

Zookeeper3.7源码剖析

能力目标

- 能基于Maven导入最新版Zookeeper源码
- 能说出Zookeeper单机启动流程
- 理解Zookeeper默认通信中4个线程的作用
- 掌握Zookeeper业务处理源码处理流程
- 能够在Zookeeper源码中Debug测试通信过程

1 Zookeeper源码导入

 Apache ZooKeeper™ Project Documentation Developers ASF

Welcome to Apache ZooKeeper™

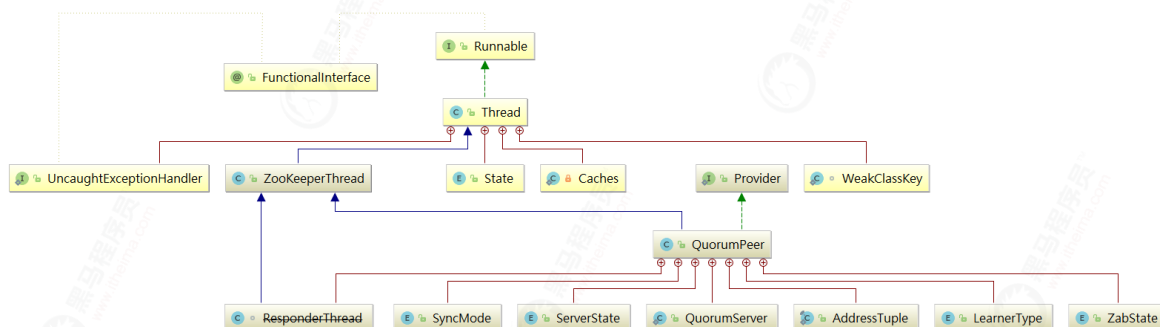
Apache ZooKeeper is an effort to develop and maintain an open-source server which enables highly reliable distributed coordination.

What is ZooKeeper?

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them, which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed.

[Learn more about ZooKeeper on the ZooKeeper Wiki.](#)

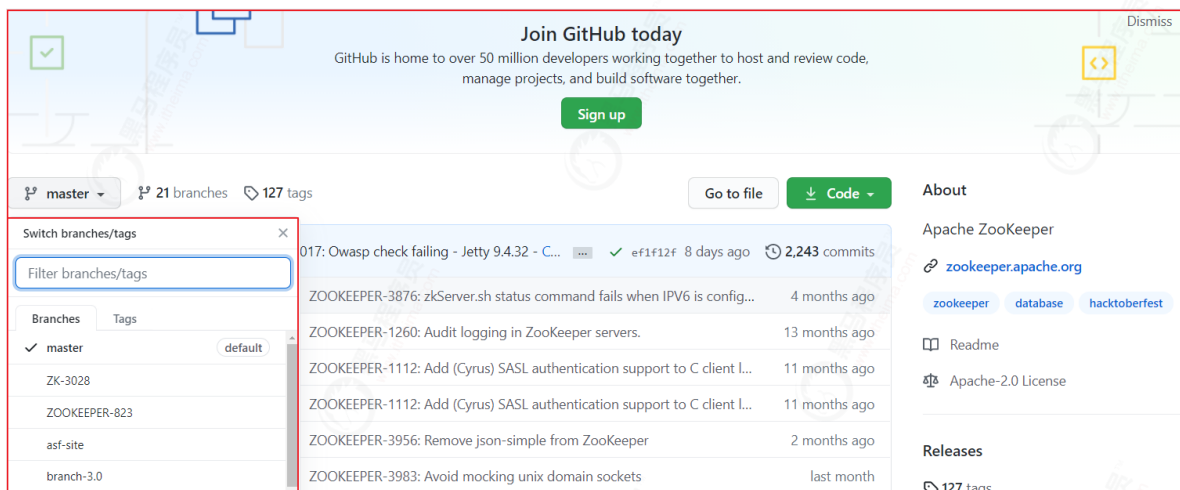
ZooKeeper是一项集中式服务，用于维护配置信息、命名、提供分布式同步和群组服务。所有服务均以某种形式通过分布式应用程序使用。众所周知，这些服务很难正确实现，每次实现都不可避免要出现各种各样的错误和争用条件（race conditions），而不得不花费大量的精力加以修复。因此，应用程序一般会在一开始忽略这些服务，而使得它们在出现变化时更加脆弱，难以管理。即使服务得到正确实现，但是，由于实现手段的不同，增加了应用部署后的管理复杂性。



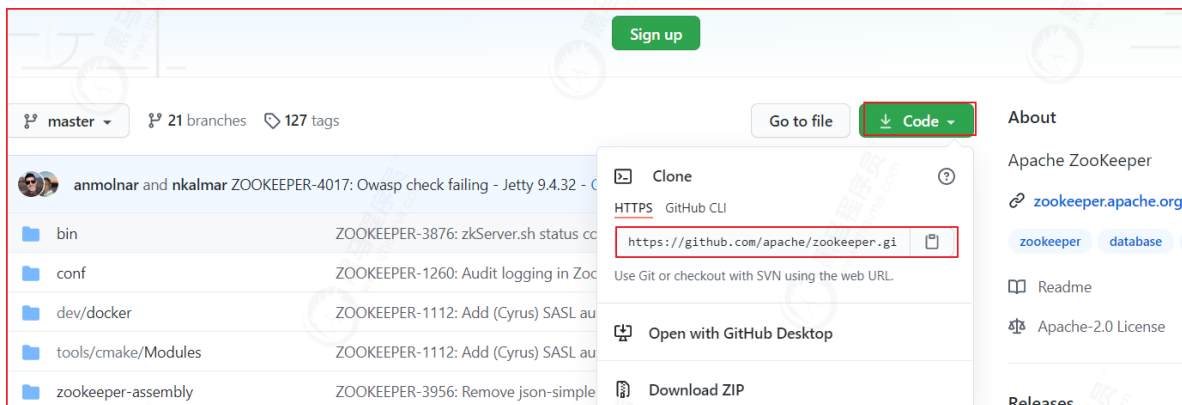
我们当前课程主要是研究Zookeeper源码，需要将Zookeeper工程导入到IDEA中，老版的zk是通过ant进行编译的，但最新的zk(3.7)源码中已经没了 `build.xml`，而多了 `pom.xml`，也就是说构建方式由原先的Ant变成了Maven，源码下下来后，直接编译、运行是跑不起来的，有一些配置需要调整。

1.1 工程导入

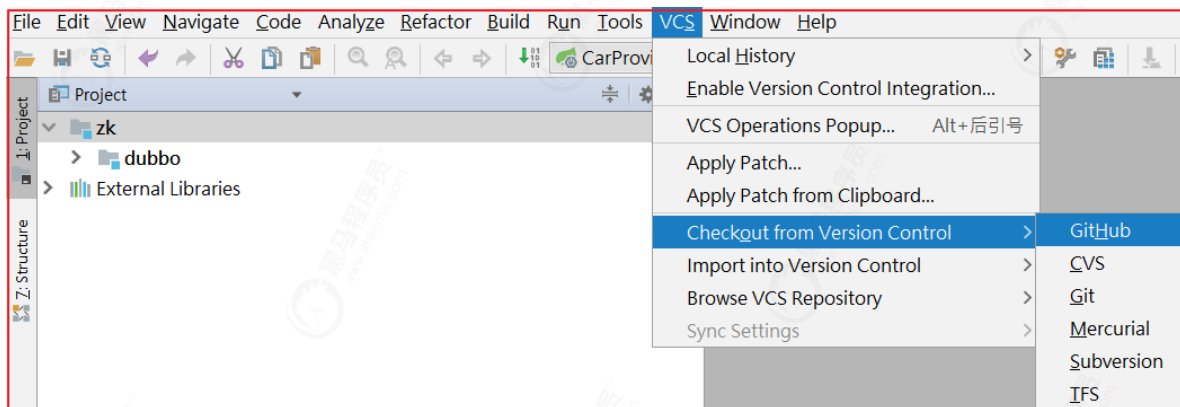
Zookeeper各个版本源码下载地址<https://github.com/apache/zookeeper>，我们可以在该仓库下选择不同的版本，我们选择最新版本，当前最新版本为3.7，如下图：



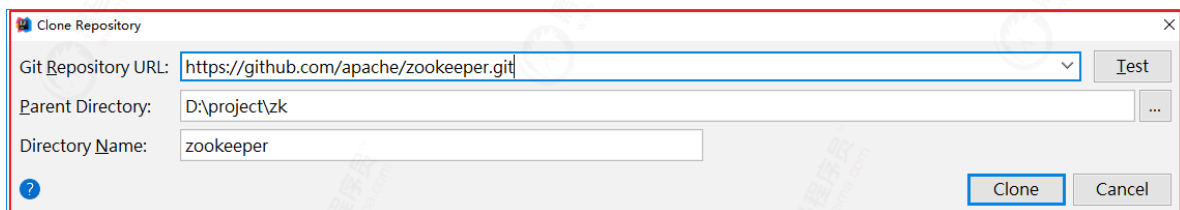
找到项目下载地址，我们选择 `https` 地址，并复制该地址，通过该地址把项目导入到 `IDEA` 中。



点击IDEA的 `VCS>Checkout from Version Controller>GitHub`，操作如下图：



克隆项目到本地：



项目导入本地后，效果如下：

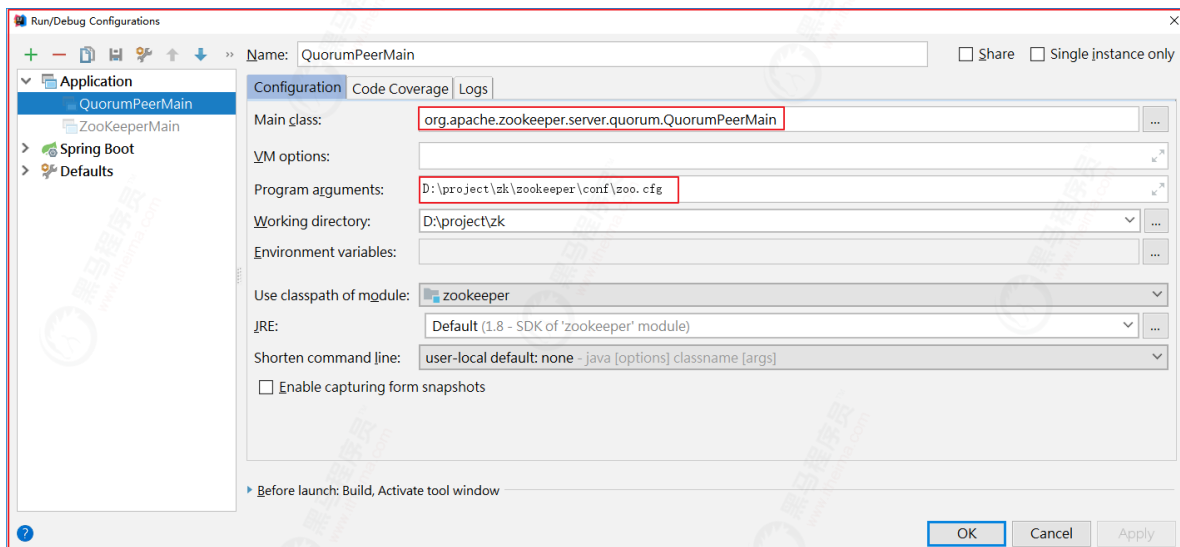


项目运行的时候，缺一个版本对象，创建 `org.apache.zookeeper.version.Info`，代码如下：

```
public interface Info {
    public static final int MAJOR=3;
    public static final int MINOR=4;
    public static final int MICRO=6;
    public static final String QUALIFIER=null;
    public static final int REVISION=-1;
    public static final String REVISION_HASH = "1";
    public static final String BUILD_DATE="2020-12-03 09:29:06";
}
```

1.2 Zookeeper源码错误解决

在 `zookeeper-server` 中找到 `org.apache.zookeeper.server.quorum.QuorumPeerMain` 并启动该类，启动前做如下配置：



