# Dubbo版本的一些细节

## 解决provider端线程池满后，无法通知给consumer的问题

在`provider`端接收到消息的时候，消息会走过一个`ChannelHandler`链，其中有一个节点是类`AllChannelHandler`，我们看一下它的`received`方法：

```
try {
    cexecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState
.RECEIVED, message));
} catch (Throwable t) {
    //TODO A temporary solution to the problem that the exception information
can not be sent to the opposite end after the thread pool is full. Need a ref
actoring
    //fix The thread pool is full, refuses to call, does not return, and caus
es the consumer to wait for time out
    if(message instanceof Request && t instanceof RejectedExecutionException)
{
        Request request = (Request)message;
        if(request.isTwoWay()){
            String msg = "Server side(" + url.getIp() + "," + url.getPort() +
") threadpool is exhausted ,detail msg:" + t.getMessage();
            Response response = new Response(request.getId(), request.getVers
ion());
            response.setStatus(Response.SERVER_THREADPOOL_EXHAUSTED_ERROR);
            response.setErrorMessage(msg);
            channel.send(response);
            return;
        }
    }
    throw new ExecutionException(message, channel, getClass() + " error when
process received event .", t);
}
```

如果抛出的是`RejectedExecutionException`异常，则直接回写一个`respons`告诉调用方，线程池已经打满了。

这个问题是在`2.5.6`版本修复的。

## Provider下线过程中，Consumer正在进行调用

我们一般情况下，`provider`都不止一个，这个时候，我们实际上拿到的`invoker`是`ClusterInvoker`，而真正的执行下放到子类，子类中实现了各种策略，默认是`FailoverClusterInvoker`，这个类的`doInvoke`方法：

```
Set<String> providers = new HashSet<String>(len);
for (int i = 0; i < len; i++) {
    //Reselect before retry to avoid a change of candidate `invokers`.
    //NOTE: if `invokers` changed, then `invoked` also lose accuracy.
    if (i > 0) {
        checkWhetherDestroyed();
        copyinvokers = list(invocation);
        // check again
        checkInvokers(copyinvokers, invocation);
    }
    Invoker<T> invoker = select(loadbalance, invocation, copyinvokers, invoke
d);
    invoked.add(invoker);
    RpcContext.getContext().setInvokers((List) invoked);
    try {
        Result result = invoker.invoke(invocation);
        if (le != null && logger.isWarnEnabled()) {
            logger.warn(...);
        }
        return result;
    } catch (RpcException e) {
        if (e.isBiz()) { // biz exception.
            throw e;
        }
        le = e;
    } catch (Throwable e) {
        le = new RpcException(e.getMessage(), e);
    } finally {
        providers.add(invoker.getUrl().getAddress());
    }
}
```

如果`Result result = invoker.invoke(invocation);`执行失败了，这时候我们因为`provider`下线了，那么调用肯定是会抛出异常的，这时候策略是：**捕获异常，重新尝试另外一个`invoker`，直到成功或者重试次数用完。**

## 关于Dubbo的`LeastActiveLoadBalance`负载均衡策略

负载均衡策略是在`consumer`端生效，选择`invoker`时起作用的`dubbo`的策略，默认是`random`。真正生产环境一般选`LeastActiveLoadBalance`，作用是选择最小活跃的那个`invoker`。一个`invoker`实际上对应的就是一台真正的机器（这个`invoker`是最小粒度的`invoker`，不是`clusterInvoker`），如果开启了最下活跃策略，那么`invoker`上如果有一个调用开始，则计数器+1，如果结束则-1。这个策略会优先选择活跃数量比较小的那个`invoker`。注意这里并不一定会选择最小的，而是大概率选择活跃数小的。

