04-Dubbo特性详解-服务相关

1 启动时检查

Dubbo 缺省会在启动时检查依赖的服务是否可用，不可用时会抛出异常，阻止 Spring 初始化完成，以便上线时，能及早发现问题，默认 `check="true"`。

可以通过 `check="false"` 关闭检查，比如，测试时，有些服务不关心，或者出现了循环依赖，必须有一方先启动。

另外，如果你的 Spring 容器是懒加载的，或者通过 API 编程延迟引用服务，请关闭 check，否则服务临时不可用时，会抛出异常，拿到 null 引用，如果 `check="false"`，总是会返回引用，当服务恢复时，能自动连上。

示例

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)：

```
<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.reference.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.reference.check=false
java -Ddubbo.consumer.check=false
java -Ddubbo.registry.check=false
```

配置的含义

`dubbo.reference.check=false`，强制改变所有 reference 的 check 值，就算配置中有声明，也会被覆盖。

`dubbo.consumer.check=false`，是设置 check 的缺省值，如果配置中有显式的声明，如：`<dubbo:reference check="true"/>`，不会受影响。

`dubbo.registry.check=false`，前面两个都是指订阅成功，但提供者列表是否为空是否报错，如果注册订阅失败时，也允许启动，需使用此选项，将在后台定时重试。

2 多版本

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

老版本服务提供者配置：

```
<dubbo:service interface="com.foo.BarService" version="1.0.0" />
```

新版本服务提供者配置：

```
<dubbo:service interface="com.foo.BarService" version="2.0.0" />
```

老版本服务消费者配置：

```
<dubbo:reference id="barService" interface="com.foo.BarService"
version="1.0.0" />
```

新版本服务消费者配置：

```
<dubbo:reference id="barService" interface="com.foo.BarService"
version="2.0.0" />
```

如果不需要区分版本，可以按照以下的方式配置 [1]：

```
<dubbo:reference id="barService" interface="com.foo.BarService"
version="*" />
```

---

1. `2.2.0` 以上版本支持 ↩

3 服务分组

当一个接口有多种实现时，可以用 group 区分。

服务

```
<dubbo:service group="feedback" interface="com.xxx.IndexService" />
<dubbo:service group="member" interface="com.xxx.IndexService" />
```

引用

```
<dubbo:reference id="feedbackIndexService" group="feedback"
interface="com.xxx.IndexService" />
<dubbo:reference id="memberIndexService" group="member"
interface="com.xxx.IndexService" />
```

任意组 [1]：

```
<dubbo:reference id="barService" interface="com.foo.BarService" group="*"
/>
```

---

1. `2.2.0` 以上版本支持，总是只调一个可用组的实现 ↩

4 分组聚合

按组合并返回结果，比如菜单服务，接口一样，但有多种实现，用group区分，现在消费方需从每种group中调用一次返回结果，合并结果返回，这样就可以实现聚合菜单项。

相关代码可以参考 dubbo 项目中的示例

配置

搜索所有分组

```
<dubbo:reference interface="com.xxx.MenuService" group="*" merger="true"
/>
```

合并指定分组

```
<dubbo:reference interface="com.xxx.MenuService" group="aaa,bbb"
merger="true" />
```

指定方法合并结果，其它未指定的方法，将只调用一个 Group

```
<dubbo:reference interface="com.xxx.MenuService" group="*">
    <dubbo:method name="getMenuItems" merger="true" />
</dubbo:reference>
```

某个方法不合并结果，其它都合并结果

```
<dubbo:reference interface="com.xxx.MenuService" group="*" merger="true">
    <dubbo:method name="getMenuItems" merger="false" />
</dubbo:reference>
```

指定合并策略，缺省根据返回值类型自动匹配，如果同一类型有两个合并器时，需指定合并器的名称 [2]

```
<dubbo:reference interface="com.xxx.MenuService" group="*">
    <dubbo:method name="getMenuItems" merger="mymerge" />
</dubbo:reference>
```

指定合并方法，将调用返回结果的指定方法进行合并，合并方法的参数类型必须是返回结果类型本身

```
<dubbo:reference interface="com.xxx.MenuService" group="*">
    <dubbo:method name="getMenuItems" merger=".addAll" />
</dubbo:reference>
```

---

1. 从 `2.1.0` 版本开始支持 ↩
2. 参见：合并结果扩展 ↩

合并结果扩展

扩展说明

合并返回结果，用于分组聚合。

扩展接口

```
org.apache.dubbo.rpc.cluster.Merger
```

扩展配置

```
<dubbo:method merger="xxx" />
```

已知扩展

- `org.apache.dubbo.rpc.cluster.merger.ArrayMerger`
- `org.apache.dubbo.rpc.cluster.merger.ListMerger`
- `org.apache.dubbo.rpc.cluster.merger.SetMerger`
- `org.apache.dubbo.rpc.cluster.merger.MapMerger`

扩展示例

Maven 项目结构：

```
src
 |-main
    |-java
        |-com
            |-xxx
                |-XxxMerger.java (实现Merger接口)
    |-resources
        |-META-INF
            |-dubbo
                |-org.apache.dubbo.rpc.cluster.Merger (纯文本文件，内容为：
xxx=com.xxx.XxxMerger)
```

XxxMerger.java：

```
package com.xxx;

import org.apache.dubbo.rpc.cluster.Merger;

public class XxxMerger<T> implements Merger<T> {
    public T merge(T... results) {
        // ...
    }
}
```

META-INF/dubbo/org.apache.dubbo.rpc.cluster.Merger：

```
xxx=com.xxx.XxxMerger
```

