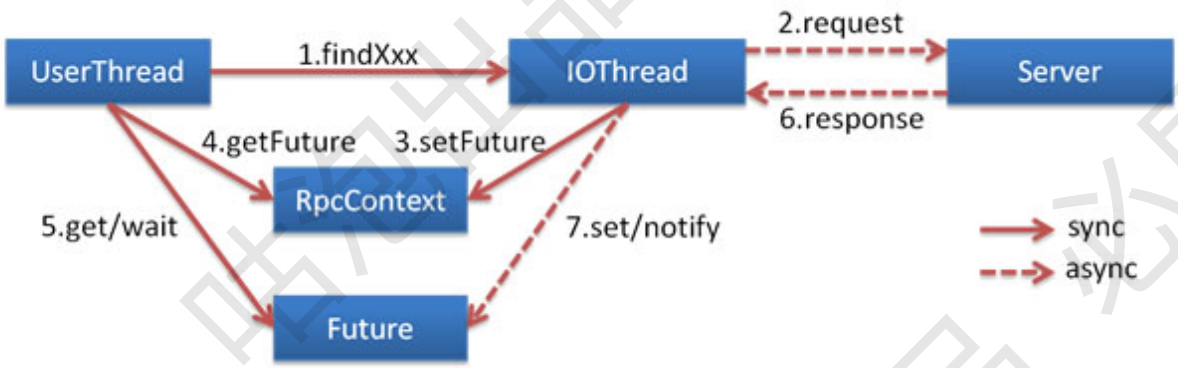
```
<dubbo:service interface="..." loadbalance="roundrobin" />
```

异步调用

在 Dubbo 中发起异步调用

从 2.7.0 开始，Dubbo 的所有异步编程接口开始以 CompletableFuture 为基础

基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小。



代码案例

生产者：

```
public interface AsyncService {
    String asynctoDo(String var1);
}

@DubboService
public class AsyncServiceImpl implements AsyncService {
```

```
    @Override
    public String asynctoDo(String s) {
        for (int i = 0; i < 10; i++) {
            System.out.println("===============AsyncServiceImpl.asynctoDo");
            try {
                Thread.sleep(100);
            }
            catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        return "hello," + s;
    }
}
```

消费者：

```
@DubboReference(check = false,timeout = 1000000000,methods = {@Method(name =
"asynctoDo",async = true)})
AsyncService asyncService;

@Test
public void async() throws ExecutionException, InterruptedException {
    String aa = asyncService.asynctoDo("aa");
    System.out.println("main====" + aa);
    //这里是主线程阻塞等待返回结果
    CompletableFuture<Object> resultFuture = RpcContext.getContext().getCompletableFuture();
    resultFuture.whenComplete((retValue,exception)->{
        if(exception == null) {
            System.out.println(retValue);
        } else {
            exception.printStackTrace();
        }
    });
    Thread.currentThread().join();
}
```

异步执行

Dubbo 服务提供方的异步执行

Provider端异步执行将阻塞的业务从Dubbo内部线程池切换到业务自定义线程，避免Dubbo线程池的过度占用，有助于避免不同服务间的互相影响。异步执行无异于节省资源或提升RPC响应性能，因为如果业务执行需要阻塞，则始终还是要有线程来负责执行。

注意：

Provider 端异步执行和 Consumer 端异步调用是相互独立的，你可以任意正交组合两端配置

- Consumer同步 - Provider同步
- Consumer异步 - Provider同步
- Consumer同步 - Provider异步

- Consumer异步 - Provider异步

代码案例

生产者：

```
public interface AsyncService {
    CompletableFuture<String> doOne(String var1);
}

@Override
public CompletableFuture<String> doOne(String s) {
    return CompletableFuture.supplyAsync(()->{
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "doOne -- OK";
    });
}
```

消费者：

```
@Test
public void asyncDoone() throws ExecutionException, InterruptedException {
    CompletableFuture<String> wuya = asyncService.doOne("wuya");
    wuya.whenComplete((retValue,exception)->{
        if(exception == null) {
            System.out.println(retValue);
        } else {
            exception.printStackTrace();
        }
    });
    Thread.currentThread().join();
}
```

Protobuf

使用 IDL 定义服务

当前 Dubbo 的服务定义和具体的编程语言绑定，没有提供一种语言中立的服务描述格式，比如 Java 就是定义 Interface 接口，到了其他语言又得重新以另外的格式定义一遍。 2.7.5 版本通过支持 Protobuf IDL 实现了语言中立的服务定义。