## 启动阶段zk无法连接导致应用无限阻塞

这个问题在`2.5.7`中已经被修复，`dubbo`默认使用`zk`作为注册中心，我们看一下`ZkclientZookeeperClient`的构造函数：

```
super(url);
//这里多了一个30000，这就是超时时间，30秒.
client = new ZkClientWrapper(url.getBackupAddress(), 30000);
client.addListener(new IZkStateListener() {
    public void handleStateChanged(KeeperState state) throws Exception {
        ZkclientZookeeperClient.this.state = state;
        if (state == KeeperState.Disconnected) {
            stateChanged(StateListener.DISCONNECTED);
        } else if (state == KeeperState.SyncConnected) {
            stateChanged(StateListener.CONNECTED);
        }
    }

    public void handleNewSession() throws Exception {
        stateChanged(StateListener.RECONNECTED);
    }
});
client.start();
```

继续看一下`ZkClientWrapper`的`start`方法：

```
if (!started) {
    Thread connectThread = new Thread(listenableFutureTask);
    connectThread.setName("DubboZkclientConnector");
    connectThread.setDaemon(true);
    connectThread.start();
    try {
        client = listenableFutureTask.get(timeout, TimeUnit.MILLISECONDS);
    } catch (Throwable t) {
        logger.error("Timeout! zookeeper server can not be connected in : " +
```

```
timeout + "ms!", t);
    }
    started = true;
} else {
    logger.warn("Zkclient has already been started!");
}
```

会从`listenableFutureTask`尝试获取一个`client`，而创建这个`client`是在构造函数里：

```
public ZkClientWrapper(final String serverAddr, long timeout) {
    this.timeout = timeout;
    //创建一个任务，主要是创建一个zkclient
    listenableFutureTask = ListenableFutureTask.create(new Callable<ZkClient>
() {
        @Override
        public ZkClient call() throws Exception {
            return new ZkClient(serverAddr, Integer.MAX_VALUE);
        }
    });
}
```

## Dubbo的provider线程池满，自动JStack

上面说过，`dubbo`的`provider`端接受请求会启动一个线程池来处理（`AllChannelHandler`），这个线池如果满了会把线程池满的`response`返回给`consumer`。

这个线程池默认使用的是`FixedThreadPool`，拒绝策略默认使用的是`AbortPolicyWithReport`，看一下`rejectedExecution`：

```
String msg = String.format("Thread pool is EXHAUSTED!" +
                " Thread Name: %s, Pool Size: %d (active: %d, core: %d, max:
%d, largest: %d), Task: %d (completed: %d)," +
                " Executor status:(isShutdown:%s, isTerminated:%s, isTerminat
ing:%s), in %s://%s:%d!",
        threadName, e.getPoolSize(), e.getActiveCount(), e.getCorePoolSize(),
e.getMaximumPoolSize(), e.getLargestPoolSize(),
        e.getTaskCount(), e.getCompletedTaskCount(), e.isShutdown(), e.isTerm
inated(), e.isTerminating(),
        url.getProtocol(), url.getIp(), url.getPort());
logger.warn(msg);
```

```
dumpJStack();
throw new RejectedExecutionException(msg);
```

里面有一句`dumpJStack`，这个也是`dubbo`的新特性，最后通过`JVMUtil`来`dump`线程。

## Dubbo的SerializerFactory在获取Serializer和Deserializer时锁住了整个cachedMap导致线程block

这个`SerializerFactory`是从`hessian`中移植过来的，我们看一下获取`Serializer`的方法：

```
public Serializer getSerializer(Class cl)
        throws HessianProtocolException {
    Serializer serializer;

    serializer = (Serializer) _staticSerializerMap.get(cl);
    if (serializer != null)
        return serializer;

    if (_cachedSerializerMap != null) {
        synchronized (_cachedSerializerMap) {
            serializer = (Serializer) _cachedSerializerMap.get(cl);
        }

        if (serializer != null)
            return serializer;
    }
```

这个`_cachedSerializerMap`是一个`hashmap`，获取时直接锁这个`map`，请求量比较大的时候，获取`Serializer`就会阻塞。修改方式很简单，把`hashmap`全部替换为`concurrentHashMap`。

刚刚说过这个`Factory`是从`hessian`中移植过来的，`dubbo`目前依赖的`hessian`是**4.0.38**，目前在这个版本中`hessian`也已经把这个map替换为`concurrentHashMap`。这个问题笔者已经提交过了PR并且已经被merge到了`master`分支上了，应该会随着下一个版本发布。