【Nacos, Java】Core 源码分析

李国峰 2018K8009922027

一．Nacos 简介

Nacos 是阿里巴巴的开源项目，全名 Dynamic Naming and Configuration Service，专注于服务发现和配置管理领域。

Nacos 提供四个主要功能：

服务发现和服务运行状况检查：

Nacos 使服务自身的注册变得容易，并可通过 DNS 或 HTTP 接口发现其他服务。Nacos 还提供服务的实时运行状况检查，以防止向不正常的主机或服务实例发送请求。

动态配置管理：

动态配置服务允许以集中和动态的方式跨所有环境管理所有服务的配置。Nacos 无需在更新配置时重新部署应用程序和服务，这使得配置更改更加高效和敏捷。

动态 DNS 服务：

Nacos 支持加权路由，使您更轻松地在数据中心内的生产环境中实现中层负载平衡、灵活的路由策略、流控制和简单的 DNS 解析服务。它可以帮助您轻松实现基于 DNS 的服务发现，并防止应用程序耦合到特定于供应商的服务发现 API。

服务和元数据管理：

Nacos 提供易于使用的服务仪表板，可帮助您管理服务元数据、配置、kubernetes DNS、服务运行状况和指标统计信息。

原始介绍参考 https://github.com/alibaba/nacos

二．【Nacos, Java】Core 功能点——底层一致性协议的分析（部分）

0.什么是一致性协议？

当使用分布式系统来运行程序时，各个节点之间在物理上相互独立，通过网络进行协调和沟通，但相互独立的节点之间无法准确知道其他节点间事务的执行情况，为了保证计算结果的正确性，需要有一种“约定”来使得不同的节点能够正确获取彼此的状态，通过引入一个“协调者”组件来统一调度所有分布式节点的执行。

1. Nacos 一致性协议算法 Distro

Distro 协议被定位为临时数据的一致性协议，该协议不需要把数据存储到磁盘或者数据库，因为临时数据通常和服务器保持一个 session 会话，该会话只要存在，数据就不会丢失。

（1） Distro 协议算法大致流程

（a） Nacos 启动，首先从其他远程节点同步全部数据

（b） Nacos 每个节点都是平等的，都可以处理写入请求，同时把新数据同步到其他节点

（c） 每个节点只负责部分数据，定时发送自己负责数据校验值到其他节点来保持数据一致性

（2） Distro 协议算法数据存储的实现

```
/** Distro data storage. ...*/
public interface DistroDataStorage {

    /** Get distro datum. ...*/
    DistroData getDistroData(DistroKey distroKey);

    /** Get all distro datum snapshot. ...*/
    DistroData getDatumSnapshot();

    /** Get verify datum. ...*/
    DistroData getVerifyData();
}
```

定义了一个接口类型（interface）的对象，将数据的存储与获取整合到一个

对象中，在接口中的方法只有特征没有具体实现，意味着这些方法可以在不同的地方以不同的方式实现。当其他地方的某个类需要进行相关的操作时，可以在该类中实现该接口。

（3） Distro 数据初始全量同步过程（仅包含 Core 里的部分）

```
public void run() {
    try {
        load();
        if (!checkCompleted()) {
            GlobalExecutor.submitLoadDataTask( runnable: this, distroConfig.getLoadDataRetryDelayMillis());
        } else {
            loadCallback.onSuccess();
            Loggers.DISTRO.info("[DISTRO-INIT] load snapshot data success");
        }
    } catch (Exception e) {
        loadCallback.onFailed(e);
        Loggers.DISTRO.error("[DISTRO-INIT] load snapshot data failed. ", e);
    }
}

/** 初始化时从其他节点同步数据 */
private void load() throws Exception {
    while (memberManager.allMembersWithoutSelf().isEmpty()) {
        Loggers.DISTRO.info("[DISTRO-INIT] waiting server list init...");
        TimeUnit.SECONDS.sleep( timeout: 1);
    }
    while (distroComponentHolder.getDataStorageTypes().isEmpty()) {
        Loggers.DISTRO.info("[DISTRO-INIT] waiting distro data storage register...");
        TimeUnit.SECONDS.sleep( timeout: 1);
    }
    for (String each : distroComponentHolder.getDataStorageTypes()) {
        if (!loadCompletedMap.containsKey(each) || !loadCompletedMap.get(each)) {
            loadCompletedMap.put(each, loadAllDataSnapshotFromRemote(each));
        }
    }
}
```

