

Maven

基本介绍

Mvn介绍

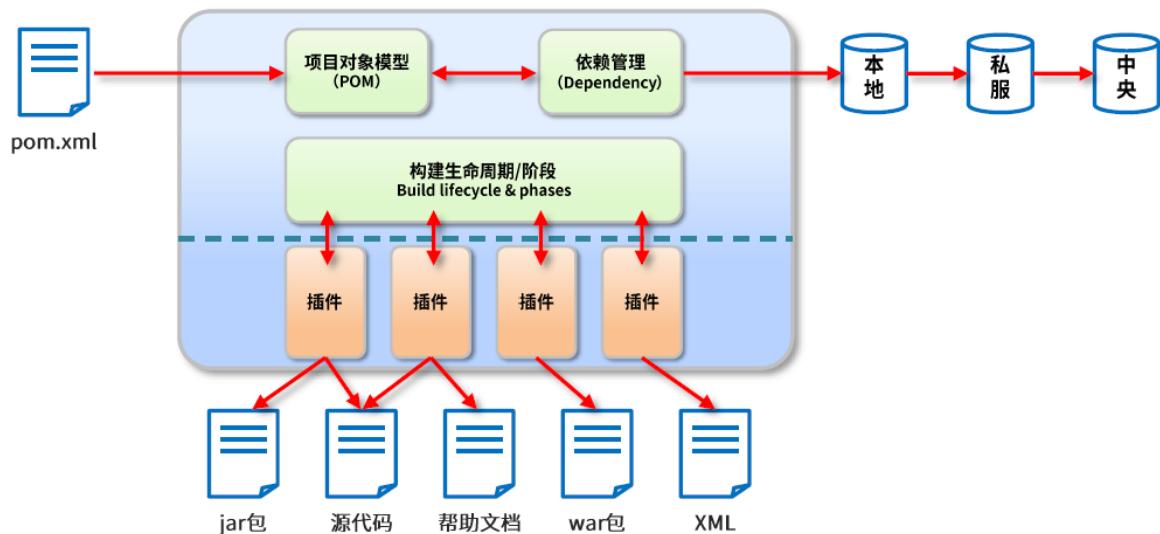
Maven：本质是一个项目管理工具，将项目开发和管理过程抽象成一个项目对象模型（POM）

POM：Project Object Model 项目对象模型。Maven 是用 Java 语言编写的，管理的东西以面向对象的形式进行设计，最终把一个项目看成一个对象，这个对象叫做 POM

pom.xml：Maven 需要一个 pom.xml 文件，Maven 通过加载这个配置文件可以知道项目的相关信息，这个文件代表就一个项目。如果做 8 个项目，对应的是 8 个 pom.xml 文件

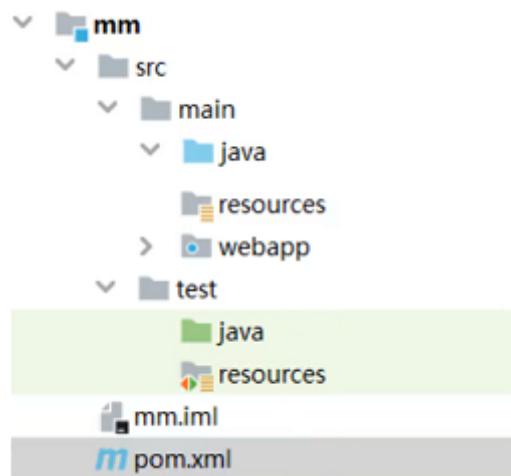
依赖管理：Maven 对项目所有依赖资源的一种管理，它和项目之间是一种双向关系，即做项目时可以管理所需要的其他资源，当其他项目需要依赖我们项目时，Maven 也会把我们的项目当作一种资源去进行管理。

管理资源的存储位置：本地仓库，私服，中央仓库



基本作用：

- 项目构建：提供标准的，跨平台的自动化构建项目的方式
- 依赖管理：方便快捷的管理项目依赖的资源（jar 包），避免资源间的版本冲突等问题
- 统一开发结构：提供标准的，统一的项目开发结构



各目录存放资源类型说明：

- src/main/java：项目 java 源码
- src/main/resources：项目的相关配置文件（比如 mybatis 配置，xml 映射配置，自定义配置文件等）
- src/main/webapp：web 资源（比如 html、css、js 等）
- src/test/java：测试代码
- src/test/resources：测试相关配置文件
- src/pom.xml：项目 pom 文件

参考视频：<https://www.bilibili.com/video/BV1Ah411S7ZE>

基础概念

仓库：用于存储资源，主要是各种 jar 包。有本地仓库，私服，中央仓库，私服和中央仓库都是远程仓库

- 中央仓库：Maven 团队自身维护的仓库，属于开源的
- 私服：各公司/部门等小范围内存储资源的仓库，私服也可以从中央仓库获取资源，作用：
 - 保存具有版权的资源，包含购买或自主研发的 jar
 - 一定范围内共享资源，能做到仅对内不对外开放
- 本地仓库：开发者自己电脑上存储资源的仓库，也可从远程仓库获取资源

坐标：Maven 中的坐标用于描述仓库中资源的位置

- 作用：使用唯一标识，唯一性定义资源位置，通过该标识可以将资源的识别与下载工作交由机器完成
 - <https://mvnrepository.com>：查询 maven 某一个资源的坐标，输入资源名称进行检索
- 依赖设置：
 - groupId：定义当前资源隶属组织名称（通常是域名反写，如：org.mybatis）
 - artifactId：定义当前资源的名称（通常是项目或模块名称，如：crm、sms）
 - version：定义当前资源的版本号
- packaging：定义资源的打包方式，取值一般有如下三种
 - jar：该资源打成 jar 包，默认是 jar
 - war：该资源打成 war 包
 - pom：该资源是一个父资源（表明使用 Maven 分模块管理），打包时只生成一个 pom.xml 不生成 jar 或其他包结构

本章学习了 Maven 的基本概念，包括仓库、坐标、依赖设置、打包方式等。通过这些概念，我们可以更深入地理解 Maven 的工作原理和使用方法。

环境搭建

环境配置

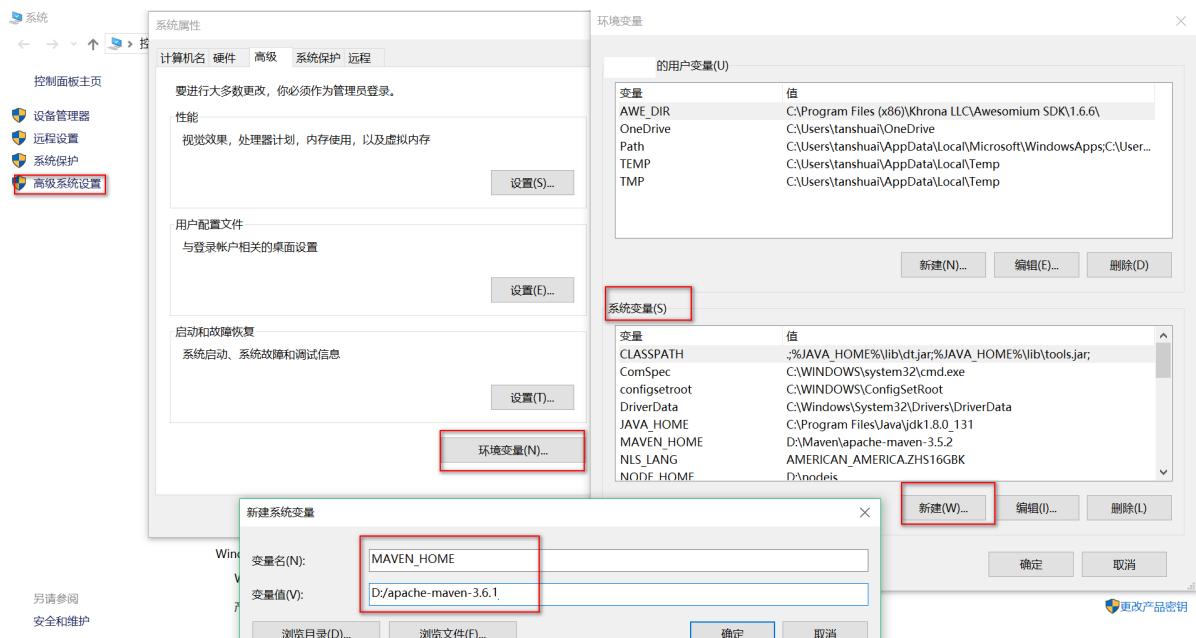
Maven 的官网: <http://maven.apache.org/>

下载安装: Maven 是一个绿色软件, 解压即安装

目录结构:

- bin: 可执行程序目录
- boot: Maven 自身的启动加载器
- conf: Maven 配置文件的存放目录
- lib: Maven 运行所需库的存放目录

配置 MAVEN_HOME:



Path 下配置: %MAVEN_HOME%\bin

环境变量配置好之后需要测试环境配置结果, 在 DOS 命令窗口下输入以下命令查看输出: mvn -v

仓库配置

默认情况 Maven 本地仓库在系统用户目录下的 .m2/repository, 修改 Maven 的配置文件 conf/settings.xml 来修改仓库位置

- 修改本地仓库位置: 找到 `<localRepository>` 标签, 修改默认值

```
<!-- localRepository
| The path to the local repository maven will use to store artifacts.
| Default: ${user.home}/.m2/repository
<localRepository>/path/to/local/repo</localRepository>
-->
<localRepository>E:\workspace\Java\Project\.m2\repository</localRepository>
```

注意：在仓库的同级目录即 `.m2` 也应该包含一个 `settings.xml` 配置文件，局部用户配置优先与全局配置

- 全局 setting 定义了 Maven 的公共配置
- 用户 setting 定义了当前用户的配置
- 修改远程仓库：在配置文件中找到 `<mirrors>` 标签，在这组标签下添加国内镜像

```
<mirror>
  <id>nexus-aliyun</id>
  <mirrorof>central</mirrorof> <!--必须是central-->
  <name>Nexus aliyun</name>
  <url>http://maven.aliyun.com/nexus/content/groups/public</url>
</mirror>
```

- 修改默认 JDK：在配置文件中找到 `<profiles>` 标签，添加配置

```
<profile>
  <id>jdk-10</id>
  <activation>
    <activeByDefault>true</activeByDefault>
    <jdk>10</jdk>
  </activation>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>10</maven.compiler.source>
    <maven.compiler.target>10</maven.compiler.target>
  </properties>
</profile>
```

项目搭建

手动搭建

1. 在 E 盘下创建目录 `mvnproject` 进入该目录，作为我们的操作目录
2. 创建我们的 Maven 项目，创建一个目录 `project-java` 作为我们的项目文件夹，并进入到该目录
3. 创建 Java 代码（源代码）所在目录，即创建 `src/main/java`
4. 创建配置文件所在目录，即创建 `src/main/resources`
5. 创建测试源代码所在目录，即创建 `src/test/java`
6. 创建测试存放配置文件存放目录，即 `src/test/resources`
7. 在 `src/main/java` 中创建一个包（注意在 Windows 文件夹下就是创建目录）`demo`，在该目录下创建 `Demo.java` 文件，作为演示所需 Java 程序，内容如下

```
package demo;
public class Demo{
    public String say(String name){
        System.out.println("hello "+name);
        return "hello "+name;
    }
}
```

8. 在 `src/test/java` 中创建一个测试包（目录）`demo`，在该包下创建测试程序 `DemoTest.java`

```
package demo;
import org.junit.*;
public class DemoTest{
    @Test
    public void testSay(){
        Demo d = new Demo();
        String ret = d.say("maven");
        Assert.assertEquals("hello maven",ret);
    }
}
```

9. 在 `project-java/src` 下创建 `pom.xml` 文件，格式如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/maven-v4_0_0.xsd">

    <!--指定pom的模型版本-->
    <modelVersion>4.0.0</modelVersion>
    <!--打包方式， web工程打包为war, java工程打包为jar -->
    <packaging>jar</packaging>

    <!--组织id-->
    <groupId>demo</groupId>
    <!--项目id-->
    <artifactId>project-java</artifactId>
    <!--版本号:release,snapshot-->
    <version>1.0</version>

    <!--设置当前工程的所有依赖-->
    <dependencies>
        <!--具体的依赖-->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
</project>
```

10. 搭建完成 Maven 的项目结构，通过 Maven 来构建项目。Maven 的构建命令以 `mvn` 开头，后面添加功能参数，可以一次性执行多个命令，用空格分离

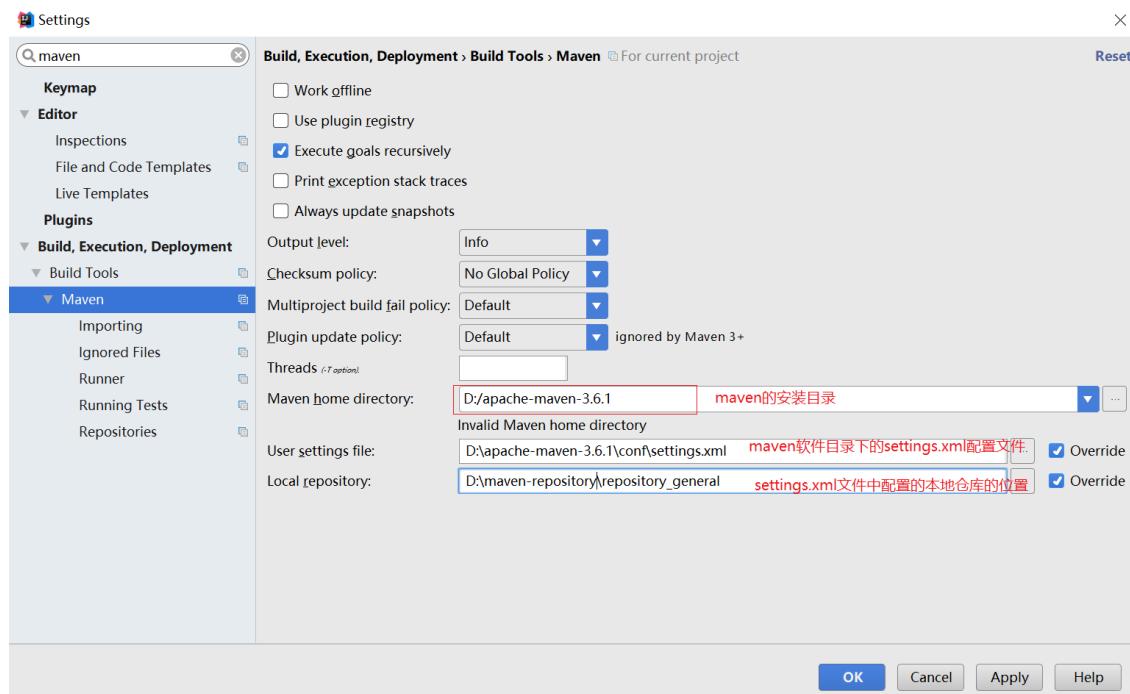
- `mvn compile`：编译
- `mvn clean`：清理
- `mvn test`：测试
- `mvn package`：打包
- `mvn install`：安装到本地仓库

注意：执行某一条命令，则会把前面所有的都执行一遍

IDEA搭建

不用原型

1. 在 IDEA 中配置 Maven，选择 maven3.6.1 防止依赖问题

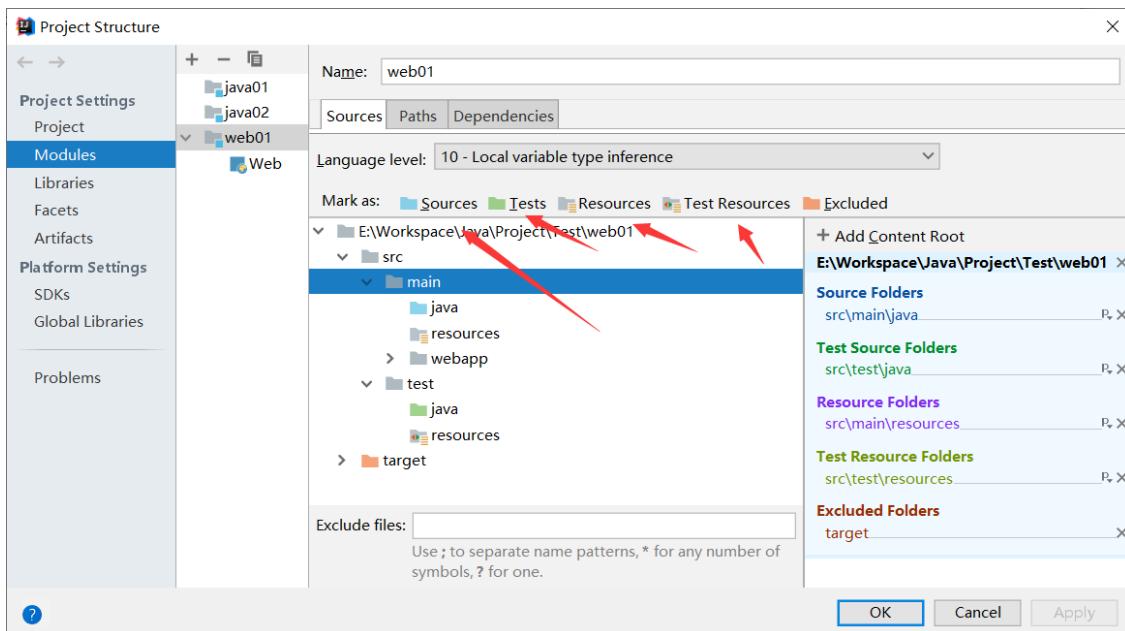


2. 创建 Maven，New Module → Maven → 不选中 Create from archetype

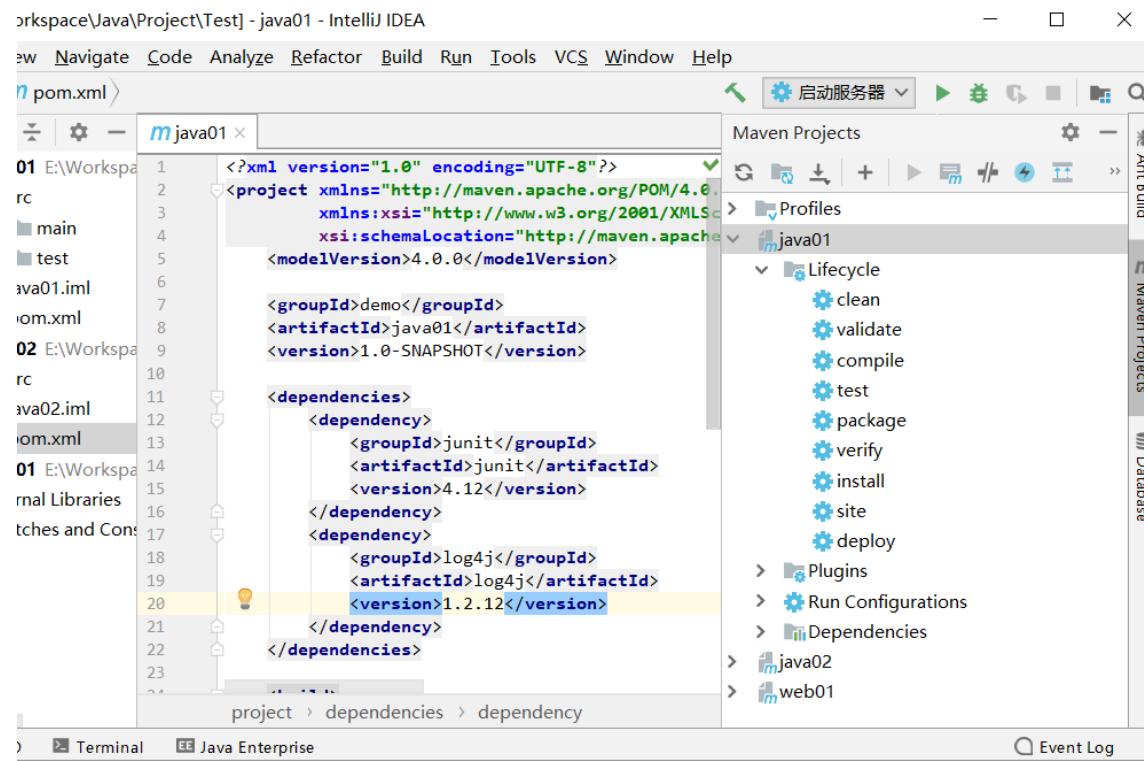
3. 填写项目的坐标

- GroupId: demo
- ArtifactId: project-java

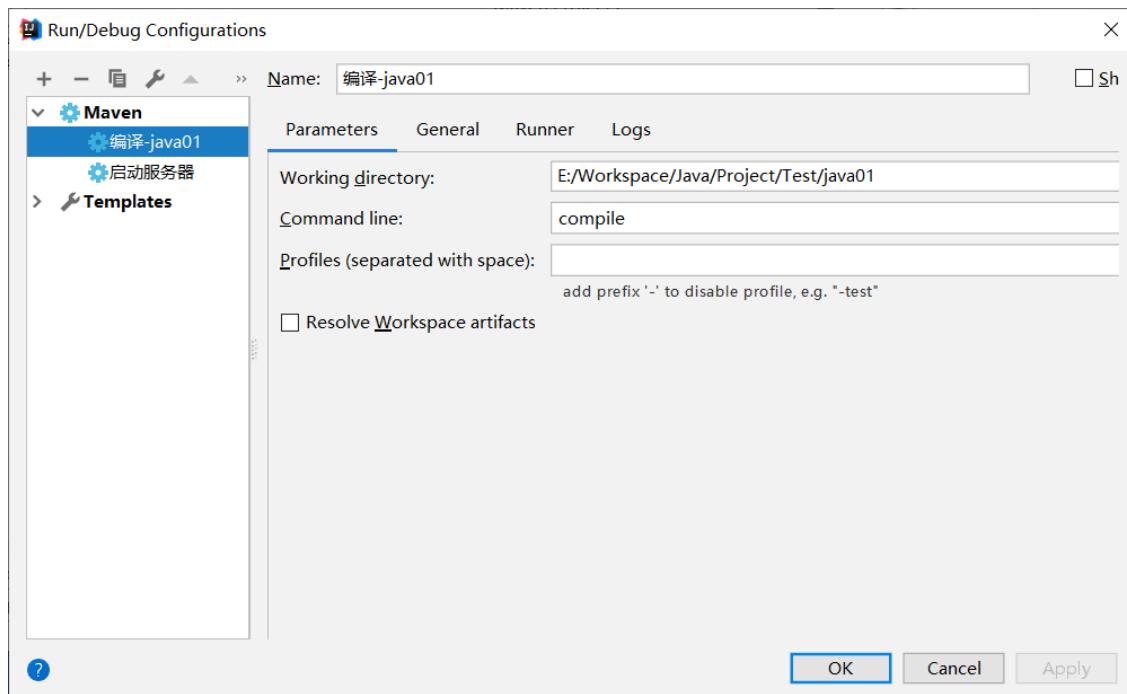
4. 查看各目录颜色标记是否正确



5. IDEA 右侧侧栏有 Maven Project, 打开后有 Lifecycle 生命周期



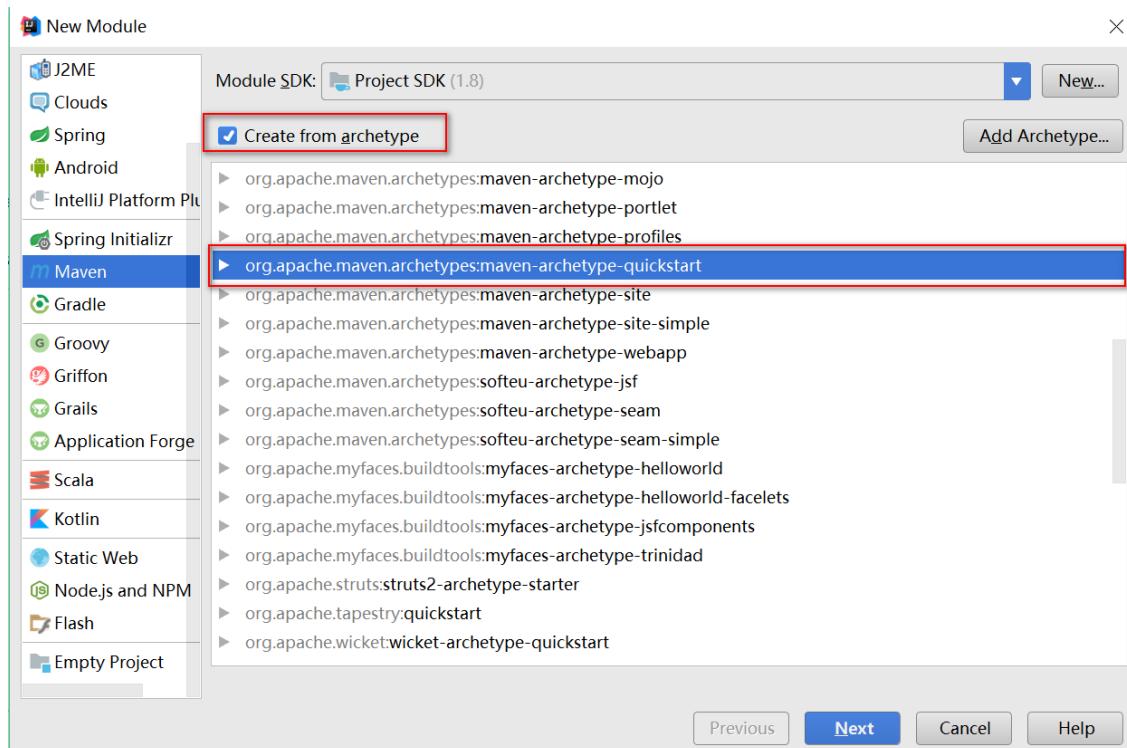
6. 自定义 Maven 命令: Run → Edit Configurations → 左上角 + → Maven



使用原型

普通工程：

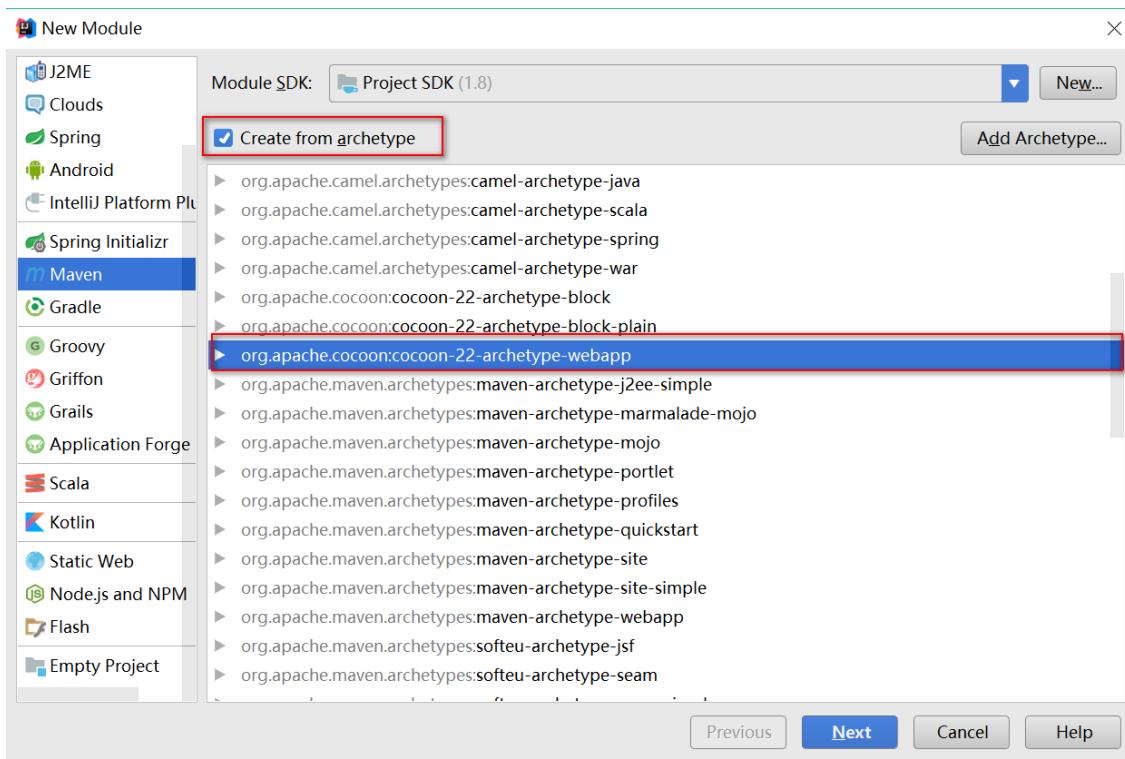
1. 创建 Maven 项目的时候选择使用原型骨架



2. 创建完成后发现通过这种方式缺少一些目录，需要手动去补全目录，并且要对补全的目录进行标记

Web 工程：

1. 选择 Web 对应的原型骨架（选择 Maven 开头的是简化的）



2. 通过原型创建 Web 项目得到的目录结构是不全的，因此需要我们自行补全，同时要标记正确
3. Web 工程创建之后需要启动运行，使用 tomcat 插件来运行项目，在 `pom.xml` 中添加插件的坐标：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <packaging>war</packaging>

    <name>web01</name>
    <groupId>demo</groupId>
    <artifactId>web01</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
    </dependencies>

    <!--构建-->
    <build>
        <!--设置插件-->
        <plugins>
            <!--具体的插件配置-->
            <plugin>
                <!--https://mvnrepository.com/ 搜索-->
                <groupId>org.apache.tomcat.maven</groupId>
                <artifactId>tomcat7-maven-plugin</artifactId>
                <version>2.1</version>
                <configuration>
                    <port>80</port> <!--80端口默认不显示-->
                    <path>/</path>
                
```

```
</configuration>
</plugin>
</plugins>
</build>
</project>
```

4. 插件配置以后，在 IDEA 右侧 maven-project 操作面板看到该插件，并且可以利用该插件启动项目，web01 → Plugins → tomcat7 → tomcat7:run
-

依赖管理

依赖配置

依赖是指在当前项目中运行所需的 jar，依赖配置的格式如下：

```
<!--设置当前项目所依赖的所有jar-->
<dependencies>
    <!--设置具体的依赖-->
    <dependency>
        <!--依赖所属群组id-->
        <groupId>junit</groupId>
        <!--依赖所属项目id-->
        <artifactId>junit</artifactId>
        <!--依赖版本号-->
        <version>4.12</version>
    </dependency>
</dependencies>
```

依赖传递

依赖具有传递性，分两种：

- 直接依赖：在当前项目中通过依赖配置建立的依赖关系
- 间接依赖：被依赖的资源如果依赖其他资源，则表明当前项目间接依赖其他资源

注意：直接依赖和间接依赖其实也是一个相对关系

依赖传递的冲突问题：在依赖传递过程中产生了冲突，有三种优先法则

- 路径优先：当依赖中出现相同资源时，层级越深，优先级越低，反之则越高
- 声明优先：当资源在相同层级被依赖时，配置顺序靠前的覆盖靠后的
- 特殊优先：当同级配置了相同资源的不同版本时，后配置的覆盖先配置的

可选依赖：对外隐藏当前所依赖的资源，不透明

```

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <optional>true</optional>
    <!--默认是false, true以后就变得不透明-->
</dependency>

```

排除依赖：主动断开依赖的资源，被排除的资源无需指定版本

```

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <exclusions>
        <exclusion>
            <groupId>org.hamcrest</groupId> <!--排除这个资源-->
            <artifactId>hamcrest-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>

```

依赖范围

依赖的 jar 默认情况可以在任何地方可用，可以通过 `scope` 标签设定其作用范围，有三种：

- 主程序范围有效 (src/main 目录范围内)
- 测试程序范围内有效 (src/test 目录范围内)
- 是否参与打包 (package 指令范围内)

`scope` 标签的取值有四种： `compile, test, provided, runtime`

scope	主代码	测试代码	打包	范例
<code>compile</code> (默认)	Y	Y	Y	<code>log4j</code>
<code>test</code>		Y		<code>junit</code>
<code>provided</code>	Y	Y		<code>servlet-api</code>
<code>runtime</code>			Y	<code>jdbc</code>

依赖范围的传递性：

带有依赖范围的资源在进行传递时，作用范围将受到影响

	compile	test	provided	runtime	← 直接依赖
compile	compile	test	provided	runtime	
test					
provided					
runtime	runtime	test	provided	runtime	

间接依赖

生命周期

相关事件

Maven 的构建生命周期描述的是一次构建过程经历了多少个事件

最常用的一套流程： compile → test-compile → test → package → install

- clean: 清理工作
 - pre-clean: 执行一些在 clean 之前的工作
 - clean: 移除上一次构建产生的所有文件
 - post-clean: 执行一些在 clean 之后立刻完成的工作
- default: 核心工作，例如编译，测试，打包，部署等，每个事件在执行之前都会**将之前的所有事件**

依次执行一遍

- | | |
|--------------------------------------|--------------------------------------|
| ● validate (校验) | 校验项目是否正确并且所有必要的信息可以完成项目的构建过程。 |
| ● initialize (初始化) | 初始化构建状态，比如设置属性值。 |
| ● generate-sources (生成源代码) | 生成包含在编译阶段中的任何源代码。 |
| ● process-sources (处理源代码) | 处理源代码，比如说，过滤任意值。 |
| ● generate-resources (生成资源文件) | 生成将会包含在项目包中的资源文件。 |
| ● process-resources (处理资源文件) | 复制和处理资源到目标目录，为打包阶段最好准备。 |
| ● compile (编译) | 编译项目的源代码。 |
| ● process-classes (处理类文件) | 处理编译生成的文件，比如说对Java class文件做字节码改善优化。 |
| ● generate-test-sources (生成测试源代码) | 生成包含在编译阶段中的任何测试源代码。 |
| ● process-test-sources (处理测试源代码) | 处理测试源代码，比如说，过滤任意值。 |
| ● generate-test-resources (生成测试资源文件) | 为测试创建资源文件。 |
| ● process-test-resources (处理测试资源文件) | 复制和处理测试资源到目标目录。 |
| ● test-compile (编译测试原码) | 编译测试源代码到测试目标目录。 |
| ● process-test-classes (处理测试类文件) | 处理测试源码编译生成的文件。 |
| ● test (测试) | 使用合适的单元测试框架运行测试 (Junit是其中之一)。 |
| ● prepare-package (准备打包) | 在实际打包之前，执行任何的必要的操作为打包做准备。 |
| ● package (打包) | 将编译后的代码打包成可分发格式的文件，比如JAR、WAR或者EAR文件。 |
| ● pre-integration-test (集成测试前) | 在执行集成测试前进行必要的动作。比如说，搭建需要的环境。 |
| ● integration-test (集成测试) | 处理和部署项目到可以运行集成测试环境中。 |
| ● post-integration-test (集成测试后) | 在执行集成测试完成后进行必要的动作。比如说，清理集成测试环境。 |
| ● verify (验证) | 运行任意的检查来验证项目包有效且达到质量标准。 |
| ● install (安装) | 安装项目包到本地仓库，这样项目包可以用作其他本地项目的依赖。 |
| ● deploy (部署) | 将最终的项目包复制到远程仓库中与其他开发者和项目共享。 |