启动的时候会报很多错误，比如缺包、缺对象，如下几幅图：

```
Exception in thread "main" java.lang.NoClassDefFoundError: com/codahale/metrics/Reservoir
    at org.apache.zookeeper.metrics.impl.DefaultMetricsProvider$DefaultMetricsContext.lambda$getSummary$2(DefaultMetricsProvider.java:126)
    <1 internal call>
    at org.apache.zookeeper.metrics.impl.DefaultMetricsProvider$DefaultMetricsContext.getSummary(DefaultMetricsProvider.java:122)
    at org.apache.zookeeper.server.ServerMetrics.<init>(ServerMetrics.java:74)
    at org.apache.zookeeper.server.ServerMetrics.<clinit>(ServerMetrics.java:44)
    at org.apache.zookeeper.server.ZooKeeperServerMain.runFromConfig(ZooKeeperServerMain.java:133)
    at org.apache.zookeeper.server.ZooKeeperServerMain.initializeAndRun(ZooKeeperServerMain.java:113)
    at org.apache.zookeeper.server.ZooKeeperServerMain.main(ZooKeeperServerMain.java:68)
    at org.apache.zookeeper.server.quorum.QuorumPeerMain.initializeAndRun(QuorumPeerMain.java:141)
    at org.apache.zookeeper.server.quorum.QuorumPeerMain.main(QuorumPeerMain.java:91)
Caused by: java.lang.ClassNotFoundException: com.codahale.metrics.Reservoir
    at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
```

```
import org.apache.zookeeper.data.ACL;
import org.apache.zookeeper.data.Id;
ACL和Id缺失，需要重新编译zookeeper-jute
```

```
java.lang.BootstrapMethodError: java.lang.IllegalAccessError: no such constructor: org.apache.zookeeper.cli.CreateCommand.<init>()
void/newInvokeSpecial
    at org.apache.zookeeper.cli.CommandFactory$Command.<clinit>(CommandFactory.java:33)
    at org.apache.zookeeper.ZooKeeperMain.<clinit>(ZooKeeperMain.java:82)
Caused by: java.lang.IllegalAccessError: no such constructor: org.apache.zookeeper.cli.CreateCommand.<init>() void/newInvokeSpecial
    at java.lang.invoke.MethodHandleNatives.linkMethodHandleConstant(MethodHandleNatives.java:483)
    ... 2 more
Caused by: java.lang.NoClassDefFoundError: org/apache/commons/cli/ParseException
    at java.lang.invoke.MethodHandleNatives.resolve(Native Method)
    at java.lang.invoke.MemberName$Factory.resolve(MemberName.java:977)
    at java.lang.invoke.MemberName$Factory.resolveOrFail(MemberName.java:1002)
    at java.lang.invoke.MethodHandles$Lookup.resolveOrFail(MethodHandles.java:1390)
    at java.lang.invoke.MethodHandles$Lookup.linkMethodHandleConstant(MethodHandles.java:1746)
    at java.lang.invoke.MethodHandleNatives.linkMethodHandleConstant(MethodHandleNatives.java:477)
```

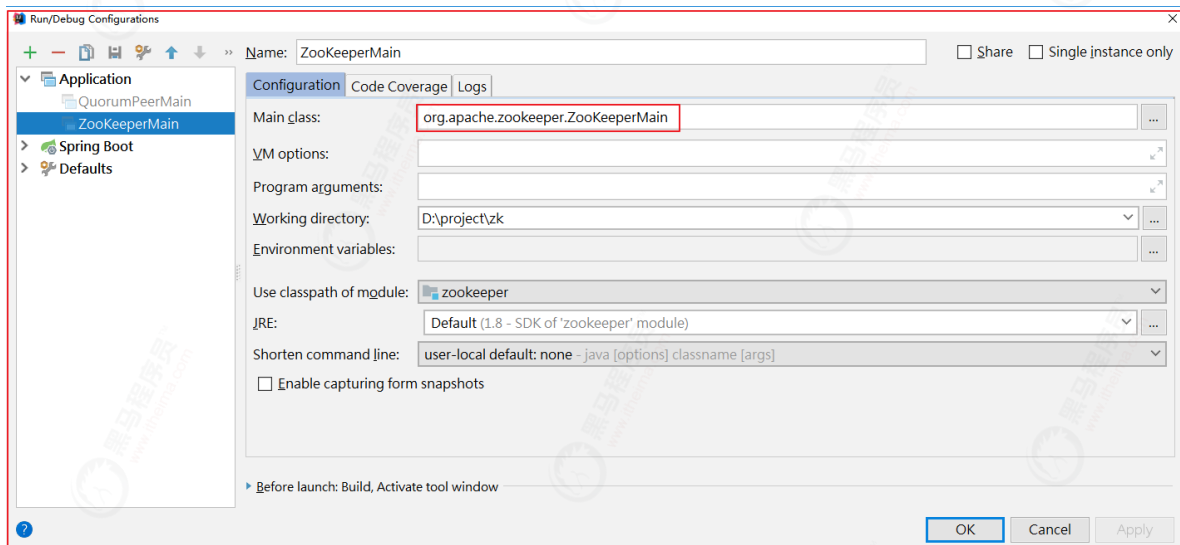
为了解决上面的错误，我们需要手动引入一些包，pom.xml引入如下依赖：

```
<!--引入依赖-->
<dependency>
    <groupId>io.dropwizard.metrics</groupId>
    <artifactId>metrics-core</artifactId>
    <version>3.1.0</version>
</dependency>
<dependency>
    <groupId>org.xerial.snappy</groupId>
    <artifactId>snappy-java</artifactId>
    <version>1.1.7.3</version>
</dependency>
<dependency>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-server</artifactId>
</dependency>
<dependency>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-servlet</artifactId>
</dependency>
<dependency>
    <groupId>commons-cli</groupId>
    <artifactId>commons-cli</artifactId>
</dependency>
```

1.3 Zookeeper命令(自学)

我们要想学习Zookeeper，需要先学会使用Zookeeper，它有很多丰富的命令，借助这些命令可以深入理解Zookeeper，我们启动源码中的客户端就可以使用Zookeeper相关命令。

启动客户端 `org.apache.zookeeper.ZooKeeperMain`，如下图：



启动后，日志如下：

```
WATCHER: :  
  
WatchedEvent state:SyncConnected type:None path:null  
[dubbo, zookeeper]
```

1)节点列表: `ls /`

```
ls /  
[dubbo, zookeeper]  
ls /dubbo  
[com.itheima.service.CarService]
```

2)查看节点状态: `stat /dubbo`

```
stat /dubbo  
cZxid = 0x3  
ctime = Thu Dec 03 09:19:29 CST 2020  
mZxid = 0x3  
mtime = Thu Dec 03 09:19:29 CST 2020  
pZxid = 0x4  
cversion = 1  
dataVersion = 0  
aclVersion = 0  
ephemeralOwner = 0x0  
dataLength = 13  
numChildren = 1
```

节点信息参数说明如下：

key	value
<code>czxid = 0x3</code>	节点被创建时的事物的ID
<code>ctime = Thu Dec 03 09:19:29 CST 2020</code>	创建时间
<code>mzxid = 0x31</code>	节点最后一次被修改时的事物的ID
<code>mtime = Sat Mar 16 15:38:34 CST 2019</code>	最后一次修改时间
<code>pzxid = 0x31</code>	子节点列表最近一次被修改的事物ID
<code>cversion = 0</code>	子节点版本号
<code>dataVersion = 0</code>	数据版本号
<code>aclVersion = 0</code>	ACL版本号
<code>ephemeralOwner = 0x0</code>	创建临时节点的事物ID，持久节点事物为0
<code>dataLength = 22</code>	数据长度，每个节点都可保存数据
<code>numChildren = 0</code>	子节点的个数

3)创建节点: `create /dubbo/code java`

```
create /dubbo/code java
Created /dubbo/code
```

其中code表示节点，java表示节点下的内容。

4)查看节点数据: `get /dubbo/code`

```
get /dubbo/code
java
```

5)删除节点: `delete /dubbo/code || deleteall /dubbo/code`

删除没有子节点的节点：`delete /dubbo/code`

删除所有子节点：`deleteall /dubbo/code`

6)历史操作命令: `history`

```
history
1 - ls /dubbo
2 - ls /dubbo/code
3 - get /dubbo/code
4 - get /dubbo/code
5 - create /dubbo/code java
6 - get /dubbo/code
7 - get /dubbo/code
8 - delete /dubbo/code
9 - get /dubbo/code
10 - listquota path
11 - history
```

1.4 Zookeeper分析工具

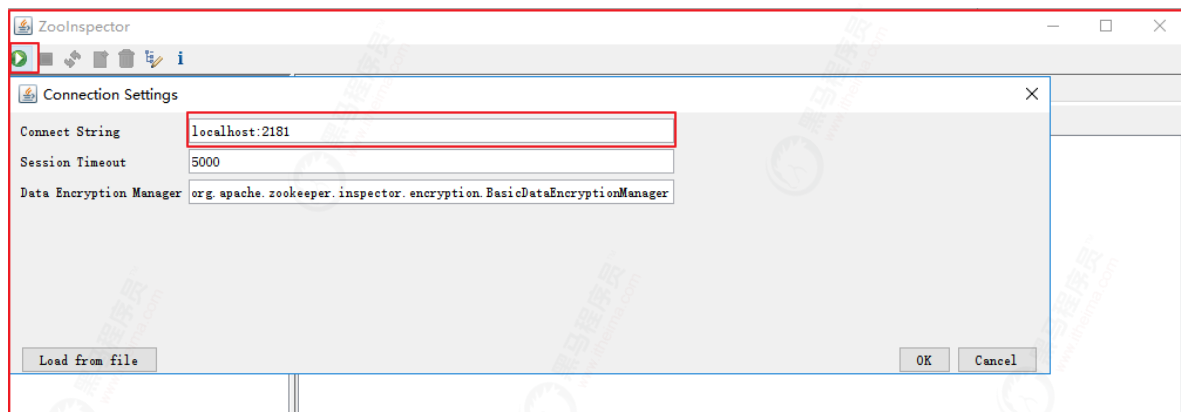
Zookeeper安装比较方便，在安装一个集群以后，查看数据却比较麻烦，下面介绍Zookeeper的数据查看工具——ZooInspector。

下载地址：<https://issues.apache.org/jira/secure/attachment/12436620/ZooInspector.zip>

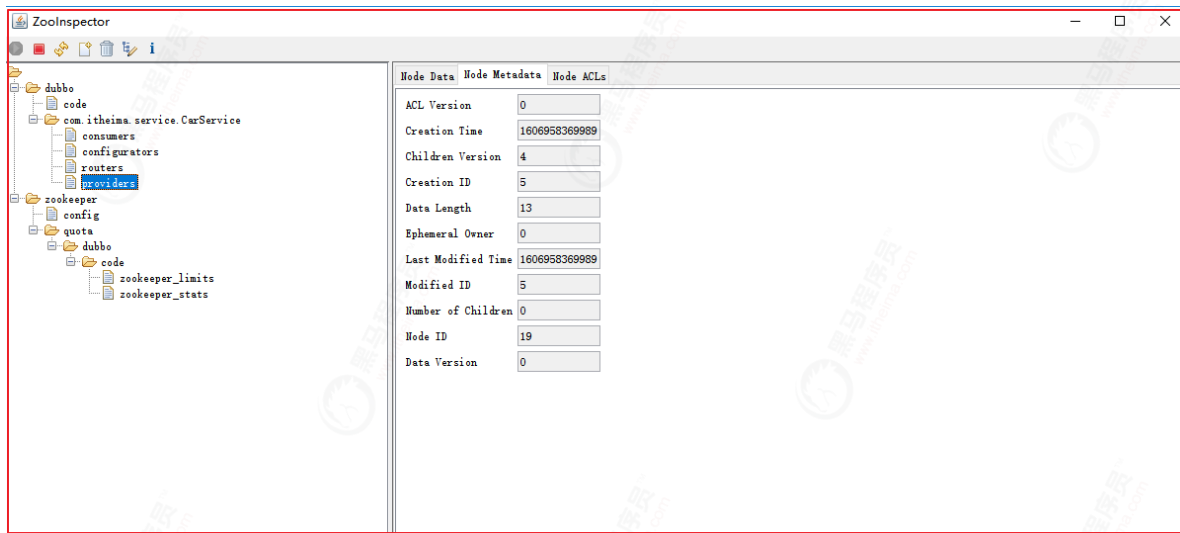
下载压缩包后，解压后，我们需要运行 `zookeeper-dev-ZooInspector.jar`：



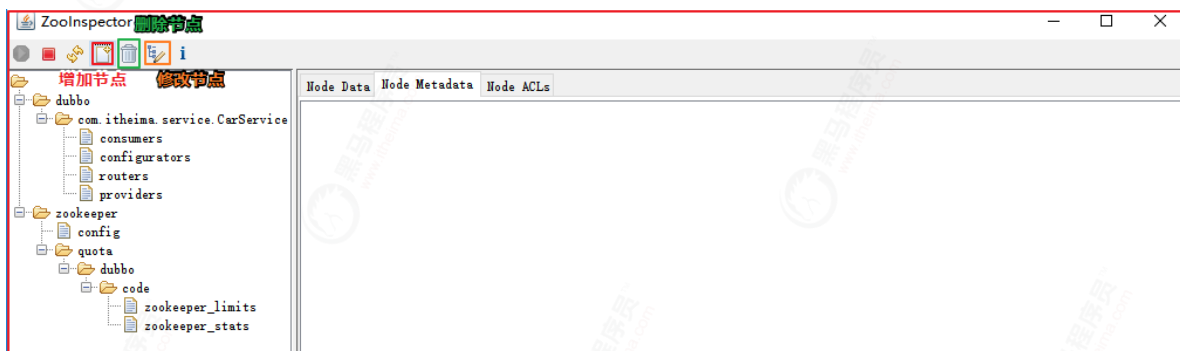
输入账号密码，就可以连接Zookeeper了，如下图：



连接后，Zookeeper信息如下：



节点操作：增加节点、修改节点、删除节点

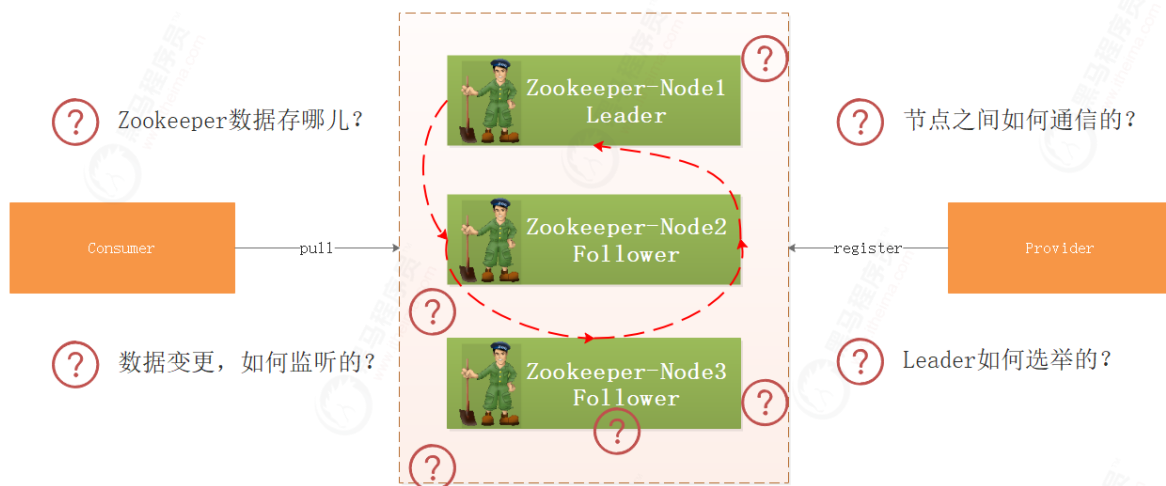


1.5 Zookeeper案例应用



我们将资料中 工程\dubbo 工程导入到IDEA中，上图是他们的调用关系，那么问题来了：

- 生产者向Zookeeper注册服务信息，Zookeeper把数据存哪儿了？
- 集群环境下，如果某个节点数据变更了，Zookeeper如何监听到的？
- 集群环境下各个节点的数据如何同步？
- 如果某个节点挂了，Zookeeper如何选举呢？
-



