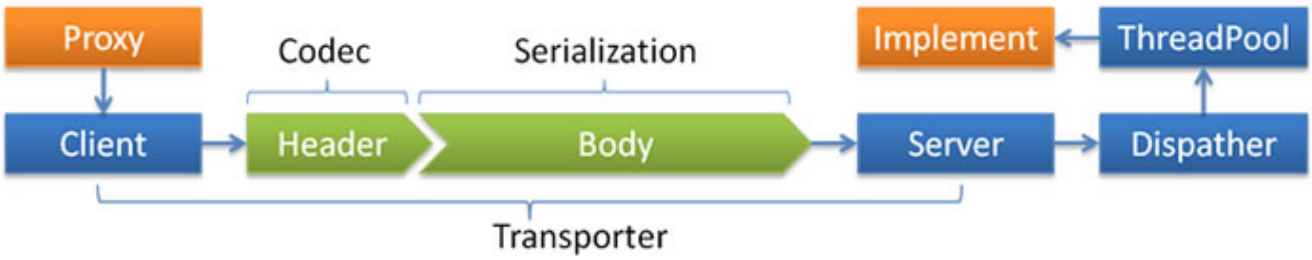
05-Dubbo特性详解-其他

1 线程模型【掌握】

如果事件处理的逻辑能迅速完成，并且不会发起新的 IO 请求，比如只是在内存中记个标识，则直接在 IO 线程上处理更快，因为减少了线程池调度。

但如果事件处理逻辑较慢，或者需要发起新的 IO 请求，比如需要查询数据库，则必须派发到线程池，否则 IO 线程阻塞，将导致不能接收其它请求。

如果用 IO 线程处理事件，又在事件处理过程中发起新的 IO 请求，比如在连接事件中发起登录请求，会报“可能引发死锁”异常，但不会真死锁。



因此，需要通过不同的派发策略和不同的线程池配置的组合来应对不同的场景:

```
<dubbo:protocol name="dubbo" dispatcher="all" threadpool="fixed"
threads="100" />
```

Dispatcher

- `all` 所有消息都派发到线程池，包括请求，响应，连接事件，断开事件，心跳等。
- `direct` 所有消息都不派发到线程池，全部在 IO 线程上直接执行。
- `message` 只有请求响应消息派发到线程池，其它连接断开事件，心跳等消息，直接在 IO 线程上执行。
- `execution` 只请求消息派发到线程池，不含响应，响应和其它连接断开事件，心跳等消息，直接在 IO 线程上执行。
- `connection` 在 IO 线程上，将连接断开事件放入队列，有序逐个执行，其它消息派发到线程池。

ThreadPool

- `fixed` 固定大小线程池，启动时建立线程，不关闭，一直持有。(缺省)
- `cached` 缓存线程池，空闲一分钟自动删除，需要时重建。
- `limited` 可伸缩线程池，但池中的线程数只会增长不会收缩。只增长不收缩的目的是为了避免收缩时突然来了大流量引起的性能问题。
- `eager` 优先创建`Worker`线程池。在任务数量大于`corePoolSize`但是小于`maximumPoolSize`时，优先创建`Worker`来处理任务。当任务数量大于`maximumPoolSize`时，将任务放入阻塞队列中。阻塞队列充满时抛出`RejectedExecutionException`。(相比于`cached`:`cached`在任务数量超过`maximumPoolSize`时直接抛出异常而不是将任务放入阻塞队列)

2 并发控制【掌握】

配置样例

样例 1

限制 `com.foo.BarService` 的每个方法，服务器端并发执行（或占用线程池线程数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService" executes="10" />
```

样例 2

限制 `com.foo.BarService` 的 `sayHello` 方法，服务器端并发执行（或占用线程池线程数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService">
    <dubbo:method name="sayHello" executes="10" />
</dubbo:service>
```

样例 3

限制 `com.foo.BarService` 的每个方法，每客户端并发执行（或占用连接的请求数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService" actives="10" />
```

或

```
<dubbo:reference interface="com.foo.BarService" actives="10" />
```

样例 4

限制 `com.foo.BarService` 的 `sayHello` 方法，每客户端并发执行（或占用连接的请求数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService">
    <dubbo:method name="sayHello" actives="10" />
</dubbo:service>
```

