crazyhzm 265

Dubbo源码解析（四十八）异步化改造

dubbo java 阅读约 27 分钟

2.7大揭秘——异步化改造

> 目标：从源码的角度分析2.7的新特性中对于异步化的改造原理。

前言

dubbo中提供了很多类型的协议，关于协议的系列可以查看下面的文章：

- dubbo源码解析（二十四）远程调用——dubbo协议
- dubbo源码解析（二十五）远程调用——hessian协议
- dubbo源码解析（二十六）远程调用——http协议
- dubbo源码解析（二十七）远程调用——injvm本地调用
- dubbo源码解析（二十八）远程调用——memcached协议
- dubbo源码解析（二十九）远程调用——redis协议
- dubbo源码解析（三十）远程调用——rest协议
- dubbo源码解析（三十一）远程调用——rmi协议
- dubbo源码解析（三十二）远程调用——thrift协议
- dubbo源码解析（三十三）远程调用——webservice协议

官方推荐的是使用dubbo协议，而异步调用的支持也是在dubbo协议中实现的。

看了我之前写的2.7新特性的文章，应该对于异步化改造有个大致的印象。要弄懂异步在什么时候起作用，先要弄懂dubbo 的服务暴露和引用过程以及消费端发送请求过程和服务端处理请求过程。我在前四篇文章已经讲述了相关内容，异步请求只是dubbo的一种请求方式，基于 dubbo 底层的异步 NIO 实现异步调用，对于 Provider 响应时间较长的场景是必须的，它能有效利用 Consumer 端的资源，相对于 Consumer 端使用多线程来说开销较小。可以让消费者无需阻塞等待返回结果。

经过改良后，Provider端也支持异步处理请求，引用官网的话就是现在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。

如何开启和使用异步可以查看以下链接：

> Provider异步执行：http://dubbo.apache.org/zh-cn/docs/user/demos/async-execute-on-provider.html
> Consumer异步调用：http://dubbo.apache.org/zh-cn/docs/user/demos/async-call.html

异步的改造

Listener做为Filter的内部接口

从设计上

1. 废弃了Filter原先的onResponse()方法
2. 在Filter接口新增了内部接口Listener，相关接口设计如下。
    - 优点：职责划分更加明确，进行逻辑分组，增强可读性，Filter本身应仅是传递调用的响应，而所有回调都放入Listener。这样做以后可以把之前回调的逻辑从invoke里面剥离出来，放到Listener的onResponse或者onError中。

```
interface Listener {

    /**
     * 回调正常的调用结果
     * @param appResponse
     * @param invoker
     * @param invocation
     */
    void onResponse(Result appResponse, Invoker<?> invoker, Invocation invocation);

    /**
     * 回调异常结果
     * @param t
     * @param invoker
     * @param invocation
     */
    void onError(Throwable t, Invoker<?> invoker, Invocation invocation);
}
```

1. 新增抽象类ListenableFilter，实现了Filter接口，其中只记录了一个该过滤器的内部Listener实例。

```
public abstract class ListenableFilter implements Filter {

    protected Listener listener = null;

    public Listener listener() {
        // 提供该过滤器的内部类listener
        return listener;
    }
}
```

异步转同步，新增InvokeMode

不变的是配置来决定调用方式，变的是在何时去做同步异步的不同逻辑处理。看《dubbo源码解析（四十六）消费端发送请求过程》讲到的（十四）DubboInvoker的doInvoke，在以前的逻辑会直接在doInvoke方法中根据配置区分同步、异步、单向调用。现在只单独做了单向调用和需要返回结果的区分，统一先使用AsyncRpcResult来表示结果，也就是说一开始统一都是异步调用，然后在调用回到AsyncToSyncInvoker的invoke中时，才对同步异步做区分，这里新增了InvokeMode，InvokeMode现在有三种模式：SYNC, ASYNC, FUTURE。前两种很显而易见，后面一种是调用的返回类型是Future类型，代表调用的方法的返回类型是CompletableFuture类型，这种模式专门用来支持服务端异步的。看下面的源码。

```
public static InvokeMode getInvokeMode(URL url, Invocation inv) {
    // 如果返回类型是future
    if (isReturnTypeFuture(inv)) {
        return InvokeMode.FUTURE;
    } else if (isAsync(url, inv)) {
        // 如果是异步调用
        return InvokeMode.ASYNC;
    } else {
        // 如果是同步
        return InvokeMode.SYNC;
    }
}
```