5 参数验证

参数验证功能 [1] 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 [2]。

Maven 依赖

```xml
<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>
```

示例

参数标注示例

```java
import java.io.Serializable;
import java.util.Date;

import javax.validation.constraints.Future;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;

public class ValidationParameter implements Serializable {
    private static final long serialVersionUID = 7158911668568000392L;

    @NotNull // 不允许为空
    @Size(min = 1, max = 20) // 长度或大小范围
    private String name;

    @NotNull(groups = ValidationService.Save.class) // 保存时不允许为空，更新时允许为空 ，表示不更新该字段
    @Pattern(regexp = "^\\s*\\w+(?:\\.{0,1}[\\w-]+)*@[a-zA-Z0-9]+(?:[-.][a-zA-Z0-9]+)*\\.[a-zA-Z]+\\s*$")
    private String email;

    @Min(18) // 最小值
    @Max(100) // 最大值
```

```
    private int age;

    @Past // 必须为一个过去的时间
    private Date loginDate;

    @Future // 必须为一个未来的时间
    private Date expiryDate;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public Date getLoginDate() {
        return loginDate;
    }

    public void setLoginDate(Date loginDate) {
        this.loginDate = loginDate;
    }

    public Date getExpiryDate() {
        return expiryDate;
```

```
    }

    public void setExpiryDate(Date expiryDate) {
        this.expiryDate = expiryDate;
    }
}
```

分组验证示例

```
public interface ValidationService { // 缺省可按服务接口区分验证场景，如：
@NotNull(groups = ValidationService.class)
    @interface Save{} // 与方法同名接口，首字母大写，用于区分验证场景，如：
@NotNull(groups = ValidationService.Save.class)，可选
    void save(ValidationParameter parameter);
    void update(ValidationParameter parameter);
}
```

关联验证示例

```
import javax.validation.GroupSequence;

public interface ValidationService {
    @GroupSequence(Update.class) // 同时验证Update组规则
    @interface Save{}
    void save(ValidationParameter parameter);

    @interface Update{}
    void update(ValidationParameter parameter);
}
```

参数验证示例

```
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;

public interface ValidationService {
    void save(@NotNull ValidationParameter parameter); // 验证参数不为空
    void delete(@Min(1) int id); // 直接对基本类型参数验证
}
```

配置

在客户端验证参数

```
<dubbo:reference id="validationService"
interface="org.apache.dubbo.examples.validation.api.ValidationService"
validation="true" />
```

在服务器端验证参数

```
<dubbo:service
interface="org.apache.dubbo.examples.validation.api.ValidationService"
ref="validationService" validation="true" />
```

验证异常信息

```
import javax.validation.ConstraintViolationException;
import javax.validation.ConstraintViolationException;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.apache.dubbo.examples.validation.api.ValidationParameter;
import org.apache.dubbo.examples.validation.api.ValidationService;
import org.apache.dubbo.rpc.RpcException;

public class ValidationConsumer {
    public static void main(String[] args) throws Exception {
        String config =
ValidationConsumer.class.getPackage().getName().replace('.', '/') +
"/validation-consumer.xml";
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext(config);
        context.start();
        ValidationService validationService =
(ValidationService)context.getBean("validationService");
        // Error
        try {
            parameter = new ValidationParameter();
            validationService.save(parameter);
            System.out.println("Validation ERROR");
```

```
        } catch (RpcException e) { // 抛出的是RpcException
            ConstraintViolationException ve =
(ConstraintViolationException) e.getCause(); // 里面嵌了一个
ConstraintViolationException
            Set<ConstraintViolation<?>> violations =
ve.getConstraintViolations(); // 可以拿到一个验证错误详细信息的集合
            System.out.println(violations);
        }
    }
}
```

---

1. 自 `2.1.0` 版本开始支持, 如何使用可以参考 dubbo 项目中的示例代码 ↩
2. 验证方式可扩展，扩展方式参见开发者手册中的验证扩展 ↩

6 本地伪装【掌握】

本地伪装 [1] 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。

在 spring 配置文件中按以下方式配置：

```
<dubbo:reference interface="com.foo.BarService" mock="true" />
```

或

```
<dubbo:reference interface="com.foo.BarService"
mock="com.foo.BarServiceMock" />
```

在工程中提供 Mock 实现 [2]：

```
package com.foo;
public class BarServiceMock implements BarService {
    public String sayHello(String name) {
        // 你可以伪造容错数据，此方法只在出现RpcException时被执行
        return "容错数据";
    }
}
```

如果服务的消费方经常需要 try-catch 捕获异常，如：

```
Offer offer = null;
try {
    offer = offerService.findOffer(offerId);
} catch (RpcException e) {
   logger.error(e);
}
```

请考虑改为 Mock 实现，并在 Mock 实现中 return null。如果只是想简单的忽略异常，在 `2.0.11` 以上版本可用：

```
<dubbo:reference interface="com.foo.BarService" mock="return null" />
```

进阶用法

return

使用 `return` 来返回一个字符串表示的对象，作为 Mock 的返回值。合法的字符串可以是：

- *empty*: 代表空，基本类型的默认值，或者集合类的空值
- *null*: `null`
- *true*: `true`
- *false*: `false`
- *JSON 格式*: 反序列化 JSON 所得到的对象

throw

使用 `throw` 来返回一个 Exception 对象，作为 Mock 的返回值。

当调用出错时，抛出一个默认的 RPCException:

```
<dubbo:reference interface="com.foo.BarService" mock="throw" />
```

当调用出错时，抛出指定的 Exception：

```
<dubbo:reference interface="com.foo.BarService" mock="throw
com.foo.MockException" />
```

force 和 fail

在 `2.6.6` 以上的版本，可以开始在 Spring XML 配置文件中使用 `fail:` 和 `force:`。`force:` 代表强制使用 Mock 行为，在这种情况下不会走远程调用。`fail:` 与默认行为一致，只有当远程调用发生错误时才使用 Mock 行为。`force:` 和 `fail:` 都支持与 `throw` 或者 `return` 组合使用。

强制返回指定值：

```
<dubbo:reference interface="com.foo.BarService" mock="force:return fake"
/>
```

强制抛出指定异常：

```
<dubbo:reference interface="com.foo.BarService" mock="force:throw
com.foo.MockException" />
```

在方法级别配置 Mock

Mock 可以在方法级别上指定，假定 `com.foo.BarService` 上有好几个方法，我们可以单独为 `sayHello()` 方法指定 Mock 行为。具体配置如下所示，在本例中，只要 `sayHello()` 被调用到时，强制返回 "fake":

```
<dubbo:reference id="demoService" check="false"
interface="com.foo.BarService">
    <dubbo:parameter key="sayHello.mock" value="force:return fake"/>
</dubbo:reference>
```