Distro 协议节点启动时会从其他节点全量同步数据，启动一个定时任务线程 DistroLoadDataTask 加载数据，调用 load()方法加载数据；

调用 loadAllDataSnapshotFromRemote()方法从远程机器同步所有数据并返回同步结果（boolean）；

```java
/** 从远程机器同步所有的数据 */
private boolean loadAllDataSnapshotFromRemote(String resourceType) {
    DistroTransportAgent transportAgent = distroComponentHolder.findTransportAgent(resourceType);
    DistroDataProcessor dataProcessor = distroComponentHolder.findDataProcessor(resourceType);
    if (null == transportAgent || null == dataProcessor) {
        Loggers.DISTRO.warn("[DISTRO-INIT] Can't find component for type {}, transportAgent: {}, dataProcessor: {}",
                resourceType, transportAgent, dataProcessor);
        return false;
    }
    for (Member each : memberManager.allMembersWithoutSelf()) {
        try {
            Loggers.DISTRO.info("[DISTRO-INIT] load snapshot {} from {}", resourceType, each.getAddress());
            DistroData distroData = transportAgent.getDatumSnapshot(each.getAddress());
            boolean result = dataProcessor.processSnapshot(distroData);
            Loggers.DISTRO
                    .info("[DISTRO-INIT] load snapshot {} from {} result: {}", resourceType, each.getAddress(),
                            result);
            if (result) {
                return true;
            }
        } catch (Exception e) {
            Loggers.DISTRO.error("[DISTRO-INIT] load snapshot {} from {} failed.", resourceType, each.getAddress(), e);
        }
    }
    return false;
}
```

（4） Distro 增量同步过程

新增数据使用异步广播进行同步:
DistroProtocol 使用 sync()方法接收增量数据;

```java
public void sync(DistroKey distroKey, DataOperation action, long delay) {
    for (Member each : memberManager.allMembersWithoutSelf()) {
        DistroKey distroKeyWithTarget = new DistroKey(distroKey.getResourceKey(), distroKey.getResourceType(),
                each.getAddress());
        DistroDelayTask distroDelayTask = new DistroDelayTask(distroKeyWithTarget, action, delay);
        distroTaskEngineHolder.getDelayTaskExecuteEngine().addTask(distroKeyWithTarget, distroDelayTask);
        if (Loggers.DISTRO.isDebugEnabled()) {
            Loggers.DISTRO.debug("[DISTRO-SCHEDULE] {} to {}", distroKey, each.getAddress());
        }
    }
}
```

调用 distroTaskEngineHolder 向其他节点发布延迟任务;

```java
public class DistroTaskEngineHolder {

    private final DistroDelayTaskExecuteEngine delayTaskExecuteEngine = new DistroDelayTaskExecuteEngine();

    private final DistroExecuteTaskExecuteEngine executeWorkersManager = new DistroExecuteTaskExecuteEngine();

    public DistroTaskEngineHolder(DistroComponentHolder distroComponentHolder) {
        DistroDelayTaskProcessor defaultDelayTaskProcessor = new DistroDelayTaskProcessor( distroTaskEngineHolder: this, distroComponentHolder);
        delayTaskExecuteEngine.setDefaultTaskProcessor(defaultDelayTaskProcessor);
    }

    public DistroDelayTaskExecuteEngine getDelayTaskExecuteEngine() { return delayTaskExecuteEngine; }

    public DistroExecuteTaskExecuteEngine getExecuteWorkersManager() { return executeWorkersManager; }

    public void registerNacosTaskProcessor(Object key, NacosTaskProcessor nacosTaskProcessor) {
        this.delayTaskExecuteEngine.addProcessor(key, nacosTaskProcessor);
    }
}
```

调用 DistroDelayTaskProcessor.process()方法进行任务投递，将延迟任务转

换成异步变更任务；

```
    public boolean process(NacosTask task) {
        if (!(task instanceof DistroDelayTask)) {...}
        DistroDelayTask distroDelayTask = (DistroDelayTask) task;
        DistroKey distroKey = distroDelayTask.getDistroKey();
        if (DataOperation.CHANGE.equals(distroDelayTask.getAction())) {
            DistroSyncChangeTask syncChangeTask = new DistroSyncChangeTask(distroKey, distroComponentHolder);
            distroTaskEngineHolder.getExecuteWorkersManager().addTask(distroKey, syncChangeTask);
            return true;
        }
        return false;
    }
}
```

三 ．主要功能分析与建模

主要功能分析已经在上一部分简要交代，这里主要分析【Nacos, Java】Core 分布式一致性协议的建模。

所谓建模，实际上就是将我们所分析的对象所具有的特征和行为进行概括和抽象，然后封装在一个类中，之后对于所有与这个类具有类似或相同特征的对象都可以在这个抽象类的基础上进行继承和扩展。