参考《dubbo源码解析（四十六）消费端发送请求过程》的（十二）AsyncToSyncInvoker的invoke逻辑，如果是同步模式，就会阻塞调用get方法。直到调用成功有结果返回。如果不是同步模式，就直接返回。

ResponseFuture改为CompletableFuture

关于ResponseFuture可以参考《dubbo源码解析（十）远程通信——Exchange层》的（六）ResponseFuture。具体的可以看它的两个实现（七）DefaultFuture和（八）SimpleFuture。

在这次改造中，最小JDK版本从以前的1.6变成了1.8。当然也要用到1.8中新特性，其中就包括CompletableFuture。dubbo的通信主要有两处，一处是Consumer发送请求消息给Provider，另一处就是Provider把结果发送给Consumer。在Consumer发送请求消息给Provider的时候，Consumer不会一直处于等待，而是生成ResponseFuture会抛给下游去做其他操作，等到Provider把结果返回放入ResponseFuture，Consumer可以通过get方法获得结果，或者它也支持回调。但是这就暴露了一些问题，也就是为在新特性里提到的缺陷：

- Future只支持阻塞式的get()接口获取结果。因为future.get()会导致线程阻塞。
- Future接口无法实现自动回调，而自定义ResponseFuture虽支持callback回调但支持的异步场景有限，如不支持Future间的相互协调或组合等；

针对以上两个不足，CompletableFuture可以很好的解决它们。

- 针对第一点不足，因为CompletableFuture实现了CompletionStage和Future接口，所以它还是可以像以前一样通过阻塞或者轮询的方式获得结果。这一点就能保证阻塞式获得结果，也就是同步调用不会被抛弃。当然本身也不是很建议用get()这样阻塞的方式来获取结果。
- 针对第二点不足，首先是自动回调，CompletableFuture提供了良好的回调方法。比如下面四个方法有关计算结果完成时的处理：

```
public CompletableFuture<T>     whenComplete(BiConsumer<? super T,? super Throwable> action)
public CompletableFuture<T>     whenCompleteAsync(BiConsumer<? super T,? super Throwable> action)
public CompletableFuture<T>     whenCompleteAsync(BiConsumer<? super T,? super Throwable> action, Executor executor)
public CompletableFuture<T>     exceptionally(Function<Throwable,? extends T> fn)
```

当计算完成后，就会执行该方法中的action方法。相比于ResponseFuture，不再需要自己去做回调注册的编码，更加易于理解。

- 还是针对第二点，自定义的ResponseFuture不支持Future间的相互协调或组合，CompletableFuture很好的解决了这个问题，在CompletableFuture中以下三个方法实现了future之间转化的功能：

```
public <U> CompletableFuture<U>     thenApply(Function<? super T,? extends U> fn)
public <U> CompletableFuture<U>     thenApplyAsync(Function<? super T,? extends U> fn)
public <U> CompletableFuture<U>     thenApplyAsync(Function<? super T,? extends U> fn, Executor executor)
```

由于回调风格的实现，我们不必因为等待一个计算完成而阻塞着调用线程，而是告诉CompletableFuture当计算完成的时候请执行某个function。而且我们还可以将这些操作串联起来，或者将CompletableFuture组合起来。这一组函数的功能是当原来的CompletableFuture计算完后，将结果传递给函数fn，将fn的结果作为新的CompletableFuture计算结果。因此它的功能相当于将CompletableFuture<T>转换成CompletableFuture<U>。

除了转化之外，还有future之间组合的支持，例如以下三个方法：

```
public <U> CompletableFuture<U>     thenCompose(Function<? super T,? extends CompletionStage<U>> fn)
public <U> CompletableFuture<U>     thenComposeAsync(Function<? super T,? extends CompletionStage<U>> fn)
public <U> CompletableFuture<U>     thenComposeAsync(Function<? super T,? extends CompletionStage<U>> fn, Executor
executor)
```

这一组方法接受一个Function作为参数，这个Function的输入是当前的CompletableFuture的计算值，返回结果将是一个新的CompletableFuture，这个新的CompletableFuture会组合原来的CompletableFuture和函数返回的CompletableFuture。