---

1. Mock 是 Stub 的一个子集，便于服务提供方在客户端执行容错逻辑，因经常需要在出现 RpcException (比如网络失败，超时等)时进行容错，而在出现业务异常(比如登录用户名密码错误)时不需要容错，如果用 Stub，可能就需要捕获并依赖 RpcException 类，而用 Mock 就可以不依赖 RpcException，因为它的约定就是只有出现 RpcException 时才执行。 ↩
2. 在 interface 旁放一个 Mock 实现，它实现 BarService 接口，并有一个无参构造函数 ↩

7 服务降级【掌握】

可以通过服务降级功能 [1] 临时屏蔽某个出错的非关键服务，并定义降级后的返回策略。

向注册中心写入动态配置覆盖规则：

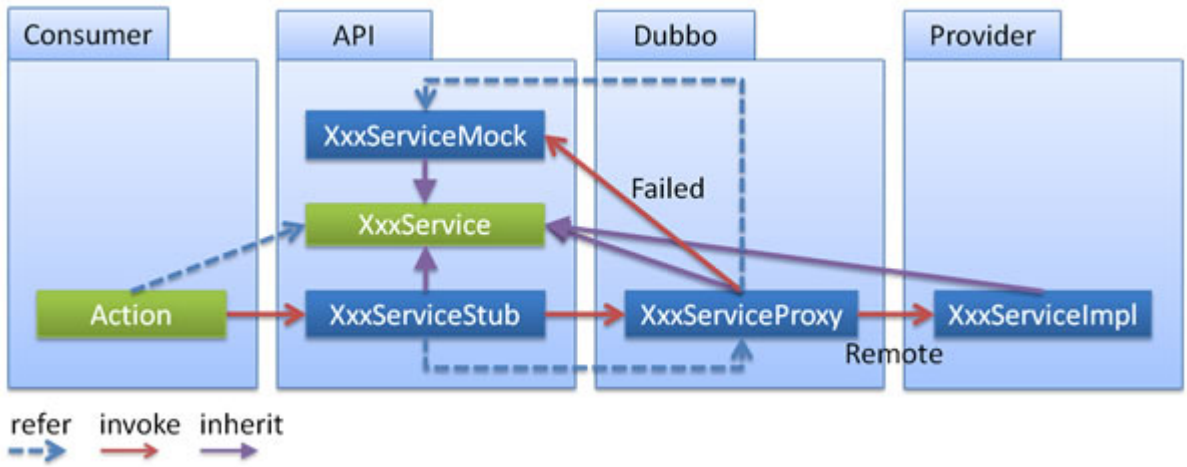
```
RegistryFactory registryFactory =
ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExten
sion();
Registry registry =
registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));
registry.register(URL.valueOf("override://0.0.0.0/com.foo.BarService?
category=configurators&dynamic=false&application=foo&mock=force:return+nul
l"));
```

其中：

- `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。

8 本地存根【掌握】

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。



在 spring 配置文件中按以下方式配置：

```
<dubbo:service interface="com.foo.BarService" stub="true" />
```

或

```
<dubbo:service interface="com.foo.BarService"
stub="com.foo.BarServiceStub" />
```

提供 Stub 的实现 [2]：

```
package com.foo;
public class BarServiceStub implements BarService {
    private final BarService barService;

    // 构造函数传入真正的远程代理对象
    public BarServiceStub(BarService barService){
        this.barService = barService;
    }

    public String sayHello(String name) {
        // 此代码在客户端执行，你可以在客户端做ThreadLocal本地缓存，或预先验证参数是否合法，等等
        try {
            return barService.sayHello(name);
        } catch (Exception e) {
            // 你可以容错，可以做任何AOP拦截事项
            return "容错数据";
        }
    }
}
```

---

1. Stub 必须有可传入 Proxy 的构造函数。 ↩
2. 在 interface 旁边放一个 Stub 实现，它实现 BarService 接口，并有一个传入远程 BarService 实例的构造函数 ↩

9 参数回调

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 [1]。可以参考 dubbo 项目中的示例代码。

服务接口示例

CallbackService.java

```
package com.callback;

public interface CallbackService {
    void addListener(String key, CallbackListener listener);
}
```

CallbackListener.java

```
package com.callback;

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

服务提供者接口实现示例

```
package com.callback.impl;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import com.callback.CallbackListener;
import com.callback.CallbackService;

public class CallbackServiceImpl implements CallbackService {

    private final Map<String, CallbackListener> listeners = new
ConcurrentHashMap<String, CallbackListener>();

    public CallbackServiceImpl() {
        Thread t = new Thread(new Runnable() {
            public void run() {
                while(true) {
                    try {
```

```
                        for(Map.Entry<String, CallbackListener> entry :
listeners.entrySet()){
                            try {

entry.getValue().changed(getChanged(entry.getKey()));
                            } catch (Throwable t) {
                                listeners.remove(entry.getKey());
                            }
                        }
                        Thread.sleep(5000); // 定时触发变更通知
                    } catch (Throwable t) { // 防御容错
                        t.printStackTrace();
                    }
                }
            }
        });
        t.setDaemon(true);
        t.start();
    }

    public void addListener(String key, CallbackListener listener) {
        listeners.put(key, listener);
        listener.changed(getChanged(key)); // 发送变更通知
    }

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

服务提供者配置示例

```
<bean id="callbackService" class="com.callback.impl.CallbackServiceImpl"
/>
<dubbo:service interface="com.callback.CallbackService"
ref="callbackService" connections="1" callbacks="1000">
    <dubbo:method name="addListener">
        <dubbo:argument index="1" callback="true" />
        <!--也可以通过指定类型的方式-->
        <!--<dubbo:argument type="com.demo.CallbackListener"
callback="true" />-->
    </dubbo:method>
</dubbo:service>
```

