0、文件结构

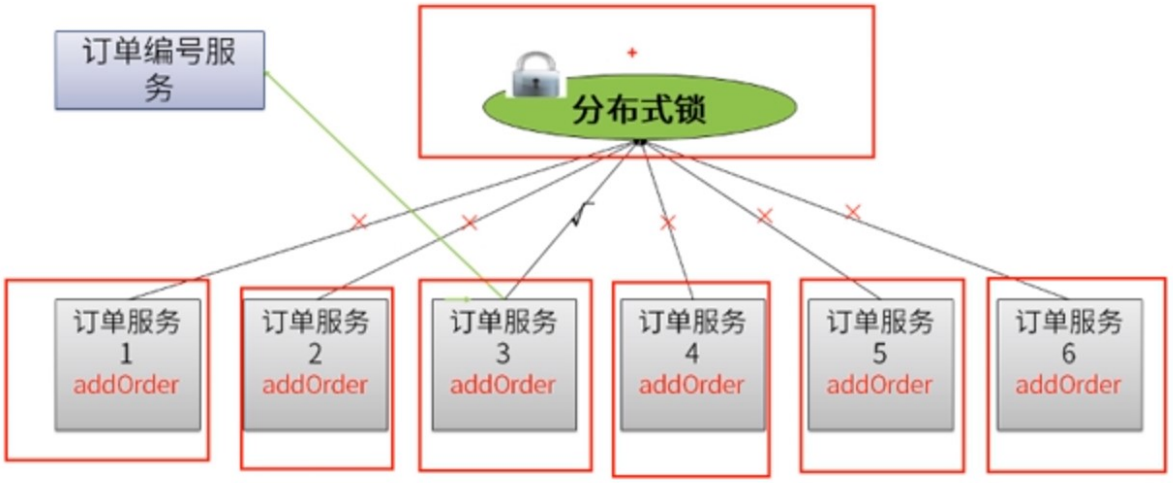
```
└src
   └com
       └company
               CreateNumImpWiehZkLock.java
               CreatNumImp1.java
               CreatNumImpWithLock.java
               CreatService.java
               Main.java
               MyZkSerializer.java
               NumCreat.java
               ZkImproveLock.java
               ZkLock.java
```

- Main是主函数，用于模拟并发请求的实现；
- CreatService是一个订单编号创建的接口，同时基于该接口提供了多种服务
- CreatNumImp1是基于CreatService接口实现的不加锁的订单生成服务，用来验证高并发情况下会出现订单号重复的情况
- CreatNumImpWithLock是基于CreatService接口实现的加JVM锁的订单生成服务，用于验证在分布式服务器情况下的不可用性
- CreateNumImpWiehZkLock.是基于CreatService接口实现的基于zookeeper的分布式锁的订单生成服务
- MyZkSerializer用于实现数据的序列化和逆序列化，因为zookeeper是一个服务，java程序与该服务进行数据交互需要建立连接，在网络中数据的传输以字节为单位，为了可读需要进行序列化。
- NumCreat用于生成订单编号，以时间和一个递增序列来组成订单编号
- ZkImproveLock一个基于Lock接口实现的zookeeper分布式锁，重写了Lock、TryLock和unlock，但是这个版本会发生惊群效应
- ZkImproveLock一个基于Lock接口实现的zookeeper分布式锁，重写了Lock、TryLock和unlock，利用顺序节点，解决了惊群效应。