- site: 产生报告，发布站点等

- pre-site: 执行一些在生成站点文档之前的工作
 - site: 生成项目的站点文档
 - post-site: 执行一些在生成站点文档之后完成的工作，并为部署做准备
 - site-deploy: 将生成的站点文档部署到特定的服务器上
-

执行事件

Maven 的插件用来执行生命周期中的相关事件

- 插件与生命周期内的阶段绑定，在执行到对应生命周期时执行对应的插件
- Maven 默认在各个生命周期上都绑定了预先设定的插件来完成相应功能
- 插件还可以完成一些自定义功能

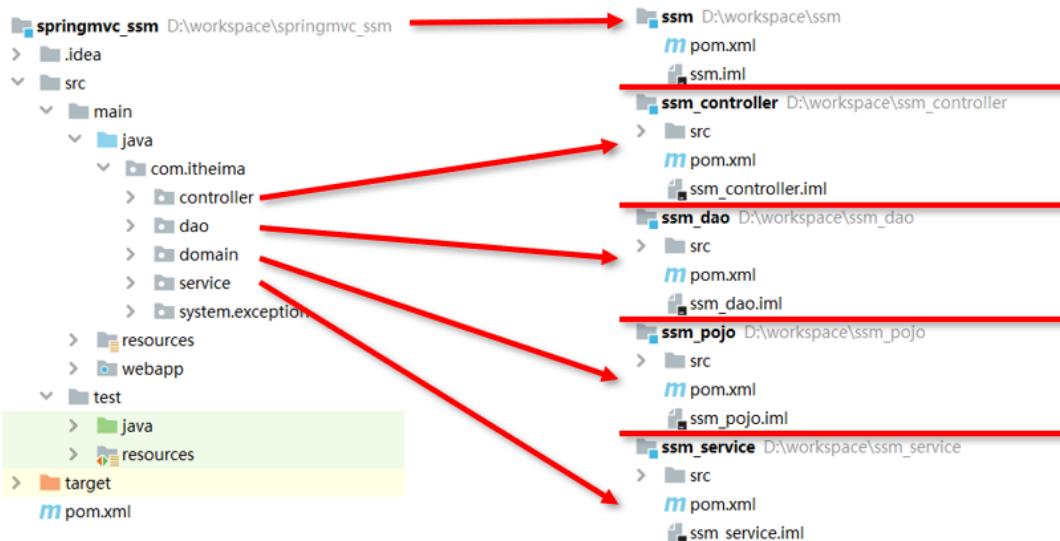
```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-source-plugin</artifactId>
      <version>2.2.1</version>
      <!--执行-->
      <executions>
        <!--具体执行位置-->
        <execution>
          <goals>
            <!--对源码进行打包，打包放在target目录-->

              <goal>jar</goal>
              <!--对测试代码进行打包-->
              <goal>test-jar</goal>
            </goals>
            <!--执行的生命周期-->
            <phase>generate-test-resources</phase>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
```

模块开发

拆分

工程模块与模块划分：



- `ssm_pojo` 拆分

- 新建模块，拷贝原始项目中对应的相关内容到 `ssm_pojo` 模块中
- 实体类 (`User`)
- 配置文件 (无)

- `ssm_dao` 拆分

- 新建模块
- 拷贝原始项目中对应的相关内容到 `ssm_dao` 模块中
 - 数据层接口 (`UserDao`)
 - 配置文件：保留与数据层相关配置文件(3个)
 - 注意：分页插件在配置中与 `SqlSessionFactoryBean` 绑定，需要保留
 - `pom.xml`：引入数据层相关坐标即可，删除 SpringMVC 相关坐标
 - Spring
 - MyBatis
 - Spring 整合 MyBatis
 - MySQL
 - druid
 - pagehelper
 - 直接依赖 `ssm_pojo` (对 `ssm_pojo` 模块执行 `install` 指令，将其安装到本地仓库)

```
<dependencies>      <!--导入资源文件pojo-->
    <dependency>
        <groupId>demo</groupId>
        <artifactId>ssm_pojo</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <!--spring环境-->
    <!--mybatis环境-->
    <!--mysql环境-->
    <!--spring整合jdbc-->
    <!--spring整合mybatis-->
    <!--druid连接池-->
    <!--分页插件坐标-->
```

```
</dependencies>
```

- ssm_service 拆分
 - 新建模块
 - 拷贝原始项目中对应的相关内容到 ssm_service 模块中
 - 业务层接口与实现类 (UserService、UserServiceImpl)
 - 配置文件：保留与数据层相关配置文件(1 个)
 - pom.xml：引入数据层相关坐标即可，删除 SpringMVC 相关坐标
 - spring
 - junit
 - spring 整合 junit
 - 直接依赖 ssm_dao (对 ssm_dao 模块执行 install 指令，将其安装到本地仓库)
 - 间接依赖 ssm_pojo (由 ssm_dao 模块负责依赖关系的建立)
 - 修改 service 模块 Spring 核心配置文件名，添加模块名称，格式：applicationContext-service.xml
 - 修改 dao 模块 Spring 核心配置文件名，添加模块名称，格式：applicationContext-dao.xml
 - 修改单元测试引入的配置文件名称，由单个文件修改为多个文件
 - ssm_control 拆分
 - 新建模块 (使用 webapp 模板)
 - 拷贝原始项目中对应的相关内容到 ssm_controller 模块中
 - 现层控制器类与相关设置类 (UserController、异常相关.....)
 - 配置文件：保留与表现层相关配置文件(1 个)、服务器相关配置文件 (1 个)
 - pom.xml：引入数据层相关坐标即可，删除 SpringMVC 相关坐标
 - spring
 - springmvc
 - jackson
 - servlet
 - tomcat 服务器插件
 - 直接依赖 ssm_service (对 ssm_service 模块执行 install 指令，将其安装到本地仓库)
 - 间接依赖 ssm_dao、ssm_pojo

```
<dependencies>
    <!--导入资源文件service-->
    <dependency>
        <groupId>demo</groupId>
        <artifactId>ssm_service</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <!--springmvc环境-->
    <!--jackson相关坐标3个-->
    <!--servlet环境-->
</dependencies>
<build>
    <!--设置插件-->
```

```
<plugins>
    <!--具体的插件配置-->
    <plugin>
    </plugin>
</plugins>
</build>
```

- 修改 web.xml 配置文件中加载 Spring 环境的配置文件名称，使用*通配，加载所有 applicationContext- 开始的配置文件：

```
<!--加载配置文件-->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath*:applicationContext-*.xml</param-value>
</context-param>
```

- spring-mvc

```
<mvc:annotation-driven/><context:component-scan base-
package="controller"/>
```

聚合

作用：聚合用于快速构建 Maven 工程，一次性构建多个项目/模块

制作方式：

- 创建一个空模块，打包类型定义为 pom

```
<packaging>pom</packaging>
```

- 定义当前模块进行构建操作时关联的其他模块名称

```
<?xml version="1.0" encoding="UTF-8"?><project xmlns=".....">
    <modelVersion>4.0.0</modelVersion>
    <groupId>demo</groupId>
    <artifactId>ssm</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!--定义该工程用于构建管理-->
    <packaging>pom</packaging>
    <!--管理的工程列表-->
    <modules>
        <!--具体的工程名称-->
        <module>../ssm_pojo</module>
        <module>../ssm_dao</module>
        <module>../ssm_service</module>
        <module>../ssm_controller</module>
    </modules>
</project>
```

继承

Maven 中的继承与 Java 中的继承相似，可以实现在子工程中沿用父工程中的配置

dependencyManagement 里只是声明依赖，并不实现引入，所以子工程需要显式声明需要的依赖

- 如果子工程中未声明依赖，则不会从父项目继承下来
- 在子工程中声明该依赖项，并且不指定具体版本，才会从父项目中继承该项，version 和 scope 都继承取自父工程 pom 文件
- 如果子工程中指定了版本号，那么使用子工程中指定的 jar 版本

制作方式：

- 在子工程中声明其父工程坐标与对应的位置

```
<!--定义该工程的父工程-->
<parent>
    <groupId>com.seazean</groupId>
    <artifactId>ssm</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!--填写父工程的pom文件-->
    <relativePath>../ssm/pom.xml</relativePath>
</parent>
```

- 继承依赖的定义：在父工程中定义依赖管理

```
<!--声明此处进行依赖管理，版本锁定-->
<dependencyManagement>
    <!--具体的依赖-->
    <dependencies>
        <!--spring环境-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.9.RELEASE</version>
        </dependency>
        <!--等等所有-->
    </dependencies>
</dependencyManagement>
```

- 继承依赖的使用：在子工程中定义依赖关系，**无需声明依赖版本**，版本参照父工程中依赖的版本

```
<dependencies>
    <!--spring环境-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
    </dependency>
</dependencies>
```

- 继承的资源：

groupId: 项目组ID，项目坐标的核心元素
version: 项目版本，项目坐标的核心因素
description: 项目的描述信息
organization: 项目的组织信息
inceptionYear: 项目的创始年份
url: 项目的URL地址
developers: 项目的开发者信息
contributors: 项目的贡献者信息
distributionManagement: 项目的部署配置
issueManagement: 项目的缺陷跟踪系统信息
ciManagement: 项目的持续集成系统信息
scm: 项目的版本控制系统信息
mailingLists: 项目的邮件列表信息
properties: 自定义的Maven属性
dependencies: 项目的依赖配置
dependencyManagement: 项目的依赖管理配置
repositories: 项目的仓库配置
build: 包括项目的源码目录配置、输出目录配置、插件配置、插件管理配置等
reporting: 包括项目的报告输出目录配置、报告插件配置等

- 继承与聚合：

作用：

- 聚合用于快速构建项目
- 继承用于快速配置

相同点：

- 聚合与继承的 pom.xml 文件打包方式均为 pom，可以将两种关系制作到同一个 pom 文件中
- 聚合与继承均属于设计型模块，并无实际的模块内容

不同点：

- 聚合是在当前模块中配置关系，聚合可以感知到参与聚合的模块有哪些
- 继承是在子模块中配置关系，父模块无法感知哪些子模块继承了自己

属性

- 版本统一的重要性：

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.9.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.3</version>
</dependency>
<!--spring整合jdbca-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>
```

```
String name = "Jock";
System.out.println(name);
```

```
String spring_version = "5.1.9.RELEASE";
System.out.println(spring_version);
```

- 属性类别：

1. 自定义属性
2. 内置属性
3. setting 属性
4. Java 系统属性
5. 环境变量属性

- 自定义属性：

作用：等同于定义变量，方便统一维护

定义格式：

```
<!--定义自定义属性，放在dependencyManagement上方-->
<properties>
    <spring.version>5.1.9.RELEASE</spring.version>
    <junit.version>4.12</junit.version>
</properties>
```

- 聚合与继承的 pom.xml 文件打包方式均为 pom，可以将两种关系制作到同一个 pom 文件中
- 聚合与继承均属于设计型模块，并无实际的模块内容

调用格式：

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
</dependency>
```

- 内置属性：

作用：使用 Maven 内置属性，快速配置

调用格式：

```
 ${project.basedir} or ${project.basedir} <!-- ../ssm根目录--> ${version} or  
 ${project.version}
```

- version 是 1.0-SNAPSHOT

```
<groupId>demo</groupId>  
<artifactId>ssm</artifactId>  
<version>1.0-SNAPSHOT</version>
```

- setting 属性

- 使用 Maven 配置文件 setting.xml 中的标签属性，用于动态配置

调用格式：

```
 ${settings.localRepository}
```

- Java 系统属性：

作用：读取 Java 系统属性

调用格式：

```
 ${user.home}
```

系统属性查询方式 cmd 命令：

```
 mvn help:system
```

- 环境变量属性

作用：使用 Maven 配置文件 setting.xml 中的标签属性，用于动态配置

调用格式：

```
 ${env.JAVA_HOME}
```

环境变量属性查询方式：

```
 mvn help:system
```

工程版本

SNAPSHOT (快照版本)

- 项目开发过程中，为方便团队成员合作，解决模块间相互依赖和时时更新的问题，开发者对每个模块进行构建的时候，输出的临时性版本叫快照版本（测试阶段版本）
- 快照版本会随着开发的进展不断更新

RELEASE (发布版本)

- 项目开发到进入阶段里程碑后，向团队外部发布较为稳定的版本，这种版本所对应的构件文件是稳定的，即便进行功能的后续开发，也不会改变当前发布版本内容，这种版本称为发布版本

约定规范：

- <主版本>.<次版本>.<增量版本>.<里程碑版本>
- 主版本：表示项目重大架构的变更，如：Spring5 相较于 Spring4 的迭代
- 次版本：表示有较大的功能增加和变化，或者全面系统地修复漏洞
- 增量版本：表示有重大漏洞的修复
- 里程碑版本：表明一个版本的里程碑（版本内部）。这样的版本同下一个正式版本相比，相对来说不是很稳定，有待更多的测试

资源配置

作用：在任意配置文件中加载 pom 文件中定义的属性

- 父文件 pom.xml

```
<properties>
    <jdbc.url>jdbc:mysql://192.168.0.137:3306/ssm_db?useSSL=false</jdbc.url>
</properties>
```

- 开启配置文件加载 pom 属性：

```
<!--配置资源文件对应的信息-->
<resources>
    <resource>
        <!--设定配置文件对应的位置目录，支持使用属性动态设定路径-->
        <directory>${project.basedir}/src/main/resources</directory>
        <!--开启对配置文件的资源加载过滤-->
        <filtering>true</filtering>
    </resource>
</resources>
```

- properties 文件中调用格式：

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=${jdbc.url}
jdbc.username=root
jdbc.password=123456
```

多环境配置

- 环境配置

```
<!--创建多环境-->
<profiles>
    <!--定义具体的环境：生产环境-->
    <profile>
        <!--定义环境对应的唯一名称-->
        <id>pro_env</id>
        <!--定义环境中专用的属性值-->
        <properties>
            <jdbc.url>jdbc:mysql://127.1.1.1:3306/ssm_db</jdbc.url>
        </properties>
        <!--设置默认启动-->
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
    </profile>
    <!--定义具体的环境：开发环境-->
    <profile>
        <id>dev_env</id>
        .....
    </profile>
</profiles>
```

- 加载指定环境

作用：加载指定环境配置

调用格式：

```
mvn 指令 -P 环境定义id
```

范例：

```
mvn install -P pro_env
```

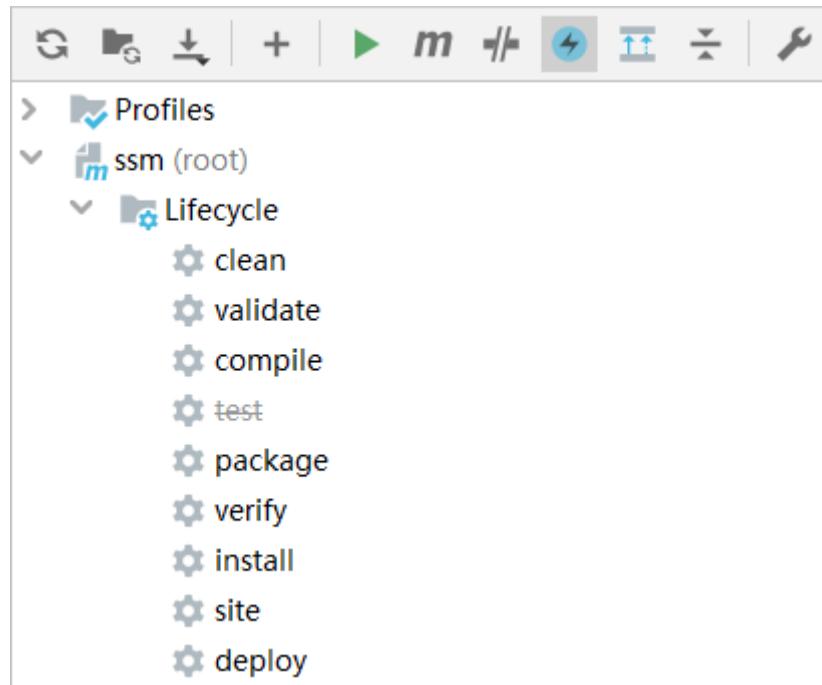
跳过测试

命令：

```
mvn 指令 -D skipTests
```

注意事项：执行的指令生命周期必须包含测试环节

IEDA 界面：



配置跳过：

```
<plugin>
    <!--<groupId>org.apache.maven</groupId>-->
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.1</version>
    <configuration>
        <skipTests>true</skipTests><!--设置跳过测试-->
        <includes> <!--包含指定的测试用例-->
            <include>**/User*Test.java</include>
        </includes>
        <excludes><!--排除指定的测试用例-->
            <exclude>**/User*TestCase.java</exclude>
        </excludes>
    </configuration>
</plugin>
```

私服

Nexus

Nexus 是 Sonatype 公司的一款 Maven 私服产品

下载地址：<https://help.sonatype.com/repomanager3/download>

启动服务器（命令行启动）：

```
nexus.exe /run nexus
```

访问服务器（默认端口：8081）：

<http://localhost:8081>

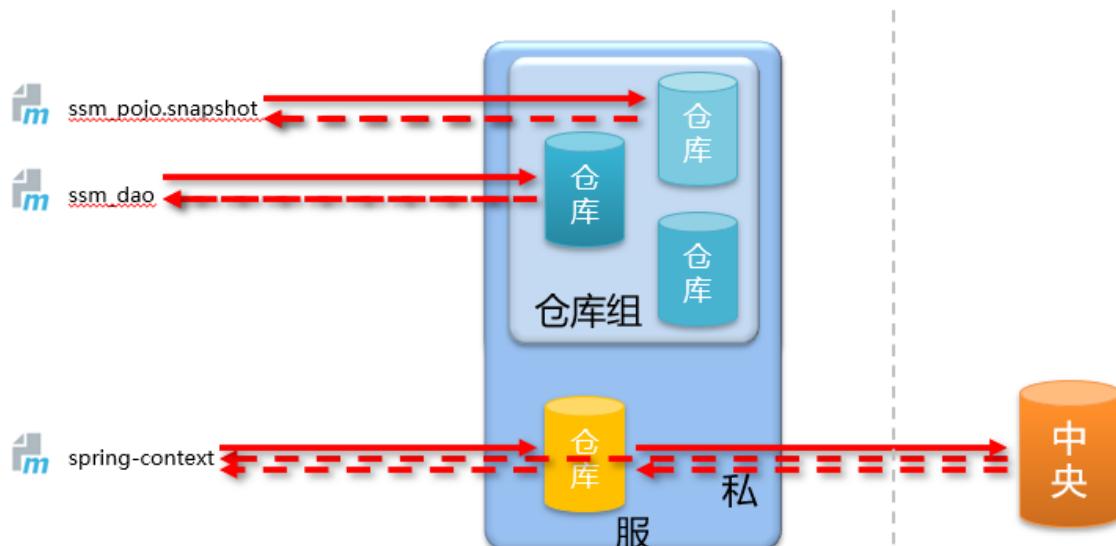
修改基础配置信息

- 安装路径下 etc 目录中 nexus-default.properties 文件保存有 nexus 基础配置信息，例如默认访问端口

修改服务器运行配置信息

- 安装路径下 bin 目录中 nexus.vmoptions 文件保存有 nexus 服务器启动的配置信息，例如默认占用内存空间

资源操作



仓库分类：

- 宿主仓库 hosted
 - 保存无法从中央仓库获取的资源
 - 自主研发
 - 第三方非开源项目
- 代理仓库 proxy
 - 代理远程仓库，通过 nexus 访问其他公共仓库，例如中央仓库
- 仓库组 group
 - 将若干个仓库组成一个群组，简化配置
 - 仓库组不能保存资源，属于设计型仓库

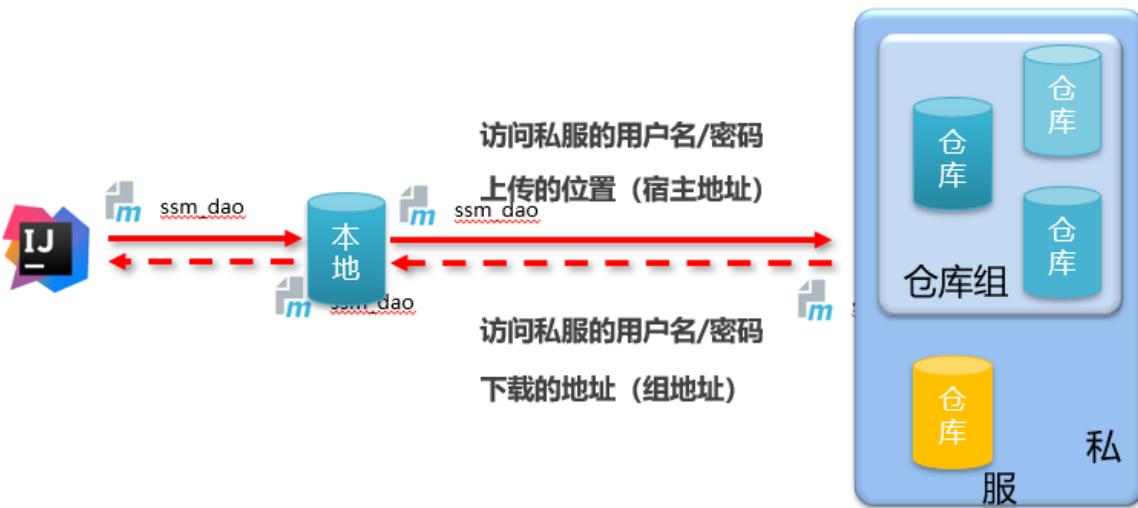
资源上传，上传资源时提供对应的信息

- 保存的位置（宿主仓库）

- 资源文件
 - 对应坐标
-

IDEA操作

上传下载



访问私服

本地访问

配置本地仓库访问私服的权限 (setting.xml)

```
<servers>
  <server>
    <id>heima-release</id>
    <username>admin</username>
    <password>admin</password>
  </server>
  <server>
    <id>heima-snapshots</id>
    <username>admin</username>
    <password>admin</password>
  </server>
</servers>
```

配置本地仓库资源来源 (setting.xml)

```
<mirrors>
  <mirror>
    <id>nexus-heima</id>
    <mirrorof>*</mirrorof>
    <url>http://localhost:8081/repository/maven-public/</url>
  </mirror>
</mirrors>
```

工程访问

配置当前项目访问私服上传资源的保存位置 (pom.xml)

```
<distributionManagement>
  <repository>
    <id>heima-release</id>
    <url>http://localhost:8081/repository/heima-release/</url>
  </repository>
  <snapshotRepository>
    <id>heima-snapshots</id>
    <url>http://localhost:8081/repository/heima-snapshots/</url>
  </snapshotRepository>
</distributionManagement>
```

发布资源到私服命令

```
mvn deploy
```

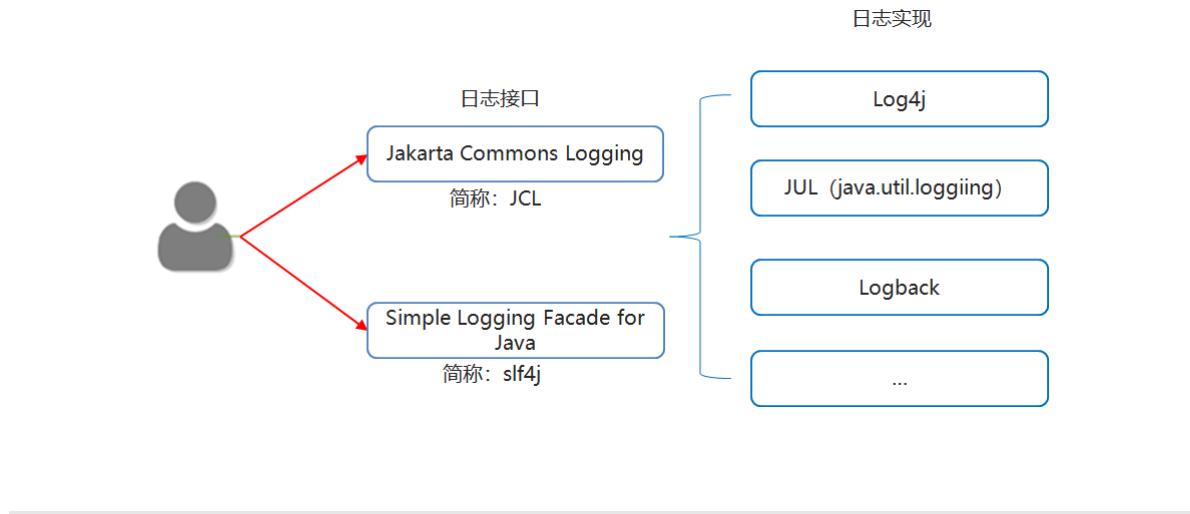
日志

Log4j

程序中的日志可以用来记录程序在运行时候的详情，并可以进行永久存储。

	输出语句	日志技术
取消日志	需要修改代码，灵活性比较差	不需要修改代码，灵活性比较好
输出位置	只能是控制台	可以将日志信息写入到文件或者数据库中
多线程	和业务代码处于一个线程中	多线程方式记录日志，不影响业务代码的性能

Log4j 是 Apache 的一个开源项目。使用 Log4j，通过一个配置文件来灵活地进行配置，而不需要修改应用的代码。我们可以控制日志信息输送的目的地是控制台、文件等位置，也可以控制每一条日志的输出格式。



配置文件

配置文件的三个核心：

- 配置根 Logger
 - 格式：log4j.rootLogger=日志级别, appenderName1, appenderName2, ...
 - 日志级别：常见的五个级别：**DEBUG < INFO < WARN < ERROR < FATAL**（可以自定义）
Log4j 规则：只输出级别不低于设定级别的日志信息
 - appenderName1：指定日志信息要输出地址。可以同时指定多个输出目的地，用逗号隔开：
例如：log4j.rootLogger = INFO, ca, fa
- Appenders（输出源）：日志要输出的地方，如控制台（Console）、文件（Files）等
 - Appenders 取值：
 - org.apache.log4j.ConsoleAppender（控制台）
 - org.apache.log4j.FileAppender（文件）
 - ConsoleAppender 常用参数
 - ImmediateFlush=true：表示所有消息都会被立即输出，设为 false 则不输出，默认值是 true
 - Target=System.out：默认值是 System.out
 - FileAppender 常用的选项
 - ImmediateFlush=true：表示所有消息都会被立即输出。设为 false 则不输出，默认值是 true
 - Append=false：true 表示将消息添加到指定文件中，原来的消息不覆盖。默认值是 true
 - File=/logs/logging.log4j：指定消息输出到 logging.log4j 文件中
- Layouts（布局）：日志输出的格式，常用的布局管理器：
 - org.apache.log4j.PatternLayout（可以灵活地指定布局模式）
 - org.apache.log4j.SimpleLayout（包含日志信息的级别和信息字符串）
 - org.apache.log4j.TTCCLayout（包含日志产生的时间、线程、类别等信息）

- PatternLayout 常用的选项

```
<font color="red" size="3">PatternLayout常用的选项</font>
```

ConversionPattern=%m%n: 设定以怎样的格式显示消息。

格式化符号说明：

```
%p: 输出日志信息的级别, 即DEBUG, INFO, WARN, ERROR, FATAL。  
%d: 输出日志时间点的日期或时间, 默认格式为ISO8601, 也可以在其后指定格式, 如: %d{yyyy/MM/dd  
HH:mm:ss,SSS}。  
%r: 输出自应用程序启动到输出该log信息耗费的毫秒数。  
%t: 输出产生该日志事件的线程名。  
%l: 输出日志事件的发生位置, 相当于%c.%M(%F:%L)的组合, 包括类全名、方法、文件名以及在代码中的行数。例如:  
test.TestLog4j.main(TestLog4j.java:10)。  
%c: 输出日志信息所属的类目, 通常就是所在类的全名。  
%M: 输出产生日志信息的方法名。  
%F: 输出日志消息产生时所在的文件名称。  
%L: 输出代码中的行号。  
%m: 输出代码中指定的具体日志信息。  
%n: 输出一个回车换行符, Windows平台为"rn", Unix平台为"n"。  
%x: 输出和当前线程相关联的NDC(嵌套诊断环境), 尤其用到像java servlets这样的多客户多线程的应用中。  
%%: 输出一个 "%" 字符。  
另外, 还可以在%与格式字符之间加上修饰符来控制其最小长度、最大长度、和文本的对齐方式。如:  
1) c: 指定输出category的名称, 最小的长度是20, 如果category的名称长度小于20的话, 默认的情况下右对齐。  
2)%-20c: "-"号表示左对齐。  
3)%.30c: 指定输出category的名称, 最大的长度是30, 如果category的名称长度大于30的话, 就会将左边多出的字符  
截掉, 但小于30的话也不会补空格。
```

日志应用

- log4j 的配置文件,名字为 log4j.properties, 放在 src 根目录下

```
log4j.rootLogger=I  
  
### direct log messages to my ###  
log4j.appender.my=org.apache.log4j.ConsoleAppender  
log4j.appender.my.ImmediateFlush = true  
log4j.appender.my.Target=System.out  
log4j.appender.my.layout=org.apache.log4j.PatternLayout  
log4j.appender.my.layout.ConversionPattern=%d %t %5p %c{1}:%L - %m%n  
  
# fileAppender演示  
log4j.appender.fileAppender=org.apache.log4j.FileAppender  
log4j.appender.fileAppender.ImmediateFlush = true  
log4j.appender.fileAppender.Append=true  
log4j.appender.fileAppender.File=E:/log4j-log.log  
log4j.appender.fileAppender.layout=org.apache.log4j.PatternLayout  
log4j.appender.fileAppender.layout.ConversionPattern=%d %5p %c{1}:%L - %m%n
```

- 测试类