1、maven插件支持

```
<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <source.level>1.8</source.level>
    <target.level>1.8</target.level>
    <dubbo.version>3.0.2.1</dubbo.version>
```

```
    <spring.version>5.2.8.RELEASE</spring.version>
    <junit.version>4.12</junit.version>
    <maven-compiler-plugin.version>3.7.0</maven-compiler-plugin.version>
    <skip_maven_deploy>true</skip_maven_deploy>
    <dubbo.compiler.version>0.0.2</dubbo.compiler.version>
</properties>

<build>
  <extensions>
    <extension>
      <groupId>kr.motd.maven</groupId>
      <artifactId>os-maven-plugin</artifactId>
      <version>1.6.1</version>
    </extension>
  </extensions>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.7.0</version>
      <configuration>
        <source>${source.level}</source>
        <target>${target.level}</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.xolstice.maven.plugins</groupId>
      <artifactId>protobuf-maven-plugin</artifactId>
      <version>0.5.1</version>
      <configuration>
        <protocArtifact>com.google.protobuf:protoc:3.7.1:exe:${os.detected.classifier}
</protocArtifact>
        <outputDirectory>build/generated/source/proto/main/java</outputDirectory>
        <clearOutputDirectory>false</clearOutputDirectory>
        <protocPlugins>
          <protocPlugin>
            <id>dubbo</id>
            <groupId>org.apache.dubbo</groupId>
            <artifactId>dubbo-compiler</artifactId>
            <version>${dubbo.compiler.version}</version>
            <mainClass>org.apache.dubbo.gen.dubbo.Dubbo3Generator</mainClass>
          </protocPlugin>
        </protocPlugins>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>test-compile</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
```