服务消费者配置示例

```
<dubbo:reference id="callbackService"
interface="com.callback.CallbackService" />
```

服务消费者调用示例

```
ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("classpath:consumer.xml");
context.start();

CallbackService callbackService = (CallbackService)
context.getBean("callbackService");

callbackService.addListener("foo.bar", new CallbackListener(){
    public void changed(String msg) {
        System.out.println("callback1:" + msg);
    }
});
```

---

1. `2.0.6` 及其以上版本支持 ↩

10 结果缓存

结果缓存 [1]，用于加速热门数据的访问速度，Dubbo 提供声明式缓存，以减少用户加缓存的工作量 [2]。

缓存类型

- `lru` 基于最近最少使用原则删除多余缓存，保持最热的数据被缓存。
- `threadlocal` 当前线程缓存，比如一个页面渲染，用到很多 portal，每个 portal 都要去查用户信息，通过线程缓存，可以减少这种多余访问。
- `jcache` 与 JSR107 集成，可以桥接各种缓存实现。

缓存类型可扩展，参见：缓存扩展

配置

```
<dubbo:reference interface="com.foo.BarService" cache="lru" />
```

或：

```
<dubbo:reference interface="com.foo.BarService">
    <dubbo:method name="findBar" cache="lru" />
</dubbo:reference>
```

---

1. `2.1.0` 以上版本支持 ↩

11 事件通知

在调用之前、调用之后、出现异常时，会触发 `oninvoke`、`onreturn`、`onthrow` 三个事件，可以配置当事件发生时，通知哪个类的哪个方法 [1]。

服务提供者与消费者共享服务接口

```
interface IDemoService {
    public Person get(int id);
}
```

服务提供者实现

```
class NormalDemoService implements IDemoService {
    public Person get(int id) {
        return new Person(id, "charles`son", 4);
    }
}
```

服务提供者配置

```
<dubbo:application name="rpc-callback-demo" />
<dubbo:registry address="zookeeper://127.0.0.1:2181"/>
<bean id="demoService"
class="org.apache.dubbo.callback.implicit.NormalDemoService" />
<dubbo:service interface="org.apache.dubbo.callback.implicit.IDemoService"
ref="demoService" version="1.0.0" group="cn"/>
```

服务消费者 Callback 接口

```
interface Notify {
    public void onreturn(Person msg, Integer id);
    public void onthrow(Throwable ex, Integer id);
}
```

服务消费者 Callback 实现

```
class NotifyImpl implements Notify {
    public Map<Integer, Person>     ret    = new HashMap<Integer, Person>
();
    public Map<Integer, Throwable> errors = new HashMap<Integer,
Throwable>();

    public void onreturn(Person msg, Integer id) {
        System.out.println("onreturn:" + msg);
        ret.put(id, msg);
    }

    public void onthrow(Throwable ex, Integer id) {
        errors.put(id, ex);
    }
}
```

服务消费者 Callback 配置

```
<bean id ="demoCallback" class =
"org.apache.dubbo.callback.implicit.NofifyImpl" />
<dubbo:reference id="demoService"
interface="org.apache.dubbo.callback.implicit.IDemoService"
version="1.0.0" group="cn" >
      <dubbo:method name="get" async="true" onreturn =
"demoCallback.onreturn" onthrow="demoCallback.onthrow" />
</dubbo:reference>
```

`callback` 与 `async` 功能正交分解，`async=true` 表示结果是否马上返回，`onreturn` 表示是否需要回调。

两者叠加存在以下几种组合情况 [2]：

- 异步回调模式：`async=true onreturn="xxx"`
- 同步回调模式：`async=false onreturn="xxx"`
- 异步无回调 ：`async=true`
- 同步无回调 ：`async=false`

测试代码

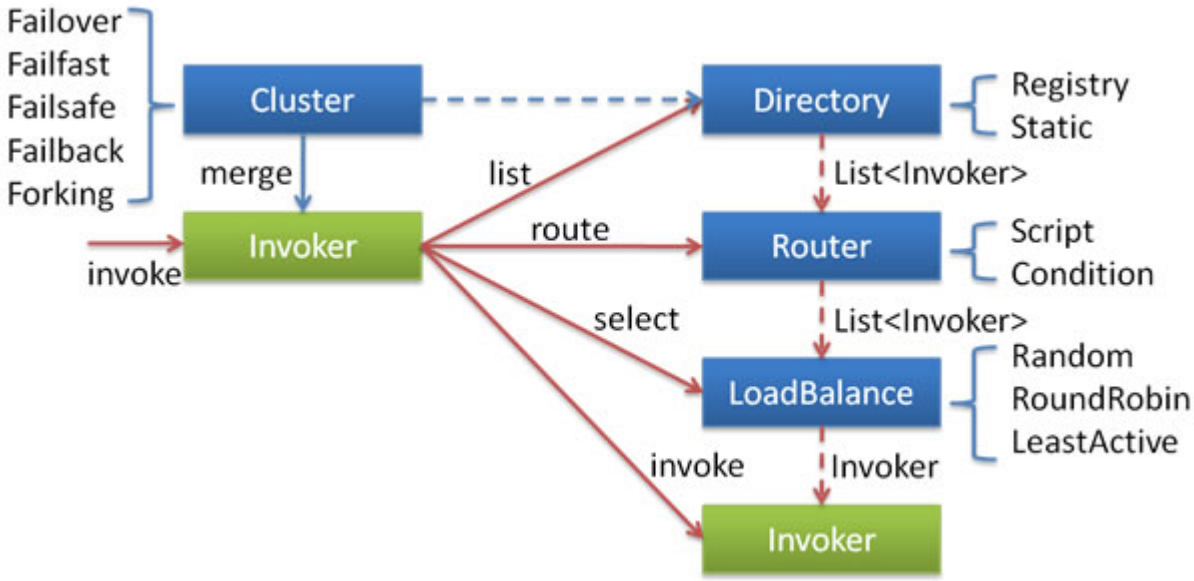
```
IDemoService demoService = (IDemoService) context.getBean("demoService");
NofifyImpl notify = (NofifyImpl) context.getBean("demoCallback");
int requestId = 2;
Person ret = demoService.get(requestId);
Assert.assertEquals(null, ret);
//for Test：只是用来说明callback正常被调用，业务具体实现自行决定.
for (int i = 0; i < 10; i++) {
    if (!notify.ret.containsKey(requestId)) {
        Thread.sleep(200);
    } else {
        break;
    }
}
Assert.assertEquals(requestId, notify.ret.get(requestId).getId());
```