1、为什么需要分布式锁

当单个服务器不能满足处理需求时，通过java的线程锁将无法应对高并发的处理环境，这时只能通过分布式锁来实现并发处理。

2、分布式锁的原理



3、分布式锁的实现方式

1、所具有的什么特点：

- 排他性：只有一个线程能获取到
- 阻塞性：其他未抢到的线程阻塞，直到锁释放出来，再抢。
- 可重入性：线程获得锁后，后续是否可重复获取该锁。

常用分布式锁实现技术

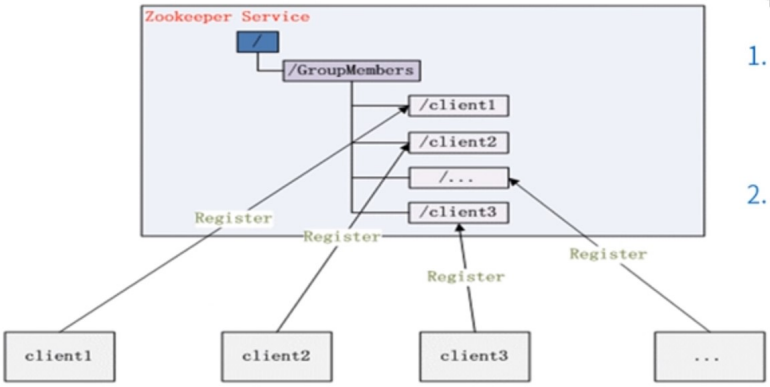
- 基于数据库实现分布式锁
  - 性能较差，容易出现单点故障
  - 锁没有失效时间，容易死锁
- 基于缓存实现分布式锁
  - 实现复杂
  - 存在死锁（或短时间死锁）的可能
- 基于zookeeper实现分布式锁
  - 实现相对简单
  - 可靠性高
  - 性能较好

zookeeper

下载地址：https://mirrors.tuna.tsinghua.edu.cn/apache/zookeeper/

配置方法：https://zookeeper.apache.org/doc/r3.5.4-beta/zookeeperStarted.html

zookeeper是一个开源的分布式应用程序**协调服务**，是Hadoop和Hbase的重要组件。



特性：

1. 节点树数据结构，znode是一个跟Unix文件系统路径相似的节点，可以往这个节点存储或获取数据；
2. 通过客户端可对znode进行增删改查的操作，还可以注册watcher监控znode的变化。

Zookeeper节点类型

- 持久节点（PERSISTENT）
- 持久顺序节点（PERSISTENT_SEQUENTIAL）
- 临时节点（EPHEMERAL）
- 临时顺序节点（EPHEMERAL_SEQUENTIAL）