接下来我们结合前一部分的功能分析简要探讨一下 Nacos Distro 的建模，亦即类的抽象与设计。

1. Distro 中的数据加载

Distro 将对数据的加载所需要的属性（成员变量）和行为（方法）进行抽象和封装，统一构成 Java 中最典型面向对象的属性——的类（class）。

```
public class DistroLoadDataTask implements Runnable {

    private final ServerMemberManager memberManager;

    private final DistroComponentHolder distroComponentHolder;

    private final DistroConfig distroConfig;

    private final DistroCallback loadCallback;

    private final Map<String, Boolean> loadCompletedMap;
```

这里还用到了接口（interface）的概念，接口与类不同，但是具有和类一致的面向对象的特征。通过接口定义了一系列成员变量和方法，然后分析我们需要定义的类和该接口的共同特征，通过实例化接口生成了一个全新的类，这个类具有接口所拥有的全部属性和行为，同时拥有自己的独立的属性和行为。在类 DistroLoadDataTask 中，将加载数据这一过程所涉及的属性（例如服务器控制器、配置管理、完成性等标志性特征）作为类的属性归纳起来。

除了属性，一个对象还需要具备方法才可以被更全面地描述。方法可以在对象的内部使用，也可以在别的类调用这个类的时候使用。

```
public void run() {...}

/** 初始化时从其他节点同步数据 */
private void load() throws Exception {...}

/** 从远程机器同步所有的数据 */
private boolean loadAllDataSnapshotFromRemote(String resourceType) {...}

/** 检查同步数据是否完成 */
private boolean checkCompleted() {...}
```

2. Distro 数据接收与任务发布

在上述 Distro 增量同步的功能分析中一共提到了三个类，分别是

```
public class DistroDelayTaskProcessor implements NacosTaskProcessor {

    private final DistroTaskEngineHolder distroTaskEngineHolder;

    private final DistroComponentHolder distroComponentHolder;

     private final DistroTaskEngineHolder distroTaskEngineHolder;
```

```
public class DistroTaskEngineHolder {

    private final DistroDelayTaskExecuteEngine delayTaskExecuteEngine = new DistroDelayTaskExecuteEngine();

    private final DistroExecuteTaskExecuteEngine executeWorkersManager = new DistroExecuteTaskExecuteEngine();

    private volatile boolean isInitialized = false;
```

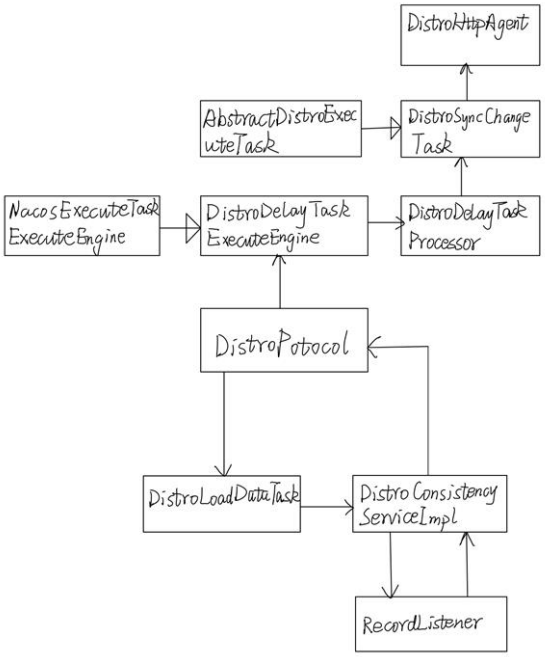
（以上截图均省略方法）

这三个类分别代表的对象是 Distro 协议内容、Distro 任务驱动引擎和 Distro 任务延迟处理器。在这里前两者都是没有父类的初始类，第三者是通过实例化接口生成的子类。