---

1. 支持版本：`2.0.7` 之后 ↩
2. `async=false` 默认 ↩

12 负载均衡【掌握】

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。



可以自行扩展负载均衡策略，参见：负载均衡扩展

负载均衡策略

Random LoadBalance

- **随机**，按权重设置随机概率。
- 在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。

RoundRobin LoadBalance

- **轮询**，按公约后的权重设置轮询比率。
- 存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。

LeastActive LoadBalance

- **最少活跃调用数**，相同活跃数的随机，活跃数指调用前后计数差。
- 使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。

ConsistentHash LoadBalance

- **一致性 Hash**，相同参数的请求总是发到同一提供者。
- 当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。
- 算法参见：http://en.wikipedia.org/wiki/Consistent_hashing
- 缺省只对第一个参数 Hash，如果要修改，请配置 `<dubbo:parameter key="hash.arguments" value="0,1" />`
- 缺省用 160 份虚拟节点，如果要修改，请配置 `<dubbo:parameter key="hash.nodes" value="320" />`

配置

服务端服务级别

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

客户端服务级别

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

服务端方法级别

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

客户端方法级别

```
<dubbo:reference interface="...">
    <dubbo:method name="..." loadbalance="roundrobin"/>
</dubbo:reference>
```

负载均衡扩展

扩展说明

从多个服务提者方中选择一个进行调用

扩展接口

```
org.apache.dubbo.rpc.cluster.LoadBalance
```

扩展配置

```
<dubbo:protocol loadbalance="xxx" />
<!-- 缺省值设置，当<dubbo:protocol>没有配置loadbalance时，使用此配置 -->
<dubbo:provider loadbalance="xxx" />
```

已知扩展

- `org.apache.dubbo.rpc.cluster.loadbalance.RandomLoadBalance`
- `org.apache.dubbo.rpc.cluster.loadbalance.RoundRobinLoadBalance`
- `org.apache.dubbo.rpc.cluster.loadbalance.LeastActiveLoadBalance`

扩展示例

Maven 项目结构：

```
src
 |-main
    |-java
        |-com
            |-xxx
                |-XxxLoadBalance.java (实现LoadBalance接口)
    |-resources
        |-META-INF
            |-dubbo
                |-org.apache.dubbo.rpc.cluster.LoadBalance (纯文本文件，内容为：xxx=com.xxx.XxxLoadBalance)
```

XxxLoadBalance.java：

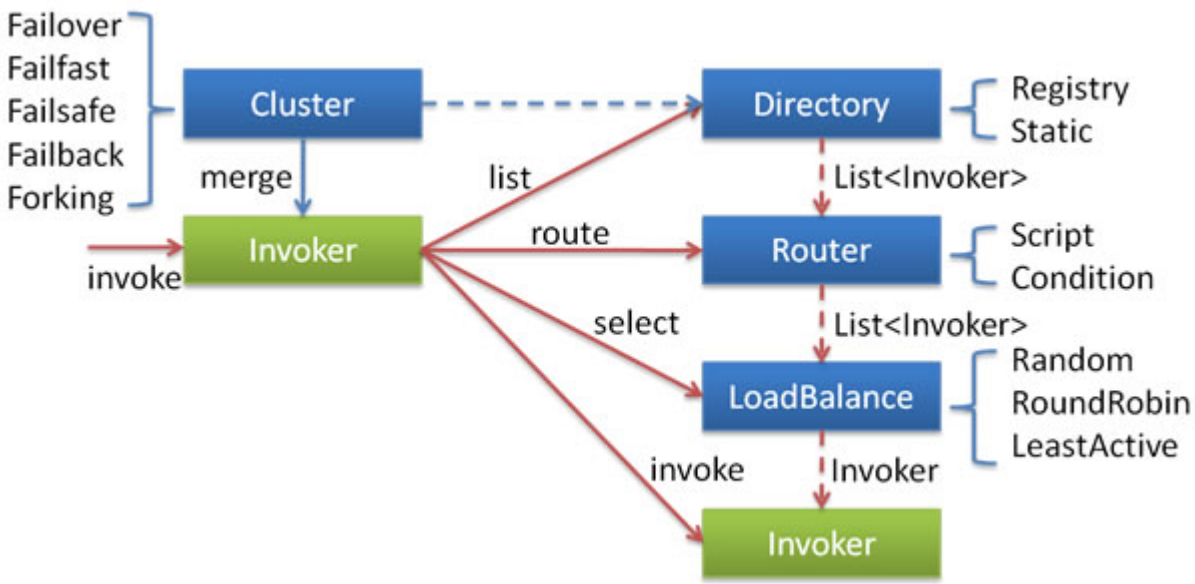
```
package com.xxx;

import org.apache.dubbo.rpc.cluster.LoadBalance;
import org.apache.dubbo.rpc.Invoker;
import org.apache.dubbo.rpc.Invocation;
import org.apache.dubbo.rpc.RpcException;

public class XxxLoadBalance implements LoadBalance {
    public <T> Invoker<T> select(List<Invoker<T>> invokers, Invocation
invocation) throws RpcException {
        // ...
    }
}
```

META-INF/dubbo/org.apache.dubbo.rpc.cluster.LoadBalance：

```
xxx=com.xxx.XxxLoadBalance
```

13 集群容错【掌握】

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 [1]。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)。

重试次数配置如下：

```
<dubbo:service retries="2" />
```

或

```
<dubbo:reference retries="2" />
```

或

```
<dubbo:reference>
    <dubbo:method name="findFoo" retries="2" />
</dubbo:reference>
```

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 [2]。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。