```
// 测试类
```

```
public class Log4JTest01 {  
    //使用log4j的api来获取日志的对象  
    //弊端：如果以后我们更换日志的实现类，那么下面的代码就需要跟着改  
    //不推荐使用  
    //private static final Logger LOGGER =  
    //Logger.getLogger(Log4JTest01.class);  
    //使用slf4j里面的api来获取日志的对象  
    //好处：如果以后我们更换日志的实现类，那么下面的代码不需要跟着修改  
    //推荐使用  
    private static final Logger LOGGER =  
    LoggerFactory.getLogger(Log4JTest01.class);  
    public static void main(String[] args) {  
        //1.导入jar包  
        //2.编写配置文件  
        //3.在代码中获取日志的对象  
        //4.按照日志级别设置日志信息  
        LOGGER.debug("debug级别的日志");  
        LOGGER.info("info级别的日志");  
        LOGGER.warn("warn级别的日志");  
        LOGGER.error("error级别的日志");  
    }  
}
```

Netty

基本介绍

Netty 是一个异步事件驱动的网络应用程序框架，用于快速开发可维护、高性能的网络服务器和客户端

Netty 官网：<https://netty.io/>

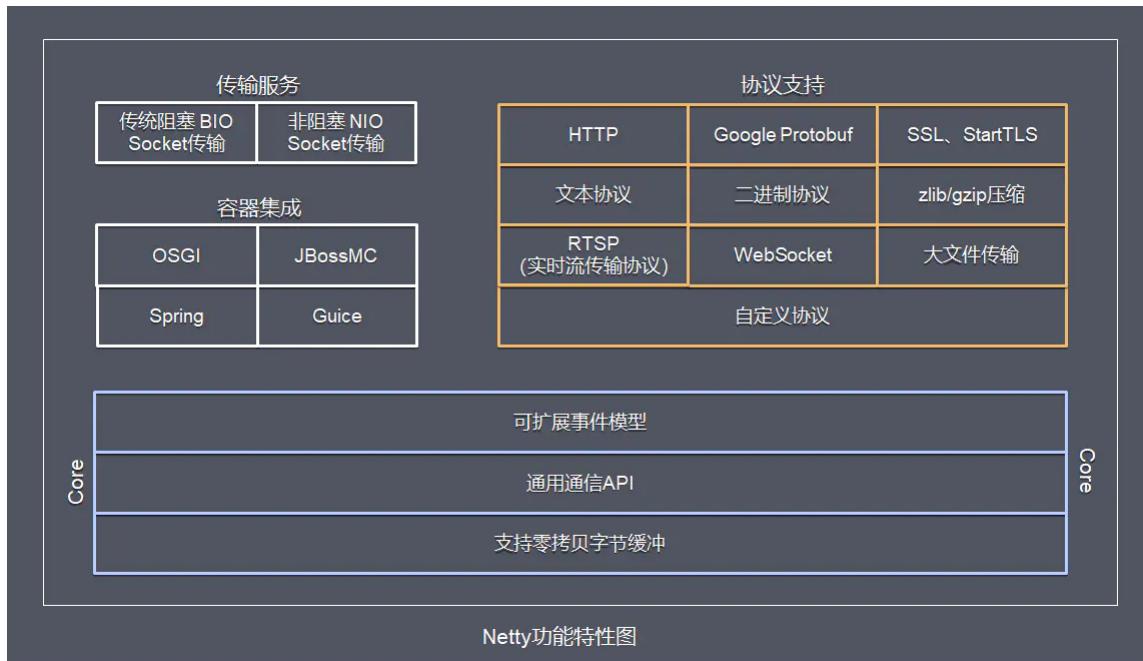
Netty 对 JDK 自带的 NIO 的 API 进行封装，解决上述问题，主要特点有：

- 设计优雅，适用于各种传输类型的统一 API，阻塞和非阻塞 Socket 基于灵活且可扩展的事件模型
- 使用方便，详细记录的 Javadoc、用户指南和示例，没有其他依赖项
- 高性能，吞吐量更高，延迟更低，减少资源消耗，最小化不必要的内存复制
- 安全，完整的 SSL/TLS 和 StartTLS 支持

Netty 的功能特性：

- 传输服务：支持 BIO 和 NIO
- 容器集成：支持 OSGI、JBossMC、Spring、Guice 容器
- 协议支持：HTTP、Protobuf、二进制、文本、WebSocket 等一系列协议都支持，也支持通过实行编码解码逻辑来实现自定义协议

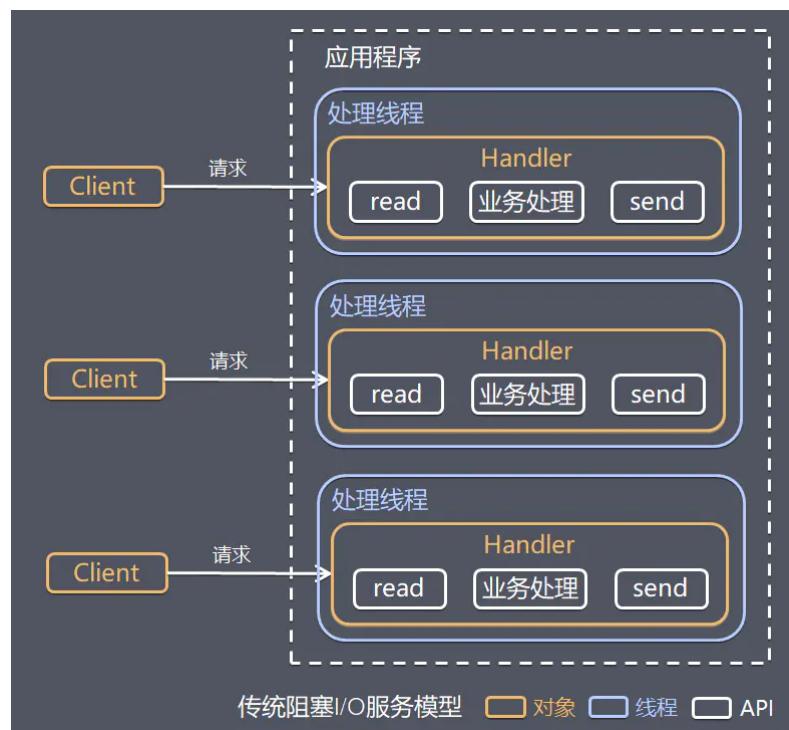
- Core 核心：可扩展事件模型、通用通信 API、支持零拷贝的 ByteBuf 缓冲对象



线程模型

阻塞模型

传统阻塞型 I/O 模式，每个连接都需要独立的线程完成数据的输入，业务处理，数据返回



模型缺点：

- 当并发数较大时，需要创建大量线程来处理连接，系统资源占用较大
- 连接建立后，如果当前线程暂时没有数据可读，则线程就阻塞在 read 操作上，造成线程资源浪费

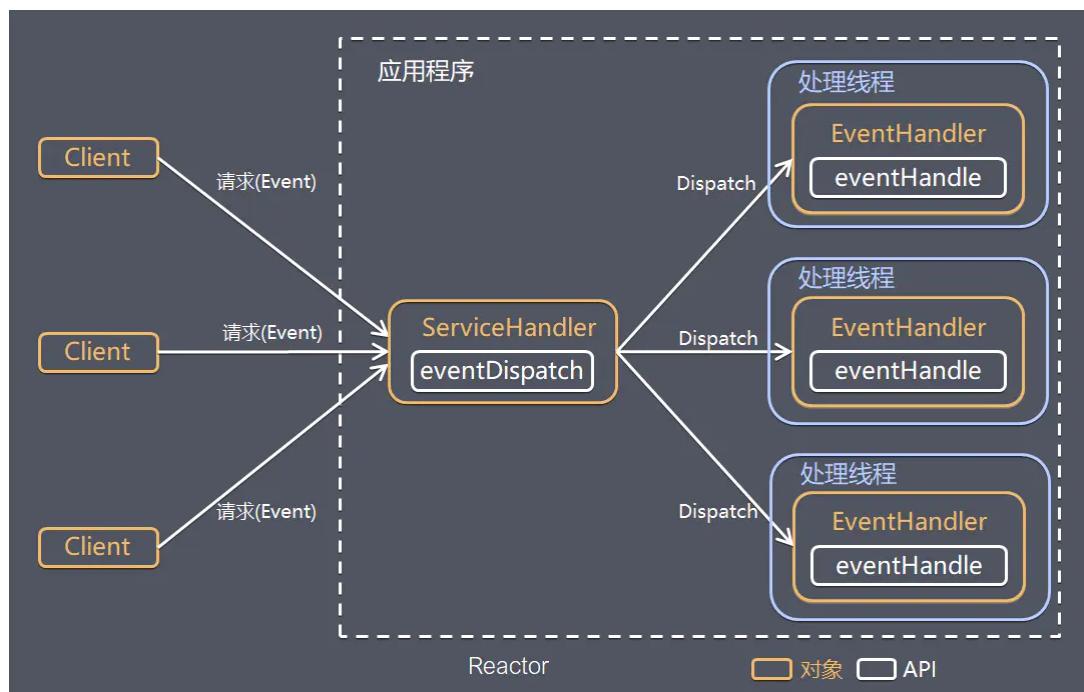
参考文章：<https://www.jianshu.com/p/2965fca6bb8f>

Reactor

设计思想

Reactor 模式，通过一个或多个输入同时传递给服务处理器的**事件驱动处理模式**。服务端程序处理传入的多路请求，并将它们同步分派给对应的处理线程，Reactor 模式也叫 Dispatcher 模式，即 I/O 多路复用统一监听事件，收到事件后分发（Dispatch 给某线程）

I/O 复用结合线程池，就是 Reactor 模式基本设计思想：



Reactor 模式关键组成：

- Reactor：在一个单独的线程中运行，负责**监听和分发事件**，分发给适当的处理程序来对 I/O 事件做出反应
- Handler：处理程序执行 I/O 要完成的实际事件，Reactor 通过调度适当的处理程序来响应 I/O 事件，处理程序执行**非阻塞操作**

Reactor 模式具有如下的优点：

- 响应快，不必为单个同步时间所阻塞，虽然 Reactor 本身依然是同步的
- 编程相对简单，可以最大程度的避免复杂的多线程及同步问题，并且避免了多线程/进程的切换开销
- 可扩展性，可以方便的通过增加 Reactor 实例个数来充分利用 CPU 资源
- 可复用性，Reactor 模型本身与具体事件处理逻辑无关，具有很高的复用性

根据 Reactor 的数量和处理资源池线程的数量不同，有三种典型的实现：

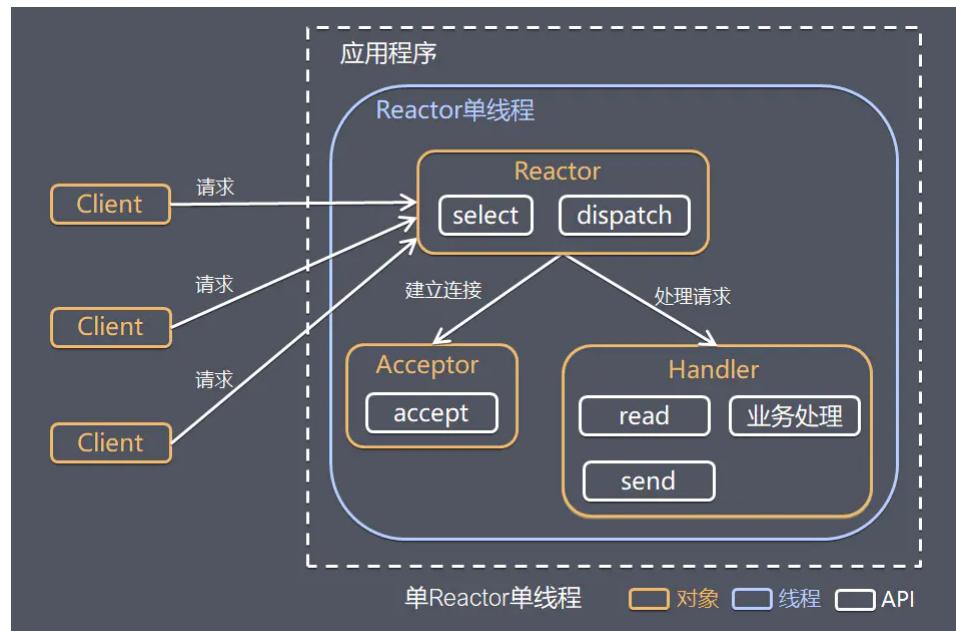
- 单 Reactor 单线程
 - 单 Reactor 多线程
 - 主从 Reactor 多线程
-

单R单线程

Reactor 对象通过 select 监控客户端请求事件，收到事件后通过 dispatch 进行分发：

- 如果是建立连接请求事件，则由 Acceptor 通过 accept 处理连接请求，然后创建一个 Handler 对象处理连接完成后的后续业务处理
- 如果不是建立连接事件，则 Reactor 会分发给连接对应的 Handler 来响应，Handler 会完成 read、业务处理、send 的完整流程

说明：Handler 和 Acceptor 属于同一个线程



模型优点：模型简单，没有多线程、进程通信、竞争的问题，全部都在一个线程中完成

模型缺点：

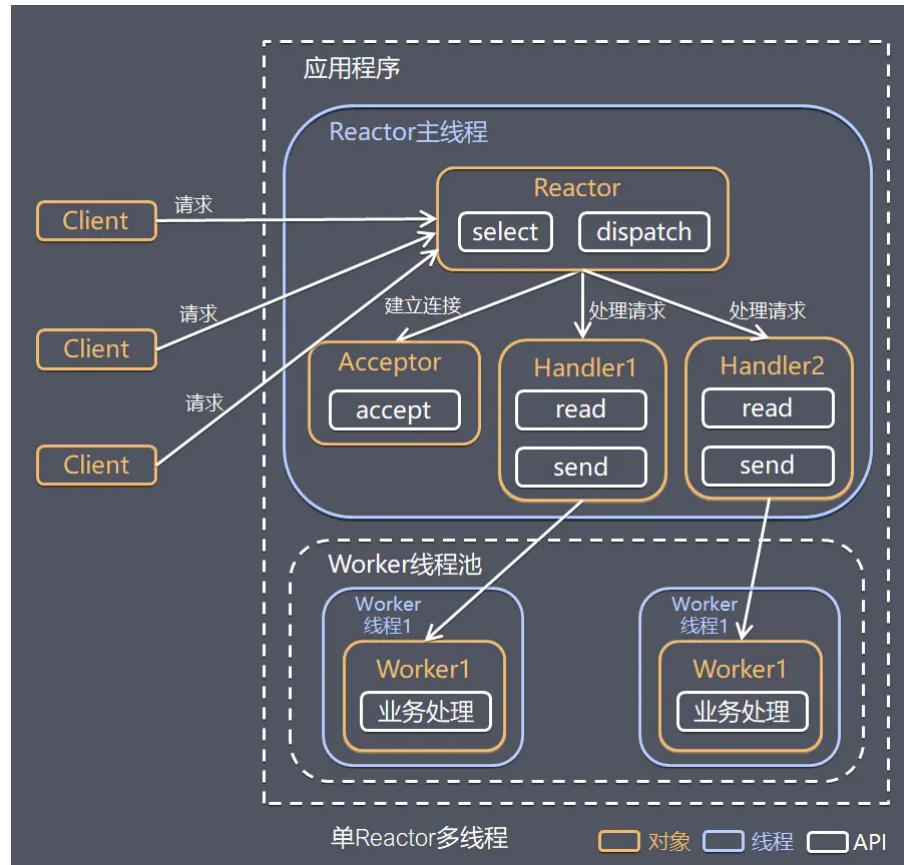
- 性能问题：只有一个线程，无法发挥多核 CPU 的性能，Handler 在处理某个连接上的业务时，整个进程无法处理其他连接事件，很容易导致性能瓶颈
- 可靠性问题：线程意外跑飞，或者进入死循环，会导致整个系统通信模块不可用，不能接收和处理外部消息，造成节点故障

使用场景：客户端的数量有限，业务处理非常快速，比如 Redis，业务处理的时间复杂度 O(1)

单R多线程

执行流程同单 Reactor 单线程，不同的是：

- Handler 只负责响应事件，不做具体业务处理，通过 read 读取数据后，会分发给后面的 Worker 线程池进行业务处理
- Worker 线程池会分配独立的线程完成真正的业务处理，将响应结果发给 Handler 进行处理，最后由 Handler 收到响应结果后通过 send 将响应结果返回给 Client



模型优点：可以充分利用多核 CPU 的处理能力

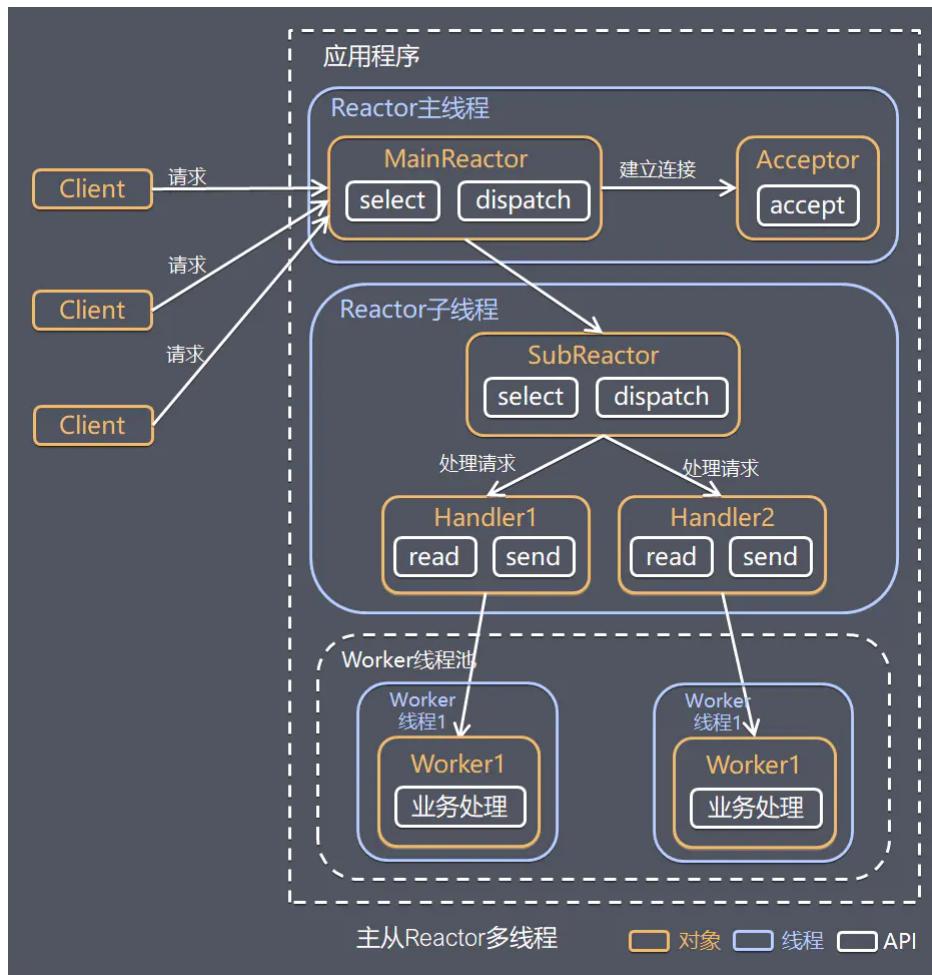
模型缺点：

- 多线程数据共享和访问比较复杂
- Reactor 承担所有事件的监听和响应，在单线程中运行，高并发场景下容易成为性能瓶颈

主从模型

采用多个 Reactor，执行流程：

- Reactor 主线程 MainReactor 通过 select 监控建立连接事件，收到事件后通过 Acceptor 接收，处理建立连接事件，处理完成后 MainReactor 会将连接分配给 Reactor 子线程的 SubReactor (有多个) 处理
- SubReactor 将连接加入连接队列进行监听其他事件，并创建一个 Handler 用于处理该连接的事件，当有新的事件发生时，SubReactor 会调用连接对应的 Handler 进行响应
- Handler 通过 read 读取数据后，会分发给 Worker 线程池进行业务处理
- Worker 线程池会分配独立的线程完成真正的业务处理，将响应结果发给 Handler 进行处理，最后由 Handler 收到响应结果后通过 send 将响应结果返回给 Client



模型优点

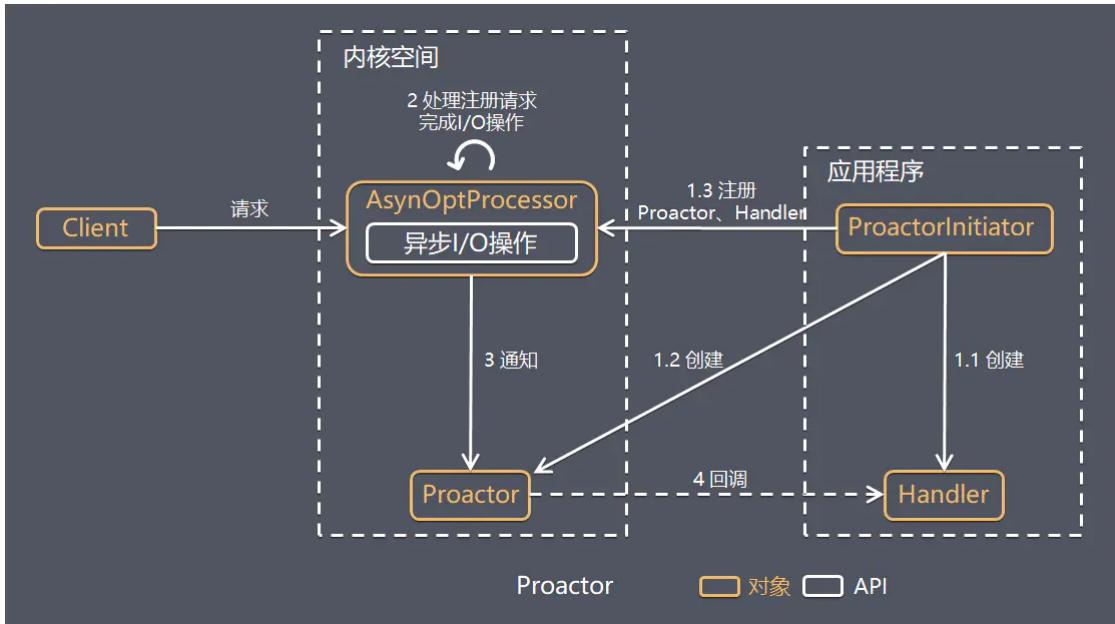
- 父线程与子线程的数据交互简单职责明确，父线程只需要接收新连接，子线程完成后续的业务处理
- 父线程与子线程的数据交互简单，Reactor 主线程只需要把新连接传给子线程，子线程无需返回数据

使用场景：Nginx 主从 Reactor 多进程模型，Memcached 主从多线程，Netty 主从多线程模型的支持

Proactor

Reactor 模式中，Reactor 等待某个事件的操作状态发生变化（文件描述符可读写，socket 可读写），然后把事件传递给事先注册的 Handler 来做实际的读写操作，其中的读写操作都需要应用程序同步操作，所以 **Reactor 是非阻塞同步网络模型（NIO）**

把 I/O 操作改为异步，交给操作系统来完成就能进一步提升性能，这就是异步网络模型 Proactor (AIO)：



工作流程：

- ProactorInitiator 创建 Proactor 和 Handler 对象，并将 Proactor 和 Handler 通过 Asynchronous Operation Processor (AsyOptProcessor) 注册到内核
- AsyOptProcessor 处理注册请求，并处理 I/O 操作，完成I/O后通知 Proactor
- Proactor 根据不同的事件类型回调不同的 Handler 进行业务处理，最后由 Handler 完成业务处理

对比：Reactor 在事件发生时就通知事先注册的处理器（读写在应用程序线程中处理完成）；Proactor 是在事件发生时基于异步 I/O 完成读写操作（内核完成），I/O 完成后才回调应用程序的处理器进行业务处理

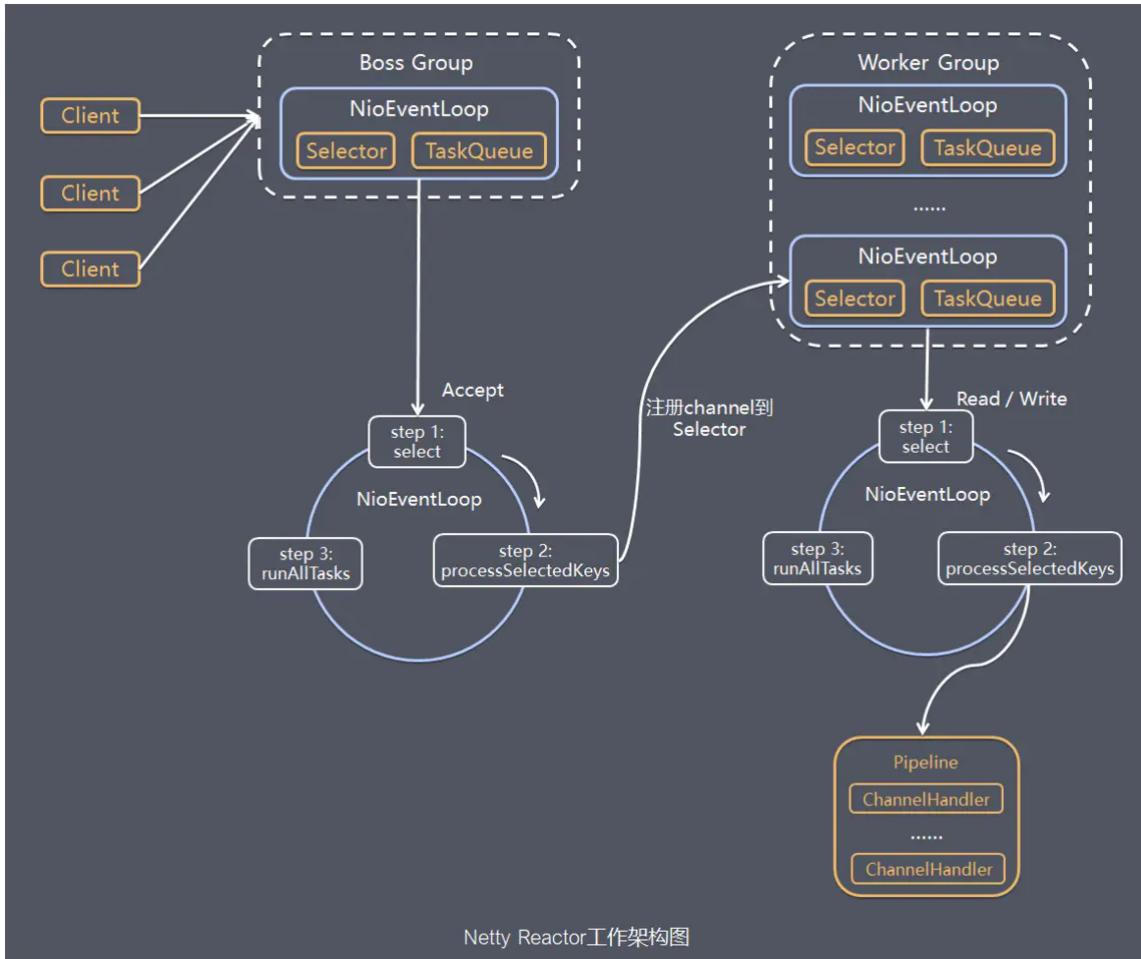
模式优点：异步 I/O 更加充分发挥 DMA (Direct Memory Access 直接内存存取) 的优势

模式缺点：

- 编程复杂性，由于异步操作流程的事件的初始化和事件完成在时间和空间上都是相互分离的，因此开发异步应用程序更加复杂，应用程序还可能因为反向的流控而变得更加难以调试
- 内存使用，缓冲区在读或写操作的时间段内必须保持住，可能造成持续的不确定性，并且每个并发操作都要求有独立的缓存，Reactor 模式在 socket 准备好读或写之前是不要求开辟缓存的
- 操作系统支持，Windows 下通过 IOCP 实现了真正的异步 I/O，而在 Linux 系统下，Linux2.6 才引入异步 I/O，目前还不完善，所以在 Linux 下实现高并发网络编程都是以 Reactor 模型为主

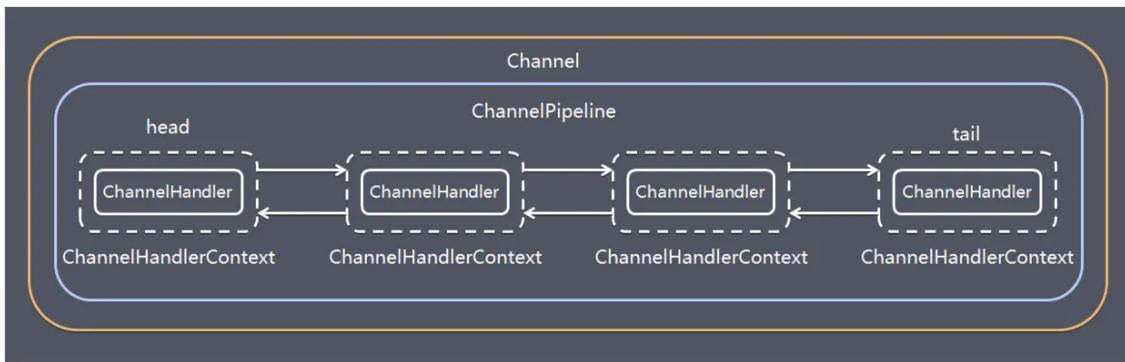
Netty

Netty 主要基于主从 Reactors 多线程模型做了一定的改进，Netty 的工作架构图：



工作流程：

1. Netty 抽象出两组线程池 BossGroup 专门负责接收客户端的连接，WorkerGroup 专门负责网络的读写
2. BossGroup 和 WorkerGroup 类型都是 NioEventLoopGroup，该 Group 相当于一个事件循环组，含有多个事件循环，每一个事件循环是 NioEventLoop，所以可以有多个线程
3. NioEventLoop 表示一个**循环处理任务的线程**，每个 NioEventLoop 都有一个 Selector，用于监听绑定在其上的 Socket 的通讯
4. 每个 Boss NioEventLoop 循环执行的步骤：
 - 轮询 accept 事件
 - 处理 accept 事件，与 client 建立连接，生成 NioSocketChannel，并将其**注册到某个 Worker 中的某个 NioEventLoop 上的 Selector**，连接就与 NioEventLoop 绑定
 - 处理任务队列的任务，即 runAllTasks
5. 每个 Worker NioEventLoop 循环执行的步骤：
 - 轮询 read、write 事件
 - 处理 I/O 事件，即 read、write 事件，在对应 NioSocketChannel 处理
 - 处理任务队列的任务，即 runAllTasks
6. 每个 Worker NioEventLoop 处理业务时，会使用 Pipeline（管道），Pipeline 中包含了 Channel，即通过 Pipeline 可以获取到对应通道，管道中维护了很多的处理器 Handler



基本实现

开发简单的服务器端和客户端，基本介绍：

- Channel 理解为数据的通道，把 msg 理解为流动的数据，最开始输入是 ByteBuf，但经过 Pipeline 的加工，会变成其它类型对象，最后输出又变成 ByteBuf
- Handler 理解为数据的处理工序，Pipeline 负责发布事件传播给每个 Handler，Handler 对自己感兴趣的事件进行处理（重写了相应事件处理方法），分 Inbound 和 Outbound 两类
- EventLoop 理解为处理数据的执行者，既可以执行 IO 操作，也可以进行任务处理。每个执行者有任务队列，队列里可以堆放多个 Channel 的待处理任务，任务分为普通任务、定时任务。按照 Pipeline 顺序，依次按照 Handler 的规划（代码）处理数据

代码实现：

- pom.xml

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-all</artifactId>
    <version>4.1.20.Final</version>
</dependency>
```

- Server.java

```
public class HelloServer {
    public static void main(String[] args) {
        EventLoopGroup boss = new NioEventLoopGroup();
        EventLoopGroup worker = new NioEventLoopGroup(2);
        // 1. 启动器，负责组装 netty 组件，启动服务器
        new ServerBootstrap()
            // 2. 线程组，boss 只负责【处理 accept 事件】， worker 只【负责
            channel 上的读写】
            .group(boss, worker)
            .option()          // 给 ServerSocketChannel 配置参数
            .childOption()     // 给 SocketChannel 配置参数
            // 3. 选择服务器的 ServerSocketChannel 实现
    }
}
```

```

        .channel(NioServerSocketChannel.class)
        // 4. boss 负责处理连接, worker(child) 负责处理读写, 决定了能执行哪些操作(handler)
        .childHandler(new ChannelInitializer<NioSocketChannel>() {
            // 5. channel 代表和客户端进行数据读写的通道 Initializer 初始化, 负责添加别的 handler
            // 7. 连接建立后, 执行初始化方法
            @Override
            protected void initChannel(NioSocketChannel ch) throws
Exception {
                // 添加具体的 handler
                ch.pipeline().addLast(new StringDecoder());// 将
ByteBuf 转成字符串
                ch.pipeline().addLast(new
ChannelInboundHandlerAdapter() { // 自定义 handler
                    // 读事件
                    @Override
                    public void channelRead(ChannelHandlerContext
ctx, Object msg) {
                        // 打印转换好的字符串
                        System.out.println(msg);
                    }
                });
            }
        })
        // 6. 绑定监听端口
        .bind(8080);
    }
}

```

- Client.java

```

public class HelloClient {
    public static void main(String[] args) throws InterruptedException {
        // 1. 创建启动器类
        new Bootstrap()
            // 2. 添加 EventLoop
            .group(new NioEventLoopGroup())
            // .option(), 给 SocketChannel 配置参数
            // 3. 选择客户端 channel 实现
            .channel(NioSocketChannel.class)
            // 4. 添加处理器
            .handler(new ChannelInitializer<NioSocketChannel>() {
                // 4.1 连接建立后被调用
                @Override
                protected void initChannel(NioSocketChannel ch) throws
Exception {
                    // 将 Hello World 转为 ByteBuf
                    ch.pipeline().addLast(new StringEncoder());
                }
            })
            // 5. 连接到服务器, 然后调用 4.1
            .connect(new InetSocketAddress("127.0.0.1", 8080))
            // 6. 阻塞方法, 直到连接建立
            .sync();
    }
}

```

```
// 7. 代表连接对象
.channel()
// 8. 向服务器发送数据
.writeAndFlush("Hello World");
}
```

参考视频: <https://www.bilibili.com/video/BV1py4y1E7oA>

组件介绍

EventLoop

基本介绍

事件循环对象 EventLoop，**本质是一个单线程执行器同时维护了一个 Selector**，有 run 方法处理 Channel 上源源不断的 IO 事件

事件循环组 EventLoopGroup 是一组 EventLoop，Channel 会调用 Boss EventLoopGroup 的 register 方法来绑定其中一个 Worker 的 EventLoop，后续这个 Channel 上的 IO 事件都由此 EventLoop 来处理，保证了事件处理时的线程安全

EventLoopGroup 类 API:

- `EventLoop next()`: 获取集合中下一个 EventLoop, EventLoopGroup 实现了 Iterable 接口提供遍历 EventLoop 的能力
- `Future<?> shutdownGracefully()`: 优雅关闭的方法，会首先切换 EventLoopGroup 到关闭状态从而拒绝新的任务的加入，然后在任务队列的任务都处理完成后，停止线程的运行，从而确保整体应用是在正常有序的状态下退出的
- `<T> Future<T> submit(Callable<T> task)`: 提交任务
- `ScheduledFuture<?> scheduleWithFixedDelay`: 提交定时任务

任务传递

把要调用的代码封装为一个任务对象，由下一个 handler 的线程来调用

```
public class EventLoopServer {
    public static void main(String[] args) {
        EventLoopGroup group = new DefaultEventLoopGroup();
        new ServerBootstrap()
            .group(new NioEventLoopGroup(), new NioEventLoopGroup(2))
            .channel(NioServerSocketChannel.class)
    }
}
```

```

        .childHandler(new ChannelInitializer<NioSocketChannel>() {
            @Override
            protected void initChannel(NioSocketChannel ch) {
                ch.pipeline().addLast("handler1", new
        ChannelInboundHandlerAdapter() {
                    @Override
                    public void channelRead(ChannelHandlerContext ctx,
Object msg) {
                        ByteBuf buf = (ByteBuf) msg;
                        log.debug(buf.toString(Charset.defaultCharset()));
                        ctx.fireChannelRead(msg); // 让消息【传递】给下一个 handler
                    }
                }).addLast(group, "handler2", new
        ChannelInboundHandlerAdapter() {
                    @Override
                    public void channelRead(ChannelHandlerContext ctx,
Object msg) {
                        ByteBuf buf = (ByteBuf) msg;
                        log.debug(buf.toString(Charset.defaultCharset()));
                    }
                });
            }
        })
        .bind(8080);
    }
}

```

源码分析：

```

public ChannelHandlerContext fireChannelRead(final Object msg) {
    invokeChannelRead(findContextInbound(MASK_CHANNEL_READ), msg);
    return this;
}
static void invokeChannelRead(final AbstractChannelHandlerContext next, Object msg) {
    final Object m = next.pipeline.touch(ObjectUtil.checkNotNull(msg, "msg"),
next);
    EventExecutor executor = next.executor();
    // 下一个 handler 的事件循环是否与当前的事件循环是同一个线程
    if (executor.inEventLoop()) {
        // 是，直接调用
        next.invokeChannelRead(m);
    } else {
        // 不是，将要执行的代码作为任务提交给下一个 handler 处理
        executor.execute(new Runnable() {
            @Override
            public void run() {
                next.invokeChannelRead(m);
            }
        });
    }
}

```

Channel

连接操作

Channel 类 API:

- `ChannelFuture close()`: 关闭通道
- `ChannelPipeline pipeline()`: 添加处理器
- `ChannelFuture write(Object msg)`: 数据写入缓冲区
- `ChannelFuture writeAndFlush(Object msg)`: 数据写入缓冲区并且刷出

ChannelFuture 类 API:

- `ChannelFuture sync()`: 同步阻塞等待连接成功
- `ChannelFuture addListener(GenericFutureListener<? super ChannelFuture> listener)`: 异步等待

代码实现:

- `connect` 方法是异步的，不等连接建立完成就返回，因此 `channelFuture` 对象中不能立刻获得到正确的 `Channel` 对象，需要等待
- 连接未建立 `channel` 打印为 `[id: 0x2e1884dd]`；建立成功打印为 `[id: 0x2e1884dd, L:/127.0.0.1:57191 - R:/127.0.0.1:8080]`

```
public class ChannelClient {
    public static void main(String[] args) throws InterruptedException {
        ChannelFuture channelFuture = new Bootstrap()
            .group(new NioEventLoopGroup())
            .channel(NioSocketChannel.class)
            .handler(new ChannelInitializer<NioSocketChannel>() {
                @Override
                protected void initChannel(NioSocketChannel ch) throws
Exception {
                    ch.pipeline().addLast(new StringEncoder());
                }
            })
            // 1. 连接服务器，【异步非阻塞】，main 调用 connect 方法，真正执行连接的是
nio 线程
            .connect(new InetSocketAddress("127.0.0.1", 8080));
        // 2.1 使用 sync 方法【同步】处理结果，阻塞当前线程，直到 nio 线程连接建立完毕
        channelFuture.sync();
        Channel channel = channelFuture.channel();
        System.out.println(channel); // 【打印】
        // 向服务器发送数据
        channel.writeAndFlush("hello world");
    }
}
```

```

@Override
// nio 线程连接建立好以后，回调该方法
public void operationComplete(ChannelFuture future) throws Exception
{
    if (future.isSuccess()) {
        Channel channel = future.channel();
        channel.writeAndFlush("hello, world");
    } else {
        // 建立失败，需要关闭
        future.channel().close();
    }
}
);
}
}