现在就能看出CompletableFuture的强大了，它解决了自定义ResponseFuture的许多问题，该类有几十个方法，感兴趣的可以去一一尝试。

随处可见的CompletableFuture

可以看到以前的版本只能在RpcContext中进行获取。而经过改良后，首先RpcContext一样能过获取，其次在过滤器链返回的Result中也能获取，可以从最新的代码中看到，原先的RpcResult类已经被去除，而在AsyncRpcResult也继承了CompletableFuture<Result>类，也就是说有AsyncRpcResult的地方，就有CompletableFuture。并且在后续的dubbo3.0中，AsyncRpcResult将会内置CompletableFuture类型的变量，CompletableFuture的获取方式也会大大增加。

AsyncRpcResult全面替代RpcResult

接下来我就来讲解一下AsyncRpcResult类。

```
/**
 * 当相同的线程用于执行另一个RPC调用时，并且回调发生时，原来的RpcContext可能已经被更改。
 * 所以我们应该保留当前RpcContext实例的引用，并在执行回调之前恢复它。
 * 存储当前的RpcContext
 */
private RpcContext storedContext;
/**
 * 存储当前的ServerContext
 */
private RpcContext storedServerContext;

/**
 * 会话域
 */
private Invocation invocation;

public AsyncRpcResult(Invocation invocation) {
    // 设置会话域
    this.invocation = invocation;
    // 获得当前线程内代表消费者端的Context
    this.storedContext = RpcContext.getContext();
    // 获得当前线程内代表服务端的Context
    this.storedServerContext = RpcContext.getServerContext();
}

/**
```

上面的是AsyncRpcResult核心的变量以及构造函数，storedContext和storedServerContext存储了相关的RpcContext实例的引用，为的就是防止在回调的时候由于相同的线程用于执行另一个RPC调用导致原来的RpcContext可能已经被更改。所以存储下来后，我们需要在执行回调之前恢复它。具体的可以看下面的thenApplyWithContext方法。

```
@Override
public Object getValue() {
    // 获得计算的结果
    return getAppResponse().getValue();
}

@Override
public void setValue(Object value) {
    // 创建一个AppResponse实例
    AppResponse appResponse = new AppResponse();
    // 把结果放入AppResponse
    appResponse.setValue(value);
    // 标志该future完成，并且把携带结果的appResponse设置为该future的结果
    this.complete(appResponse);
}

@Override
public Throwable getException() {
    // 获得抛出的异常信息
    return getAppResponse().getException();
}

@Override
public void setException(Throwable t) {
    // 创建一个AppResponse实例
    AppResponse appResponse = new AppResponse();
```

这些实现了Result接口的方法，可以发现其中都是调用了AppResponse的方法，AppResponse跟AsyncRpcResult一样也继承了AbstractResult，不过它是作为回调的数据结构。AppResponse我会在异步化过滤器链回调中讲述。

```
@Override
public Object recreate() throws Throwable {
    // 强制类型转化
    RpcInvocation rpcInvocation = (RpcInvocation) invocation;
    // 如果返回的是future类型
    if (InvokeMode.FUTURE == rpcInvocation.getInvokeMode()) {
        // 创建AppResponse实例
        AppResponse appResponse = new AppResponse();
        // 创建future
        CompletableFuture<Object> future = new CompletableFuture<>();
        // appResponse设置future值，因为返回的就是CompletableFuture类型
        appResponse.setValue(future);
        // 当该AsyncRpcResult完成的时候，把结果放入future中，这样返回的就是CompletableFuture包裹的结果
        this.whenComplete((result, t) -> {
            if (t != null) {
                if (t instanceof CompletionException) {
                    t = t.getCause();
                }
                future.completeExceptionally(t);
            } else {
                if (result.hasException()) {
                    future.completeExceptionally(result.getException());
                } else {
                    future.complete(result.getValue());
                }
            }
```