集群模式配置

按照以下示例在服务提供方和消费方配置集群模式

```
<dubbo:service cluster="failsafe" />
```

或

```
<dubbo:reference cluster="failsafe" />
```

---

1. 该配置为缺省配置 ↩
2. `2.1.0` 开始支持 ↩

集群扩展

扩展说明

当有多个服务提供方时，将多个服务提供方组织成一个集群，并伪装成一个提供方。

扩展接口

```
org.apache.dubbo.rpc.cluster.Cluster
```

扩展配置

```
<dubbo:protocol cluster="xxx" />
<!-- 缺省值配置，如果<dubbo:protocol>没有配置cluster时，使用此配置 -->
<dubbo:provider cluster="xxx" />
```

已知扩展

- `org.apache.dubbo.rpc.cluster.support.FailoverCluster`
- `org.apache.dubbo.rpc.cluster.support.FailfastCluster`
- `org.apache.dubbo.rpc.cluster.support.FailsafeCluster`
- `org.apache.dubbo.rpc.cluster.support.FailbackCluster`
- `org.apache.dubbo.rpc.cluster.support.ForkingCluster`
- `org.apache.dubbo.rpc.cluster.support.AvailableCluster`

扩展示例

Maven 项目结构：

```
src
 |-main
    |-java
        |-com
            |-xxx
                |-XxxCluster.java (实现Cluster接口)
    |-resources
        |-META-INF
            |-dubbo
                |-org.apache.dubbo.rpc.cluster.Cluster (纯文本文件，内容为：
xxx=com.xxx.XxxCluster)
```

XxxCluster.java：

```
package com.xxx;

import org.apache.dubbo.rpc.cluster.Cluster;
import org.apache.dubbo.rpc.cluster.support.AbstractClusterInvoker;
import org.apache.dubbo.rpc.cluster.Directory;
import org.apache.dubbo.rpc.cluster.LoadBalance;
import org.apache.dubbo.rpc.Invoker;
import org.apache.dubbo.rpc.Invocation;
import org.apache.dubbo.rpc.Result;
import org.apache.dubbo.rpc.RpcException;
```

```
public class XxxCluster implements Cluster {
    public <T> Invoker<T> merge(Directory<T> directory) throws
RpcException {
        return new AbstractClusterInvoker<T>(directory) {
            public Result doInvoke(Invocation invocation, List<Invoker<T>>
invokers, LoadBalance loadbalance) throws RpcException {
                // ...
            }
        };
    }
}
```

META-INF/dubbo/org.apache.dubbo.rpc.cluster.Cluster：

```
xxx=com.xxx.XxxCluster
```

14 令牌验证

通过令牌验证在注册中心控制权限，以决定要不要下发令牌给消费者，可以防止消费者绕过注册中心访问提供者，另外通过注册中心可灵活改变授权方式，而不需修改或升级提供者

可以全局设置开启令牌验证：

```
<!--随机token令牌，使用UUID生成-->
<dubbo:provider interface="com.foo.BarService" token="true" />
```

或

```
<!--固定token令牌，相当于密码-->
<dubbo:provider interface="com.foo.BarService" token="123456" />
```

也可在服务级别设置：

```
<!--随机token令牌，使用UUID生成-->
<dubbo:service interface="com.foo.BarService" token="true" />
```

或

```
<!--固定token令牌，相当于密码-->
<dubbo:service interface="com.foo.BarService" token="123456" />
```

还可在协议级别设置：

```
<!--随机token令牌，使用UUID生成-->
<dubbo:protocol name="dubbo" token="true" />
```

或

```
<!--固定token令牌，相当于密码-->
<dubbo:protocol name="dubbo" token="123456" />
```

15 使用泛化调用

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。

通过 Spring 使用泛化调用

在 Spring 配置申明 `generic="true"`：

```
<dubbo:reference id="barService" interface="com.foo.BarService"
generic="true" />
```

在 Java 代码获取 barService 并开始泛化调用：

```
GenericService barService = (GenericService)
applicationContext.getBean("barService");
Object result = barService.$invoke("sayHello", new String[] {
"java.lang.String" }, new Object[] { "World" });
```

通过 API 方式使用泛化调用

```
import org.apache.dubbo.rpc.service.GenericService;
...

// 引用远程服务
// 该实例很重量，里面封装了所有与注册中心及服务提供方连接，请缓存
ReferenceConfig<GenericService> reference = new
ReferenceConfig<GenericService>();
// 弱类型接口名
reference.setInterface("com.xxx.XxxService");
reference.setVersion("1.0.0");
// 声明为泛化接口
reference.setGeneric(true);

// 用org.apache.dubbo.rpc.service.GenericService可以替代所有接口引用
GenericService genericService = reference.get();

// 基本类型以及Date,List,Map等不需要转换，直接调用
Object result = genericService.$invoke("sayHello", new String[]
{"java.lang.String"}, new Object[] {"world"});

// 用Map表示POJO参数，如果返回值为POJO也将自动转成Map
Map<String, Object> person = new HashMap<String, Object>();
person.put("name", "xxx");
person.put("password", "yyy");
// 如果返回POJO将自动转成Map
Object result = genericService.$invoke("findPerson", new String[]
{"com.xxx.Person"}, new Object[]{person});

...
```

有关泛化类型的进一步解释

假设存在 POJO 如：

```
package com.xxx;

public class PersonImpl implements Person {
    private String name;
    private String password;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

则 POJO 数据：

```
Person person = new PersonImpl();
person.setName("xxx");
person.setPassword("yyy");
```

可用下面 Map 表示：

```
Map<String, Object> map = new HashMap<String, Object>();
// 注意：如果参数类型是接口，或者List等丢失泛型，可通过class属性指定类型。
map.put("class", "com.xxx.PersonImpl");
map.put("name", "xxx");
map.put("password", "yyy");
```

16 实现泛化调用

泛接口实现方式主要用于服务器端没有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：实现一个通用的远程服务Mock框架，可通过实现GenericService接口处理所有服务请求。

在 Java 代码中实现 `GenericService` 接口：