或

```
<dubbo:reference interface="com.foo.BarService">
    <dubbo:method name="sayHello" actives="10" />
</dubbo:service>
```

如果 `<dubbo:service>` 和 `<dubbo:reference>` 都配了actives，`<dubbo:reference>` 优先，参见：配置的覆盖策略。

Load Balance 均衡

配置服务的客户端的 `loadbalance` 属性为 `leastactive`，此 Loadbalance 会调用并发数最小的 Provider（Consumer端并发数）。

```
<dubbo:reference interface="com.foo.BarService" loadbalance="leastactive"
/>
```

或

```
<dubbo:service interface="com.foo.BarService" loadbalance="leastactive" />
```

3 异步执行【掌握】

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。异步执行无益于节省资源或提升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

定义CompletableFuture签名的接口

服务接口定义：

```
public interface AsyncService {
    CompletableFuture<String> sayHello(String name);
}
```

服务实现：

```
public class AsyncServiceImpl implements AsyncService {
    @Override
    public CompletableFuture<String> sayHello(String name) {
        RpcContext savedContext = RpcContext.getContext();
        // 建议为supplyAsync提供自定义线程池，避免使用JDK公用线程池
        return CompletableFuture.supplyAsync(() -> {
            System.out.println(savedContext.getAttachment("consumer-
key1"));
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "async response from provider.";
        });
    }
}
```

通过`return CompletableFuture.supplyAsync()`，业务执行已从Dubbo线程切换到业务线程，避免了对Dubbo线程池的阻塞。

使用AsyncContext

Dubbo提供了一个类似Serverlet 3.0的异步接口`AsyncContext`，在没有CompletableFuture签名接口的情况下，也可以实现Provider端的异步执行。

服务接口定义：

```
public interface AsyncService {
    String sayHello(String name);
}
```

服务暴露，和普通服务完全一致：

```
<bean id="asyncService"
class="org.apache.dubbo.samples.governance.impl.AsyncServiceImpl"/>
<dubbo:service
interface="org.apache.dubbo.samples.governance.api.AsyncService"
ref="asyncService"/>
```

服务实现：

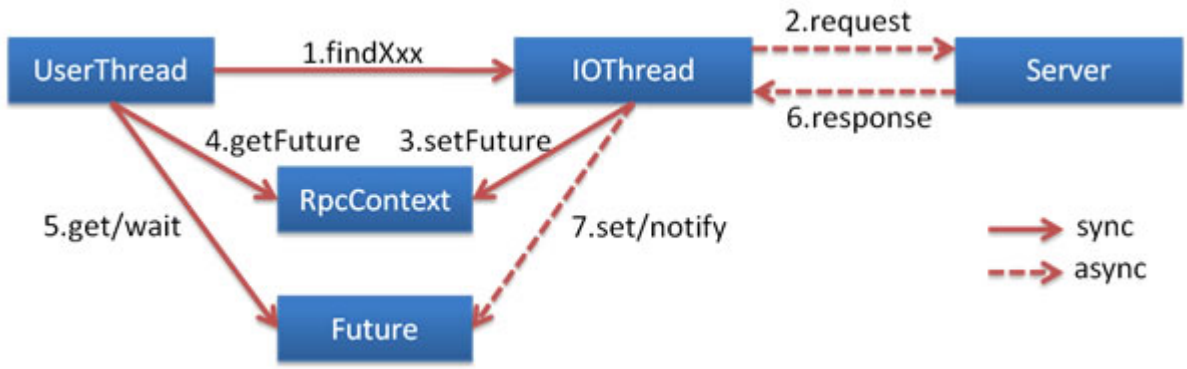
```
public class AsyncServiceImpl implements AsyncService {
    public String sayHello(String name) {
        final AsyncContext asyncContext = RpcContext.startAsync();
        new Thread(() -> {
            // 如果要使用上下文，则必须要放在第一句执行
            asyncContext.signalContextSwitch();
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // 写回响应
            asyncContext.write("Hello " + name + ", response from
provider.");
        }).start();
        return null;
    }
```

```
}
```

4 异步调用【掌握】

从v2.7.0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。



使用CompletableFuture签名的接口

需要服务提供者事先定义CompletableFuture签名的服务，具体参见服务端异步执行接口定义：

```
public interface AsyncService {
    CompletableFuture<String> sayHello(String name);
}
```