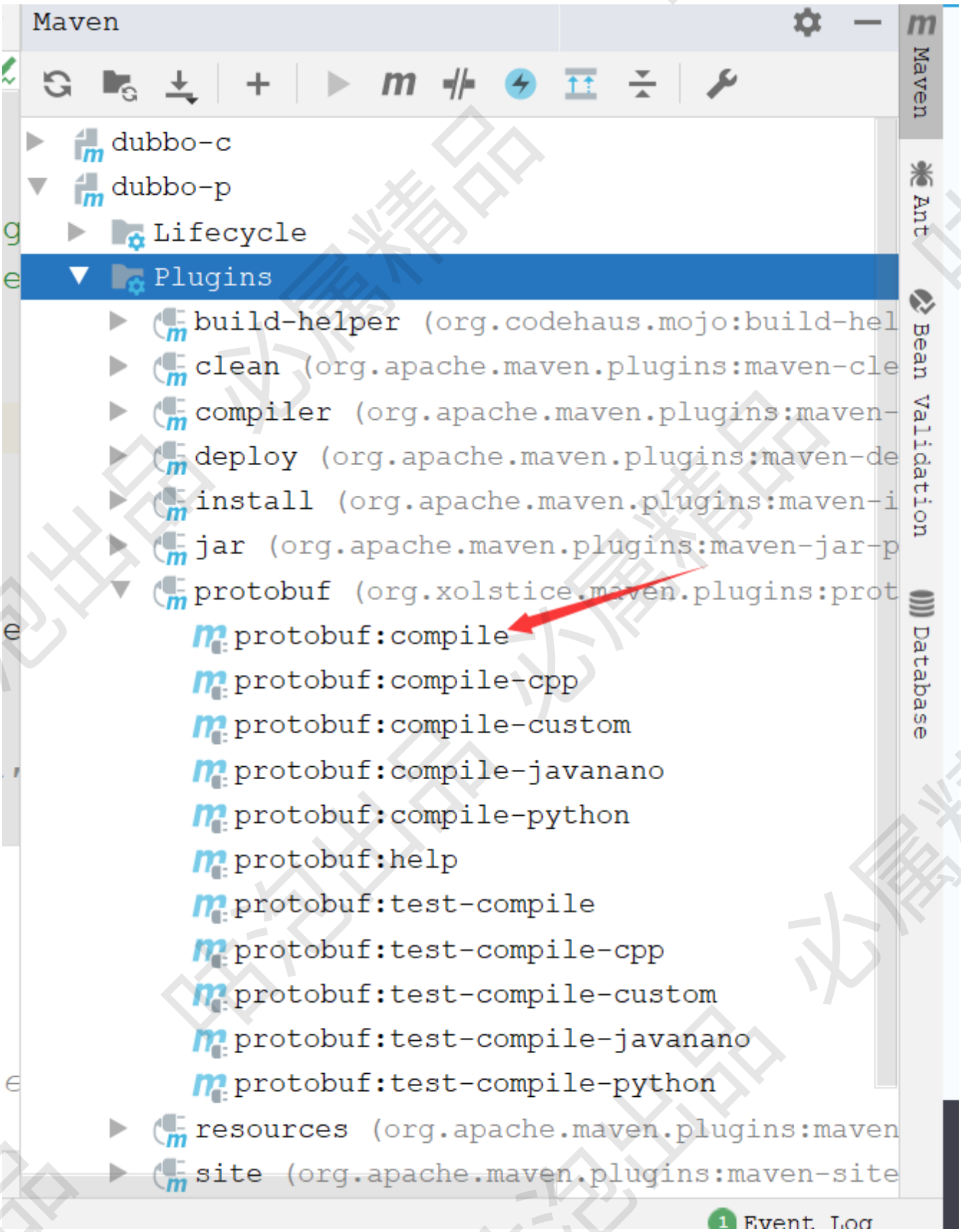
```
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>build-helper-maven-plugin</artifactId>
      <executions>
        <execution>
          <phase>generate-sources</phase>
          <goals>
            <goal>add-source</goal>
          </goals>
          <configuration>
            <sources>
              <source>build/generated/source/proto/main/java</source>
            </sources>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

2、定义proto文件

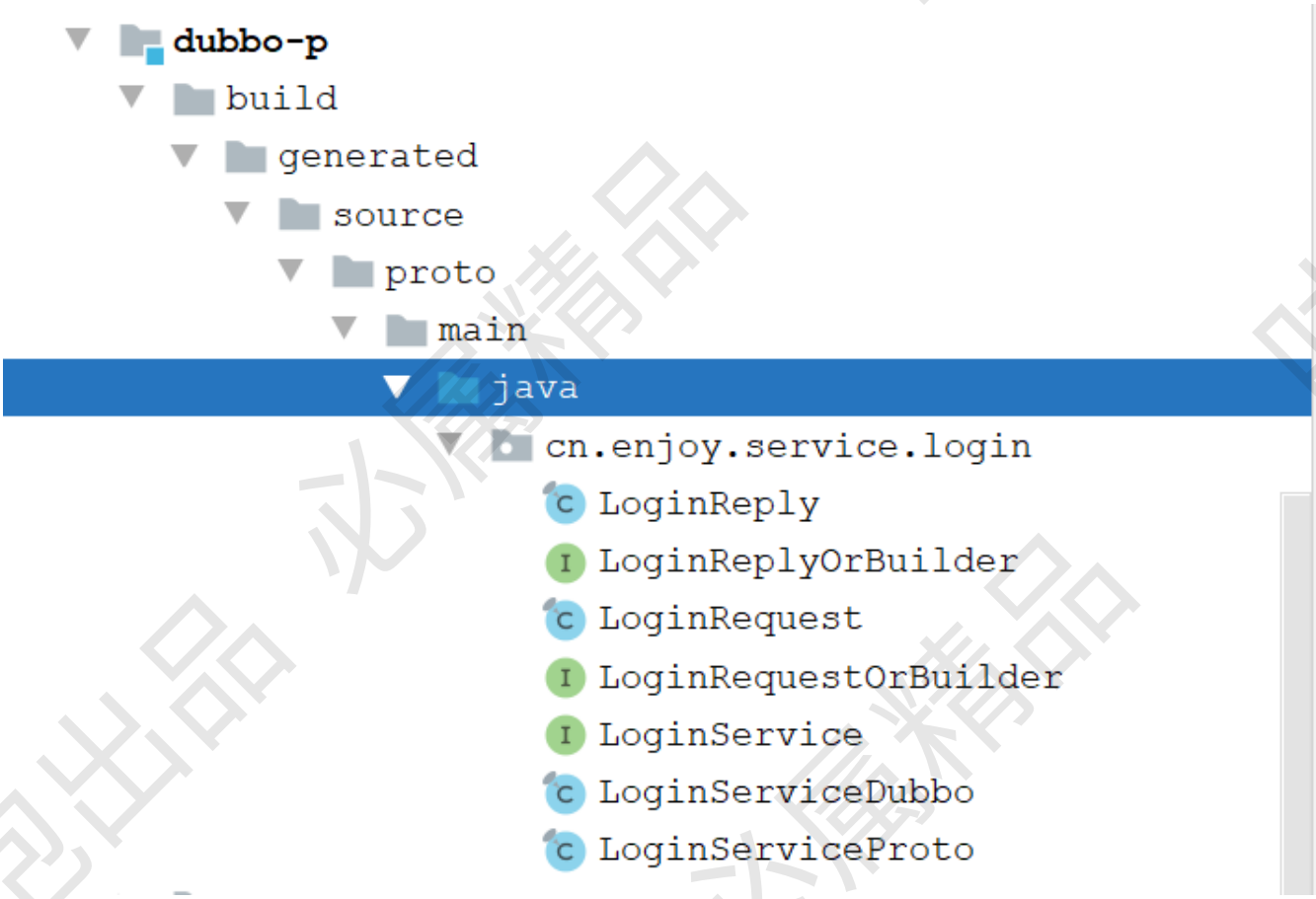
在dubbo-p工程的src/main下面创建proto文件夹在里面定义LoginService.proto文件，文件内容如下：

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "cn.enjoy.service.login";
option java_outer_classname = "LoginServiceProto";
option objc_class_prefix = "DEMOSRV";

package loginservice;

// The demo service definition.
service LoginService {
  rpc login (LoginRequest) returns (LoginReply) {}
}

// The request message containing the user's name.
message LoginRequest {
   string username = 1;
   string password = 2;
}

// The response message containing the greetings
message LoginReply {
  string message = 1;
}
```