```

关闭操作

关闭 EventLoopGroup 的运行，分为同步关闭和异步关闭

```

public class CloseFutureClient {
    public static void main(String[] args) throws InterruptedException {
        NioEventLoopGroup group = new NioEventLoopGroup();
        ChannelFuture channelFuture = new Bootstrap()
            // ....
            .connect(new InetSocketAddress("127.0.0.1", 8080));
        Channel channel = channelFuture.sync().channel();
        // 发送数据
        new Thread(() -> {
            Scanner sc = new Scanner(System.in);
            while (true) {
                String line = sc.nextLine();
                if (line.equals("q")) {
                    channel.close();
                    break;
                }
                channel.writeAndFlush(line);
            }
        }, "input").start();
        // 获取 closeFuture 对象
        ChannelFuture closeFuture = channel.closeFuture();

        // 1. 同步处理关闭
        System.out.println("waiting close...");
        closeFuture.sync();
        System.out.println("处理关闭后的操作");
        ****
        // 2. 异步处理关闭
        closeFuture.addListener(new ChannelFutureListener() {
            @Override

```

```
    public void operationComplete(ChannelFuture future) throws Exception {
        {
            System.out.println("处理关闭后的操作");
            group.shutdownGracefully();
        }
    );
}
}
```

Future

基本介绍

Netty 中的 Future 与 JDK 中的 Future 同名，但是功能的实现不同

```
package io.netty.util.concurrent;
public interface Future<V> extends java.util.concurrent.Future<V>
```

Future 类 API:

- `V get()`: 阻塞等待获取任务执行结果
- `V getNow()`: 非阻塞获取任务结果，还未产生结果时返回 null
- `Throwable cause()`: 非阻塞获取失败信息，如果没有失败，返回 null
- `Future<V> sync()`: 等待任务结束，如果任务失败，抛出异常
- `boolean cancel(boolean mayInterruptIfRunning)`: 取消任务
- `Future<V> addListener(GenericFutureListener<V> listener)`: 添加回调，异步接收结果
- `boolean isSuccess()`: 判断任务是否成功
- `boolean isCancelled()`: 判断任务是否取消

```
public class NettyFutureDemo {
    public static void main(String[] args) throws Exception {
        NioEventLoopGroup group = new NioEventLoopGroup();
        EventLoop eventLoop = group.next();
        Future<Integer> future = eventLoop.submit(new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
                System.out.println("执行计算");
                Thread.sleep(1000);
                return 70;
            }
        });
        future.getNow();
        System.out.println(new Date() + "等待结果");
        System.out.println(new Date() + "" + future.get());
    }
}
```

扩展子类

Promise 类是 Future 的子类，可以脱离任务独立存在，作为两个线程间传递结果的容器

```
public interface Promise<V> extends Future<V>
```

Promise 类 API：

- `Promise<V> setSuccess(V result)`：设置成功结果
- `Promise<V> setFailure(Throwable cause)`：设置失败结果

```
public class NettyPromiseDemo {  
    public static void main(String[] args) throws Exception {  
        // 1. 准备 EventLoop 对象  
        EventLoop eventLoop = new NioEventLoopGroup().next();  
        // 2. 主动创建 promise  
        DefaultPromise<Integer> promise = new DefaultPromise<>(eventLoop);  
        // 3. 任意一个线程执行计算，计算完毕后向 promise 填充结果  
        new Thread(() -> {  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            promise.setSuccess(200);  
        }).start();  
  
        // 4. 接受结果的线程  
        System.out.println(new Date() + "等待结果");  
        System.out.println(new Date() + "" + promise.get());  
    }  
}
```

Pipeline

ChannelHandler 用来处理 Channel 上的各种事件，分为入站出站两种，所有 ChannelHandler 连接成双向链表就是 Pipeline

- 入站处理器通常是 ChannelInboundHandlerAdapter 的子类，主要用来读取客户端数据，写回结果
- 出站处理器通常是 ChannelOutboundHandlerAdapter 的子类，主要对写回结果进行加工（入站和出站是对于服务端来说的）

```
public static void main(String[] args) {  
    new ServerBootstrap()  
        .group(new NioEventLoopGroup())
```

```

.channel(NioServerSocketChannel.class)
.childHandler(new ChannelInitializer<NiosocketChannel>() {
    @Override
    protected void initChannel(NiosocketChannel ch) throws Exception {
        // 1. 通过 channel 拿到 pipeline
        ChannelPipeline pipeline = ch.pipeline();
        // 2. 添加处理器 head -> h1 -> h2 -> h3 -> h4 -> tail
        pipeline.addLast("h1", new ChannelInboundHandlerAdapter() {
            @Override
            public void channelRead(ChannelHandlerContext ctx, Object
msg) throws Exception {
                log.debug("1");
                ByteBuf buf = (ByteBuf) msg;
                String s = buf.toString(Charset.defaultCharset());
                // 将数据传递给下一个【入站】handler, 如果不调用该方法则链会断开
                super.channelRead(ctx, s);
            }
        });
        pipeline.addLast("h2", new ChannelInboundHandlerAdapter() {
            @Override
            public void channelRead(ChannelHandlerContext ctx, Object
msg) throws Exception {
                log.debug("2");
                // 从【尾部开始向前触发】出站处理器
            }
        });
        ch.writeAndFlush(ctx.alloc().buffer().writeBytes("server".getBytes()));
        // 该方法会让管道从【当前 handler 向前】寻找出站处理器
        // ctx.writeAndFlush();
    }
});
pipeline.addLast("h3", new ChannelOutboundHandlerAdapter() {
    @Override
    public void write(ChannelHandlerContext ctx, Object msg,
ChannelPromise promise) throws Exception {
        log.debug("3");
        super.write(ctx, msg, promise);
    }
});
pipeline.addLast("h4", new ChannelOutboundHandlerAdapter() {
    @Override
    public void write(ChannelHandlerContext ctx, Object msg,
ChannelPromise promise) throws Exception {
        log.debug("4");
        super.write(ctx, msg, promise);
    }
});
}
.bind(8080);
}

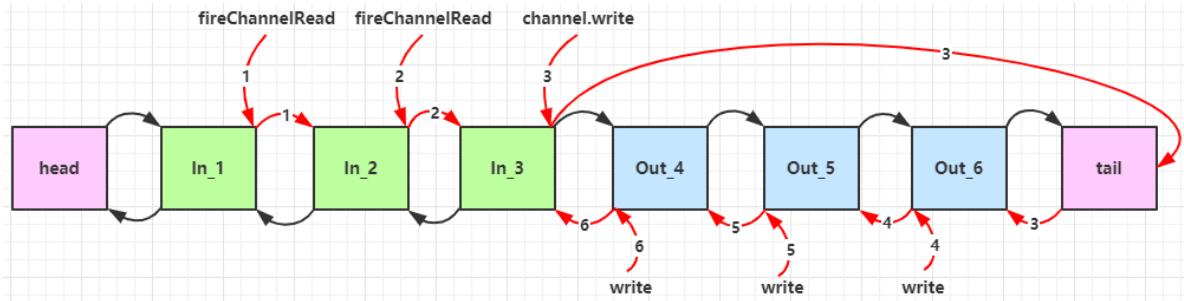
```

服务器端依次打印：1 2 4 3，所以入站是按照 addLast 的顺序执行的，出站是按照 addLast 的逆序执行

一个 Channel 包含了一个 ChannelPipeline，而 ChannelPipeline 中又维护了一个由 ChannelHandlerContext 组成的双向链表，并且每个 ChannelHandlerContext 中关联着一个 ChannelHandler

入站事件和出站事件在一个双向链表中，两种类型的 handler 互不干扰：

- 入站事件会从链表 head 往后传递到最后一个入站的 handler
- 出站事件会从链表 tail 往前传递到最前一个出站的 handler



ByteBuf

基本介绍

ByteBuf 是对字节数据的封装，优点：

- 池化，可以重用池中 ByteBuf 实例，更节约内存，减少内存溢出的可能
- 读写指针分离，不需要像 ByteBuffer 一样切换读写模式
- 可以自动扩容
- 支持链式调用，使用更流畅
- 零拷贝思想，例如 slice、duplicate、CompositeByteBuf

创建方法

创建方式

- `ByteBuf buffer = ByteBufAllocator.DEFAULT.buffer(10)`：创建了一个默认的 ByteBuf，初始容量是 10

```
public ByteBuf buffer() {  
    if (directByDefault) {  
        return directBuffer();  
    }  
    return heapBuffer();  
}
```

- `ByteBuf buffer = ByteBufAllocator.DEFAULT.heapBuffer(10)`：创建池化基于堆的 ByteBuf

- `ByteBuf buffer = ByteBufAllocator.DEFAULT.directBuffer(10)` : 创建池化基于直接内存的 ByteBuf
- 推荐的创建方式：在添加处理器的方法中

```
pipeline.addLast(new ChannelInboundHandlerAdapter() {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        ByteBuf buffer = ctx.alloc().buffer();
    }
});
```

直接内存对比堆内存：

- 直接内存创建和销毁的代价昂贵，但读写性能高（少一次内存复制），适合配合池化功能一起用
- 直接内存对 GC 压力小，因为这部分内存不受 JVM 垃圾回收的管理，但也要注意及时主动释放

池化的意义在于可以重用 **ByteBuf**，高并发时池化功能更节约内存，减少内存溢出的可能，与非池化对比：

- 非池化，每次都要创建新的 ByteBuf 实例，这个操作对直接内存代价昂贵，堆内存会增加 GC 压力
- 池化，可以重用池中 ByteBuf 实例，并且采用了与 jemalloc 类似的内存分配算法提升分配效率

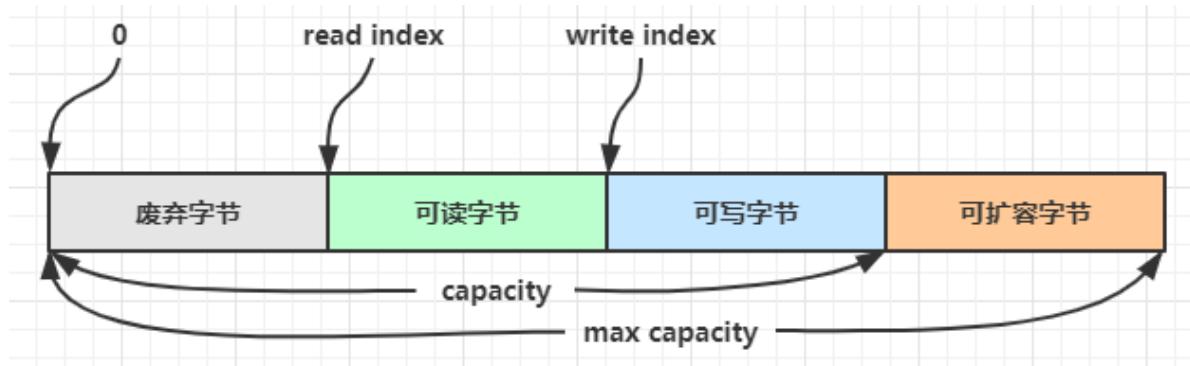
池化功能的开启，可以通过下面的系统环境变量来设置：

```
-Dio.nettyallocator.type={unpooled|pooled} # VM 参数
```

- 4.1 以后，非 Android 平台默认启用池化实现，Android 平台启用非池化实现
- 4.1 之前，池化功能还不成熟，默认是非池化实现

读写操作

ByteBuf 由四部分组成，最开始读写指针（双指针）都在 0 位置



写入方法：

方法名	说明	备注
writeBoolean(boolean value)	写入 boolean 值	用一字节 01 00 代表 true false
writeByte(int value)	写入 byte 值	
writelnt(int value)	写入 int 值	Big Endian, 即 0x250, 写入后 00 00 02 50
writeIntLE(int value)	写入 int 值	Little Endian, 即 0x250, 写入后 50 02 00 00
writeBytes(ByteBuf src)	写入 ByteBuf	
writeBytes(byte[] src)	写入 byte[]	
writeBytes(ByteBuffer src)	写入 NIO 的 ByteBuffer	
int writeCharSequence(CharSequence s, Charset c)	写入字符串	

- 这些方法的未指明返回值的，其返回值都是 ByteBuf，意味着可以链式调用
- 写入几位写指针后移几位，指向可以写入的位置
- 网络传输，默认习惯是 Big Endian

扩容：写入数据时，容量不够了（初始容量是 10），这时会引发扩容

- 如果写入后数据大小未超过 512，则选择下一个 16 的整数倍，例如写入后大小为 12，则扩容后 capacity 是 16
- 如果写入后数据大小超过 512，则选择下一个 2^n ，例如写入后大小为 513，则扩容后 capacity 是 $2^{10} = 1024$ ($2^9=512$ 不够)
- 扩容不能超过 max capacity 会报错

读取方法：

- `byte readByte()`：读取一个字节，读指针后移
- `byte getByte(int index)`：读取指定索引位置的字节，读指针不动
- `ByteBuf markReaderIndex()`：标记读数据的位置
- `ByteBuf resetReaderIndex()`：重置到标记位置，可以重复读取标记位置向后的数据

内存释放

Netty 中三种内存的回收：

- UnpooledHeapByteBuf 使用的是 JVM 内存，只需等 GC 回收内存
- UnpooledDirectByteBuf 使用的就是直接内存了，需要特殊的方法来回收内存
- PooledByteBuf 和子类使用了池化机制，需要更复杂的规则来回收内存

Netty 采用了引用计数法来控制回收内存，每个 ByteBuf 都实现了 ReferenceCounted 接口，回收的规则：

- 每个 ByteBuf 对象的初始计数为 1
- 调用 release 方法计数减 1，如果计数为 0，ByteBuf 内存被回收
- 调用 retain 方法计数加 1，表示调用者没用完之前，其它 handler 即使调用了 release 也不会造成回收
- 当计数为 0 时，底层内存会被回收，这时即使 ByteBuf 对象还在，其各个方法均无法正常使用

```
ByteBuf buf = .ByteBufAllocator.DEFAULT.buffer(10)
try {
    // 逻辑处理
} finally {
    buf.release();
}
```

Pipeline 的存在，需要将 ByteBuf 传递给下一个 ChannelHandler，如果在 finally 中 release 了，就失去了传递性，处理规则：

- 创建 ByteBuf 放入 Pipeline
- 入站 ByteBuf 处理原则
 - 对原始 ByteBuf 不做处理，调用 ctx.fireChannelRead(msg) 向后传递，这时无须 release，反之不传递需要
 - 将原始 ByteBuf 转换为其它类型的 Java 对象，这时 ByteBuf 就没用了，此时必须 release
 - 如果出现异常，ByteBuf 没有成功传递到下一个 ChannelHandler，必须 release
 - 假设消息一直向后传，那么 TailContext 会负责释放未处理消息（原始的 ByteBuf）

```
// io.netty.channel.DefaultChannelPipeline#onUnhandledInboundMessage(java.lang.Object)
protected void onUnhandledInboundMessage(Object msg) {
    try {
        logger.debug();
    } finally {
        ReferenceCountUtil.release(msg);
    }
}
// io.netty.util.ReferenceCountUtil#release(java.lang.Object)
public static boolean release(Object msg) {
    if (msg instanceof ReferenceCounted) {
        return ((ReferenceCounted) msg).release();
    }
    return false;
}
```

- 出站 ByteBuf 处理原则
 - 出站消息最终都会转为 ByteBuf 输出，一直向前传，由 HeadContext flush 后 release
 - 不确定 ByteBuf 被引用了多少次，但又必须彻底释放，可以循环调用 release 直到返回 true
-

拷贝操作

零拷贝方法：

- `ByteBuf slice(int index, int length)`：对原始 ByteBuf 进行切片成多个 ByteBuf，切片后的 ByteBuf 并没有发生内存复制，**共用原始 ByteBuf 的内存**，切片后的 ByteBuf 维护独立的 read, write 指针

```
public static void main(String[] args) {
    ByteBuf buf = ByteBufAllocator.DEFAULT.buffer(10);
    buf.writeBytes(new byte[]{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'});
    // 在切片过程中并没有发生数据复制
    ByteBuf f1 = buf.slice(0, 5);
    f1.retain();
    ByteBuf f2 = buf.slice(5, 5);
    f2.retain();
    // 对 f1 进行相关的操作也会体现在 buf 上
}
```

- `ByteBuf duplicate()`：截取原始 ByteBuf 所有内容，并且没有 max capacity 的限制，也是与原始 ByteBuf 使用同一块底层内存，只是读写指针是独立的
- `CompositeByteBuf addComponents(boolean increaseWriterIndex, ByteBuf... buffers)`：合并多个 ByteBuf

```
public static void main(String[] args) {
    ByteBuf buf1 = ByteBufAllocator.DEFAULT.buffer();
    buf1.writeBytes(new byte[]{1, 2, 3, 4, 5});

    ByteBuf buf2 = ByteBufAllocator.DEFAULT.buffer();
    buf2.writeBytes(new byte[]{6, 7, 8, 9, 10});

    CompositeByteBuf buf = ByteBufAllocator.DEFAULT.compositeBuffer();
    // true 表示增加新的 ByteBuf 自动递增 write index，否则 write index 会始终为 0
    buf.addComponents(true, buf1, buf2);
}
```

CompositeByteBuf 是一个组合的 ByteBuf，内部维护了一个 Component 数组，每个 Component 管理一个 ByteBuf，记录了这个 ByteBuf 相对于整体偏移量等信息，代表着整体中某一段的数据

- 优点：对外是一个虚拟视图，组合这些 ByteBuf 不会产生内存复制
- 缺点：复杂了很多，多次操作会带来性能的损耗

深拷贝：

- `ByteBuf copy()`: 将底层内存数据进行深拷贝，因此无论读写，都与原始 ByteBuf 无关

池化相关：

- Unpooled 是一个工具类，提供了非池化的 ByteBuf 创建、组合、复制等操作

```
ByteBuf buf1 = ByteBufAllocator.DEFAULT.buffer(5);
buf1.writeBytes(new byte[]{1, 2, 3, 4, 5});
ByteBuf buf2 = ByteBufAllocator.DEFAULT.buffer(5);
buf2.writeBytes(new byte[]{6, 7, 8, 9, 10});

// 当包装 ByteBuf 个数超过一个时，底层使用了 CompositeByteBuf，零拷贝思想
ByteBuf buf = Unpooled.wrappedBuffer(buf1, buf2);
```

粘包半包

现象演示

在 TCP 传输中，客户端发送消息时，实际上是将数据写入 TCP 的缓存，此时数据的大小和缓存的大小就会造成粘包和半包

- 当数据超过 TCP 缓存容量时，就会被拆分成多个包，通过 Socket 多次发送到服务端，服务端每次从缓存中取数据，产生半包问题
- 当数据小于 TCP 缓存容量时，缓存中可以存放多个包，客户端和服务端一次通信就可能传递多个包，这时候服务端就可能一次读取多个包，产生粘包的问题

代码演示：

- 客户端代码：

```
public class HelloWorldClient {
    public static void main(String[] args) {
        send();
    }

    private static void send() {
        NioEventLoopGroup worker = new NioEventLoopGroup();
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.group(worker);
            bootstrap.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws
                    Exception {
                    ch.pipeline().addLast(new ChannelInboundHandlerAdapter()
                }
            });
        }
    }
}
```

```
// 【在连接 channel 建立成功后，会触发 active 方法】
@Override
public void channelActive(ChannelHandlerContext ctx)
throws Exception {
    // 发送内容随机的数据包
    Random r = new Random();
    char c = '0';
    ByteBuf buf = ctx.alloc().buffer();
    for (int i = 0; i < 10; i++) {
        byte[] bytes = new byte[10];
        for (int j = 0; j < r.nextInt(9) + 1; j++) {
            bytes[j] = (byte) c;
        }
        c++;
        buf.writeBytes(bytes);
    }
    ctx.writeAndFlush(buf);
}
});

});
});

ChannelFuture channelFuture = bootstrap.connect("127.0.0.1",
8080).sync();
channelFuture.channel().closeFuture().sync();

} catch (InterruptedException e) {
    log.error("client error", e);
} finally {
    worker.shutdownGracefully();
}
}
```

- 服务器代码:

```
public class HelloWorldServer {
    public static void main(String[] args) {
        NioEventLoopGroup boss = new NioEventLoopGroup(1);
        NioEventLoopGroup worker = new NioEventLoopGroup();
        try {
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.channel(NioServerSocketChannel.class);
            // 调整系统的接受缓冲区【滑动窗口】
            //serverBootstrap.option(ChannelOption.SO_RCVBUF, 10);
            // 调整 netty 的接受缓冲区（ByteBuf）
            //serverBootstrap.childOption(ChannelOption.RCVBUF_ALLOCATOR,
            //                            new
AdaptiveRecvByteBufAllocator(16, 16, 16));
            serverBootstrap.group(boss, worker);
            serverBootstrap.childHandler(new
ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws
Exception {
                    // 【这里可以添加解码器】
                }
            });
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

        // LoggingHandler 用来打印消息
        ch.pipeline().addLast(new
LoggingHandler(LogLevel.DEBUG));
    }
}

ChannelFuture channelFuture = serverBootstrap.bind(8080);
channelFuture.sync();
channelFuture.channel().closeFuture().sync();
} catch (InterruptedException e) {
    log.error("server error", e);
} finally {
    boss.shutdownGracefully();
    worker.shutdownGracefully();
    log.debug("stop");
}
}
}
}

```

- 粘包效果展示:

```

09:57:27.140 [nioEventLoopGroup-3-1] DEBUG
io.netty.handler.logging.LoggingHandler - [id: 0xddbaaef6, L:/127.0.0.1:8080
- R:/127.0.0.1:8701] READ: 100B // 读了 100 字节, 发生粘包
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
+
|00000000| 30 30 30 30 30 00 00 00 00 00 31 00 00 00 00 00
|0000.....1....|
|00000010| 00 00 00 00 32 32 32 32 00 00 00 00 00 00 00 33 00
|.....2222.....3.|
|00000020| 00 00 00 00 00 00 00 00 34 34 00 00 00 00 00 00 00
|.....44.....|
|00000030| 00 00 35 35 35 35 00 00 00 00 00 00 36 36 36 00
|..5555.....666.|
|00000040| 00 00 00 00 00 00 37 37 37 37 00 00 00 00 00 00 00
|.....7777.....|
|00000050| 38 38 38 38 38 00 00 00 00 00 39 39 00 00 00 00 00
|88888.....99....|
|00000060| 00 00 00 00
|
+-----+
+

```

解决方法：通过调整系统的接受缓冲区的滑动窗口和 Netty 的接受缓冲区保证每条包只含有一条数据，滑动窗口的大小仅决定了 Netty 读取的**最小单位**，实际每次读取的一般是它的整数倍

解决方案

短连接

发一个包建立一次连接，这样连接建立到连接断开之间就是消息的边界，缺点就是效率很低

客户端代码改造：

```
public class HelloWorldClient {
    public static void main(String[] args) {
        // 分 10 次发送
        for (int i = 0; i < 10; i++) {
            send();
        }
    }
}
```

固定长度

服务器端加入定长解码器，每一条消息采用固定长度。如果是半包消息，会缓存半包消息并等待下个包到达之后进行拼包合并，直到读取一个完整的消息包；如果是粘包消息，空余的位置会进行补 0，会浪费空间

```
serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(new FixedLengthFrameDecoder(10));
        // LoggingHandler 用来打印消息
        ch.pipeline().addLast(new LoggingHandler(LogLevel.DEBUG));
    }
});
```

```
10:29:06.522 [nioEventLoopGroup-3-1] DEBUG
io.netty.handler.logging.LoggingHandler - [id: 0x38a70fbf, L:/127.0.0.1:8080 - R:/127.0.0.1:10144] READ: 10B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 31 31 00 00 00 00 00 00 00 00 |11..... |
+-----+
10:29:06.522 [nioEventLoopGroup-3-1] DEBUG
io.netty.handler.logging.LoggingHandler - [id: 0x38a70fbf, L:/127.0.0.1:8080 - R:/127.0.0.1:10144] READ: 10B
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 32 32 32 32 32 32 00 00 00 00 |222222.... |
+-----+
```

分隔符

服务端加入行解码器， 默认以 `\n` 或 `\r\n` 作为分隔符，如果超出指定长度仍未出现分隔符，则抛出异常：

```
serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(new FixedLengthFrameDecoder(8));
        ch.pipeline().addLast(new LoggingHandler(LogLevel.DEBUG));
    }
});
```

客户端在每条消息之后，加入 `\n` 分隔符：

```
public void channelActive(ChannelHandlerContext ctx) throws Exception {
    Random r = new Random();
    char c = 'a';
    ByteBuf buffer = ctx.alloc().buffer();
    for (int i = 0; i < 10; i++) {
        for (int j = 1; j <= r.nextInt(16)+1; j++) {
            buffer.writeByte((byte) c);
        }
        // 10 代表 '\n'
        buffer.writeByte(10);
        c++;
    }
    ctx.writeAndFlush(buffer);
}
```

预设长度

LengthFieldBasedFrameDecoder 解码器自定义长度解决 TCP 粘包黏包问题

```
int maxFrameLength      // 数据最大长度
int lengthFieldOffset   // 长度字段偏移量，从第几个字节开始是内容的长度字段
int lengthFieldLength   // 长度字段本身的长度
int lengthAdjustment    // 长度字段为基准，几个字节后才是内容
int initialBytesToStrip // 从头开始剥离几个字节解码后显示
```

```

lengthFieldOffset = 1 (= the length of HDR1)
lengthFieldLength = 2
lengthAdjustment = 1 (= the length of HDR2)
initialBytesToStrip = 3 (= the length of HDR1 + LEN)

BEFORE DECODE (16 bytes)                                AFTER DECODE (13 bytes) //解码
+-----+-----+-----+-----+
| HDR1 | Length | HDR2 | Actual Content |----->| HDR2 | Actual Content |
| 0xCA | 0x000C | 0xFE | "HELLO, WORLD" |           | 0xFE | "HELLO, WORLD" |
+-----+-----+-----+-----+

```

代码实现：

```

public class LengthFieldDecoderDemo {
    public static void main(String[] args) {
        EmbeddedChannel channel = new EmbeddedChannel(
            // int 占 4 字节, 版本号一个字节
            new LengthFieldBasedFrameDecoder(1024, 0, 4, 1, 5),
            new LoggingHandler(LogLevel.DEBUG)
        );

        // 4 个字节的内容长度, 实际内容
        ByteBuf buffer = ByteBufAllocator.DEFAULT.buffer();
        send(buffer, "Hello, world");
        send(buffer, "Hi!");
        // 写出缓存
        channel.writeInbound(buffer);
    }

    // 写入缓存
    private static void send(ByteBuf buffer, String content) {
        byte[] bytes = content.getBytes(); // 实际内容
        int length = bytes.length;          // 实际内容长度
        buffer.writeInt(length);
        buffer.writeByte(1);                // 表示版本号
        buffer.writeBytes(bytes);
    }
}

```

```

10:49:59.344 [main] DEBUG io.netty.handler.logging.LoggingHandler - [id: 0xembedded, L:embedded - R:embedded] READ: 12B

```

```

+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 |Hello, world|
+-----+

```

```

10:49:59.344 [main] DEBUG io.netty.handler.logging.LoggingHandler - [id: 0xembedded, L:embedded - R:embedded] READ: 3B

```

```

+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 48 69 21 |Hi!|
+-----+

```

协议设计

HTTP

访问 URL: <http://localhost:8080/>

```
public class HttpDemo {
    public static void main(String[] args) {
        NioEventLoopGroup boss = new NioEventLoopGroup();
        NioEventLoopGroup worker = new NioEventLoopGroup();
        try {
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.channel(NioServerSocketChannel.class);
            serverBootstrap.group(boss, worker);
            serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>()
{
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(new LoggingHandler(LogLevel.DEBUG));
        ch.pipeline().addLast(new HttpServerCodec());
        // 只针对某一种类型的请求处理，此处针对 HttpRequest
        ch.pipeline().addLast(new
SimpleChannelInboundHandler<HttpRequest>() {
            @Override
            protected void channelRead0(ChannelHandlerContext ctx,
HttpRequest msg) {
                // 获取请求
                log.debug(msg.uri());

                // 返回响应
                DefaultFullHttpResponse response = new
DefaultFullHttpResponse(
                    msg.protocolVersion(), HttpResponseStatus.OK);

                byte[] bytes = "<h1>Hello, world!</h1>".getBytes();

                response.headers().setInt(CONTENT_LENGTH,
bytes.length);
                response.content().writeBytes(bytes);

                // 写回响应
                ctx.writeAndFlush(response);
            }
        });
    }
});
        ChannelFuture channelFuture = serverBootstrap.bind(8080).sync();
        channelFuture.channel().closeFuture().sync();
    } catch (InterruptedException e) {
        log.error("n3.server error", e);
    }
}
```

```
        } finally {
            boss.shutdownGracefully();
            worker.shutdownGracefully();
        }
    }
}
```

自定义

处理器代码：

```
@Slf4j
public class MessageCodec extends ByteToMessageCodec<Message> {
    // 编码
    @Override
    public void encode(ChannelHandlerContext ctx, Message msg, ByteBuf out)
        throws Exception {
        // 4 字节的魔数
        out.writeBytes(new byte[]{1, 2, 3, 4});
        // 1 字节的版本,
        out.writeByte(1);
        // 1 字节的序列化方式 jdk 0 , json 1
        out.writeByte(0);
        // 1 字节的指令类型
        out.writeByte(msg.getMessageType());
        // 4 个字节
        out.writeInt(msg.getSequenceId());
        // 无意义, 对齐填充, 1 字节
        out.writeByte(0xff);
        // 获取内容的字节数组, msg 对象序列化
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bos);
        oos.writeObject(msg);
        byte[] bytes = bos.toByteArray();
        // 长度
        out.writeInt(bytes.length);
        // 写入内容
        out.writeBytes(bytes);
    }

    // 解码
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
        out) throws Exception {
        int magicNum = in.readInt();
        byte version = in.readByte();
        byte serializerType = in.readByte();
        byte messageType = in.readByte();
        int sequenceId = in.readInt();
```

```
    in.readByte();
    int length = in.readInt();
    byte[] bytes = new byte[length];
    in.readBytes(bytes, 0, length);
    ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(bytes));
    Message message = (Message) ois.readObject();
    log.debug("{} , {} , {} , {} , {} , {}" , magicNum, version, serializerType,
messageType, sequenceId, length);
    log.debug("{}" , message);
    out.add(message);
}
}
```

测试代码：

```
public static void main(String[] args) throws Exception {
    EmbeddedChannel channel = new EmbeddedChannel(new LoggingHandler(), new
MessageCodec());
    // encode
    LoginRequestMessage message = new LoginRequestMessage("zhangsan", "123");
    channel.writeOutbound(message);

    // decode
    ByteBuf buf = ByteBufAllocator.DEFAULT.buffer();
    new MessageCodec().encode(null, message, buf);
    // 入站
    channel.writeInbound(buf);
}

public class LoginRequestMessage extends Message {
    private String username;
    private String password;
    // set + get
}
```

	魔数	版本	序列化算法	消息类型	消息序号	消息正文长度	消息正文
	+-----+ 0 1 2 3 4 5 6 7 8 9 a b c d e f +-----+						
00000000	12 34 56 78 01 00 00 00 00 02 ff 00 00 00 db	4Vx.....					
00000010	ac ed 00 05 73 72 00 2c 63 6f 6d 2e 69 74 63 61sr.,com.itca					
00000020	73 74 2e 6e 65 74 74 79 2e 70 6f 74 6f 63 6f 6c	st.netty.protocol					
00000030	2e 4c 6f 67 69 6e 52 65 71 75 65 73 74 4d 65 73	.LoginRequestMes					
00000040	73 61 67 65 aa d6 eb c4 4b 8c ce be 02 00 03 4c	sage....K.....L					
00000050	00 08 6e 69 63 6b 6e 61 6d 65 74 00 12 4c 6a 61	..nickname..Lja					
00000060	76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 4c	va/lang/String;L					
00000070	00 08 70 61 73 73 77 6f 72 64 71 00 7e 00 01 4c	..passwordq.~..L					
00000080	00 08 75 73 65 72 6e 61 6d 65 71 00 7e 00 01 78	..usernameq.~..x					
00000090	72 00 20 63 6f 6d 2e 69 74 63 61 73 74 2e 6e 65	r. com.itcast.ne					
000000a0	74 74 79 2e 70 6f 74 6f 63 6f 6c 2e 4d 65 73 73	tty.protocol.Mess					
000000b0	61 67 65 e3 8d 05 6d af c7 fd 1d 02 00 01 49 00	age...m.....I.					
000000c0	0a 73 65 71 75 65 6e 63 65 49 64 78 70 00 00 00	.sequenceIdxp...					
000000d0	02 74 00 06 e5 b0 8f e5 bc a0 74 00 03 31 32 33	.t.....t..123					
000000e0	74 00 08 7a 68 61 6e 67 73 61 6e	it..zhangsan					
	+-----+						

Sharable

@Sharable 注解的添加时机：

- 当 handler 不保存状态时，就可以安全地在多线程下被共享
- 对于编解码器类不能继承 ByteToMessageCodec 或 CombinedChannelDuplexHandler，它们的构造方法对 @Sharable 有限制

```
protected ByteToMessageCodec(boolean preferDirect) {
    ensureNotSharable();
    outboundMsgMatcher = TypeParameterMatcher.find(this,
        ByteToMessageCodec.class, "I");
    encoder = new Encoder(preferDirect);
}
```

```
protected void ensureNotSharable() {
    // 如果类上有该注解
    if (isSharable()) {
        throw new IllegalStateException();
    }
}
```