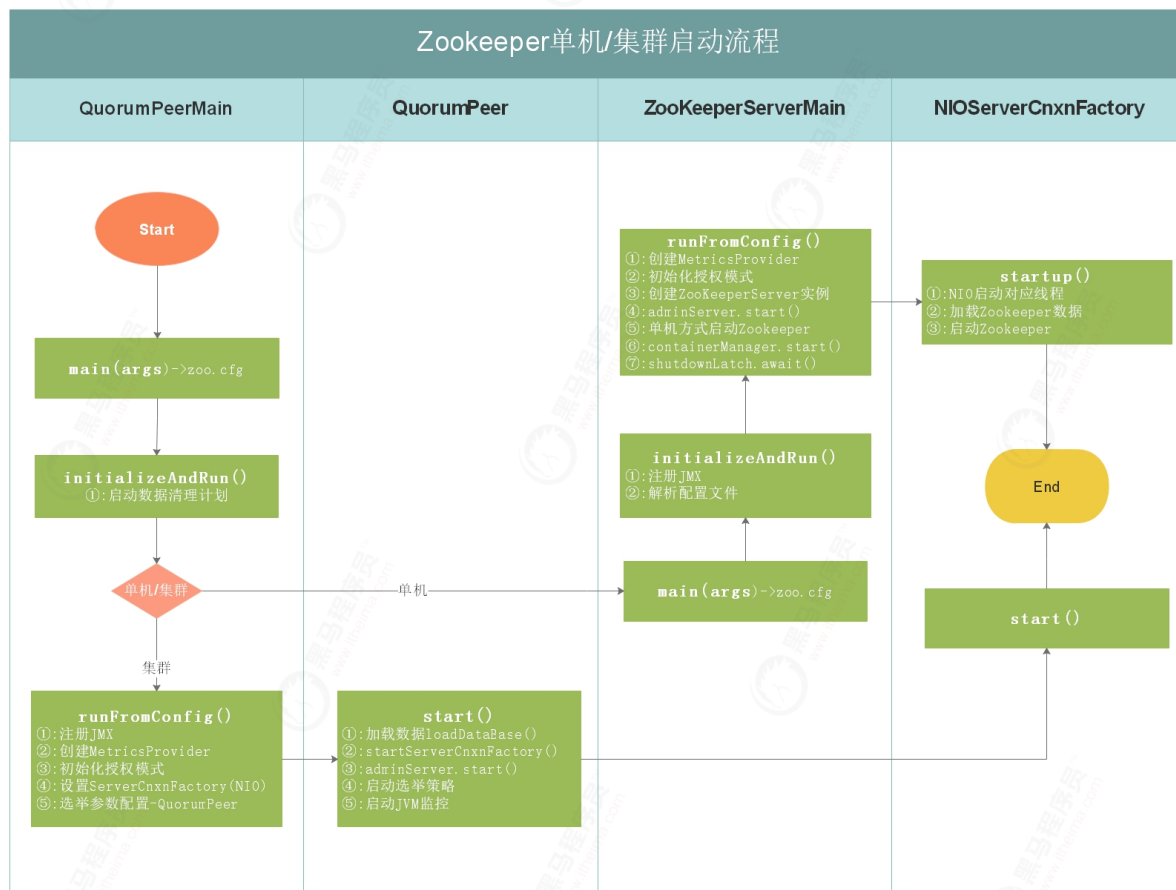
带着上面的疑问, 我们开始研究Zookeeper源码。

2 ZK服务启动流程源码剖析

Zookeeper 可以以 standalone、分布式的方式部署, standalone 模式下只有一台机器作为服务器, Zookeeper 会丧失高可用特性, 分布式是使用多个机器, 每台机器上部署一个 zookeeper 服务器, 即使有服务器宕机, 只要少于半数, zookeeper 集群依然可以正常对外提供服务, 集群状态下 zookeeper 是具备高可用特性。

我们接下来对 zookeeper 以 standalone 模式启动以及集群模式做一下源码分析。

2.1 ZK单机/集群启动流程



如上图，上图是 zookeeper 单机/集群启动流程，每个细节所做的事情都在上图有说明，我们接下来按照流程图对源码进行分析。

2.2 ZK启动入口分析

启动入口类：QuorumPeerMain

该类是 zookeeper 单机/集群的启动入口类，是用来加载配置、启动 QuorumPeer (选举相关)线程、创建 ServerCnxnFactory 等，我们可以把代码切换到该类的主方法(main)中，从该类的主方法开始分析，main 方法代码分析如下：

```
public static void main(String[] args) {
    QuorumPeerMain main = new QuorumPeerMain();
    try {
        //初始化配置，并启动服务
        main.initializeAndRun(args);
    } catch (Exception e) {
        //发生异常
    }
    //退出
    ServiceUtils.requestSystemExit(ExitCode.EXECUTION_FINISHED.getValue());
}
```

上面main方法虽然只是做了初始化配置，但调用了 initializeAndRun() 方法，initializeAndRun() 方法中会根据配置来决定启动单机Zookeeper还是集群Zookeeper，源码如下：

```
//初始化并启动ZK服务
protected void initializeAndRun(String[] args) throws ConfigException, IOException, AdminServerException {
    QuorumPeerConfig config = new QuorumPeerConfig();
    if (args.length == 1) {
        config.parse(args[0]);
    }

    // 启动数据清理计划
    DatadirCleanupManager purgeMgr = new DatadirCleanupManager(
        config.getDataDir(), //数据存储目录
        config.getDataLogDir(), //事务日志目录，默认为dataDir
        config.getSnapRetainCount(), //保留多少个快照数据，默认为3个
        config.getPurgeInterval()); //多少小时清理1次数据，默认不清理，在zoo.cfg中autopurge.purgeInterval=1设置时间
    purgeMgr.start();

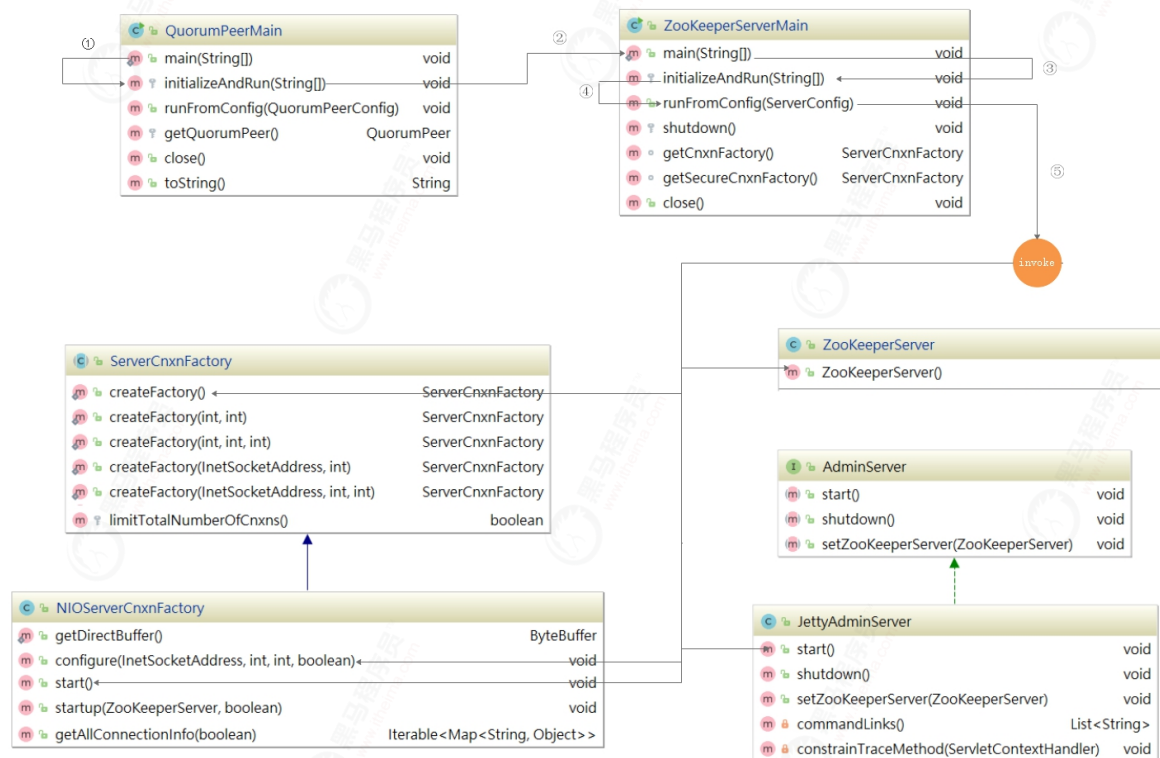
    //集群模式运行
    if (args.length == 1 && config.isDistributed()) {
        runFromConfig(config);
    } else {
        //单机模式运行
        ZooKeeperServerMain.main(args);
    }
}
```

启动单机还是集群

如果启动单机版，会调用 ZooKeeperServerMain.main(args);，如果启动集群版，会调用 QuorumPeerMain.runFromConfig(config);，我们接下来对单机版启动做源码详细剖析，集群版在后面章节中讲解选举机制时详细讲解。

2.3 ZK单机启动源码剖析

针对ZK单机启动源码方法调用链，我们已经提前做了一个方法调用关系图，我们讲解ZK单机启动源码，将和该图进行一一一对，如下图：



1)单机启动入口

按照上面的源码分析，我们找到 `zooKeeperServerMain.main(args)` 方法，该方法调用了 `ZooKeeperServerMain` 的 `initializeAndRun` 方法，在 `initializeAndRun` 方法中执行初始化操作，并运行Zookeeper服务，main方法如下：

```
/*
 * 启动Zookeeper服务
 * @param args the configfile or the port datadir [ticktime]
 */
public static void main(String[] args) {
    ZooKeeperServerMain main = new ZooKeeperServerMain();
    try {
        main.initializeAndRun(args);
    } catch (Exception e) {
        //发生异常
    }
    //正常退出
    ServiceUtils.requestSystemExit(ExitCode.EXECUTION_FINISHED.getValue());
}
```

2)配置文件解析

`initializeAndRun()` 方法会注册JMX，同时解析 `zoo.cfg` 配置文件，并调用 `runFromConfig()` 方法启动Zookeeper服务，源码如下：

//初始化配置，并运行Zookeeper服务

```
protected void initializeAndRun(String[] args) throws ConfigException, IOException, AdminServerException {  
    //注册JMX  
    try {  
        ManagedUtil.registerLog4jMBeans();  
    } catch (JMXException e) {  
        LOG.warn("Unable to register log4j JMX control", e);  
    }  
  
    //解析配置文件  
    ServerConfig config = new ServerConfig();  
    if (args.length == 1) {  
        config.parse(args[0]);  
    } else {  
        config.parse(args);  
    }  
  
    //根据解析的配置运行Zookeeper服务  
    runFromConfig(config);  
}
```

JMX (Java Management Extensions, 即Java管理扩展) 是一个为应用程序、设备、系统等植入管理功能的框架。JMX可以跨越一系列异构操作系统平台、系统体系结构和网络传输协议, 灵活的开发无缝集成的系统、网络和服务管理应用。

解析配置文件

根据配置启动服务

3)单机启动主流程

runFromConfig 方法是单机版启动的主要方法, 该方法会做如下几件事:

- 1: 初始化各类运行指标, 比如一次提交数据最大花费多长时间、批量同步数据大小等。
- 2: 初始化权限操作, 例如IP权限、Digest权限。
- 3: 创建事务日志操作对象, Zookeeper中每次增加节点、修改数据、删除数据都是一次事务操作, 都会记录日志。
- 4: 定义Jvm监控变量和常量, 例如警告时间、告警阈值次数、提示阈值次数等。
- 5: 创建ZookeeperServer, 这里只是创建, 并不在ZookeeperServerMain类中启动。
- 6: 启动Zookeeper的控制台管理对象AdminServer, 该对象采用Jetty启动。
- 7: 创建ServerCnxnFactory, 该对象其实是Zookeeper网络通信对象, 默认使用了NIOServerCnxnFactory。
- 8: 在ServerCnxnFactory中启动ZookeeperServer服务。
- 9: 创建并启动ContainerManager, 该对象通过Timer定时执行, 清理过期的容器节点和TTL节点, 执行周期为分钟。
- 10: 防止主线程结束, 阻塞主线程。