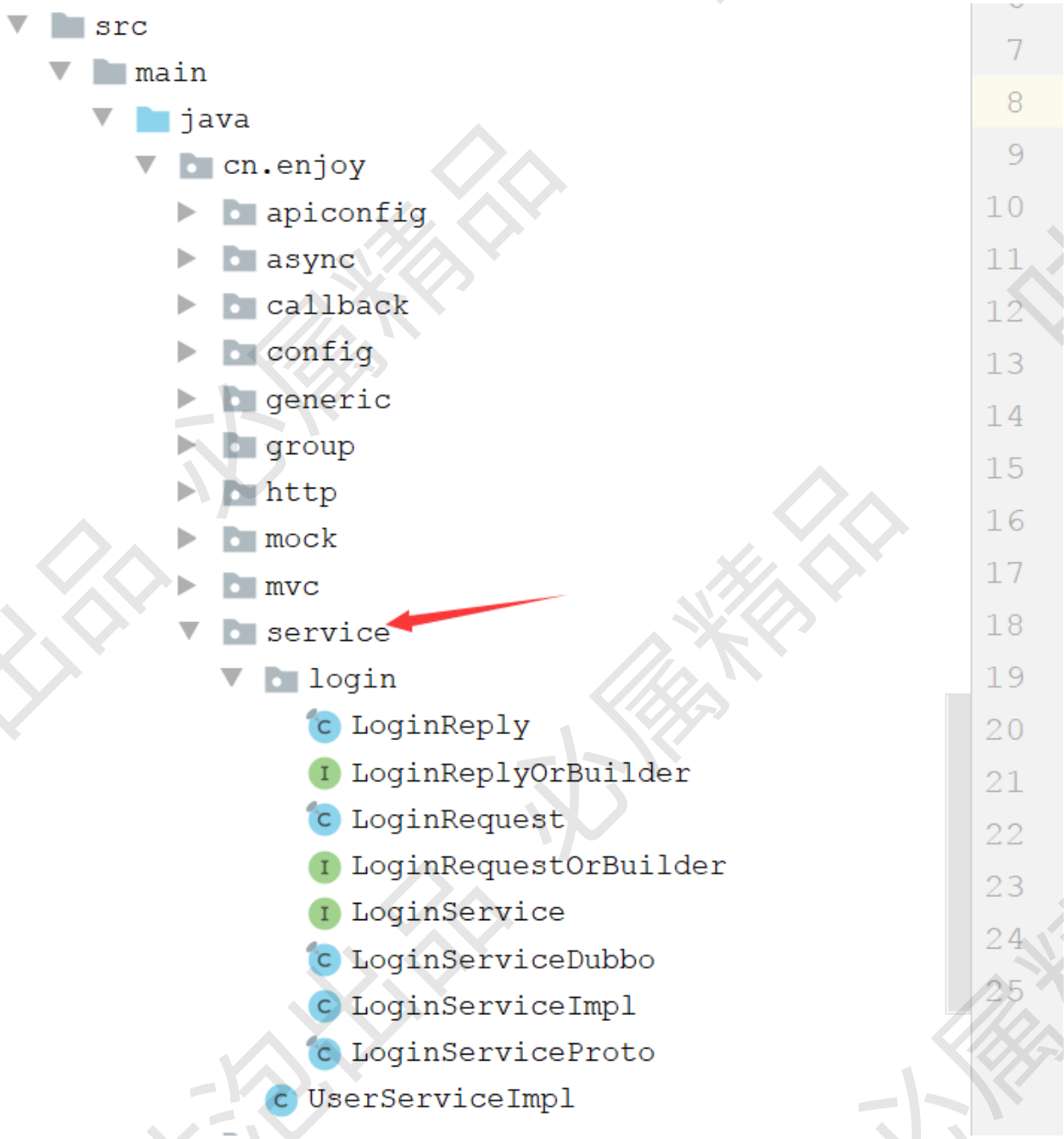
3、mvc插件根据proto文件生成service和bean，如图


Maven
dubbo-c
dubbo-p
Lifecycle
Plugins
build-helper (org.codehaus.mojo:build-hel
clean (org.apache.maven.plugins:maven-cle
compiler (org.apache.maven.plugins:maven-
deploy (org.apache.maven.plugins:maven-de
install (org.apache.maven.plugins:maven-i
jar (org.apache.maven.plugins:maven-jar-p
protobuf (org.xolstice.maven.plugins:prot
protobuf:compile
protobuf:compile-cpp
protobuf:compile-custom
protobuf:compile-javanano
protobuf:compile-python
protobuf:help
protobuf:test-compile
protobuf:test-compile-cpp
protobuf:test-compile-custom
protobuf:test-compile-javanano
protobuf:test-compile-python
resources (org.apache.maven.plugins:maven
site (org.apache.maven.plugins:maven-site
Maven
Ant
Bean Validation
Database
Event Log

生成文件如下：



4、把生成的文件移到响应的service目录下



生产者代码:

```
@DubboService
@Slf4j
public class LoginServiceImpl implements LoginService {
    @Override
    public LoginReply login(LoginRequest request) {
        log.info("Hello " + request.getUsername() + ", request from consumer: " +
RpcContext.getContext().getRemoteAddress());
        return LoginReply.newBuilder()
                .setMessage("Hello " + request.getUsername() + ", response from provider: "
                        + RpcContext.getContext().getLocalAddress())
                .build();
    }

    @Override
```

```
    public CompletableFuture<LoginReply> loginAsync(LoginRequest request) {
        return null;
    }
}
```