同一个znode下，节点的名称是唯一的！

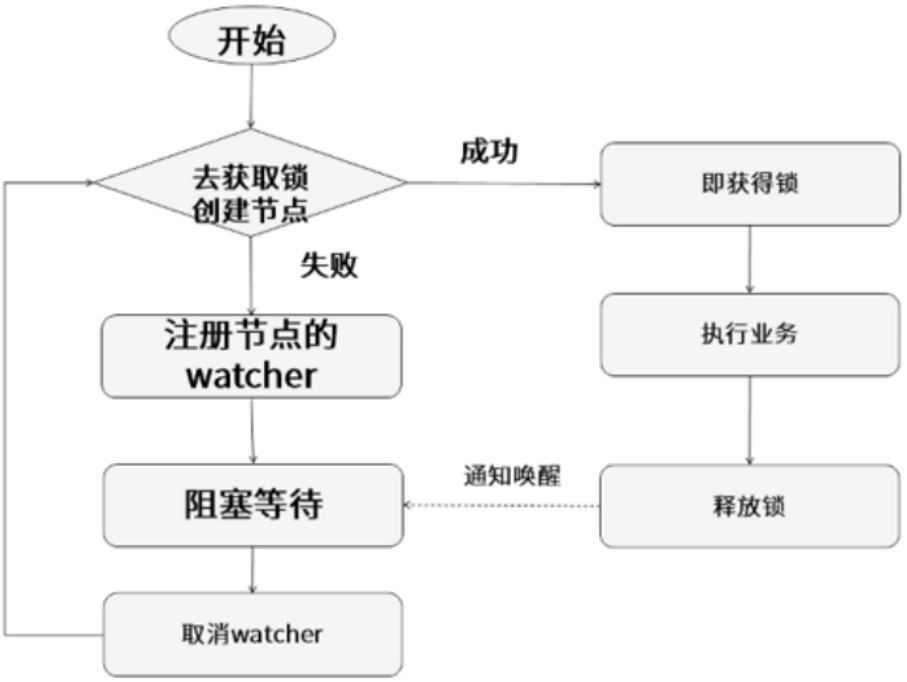
Zookeeper典型应用场景

1. 数据发布订阅（配置中心）
2. 命名服务
3. Master选举
4. 集群管理
5. 分布式队列
6. 分布式锁

4、编程实现

（1）、V1逻辑框图

■ 特性：同父的字节点不可重名

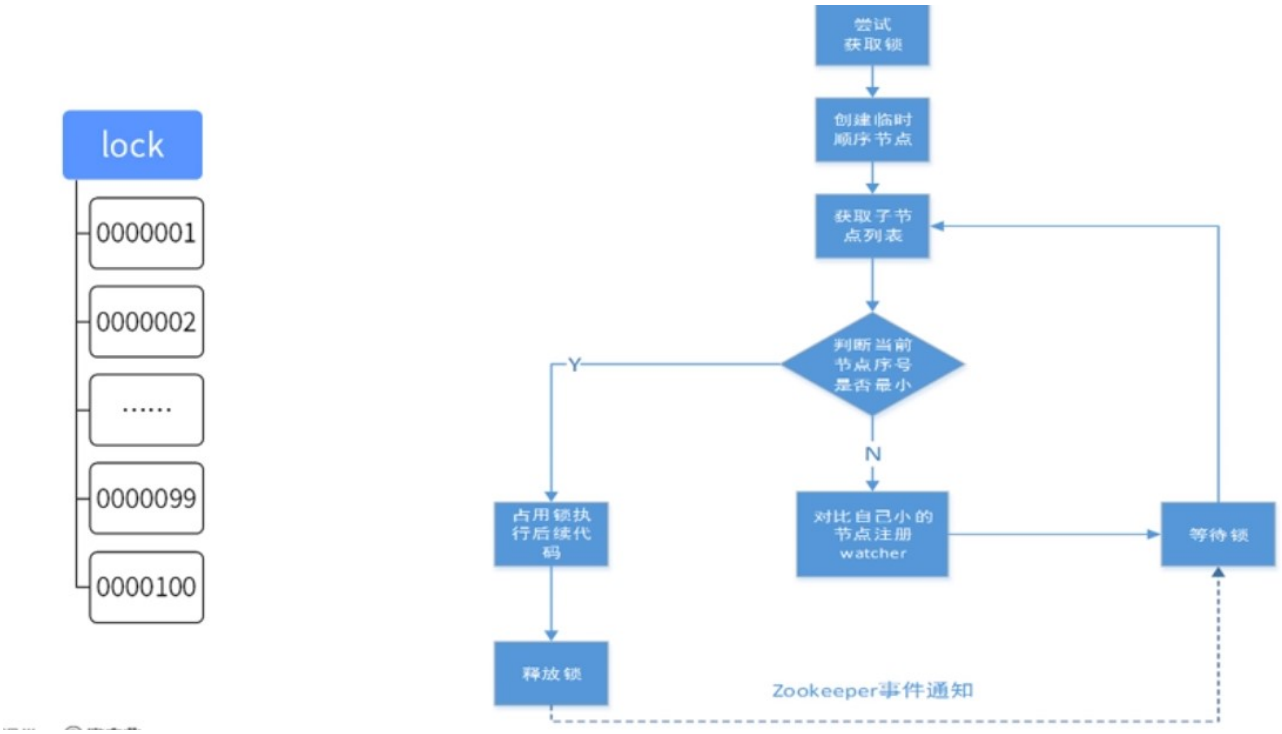


存在的问题：当一个线程解锁之后，所有剩余线程都将会重新抢锁，这种现象被称为惊群现象

在集群规模较大的环境中带来的危害：

- 巨大的服务器性能损耗
- 网络冲击
- 可能造成宕机

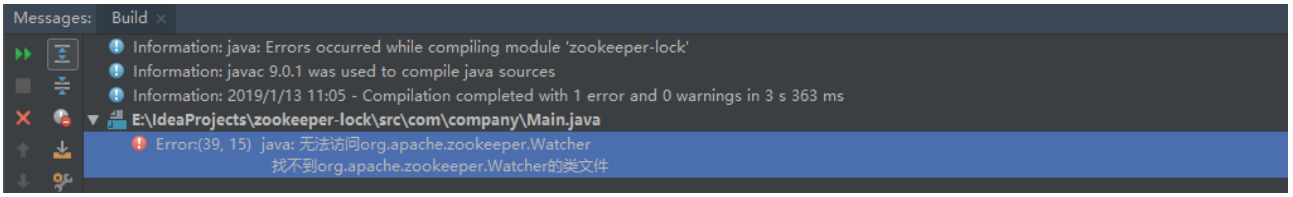
（2）、V2



不存在惊群效应，具体的编程实现参考GitHub中的代码：https://github.com/ShiJinYu2017/zookeeper-lock

5、可能出现的问题

因为程序中我们使用了ZkClient类，所以需要倒入zkclient-0.11.jar包，如果运行时出现如下错误：



说明没有找到Watcher类，需要导入zookeeper-3.4.12.jar包。

运行时如果出现如下错误：

```
Exception in thread "main" java.lang.NoClassDefFoundError: org/slf4j/LoggerFactory
    at org.IOItec.zkclient.ZkClient.<clinit>(ZkClient.java:65)
    at com.company.Main.main(Main.java:38)
Caused by: java.lang.ClassNotFoundException: org.slf4j.LoggerFactory
    at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:582)
    at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:185)
    at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:496)
    ... 2 more

Process finished with exit code 1
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.apache.zookeeper.CreateMode;
```

需要下载相应的slf4j包，下载地址：https://www.slf4j.org/download.html。用户手

册：https://www.slf4j.org/manual.html

其中包含如下内容：

Hello World