方法源码如下:

```

/**
 * 根据解析的配置运行Zookeeper服务
 * @param config 当前使用的配置
 * @throws IOException
 * @throws AdminServerException
 */
public void runFromConfig(ServerConfig config) throws IOException, AdminServerException {
    FileTxnSnapLog txnLog = null;
    try {
        try {
            //创建MetricsProvider, 它是一个收集指标并将当前值发布到外部设施的系统, 与选举有关。
            metricsProvider = MetricsProviderBootstrap.startMetricsProvider(
                config.getMetricsProviderClassName(),
                config.getMetricsProviderConfiguration());
        } catch (MetricsProviderLifecycleException error) {
            throw new IOException("Cannot boot MetricsProvider " + config.getMetricsProviderClassName(), error);
        }
        ServerMetrics.metricsProviderInitialized(metricsProvider); 各类运行指标收集器
        ProviderRegistry.initialize(); 初始化权限
        // 根据参数创建日志目录(事务日志目录、快照日志目录)
        txnLog = new FileTxnSnapLog(config.dataLogDir, config.dataDir); 创建事务操作日志对象
        JvmPauseMonitor jvmPauseMonitor = null;
        if (config.jvmPauseMonitorToRun) {
            jvmPauseMonitor = new JvmPauseMonitor(config);
        }

        //创建ZookeeperServer服务实例, 并初始化参数
        final ZooKeeperServer zkServer = new ZooKeeperServer(jvmPauseMonitor, txnLog, config.tickTime, config.minSessionTimeout,
            txnLog.getServerStats(zkServer.serverStats())); 创建ZookeeperServer服务, 并未启动该服务

        //注册服务关闭程序
        final CountDownLatch shutdownLatch = new CountDownLatch(1);
        zkServer.registerServerShutdownHandler(new ZooKeeperServerShutdownHandler(shutdownLatch));

        // 启动服务管理器(JettyAdminServer)-启动Jetty
        adminServer = AdminServerFactory.createAdminServer(); 启动ZookeeperServer服务管理对象, 这里启动了Jetty组件
        adminServer.setZooKeeperServer(zkServer);
        adminServer.start();

        boolean needStartZKServer = true;
        if (config.getClientPortAddress() != null) {
            //cnxnFactory负责zk的网络请求, createFactory中
            //从系统配置中读取ZOOKEEPER_SERVER_CNXX_FACTORY, 默认是没有这个配置的, 因此默认是使用NIOServerCnxnFactory, 如果配置了, 则使用配置中的
            cnxnFactory = ServerCnxnFactory.createFactory();
            //0.0.0.0/0.0.0.0:2181、单个客户端连接数超过限制、请求的传入连接队列的最大长度, -1不限制
            cnxnFactory.configure(config.getClientPortAddress(), config.getMaxClientCnxns(), config.getClientPortListenQueueSize);
            //启动Zookeeper服务 重点: 创建了ServerCnxnFactory, 它是ZK通信组件, 默认使用了NIO, 也可以使用Netty
            cnxnFactory.startup(zkServer); 在此处会启动ZookeeperServer服务
            // 是否需要启动Zookeeper服务, zookeeper服务已经启动了, 不需要再次启动, 设置为false
            needStartZKServer = false;
        }

        //采用了SSL(我们没有使用, 不做讲解)
        if (config.getSecureClientPortAddress() != null) {
            secureCnxnFactory = ServerCnxnFactory.createFactory();
            secureCnxnFactory.configure(config.getSecureClientPortAddress(), config.getMaxClientCnxns(), config.getClientPortListenQueueSize);
            secureCnxnFactory.startup(zkServer, needStartZKServer);
        }

        //创建ContainerManager, 它只由Leader调用, 通过Timer定时执行, 清理过期的容器节点和TTL节点, 执行周期为分钟级别
        containerManager = new ContainerManager(
            zkServer.getZKDatabase(),
            zkServer.firstProcessor,
            Integer.getInteger(nm: "znode.container.checkIntervalMs", (int) TimeUnit.MINUTES.toMillis(duration: 1)),
            Integer.getInteger(nm: "znode.container.maxPerMinute", val: 10000),
            Long.getLong(nm: "znode.container.maxNeverUsedIntervalMs", val: 0)
        );
        containerManager.start();

        //为服务器启动和注册服务器停止日志添加审计日志。
        ZKAuditProvider.addZKStartStopAuditLog();

        //服务器正常启动时, 运行到此处阻塞, 只有server的state变为ERROR或SHUTDOWN时继续运行后面的代码
        shutdownLatch.await();
    }
}

```

```
shutdown();
```

4)网络通信对象创建

上面方法在创建网络通信对象的时候调用了 `ServerCnxnFactory.createFactory()` ,该方法其实是根据系统配置创建Zookeeper通信组件, 可选的有 `NIOServerCnxnFactory` (默认) 和 `NettyServerCnxnFactory` , 关于通信对象我们会在后面进行详细讲解, 该方法源码如下:

```
public static ServerCnxnFactory createFactory() throws IOException {
    String serverCnxnFactoryName = System.getProperty("ZOOKEEPER_SERVER_CNXN_FACTORY");
    if (serverCnxnFactoryName == null) {
        //网络传输采用了NIO
        serverCnxnFactoryName = "NIOServerCnxnFactory.class.getName()";
    }
    try {
        //创建NIOServerCnxnFactory
        ServerCnxnFactory serverCnxnFactory = (ServerCnxnFactory) Class.forName(serverCnxnFactoryName)
            .getDeclaredConstructor()
            .newInstance();
        LOG.info("Using {} as server connection factory", serverCnxnFactoryName);
        return serverCnxnFactory;
    } catch (Exception e) {
        IOException ioe = new IOException("Couldn't instantiate " + serverCnxnFactoryName, e);
        throw ioe;
    }
}
```

系统变量中配置该参数可改变通信对象, 默认为NIO, 可选Netty

NIO为默认通信对象

创建NIOServerCnxnFactory实例

5)单机启动

`cnxnFactory.startup(zkServer);` 方法其实就是启动了 `zookeeperServer` , 它调用 `NIOServerCnxnFactory` 的 `startup` 方法, 该方法中会调用 `zookeeperServer` 的 `startup` 方法启动服务, `ZookeeperServerMain` 运行到 `shutdownLatch.await()`; 主线程会阻塞住, 源码如下:

```
@Override
public void startup(ZooKeeperServer zks, boolean startServer) throws IOException, InterruptedException {
    //NIO启动对应线程
    start();
    //设置Zookeeper的ServerCnxnFactory (客户端与服务端进行通信的对象, 就是当前对象NIOServerCnxnFactory)
    setZooKeeperServer(zks);
    if (startServer) {
        //加载会话和数据
        zks.startdata();
        //启动Zookeeper服务
        zks.startup();
    }
}
```

启动服务

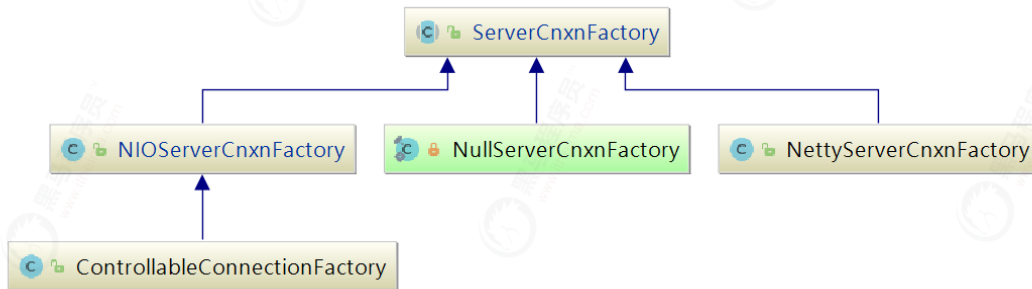
启动后, 日志如下:

```
INFO [main:ZKDatabase@290] - Snapshot loaded in 21 ms, highest zxid is 0x6a, digest is 13700439064
INFO [main:FileTxnSnapLog@470] - Snapshotting: 0x6a to D:\project\zk\zookeeper\data\zkdata\version-2\snapshot
INFO [main:ZooKeeperServer@530] - Snapshot taken in 4 ms
INFO [ProcessThread(sid:0 cport:2181)::PrepRequestProcessor@137] - PrepRequestProcessor (sid:0) started,
INFO [main:RequestThrottler@75] - zookeeper.request_throttler.shutdownTimeout = 10000
INFO [main:ContainerManager@84] - Using checkIntervalMs=60000 maxPerMinute=10000 maxNeverUsedIntervalMs=0
```

3 ZK网络通信源码剖析

zookeeper 作为一个服务器,自然要与客户端进行网络通信,如何高效的与客户端进行通信,让网络 io 不成为 zookeeper 的瓶颈是 zookeeper 急需解决的问题, zookeeper 中使用 `ServerCnxnFactory` 管理与客户端的连接,其有两个实现,一个是 `NIOServerCnxnFactory`,使用java原生 NIO 实现;一个是 `NettyServerCnxnFactory`,使用netty实现;使用 `ServerCnxn` 代表一个客户端与服务端的连接。

从单机版启动中可以发现 zookeeper 默认通信组件为 `NIOServerCnxnFactory`, 他们和 `ServerCnxnFactory` 的关系如下图:



3.1 NIOServerCnxnFactory工作流程

一般使用Java NIO的思路为使用1个线程组监听 `OP_ACCEPT` 事件,负责处理客户端的连接;使用1个线程组监听客户端连接的 `OP_READ` 和 `OP_WRITE` 事件,处理IO事件(netty也是这种实现方式). 但ZooKeeper并不是如此划分线程功能的, `NIOServerCnxnFactory` 启动时会启动四类线程:

- 1: **accept thread**: 该线程接收来自客户端的连接,并将其分配给 **selector thread**(启动一个线程)。
- 2: **selector thread**: 该线程执行 `select()`, 由于在处理大量连接时, `select()` 会成为性能瓶颈, 因此启动多个 **selector thread**, 使用系统属性 `zookeeper.nio.numSelectorThreads` 配置该类线程数, 默认个数为 核心数/2。
- 3: **worker thread**: 该线程执行基本的套接字读写, 使用系统属性 `zookeeper.nio.numWorkerThreads` 配置该类线程数, 默认为核心数*2核心数*2. 如果该类线程数为0, 则另外启动一线程进行IO处理, 见下文 **worker thread** 介绍。
- 4: **connection expiration thread**: 若连接上的 **session** 已过期, 则关闭该连接。

这四个线程在 `NIOServerCnxnFactory` 类上有说明, 如下图:

```

/**
 * NIOServerCnxnFactory implements a multi-threaded ServerCnxnFactory using
 * NIO non-blocking socket calls. Communication between threads is handled via
 * queues.
 *
 * - 1 accept thread, which accepts new connections and assigns to a
 *   selector thread
 * - 1-N selector threads, each of which selects on 1/N of the connections.
 *   The reason the factory supports more than one selector thread is that
 *   with large numbers of connections, select() itself can become a
 *   performance bottleneck.
 * - 0-M socket I/O worker threads, which perform basic socket reads and
 *   writes. If configured with 0 worker threads, the selector threads
 *   do the socket I/O directly.
 * - 1 connection expiration thread, which closes idle connections; this is
 *   necessary to expire connections on which no session is established.
 *
 * Typical (default) thread counts are: on a 32 core machine, 1 accept thread,
 * 1 connection expiration thread, 4 selector threads, and 64 worker threads.
 */
public class NIOServerCnxnFactory extends ServerCnxnFactory {