- 如果能确保编解码器不会保存状态，可以继承 MessageToMessageCodec 父类

```
@Slf4j
@ChannelHandler.Sharable
```

```

// 必须和 LengthFieldBasedFrameDecoder 一起使用，确保接到的 ByteBuf 消息是完整的
public class MessageCodecSharable extends MessageToMessageCodec<ByteBuf,
Message> {
    @Override
    protected void encode(ChannelHandlerContext ctx, Message msg,
List<Object> outList) throws Exception {
        ByteBuf out = ctx.alloc().buffer();
        // 4 字节的魔数
        out.writeBytes(new byte[]{1, 2, 3, 4});
        // ....
        outList.add(out);
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in,
List<Object> out) throws Exception {
        //....
    }
}

```

场景优化

空闲检测

连接假死

连接假死就是客户端数据发不出去，服务端也一直收不到数据，保持这种状态，假死的连接占用的资源不能自动释放，而且向假死连接发送数据，得到的反馈是发送超时

解决方案：每隔一段时间就检查这段时间内是否接收到客户端数据，没有就可以判定为连接假死

IdleStateHandler 是 Netty 提供的处理空闲状态的处理器，用来判断是不是读空闲时间或写空闲时间过长

- 参数一 long readerIdleTime：读空闲，表示多长时间没有读
- 参数二 long writerIdleTime：写空闲，表示多长时间没有写
- 参数三 long allIdleTime：读写空闲，表示多长时间没有读写

```

serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(new LengthFieldBasedFrameDecoder(1024, 12, 4, 0,
0));
        ch.pipeline().addLast(new MessageCodec());
        // 5s 内如果没有收到 channel 的数据，会触发一个 IdleState#READER_IDLE 事件，
        ch.pipeline().addLast(new IdleStateHandler(5, 0, 0));
        // ChannelDuplexHandler 【可以同时作为入站和出站】处理器
    }
})

```

```

        ch.pipeline().addLast(new ChannelDuplexHandler() {
            // 用来触发特殊事件
            @Override
            public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception{
                IdleStateEvent event = (IdleStateEvent) evt;
                // 触发了读空闲事件
                if (event.state() == IdleState.READER_IDLE) {
                    log.debug("已经 5s 没有读到数据了");
                    ctx.channel().close();
                }
            }
        });
    }
}

```

心跳机制

客户端定时向服务器端发送数据，**时间间隔要小于服务器定义的空闲检测的时间间隔**，就能防止误判连接假死，这就是心跳机制

```

bootstrap.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(new LengthFieldBasedFrameDecoder(1024, 12, 4, 0,
0));
        ch.pipeline().addLast(new MessageCodec());
        // 3s 内如果没有向服务器写数据，会触发一个 IdleState#WRITER_IDLE 事件
        ch.pipeline().addLast(new IdleStateHandler(0, 3, 0));
        // ChannelDuplexHandler 可以同时作为入站和出站处理器
        ch.pipeline().addLast(new ChannelDuplexHandler() {
            // 用来触发特殊事件
            @Override
            public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {
                IdleStateEvent event = (IdleStateEvent) evt;
                // 触发了写空闲事件
                if (event.state() == IdleState.WRITER_IDLE) {
                    // 3s 没有写数据了，【发送一个心跳包】
                    ctx.writeAndFlush(new PingMessage());
                }
            }
        });
    }
}

```

序列化

普通方式

序列化，反序列化主要用在消息正文的转换上

- 序列化时，需要将 Java 对象变为要传输的数据（可以是 byte[]，或 json 等，最终都需要变成 byte[]）
- 反序列化时，需要将传入的正文数据还原成 Java 对象，便于处理

代码实现：

- 抽象一个 Serializer 接口

```
public interface Serializer {  
    // 反序列化方法  
    <T> T deserialize(Class<T> clazz, byte[] bytes);  
    // 序列化方法  
    <T> byte[] serialize(T object);  
}
```

- 提供两个实现

```
enum SerializerAlgorithm implements Serializer {  
    // Java 实现  
    Java {  
        @Override  
        public <T> T deserialize(Class<T> clazz, byte[] bytes) {  
            try {  
                ObjectInputStream in =  
                    new ObjectInputStream(new ByteArrayInputStream(bytes));  
                Object object = in.readObject();  
                return (T) object;  
            } catch (IOException | ClassNotFoundException e) {  
                throw new RuntimeException("SerializerAlgorithm.Java 反序列化错误", e);  
            }  
        }  
  
        @Override  
        public <T> byte[] serialize(T object) {  
            try {  
                ByteArrayOutputStream out = new ByteArrayOutputStream();  
                new ObjectOutputStream(out).writeObject(object);  
                return out.toByteArray();  
            } catch (IOException e) {  
                throw new RuntimeException("SerializerAlgorithm.Java 序列化错误", e);  
            }  
        }  
    },  
    // JSON 实现(引入了 Gson 依赖)  
    JSON {  
        @Override  
        public <T> T deserialize(Class<T> clazz, byte[] bytes) {  
            try {  
                Gson gson = new Gson();  
                String jsonStr = new String(bytes);  
                return gson.fromJson(jsonStr, clazz);  
            } catch (Exception e) {  
                throw new RuntimeException("SerializerAlgorithm.JSON 反序列化错误", e);  
            }  
        }  
  
        @Override  
        public <T> byte[] serialize(T object) {  
            try {  
                Gson gson = new Gson();  
                String jsonStr = gson.toJson(object);  
                return jsonStr.getBytes();  
            } catch (Exception e) {  
                throw new RuntimeException("SerializerAlgorithm.JSON 序列化错误", e);  
            }  
        }  
    }  
}
```

```

    @Override
    public <T> T deserialize(Class<T> clazz, byte[] bytes) {
        return new Gson().fromJson(new String(bytes,
StandardCharsets.UTF_8), clazz);
    }

    @Override
    public <T> byte[] serialize(T object) {
        return new
Gson().toJson(object).getBytes(StandardCharsets.UTF_8);
    }
};

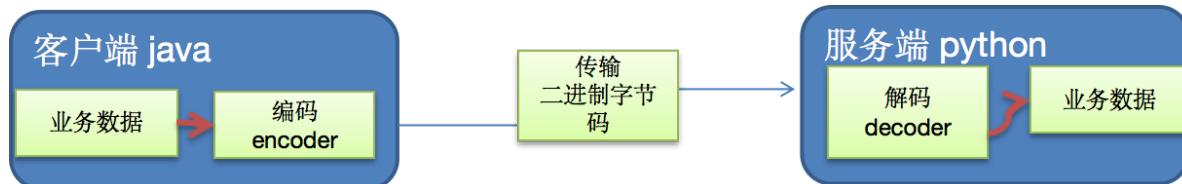
// 需要从协议的字节中得到是哪种序列化算法
public static SerializerAlgorithm getByInt(int type) {
    SerializerAlgorithm[] array = SerializerAlgorithm.values();
    if (type < 0 || type > array.length - 1) {
        throw new IllegalArgumentException("超过 SerializerAlgorithm 范
围");
    }
    return array[type];
}
}

```

ProtoBuf

基本介绍

Codec (编解码器) 的组成部分有两个: Decoder (解码器) 和 Encoder (编码器) 。Encoder 负责把业务数据转换成字节码数据, Decoder 负责把字节码数据转换成业务数据



Protobuf 是 Google 发布的开源项目, 全称 Google Protocol Buffers , 是一种轻便高效的结构化数据存储格式, 可以用于结构化数据串行化, 或者说序列化。很适合做数据存储或 RPC (远程过程调用 remote procedure call) 数据交换格式。目前很多公司从 HTTP + Json 转向 TCP + Protobuf , 效率会更高

Protobuf 是以 message 的方式来管理数据, 支持跨平台、跨语言 (客户端和服务器端可以是不同的语言编写的) , 高性能、高可靠性

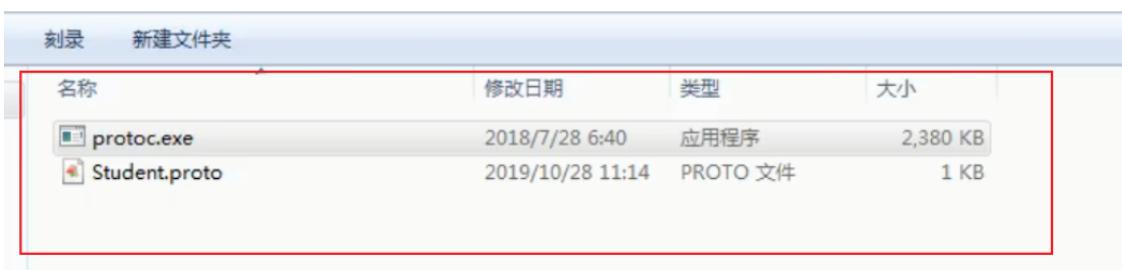
工作过程: 使用 Protobuf 编译器自动生成代码, Protobuf 是将类的定义使用 .proto 文件进行描述, 然后通过 protoc.exe 编译器根据 .proto 自动生成 .java 文件

代码实现

- 单个 message:

```
syntax = "proto3";                                     // 版本
option java_outer_classname = "StudentPOJO";           // 生成的外部类名，同时也是文件名

message Student {    // 在 StudentPOJO 外部类中生成一个内部类 Student，是真正发送的
    POJO 对象
    int32 id = 1;    // Student 类中有一个属性：名字为 id 类型为 int32(protobuf类
    型) ，1表示属性序号，不是值
    string name = 2;
}
```



编译 `protoc.exe --java_out=. Student.proto` (cmd 窗口输入) 将生成的 StudentPOJO 放入到项目使用

Server 端:

```
new ServerBootstrap() //...
    .childHandler(new ChannelInitializer<SocketChannel>() { // 创建一个通道初始化对象
        // 给 pipeline 设置处理器
        @Override
        protected void initChannel(SocketChannel ch) throws Exception {
            // 在 pipeline 加入 ProtoBufDecoder，指定对哪种对象进行解码
            ch.pipeline().addLast("decoder", new ProtobufDecoder(
                StudentPOJO.Student.getDefaultInstance()));
            ch.pipeline().addLast(new NettyServerHandler());
        }
    });
}
```

Client 端:

```

new Bootstrap().group(group)           // 设置线程组
    .channel(NioSocketChannel.class)    // 设置客户端通道的实现类(反射)
    .handler(new ChannelInitializer<SocketChannel>() {
        @Override
        protected void initChannel(SocketChannel ch) throws Exception {
            // 在pipeline中加入 ProtoBufEncoder
            ch.pipeline().addLast("encoder", new ProtobufEncoder());
            ch.pipeline().addLast(new NettyClientHandler()); // 加入自定义的业
务处理器
        }
    });
}

```

- 多个 message: Protobuf 可以使用 message 管理其他的 message。假设某个项目需要传输 20 个对象，可以在一个文件里定义 20 个 message，最后再用一个总的 message 来决定在实际传输时真正需要传输哪一个对象

```

syntax = "proto3";
option optimize_for = SPEED;           // 加快解析
option java_package="com.atguigu.netty.codec2"; // 指定生成到哪个包下
option java_outer_classname="MyDataInfo"; // 外部类名，文件名

message MyMessage {
    // 定义一个枚举类型，DataType 如果是 0 则表示一个 Student 对象实例，DataType 这个名称自定义
    enum DataType {
        StudentType = 0; //在 proto3 要求 enum 的编号从 0 开始
        WorkerType = 1;
    }

    // 用 data_type 来标识传的是哪一个枚举类型，这里才真正开始定义 Message 的数据类型
    DataType data_type = 1; // 所有后面的数字都只是编号而已

    // oneof 关键字，表示每次枚举类型进行传输时，限制最多只能传输一个对象。
    // dataBody名称也是自定义的
    // MyMessage 里出现的类型只有两个 DataType 类型，student 或者 worker 类型，在真正传输的时候只会有一个出现
    oneof dataBody {
        Student student = 2; //注意这后面的数字也都只是编号而已，上面DataType
data_type = 1 占了第一个序号了
        Worker worker = 3;
    }

}

message Student {
    int32 id = 1;          // Student类的属性
    string name = 2;       //
}

message Worker {
    string name=1;
    int32 age=2;
}

```

编译:

Server 端:

```
ch.pipeline().addLast("decoder", new  
ProtobufDecoder(MyDataInfo.MyMessage.getDefaultInstance()));
```

Client 端:

```
pipeline.addLast("encoder", new ProtobufEncoder());
```

长连接

HTTP 协议是无状态的，浏览器和服务器间的请求响应一次，下一次会重新创建连接。实现基于 WebSocket 的长连接的全双工的交互，改变 HTTP 协议多次请求的约束

开发需求:

- 实现长连接，服务器与浏览器相互通信客户端
- 浏览器和服务器端会相互感知，比如服务器关闭了，浏览器会感知，同样浏览器关闭了，服务器会感知

代码实现:

- WebSocket:
 - WebSocket 的数据是以帧 (frame) 形式传递，WebSocketFrame 下面有六个子类，代表不同的帧格式
 - 浏览器请求 URL: ws://localhost:8080/xxx

```
public class MyWebSocket {  
    public static void main(String[] args) throws Exception {  
        // 创建两个线程组  
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);  
        EventLoopGroup workerGroup = new NioEventLoopGroup();  
        try {  
  
            ServerBootstrap serverBootstrap = new ServerBootstrap();  
            serverBootstrap.group(bossGroup, workerGroup);  
            serverBootstrap.channel(NioServerSocketChannel.class);  
            serverBootstrap.handler(new LoggingHandler(LogLevel.INFO));  
            serverBootstrap.childHandler(new  
ChannelInitializer<SocketChannel>() {  
                @Override  
                protected void initChannel(SocketChannel ch) throws  
Exception {  
                    ChannelPipeline pipeline = ch.pipeline();  
  
                    // 基于 http 协议，使用 http 的编码和解码器  
                }  
            });  
            ChannelFuture future = serverBootstrap.bind(8080).sync();  
            future.channel().closeFuture().sync();  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            bossGroup.shutdownGracefully();  
            workerGroup.shutdownGracefully();  
        }  
    }  
}
```

```
        pipeline.addLast(new HttpServerCodec());
        // 是以块方式写, 添加 ChunkedWriteHandler 处理器
        pipeline.addLast(new ChunkedWriteHandler());

        // http 数据在传输过程中是分段, HttpObjectAggregator 就是可以
        // 将多个段聚合
        // 这就是为什么, 当浏览器发送大量数据时, 就会发出多次 http 请求
        pipeline.addLast(new HttpObjectAggregator(8192));

        // webSocketServerProtocolHandler 核心功能是【将 http 协议升
        // 级为 ws 协议】, 保持长连接
        pipeline.addLast(new
webSocketServerProtocolHandler("/hello"));

        // 自定义的handler , 处理业务逻辑
        pipeline.addLast(new MyTextWebSocketFrameHandler());
    }
});

// 启动服务器
ChannelFuture channelFuture = serverBootstrap.bind(8080).sync();
channelFuture.channel().closeFuture().sync();

} finally {
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}
}
```

- 处理器：

```
public class MyTextWebSocketFrameHandler extends SimpleChannelInboundHandler<TextWebSocketFrame> {
    // TextWebSocketFrame 类型，表示一个文本帧(frame)
    @Override
    protected void channelRead0(ChannelHandlerContext ctx,
        TextWebSocketFrame msg) throws Exception {
        System.out.println("服务器收到消息 " + msg.text());
        // 回复消息
        ctx.writeAndFlush(new TextWebSocketFrame("服务器时间" +
LocalDateTime.now() + " " + msg.text()));
    }

    // 当web客户端连接后， 触发方法
    @Override
    public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
        // id 表示唯一的值， LongText 是唯一的 ShortText 不是唯一
        System.out.println("handlerAdded 被调用" +
ctx.channel().id().asLongText());
        System.out.println("handlerAdded 被调用" +
ctx.channel().id().asShortText());
    }

    @Override
```

```

public void handlerRemoved(ChannelHandlerContext ctx) throws Exception {
    System.out.println("handlerRemoved 被调用" +
    ctx.channel().id().asLongText());
}

@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
    System.out.println("异常发生 " + cause.getMessage());
    ctx.close(); // 关闭连接
}
}

```

- HTML:

```

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Title</title>
</head>
<body>
<script>
var socket;
// 判断当前浏览器是否支持websocket
if(window.WebSocket) {
    //go on
    socket = new WebSocket("ws://localhost:8080/hello");
    //相当于channelReado, ev 收到服务器端回送的消息
    socket.onmessage = function (ev) {
        var rt = document.getElementById("responseText");
        rt.value = rt.value + "\n" + ev.data;
    }

    //相当于连接开启(感知到连接开启)
    socket.onopen = function (ev) {
        var rt = document.getElementById("responseText");
        rt.value = "连接开启了.."
    }

    //相当于连接关闭(感知到连接关闭)
    socket.onclose = function (ev) {

        var rt = document.getElementById("responseText");
        rt.value = rt.value + "\n" + "连接关闭了.."
    }
} else {
    alert("当前浏览器不支持websocket")
}

// 发送消息到服务器
function send(message) {
    // 先判断socket是否创建好
    if(!window.socket) {
        return;
    }
}

```

```

        }
        if(socket.readyState == websocket.OPEN) {
            // 通过socket 发送消息
            socket.send(message)
        } else {
            alert("连接没有开启");
        }
    }

```

```

</script>
<form onsubmit="return false">
    <textarea name="message" style="height: 300px; width: 300px">
    </textarea>
    <input type="button" value="发生消息"
    onclick="send(this.form.message.value)">
    <textarea id="responseText" style="height: 300px; width: 300px">
    </textarea>
    <input type="button" value="清空内容"
    onclick="document.getElementById('responseText').value=''">
</form>

```

```

</body>
</html>

```

参数调优

CONNECT

参数配置方式:

- 客户端通过 .option() 方法配置参数, 给 SocketChannel 配置参数
- 服务器端:
 - new ServerBootstrap().option(): 给 ServerSocketChannel 配置参数
 - new ServerBootstrap().childOption(): 给 SocketChannel 配置参数

CONNECT_TIMEOUT_MILLIS 参数:

- 属于 SocketChannal 参数
- 在客户端建立连接时, 如果在指定毫秒内无法连接, 会抛出 timeout 异常
- SO_TIMEOUT 主要用在阻塞 IO, 阻塞 IO 中 accept, read 等都是无限等待的, 如果不希望永远阻塞, 可以调整超时时间

```

public class ConnectionTimeoutTest {
    public static void main(String[] args) {
        NioEventLoopGroup group = new NioEventLoopGroup();
        try {
            Bootstrap bootstrap = new Bootstrap()
                .group(group)
                .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 5000)
                .channel(NioSocketChannel.class)
        }
    }
}

```

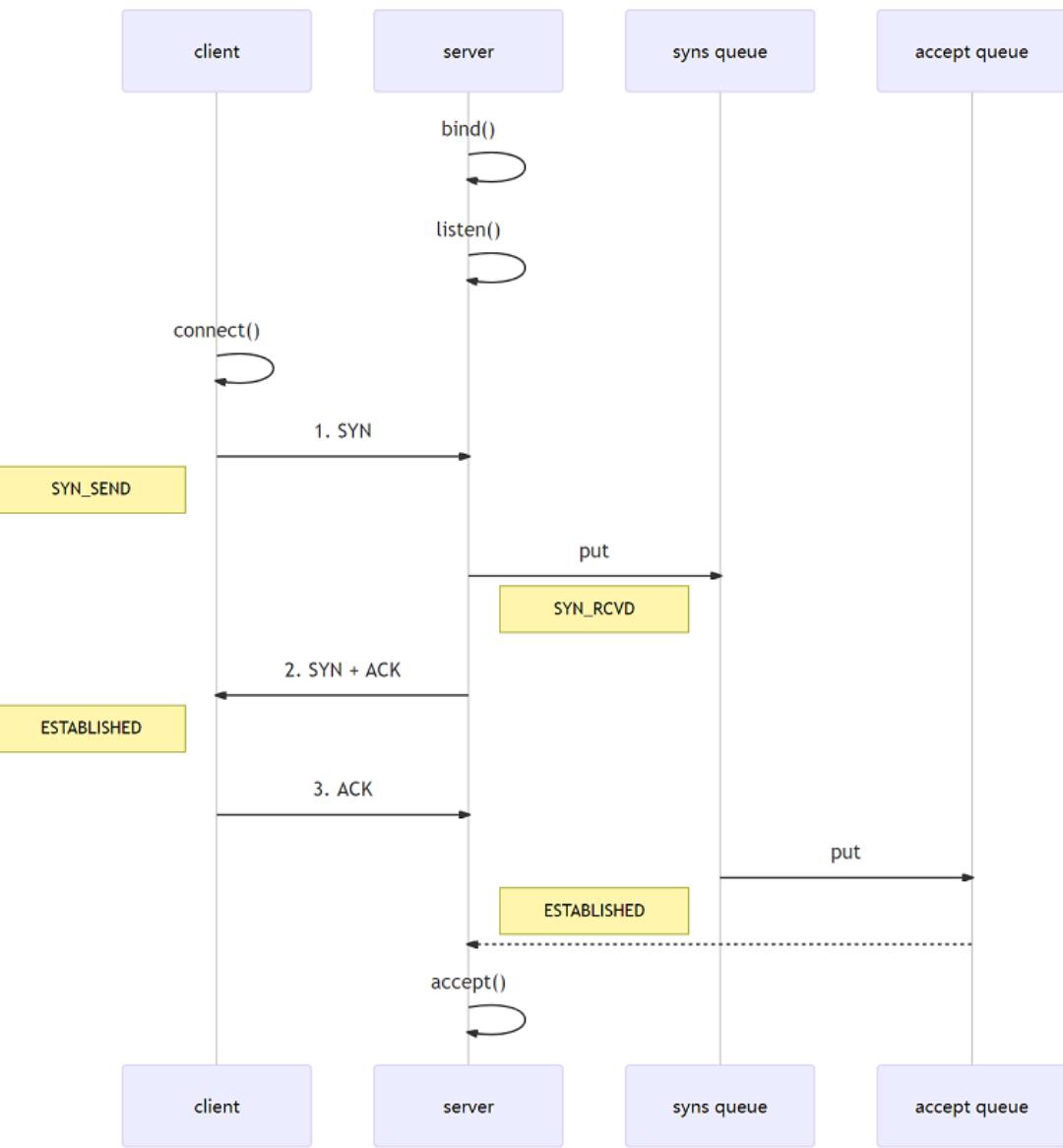
```
        .handler(new LoggingHandler());
    ChannelFuture future = bootstrap.connect("127.0.0.1", 8080);
    future.sync().channel().closeFuture().sync();
} catch (Exception e) {
    e.printStackTrace();
    log.debug("timeout");
} finally {
    group.shutdownGracefully();
}
}
```

SO_BACKLOG

属于 ServerSocketChannal 参数，通过 `option(ChannelOption.SO_BACKLOG, value)` 来设置大小

在 Linux 2.2 之前，backlog 大小包括了两个队列的大小，在 2.2 之后，分别用下面两个参数来控制

- sync queue：半连接队列，大小通过 `/proc/sys/net/ipv4/tcp_max_syn_backlog` 指定，在 `syncookies` 启用的情况下，逻辑上没有最大值限制
- accept queue：全连接队列，大小通过 `/proc/sys/net/core/somaxconn` 指定，在使用 listen 函数时，内核会根据传入的 backlog 参数与系统参数，取二者的较小值。如果 accpet queue 队列满了，server 将发送一个拒绝连接的错误信息到 client



其他参数

ALLOCATOR: 属于 SocketChannal 参数，用来分配 ByteBuf, ctx.alloc()

RCVBUF_ALLOCATOR: 属于 SocketChannal 参数

- 控制 Netty 接收缓冲区大小
- 负责入站数据的分配，决定入站缓冲区的大小（并可动态调整），统一采用 direct 直接内存，具体池化还是非池化由 allocator 决定

RocketMQ

基本介绍

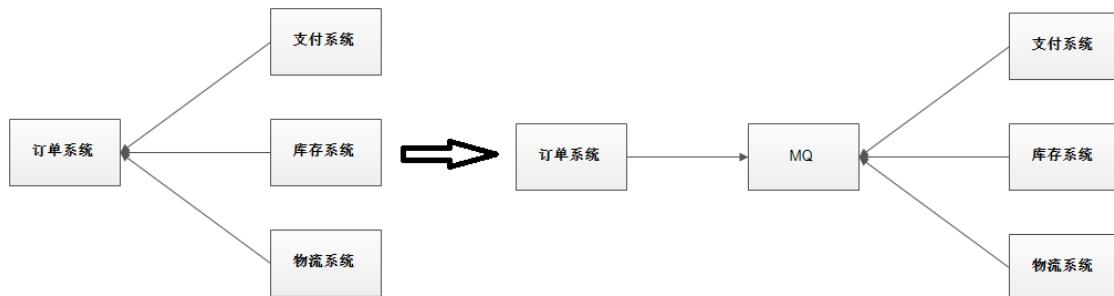
消息队列

应用场景

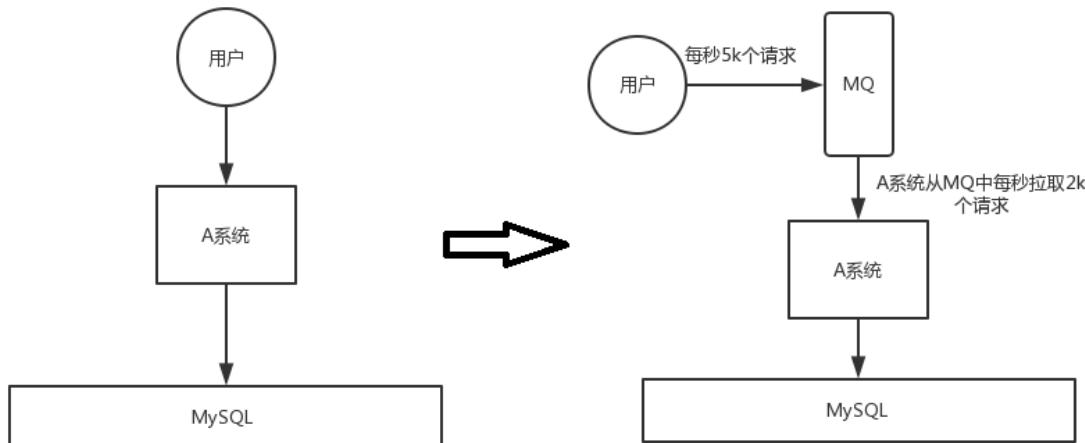
消息队列是一种先进先出的数据结构，常见的应用场景：

- **应用解耦：**系统的耦合性越高，容错性就越低

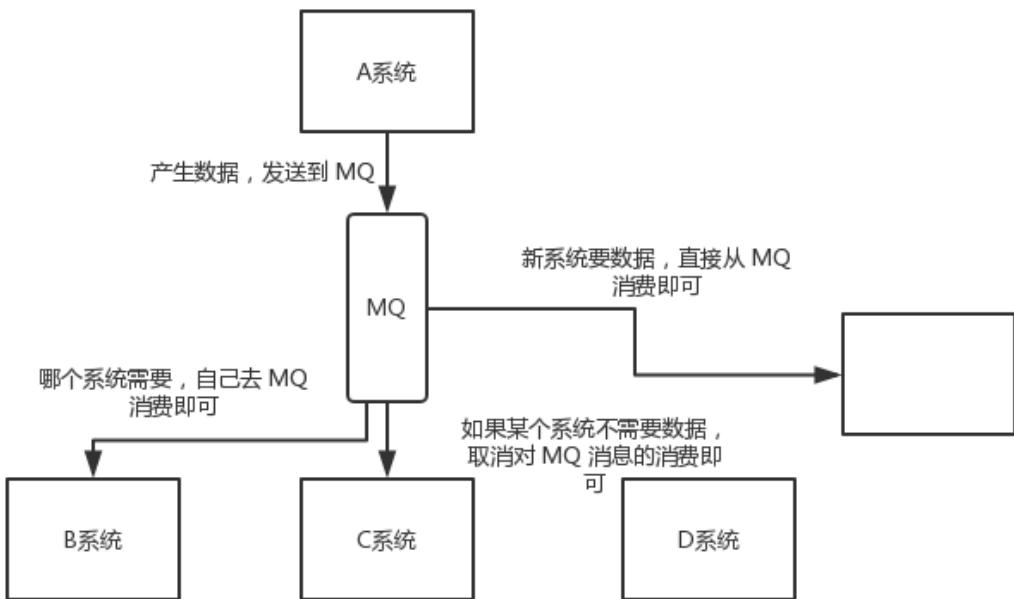
实例：用户创建订单后，耦合调用库存系统、物流系统、支付系统，任何一个子系统出了故障都会造成下单异常，影响用户使用体验。使用消息队列解耦合，比如物流系统发生故障，需要几分钟恢复，将物流系统要处理的数据缓存到消息队列中，用户的下单操作正常完成。等待物流系统正常后处理存在消息队列中的订单消息即可，终端系统感知不到物流系统发生过几分钟故障



- **流量削峰：**应用系统如果遇到系统请求流量的瞬间猛增，有可能会将系统压垮，使用消息队列可以将大量请求缓存起来，分散到很长一段时间处理，这样可以提高系统的稳定性和用户体验



- **数据分发：**让数据在多个系统之间进行流通，数据的产生方不需要关心谁来使用数据，只需要将数据发送到消息队列，数据使用方直接在消息队列中直接获取数据



参考视频：<https://www.bilibili.com/video/BV1L4411y7mn>

技术选型

RocketMQ 对比 Kafka 的优点

- 支持 Pull 和 Push 两种消息模式
- 支持延时消息、死信队列、消息重试、消息回溯、消息跟踪、事务消息等高级特性
- 对消息可靠性做了改进，**保证消息不丢失并且至少消费一次**，与 Kafka 一样是先写 PageCache 再落盘，并且数据有多副本
- RocketMQ 存储模型是所有的 Topic 都写到同一个 Commitlog 里，是一个 append only 操作，在海量 Topic 下也能将磁盘的性能发挥到极致，并且保持稳定的写入时延。Kafka 的吞吐非常高（零拷贝、操作系统页缓存、磁盘顺序写），但是在多 Topic 下时延不够稳定（顺序写入特性会被破坏从而引入大量的随机 I/O），不适合实时在线业务场景
- 经过阿里巴巴多年双 11 验证过、可以支持亿级并发的开源消息队列

Kafka 比 RocketMQ 吞吐量高：

- Kafka 将 Producer 端将多个小消息合并，采用异步批量发送的机制，当发送一条消息时，消息并没有发送到 Broker 而是缓存起来，直接向业务返回成功，当缓存的消息达到一定数量时再批量发送
- 减少了网络 I/O，提高了消息发送的性能，但是如果消息发送者宕机，会导致消息丢失，降低了可靠性
- RocketMQ 缓存过多消息会导致频繁 GC，并且为了保证可靠性没有采用这种方式

Topic 的 partition 数量过多时，Kafka 的性能不如 RocketMQ：

- 两者都使用文件存储，但是 Kafka 是一个分区一个文件，Topic 过多时分区的总量也会增加，过多的文件导致对消息刷盘时出现文件竞争磁盘，造成性能的下降。**一个分区只能被一个消费组中的一个消费线程进行消费**，因此可以同时消费的消费端也比较少

- RocketMQ 所有队列都存储在一个文件中，每个队列存储的消息量也比较小，因此多 Topic 的对 RocketMQ 的性能的影响较小
-

安装测试

安装需要 Java 环境，下载解压后进入安装目录，进行启动：

- 启动 NameServer

```
# 1.启动 NameServer  
nohup sh bin/mqnamesrv &  
# 2.查看启动日志  
tail -f ~/logs/rocketmqlogs/namesrv.log
```

RocketMQ 默认的虚拟机内存较大，需要编辑如下两个配置文件，修改 JVM 内存大小

```
# 编辑runbroker.sh和runserver.sh修改默认JVM大小  
vi runbroker.sh  
vi runserver.sh
```

参考配置：JAVA_OPT="\${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"

- 启动 Broker

```
# 1.启动 Broker  
nohup sh bin/mqbroker -n localhost:9876 autoCreateTopicEnable=true &  
# 2.查看启动日志  
tail -f ~/logs/rocketmqlogs/broker.log
```

- 发送消息：

```
# 1.设置环境变量  
export NAMESRV_ADDR=localhost:9876  
# 2.使用安装包的 Demo 发送消息  
sh bin/tools.sh org.apache.rocketmq.example.quickstart.Producer
```

- 接受消息：

```
# 1.设置环境变量  
export NAMESRV_ADDR=localhost:9876  
# 2.接收消息  
sh bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer
```

- 关闭 RocketMQ：

```
# 1. 关闭 NameServer  
sh bin/mqshutdown namesrv  
# 2. 关闭 Broker  
sh bin/mqshutdown broker
```

相关概念

RocketMQ 主要由 Producer、Broker、Consumer 三部分组成，其中 Producer 负责生产消息，Consumer 负责消费消息，Broker 负责存储消息，NameServer 负责管理 Broker

- 代理服务器（Broker Server）：消息中转角色，负责**存储消息、转发消息**。在 RocketMQ 系统中负责接收从生产者发送来的消息并存储、同时为消费者的拉取请求作准备，也存储消息相关的元数据，包括消费者组、消费进度偏移和主题和队列消息等
- 名字服务（Name Server）：充当**路由消息的提供者**。生产者或消费者能够通过名字服务查找各主题相应的 Broker IP 列表
- 消息生产者（Producer）：负责**生产消息**，把业务应用系统里产生的消息发送到 Broker 服务器。RocketMQ 提供多种发送方式，同步发送、异步发送、顺序发送、单向发送，同步和异步方式均需要 Broker 返回确认信息，单向发送不需要；可以通过 MQ 的负载均衡模块选择相应的 Broker 集群队列进行消息投递，投递的过程支持快速失败并且低延迟
- 消息消费者（Consumer）：负责**消费消息**，一般是后台系统负责异步消费，一个消息消费者会从 Broker 服务器拉取消息、并将其提供给应用程序。从用户应用的角度而提供了两种消费形式：
 - 拉取式消费（Pull Consumer）：应用通过主动调用 Consumer 的拉消息方法从 Broker 服务器拉消息，主动权由应用控制，一旦获取了批量消息，应用就会启动消费过程
 - 推动式消费（Push Consumer）：该模式下 Broker 收到数据后会主动推送给消费端，实时性较高
- 生产者组（Producer Group）：同一类 Producer 的集合，发送同一类消息且发送逻辑一致。如果发送的是事务消息且原始生产者在发送之后崩溃，则**Broker 服务器会联系同一生产者组的其他生产者实例以提交或回溯消费**
- 消费者组（Consumer Group）：同一类 Consumer 的集合，消费者实例必须订阅完全相同的 Topic，消费同一类消息且消费逻辑一致。消费者组使得在消息消费方面更容易的实现负载均衡和容错。RocketMQ 支持两种消息模式：
 - 集群消费（Clustering）：相同 Consumer Group 的每个 Consumer 实例平均分摊消息
 - 广播消费（Broadcasting）：相同 Consumer Group 的每个 Consumer 实例都接收全量的消息