注意接口的返回类型是`CompletableFuture<String>`。

XML引用服务：

```
<dubbo:reference id="asyncService" timeout="10000"
interface="com.alibaba.dubbo.samples.async.api.AsyncService"/>
```

调用远程服务：

```
// 调用直接返回CompletableFuture
CompletableFuture<String> future = asyncService.sayHello("async call
request");
// 增加回调
future.whenComplete((v, t) -> {
    if (t != null) {
        t.printStackTrace();
    } else {
        System.out.println("Response: " + v);
    }
});
// 早于结果输出
System.out.println("Executed before response return.");
```

使用RpcContext

在 consumer.xml 中配置：

```
<dubbo:reference id="asyncService"
interface="org.apache.dubbo.samples.governance.api.AsyncService">
      <dubbo:method name="sayHello" async="true" />
</dubbo:reference>
```

调用代码:

```
// 此调用会立即返回null
asyncService.sayHello("world");
// 拿到调用的Future引用，当结果返回后，会被通知和设置到此Future
CompletableFuture<String> helloFuture =
RpcContext.getContext().getCompletableFuture();
// 为Future添加回调
helloFuture.whenComplete((retValue, exception) -> {
    if (exception == null) {
        System.out.println(retValue);
    } else {
        exception.printStackTrace();
    }
});
```

或者，你也可以这样做异步调用:

```
CompletableFuture<String> future = RpcContext.getContext().asyncCall(
    () -> {
        asyncService.sayHello("oneway call request1");
    }
);

future.get();
```

重载服务接口

如果你只有这样的同步服务定义，而又不喜欢RpcContext的异步使用方式。

```
public interface GreetingsService {
    String sayHi(String name);
}
```

那还有一种方式，就是利用Java 8提供的default接口实现，重载一个带有带有CompletableFuture签名的方法。

有两种方式来实现：

1. 提供方或消费方自己修改接口签名

```
public interface GreetingsService {
    String sayHi(String name);

    // AsyncSignal is totally optional, you can use any parameter type as long as java allows your to do that.
    default CompletableFuture<String> sayHi(String name, AsyncSignal signal) {
        return CompletableFuture.completedFuture(sayHi(name));
    }
}
```

1. Dubbo官方提供compiler hacker，编译期自动重写同步方法，请在此讨论和跟进具体进展。

你也可以设置是否等待消息发出：[1]

- `sent="true"` 等待消息发出，消息发送失败将抛出异常。
- `sent="false"` 不等待消息发出，将消息放入 IO 队列，即刻返回。

```
<dubbo:method name="findFoo" async="true" sent="true" />
```

如果你只是想异步，完全忽略返回值，可以配置 `return="false"`，以减少 Future 对象的创建和管理成本：

```
<dubbo:method name="findFoo" async="true" return="false" />
```

---

1. 异步总是不等待返回 ↩

5 连接控制【掌握】

服务端连接控制

限制服务器端接受的连接不能超过 10 个 [1]：

```
<dubbo:provider protocol="dubbo" accepts="10" />
```

或

```
<dubbo:protocol name="dubbo" accepts="10" />
```

客户端连接控制

限制客户端服务使用连接不能超过 10 个 [2]：

```
<dubbo:reference interface="com.foo.BarService" connections="10" />
```

或

```
<dubbo:service interface="com.foo.BarService" connections="10" />
```

如果 `<dubbo:service>` 和 `<dubbo:reference>` 都配了 connections，`<dubbo:reference>` 优先，参见：配置的覆盖策略

---

1. 因为连接在 Server上，所以配置在 Provider 上 ↩

2. 如果是长连接，比如 Dubbo 协议，connections 表示该服务对每个提供者建立的长连接数 ↩

6 延迟连接

延迟连接用于减少长连接数。当有调用发起时，再创建长连接。[1]

```
<dubbo:protocol name="dubbo" lazy="true" />
```

---

1. 注意：该配置只对使用长连接的 dubbo 协议生效。 ↩

7 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。

```
<dubbo:reference id="xxxService" interface="com.xxx.XxxService"
sticky="true" />
```

Dubbo 支持方法级别的粘滞连接，如果你想进行更细力度的控制，还可以这样配置。

```
<dubbo:reference id="xxxService" interface="com.xxx.XxxService">
    <dubbo:mothod name="sayHello" sticky="true" />
</dubbo:reference>
```