```

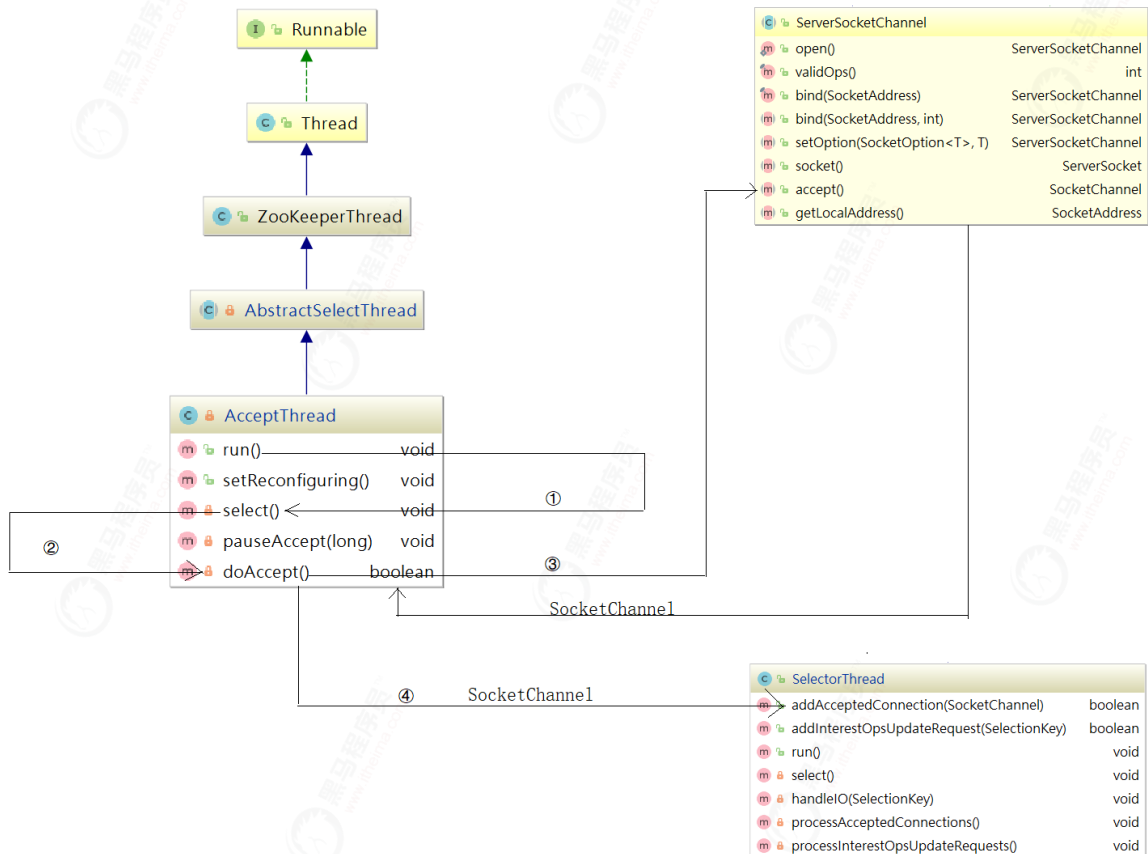
zookeeper 中对线程需要处理的工作做了更细的拆分,解决了有大量客户端连接的情况下,selector.select() 会成为性能瓶颈,将 selector.select() 拆分出来,交由 selector thread 处理。

3.2 NIOServerCnxnFactory源码

NIOServerCnxnFactory的源码分析我们将按照上面所介绍的4个线程实现相关分析,并实现数据操作,在程序中获取指定数据。

3.2.1 AcceptThread剖析

为了让大家更容易理解AcceptThread,我们把它的结构和方法调用关系画了一个详细的流程图,如下图:



在 `NIOServerCnxnFactory` 类中有一个 `AcceptThread` 线程，为什么说它是一个线程？我们看下它的继承关系：`AcceptThread > AbstractSelectThread > ZooKeeperThread > Thread`，该线程接收来自客户端的连接，并将其分配给 selector thread (启动一个线程)。

该线程执行流程：`run` 执行 `selector.select()`，并调用 `doAccept()` 接收客户端连接，因此我们可以着重关注 `doAccept()` 方法，该类源码如下：

```

private class AcceptThread extends AbstractSelectThread {

    private final ServerSocketChannel acceptSocket;
    private final SelectionKey acceptKey;
    private final RateLogger acceptErrorLogger = new RateLogger(LOG);
    private final Collection<SelectorThread> selectorThreads;
    private Iterator<SelectorThread> selectorIterator;
    private volatile boolean reconfiguring = false;

    public AcceptThread(ServerSocketChannel ss, InetSocketAddress addr, Set<SelectorThread> selectorThreads) throws

    public void run() {...}

    public void setReconfiguring() { reconfiguring = true; }

    private void select() {...}

    /**...*/
    private void pauseAccept(long millisecs) {...}

    /**...*/
    private boolean doAccept() {...}
}
  
```

接收新的socket连接, 每个IP地址有其连接个数上限. 使用轮询为新连接分配 selector thread

`doAccept()` 方法用于处理客户端链接，当客户端链接 zookeeper 的时候，首先会调用该方法，调用该方法执行过程如下：

- 1: 和当前服务建立链接。
- 2: 获取远程客户端计算机地址信息。
- 3: 判断当前链接是否超出最大限制。
- 4: 调整为非阻塞模式。
- 5: 轮询获取一个SelectorThread, 将当前链接分配给该SelectorThread。
- 6: 将当前请求添加到该SelectorThread的acceptedQueue中, 并唤醒该SelectorThread。

doAccept() 方法源码如下：

```
/**
 * 接收新的socket连接, 每个IP地址有其连接个数上限, 使用轮询为新连接分配selector thread
 */
private boolean doAccept() {
    boolean accepted = false;
    SocketChannel sc = null;
    try {
        //建立链接
        sc = acceptSocket.accept();
        accepted = true;
        if (limitTotalNumberOfCnxns()) {
            throw new IOException("Too many connections max allowed is " + maxCnxns);
        }
        //获取远程计算机地址信息
        InetAddress ia = sc.socket().getInetAddress();
        System.out.println("AcceptThread-----链接服务的IP: "+ia.getHostName());
        int cnxncount = getClientCnxnCount(ia);

        //判断是否超出最大客户端连接的限制
        if (maxClientCnxns > 0 && cnxncount >= maxClientCnxns) {
            throw new IOException("Too many connections from " + ia + " - max is " + maxClientCnxns);
        }
        //调整此通道的阻塞模式。
        sc.configureBlocking(false);

        // 轮询将此连接分配给一个selector thread
        if (!selectorIterator.hasNext()) {
            selectorIterator = selectorThreads.iterator();
        }
        SelectorThread selectorThread = selectorIterator.next();
        //将新连接加入selector thread 的acceptedQueue中, 并唤醒SelectorThread
        if (!selectorThread.addAcceptedConnection(sc)) {
            throw new IOException("Unable to add connection to selector queue"
                + (stopped ? " (shutdown in progress)" : ""));
        }
        acceptErrorLogger.flush();
    } catch (IOException e) {
        // accept, maxClientCnxns, configureBlocking
        ServerMetrics.getMetrics().CONNECTION_REJECTED.add(1);
        acceptErrorLogger.rateLimitLog(newMsg: "Error accepting new connection: " + e.getMessage());
        fastCloseSock(sc);
    }
    return accepted;
}
```

上面代码中 addAcceptedConnection 方法如下：

```

/**
 * 将新的请求添加到acceptedQueue，并唤醒SelectorThread
 */
public boolean addAcceptedConnection(SocketChannel accepted) {
    //将accepted添加到acceptedQueue
    if (stopped || !acceptedQueue.offer(accepted)) {
        return false;
    }
    //唤醒SelectorThread
    wakeupSelector();
    return true;
}

public void wakeupSelector() {
    selector.wakeup();
}

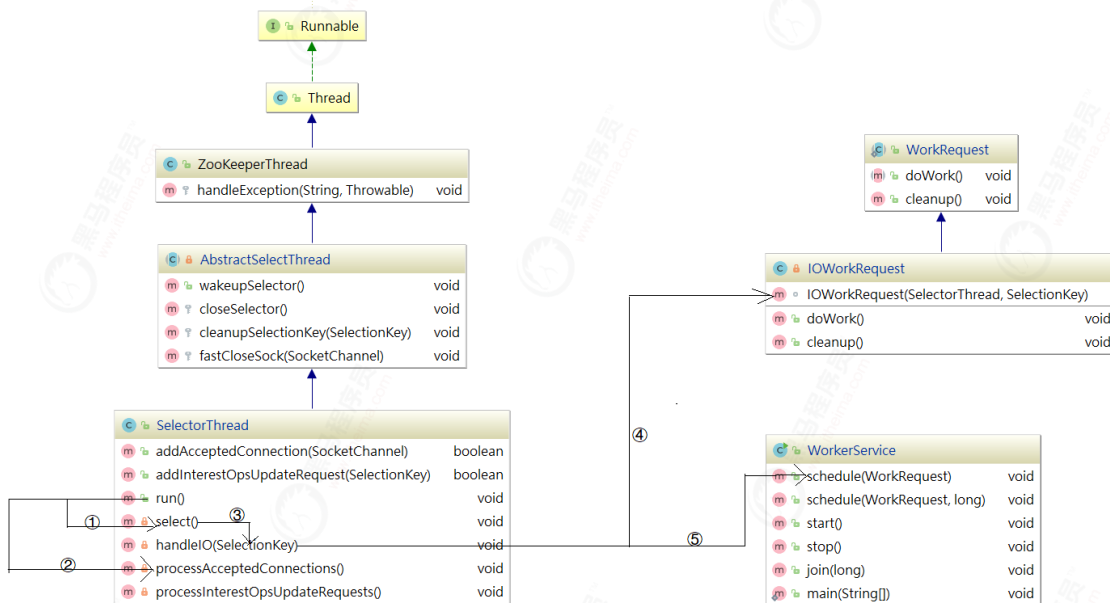
```

我们把项目中的分布式案例服务启动，可以看到如下日志打印：

AcceptThread-----链接服务的IP: 127.0.0.1

3.2.2 SelectorThread剖析

同样为了更容易梳理 SelectorThread，我们也把它的结构和方法调用关系梳理成了流程图，如下图：



该线程的主要作用是从Socket读取数据，并封装成 workRequest，并将 workRequest 交给 workerPool 工作线程池处理，同时将 acceptedQueue 中未处理的链接取出，并为每个链接绑定 OP_READ 读事件，并封装对应的上下文对象 NIOServerCnxn。SelectorThread 的 run 方法如下：

```

public void run() {
    try {
        while (!stopped) {
            try {
                //调用select() 读取就绪的IO事件, 交由worker thread处理, 在ZookeeperServer的processPacket() 中处理数据
                select();
                //把acceptedQueue队列中接收的连接, 取出来注册OP_READ读事件,
                //并添加NIOServerCnxn对象与当前key绑定。
                //相当于给每个连接添加附加对象NIOServerCnxn (上下文对象)
                processAcceptedConnections();
                //遍历所有updateQueue, 更新updateQueue中连接的监听事件
                processInterestOpsUpdateRequests();
            } catch (RuntimeException e) {
                LOG.warn("Ignoring unexpected runtime exception", e);
            } catch (Exception e) {
                LOG.warn("Ignoring unexpected exception", e);
            }
        }
    }
}

```

关键

其他未处理的链接封装NIO上下文和注册读写事件

run() 方法中会调用 select(), 而 select() 中的核心调用地方是 handleIO(), 我们看名字其实就知道这里是处理客户端请求的数据, 但客户端请求数据并非在 selectorThread 线程中处理, 我们接着看 handleIO() 方法。

```

private void select() {
    try {
        selector.select();
        Set<SelectionKey> selected = selector.selectedKeys();
        ArrayList<SelectionKey> selectedList = new ArrayList<>(selected);
        Collections.shuffle(selectedList);
        Iterator<SelectionKey> selectedKeys = selectedList.iterator();
        while (!stopped && selectedKeys.hasNext()) {
            SelectionKey key = selectedKeys.next();
            selected.remove(key);

            if (!key.isValid()) {
                cleanupSelectionKey(key);
                continue;
            }
            //处理和key对应的数据
            if (key.isReadable() || key.isWritable()) {
                handleIO(key);
            } else {
                LOG.warn("Unexpected ops in select {}", key.readyOps());
            }
        }
    } catch (IOException e) {
        LOG.warn("Ignoring IOException while selecting", e);
    }
}

```

该方法读取客户端数据请求的开始

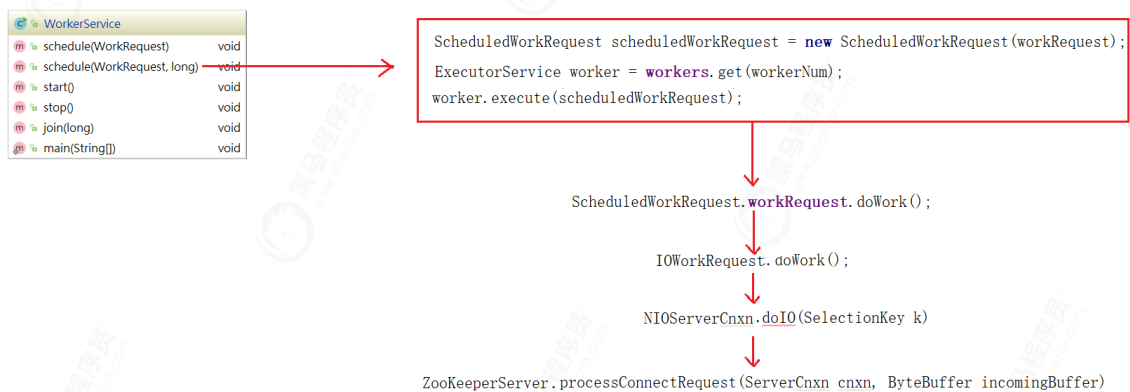
handleIO() 方法会封装当前 SelectorThread 为 IOWorkRequest, 并将 IOWorkRequest 交给 workerPool 来调度, 而 workerPool 调度才是读数据的开始, 源码如下:

```
private void handleIO(SelectionKey key) {
    IOWorkRequest workRequest = new IOWorkRequest(selectorThread: this, key); 将SelectorThread封装成workRequest
    NIOServerCnxn cnxn = (NIOServerCnxn) key.attachment();

    // Stop selecting this key while processing on its
    // connection
    cnxn.disableSelectable();
    key.interestOps(0);
    touchCnxn(cnxn);
    //线程池中获取
    workerPool.schedule(workRequest); 将封装好的workRequest交给workerPool线程池处理, 在这里会读取客户端数据
}
}
```