每个 Broker 可以存储多个 Topic 的消息，每个 Topic 的消息也可以分片存储于不同的 Broker，Message Queue（消息队列）是用于存储消息的物理地址，每个 Topic 中的消息地址存储于多个 Message Queue 中

- 主题（Topic）：表示一类消息的集合，每个主题包含若干条消息，每条消息只属于一个主题，是 RocketMQ 消息订阅的基本单位
- 消息（Message）：消息系统所传输信息的物理载体，生产和消费数据的最小单位，每条消息必须属于一个主题。RocketMQ 中每个消息拥有唯一的 Message ID，且可以携带具有业务标识的 Key，系统提供了通过 Message ID 和 Key 查询消息的功能

- 标签 (Tag)：为消息设置的标志，用于同一主题下区分不同类型的消息。标签能够有效地保持代码的清晰度和连贯性，并优化 RocketMQ 提供的查询系统，消费者可以根据 Tag 实现对不同子主题的不同消费逻辑，实现更好的扩展性
- 普通顺序消息 (Normal Ordered Message)：消费者通过同一个消息队列 (Topic 分区) 收到的消息是有顺序的，不同消息队列收到的消息则可能是无顺序的
- 严格顺序消息 (Strictly Ordered Message)：消费者收到的所有消息均是有顺序的

官方文档：<https://github.com/apache/rocketmq/tree/master/docs/cn> (基础知识部分的笔记参考官方文档编写)

消息操作

基本样例

订阅发布

消息的发布是指某个生产者向某个 Topic 发送消息，消息的订阅是指某个消费者关注了某个 Topic 中带有某些 Tag 的消息，进而从该 Topic 消费数据

导入 MQ 客户端依赖：

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.4.0</version>
</dependency>
```

消息发送者步骤分析：

1. 创建消息生产者 Producer，并制定生产者组名
2. 指定 Nameserver 地址
3. 启动 Producer
4. 创建消息对象，指定主题 Topic、Tag 和消息体
5. 发送消息
6. 关闭生产者 Producer

消息消费者步骤分析：

1. 创建消费者 Consumer，制定消费者组名
2. 指定 Nameserver 地址
3. 订阅主题 Topic 和 Tag
4. 设置回调函数，处理消息
5. 启动消费者 Consumer

发送消息

同步发送

使用 RocketMQ 发送三种类型的消息：同步消息、异步消息和单向消息，其中前两种消息是可靠的，因为会有发送是否成功的应答

这种可靠性同步地发送方式使用的比较广泛，比如：重要的消息通知，短信通知

```
public class SyncProducer {  
    public static void main(String[] args) throws Exception {  
        // 实例化消息生产者Producer  
        DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");  
        // 设置NameServer的地址  
        producer.setNamesrvAddr("localhost:9876");  
        // 启动Producer实例  
        producer.start();  
        for (int i = 0; i < 100; i++) {  
            // 创建消息，并指定Topic, Tag和消息体  
            Message msg = new Message()  
                "TopicTest" /* Topic */,  
                "TagA" /* Tag */,  
                ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET)  
            /* Message body */;  
  
            // 发送消息到一个Broker  
            SendResult sendResult = producer.send(msg);  
            // 通过sendResult返回消息是否成功送达  
            System.out.printf("%s%n", sendResult);  
        }  
        // 如果不再发送消息，关闭Producer实例。  
        producer.shutdown();  
    }  
}
```

异步发送

异步消息通常用在对响应时间敏感的业务场景，即发送端不能容忍长时间地等待 Broker 的响应

```
public class AsyncProducer {  
    public static void main(String[] args) throws Exception {  
        // 实例化消息生产者Producer
```

```

DefaultMQProducer producer = new
DefaultMQProducer("please_rename_unique_group_name");
    // 设置NameServer的地址
    producer.setNamesrvAddr("localhost:9876");
    // 启动Producer实例
    producer.start();
    producer.setRetryTimesWhenSendAsyncFailed(0);

    int messageCount = 100;
    // 根据消息数量实例化倒计时计算器
    final CountDownLatch2 countDownLatch = new
CountDownLatch2(messageCount);
    for (int i = 0; i < messageCount; i++) {
        final int index = i;
        // 创建消息，并指定Topic, Tag和消息体
        Message msg = new Message("TopicTest", "TagA", "OrderID188",
                                "Hello
world".getBytes(RemotingHelper.DEFAULT_CHARSET));

        // SendCallback接收异步返回结果的回调
        producer.send(msg, new SendCallback() {
            // 发送成功回调函数
            @Override
            public void onSuccess(SendResult sendResult) {
                countDownLatch.countDown();
                System.out.printf("%-10d OK %s %n", index,
sendResult.getMsgId());
            }

            @Override
            public void onException(Throwable e) {
                countDownLatch.countDown();
                System.out.printf("%-10d Exception %s %n", index, e);
                e.printStackTrace();
            }
        });
    }
    // 等待5s
    countDownLatch.await(5, TimeUnit.SECONDS);
    // 如果不再发送消息，关闭Producer实例。
    producer.shutdown();
}
}

```

单向发送

单向发送主要用在不特别关心发送结果的场景，例如日志发送

```

public class onewayProducer {
    public static void main(String[] args) throws Exception{

```

```

// 实例化消息生产者Producer
DefaultMQProducer producer = new
DefaultMQProducer("please_rename_unique_group_name");
// 设置NameServer的地址
producer.setNamesrvAddr("localhost:9876");
// 启动Producer实例
producer.start();
for (int i = 0; i < 100; i++) {
    // 创建消息，并指定Topic, Tag和消息体
    Message msg = new Message("TopicTest", "TagA",
        ("Hello RocketMQ " +
    i).getBytes(RemotingHelper.DEFAULT_CHARSET));
    // 发送单向消息，没有任何返回结果
    producer.sendOneWay(msg);
}
// 如果不再发送消息，关闭Producer实例。
producer.shutdown();
}
}

```

消费消息

```

public class Consumer {
    public static void main(String[] args) throws InterruptedException,
MQClientException {
        // 实例化消费者
        DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("please_rename_unique_group_name");
        // 设置NameServer的地址
        consumer.setNamesrvAddr("localhost:9876");

        // 订阅一个或者多个Topic，以及Tag来过滤需要消费的消息
        consumer.subscribe("TopicTest", "*");
        // 注册消息监听器，回调实现类来处理从broker拉取回来的消息
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            // 接受消息内容
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt>
msgs, ConsumeConcurrentlyContext context) {
                System.out.printf("%s Receive New Messages: %s %n",
Thread.currentThread().getName(), msgs);
                // 标记该消息已经被成功消费
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        // 启动消费者实例
        consumer.start();
        System.out.printf("Consumer Started.%n");
    }
}

```

顺序消息

原理解析

消息有序指的是一类消息消费时，能按照发送的顺序来消费。例如：一个订单产生了三条消息分别是订单创建、订单付款、订单完成。消费时要按照这个顺序消费才能有意义，但是同时订单之间是可以并行消费的，RocketMQ 可以严格的保证消息有序。

顺序消息分为全局顺序消息与分区顺序消息，

- 全局顺序：对于指定的一个 Topic，所有消息按照严格的先入先出（FIFO）的顺序进行发布和消费，适用于性能要求不高，所有的消息严格按照 FIFO 原则进行消息发布和消费的场景
- 分区顺序：对于指定的一个 Topic，所有消息根据 Sharding key 进行分区，同一个分组内的消息按照严格的 FIFO 顺序进行发布和消费。Sharding key 是顺序消息中用来区分不同分区的关键字段，和普通消息的 Key 是完全不同的概念，适用于性能要求高的场景

在默认的情况下消息发送会采取 Round Robin 轮询方式把消息发送到不同的 queue（分区队列），而消费消息是从多个 queue 上拉取消息，这种情况发送和消费是不能保证顺序。但是如果控制发送的顺序消息只依次发送到同一个 queue 中，消费的时候只从这个 queue 上依次拉取，则就保证了顺序。当发送和消费参与的 queue 只有一个，则是全局有序；如果多个queue 参与，则为分区有序，即相对每个 queue，消息都是有序的

代码实现

一个订单的顺序流程是：创建、付款、推送、完成，订单号相同的消息会被先后发送到同一个队列中，消费时同一个 OrderId 获取到的肯定是同一个队列

```
public class Producer {  
    public static void main(String[] args) throws Exception {  
        DefaultMQProducer producer = new  
DefaultMQProducer("please_rename_unique_group_name");  
        producer.setNamesrvAddr("127.0.0.1:9876");  
        producer.start();  
        // 标签集合  
        String[] tags = new String[]{"TagA", "TagC", "TagD"};  
  
        // 订单列表  
        List<OrderStep> orderList = new Producer().buildOrders();  
  
        Date date = new Date();  
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
        String dateStr = sdf.format(date);  
        for (int i = 0; i < 10; i++) {  
            // 加个时间前缀  
            producer.sendMessage(tags, dateStr + " " + i, "orderTopic");  
        }  
    }  
}
```

```
        String body = dateStr + " Hello RocketMQ " + orderList.get(i);
        Message msg = new Message("OrderTopic", tags[i % tags.length], "KEY"
+ i, body.getBytes());
        /**
         * 参数一：消息对象
         * 参数二：消息队列的选择器
         * 参数三：选择队列的业务标识（订单 ID）
         */
        SendResult sendResult = producer.send(msg, new
MessageQueueSelector() {
    @Override
    /**
     * mqS: 队列集合
     * msg: 消息对象
     * arg: 业务标识的参数
     */
    public MessageQueue select(List<MessageQueue> mqs, Message msg,
Object arg) {
        Long id = (Long) arg;
        long index = id % mqs.size(); // 根据订单id选择发送queue
        return mqs.get((int) index);
    }
}, orderList.get(i).getOrderId());//订单id

        System.out.println(String.format("SendResult status:%s, queueId:%d,
body:%s",
sendResult.getSendStatus(),
sendResult.getMessageQueue().getQueueId(),
body));
    }

    producer.shutdown();
}

// 订单的步骤
private static class OrderStep {
    private long orderId;
    private String desc;
    // set + get
}

// 生成模拟订单数据
private List<OrderStep> buildOrders() {
    List<OrderStep> orderList = new ArrayList<OrderStep>();

    OrderStep orderDemo = new OrderStep();
    orderDemo.setOrderId(15103111039L);
    orderDemo.setDesc("创建");
    orderList.add(orderDemo);

    orderDemo = new OrderStep();
    orderDemo.setOrderId(15103111065L);
    orderDemo.setDesc("创建");
    orderList.add(orderDemo);
}
```

```

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(151031117235L);
        orderDemo.setDesc("创建");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111065L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(151031117235L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111065L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("推送");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(151031117235L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(15103111039L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        return orderList;
    }
}

```

```

// 顺序消息消费，带事务方式（应用可控制offset什么时候提交）
public class ConsumerInOrder {
    public static void main(String[] args) throws Exception {
        DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("please_rename_unique_group_name_3");
        consumer.setNamesrvAddr("127.0.0.1:9876");
        // 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费
        // 如果非第一次启动，那么按照上次消费的位置继续消费

        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
        // 订阅三个tag
    }
}

```

```

        consumer.subscribe("OrderTopic", "TagA || TagC || TagD");
        consumer.registerMessageListener(new MessageListenerOrderly() {
            Random random = new Random();
            @Override
            public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs,
            ConsumeOrderlyContext context) {
                context.setAutoCommit(true);
                for (MessageExt msg : msgs) {
                    // 可以看到每个queue有唯一的consume线程来消费，订单对每个queue(分区)
                    有序
                    System.out.println("consumeThread=" +
                    Thread.currentThread().getName() + "queueId=" + msg.getQueueId() + ", content:"
                    + new String(msg.getBody()));
                }
                return ConsumeOrderlyStatus.SUCCESS;
            }
        });
        consumer.start();
        System.out.println("Consumer Started.");
    }
}

```

延时消息

原理解析

定时消息（延迟队列）是指消息发送到 Broker 后，不会立即被消费，等待特定时间投递给真正的 Topic。RocketMQ 并不支持任意时间的延时，需要设置几个固定的延时等级，从 1s 到 2h 分别对应着等级 1 到 18，消息消费失败会进入延时消息队列，消息发送时间与设置的延时等级和重试次数有关，详见代码 `SendMessageProcessor.java`

```

private String messageDelayLevel = "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m
20m 30m 1h 2h";

```

Broker 可以配置 `messageDelayLevel`，该属性是 Broker 的属性，不属于某个 Topic

发消息时，可以设置延迟等级 `msg.setDelayLevel(level)`，level 有以下三种情况：

- level == 0：消息为非延迟消息
- 1 <= level <= maxLevel：消息延迟特定时间，例如 level==1，延迟 1s
- level > maxLevel：则 level== maxLevel，例如 level==20，延迟 2h

定时消息会暂存在名为 SCHEDULE_TOPIC_XXXX 的 Topic 中，并根据 `delayTimeLevel` 存入特定的 queue，队列的标识 `queueId = delayTimeLevel - 1`，即一个 queue 只存相同延迟的消息，保证具有相同发送延迟的消息能够顺序消费。Broker 会为每个延迟级别提交一个定时任务，调度地消费 SCHEDULE_TOPIC_XXXX，将消息写入真实的 Topic

注意：定时消息在第一次写入和调度写入真实 Topic 时都会计数，因此发送数量、tps 都会变高

代码实现

提交了一个订单就可以发送一个延时消息，1h 后去检查这个订单的状态，如果还是未付款就取消订单释放库存

```
public class ScheduledMessageProducer {
    public static void main(String[] args) throws Exception {
        // 实例化一个生产者来产生延时消息
        DefaultMQProducer producer = new
DefaultMQProducer("ExampleProducerGroup");
        producer.setNamesrvAddr("127.0.0.1:9876");
        // 启动生产者
        producer.start();
        int totalMessagesToSend = 100;
        for (int i = 0; i < totalMessagesToSend; i++) {
            Message message = new Message("DelayTopic", ("Hello scheduled
message " + i).getBytes());
            // 设置延时等级3,这个消息将在10s之后发送(现在只支持固定的几个时间,详看
delayTimeLevel)
            message.setDelayTimeLevel(3);
            // 发送消息
            producer.send(message);
        }
        // 关闭生产者
        producer.shutdown();
    }
}
```

```
public class ScheduledMessageConsumer {
    public static void main(String[] args) throws Exception {
        // 实例化消费者
        DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("ExampleConsumer");
        consumer.setNamesrvAddr("127.0.0.1:9876");
        // 订阅Topics
        consumer.subscribe("DelayTopic", "*");
        // 注册消息监听者
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt>
messages, ConsumeConcurrentlyContext context) {
                for (MessageExt message : messages) {
                    // 打印延迟的时间段
                    System.out.println("Receive message[msgId=" +
message.getMsgId() + "] " + (System.currentTimeMillis() -
message.getBornTimestamp()) + "ms later");
                }
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        // 启动消费者
    }
}
```

```
    consumer.start();
}
}
```

批量消息

批量发送消息能显著提高传递小消息的性能，限制是这些批量消息应该有相同的 topic，相同的 waitStoreMsgOK，而且不能是延时消息，并且这一批消息的总大小不应超过 4MB

```
public class Producer {

    public static void main(String[] args) throws Exception {
        DefaultMQProducer producer = new
DefaultMQProducer("ExampleProducerGroup")
        producer.setNamesrvAddr("127.0.0.1:9876");
        //启动producer
        producer.start();

        List<Message> msgs = new ArrayList<Message>();
        // 创建消息对象，指定主题Topic、Tag和消息体
        Message msg1 = new Message("BatchTopic", "Tag1", ("Hello world" +
1).getBytes());
        Message msg2 = new Message("BatchTopic", "Tag1", ("Hello world" +
2).getBytes());
        Message msg3 = new Message("BatchTopic", "Tag1", ("Hello world" +
3).getBytes());

        msgs.add(msg1);
        msgs.add(msg2);
        msgs.add(msg3);

        // 发送消息
        SendResult result = producer.send(msgs);
        System.out.println("发送结果：" + result);
        // 关闭生产者producer
        producer.shutdown();
    }
}
```

当发送大批量数据时，可能不确定消息是否超过了大小限制（4MB），所以需要将消息列表分割一下

```
public class ListSplitter implements Iterator<List<Message>> {
    private final int SIZE_LIMIT = 1024 * 1024 * 4;
    private final List<Message> messages;
    private int currIndex;

    public ListSplitter(List<Message> messages) {
        this.messages = messages;
    }
```

```

@Override
public boolean hasNext() {
    return currIndex < messages.size();
}

@Override
public List<Message> next() {
    int startIndex = getstartIndex();
    int nextIndex = startIndex;
    int totalsize = 0;
    for (; nextIndex < messages.size(); nextIndex++) {
        Message message = messages.get(nextIndex);
        int tmpSize = calcMessagesize(message);
        // 单个消息超过了最大的限制
        if (tmpSize + totalsize > SIZE_LIMIT) {
            break;
        } else {
            totalsize += tmpSize;
        }
    }
    List<Message> subList = messages.subList(startIndex, nextIndex);
    currIndex = nextIndex;
    return subList;
}

private int getstartIndex() {
    Message currMessage = messages.get(currIndex);
    int tmpsize = calcMessagesize(currMessage);
    while (tmpsize > SIZE_LIMIT) {
        currIndex += 1;
        Message message = messages.get(curIndex);
        tmpsize = calcMessagesize(message);
    }
    return currIndex;
}

private int calcMessagesize(Message message) {
    int tmpsize = message.getTopic().length() + message.getBody().length;
    Map<String, String> properties = message.getProperties();
    for (Map.Entry<String, String> entry : properties.entrySet()) {
        tmpsize += entry.getKey().length() + entry.getValue().length();
    }
    tmpsize = tmpsize + 20; // 增加日志的开销20字节
    return tmpsize;
}

public static void main(String[] args) {
    //把大的消息分裂成若干个小的消息
    Listsplitter splitter = new Listsplitter(messages);
    while (splitter.hasNext()) {
        try {
            List<Message> listItem = splitter.next();
            producer.send(listItem);
        } catch (Exception e) {

```

```
        e.printStackTrace();
        //处理error
    }
}
}
```

过滤消息

基本语法

RocketMQ 定义了一些基本语法来支持过滤特性，可以很容易地扩展：

- 数值比较，比如：>, >=, <, <=, BETWEEN, =
- 字符比较，比如：=, <>, IN
- IS NULL 或者 IS NOT NULL
- 逻辑符号 AND, OR, NOT

常量支持类型为：

- 数值，比如 123, 3.1415
- 字符，比如 'abc'，必须用单引号包裹起来
- NULL，特殊的常量
- 布尔值，TRUE 或 FALSE

只有使用 push 模式的消费者才能用使用 SQL92 标准的 sql 语句，接口如下：

```
public void subscribe(final String topic, final MessageSelector messageSelector)
```

例如：消费者接收包含 TAGA 或 TAGB 或 TAGC 的消息

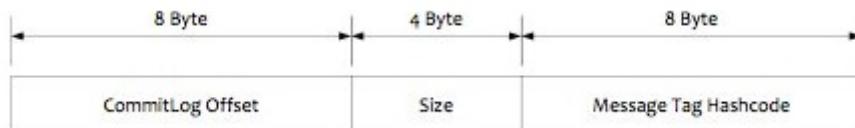
```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("CID_EXAMPLE");
consumer.subscribe("TOPIC", "TAGA || TAGB || TAGC");
```

原理解析

RocketMQ 分布式消息队列的消息过滤方式是在 Consumer 端订阅消息时再做消息过滤的，所以是在 Broker 端实现的，优点是减少了对于 Consumer 无用消息的网络传输，缺点是增加了 Broker 的负担，而且实现相对复杂

RocketMQ 在 Producer 端写入消息和在 Consumer 端订阅消息采用分离存储的机制实现，Consumer 端订阅消息是需要通过 ConsumeQueue 这个消息消费的逻辑队列拿到一个索引，然后再从 CommitLog 里面读取真正的消息实体内容

ConsumeQueue 的存储结构如下，有 8 个字节存储的 Message Tag 的哈希值，基于 Tag 的消息过滤就是基于这个字段



- **Tag 过滤:** Consumer 端订阅消息时指定 Topic 和 TAG，然后将订阅请求构建成一个 SubscriptionData，发送一个 Pull 消息的请求给 Broker 端。Broker 端用这些数据先构建一个 MessageFilter，然后传给文件存储层 Store。Store 从 ConsumeQueue 读取到一条记录后，会用它记录的消息 tag hash 值去做过滤。因为在服务端只是根据 hashCode 进行判断，无法精确对 tag 原始字符串进行过滤，所以消费端拉取到消息后，还需要对消息的原始 tag 字符串进行比对，如果不同，则丢弃该消息，不进行消息消费
- **SQL92 过滤:** 工作流程和 Tag 过滤大致一样，只是在 Store 层的具体过滤方式不一样。真正的 SQL expression 的构建和执行由 rocketmq-filter 模块负责，每次过滤都去执行 SQL 表达式会影响效率，所以 RocketMQ 使用了 BloomFilter 来避免了每次都去执行

代码实现

发送消息时，通过 putUserProperty 来设置消息的属性，SQL92 的表达式上下文为消息的属性

```
public class Producer {
    public static void main(String[] args) throws Exception {
        DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");
        producer.setNamesrvAddr("127.0.0.1:9876");
        producer.start();
        for (int i = 0; i < 10; i++) {
            Message msg = new Message("FilterTopic", "tag",
                ("Hello RocketMQ " +
                i).getBytes(RemotingHelper.DEFAULT_CHARSET));
            // 设置一些属性
            msg.putUserProperty("i", String.valueOf(i));
            SendResult sendResult = producer.send(msg);
        }
        producer.shutdown();
    }
}
```

使用 SQL 筛选过滤消息：

```
public class Consumer {
    public static void main(String[] args) throws Exception {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_group_name");
        consumer.setNamesrvAddr("127.0.0.1:9876");
        // 过滤属性大于 5 的消息
    }
}
```

```

consumer.subscribe("FilterTopic", MessageSelector.bySql("i>5"));

// 设置回调函数，处理消息
consumer.registerMessageListener(new MessageListenerConcurrently() {
    //接受消息内容
    @Override
    public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt>
msgs, ConsumeConcurrentlyContext context) {
        for (MessageExt msg : msgs) {
            System.out.println("consumeThread=" +
Thread.currentThread().getName() + "," + new String(msg.getBody()));
        }
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
});

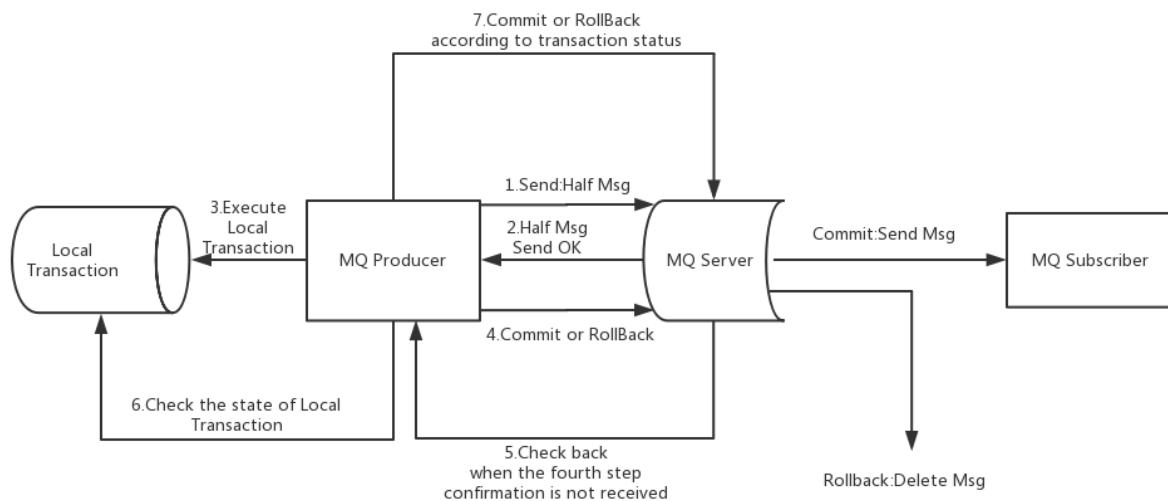
// 启动消费者consumer
consumer.start();
}
}

```

事务消息

工作流程

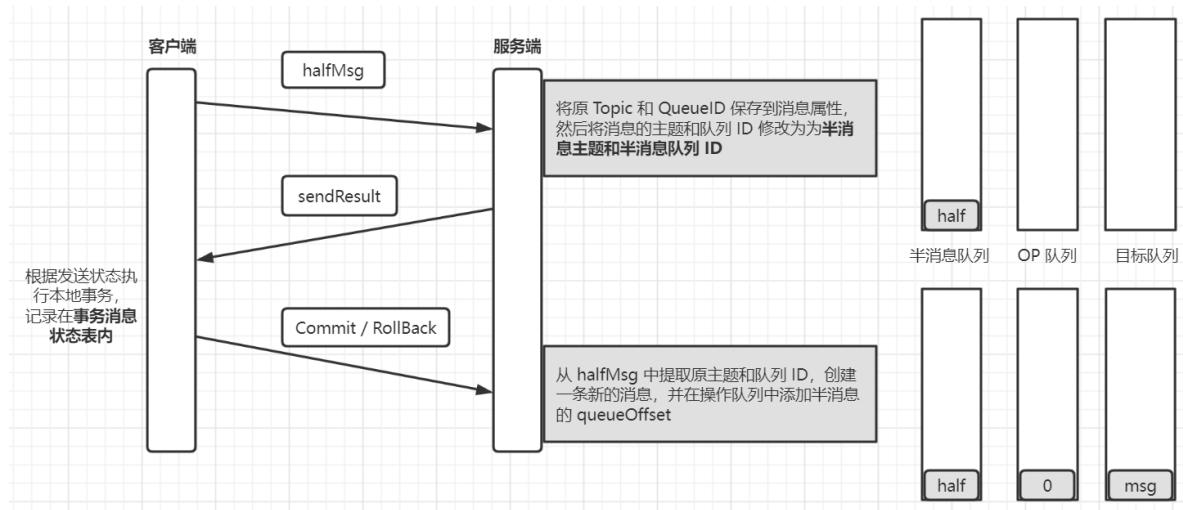
RocketMQ 支持分布式事务消息，采用了 2PC 的思想来实现了提交事务消息，同时增加一个**补偿逻辑**来处理二阶段超时或者失败的消息，如下图所示：



事务消息的大致方案分为两个流程：正常事务消息的发送及提交、事务消息的补偿流程

1. 事务消息发送及提交：

- 发送消息（Half 消息），服务器将消息的主题和队列改为半消息状态，并放入半消息队列
- 服务端响应消息写入结果（如果写入失败，此时 Half 消息对业务不可见）
- 根据发送结果执行本地事务
- 根据本地事务状态执行 Commit 或者 Rollback



2. 补偿机制：用于解决消息 Commit 或者 Rollback 发生超时或者失败的情况，比如出现网络问题

- Broker 服务端通过**对比 Half 消息和 Op 消息**，对未确定状态的消息推进 CheckPoint
- 没有 Commit/Rollback 的事务消息，服务端根据根据半消息的生产者组，到 ProducerManager 中获取生产者（同一个 Group 的 Producer）的会话通道，发起一次回查（**单向请求**）
- Producer 收到回查消息，检查事务消息状态表内对应的本地事务的状态
- 根据本地事务状态，重新 Commit 或者 Rollback

RocketMQ 并不会无休止的进行事务状态回查，最大回查 15 次，如果 15 次回查还是无法得知事务状态，则默认回滚该消息，

回查服务：`TransactionalMessageCheckService#run`

两阶段

一阶段

事务消息相对普通消息最大的特点就是**一阶段发送的消息对用户是不可见的**，因为对于 Half 消息，会备份原消息的主题与消息消费队列，然后改变主题为 RMQ_SYS_TRANS_HALF_TOPIC，由于消费组未订阅该主题，故消费端无法消费 Half 类型的消息

RocketMQ 会开启一个**定时任务**，从 Topic 为 RMQ_SYS_TRANS_HALF_TOPIC 中拉取消息进行消费，根据生产者组获取一个服务提供者发送回查事务状态请求，根据事务状态来决定是提交或回滚消息

RocketMQ 的具体实现策略：如果写入的是事务消息，对消息的 Topic 和 Queue 等属性进行替换，同时将原来的 Topic 和 Queue 信息存储到**消息的属性**中，因为消息的主题被替换，所以消息不会转发到该原主题的消息消费队列，消费者无法感知消息的存在，不会消费

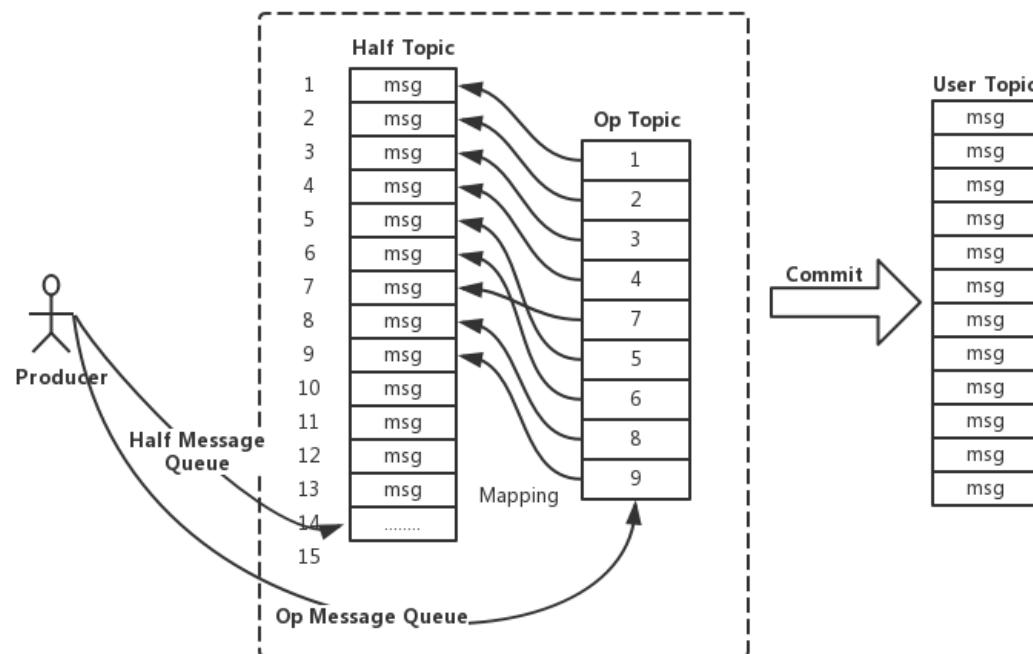
二阶段

一阶段写入不可见的消息后，二阶段操作：

- 如果执行 Commit 操作，则需要让消息对用户可见，构建出 Half 消息的索引。一阶段的 Half 消息写到一个特殊的 Topic，构建索引时需要读取出 Half 消息，然后通过一次普通消息的写入操作将 Topic 和 Queue 替换成真正的目标 Topic 和 Queue，生成一条对用户可见的消息。其实就是利用了一阶段存储的消息的内容，在二阶段时恢复出一条完整的普通消息，然后走一遍消息写入流程
- 如果是 Rollback 则需要撤销一阶段的消息，因为消息本就不可见，所以并**不需要真正撤销消息**（实际上 RocketMQ 也无法去删除一条消息，因为是顺序写文件的）。RocketMQ 为了区分这条消息没有确定状态的消息，采用 Op 消息标识已经确定状态的事务消息（Commit 或者 Rollback）

事务消息无论是 Commit 或者 Rollback 都会记录一个 Op 操作，两者的区别是 Commit 相对于 Rollback 在写入 Op 消息前将原消息的主题和队列恢复。如果一条事务消息没有对应的 Op 消息，说明这个事务的状态还无法确定（可能是二阶段失败了）

RocketMQ 将 Op 消息写入到全局一个特定的 Topic 中，通过源码中的方法 `TransactionalMessageUtil.buildOpTopic()`，这个主题是一个内部的 Topic（像 Half 消息的 Topic 一样），不会被用户消费。Op 消息的内容为对应的 Half 消息的存储的 Offset，这样**通过 Op 消息能索引到 Half 消息**



基本使用

使用方式

事务消息共有三种状态，提交状态、回滚状态、中间状态：

- `TransactionStatus.CommitTransaction`: 提交事务，允许消费者消费此消息。
- `TransactionStatus.RollbackTransaction`: 回滚事务，代表该消息将被删除，不允许被消费
- `TransactionStatus.Unknown`: 中间状态，代表需要检查消息队列来确定状态

使用限制：

1. 事务消息不支持延时消息和批量消息
 2. Broker 配置文件中的参数 `transactionTimeout` 为特定时间，事务消息将在特定时间长度之后被检查。当发送事务消息时，还可以通过设置用户属性 `CHECK_IMMUNITY_TIME_IN_SECONDS` 来改变这个限制，该参数优先于 `transactionTimeout` 参数
 3. 为了避免单个消息被检查太多次而导致半队列消息累积，默认将单个消息的检查次数限制为 15 次，开发者可以通过 Broker 配置文件的 `transactionCheckMax` 参数来修改此限制。如果已经检查某条消息超过 N 次 ($N = \text{transactionCheckMax}$)，则 Broker 将丢弃此消息，在默认情况下会打印错误日志。可以通过重写 `AbstractTransactionalMessageCheckListener` 类来修改这个行为
 4. 事务性消息可能不止一次被检查或消费
 5. 提交给用户的目标主题消息可能会失败，可以查看日志的记录。事务的高可用性通过 RocketMQ 本身的高可用性机制来保证，如果希望事务消息不丢失、并且事务完整性得到保证，可以使用同步的双重写入机制
 6. 事务消息的生产者 ID 不能与其他类型消息的生产者 ID 共享。与其他类型的消息不同，事务消息允许反向查询，MQ 服务器能通过消息的生产者 ID 查询到消费者
-

代码实现

实现事务的监听接口，当发送半消息成功时：

- `executeLocalTransaction` 方法来执行本地事务，返回三个事务状态之一
- `checkLocalTransaction` 方法检查本地事务状态，响应消息队列的检查请求，返回三个事务状态之一