```
package com.foo;
public class MyGenericService implements GenericService {

    public Object $invoke(String methodName, String[] parameterTypes,
Object[] args) throws GenericException {
        if ("sayHello".equals(methodName)) {
            return "Welcome " + args[0];
        }
    }
}
```

通过 Spring 暴露泛化实现

在 Spring 配置申明服务的实现：

```
<bean id="genericService" class="com.foo.MyGenericService" />
<dubbo:service interface="com.foo.BarService" ref="genericService" />
```

通过 API 方式暴露泛化实现

```
...
// 用org.apache.dubbo.rpc.service.GenericService可以替代所有接口实现
GenericService xxxService = new XxxGenericService();

// 该实例很重量，里面封装了所有与注册中心及服务提供方连接，请缓存
ServiceConfig<GenericService> service = new ServiceConfig<GenericService>
();
// 弱类型接口名
service.setInterface("com.xxx.XxxService");
```

```
service.setVersion("1.0.0");
// 指向一个通用服务实现
service.setRef(xxxService);

// 暴露及注册服务
service.export();
```

17 上下文信息

上下文中存放的是当前调用过程中所需的环境信息。所有配置信息都将转换为 URL 的参数，参见 schema 配置参考手册 中的**对应URL参数**一列。

RpcContext 是一个 ThreadLocal 的临时状态记录器，当接收到 RPC 请求，或发起 RPC 请求时，RpcContext 的状态都会变化。比如：A 调 B，B 再调 C，则 B 机器上，在 B 调 C 之前，RpcContext 记录的是 A 调 B 的信息，在 B 调 C 之后，RpcContext 记录的是 B 调 C 的信息。

服务消费方

```
// 远程调用
xxxService.xxx();
// 本端是否为消费端，这里会返回true
boolean isConsumerSide = RpcContext.getContext().isConsumerSide();
// 获取最后一次调用的提供方IP地址
String serverIP = RpcContext.getContext().getRemoteHost();
// 获取当前服务配置信息，所有配置信息都将转换为URL的参数
String application =
RpcContext.getContext().getUrl().getParameter("application");
// 注意：每发起RPC调用，上下文状态会变化
yyyService.yyy();
```

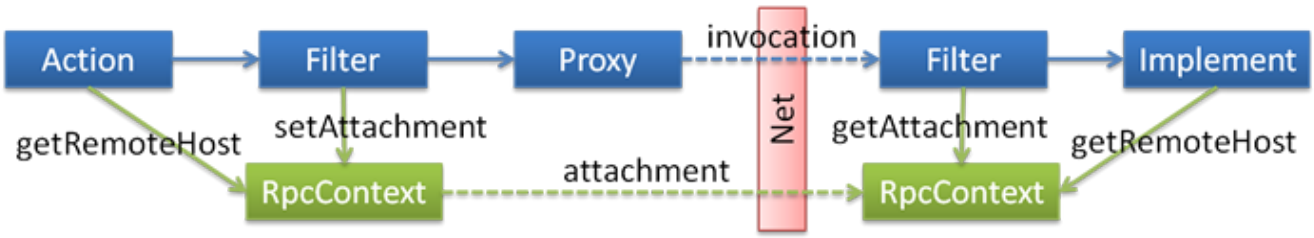
服务提供方

```
public class XxxServiceImpl implements XxxService {

    public void xxx() {
        // 本端是否为提供端，这里会返回true
        boolean isProviderSide = RpcContext.getContext().isProviderSide();
        // 获取调用方IP地址
        String clientIP = RpcContext.getContext().getRemoteHost();
```

```
        // 获取当前服务配置信息，所有配置信息都将转换为URL的参数
        String application =
RpcContext.getContext().getUrl().getParameter("application");
        // 注意：每发起RPC调用，上下文状态会变化
        yyyService.yyy();
        // 此时本端变成消费端，这里会返回false
        boolean isProviderSide = RpcContext.getContext().isProviderSide();
    }
}
```

18 隐式参数

可以通过 `RpcContext` 上的 `setAttachment` 和 `getAttachment` 在服务消费方和提供方之间进行参数的隐式传递。 [1]



在服务消费方端设置隐式参数

`setAttachment` 设置的 KV 对，在完成下面一次远程调用会被清空，即多次远程调用要多次设置。

```
RpcContext.getContext().setAttachment("index", "1"); // 隐式传参，后面的远程调用都会隐式将这些参数发送到服务器端，类似cookie，用于框架集成，不建议常规业务使用
xxxService.xxx(); // 远程调用
// ...
```

在服务提供方端获取隐式参数

```
public class XxxServiceImpl implements XxxService {

    public void xxx() {
        // 获取客户端隐式传入的参数，用于框架集成，不建议常规业务使用
        String index = RpcContext.getContext().getAttachment("index");
    }
}
```

---

1. 注意：path, group, version, dubbo, token, timeout 几个 key 是保留字段，请使用其它值。 ↩

19 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。

配置

定义 injvm 协议

```
<dubbo:protocol name="injvm" />
```

设置默认协议

```
<dubbo:provider protocol="injvm" />
```

设置服务协议

```
<dubbo:service protocol="injvm" />
```

优先使用 injvm

```
<dubbo:consumer injvm="true" .../>
<dubbo:provider injvm="true" .../>
```

或

```
<dubbo:reference injvm="true" .../>
<dubbo:service injvm="true" .../>
```

注意：dubbo从 `2.2.0` 每个服务默认都会在本地暴露,无需进行任何配置即可进行本地引用,如果不希望服务进行远程暴露,只需要在provider将protocol设置成injvm即可

自动暴露、引用本地服务

从 `2.2.0` 开始，每个服务默认都会在本地暴露。在引用服务的时候，默认优先引用本地服务。如果希望引用远程服务可以使用一下配置强制引用远程服务。

```
<dubbo:reference ... scope="remote" />
```

20 延迟暴露

如果你的服务需要预热时间，比如初始化缓存，等待相关资源就位等，可以使用 delay 进行延迟暴露。我们在 Dubbo 2.6.5 版本中对服务延迟暴露逻辑进行了细微的调整，将需要延迟暴露（delay > 0）服务的倒计时动作推迟到了 Spring 初始化完成后进行。你在使用 Dubbo 的过程中，并不会感知到此变化，因此请放心使用。

Dubbo-2.6.5 之前版本

延迟到 Spring 初始化完成后，再暴露服务[1]