生产者配置：

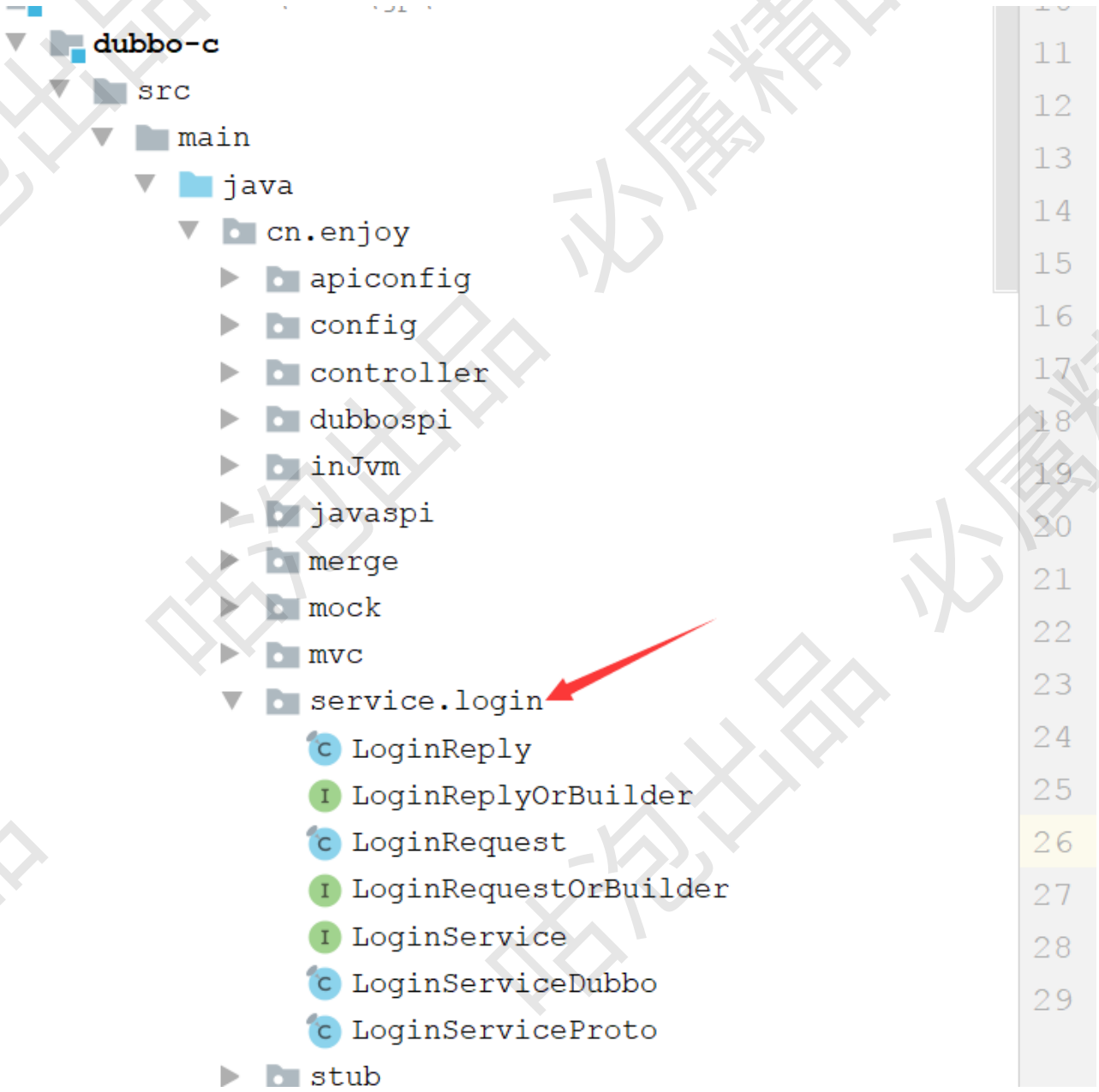
必须要加上序列化的配置属性，在dubbo-provider.properties中配置

```
dubbo.service.cn.enjoy.service.login.LoginService.serialization=protobuf
```

消费者：

消费者的1,2，3,4步骤是相同的

消费者代码：



```
@Test
public void protobuf() throws IOException {
    LoginRequest request =
LoginRequest.newBuilder().setUsername("wuya").setPassword("123").build();
    LoginReply reply = loginService.login(request);
    System.out.println(reply.getMessage());
}
```

主机绑定

在 Dubbo 中绑定主机名

缺省主机 IP 查找顺序：

- 通过 `LocalHost.getLocalHost()` 获取本机地址。
- 如果是 `127.*` 等 loopback 地址，则扫描各网卡，获取网卡 IP。