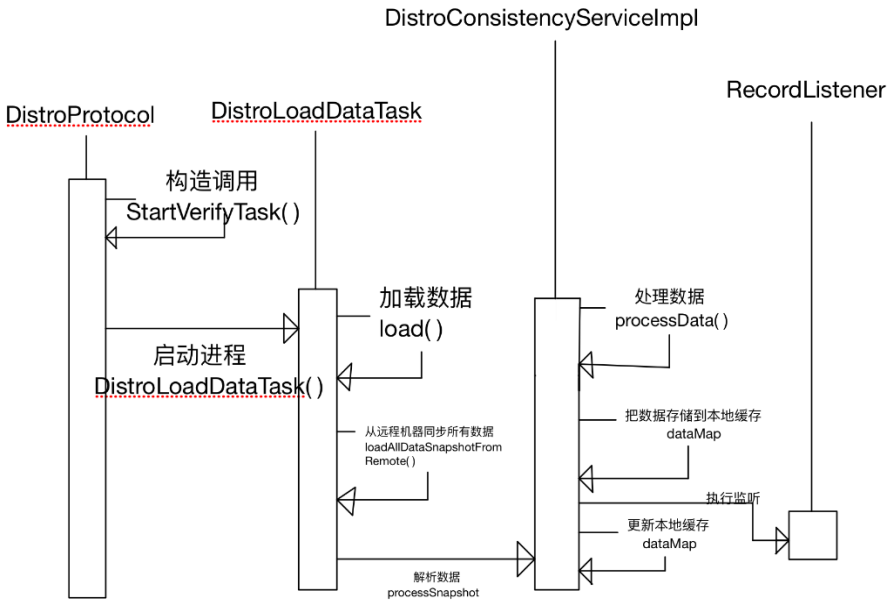
四 ．核心流程分析与设计

1. 主要类的关联

下图为初始数据同步和增量数据同步中，各类在执行中的信息传递以及某些类的继承关系（这些类所具备的属性和方法很多，用标准 UML 图不便绘制，故只表示类与类之间的关系）。



2. 时序图



五．高级设计意图分析

1. 设计模式

设计模式（Design Pattern）代表了最佳的实践，通常被有经验的面向对象软件的开发人员所采用。抽象地说，软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。
具体来说，设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了**重用代码**，让代码更容易被理解，保证代码的可靠性。

2. 设计模式实例

a) 工厂模式

工厂模式通过定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，其主要解决端口选择的问题。
先创建一个 DistroCallBack 接口的实体类。

```
public interface DistroCallback {

    /**
     * Callback when distro task execute successfully.
     */
    void onSuccess();

    /**
     * Callback when distro task execute failed.
     *
     * @param throwable throwable if execute failed caused by exception
     */
    void onFailed(Throwable throwable);
}
```

下一步是创建实现接口的实体类，创建工厂，生成基于给定信息的实体类的对象。
使用该工厂，通过传递类型信息来获取实体类的对象。

```
import com.alibaba.nacos.core.distributed.distro.component.DistroCallback;
```

```java
public class DistroProtocol {

    private final ServerMemberManager memberManager;

    private final DistroComponentHolder distroComponentHolder;

    private final DistroTaskEngineHolder distroTaskEngineHolder;

    private final DistroConfig distroConfig;

    private volatile boolean isInitialized = false;

    public DistroProtocol(ServerMemberManager memberManager, DistroComponentHolder distroComponentHolder,
            DistroTaskEngineHolder distroTaskEngineHolder, DistroConfig distroConfig) {...}

    private void startDistroTask() {...}

    private void startLoadTask() {...}

    private void startVerifyTask() {...}

    public boolean isInitialized() { return isInitialized; }

    /** Start to sync by configured delay. ...*/
    public void sync(DistroKey distroKey, DataOperation action) {...}

    /** Start to sync data to all remote server. ...*/
    public void sync(DistroKey distroKey, DataOperation action, long delay) {...}

    /** Query data from specified server. ...*/
    public DistroData queryFromRemote(DistroKey distroKey) {...}

    /** Receive synced distro data, find processor to process. ...*/
    public boolean onReceive(DistroData distroData) {...}

    /** Receive verify data, find processor to process. ...*/
    public boolean onVerify(DistroData distroData) {...}

    /** Query data of input distro key. ...*/
    public DistroData onQuery(DistroKey distroKey) {...}

    /** Query all datum snapshot. ...*/
    public DistroData onSnapshot(String type) {...}
}
```

3. 面向对象思想的总结（实例参考第三部分）

将某个对象所具有的属性和行为进行分析、概括和抽象，并将它们集合到同一个抽象类中。这样，对于同类对象，可以直接调用这个类进行描述；对于与该类相似但也拥有自己独立特征的对象，可以使用继承、关联和多态的方式生成新的类。

将复杂系统简化，提高系统稳定性，降低维护成本。

将复杂系统进行不同层次的抽象，最大程度提高对象可复用性。

六．结语

在《面向对象的程序设计》这门课程的学习中，我学到了一些基础的 java 编程语法以及面向对象的设计思想，而后者才是本门课程最核心的部分。在有 C 语言的基础之后，学习新的语言易如反掌，但不同语言的精髓是不同的，这体现了对于编程的不同思路、理解以及对问题的分析方式。掌握使用工具的方法比认识工具本身更为重要。

最后，感谢两位老师一整个学期的辛勤付出，课程本身足够有趣，老师上课的方式也非常让人喜爱，以后一定给学弟学妹大力安利这门课。