3.2.3 WorkerThread剖析

WorkerThread相比上面的线程而言, 调用关系颇为复杂, 设计到了多个对象方法调用, 主要用于处理IO, 但并未对数据做出处理, 数据处理将有业务链对象RequestProcessor处理, 调用关系图如下:



ZooKeeper 中通过 WorkerService 管理一组 worker thread 线程, 前面我们在看 SelectorThread 的时候, 能够看到 workerPool 的 schedule 方法被执行, 如下图:

```
private void handleIO(SelectionKey key) {
    IOWorkRequest workRequest = new IOWorkRequest(selectorThread: this, key);
    NIOServerCnxn cnxn = (NIOServerCnxn) key.attachment();

    // Stop selecting this key while processing on its
    // connection
    //不处理当前链接的其他请求
    cnxn.disableSelectable();
    key.interestOps(0);
    touchCnxn(cnxn);
    //线程池中获取
    workerPool.schedule(workRequest);
}
}
```

我们跟踪 workerPool.schedule(workRequest); 可以发现它调用了 workerService.schedule(workRequest) > WorkerService.schedule(workRequest, long), 该方法创建了一个新的线程 ScheduledWorkRequest, 并启动了该线程, 源码如下:

```

public void schedule(WorkRequest workRequest, long id) {
    if (stopped) {
        workRequest.cleanup();
        return;
    }

    //创建ScheduledWorkRequest线程
    ScheduledWorkRequest scheduledWorkRequest = new ScheduledWorkRequest(workRequest);
    // If we have a worker thread pool, use that; otherwise, do the work
    // directly.
    int size = workers.size();
    if (size > 0) {
        try {
            // make sure to map negative ids as well to [0, size-1]
            int workerNum = ((int) (id % size) + size) % size;
            ExecutorService worker = workers.get(workerNum);
            //执行对应线程
            worker.execute(scheduledWorkRequest);
        } catch (RejectedExecutionException e) {
            LOG.warn("ExecutorService rejected execution", e);
            workRequest.cleanup();
        }
    }
}

```

创建线程

执行对应线程

ScheduledWorkRequest 实现了 Runnable 接口，并在 run() 方法中调用了 IOWorkRequest 中的 doWork 方法，在该方法中会调用 doIO 执行IO数据处理，源码如下：

```

@Override
public void run() {
    try {
        // Check if stopped while request was on queue
        if (stopped) {
            workRequest.cleanup();
            return;
        }
        //执行IOWorkRequest中的doWork方法
        workRequest.doWork();
    } catch (Exception e) {
        LOG.warn("Unexpected exception", e);
        workRequest.cleanup();
    }
}

```

执行了IOWorkRequest中的doWork方法

IOWorkRequest 的 doWork 源码如下：

```

public void doWork() throws InterruptedException {
    if (!key.isValid()) {
        selectorThread.cleanupSelectionKey(key);
        return;
    }

    if (key.isReadable() || key.isWritable()) {
        //执行IO数据处理
        cnxn.doIO(key);

        // Check if we shutdown or doIO() closed this connection
        if (stopped) {
            cnxn.close(ServerCnxn.DisconnectReason. SERVER_SHUTDOWN);
            return;
        }
        if (!key.isValid()) {
            selectorThread.cleanupSelectionKey(key);
            return;
        }
        touchCnxn(cnxn);
    }
}

```

接下来的调用链路比较复杂，我们把核心步骤列出，在能直接看到数据读取的地方详细分析源码。上面方法调用链路：`NIOServerCnxn.doIO()`>`readPayload()`>`readRequest()`>`ZookeeperServer.processPacket()`，最后一步方法是获取核心数据的地方，我们可以修改下代码读取数据：

```

* 数据包处理
* @param cnxn
* @param incomingBuffer
* @throws IOException
*/
public void processPacket(ServerCnxn cnxn, ByteBuffer incomingBuffer) throws IOException {
    // We have the request, now process and setup for next
    InputStream bais = new ByteBufferInputStream(incomingBuffer);

    //=====测试 Start=====
    //定义接收输入流对象（输出流）
    ByteArrayOutputStream os = new ByteArrayOutputStream();

    //将网络输入流读取到输出流中
    byte[] buffer = new byte[1024];
    int len=0;
    while ((len=bais.read(buffer))!=-1) {
        os.write(buffer, off: 0, len);
    }

    String result = new String(os.toByteArray(), charsetName: "UTF-8");
    System.out.println("processPacket-----读到的数据: "+result);
    //=====测试 End=====
}

```

添加测试代码如下：

```
//=====测试 Start=====
//定义接收输入流对象（输出流）
ByteArrayOutputStream os = new ByteArrayOutputStream();

//将网络输入流读取到输出流中
byte[] buffer = new byte[1024];
int len=0;
while ((len=bais.read(buffer))!=-1){
    os.write(buffer,0,len);
}
String result = new String(os.toByteArray(),"UTF-8");
System.out.println("processPacket-----读到的数据："+result);
//=====测试 End=====
```

我们启动客户端创建一个demo节点，并添加数据为 abcdefg

```
create /demo abcdefg
```

控制台数据如下：

```
processPacket-----读到的数据：    /demo  abcdefg
```

测试完成后，不要忘了将该测试注释掉。我们可以执行其他增删改查操作，可以输出 `RequestHeader.type` 查看操作类型，操作类型代码在 `zooDefs` 中有标识,常用的操作类型如下：

```
int create = 1;
int delete = 2;
int exists = 3;
int getData = 4;
int setData = 5;
int getACL = 6;
int setACL = 7;
int getChildren = 8;
int sync = 9;
int ping = 11;
```

2.3.4 ConnectionExpirerThread剖析

后台启动 `ConnectionExpirerThread` 清理线程清理过期的 `session`,线程中无限循环,执行工作如下:

```

public void run() {
    try {
        while (!stopped) {
            //获取需要等待的时间
            long waitTime = cnxnExpiryQueue.getWaitTime();
            //执行休眠
            if (waitTime > 0) {
                Thread.sleep(waitTime); 执行休眠
                continue;
            }
            //清理过期会话
            for (NIOServerCnxn conn : cnxnExpiryQueue.poll()) {
                ServerMetrics.getMetrics().SESSIONLESS_CONNECTIONS_EXPIRED.add(1);
                conn.close(ServerCnxn.DisconnectReason.CONNECTION_EXPIRED);
            }
        }
    } catch (InterruptedException e) {
        LOG.info("ConnectionExpirerThread interrupted");
    }
}

```

获取当前会话过去等待时间

清理过期会话

2.3 ZK通信优劣总结

Zookeeper在通信方面默认使用了NIO，并支持扩展Netty实现网络数据传输。相比传统IO，NIO在网络数据传输方面有很多明显优势：

- 1: 传统IO在处理数据传输请求时，针对每个传输请求生成一个线程，如果IO异常，那么线程阻塞，在IO恢复后唤醒处理线程。在同时处理大量连接时，会实例化大量的线程对象。每个线程的实例化和回收都需要消耗资源，jvm需要为其分配TLAB，然后初始化TLAB，最后绑定线程，线程结束时又需要回收TLAB，这些都需要CPU资源。
- 2: NIO使用selector来轮询IO流，内部使用poll或者epoll，以事件驱动形式来相应IO事件的处理。同一时间只需实例化很少的线程对象，通过对线程的复用来提高CPU资源的使用效率。
- 3: CPU轮流为每个线程分配时间片的形式，间接的实现单物理核处理多线程。当线程越多时，每个线程分配到的时间片越短，或者循环分配的周期越长，CPU很多时间都耗费在了线程的切换上。线程切换包含线程上个线程数据的同步(TLAB同步)，同步变量同步至主存，下个线程数据的加载等等，他们都是很耗费CPU资源的。
- 4: 在同时处理大量连接，但活跃连接不多时，NIO的事件响应模式相比于传统IO有着极大的性能提升。NIO还提供了FileChannel，以zero-copy的形式传输数据，相较于传统的IO，数据不需要拷贝至用户空间，可直接由物理硬件(磁盘等)通过内核缓冲区后直接传递至网关，极大的提高了性能。
- 5: NIO提供了MappedByteBuffer，其将文件直接映射到内存（这里的内存指的是虚拟内存，并不是物理内存），能极大的提高IO吞吐能力。

ZK在使用NIO通信虽然大幅提升了数据传输能力，但也存在一些代码诟病问题：

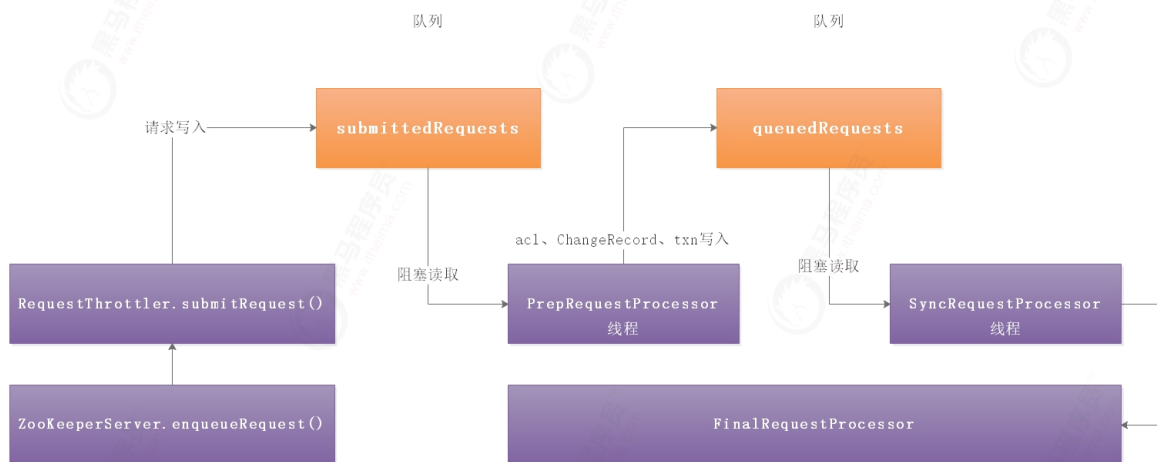
- 1: Zookeeper通信源码部分学习成本高，需要掌握NIO和多线程
- 2: 多线程使用频率高，消耗资源多，但性能得到提升
- 3: Zookeeper数据处理调用链路复杂，多处存在内部类，代码结构不清晰，写法比较经典

4 RequestProcessor处理请求源码剖析

zookeeper 的业务处理流程就像工作流一样，其实就是一个单链表；在 zookeeper 启动的时候，会确立各个节点的角色特性，即 leader、follower 和 observer，每个角色确立后，就会初始化它的工作责任链；

4.1 RequestProcessor结构

客户端请求过来，每次执行不同事务操作的时候，Zookeeper也提供了一套业务处理流程 RequestProcessor，RequestProcessor 的处理流程如下图：



我们来看一下 RequestProcessor 初始化流程，ZooKeeperServer.setupRequestProcessors() 方法源码如下：

```
/**
 * 初始化业务处理流程
 */
protected void setupRequestProcessors() {
    //创建FinalRequestProcessor
    RequestProcessor finalProcessor = new FinalRequestProcessor(zks: this);
    //创建SyncRequestProcessor，并将finalProcessor作为它的下一个业务链
    RequestProcessor syncProcessor = new SyncRequestProcessor(zks: this, finalProcessor);
    //启动syncProcessor
    ((SyncRequestProcessor) syncProcessor).start();
    //创建RequestProcessor，并作为第一个处理业务的RequestProcessor，将syncProcessor作为它的下一个业务链
    firstProcessor = new PrepRequestProcessor(zks: this, syncProcessor);
    //启动firstProcessor
    ((PrepRequestProcessor) firstProcessor).start();
}
```

它的创建步骤：

- 1: 创建 finalProcessor。
- 2: 创建 syncProcessor，并将 finalProcessor 作为它的下一个业务链。
- 3: 启动 syncProcessor。
- 4: 创建 firstProcessor (PrepRequestProcessor)，将 syncProcessor 作为 firstProcessor 的下一个业务链。
- 5: 启动 firstProcessor。

syncProcessor 创建时，将 finalProcessor 作为参数传递进来源码如下：

```

/**
 * 创建SyncRequestProcessor
 * @param nextProcessor: 下一个责任链 FinalRequestProcessor
 */
public SyncRequestProcessor(ZooKeeperServer zks, RequestProcessor nextProcessor) {
    super(threadName: "SyncThread:" + zks.getServerId(), zks.getZooKeeperServerListener());
    this.zks = zks;
    //下一个责任链
    this.nextProcessor = nextProcessor;
    this.toFlush = new ArrayDeque<>(zks.getMaxBatchSize());
}