```
<dubbo:service delay="-1" />
```

延迟 5 秒暴露服务

```
<dubbo:service delay="5000" />
```

Dubbo-2.6.5 及以后版本

所有服务都将在 Spring 初始化完成后进行暴露，如果你不需要延迟暴露服务，无需配置 delay。

延迟 5 秒暴露服务

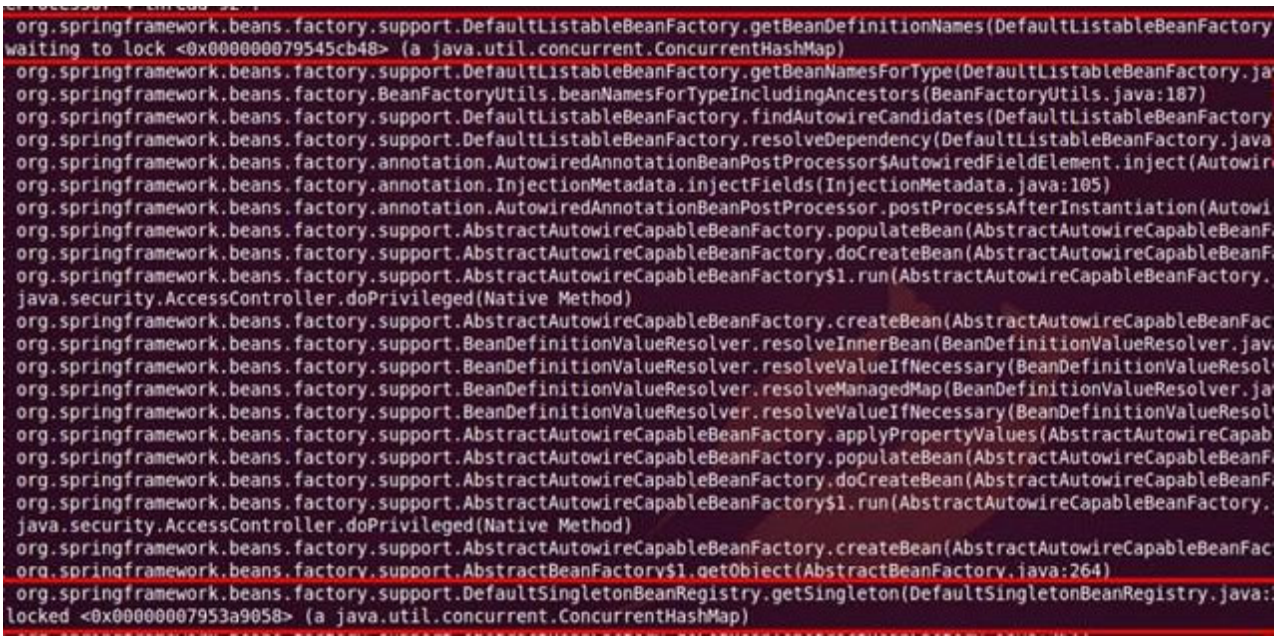
```
<dubbo:service delay="5000" />
```

Spring 2.x 初始化死锁问题

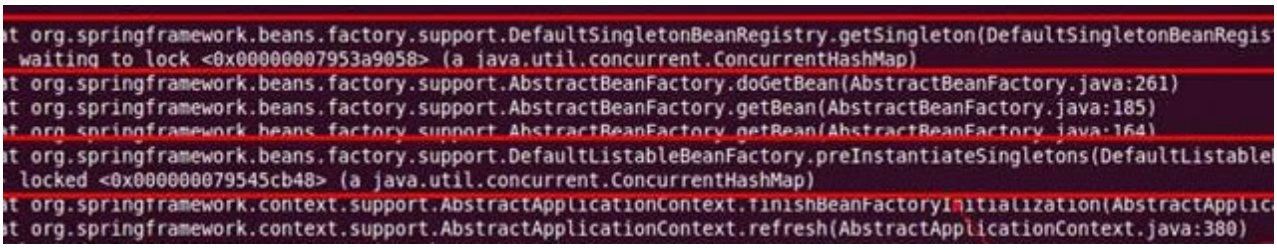
触发条件

在 Spring 解析到 `<dubbo:service />` 时，就已经向外暴露了服务，而 Spring 还在接着初始化其它 Bean。如果这时有请求进来，并且服务的实现类里有调用 `applicationContext.getBean()` 的用法。

1. 请求线程的 applicationContext.getBean() 调用，先同步 singletonObjects 判断 Bean 是否存在，不存在就同步 beanDefinitionMap 进行初始化，并再次同步 singletonObjects 写入 Bean 实例缓存。



2. 而 Spring 初始化线程，因不需要判断 Bean 的存在，直接同步 beanDefinitionMap 进行初始化，并同步 singletonObjects 写入 Bean 实例缓存。



   这样就导致 getBean 线程，先锁 singletonObjects，再锁 beanDefinitionMap，再次锁 singletonObjects。 而 Spring 初始化线程，先锁 beanDefinitionMap，再锁 singletonObjects。反向锁导致线程死锁，不能提供服务，启动不了。

规避办法

1. 强烈建议不要在服务的实现类中有 applicationContext.getBean() 的调用，全部采用 IoC 注入的方式使用 Spring的Bean。
2. 如果实在要调 getBean()，可以将 Dubbo 的配置放在 Spring 的最后加载。
3. 如果不想依赖配置顺序，可以使用 `<dubbo:provider delay=”-1” />`，使 Dubbo 在 Spring 容器初始化完后，再暴露服务。
4. 如果大量使用 getBean()，相当于已经把 Spring 退化为工厂模式在用，可以将 Dubbo 的服务隔离单独的 Spring 容器。

---

1. 基于 Spring 的 ContextRefreshedEvent 事件触发暴露 ↩