主机配置

注册的地址如果获取不正确，比如需要注册公网地址，可以：

1、可以在 `/etc/hosts` 中加入：机器名 公网 IP，比如：

```
test1 205.182.23.201
```

2、在 `dubbo.xml` 中加入主机地址的配置：

```
<dubbo:protocol host="205.182.23.201">
```

3、或在 `dubbo.properties` 中加入主机地址的配置：

```
dubbo.protocol.host=205.182.23.201
```

端口配置

缺省主机端口与协议相关：

协议	端口
dubbo	20880
rmi	1099
http	80
hessian	80
webservice	80
memcached	11211
redis	6379

可以按照下面的方式配置端口：

1、在 `dubbo.xml` 中加入主机地址的配置：

```
<dubbo:protocol name="dubbo" port="20880">
```

2、或在 `dubbo.properties` 中加入主机地址的配置：

```
dubbo.protocol.dubbo.port=20880
```

主机配置

自定义 Dubbo 服务对外暴露的主机地址

背景

在 Dubbo 中， Provider 启动时主要做两个事情，一是启动 server，二是向注册中心注册服务。启动 server 时需要绑定 socket，向注册中心注册服务时也需要发送 socket 唯一标识服务地址。

1. `dubbo` 中不设置 `host` 时默认 `host` 是什么?
2. 那在 `dubbo` 中如何指定服务的 `host` ,我们是否可以用hostname或domain代替IP地址作为 `host` ?
3. 在使用docker时,有时需要设置端口映射,此时,启动server时绑定的socket和向注册中心注册的socket使用不同的端口号,此时又该如何设置?

dubbo 中不设置 host 时默认 host 是什么

一般的 dubbo 协议配置如下:

```
...
<dubbo:protocol name="dubbo" port="20890" />
...
```

可以看到,只配置了端口号,没有配置 host，此时设置的 host 又是什么呢?

查看代码发现,在 `org.apache.dubbo.config.ServiceConfig#findConfigedHosts()` 中,通过 `InetAddress.getLocalHost().getHostAddress()` 获取默认 host。其返回值如下：

1. 未联网时，返回 127.0.0.1
2. 在阿里云服务器中，返回私有地址,如: 172.18.46.234
3. 在本机测试时，返回公有地址，如: 30.5.10.11

那在 dubbo 中如何指定服务的 socket?

除此之外,可以通过 `dubbo.protocol` 或 `dubbo.provider` 的 `host` 属性对 `host` 进行配置,支持IP地址和域名,如下:

```
...
<dubbo:protocol name="dubbo" port="20890" host="www.example.com"/>
...
```

在使用 docker 时，有时需要设置端口映射，此时，启动 server 时绑定的 socket 和向注册中心注册的 socket 使用不同的端口号，此时又该如何设置?

见 dubbo 通过环境变量设置 host

有些部署场景需要动态指定服务注册的地址，如 docker bridge 网络模式下要指定注册宿主机 ip 以实现外网通信。dubbo 提供了两对启动阶段的系统属性，用于设置对外通信的ip、port地址。

- DUBBO_IP_TO_REGISTRY — 注册到注册中心的ip地址
- DUBBO_PORT_TO_REGISTRY — 注册到注册中心的port端口
- DUBBO_IP_TO_BIND — 监听ip地址
- DUBBO_PORT_TO_BIND — 监听port端口

以上四个配置项均为可选项，如不配置 dubbo 会自动获取 ip 与端口，请根据具体的部署场景灵活选择配置。
dubbo 支持多协议，如果一个应用同时暴露多个不同协议服务，且需要为每个服务单独指定 ip 或 port，请分别在以上属性前加协议前缀。 如：

- HESSIAN_DUBBO_PORT_TO_BIND hessian协议绑定的port
- DUBBO_DUBBO_PORT_TO_BIND dubbo协议绑定的port
- HESSIAN_DUBBO_IP_TO_REGISTRY hessian协议注册的ip
- DUBBO_DUBBO_PORT_TO_BIND dubbo协议注册的ip

PORT_TO_REGISTRY 或 IP_TO_REGISTRY 不会用作默认 PORT_TO_BIND 或 IP_TO_BIND，但是反过来是成立的 如设置 PORT_TO_REGISTRY=20881 IP_TO_REGISTRY=30.5.97.6，则 PORT_TO_BIND IP_TO_BIND 不受影响 如果设置 PORT_TO_BIND=20881 IP_TO_BIND=30.5.97.6，则默认 PORT_TO_REGISTRY=20881 IP_TO_REGISTRY=30.5.97.6

总结

1. 可以通过 `dubbo.protocol` 或 `dubbo.provider` 的 `host` 属性对 `host` 进行配置,支持IP地址和域名.但此时注册到注册中心的IP地址和监听IP地址是同一个值
2. 为了解决在虚拟环境或局域网内consumer无法与provider通信的问题,可以通过环境变量分别设置注册到注册中心的IP地址和监听IP地址,其优先级高于 `dubbo.protocol` 或 `dubbo.provider` 的 `host` 配置

注册信息简化

减少注册中心上服务的注册数据

背景

Dubbo provider 中的服务配置项有接近 30 个配置项。 排除注册中心服务治理需要之外，很大一部分配置项是 provider 自己使用，不需要透传给消费者。这部分数据不需要进入注册中心，而只需要以 key-value 形式持久化存储。