```

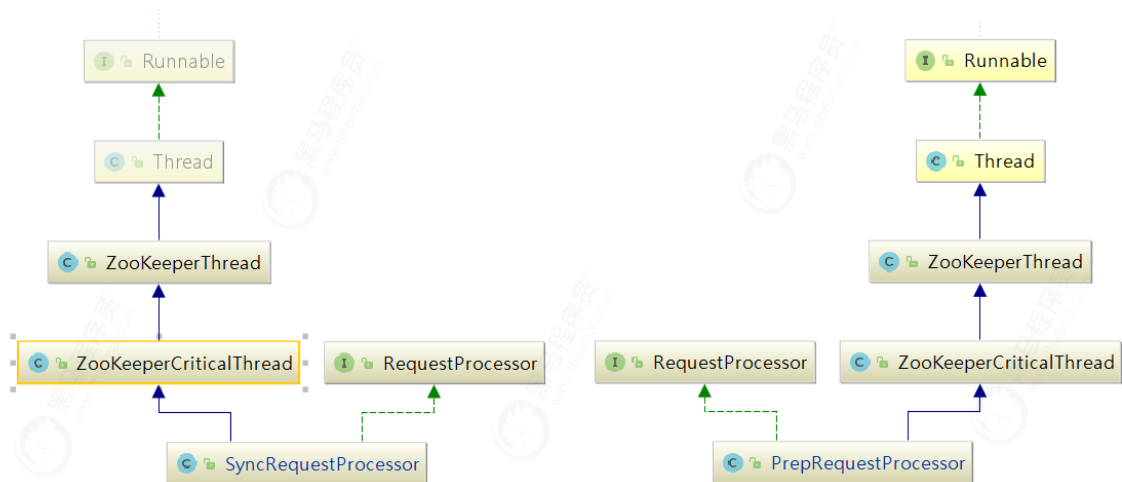
firstProcessor 创建时，将 syncProcessor 作为参数传递进来源码如下：

```

/**
 * 创建PrepRequestProcessor
 * @param nextProcessor: 下一个责任链 SyncRequestProcessor
 */
public PrepRequestProcessor(ZooKeeperServer zks, RequestProcessor nextProcessor) {
    super(
        threadName: "ProcessThread(sid:" + zks.getServerId()
            + " cport:" + zks.getClientPort()
            + "):", zks.getZooKeeperServerListener());
    //下一个责任链
    this.nextProcessor = nextProcessor;
    this.zks = zks;
    this.digestEnabled = ZooKeeperServer.isDigestEnabled();
    if (this.digestEnabled) {
        this.digestCalculator = new DigestCalculator();
    }
}

```

PrepRequestProcessor/SyncRequestProcessor 关系图：



PrepRequestProcessor 和 SyncRequestProcessor 的结构一样，都是实现了 Thread 的一个线程，所以在这里初始化时便启动了这两个线程。

4.2 PrepRequestProcessor剖析

PrepRequestProcessor 是请求处理器的第1个处理器，我们把之前的请求业务处理衔接起来，一步一步分析。

ZookeeperServer.processPacket()>submitRequest()>enqueueRequest()>RequestThrottler.submitRequest()，我们来看下 RequestThrottler.submitRequest() 源码,它将当前请求添加到 submittedRequests 队列中了，源码如下：

```
public void submitRequest(Request request) {
    if (stopping) {
        LOG.debug("Shutdown in progress. Request cannot be processed");
        dropRequest(request);
    } else {
        request.requestThrottleQueueTime = Time.currentElapsedTime();
        //将当前请求添加到submittedRequests队列中
        submittedRequests.add(request);
    }
}
```

而 RequestThrottler 继承了 ZookeeperCriticalThread > ZookeeperThread > Thread，也就是说当前 RequestThrottler 是个线程，我们看看它的 run 方法做了什么事，源码如下：

```
@Override
public void run() {
    try {
        while (true) {
            //...略
            // A dropped stale request will be null
            if (request != null) {
                if (request.isStale()) {
                    ServerMetrics.getMetrics().STALE_REQUESTS.add(1);
                }
                final long elapsedTime = Time.currentElapsedTime() - request.requestThrottleQueueTime;
                ServerMetrics.getMetrics().REQUEST_THROTTLE_QUEUE_TIME.add(elapsedTime);
                if (shouldThrottleOp(request, elapsedTime)) {
                    request.setIsThrottled(true);
                    ServerMetrics.getMetrics().THROTTLED_OPS.add(1);
                }
                zks.submitRequestNow(request); //调用ZookeeperServer.submitRequestNow(request)方法
            }
        }
    } catch (InterruptedException e) {
        LOG.error("Unexpected interruption", e);
    }
}
```

RequestThrottler 调用了 zookeeperServer.submitRequestNow() 方法，而该方法又调用了 firstProcessor 的方法，源码如下：

```
public void submitRequestNow(Request si) {
    //...略
    try {
        touch(si.cnxn);
        boolean validpacket = Request.isValid(si.type);
        if (validpacket) {
            setLocalSessionFlag(si);
            firstProcessor.processRequest(si); //firstProcessor=PreRequestProcessor
            if (si.cnxn != null) {
                incInProcess();
            }
        }
    }
}
```

ZookeeperServer.submitRequestNow() 方法调用了 firstProcessor.processRequest() 方法，而这里的 firstProcessor 就是初始化业务处理链中的 PrepRequestProcessor，也就是说三个 RequestProcessor 中最先调用的是 PrepRequestProcessor。

PrepRequestProcessor.processRequest() 方法将当前请求添加到了队列 submittedRequests 中，源码如下：

```
public void processRequest(Request request) {
    request.prepQueueStartTime = Time.currentElapsedTime();
    submittedRequests.add(request);  // 添加到队列submittedRequests中
    ServerMetrics.getMetrics().PREP_PROCESSOR_QUEUED.add(1);
}
```

上面方法中并未从 submittedRequests 队列中获取请求，如何执行请求的呢，因为 PrepRequestProcessor 是一个线程，因此会在 run 中执行，我们查看 run 方法源码的时候发现它调用了 pRequest() 方法，pRequest() 方法源码如下：

```
protected void pRequest(Request request) throws RequestProcessorException {
    // LOG.info("Prep>>> cxid = " + request.cxid + " type = " +
    // request.type + " id = 0x" + Long.toHexString(request.sessionId));
    request.setHdr(null);
    request.setTxn(null);

    if (!request.isThrottled()) {
        //处理业务流程
        pRequestHelper(request);
    }

    request.zxid = zks.getZxid();
    long timeFinishedPrepare = Time.currentElapsedTime();
    ServerMetrics.getMetrics().PREP_PROCESS_TIME.add(timeFinishedPrepare - request.prepStartTime);
    //交给下一个业务链处理->SyncRequestProcessor
    nextProcessor.processRequest(request);
    ServerMetrics.getMetrics().PROPOSAL_PROCESS_TIME.add(Time.currentElapsedTime() - timeFinishedPrepare);
}
```

首先先执行 pRequestHelper() 方法，该方法是 PrepRequestProcessor 处理核心业务流程，主要是一些过滤操作，操作完成后，会将请求交给下一个业务链，也就是 SyncRequestProcessor.processRequest() 方法处理请求。

我们来看一下 PrepRequestProcessor.pRequestHelper() 方法做了哪些事，源码如下：

```

private void pRequestHelper(Request request) throws RequestProcessorException {
    try {
        //操作类型，在ZooDefs中可以查看对应操作编码
        switch (request.type) {
            case OpCode.createContainer:
            case OpCode.create:
            case OpCode.create2:
                CreateRequest create2Request = new CreateRequest();
                pRequest2Txn(request.type, zks.getNextZxid(), request, create2Request, deserialize: true);
                break;
            case OpCode.createTTL:
                CreateTTLRequest createTtlRequest = new CreateTTLRequest();
                pRequest2Txn(request.type, zks.getNextZxid(), request, createTtlRequest, deserialize: true);
                break;
            case OpCode.deleteContainer:
            case OpCode.delete:
                DeleteRequest deleteRequest = new DeleteRequest();
                pRequest2Txn(request.type, zks.getNextZxid(), request, deleteRequest, deserialize: true);
                break;
            case OpCode.setData:
                SetDataRequest setDataRequest = new SetDataRequest();
                pRequest2Txn(request.type, zks.getNextZxid(), request, setDataRequest, deserialize: true);
                break;
            case OpCode.reconfig:
                ReconfigRequest reconfigRequest = new ReconfigRequest();
                ByteBufferInputStream.byteBuffer2Record(request.request, reconfigRequest);
                pRequest2Txn(request.type, zks.getNextZxid(), request, reconfigRequest, deserialize: true);
                break;
            case OpCode.setACL:
                SetACLRequest setAclRequest = new SetACLRequest();
                pRequest2Txn(request.type, zks.getNextZxid(), request, setAclRequest, deserialize: true);
                break;
            case OpCode.check:
                CheckVersionRequest checkRequest = new CheckVersionRequest();
                pRequest2Txn(request.type, zks.getNextZxid(), request, checkRequest, deserialize: true);
                break;
        }
    }
}

```

从上面源码可以看出 `PrepRequestProcessor.pRequestHelper()` 方法判断了客户端操作类型，但无论哪种操作类型几乎都调用了 `pRequest2Txn()` 方法，我们来看看源码：

```

protected void pRequest2Txn(int type, long zxid, Request request, Record record, boolean deserialize) throws KeeperException {
    if (request.getHdr() == null) {
        request.setHdr(new TxnHeader(request.sessionId, request.cxid, zxid,
            Time.currentTimeMillis(), type));
    }

    PrecalculatedDigest precalculatedDigest;
    switch (type) {
        case OpCode.create:
        case OpCode.create2:
        case OpCode.createTTL:
        case OpCode.createContainer: {
            pRequest2TxnCreate(type, request, record, deserialize);
            break;
        }
        case OpCode.deleteContainer: {
            String path = new String(request.request.array(), UTF_8);
            String parentPath = getParentPathAndValidate(path);
            ChangeRecord nodeRecord = getRecordForPath(path);
            if (nodeRecord.childCount > 0) {
                throw new KeeperException.NotEmptyException(path);
            }
            if (EphemeralType.get(nodeRecord.stat.getEphemeralOwner()) == EphemeralType.NORMAL) {
                throw new KeeperException.BadVersionException(path);
            }
            //修改快照数据记录
            ChangeRecord parentRecord = getRecordForPath(parentPath);
            //事务信息记录
            request.setTxn(new DeleteTxn(path));
            parentRecord = parentRecord.duplicate(request.getHdr().getZxid());
            parentRecord.childCount--;
            //状态数据记录
            parentRecord.stat.setPzxid(request.getHdr().getZxid());
            parentRecord.precalculatedDigest = precalculateDigest(
                DigestOpCode.UPDATE, parentPath, parentRecord.data, parentRecord.stat);
            addChangeRecord(parentRecord);

            nodeRecord = new ChangeRecord(request.getHdr().getZxid(), path, stat:null, childCount:-1, acl:null);
            nodeRecord.precalculatedDigest = precalculateDigest(DigestOpCode.REMOVE, path);
            setTxnDigest(request, nodeRecord.precalculatedDigest);
            addChangeRecord(nodeRecord);
            break;
        }
        case OpCode.delete:
            zks.sessionTracker.checkSession(request.sessionId, request.getOwner());
            DeleteRequest deleteRequest = (DeleteRequest) record;
            if (deserialize) {
                ByteBufferInputStream.byteBuffer2Record(request.request, deleteRequest);
            }
            String path = deleteRequest.getPath();
            String parentPath = getParentPathAndValidate(path);
            ChangeRecord parentRecord = getRecordForPath(parentPath);
            //权限检查
            zks.checkACL(request.cnxn, parentRecord.acl, ZooDefs.Perms.DELETE, request.authInfo, path, setAcls:null);
            ChangeRecord nodeRecord = getRecordForPath(path);
            checkAndIncVersion(nodeRecord.stat.getVersion(), deleteRequest.getVersion(), path);
            if (nodeRecord.childCount > 0) {
                throw new KeeperException.NotEmptyException(path);
            }
            request.setTxn(new DeleteTxn(path));
            parentRecord = parentRecord.duplicate(request.getHdr().getZxid());
            parentRecord.childCount--;
            parentRecord.stat.setPzxid(request.getHdr().getZxid());
            parentRecord.precalculatedDigest = precalculateDigest(
                DigestOpCode.UPDATE, parentPath, parentRecord.data, parentRecord.stat);
            addChangeRecord(parentRecord);

            nodeRecord = new ChangeRecord(request.getHdr().getZxid(), path, stat:null, childCount:-1, acl:null);
            nodeRecord.precalculatedDigest = precalculateDigest(DigestOpCode.REMOVE, path);
            setTxnDigest(request, nodeRecord.precalculatedDigest);
    }
}

```

```

        addChangeRecord(nodeRecord);
        break;
    case OpCode.setData:
        zks.sessionTracker.checkSession(request.sessionId, request.getOwner());
        SetDataRequest setDataRequest = (SetDataRequest) record;
        if (deserialize) {
            ByteBufferInputStream.byteBuffer2Record(request.request, setDataRequest);
        }
        path = setDataRequest.getPath();
        validatePath(path, request.sessionId);
        nodeRecord = getRecordForPath(path);
        //权限检查
        zks.checkACL(request.cnxn, nodeRecord.acl, ZooDefs.Perms.WRITE, request.authInfo, path, setAcls: null);
        int newVersion = checkAndIncVersion(nodeRecord.stat.getVersion(), setDataRequest.getVersion(), path);
        request.setTxn(new SetDataTxn(path, setDataRequest.getData(), newVersion));
        nodeRecord = nodeRecord.duplicate(request.getHdr().getZxid());
        nodeRecord.stat.setVersion(newVersion);
        nodeRecord.stat.setMtime(request.getHdr().getTime());
        nodeRecord.stat.setMxid(zxid);
        nodeRecord.data = setDataRequest.getData();
        nodeRecord.precalculatedDigest = precalculateDigest(
            DigestOpCode.UPDATE, path, nodeRecord.data, nodeRecord.stat);
        setTxnDigest(request, nodeRecord.precalculatedDigest);
        addChangeRecord(nodeRecord);
        break;
    case OpCode.reconfig:
        if (!zks.isReconfigEnabled()) {
            LOG.error("Reconfig operation requested but reconfig feature is disabled.");
            throw new KeeperException.ReconfigDisabledException();
        }

        //权限检查
        if (ZooKeeperServer.skipACL) {
            LOG.warn("skipACL is set, reconfig operation will skip ACL checks!");
        }

        zks.sessionTracker.checkSession(request.sessionId, request.getOwner());
        LeaderZooKeeperServer lzks;
        try {
            lzks = (LeaderZooKeeperServer) zks;
        } catch (ClassCastException e) {
            // standalone mode - reconfiguration currently not supported
            throw new KeeperException.UnimplementedException();
        }