As customary in programming tradition, here is an example illustrating the simplest way to output "Hello world" using SLF4J. It begins by getting a logger with the name "HelloWorld". This logger is in turn used to log the message "Hello World".

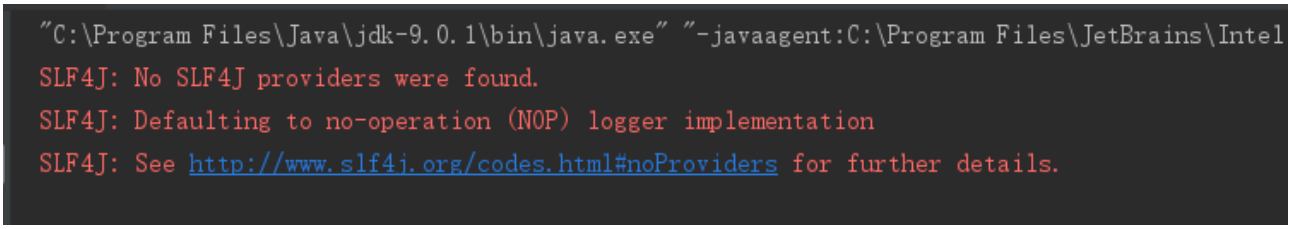
```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloWorld {
  public static void main(String[] args) {
    Logger logger = LoggerFactory.getLogger(HelloWorld.class);
    logger.info("Hello World");
  }
}
```

To run this example, you first need to download the slf4j distribution, and then to unpack it. Once that is done, add the file *slf4j-api-1.8.0-beta2.jar* to your class path.

我们导入slf4j-api-1.8.0-beta2.jar包。

继续运行可能显示如下错误：



SLF4J警告或错误消息及其含义

没有找到SLF4J提供商。

如果在类路径上找不到SLF4J提供程序，则会报告此警告，即非错误消息。在类路径上放置一个（并且只有一个）*slf4j-nop.jar slf4j-simple.jar*，*slf4j-log4j12.jar*，*slf4j-jdk14.jar*或*logback-classic.jar*应该可以解决问题。请注意，这些提供程序必须以slf4j-api 1.8或更高版本为目标。

如果没有提供程序，SLF4J将默认为无操作（NOP）记录器提供程序。

请注意，slf4j-api版本1.8.x及更高版本使用ServiceLoader 机制。早期的版本依赖于slf4j-api不再支持的静态绑定机制。 请阅读FAQ条目SLF4J版本1.8.0中有哪些变化？有关更多重要细节。

如果您负责打包应用程序而不关心日志记录，那么将*slf4j-nop.jar*放在应用程序的类路径上将消除此警告消息。请注意，嵌入式组件（如库或框架）不应声明对任何SLF4J提供程序的依赖性，而只依赖于slf4j-api。当库声明SLF4J提供程序的编译时依赖性时，它会在最终用户上强制使用该提供程序，从而否定SLF4J的用途。

这时需要导入slf4j-nop-1.8.0-beta2.jar包。