Dubbo consumer 中的配置项也有 20+个配置项。在注册中心之中，服务消费者列表中只需要关注 application，version，group，ip，dubbo 版本等少量配置，其他配置也可以以 key-value 形式持久化存储。

这些数据是以服务为维度注册进入注册中心，导致了数据量的膨胀，进而引发注册中心(如 zookeeper)的网络开销增大，性能降低。

现有功能 sample

当前现状一个简单展示。通过这个展示，分析下为什么需要做简化配置。

参考 sample 子工程： dubbo-samples-simplified-registry/dubbo-samples-simplified-registry-nosimple （跑 sample 前，先跑下 ZKClean 进行配置项清理）

dubbo-provider.xml配置

```
<dubbo:application name="simplified-registry-nosimple-provider"/>
<dubbo:registry address="zookeeper://127.0.0.1:2181"/>
<bean id="demoService"
class="org.apache.dubbo.samples.simplified.registry.nosimple.impl.DemoServiceImpl"/>
<dubbo:service async="true"
interface="org.apache.dubbo.samples.simplified.registry.nosimple.api.DemoService"
              version="1.2.3" group="dubbo-simple" ref="demoService"
              executes="4500" retries="7" owner="vict" timeout="5300"/>
```

启动 provider 的 main 方法之后，查看 zookeeper 的叶子节点（路径为：/dubbo/org.apache.dubbo.samples.simplified.registry.nosimple.api.DemoService/providers 目录下）的内容如下：

```
dubbo%3A%2F%2F30.5.124.158%3A20880%2Forg.apache.dubbo.samples.simplified.registry.nosimple.a
pi.DemoService
%3Fanyhost%3Dtrue%26application%3Dsimplified-registry-xml-provider%26async%3Dtrue%26dubbo%3D
2.0.2%26**executes**%3D4500%26generic%3Dfalse%26group%3Ddubbo-simple%26interface%3D
org.apache.dubbo.samples.simplified.registry.nosimple.api.DemoService%26methods%3D
sayHello%26**owner**%3Dvict%26pid%3D2767%26**retries**%3D7%26revision%3D1.2.3%26side%3D
provider%26**timeout**%3D5300%26timestamp%3D1542361152795%26valid%3Dtrue%26version%3D1.2.3
```

从加粗字体中能看到有：executes, retries, owner, timeout。但是这些字段不是每个都需要传递给 dubbo ops 或者 dubbo consumer。 同样的，consumer 也有这个问题，可以在例子中启动 Consumer 的 main 方法进行查看。

设计目标和宗旨

期望简化进入注册中心的 provider 和 consumer 配置数量。 期望将部分配置项以其他形式存储。这些配置项需要满足：不在服务调用链路上，同时这些配置项不在注册中心的核心链路上(服务查询，服务列表)。

配置

简化注册中心的配置，只在 2.7 之后的版本中进行支持。 开启 provider 或者 consumer 简化配置之后，默认保留的配置项如下：

provider：

Constant Key	Key	remark
APPLICATION_KEY	application	
CODEC_KEY	codec	
EXCHANGER_KEY	exchanger	
SERIALIZATION_KEY	serialization	
CLUSTER_KEY	cluster	
CONNECTIONS_KEY	connections	
DEPRECATED_KEY	deprecated	
GROUP_KEY	group	
LOADBALANCE_KEY	loadbalance	
MOCK_KEY	mock	
PATH_KEY	path	
TIMEOUT_KEY	timeout	
TOKEN_KEY	token	
VERSION_KEY	version	
WARMUP_KEY	warmup	
WEIGHT_KEY	weight	
TIMESTAMP_KEY	timestamp	
DUBBO_VERSION_KEY	dubbo	
SPECIFICATION_VERSION_KEY	**specVersion**	新增，用于表述dubbo版本，如2.7.0

consumer：

Constant Key	Key	remark
APPLICATION_KEY	application	
VERSION_KEY	version	
GROUP_KEY	group	
DUBBO_VERSION_KEY	dubbo	
SPECIFICATION_VERSION_KEY	**specVersion**	新增，用于表述dubbo版本，如2.7.0

Constant Key 表示来自于类 org.apache.dubbo.common.Constants 的字段。

下面介绍几种常用的使用方式。所有的 sample，都可以查看sample-2.7

方式1. 配置dubbo.properties

sample 在 dubbo-samples-simplified-registry/dubbo-samples-simplified-registry-xml 工程下 （跑 sample 前，先跑下ZKClean 进行配置项清理）

dubbo.properties

```
dubbo.registry.simplified=true
dubbo.registry.extra-keys=retries,owner
```

怎么去验证呢?

provider端验证

provider端配置

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">
    <!-- optional -->
    <dubbo:application name="simplified-registry-xml-provider"/>
    <dubbo:registry address="zookeeper://127.0.0.1:2181"/>
    <bean id="demoService"
class="org.apache.dubbo.samples.simplified.registry.nosimple.impl.DemoServiceImpl"/>
    <dubbo:service async="true"
interface="org.apache.dubbo.samples.simplified.registry.nosimple.api.DemoService"
version="1.2.3" group="dubbo-simple"
                   ref="demoService" executes="4500" retries="7" owner="vict"
timeout="5300"/>