8 日志适配【掌握】

自 `2.2.1` 开始，dubbo 开始内置 log4j、slf4j、jcl、jdk 这些日志框架的适配[1]，也可以通过以下方式显示配置日志输出策略：

1. 命令行

```
java -Ddubbo.application.logger=log4j
```

2. 在 `dubbo.properties` 中指定

```
dubbo.application.logger=log4j
```

3. 在 `dubbo.xml` 中配置

```
<dubbo:application logger="log4j" />
```

[1]: 自定义扩展可以参考 日志适配扩展

9 访问日志

如果你想记录每一次请求信息，可开启访问日志，类似于apache的访问日志。**注意**：此日志量比较大，请注意磁盘容量。

将访问日志输出到当前应用的log4j日志：

```
<dubbo:protocol accesslog="true" />
```

将访问日志输出到指定文件：

```
<dubbo:protocol
accesslog="http://10.20.160.198/wiki/display/dubbo/foo/bar.log" />
```

10 动态配置中心

http://dubbo.apache.org/zh-cn/docs/user/configuration/config-center.html

11 配置加载流程【掌握】

http://dubbo.apache.org/zh-cn/docs/user/configuration/configuration-load-process.html

12 调用拦截扩展【掌握】

扩展说明

服务提供方和服务消费方调用过程拦截，Dubbo 本身的大多功能均基于此扩展点实现，每次远程方法执行，该拦截都会被执行，请注意对性能的影响。

约定：

- 用户自定义 filter 默认在内置 filter 之后。
- 特殊值 `default`，表示缺省扩展点插入的位置。比如：`filter="xxx,default,yyy"`，表示 `xxx` 在缺省 filter 之前，`yyy` 在缺省 filter 之后。
- 特殊符号 `-`，表示剔除。比如：`filter="-foo1"`，剔除添加缺省扩展点 `foo1`。比如：`filter="-default"`，剔除添加所有缺省扩展点。
- provider 和 service 同时配置的 filter 时，累加所有 filter，而不是覆盖。比如：`<dubbo:provider filter="xxx,yyy"/>` 和 `<dubbo:service filter="aaa,bbb" />`，则 `xxx,yyy,aaa,bbb` 均会生效。如果要覆盖，需配置：`<dubbo:service filter="-xxx,-yyy,aaa,bbb" />`

扩展接口

```
org.apache.dubbo.rpc.Filter
```

扩展配置

```
<!-- 消费方调用过程拦截 -->
<dubbo:reference filter="xxx,yyy" />
<!-- 消费方调用过程缺省拦截器，将拦截所有reference -->
<dubbo:consumer filter="xxx,yyy"/>
<!-- 提供方调用过程拦截 -->
<dubbo:service filter="xxx,yyy" />
<!-- 提供方调用过程缺省拦截器，将拦截所有service -->
<dubbo:provider filter="xxx,yyy"/>
```

已知扩展

- `org.apache.dubbo.rpc.filter.EchoFilter`
- `org.apache.dubbo.rpc.filter.GenericFilter`
- `org.apache.dubbo.rpc.filter.GenericImplFilter`
- `org.apache.dubbo.rpc.filter.TokenFilter`
- `org.apache.dubbo.rpc.filter.AccessLogFilter`
- `org.apache.dubbo.rpc.filter.CountFilter`
- `org.apache.dubbo.rpc.filter.ActiveLimitFilter`
- `org.apache.dubbo.rpc.filter.ClassLoaderFilter`
- `org.apache.dubbo.rpc.filter.ContextFilter`
- `org.apache.dubbo.rpc.filter.ConsumerContextFilter`
- `org.apache.dubbo.rpc.filter.ExceptionFilter`
- `org.apache.dubbo.rpc.filter.ExecuteLimitFilter`
- `org.apache.dubbo.rpc.filter.DeprecatedFilter`

扩展示例

Maven 项目结构：

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

XxxFilter.java：

```
package com.xxx;

import org.apache.dubbo.rpc.Filter;
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 XxxFilter implements Filter {
    public Result invoke(Invoker<?> invoker, Invocation invocation) throws
RpcException {
        // before filter ...
        Result result = invoker.invoke(invocation);
        // after filter ...
        return result;
    }
}
```

META-INF/dubbo/org.apache.dubbo.rpc.Filter :

```
xxx=com.xxx.XxxFilter
```