```
public class TransactionListenerImpl implements TransactionListener {  
    private AtomicInteger transactionIndex = new AtomicInteger(0);  
    private ConcurrentHashMap<String, Integer> localTrans = new  
ConcurrentHashMap<>();  
  
    @Override  
    public LocalTransactionState executeLocalTransaction(Message msg, Object  
arg) {  
        int value = transactionIndex.getAndIncrement();  
        int status = value % 3;  
        // 将事务ID和状态存入 map 集合  
        localTrans.put(msg.getTransactionId(), status);  
        return LocalTransactionState.UNKNOW;  
    }  
  
    @Override  
    public LocalTransactionState checkLocalTransaction(MessageExt msg) {  
        // 从 map 集合读出当前事务对应的状态  
        Integer status = localTrans.get(msg.getTransactionId());  
        if (null != status) {  
            switch (status) {  
                case 0:  
                    return LocalTransactionState.UNKNOW;  
                case 1:  
                    return LocalTransactionState.COMMIT_MESSAGE;  
            }  
        }  
        return LocalTransactionState.RETRY;  
    }  
}
```

```

        case 2:
            return LocalTransactionState.ROLLBACK_MESSAGE;
        }
    }
    return LocalTransactionState.COMMIT_MESSAGE;
}
}

```

使用 **TransactionMQProducer** 类创建事务性生产者，并指定唯一的 `ProducerGroup`，就可以设置自定义线程池来处理这些检查请求，执行本地事务后，需要根据执行结果对消息队列进行回复

```

public class Producer {
    public static void main(String[] args) throws MQClientException,
InterruptedException {
    // 创建消息生产者
    TransactionMQProducer producer = new
    TransactionMQProducer("please_rename_unique_group_name");
    ExecutorService executorService = new ThreadPoolExecutor(2, 5, 100,
TimeUnit.SECONDS);
    producer.setExecutorService(executorService);

    // 创建事务监听器
    TransactionListener transactionListener = new TransactionListenerImpl();
    // 生产者的监听器
    producer.setTransactionListener(transactionListener);
    // 启动生产者
    producer.start();
    String[] tags = new String[] {"TagA", "TagB", "TagC", "TagD", "TagE"};
    for (int i = 0; i < 10; i++) {
        try {
            Message msg = new Message("TransactionTopic", tags[i %
tags.length], "KEY" + i,
                ("Hello RocketMQ " +
i).getBytes(RemotingHelper.DEFAULT_CHARSET));
            // 发送消息
            SendResult sendResult = producer.sendMessageInTransaction(msg,
null);
            System.out.printf("%s%n", sendResult);
            Thread.sleep(10);
        } catch (MQClientException | UnsupportedEncodingException e) {
            e.printStackTrace();
        }
    }
    //Thread.sleep(1000000);
    //producer.shutdown();暂时不关闭
}
}

```

消费者代码和前面的实例相同的

系统特性

工作流程

模块介绍

NameServer 是一个简单的 Topic 路由注册中心，支持 Broker 的动态注册与发现，生产者或消费者能够通过名字服务查找各主题相应的 Broker IP 列表

NameServer 主要包括两个功能：

- Broker 管理，NameServer 接受 Broker 集群的注册信息，保存下来作为路由信息的基本数据，提供**心跳检测机制**检查 Broker 是否还存活，每 10 秒清除一次两小时没有活跃的 Broker
- 路由信息管理，每个 NameServer 将保存关于 Broker 集群的整个路由信息和用于客户端查询的队列信息，然后 Producer 和 Consumer 通过 NameServer 就可以知道整个 Broker 集群的路由信息，从而进行消息的投递和消费

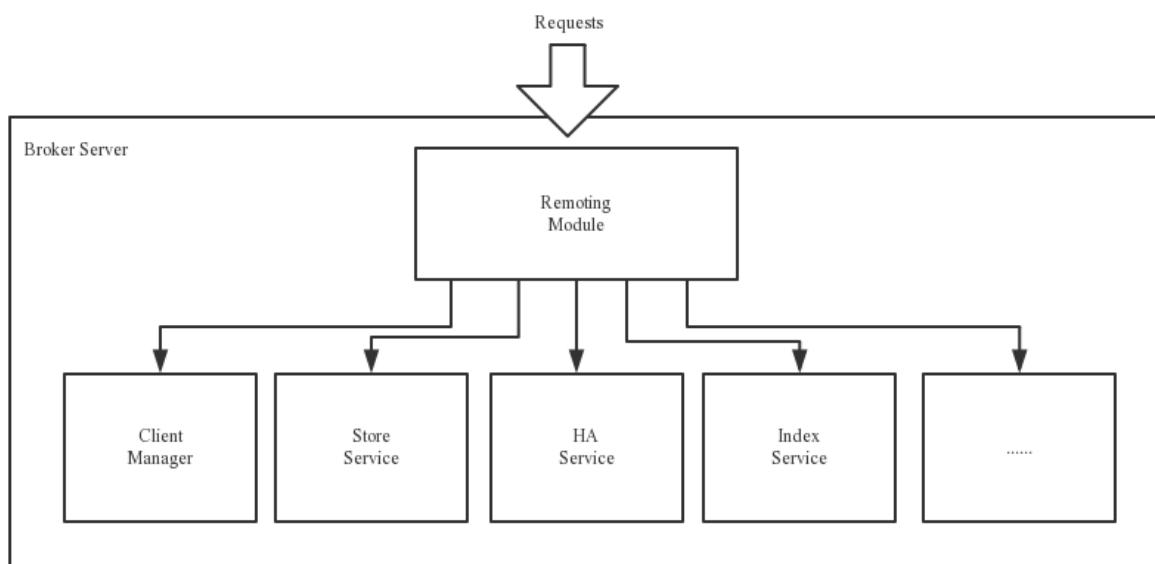
NameServer 特点：

- NameServer 通常是集群的方式部署，**各实例间相互不进行信息通讯**
- Broker 向每一台 NameServer（集群）注册自己的路由信息，所以每个 NameServer 实例上面都**保存一份完整的路由信息**
- 当某个 NameServer 因某种原因下线了，Broker 仍可以向其它 NameServer 同步其路由信息

BrokerServer 主要负责消息的存储、投递和查询以及服务高可用保证，在 RocketMQ 系统中接收从生产者发送来的消息并存储、同时为消费者的拉取请求作准备，也存储消息相关的元数据，包括消费者组、消费进度偏移和主题和队列消息等

Broker 包含了以下几个重要子模块：

- Remoting Module：整个 Broker 的实体，负责处理来自 Clients 端的请求
- Client Manager：负责管理客户端（Producer/Consumer）和维护 Consumer 的 Topic 订阅信息
- Store Service：提供方便简单的 API 接口处理消息存储到物理硬盘和查询功能
- HA Service：高可用服务，提供 Master Broker 和 Slave Broker 之间的数据同步功能
- Index Service：根据特定的 Message key 对投递到 Broker 的消息进行索引服务，以提供消息的快速查询



总体流程

RocketMQ 的工作流程：

- 启动 NameServer 监听端口，等待 Broker、Producer、Consumer 连上来，相当于一个路由控制中心
- Broker 启动，跟**所有的 NameServer 保持长连接**，每隔 30s 时间向 NameServer 上报 Topic 路由信息（心跳包）。心跳包中包含当前 Broker 信息（IP、端口等）以及存储所有 Topic 信息。注册成功后，NameServer 集群中就有 Topic 跟 Broker 的映射关系
- 收发消息前，先创建 Topic，创建 Topic 时需要指定该 Topic 要存储在哪些 Broker 上，也可以在发送消息时自动创建 Topic
- Producer 启动时先跟 NameServer 集群中的**其中一台**建立长连接，并从 NameServer 中获取当前发送的 Topic 存在哪些 Broker 上，同时 Producer 会默认每隔 30s 向 NameServer **定时拉取**一次路由信息
- Producer 发送消息时，根据消息的 Topic 从本地缓存的 TopicPublishInfoTable 获取路由信息，如果没有则会从 NameServer 上重新拉取并更新，轮询队列列表并选择一个队列 MessageQueue，然后与队列所在的 Broker 建立长连接，向 Broker 发消息
- Consumer 跟 Producer 类似，跟其中一台 NameServer 建立长连接，**定时获取路由信息**，根据当前订阅 Topic 存在哪些 Broker 上，直接跟 Broker 建立连接通道，在完成客户端的负载均衡后，选择其中的某一个或者某几个 MessageQueue 来拉取消息并进行消费

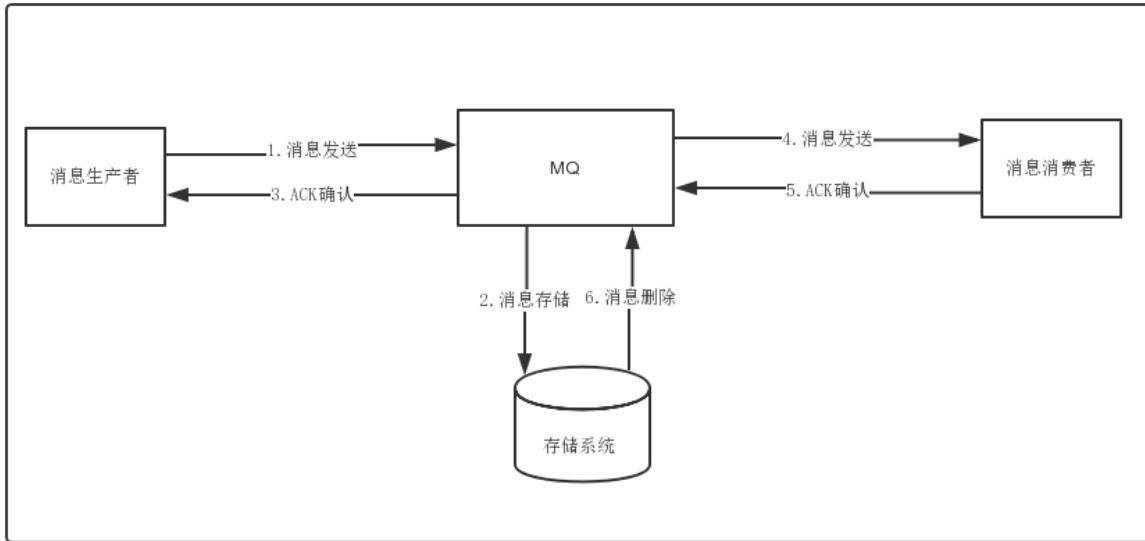
生产消费

At least Once：至少一次，指每个消息必须投递一次，Consumer 先 Pull 消息到本地，消费完成后才向服务器返回 ACK，如果没有消费一定不会 ACK 消息

回溯消费：指 Consumer 已经消费成功的消息，由于业务上需求需要重新消费，Broker 在向 Consumer 投递成功消息后，消息仍然需要保留。并且重新消费一般是按照时间维度，例如由于 Consumer 系统故障，恢复后需要重新消费 1 小时前的数据，RocketMQ 支持按照时间回溯消费，时间维度精确到毫秒

分布式队列因为有高可靠性的要求，所以数据要进行**持久化存储**

1. 消息生产者发送消息
2. MQ 收到消息，将消息进行持久化，在存储中新增一条记录
3. 返回 ACK 给生产者
4. MQ push 消息给对应的消费者，然后等待消费者返回 ACK
5. 如果消息消费者在指定时间内成功返回 ACK，那么 MQ 认为消息消费成功，在存储中删除消息；如果 MQ 在指定时间内没有收到 ACK，则认为消息消费失败，会尝试重新 push 消息，重复执行 4、5、6 步骤
6. MQ 删除消息



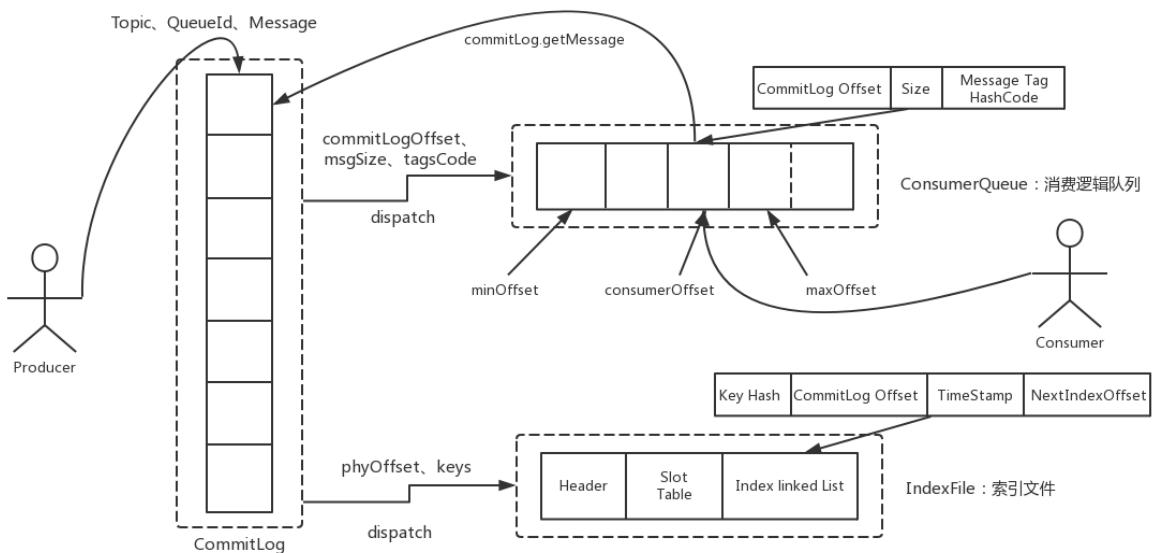
存储机制

存储结构

RocketMQ 中 Broker 负责存储消息转发消息，所以以下的结构是存储在 Broker Server 上的，生产者和消费者与 Broker 进行消息的收发是通过主题对应的 Message Queue 完成，类似于通道

RocketMQ 消息的存储是由 ConsumeQueue 和 CommitLog 配合完成的，CommitLog 是消息真正的**物理存储**文件，ConsumeQueue 是消息的逻辑队列，类似数据库的**索引节点**，存储的是指向物理存储的地址。**每个 Topic 下的每个 Message Queue 都有一个对应的 ConsumeQueue 文件**

每条消息都会有对应的索引信息，Consumer 通过 ConsumeQueue 这个结构来读取消息实体内容



- CommitLog：消息主体以及元数据的存储主体，存储 Producer 端写入的消息内容，消息内容不是定长的。消息主要是**顺序写入**日志文件，单个文件大小默认 1G，偏移量代表下一次写入的位置，当文件写满了就继续写入下一个文件

- ConsumerQueue：消息消费队列，存储消息在 CommitLog 的索引。RocketMQ 消息消费时要遍历 CommitLog 文件，并根据主题 Topic 检索消息，这是非常低效的。引入 ConsumeQueue 作为消费消息的索引，**保存了指定 Topic 下的队列消息在 CommitLog 中的起始物理偏移量 offset**，消息大小 size 和消息 Tag 的HashCode 值，每个 ConsumeQueue 文件大小约 5.72M
- IndexFile：为了消息查询提供了一种通过 Key 或时间区间来查询消息的方法，通过 IndexFile 来查找消息的方法**不影响发送与消费消息的主流程**。IndexFile 的底层存储为在文件系统中实现的 HashMap 结构，故 RocketMQ 的索引文件其底层实现为 **hash 索引**

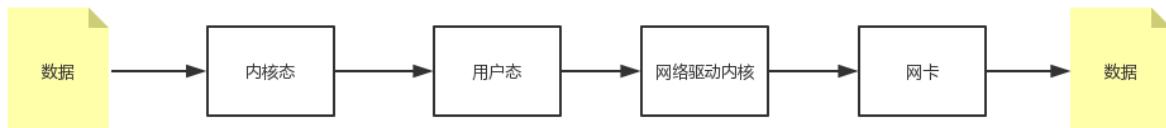
RocketMQ 采用的是混合型的存储结构，即为 Broker 单个实例下所有的队列共用一个日志数据文件（CommitLog）来存储，多个 Topic 的消息实体内容都存储于一个 CommitLog 中。混合型存储结构针对 Producer 和 Consumer 分别采用了**数据和索引部分相分离**的存储结构，Producer 发送消息至 Broker 端，然后 Broker 端使用同步或者异步的方式对消息刷盘持久化，保存至 CommitLog 中。只要消息被持久化至磁盘文件 CommitLog 中，Producer 发送的消息就不会丢失，Consumer 也就肯定有机会去消费这条消息

服务端支持长轮询模式，当消费者无法拉取到消息后，可以等下一次消息拉取，Broker 允许等待 30s 的时间，只要这段时间内有新消息到达，将直接返回给消费端。RocketMQ 的具体做法是，使用 Broker 端的后台服务线程 ReputMessageService 不停地分发请求并异步构建 ConsumeQueue（逻辑消费队列）和 IndexFile（索引文件）数据

内存映射

操作系统分为用户态和内核态，文件操作、网络操作需要涉及这两种形态的切换，需要进行数据复制。一台服务器把本机磁盘文件的内容发送到客户端，分为两个步骤：

- read：读取本地文件内容
- write：将读取的内容通过网络发送出去



补充：Prog → NET → I/O → 零拷贝部分的笔记详解相关内容

通过使用 mmap 的方式，可以省去向用户态的内存复制，RocketMQ 充分利用**零拷贝技术**，提高消息存盘和网络发送的速度

RocketMQ 通过 MappedByteBuffer 对文件进行读写操作，利用了 NIO 中的 FileChannel 模型将磁盘上的物理文件直接映射到用户态的内存地址中，将对文件的操作转化为直接对内存地址进行操作，从而极大地提高了文件的读写效率

MappedByteBuffer 内存映射的方式**限制**一次只能映射 1.5~2G 的文件至用户态的虚拟内存，所以 RocketMQ 默认设置单个 CommitLog 日志数据文件为 1G。RocketMQ 的文件存储使用定长结构来存储，方便一次将整个文件映射至内存

页面缓存

页缓存（PageCache）是 OS 对文件的缓存，每一页的大小通常是 4K，用于加速对文件的读写。因为 OS 将一部分的内存用作 PageCache，所以程序对文件进行顺序读写的速度几乎接近于内存的读写速度

- 对于数据的写入，OS 会先写入至 Cache 内，随后通过异步的方式由 **pdflush** 内核线程将 Cache 内的数据刷盘至物理磁盘上
- 对于数据的读取，如果一次读取文件时出现未命中 PageCache 的情况，OS 从物理磁盘上访问读取文件的同时，会顺序对其他相邻块的数据文件进行**预读取**（局部性原理，最大 128K）

在 RocketMQ 中，ConsumeQueue 逻辑消费队列存储的数据较少，并且是顺序读取，在 PageCache 机制的预读取作用下，Consume Queue 文件的读性能几乎接近读内存，即使在有消息堆积情况下也不会影响性能。但是 CommitLog 消息存储的日志数据文件读取内容时会产生较多的随机访问读取，严重影响性能。选择合适的系统 IO 调度算法和固态硬盘，比如设置调度算法为 Deadline，随机读的性能也会有所提升

刷盘机制

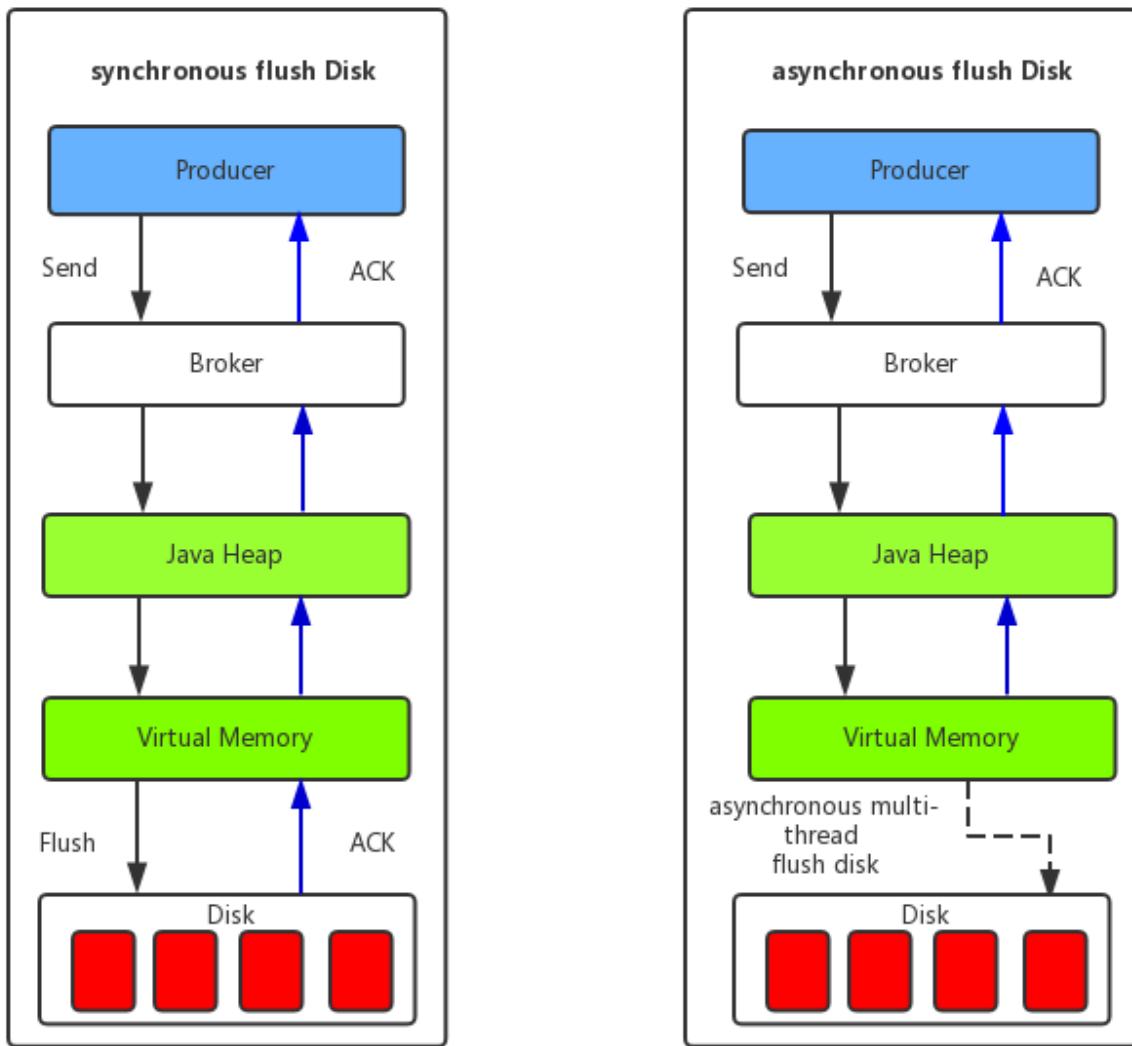
两种持久化的方案：

- 关系型数据库 DB：IO 读写性能比较差，如果 DB 出现故障，则 MQ 的消息就无法落盘存储导致线上故障，可靠性不高
- 文件系统：消息刷盘至所部署虚拟机/物理机的文件系统来做持久化，分为异步刷盘和同步刷盘两种模式。消息刷盘为消息存储提供了一种高效率、高可靠性和高性能的数据持久化方式，除非部署 MQ 机器本身或是本地磁盘挂了，一般不会出现无法持久化的问题

RocketMQ 采用文件系统的方式，无论同步还是异步刷盘，都使用**顺序 IO**，因为磁盘的顺序读写要比随机读写快很多

- 同步刷盘：只有在消息真正持久化至磁盘后 RocketMQ 的 Broker 端才会真正返回给 Producer 端一个成功的 ACK 响应，保障 MQ 消息的可靠性，但是性能上会有较大影响，一般适用于金融业务应用该模式较多
- 异步刷盘：利用 OS 的 PageCache，只要消息写入内存 PageCache 即可将成功的 ACK 返回给 Producer 端，降低了读写延迟，提高了 MQ 的性能和吞吐量。消息刷盘采用**后台异步线程**提交的方式进行，当内存里的消息量积累到一定程度时，触发写磁盘动作

通过 Broker 配置文件里的 flushDiskType 参数设置采用什么方式，可以配置成 SYNC_FLUSH、ASYNC_FLUSH 中的一个



官方文档: <https://github.com/apache/rocketmq/blob/master/docs/cn/design.md>

集群设计

集群模式

常用的以下几种模式：

- 单 Master 模式：这种方式风险较大，一旦 Broker 重启或者宕机，会导致整个服务不可用
- 多 Master 模式：一个集群无 Slave，全是 Master
 - 优点：配置简单，单个 Master 宕机或重启维护对应用无影响，在磁盘配置为 RAID10 时，即使机器宕机不可恢复情况下，由于 RAID10 磁盘非常可靠，消息也不会丢（异步刷盘丢失少量消息，同步刷盘一条不丢），性能最高
 - 缺点：单台机器宕机期间，这台机器上未被消费的消息在机器恢复之前不可订阅，消息实时性会受到影响
- 多 Master 多 Slave 模式（同步）：每个 Master 配置一个 Slave，有多对 Master-Slave，HA 采用 **同步双写**方式，即只有主备都写成功，才向应用返回成功

- 优点：数据与服务都无单点故障，Master 崩机情况下，消息无延迟，服务可用性与数据可用性都非常高
 - 缺点：性能比异步复制略低（大约低 10% 左右），发送单个消息的 RT 略高，目前不能实现主节点宕机，备机自动切换为主机
 - 多 Master 多 Slave 模式（异步）：HA 采用异步复制的方式，会造成主备有短暂的消息延迟（毫秒级别）
 - 优点：即使磁盘损坏，消息丢失的非常少，且消息实时性不会受影响，同时 Master 崩机后，消费者仍然可以从 Slave 消费，而且此过程对应用透明，不需要人工干预，性能同多 Master 模式几乎一样
 - 缺点：Master 崩机，磁盘损坏情况下会丢失少量消息
-

集群架构

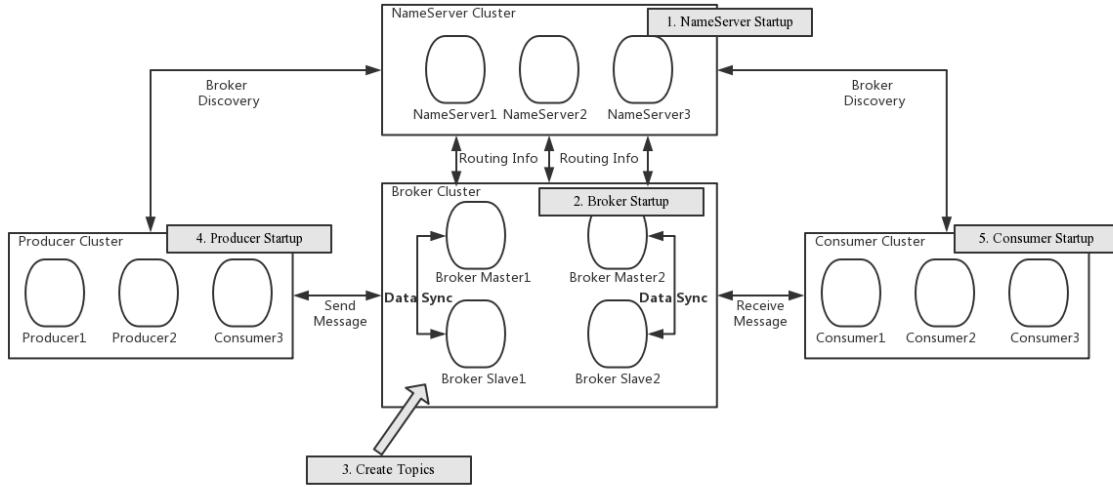
RocketMQ 网络部署特点：

- NameServer 是一个几乎**无状态节点**，节点之间相互独立，无任何信息同步
- Broker 部署相对复杂，Broker 分为 Master 与 Slave，Master 可以部署多个，一个 Master 可以对应多个 Slave，但是一个 Slave 只能对应一个 Master，Master 与 Slave 的对应关系通过指定相同 BrokerName、不同 BrokerId 来定义，BrokerId 为 0 是 Master，非 0 表示 Slave。**每个 Broker 与 NameServer 集群中的所有节点建立长连接**，定时注册 Topic 信息到所有 NameServer

说明：部署架构上也支持一 Master 多 Slave，但只有 BrokerId=1 的从服务器才会参与消息的读负载（读写分离）

- Producer 与 NameServer 集群中的其中一个节点（随机选择）建立长连接，定期从 NameServer 获取 Topic 路由信息，并向提供 Topic 服务的 Master 建立长连接，且定时向 Master **发送心跳**。Producer 完全无状态，可集群部署
- Consumer 与 NameServer 集群中的其中一个节点（随机选择）建立长连接，定期从 NameServer 获取 Topic 路由信息，并向提供 Topic 服务的 Master、Slave 建立长连接，且定时向 Master、Slave 发送心跳

Consumer 既可以从 Master 订阅消息，也可以从 Slave 订阅消息，在向 Master 拉取消息时，Master 服务器会根据拉取偏移量与最大偏移量的距离（判断是否读老消息，产生读 I/O），以及从服务器是否可读等因素建议下一次是从 Master 还是 Slave 拉取



官方文档: <https://github.com/apache/rocketmq/blob/master/docs/cn/architecture.md>

高可用性

NameServer 节点是无状态的，且各个节点直接的数据是一致的，部分 NameServer 不可用也可以保证 MQ 服务正常运行

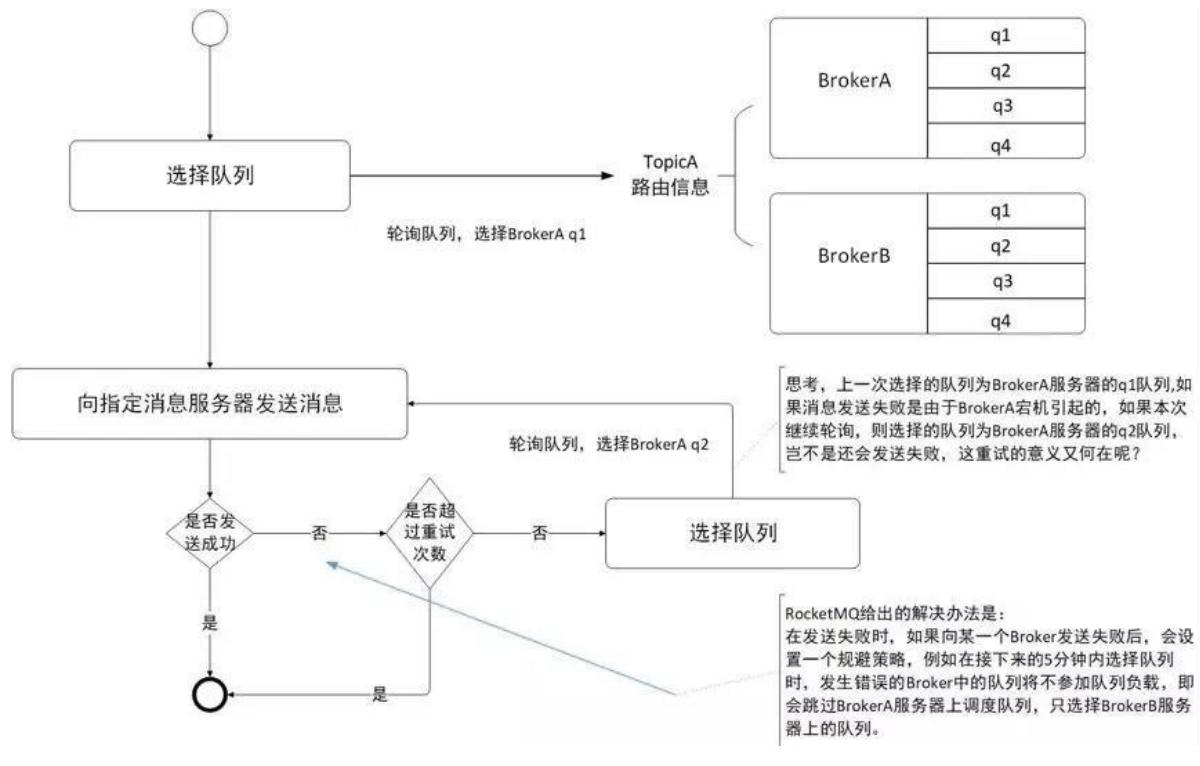
BrokerServer 的高可用通过 Master 和 Slave 的配合：

- Slave 只负责读，当 Master 不可用，对应的 Slave 仍能保证消息被正常消费
- 配置多组 Master-Slave 组，其他的 Master-Slave 组也会保证消息的正常发送和消费
- **目前不支持把 Slave 自动转成 Master**，需要手动停止 Slave 角色的 Broker，更改配置文件，用新的配置文件启动 Broker

所以需要配置多个 Master 保证可用性，否则一个 Master 挂了导致整体系统的写操作不可用

生产端的高可用：在创建 Topic 的时候，把 Topic 的**多个 Message Queue 创建在多个 Broker 组上**（相同 Broker 名称，不同 brokerId 的机器），当一个 Broker 组的 Master 不可用后，其他组的 Master 仍然可用，Producer 仍然可以发送消息

消费端的高可用：在 Consumer 的配置文件中，并不需要设置是从 Master Broker 读还是从 Slave 读，当 Master 不可用或者繁忙的时候，Consumer 会被自动切换到从 Slave 读。有了自动切换的机制，当一个 Master 机器出现故障后，Consumer 仍然可以从 Slave 读取消息，不影响 Consumer 程序，达到了消费端的高可用性



主从复制

如果一个 Broker 组有 Master 和 Slave, 消息需要从 Master 复制到 Slave 上, 有同步和异步两种复制方式:

- 同步复制方式: Master 和 Slave 均写成功后才反馈给客户端写成功状态 (写 Page Cache)。在同步复制方式下, 如果 Master 出故障, Slave 上有全部的备份数据, 容易恢复, 但是同步复制会增大数据写入延迟, 降低系统吞吐量
- 异步复制方式: 只要 Master 写成功, 即可反馈给客户端写成功状态, 系统拥有较低的延迟和较高的吞吐量, 但是如果 Master 出了故障, 有些数据因为没有被写入 Slave, 有可能会丢失

同步复制和异步复制是通过 Broker 配置文件里的 brokerRole 参数进行设置的, 可以设置成 ASYNC_MASTER、RSYNC_MASTER、SLAVE 三个值中的一个

一般把刷盘机制配置成 ASYNC_FLUSH, 主从复制为 SYNC_MASTER, 这样即使有一台机器出故障, 仍然能保证数据不丢

RocketMQ 支持消息的高可靠, 影响消息可靠性的几种情况:

1. Broker 非正常关闭
2. Broker 异常 Crash
3. OS Crash
4. 机器掉电, 但是能立即恢复供电情况
5. 机器无法开机 (可能是 CPU、主板、内存等关键设备损坏)
6. 磁盘设备损坏

前四种情况都属于硬件资源可立即恢复情况, RocketMQ 在这四种情况下能保证消息不丢, 或者丢失少量数据 (依赖刷盘方式)

后两种属于单点故障，且无法恢复，一旦发生，在此单点上的消息全部丢失。RocketMQ 在这两种情况下，通过主从异步复制，可保证 99% 的消息不丢，但是仍然会有极少量的消息可能丢失。通过**同步双写技术**可以完全避免单点，但是会影响性能，适合对消息可靠性要求极高的场合，RocketMQ 从 3.0 版本开始支持同步双写。

一般而言，我们会建议采取同步双写 + 异步刷盘的方式，在消息的可靠性和性能间有一个较好的平衡。

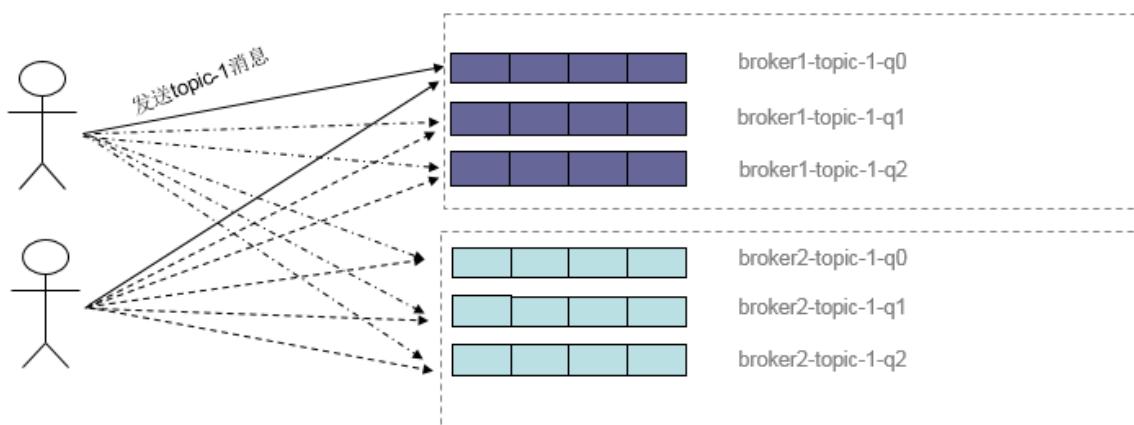
负载均衡

生产端

RocketMQ 中的负载均衡可以分为 Producer 端发送消息时候的负载均衡和 Consumer 端订阅消息的负载均衡。

Producer 端在发送消息时，会先根据 Topic 找到指定的 TopicPublishInfo，在获取了 TopicPublishInfo 路由信息后，RocketMQ 的客户端在默认方式调用 `selectOneMessageQueue()` 方法从 TopicPublishInfo 中的 messageQueueList 中选择一个队列 MessageQueue 进行发送消息。

默认会**轮询所有的 Message Queue 发送**，以让消息平均落在不同的 queue 上，而由于 queue 可以散落在不同的 Broker，所以消息就发送到不同的 Broker 下，图中箭头线条上的标号代表顺序，发布方会把第一条消息发送至 Queue 0，然后第二条消息发送至 Queue 1，以此类推：



每个producer默认采用RoundRobin方式轮训发送每个Queue

容错策略均在 MQFaultStrategy 这个类中定义，有一个 `sendLatencyFaultEnable` 开关变量：

- 如果开启，会在随机（只有初始化索引变量时才随机，正常都是递增）递增取模的基础上，再过滤掉 not available 的 Broker
- 如果关闭，采用随机递增取模的方式选择一个队列（MessageQueue）来发送消息

LatencyFaultTolerance 机制是实现消息发送高可用的核心关键所在，对之前失败的，按一定的时间做退避。例如上次请求的 latency 超过 550Lms，就退避 3000Lms；超过 1000L，就退避 60000L

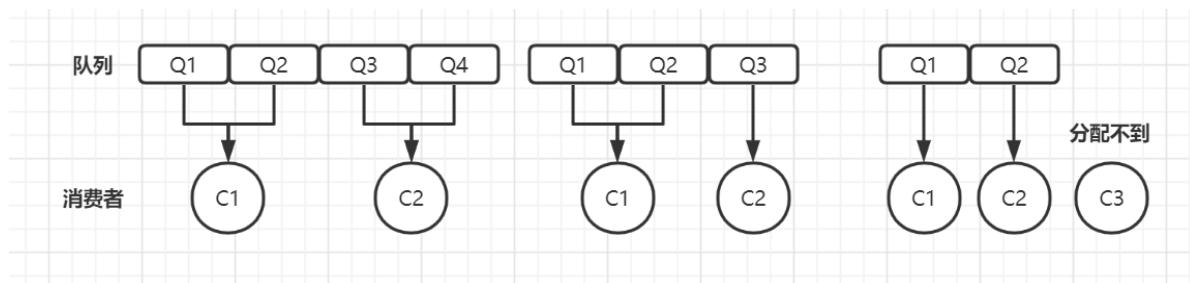
消费端

在 RocketMQ 中，Consumer 端的两种消费模式（Push/Pull）都是基于拉模式来获取消息的，而在 Push 模式只是对 Pull 模式的一种封装，其本质实现为消息拉取线程在从服务器拉取到一批消息，提交到消息消费线程池后，又继续向服务器再次尝试拉取消息，如果未拉取到消息，则延迟一下又继续拉取

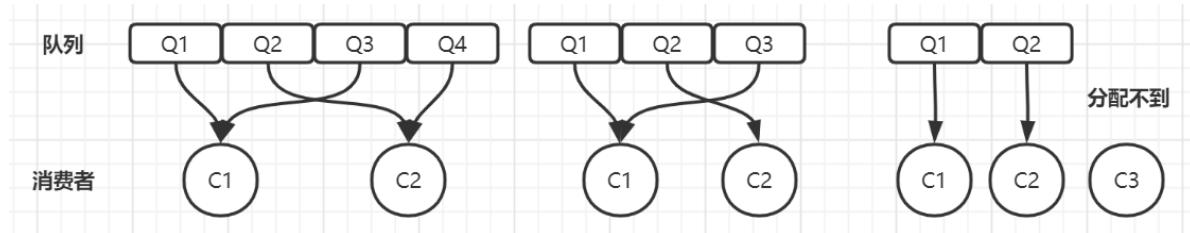
在两种基于拉模式的消费方式（Push/Pull）中，均需要 Consumer 端在知道从 Broker 端的哪一个消息队列中去获取消息，所以在 Consumer 端来做负载均衡，即 Broker 端中多个 MessageQueue 分配给同一个 Consumer Group 中的哪些 Consumer 消费

- 广播模式下要求一条消息需要投递到一个消费组下面所有的消费者实例，所以不存在负载均衡，在实现上，Consumer 分配 queue 时，所有 Consumer 都分到所有的 queue。
- 在集群消费模式下，每条消息只需要投递到订阅这个 Topic 的 Consumer Group 下的一个实例即可，RocketMQ 采用主动拉取的方式拉取并消费消息，在拉取的时候需要明确指定拉取哪一条 Message Queue

集群模式下，每当消费者实例的数量有变更，都会触发一次所有实例的负载均衡，这时候会按照 queue 的数量和实例的数量平均分配 queue 给每个实例。默认的分配算法是 AllocateMessageQueueAveragely：



还有一种平均的算法是 AllocateMessageQueueAveragelyByCircle，以环状轮流均分 queue 的形式：



集群模式下，queue 都是只允许分配一个实例，如果多个实例同时消费一个 queue 的消息，由于拉取哪些消息是 Consumer 主动控制的，会导致同一个消息在不同的实例下被消费多次

通过增加 Consumer 实例去分摊 queue 的消费，可以起到水平扩展的消费能力的作用。而当有实例下线时，会重新触发负载均衡，这时候原来分配到的 queue 将分配到其他实例上继续消费。但是如果 Consumer 实例的数量比 Message Queue 的总数量还多的话，多出来的 Consumer 实例将无法分到 queue，也就无法消费到消息，也就无法起到分摊负载的作用了，所以需要**控制让 queue 的总数量大于等于 Consumer 的数量**

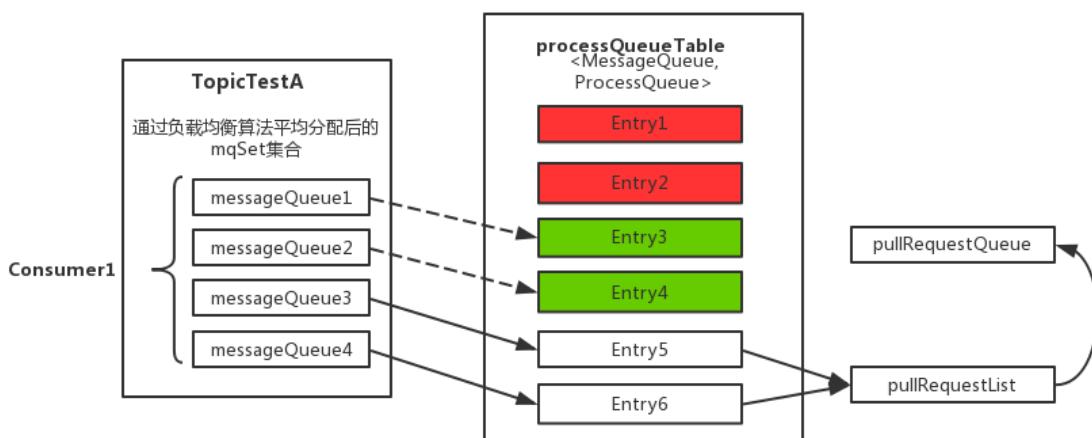
原理解析

在 Consumer 启动后，会通过定时任务不断地向 RocketMQ 集群中的所有 Broker 实例发送心跳包。Broker 端在收到 Consumer 的心跳消息后，会将它维护在 ConsumerManager 的本地缓存变量 consumerTable，同时并将封装后的客户端网络通道信息保存在本地缓存变量 channelInfoTable 中，为 Consumer 端的负载均衡提供可以依据的元数据信息。

Consumer 端实现负载均衡的核心类 RebalanceImpl

在 Consumer 实例的启动流程中的会启动 MQClientInstance 实例，完成负载均衡服务线程 RebalanceService 的启动（每隔 20s 执行一次负载均衡），RebalanceService 线程的 run() 方法最终调用的是 RebalanceImpl 类的 rebalanceByTopic() 方法，该方法是实现 Consumer 端负载均衡的核心。rebalanceByTopic() 方法会根据广播模式还是集群模式做不同的逻辑处理。主要看集群模式：

- 从 rebalanceImpl 实例的本地缓存变量 topicSubscribeInfoTable 中，获取该 Topic 主题下的消息消费队列集合 mqSet
- 根据 Topic 和 consumerGroup 为参数调用 `mqClientFactory.findConsumerIdList()` 方法向 Broker 端发送获取该消费组下消费者 ID 列表的 RPC 通信请求（Broker 端基于前面 Consumer 端上报的心跳包数据而构建的 consumerTable 做出响应返回，业务请求码 `GET_CONSUMER_LIST_BY_GROUP`）
- 先对 Topic 下的消息消费队列、消费者 ID 排序，然后用消息队列分配策略算法（默认是消息队列的平均分配算法），计算出待拉取的消息队列。平均分配算法类似于分页的算法，将所有 MessageQueue 排好序类似于记录，将所有消费端 Consumer 排好序类似页数，并求出每一页需要包含的平均 size 和每个页面记录的范围 range，最后遍历整个 range 而计算出当前 Consumer 端应该分配到的记录（这里即为 MessageQueue）
- 调用 updateProcessQueueTableInRebalance() 方法，先将分配到的消息队列集合 mqSet 与 processQueueTable 做一个过滤比对



- processQueueTable 标注的红色部分，表示与分配到的消息队列集合 mqSet 互不包含，将这些队列设置 Dropped 属性为 true，然后查看这些队列是否可以移除出 processQueueTable 缓存变量。具体执行 removeUnnecessaryMessageQueue() 方法，即每隔 1s 查看是否可以获取当前消费处理队列的锁，拿到的话返回 true；如果等待 1s 后，仍然拿不到当前消费处理队列的锁则返回 false。如果返回 true，则从 processQueueTable 缓存变量中移除对应的 Entry
- processQueueTable 的绿色部分，表示与分配到的消息队列集合 mqSet 的交集，判断该 ProcessQueue 是否已经过期了，在 Pull 模式的不用管，如果是 Push 模式的，设置 Dropped 属性为 true，并且调用 removeUnnecessaryMessageQueue() 方法，像上面一样尝试移除 Entry

- 为过滤后的消息队列集合 mqSet 中每个 MessageQueue 创建 ProcessQueue 对象存入 RebalanceImpl 的 processQueueTable 队列中（其中调用 RebalanceImpl 实例的 computePullFromWhere(MessageQueue mq) 方法获取该 MessageQueue 对象的下一个进度消费值 offset，随后填充至接下来要创建的 pullRequest 对象属性中），并创建拉取请求对象 pullRequest 添加到拉取列表 pullRequestList 中，最后执行 dispatchPullRequest() 方法，将 Pull 消息的请求对象 PullRequest 放入 PullMessageService 服务线程的阻塞队列 pullRequestQueue 中，待该服务线程取出后向 Broker 端发起 Pull 消息的请求

对比下 RebalancePushImpl 和 RebalancePullImpl 两个实现类的 dispatchPullRequest() 方法，RebalancePullImpl 类里面的该方法为空

消息消费队列在同一消费组不同消费者之间的负载均衡，其核心设计理念是在一个消息消费队列在同一时间只允许被同一消费组内的一个消费者消费，一个消息消费者能同时消费多个消息队列

消息查询

查询方式

RocketMQ 支持按照两种维度进行消息查询：按照 Message ID 查询消息、按照 Message Key 查询消息

- RocketMQ 中的 MessageID 的长度总共有 16 字节，其中包含了消息存储主机地址（IP 地址和端口），消息 Commit Log offset

实现方式：Client 端从 MessageID 中解析出 Broker 的地址（IP 地址和端口）和 Commit Log 的偏移地址，封装成一个 RPC 请求后通过 Remoting 通信层发送（业务请求码

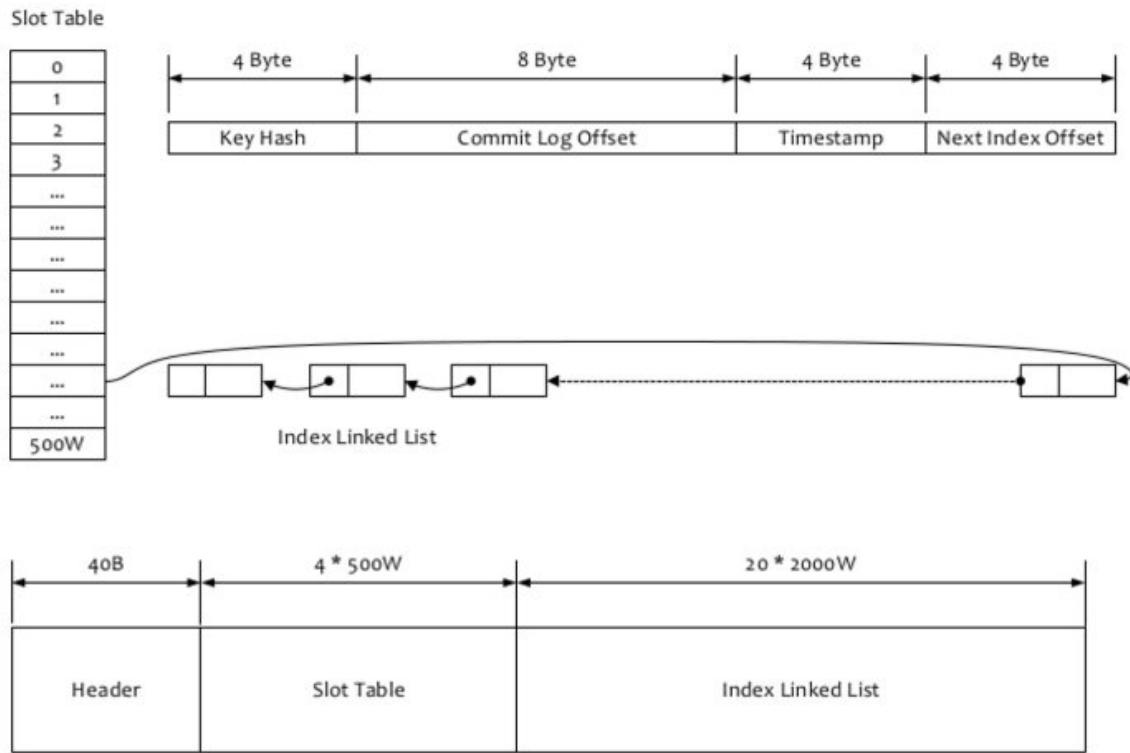
VIEW_MESSAGE_BY_ID）。Broker 端走的是 QueryMessageProcessor，读取消息的过程用其中的 CommitLog 的 offset 和 size 去 CommitLog 中找到真正的记录并解析成一个完整的消息返回

- 按照 Message Key 查询消息，IndexFile 索引文件为提供了通过 Message Key 查询消息的服务

实现方式：通过 Broker 端的 QueryMessageProcessor 业务处理器来查询，读取消息的过程用 **Topic 和 Key** 找到 IndexFile 索引文件中的一条记录，根据其中的 CommitLog Offset 从 CommitLog 文件中读取消息的实体内容

索引机制

RocketMQ 的索引文件逻辑结构，类似 JDK 中 HashMap 的实现，具体结构如下：



IndexFile 文件的存储在 `$HOME\store\index${fileName}`，文件名 fileName 是以创建时的时间戳命名，文件大小是固定的，等于 $40+500W*4+2000W*20= 420000040$ 个字节大小。如果消息的 properties 中设置了 UNIQ_KEY 这个属性，就用 `topic + "#" + UNIQ_KEY` 作为 key 来做写入操作；如果消息设置了 KEYS 属性（多个 KEY 以空格分隔），也会用 `topic + "#" + KEY` 来做索引

整个 Index File 的结构如图，40 Byte 的 Header 用于保存一些总的统计信息， $4*500W$ 的 Slot Table 并不保存真正的索引数据，而是保存每个槽位对应的单向链表的头指针，即一个 Index File 可以保存 2000W 个索引， $20*2000W$ 是真正的索引数据

索引数据包含了 Key Hash/CommitLog Offset/Timestamp/NextIndex offset 这四个字段，一共 20 Byte

- NextIndex offset 即前面读出来的 slotValue，如果有 hash 冲突，就可以用这个字段将所有冲突的索引用链表的方式串起来
- Timestamp 记录的是消息 storeTimestamp 之间的差，并不是一个绝对的时间

参考文档：<https://github.com/apache/rocketmq/blob/master/docs/cn/design.md>

消息重试

消息重投

生产者在发送消息时，同步消息和异步消息失败会重投，oneway 没有任何保证。消息重投保证消息尽可能发送成功、不丢失，但当出现消息量大、网络抖动时，可能会造成消息重复；生产者主动重发、Consumer 负载变化也会导致重复消息

如下方法可以设置消息重投策略：

- `retryTimesWhenSendFailed`: 同步发送失败重投次数，默认为 2，因此生产者会最多尝试发送 `retryTimesWhenSendFailed + 1` 次。不会选择上次失败的 Broker，尝试向其他 Broker 发送，**最大程度保证消息不丢**。超过重投次数抛出异常，由客户端保证消息不丢。当出现 `RemotingException`、`MQClientException` 和部分 `MQBrokerException` 时会重投
- `retryTimesWhenSendAsyncFailed`: 异步发送失败重试次数，异步重试不会选择其他 Broker，仅在同一个 Broker 上做重试，**不保证消息不丢**
- `retryAnotherBrokerWhenNotStoreOK`: 消息刷盘（主或备）超时或 slave 不可用（返回状态非 `SEND_OK`），是否尝试发送到其他 Broker，默认 false，十分重要的消息可以开启

注意点：

- 如果同步模式发送失败，则选择到下一个 Broker，如果异步模式发送失败，则**只会在当前 Broker 进行重试**
 - 发送消息超时时间默认 3000 毫秒，就不会再尝试重试
-

消息重试

Consumer 消费消息失败后，提供了一种重试机制，令消息再消费一次。Consumer 消费消息失败可以认为有以下几种情况：

- 由于消息本身的原因，例如反序列化失败，消息数据本身无法处理等。这种错误通常需要跳过这条消息，再消费其它消息，而这条失败的消息即使立刻重试消费，99% 也不成功，所以需要提供一种定时重试机制，即过 10 秒后再重试
- 由于依赖的下游应用服务不可用，例如 DB 连接不可用，外系统网络不可达等。这种情况即使跳过当前失败的消息，消费其他消息同样也会报错，这种情况建议应用 sleep 30s，再消费下一条消息，这样可以减轻 Broker 重试消息的压力

RocketMQ 会为每个消费组都设置一个 Topic 名称为 `%RETRY%+consumerGroup` 的重试队列（这个 Topic 的重试队列是**针对消费组**，而不是针对每个 Topic 设置的），用于暂时保存因为各种异常而导致 Consumer 端无法消费的消息

- 顺序消息的重试，当消费者消费消息失败后，消息队列 RocketMQ 会自动不断进行消息重试（每次间隔时间为 1 秒），这时应用会出现消息消费被阻塞的情况。所以在使用顺序消息时，必须保证应用能够及时监控并处理消费失败的情况，避免阻塞现象的发生
- 无序消息（普通、定时、延时、事务消息）的重试，可以通过设置返回状态达到消息重试的结果。无序消息的重试只针对集群消费方式生效，广播方式不提供失败重试特性，即消费失败后，失败消息不再重试，继续消费新的消息

无序消息情况下，因为异常恢复需要一些时间，会为重试队列设置多个重试级别，每个重试级别都有对应的重新投递延时，重试次数越多投递延时就越大。RocketMQ 对于重试消息的处理是先保存至 Topic 名称为 `SCHEDULE_TOPIC_XXXX` 的延迟队列中，后台定时任务按照对应的时间进行 Delay 后重新保存至 `%RETRY%+consumerGroup` 的重试队列中

消息队列 RocketMQ 默认允许每条消息最多重试 16 次，每次重试的间隔时间如下表示：

第几次重试	与上次重试的间隔时间	第几次重试	与上次重试的间隔时间
1	10 秒	9	7 分钟
2	30 秒	10	8 分钟
3	1 分钟	11	9 分钟
4	2 分钟	12	10 分钟
5	3 分钟	13	20 分钟
6	4 分钟	14	30 分钟
7	5 分钟	15	1 小时
8	6 分钟	16	2 小时

如果消息重试 16 次后仍然失败，消息将不再投递，如果严格按照上述重试时间间隔计算，某条消息在一直消费失败的前提下，将会在接下来的 4 小时 46 分钟之内进行 16 次重试，超过这个时间范围消息将不再重试投递

时间间隔不支持自定义配置，最大重试次数可通过自定义参数 `MaxReconsumeTimes` 取值进行配置，若配置超过 16 次，则超过的间隔时间均为 2 小时

说明：一条消息无论重试多少次，消息的 Message ID 是不会改变的

重试操作

集群消费方式下，消息消费失败后期望消息重试，需要在消息监听器接口的实现中明确进行配置（三种方式任选一种）：

- 返回 `Action.ReconsumeLater`（推荐）
- 返回 `null`
- 抛出异常