```

从上面代码可以看出 `pRequest2Txn()` 方法主要做了权限校验、快照记录、事务信息记录相关的事，还并未涉及数据处理，也就是说 `PrepRequestProcessor` 其实是做了操作前权限校验、快照记录、事务信息记录相关的事。

我们DEBUG调试一次，看看业务处理流程是否和我们上面所分析的一致。

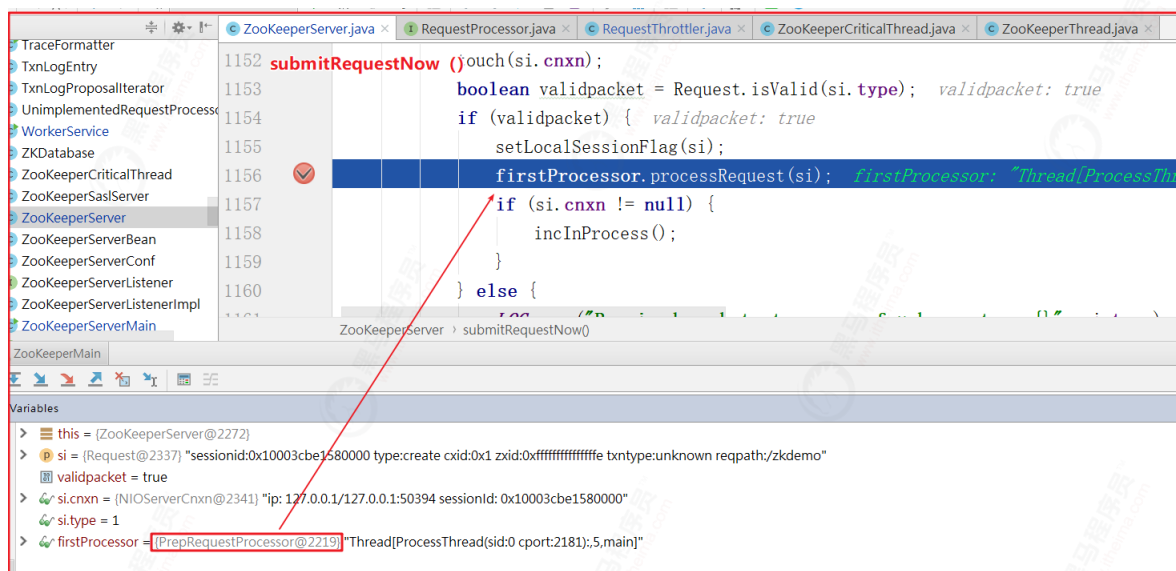
添加节点：

```
create /zkdemo itheima
```

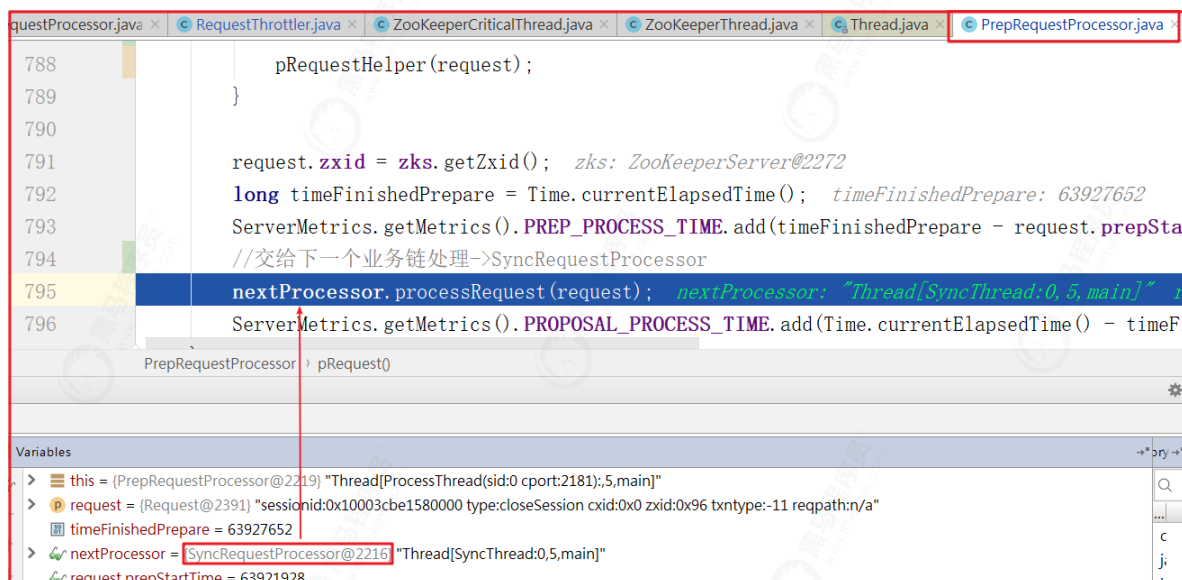
DEBUG测试如下：

客户端请求先经过 `zooKeeperServer.submitRequestNow()` 方法，并调用

`firstProcessor.processRequest()` 方法，而 `firstProcessor = PrepRequestProcessor`，如下图：



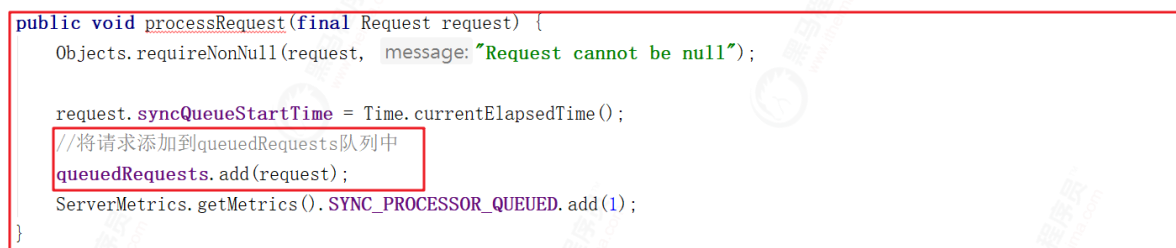
进入 `PrepRequestProcessor.pRequest()` 方法，执行完 `pRequestHelper()` 方法后，开始执行下一个业务链的方法，而下一个业务链 `nextProcessor = SyncRequestProcessor`，如下测试图：



4.3 SyncRequestProcessor剖析

分析了 `PrepRequestProcessor` 处理器后，接着来分析 `SyncRequestProcessor`，该处理器主要是将请求数据高效率存入磁盘，并且请求在写入磁盘之前是不会被转发到下个处理器的。

我们先看请求被添加到队列的方法：



同样 `SyncRequestProcessor` 是一个线程，执行队列中的请求也在线程中触发，我们看它的 `run` 方法，源码如下：

 1608202536451

`run` 方法会从 `queuedRequests` 队列中获取一个请求，如果获取不到就会阻塞等待直到获取到一个请求对象，程序才会继续往下执行，接下来会调用 `Snapshot Thread` 线程实现将客户端发送的数据以快照的方式写入磁盘，最终调用 `flush()` 方法实现数据提交，`flush()` 方法源码如下：

```
/**
 * 提交刷新事务
 * @throws IOException
 * @throws RequestProcessorException
 */
private void flush() throws IOException, RequestProcessorException {
    if (this.toFlush.isEmpty()) {
        return;
    }

    ServerMetrics.getMetrics().BATCH_SIZE.add(toFlush.size());

    long flushStartTime = Time.currentElapsedTime();
    //数据提交
    zks.getZKDatabase().commit(); //提交数据
    ServerMetrics.getMetrics().SYNC_PROCESSOR_FLUSH_TIME.add(Time.currentElapsedTime() - flushStartTime);

    if (this.nextProcessor == null) {
        this.toFlush.clear();
    } else {
        //this.toFlush.isEmpty() 不为空，上面往toFlush中添加了当前请求
        while (!this.toFlush.isEmpty()) {
            final Request i = this.toFlush.remove();
            long latency = Time.currentElapsedTime() - i.syncQueueStartTime;
            ServerMetrics.getMetrics().SYNC_PROCESSOR_QUEUE_AND_FLUSH_TIME.add(latency);
            //执行下一个业务链对象，下一个业务链对象是FinalRequestProcessor
            this.nextProcessor.processRequest(i); //调用下一个业务链
        }
        if (this.nextProcessor instanceof Flushable) {
            ((Flushable) this.nextProcessor).flush();
        }
    }

    lastFlushTime = Time.currentElapsedTime();
}
```

`flush()` 方法实现了数据提交，并且会将请求交给下一个业务链，下一个业务链为 `FinalRequestProcessor`。

4.4 FinalRequestProcessor剖析

前面分析了 `SyncRequestProcessor`，接着分析请求处理链中最后的一个处理器 `FinalRequestProcessor`，该业务处理对象主要用于返回 `Response`。

```

public void processRequest(Request request) {
    LOG.debug("Processing request:: {}", request);

    if (LOG.isTraceEnabled()) {
        long traceMask = ZooTrace.CLIENT_REQUEST_TRACE_MASK;
        if (request.type == OpCode.ping) {
            traceMask = ZooTrace.SERVER_PING_TRACE_MASK;
        }
        ZooTrace.logRequest(LOG, traceMask, rp:'E', request, header:"");
    }

    ProcessTxnResult rc = null;
    if (!request.isThrottled()) {
        rc = applyRequest(request);
    }
    if (request.cnxn == null) {
        return;
    }
    ServerCnxn cnxn = request.cnxn;

    long lastZxid = zks.getZKDatabase().getDataTreeLastProcessedZxid();

    String lastOp = "NA";
    // Notify ZooKeeperServer that the request has finished so that it can
    // update any request accounting/throttling limits
    zks.decInProcess();
    zks.requestFinished(request);
    Code err = Code.OK;
    Record rsp = null;
    String path = null;
    int responseSize = 0;
    try {
        if (request.getHdr() != null && request.getHdr().getType() == OpCode.error) {
            AuditHelper.addAuditLog(request, rc, failedTxn: true);
            /*
             * When local session upgrading is disabled, leader will
             * reject the ephemeral node creation due to session expire.
             * However, if this is the follower that issue the request,
             * it will have the correct error code, so we should use that
             * and report to user
             */
            if (request.getException() != null) {
                throw request.getException();
            } else {
                throw KeeperException.create(KeeperException.Code.get(((ErrorTxn) request.getTxn()).getErr()));
            }
        }

        KeeperException ke = request.getException();
        if (ke instanceof SessionMovedException) {
            throw ke;
        }
        if (ke != null && request.type != OpCode.multi) {
            throw ke;
        }

        LOG.debug("{} ", request);

        if (request.isStale()) {
            ServerMetrics.getMetrics().STALE_REPLIES.add(1);
        }

        if (request.isThrottled()) {
            throw KeeperException.create(Code.THROTTLEDOP);
        }

        AuditHelper.addAuditLog(request, rc);

        switch (request.type) {
            case OpCode.ping: {
                lastOp = "PING";
            }

```

```

//设置响应状态
updateStats(request, lastOp, lastZxid);
//响应数据
responseSize = cnxn.sendResponse(new ReplyHeader(ClientCnxn.PING_XID, lastZxid, err: 0), r: null, tag: "re
return;
}
case OpCode.createSession: {
    lastOp = "SESS";
    //设置响应状态
    updateStats(request, lastOp, lastZxid);
    //响应数据并结束会话
    zks.finishSessionInit(request.cnxn, valid: true);
    return;
}
case OpCode.multi: {
    // 多重操作
    lastOp = "MULT";
    rsp = new MultiResponse();

    // 遍历多重操作结果
    for (ProcessTxnResult subTxnResult : rc.multiResult) {

        OpResult subResult;

        switch (subTxnResult.type) {
            case OpCode.check:
                // 检查
                subResult = new CheckResult();
                break;
            case OpCode.create:
                // 创建
                subResult = new CreateResult(subTxnResult.path);
                break;
            case OpCode.create2:
            case OpCode.createTTL:
            case OpCode.createContainer:
                subResult = new CreateResult(subTxnResult.path, subTxnResult.stat);
                break;
            case OpCode.delete:
            case OpCode.deleteContainer:
                // 删除
                subResult = new DeleteResult();
                break;
            case OpCode.setData:
                // 设置数据
                subResult = new SetDataResult(subTxnResult.stat);

```

4.5 ZK业务链处理优劣总结

Zookeeper业务链处理，思想遵循了AOP思想，但并未采用相关技术，为了提升效率，仍然大幅使用到了多线程。正因为有了业务链路处理先后顺序，似的Zookeeper业务处理流程更清晰更容易理解，但大量混入了多线程，也似的学习成本增加。