</beans>
```

得到的 zookeeper 的叶子节点的值如下:

```
dubbo%3A%2F%2F30.5.124.149%3A20880%2Forg.apache.dubbo.samples.simplified.registry.nosimple.a
pi.DemoService%3F
application%3Dsimplified-registry-xml-provider%26dubbo%3D2.0.2%26group%3Ddubbo-
simple%26**owner**%3D
vict%26**retries**%3D7%26**timeout**%3D5300%26timestamp%3D1542594503305%26version%3D1.2.3
```

和上面的**现有功能 sample**进行对比，上面的 sample 中，executes, retries, owner, timeout 四个配置项都进入了注册中心。但是本实例不是:

- 配置了：dubbo.registry.simplified=true， 默认情况下，timeout 在默认的配置项列表，所以还是会进入注册中心;
- 配置了：dubbo.registry.extra-keys=retries,owner ， 所以 retries，owner 也会进入注册中心。

总结：timeout，retries,owner 进入了注册中心，而 executes 没有进入。

consumer 端配置

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

    <!-- optional -->
    <dubbo:application name="simplified-registry-xml-consumer"/>

    <dubbo:registry address="zookeeper://127.0.0.1:2181" username="xxx" password="yyy"
check="true"/>

    <dubbo:reference id="demoService"
interface="org.apache.dubbo.samples.simplified.registry.nosimple.api.DemoService"
                     owner="vvv" retries="4" actives="6" timeout="4500" version="1.2.3"
group="dubbo-simple"/>

</beans>
```

得到的 zookeeper 的叶子节点的值如下：

```
consumer%3A%2F%2F30.5.124.149%2Forg.apache.dubbo.samples.simplified.registry.nosimple.api.De
moService%3F
actives%3D6%26application%3Dsimplified-registry-xml-consumer%26category%3D
consumers%26check%3Dfalse%26dubbo%3D2.0.2%26group%3Ddubbo-
simple%26owner%3Dvvv%26version%3D1.2.3
```

- 配置了：dubbo.registry.simplified=true ， 默认情况下，application,version,group,dubbo 在默认的配置项列表，所以还是会进入注册中心；

方式2. 声明spring bean

sample在dubbo-samples-simplified-registry/dubbo-samples-simplified-registry-annotation 工程下 （跑 sample 前，先跑下ZKClean 进行配置项清理）

Provider配置

privide 端 bean 配置：

```
// 等同于dubbo.properties配置，用@Bean形式进行配置
@Bean
public RegistryConfig registryConfig() {
    RegistryConfig registryConfig = new RegistryConfig();
    registryConfig.setAddress("zookeeper://127.0.0.1:2181");
    registryConfig.setSimplified(true);
    registryConfig.setExtraKeys("retries,owner");
    return registryConfig;
}
```

```
// 暴露服务
@Service(version = "1.1.8", group = "d-test", executes = 4500, retries = 7, owner =
"victanno", timeout = 5300)
public class AnnotationServiceImpl implements AnnotationService {
    @Override
    public String sayHello(String name) {
        System.out.println("async provider received: " + name);
        return "annotation: hello, " + name;
    }
}
```

和上面 sample 中的 dubbo.properties 的效果是一致的。结果如下：

- 默认情况下，timeout 在默认的配置项列表，所以还是会进入注册中心；
- 配置了 retries,owner 作为额外的 key 进入注册中心， 所以 retries，owner 也会进入注册中心。

总结：timeout，retries,owner 进入了注册中心，而 executes 没有进入。

Consumer配置

consumer 端 bean 配置：

```
@Bean
public RegistryConfig registryConfig() {
    RegistryConfig registryConfig = new RegistryConfig();
    registryConfig.setAddress("zookeeper://127.0.0.1:2181");
    registryConfig.setSimplified(true);
    return registryConfig;
  }
```

消费服务：

```
@Component("annotationAction")
public class AnnotationAction {

    @Reference(version = "1.1.8", group = "d-test", owner = "vvvanno", retries = 4, actives
= 6, timeout = 4500)
    private AnnotationService annotationService;
    public String doSayHello(String name) {
        return annotationService.sayHello(name);
    }
}
```

和上面 sample 中 consumer 端的配置是一样的。结果如下：

- 默认情况下，application,version,group,dubbo 在默认的配置项列表，所以还是会进入注册中心。

注意：

如果一个应用中既有provider又有consumer，那么配置需要合并成：

```
@Bean
public RegistryConfig registryConfig() {
    RegistryConfig registryConfig = new RegistryConfig();
    registryConfig.setAddress("zookeeper://127.0.0.1:2181");
    registryConfig.setSimplified(true);
    //只对provider生效
    registryConfig.setExtraKeys("retries,owner");
    return registryConfig;
}
```

后续规划

本版本还保留了大量的配置项，接下来的版本中，会逐渐删除所有的配置项。