该方法是重置，本来也是直接调用了AppResponse的方法，不过因为支持了以CompletableFuture为返回类型的服务方法调用，所以这里做了一些额外的逻辑，也就是把结果用CompletableFuture包裹，作为返回的结果放入AppResponse实例中。可以对标使用了CompletableFuture签名的服务。

```
@Override
public Result thenApplyWithContext(Function<Result, Result> fn) {
    // 当该AsyncRpcResult完成后，结果作为参数先执行beforeContext，再执行fn，最后执行andThen
    this.thenApply(fn.compose(beforeContext).andThen(afterContext));
    // You may need to return a new Result instance representing the next async stage,
    // like thenApply will return a new CompletableFuture.
    return this;
}


/**
 * tmp context to use when the thread switch to Dubbo thread.
 * 临时的RpcContext，当用户线程切换为Dubbo线程时候使用
 */
/**
 * 临时的RpcContext
 */
private RpcContext tmpContext;
private RpcContext tmpServerContext;

private Function<Result, Result> beforeContext = (appResponse) -> {
    // 获得当前线程消费者端的RpcContext
    tmpContext = RpcContext.getContext();
    // 获得当前线程服务端的RpcContext
    tmpServerContext = RpcContext.getServerContext();
    // 重新设置消费者端的RpcContext
```

把这几部分代码放在一起时因为当用户线程切换为Dubbo线程时候需要用到临时的RpcContext来记录，如何使用该thenApplyWithContext方法，我也会在异步化过滤器链回调中讲到。

其他的方法比较好理解，我就不一一讲解。

异步化过滤器链回调

如果看过前两篇关于发送请求和处理请求的过程，应该就知道在整个调用链中有许多的过滤器，而Consumer和Provider分别都有各自的过滤器来做一些功能增强。过滤器有执行链，也有回调链，如果整一个链路都是同步的，那么过滤器一旦增多，链路增长，就会带来请求响应时间的增加，这当然是最不想看到的事情。那如果把过滤器的调用链异步化，那么我们就可以用一个future来代替结果抛给下游，让下游不再阻塞。这样就大大降低了响应时间，节省资源，提升RPC响应性能。而这里的future就是下面要介绍的AppResponse。那我先来介绍一下如何实现异步化过滤器链回调。我就拿消费端发送请求过程来举例子说明。

参考《dubbo源码解析（四十六）消费端发送请求过程》的（六）ProtocolFilterWrapper的内部类CallbackRegistrationInvoker的invoke，可以看到当所有的过滤器执行完后，会遍历每一个过滤器链，获得上面所说的内部接口Listener实现类，进行异步回调，因为请求已经在（十四）DubboInvoker的doInvoke中进行了发送，返回给下游一个AsyncRpcResult，而AsyncRpcResult内包裹的是AppResponse，可以看《dubbo源码解析（四十七）服务端处理请求过程》的（二十三）AbstractProxyInvoker的invoke，当代理类执行相关方法后，会创建一个AppResponse，把结果放入AppResponse中。所以AsyncRpcResult中包裹的是AppResponse，然后调用回调方法onResponse。并且会执行thenApplyWithContext把回调结果放入上下文中。而这个上下文如何避免相同的线程用于执行另一个RPC调用导致原来的RpcContext可能已经被更改的情况，我也在上面已经说明。

新增AppResponse

AppResponse继承了AbstractResult，同样也是CompletableFuture<Result>类型，但是AppResponse跟AsyncRpcResult职能不一样，AsyncRpcResult作为一个future，而AppResponse可以说是作为rpc调用结果的一个数据结构，它的实现很简单，就是封装了以下三个属性和对应的一些方法。

```
/**
 * 调用结果
 */
private Object result;

/**
 * rpc调用时的异常
 */
private Throwable exception;

/**
 * 附加值
 */
private Map<String, String> attachments = new HashMap<String, String>();
```

前面我也讲了，Provider处理请求完成后，会把结果放在AppResponse内，在整个链路调用过程中AsyncRpcResult内部必然会有一个AppResponse存在，而为上文提到的过滤器内置接口Listener的onResponse方法中的appResponse就是AppResponse类型的，它作为一个回调的数据类型。

后记

该文章讲解了dubbo 2.7.x版本对于异步化改造的介绍，上面只是罗列了所有改动的点，没有具体讲述在哪些新增功能上的应用，如果感兴趣，可以参考前几篇的调用过程文章，来看看新增的功能点如何运用上述的设计的，比如Provider异步，有一种实现方式就用到了上述的InvokeMode。接下来一篇我会讲述元数据的改造。

crazyhzm

265

关注作者

0 条评论

得票 • 时间