```
public class MessageListenerImpl implements MessageListener {
    @Override
    public Action consume(Message message, ConsumeContext context) {
        // 处理消息
        doConsumeMessage(message);
        // 方式1：返回 Action.ReconsumeLater，消息将重试
        return Action.ReconsumeLater;
        // 方式2：返回 null，消息将重试
        return null;
        // 方式3：直接抛出异常， 消息将重试
        throw new RuntimeException("Consumer Message exception");
    }
}
```

集群消费方式下，消息失败后期望消息不重试，需要捕获消费逻辑中可能抛出的异常，最终返回 Action.CommitMessage，此后这条消息将不会再重试

```
public class MessageListenerImpl implements MessageListener {
    @Override
    public Action consume(Message message, ConsumeContext context) {
        try {
            doConsumeMessage(message);
        } catch (Throwable e) {
            // 捕获消费逻辑中的所有异常，并返回 Action.CommitMessage;
            return Action.CommitMessage;
        }
        //消息处理正常，直接返回 Action.CommitMessage;
        return Action.CommitMessage;
    }
}
```

自定义消息最大重试次数，RocketMQ 允许 Consumer 启动的时候设置最大重试次数，重试时间间隔将按照如下策略：

- 最大重试次数小于等于 16 次，则重试时间间隔同上表描述
- 最大重试次数大于 16 次，超过 16 次的重试时间间隔均为每次 2 小时

```
Properties properties = new Properties();
// 配置对应 Group ID 的最大消息重试次数为 20 次
properties.put(PropertyKeyConst.MaxReconsumeTimes, "20");
Consumer consumer = ONSFactory.createConsumer(properties);
```

注意：

- 消息最大重试次数的设置对相同 Group ID 下的所有 Consumer 实例有效。例如只对相同 Group ID 下两个 Consumer 实例中的其中一个设置了 MaxReconsumeTimes，那么该配置对两个 Consumer 实例均生效
- 配置采用覆盖的方式生效，即最后启动的 Consumer 实例会覆盖之前的启动实例的配置

消费者收到消息后，可按照如下方式获取消息的重试次数：

```
public class MessageListenerImpl implements MessageListener {
    @Override
    public Action consume(Message message, ConsumeContext context) {
        // 获取消息的重试次数
        System.out.println(message.getReconsumeTimes());
        return Action.CommitMessage;
    }
}
```

死信队列

正常情况下无法被消费的消息称为死信消息（Dead-Letter Message），存储死信消息的特殊队列称为死信队列（Dead-Letter Queue）

当一条消息初次消费失败，消息队列 RocketMQ 会自动进行消息重试，达到最大重试次数后，若消费依然失败，则表明消费者在正常情况下无法正确地消费该消息，此时 RocketMQ 不会立刻将消息丢弃，而是将其发送到该消费者对应的死信队列中

死信消息具有以下特性：

- 不会再被消费者正常消费
- 有效期与正常消息相同，均为 3 天，3 天后会被自动删除，所以请在死信消息产生后的 3 天内及时处理

死信队列具有以下特性：

- 一个死信队列对应一个 Group ID，而不是对应单个消费者实例**
- 如果一个 Group ID 未产生死信消息，消息队列 RocketMQ 不会为其创建相应的死信队列
- 一个死信队列包含了对应 Group ID 产生的所有死信消息，不论该消息属于哪个 Topic

一条消息进入死信队列，需要排查可疑因素并解决问题后，可以在消息队列 RocketMQ 控制台重新发送该消息，让消费者重新消费一次

高可靠性

RocketMQ 消息丢失可能发生在以下三个阶段：

- 生产阶段：消息在 Producer 发送端创建出来，经过网络传输发送到 Broker 存储端
 - 生产者得到一个成功的响应，就可以认为消息的存储和消息的消费都是可靠的
 - 消息重投机制
- 存储阶段：消息在 Broker 端存储，如果是主备或者多副本，消息会在这个阶段被复制到其他的节点或者副本上
 - 单点：刷盘机制（同步或异步）
 - 主从：消息同步机制（异步复制或同步双写，主从复制章节详解）
 - 过期删除：操作 CommitLog、ConsumeQueue 文件是基于文件内存映射机制，并且在启动的时候会将所有的文件加载，为了避免内存与磁盘的浪费，让磁盘能够循环利用，防止磁盘不足导致消息无法写入等引入了文件过期删除机制。最终使得磁盘水位保持在一定水平，最终保证新写入消息的可靠存储
- 消费阶段：Consumer 消费端从 Broker 存储端拉取消息，经过网络传输发送到 Consumer 消费端上
 - 消息重试机制来最大限度的保证消息的消费
 - 消费失败的进行消息回退，重试次数过多的消息放入死信队列

推荐文章：<https://cdn.modb.pro/db/394751>

幂等消费

消息队列 RocketMQ 消费者在接收到消息以后，需要根据业务上的唯一 Key 对消息做幂等处理

At least Once 机制保证消息不丢失，但是可能会造成消息重复，RocketMQ 中无法避免消息重复 (Exactly-Once)，在互联网应用中，尤其在网络不稳定的情况下，几种情况：

- **发送时消息重复**：当一条消息已被成功发送到服务端并完成持久化，此时出现了网络闪断或客户端宕机，导致服务端对客户端应答失败。此时生产者意识到消息发送失败并尝试再次发送消息，消费者后续会收到两条内容相同并且 Message ID 也相同的消息
- **投递时消息重复**：消息消费的场景下，消息已投递到消费者并完成业务处理，当客户端给服务端反馈应答的时候网络闪断。为了保证消息至少被消费一次，消息队列 RocketMQ 的服务端将在网络恢复后再次尝试投递之前已被处理过的消息，消费者后续会收到两条内容相同并且 Message ID 也相同的消息
- **负载均衡时消息重复**：当消息队列 RocketMQ 的 Broker 或客户端重启、扩容或缩容时，会触发 Rebalance，此时消费者可能会收到重复消息

处理方式：

- 因为 Message ID 有可能出现冲突（重复）的情况，所以真正安全的幂等处理，不建议以 Message ID 作为处理依据，最好的方式是以业务唯一标识作为幂等处理的关键依据，而业务的唯一标识可以通过消息 Key 进行设置：

```
Message message = new Message();
message.setKey("ORDERID_100");
SendResult sendResult = producer.send(message);
```

- 订阅方收到消息时可以根据消息的 Key 进行幂等处理：

```
consumer.subscribe("ons_test", "*", new MessageListener() {
    public Action consume(Message message, ConsumeContext context) {
        String key = message.getKey()
        // 根据业务唯一标识的 key 做幂等处理
    }
});
```

流量控制

生产者流控，因为 Broker 处理能力达到瓶颈；消费者流控，因为消费能力达到瓶颈

生产者流控：

- CommitLog 文件被锁时间超过 osPageCacheBusyTimeOutMills 时，参数默认为 1000ms，返回流控

- 如果开启 transientStorePoolEnable == true，且 Broker 为异步刷盘的主机，且 transientStorePool 中资源不足，拒绝当前 send 请求，返回流控
- Broker 每隔 10ms 检查 send 请求队列头部请求的等待时间，如果超过 waitTimeMillsInSendQueue，默认 200ms，拒绝当前 send 请求，返回流控。
- Broker 通过拒绝 send 请求方式实现流量控制

注意：生产者流控，不会尝试消息重投

消费者流控：

- 消费者本地缓存消息数超过 pullThresholdForQueue 时，默认 1000
- 消费者本地缓存消息大小超过 pullThresholdSizeForQueue 时，默认 100MB
- 消费者本地缓存消息跨度超过 consumeConcurrentlyMaxSpan 时，默认 2000

消费者流控的结果是降低拉取频率

原理解析

Namesrv

服务启动

启动方法

NamesrvStartup 类中有 Namesrv 服务的启动方法：

```
public static void main(String[] args) {
    // 如果启动时 使用 -c -p 设置参数了，这些参数存储在 args 中
    main0(args);
}

public static NamesrvController main0(String[] args) {
    try {
        // 创建 namesrv 控制器，用来初始化 namesrv 启动 namesrv 关闭 namesrv
        NamesrvController controller = createNamesrvController(args);
        // 启动 controller
        start(controller);
        return controller;
    } catch (Throwable e) {
        // 出现异常，停止系统
        System.exit(-1);
    }
    return null;
}
```

NamesrvStartup#createNamesrvController：读取配置信息，初始化 Namesrv 控制器

- `ServerUtil.parseCmdLine("mqnamesrv", args, buildCommandLineOptions(options), ..)`: 解析启动时的参数信息
- `namesrvConfig = new NamesrvConfig()`: 创建 Namesrv 配置对象
 - `private String rocketmqHome`: 获取 ROCKETMQ_HOME 值
 - `private boolean orderMessageEnable = false`: 顺序消息功能是否开启
- `nettyServerConfig = new NettyServerConfig()`: Netty 的服务器配置对象
- `nettyServerConfig.setListenPort(9876)`: Namesrv 服务器的监听端口设置为 9876
- `if (commandLine.hasOption('c'))`: 读取命令行 -c 的参数值
 - `in = new BufferedInputStream(new FileInputStream(file))`: 读取指定目录的配置文件
 - `properties.load(in)`: 将配置文件信息加载到 properties 对象，相关属性会复写到 Namesrv 配置和 Netty 配置对象
 - `namesrvConfig.setConfigStorePath(file)`: 将配置文件的路径保存到配置保存字段
- `if (null == namesrvConfig.getRocketmqHome())`: 检查 ROCKETMQ_HOME 配置是否是空，是空就报错
- `lc = (LoggerContext) LoggerFactory.getLoggerFactory()`: 创建日志对象
- `controller = new NamesrvController(namesrvConfig, nettyServerConfig)`: 创建 Namesrv 控制器

NamesrvStartup#start: 启动 Namesrv 控制器

- `boolean initResult = controller.initialize()`: 初始化方法
- `Runtime.getRuntime().addShutdownHook(new ShutdownHookThread())`: JVM HOOK 平滑关闭的逻辑，当 JVM 被关闭时，主动调用 controller.shutdown() 方法，让服务器平滑关机
- `controller.start()`: 启动服务器

源码解析参考视频: <https://space.bilibili.com/457326371>

控制器类

NamesrvController 用来初始化和启动 Namesrv 服务器

- 成员变量:

```

private final ScheduledExecutorService scheduledExecutorService;           // 调度线程池，用来执行定时任务
private final RouteInfoManager routeInfoManager;                         // 管理【路由信息】的对象
private RemotingServer remotingServer;                                    // 【网络层】封装对象
private BrokerHousekeepingService brokerHousekeepingService;           // 用于监听 channel 状态

```

`private ExecutorService remotingExecutor`: 业务线程池, netty 线程解析报文成 RemotingCommand 对象, 然后将该对象交给业务线程池再继续处理

- 初始化:

```
public boolean initialize() {
    // 加载本地kv配置 (我还不明白 kv 配置是啥)
    this.kvConfigManager.load();
    // 创建网络服务器对象, 【将 netty 的配置和监听器传入】
    // 监听器监听 channel 状态的改变, 会向事件队列发起事件, 最后交由 service 处理
    this.remotingServer = new NettyRemotingServer(this.nettyServerConfig,
this.brokerHousekeepingService);
    // 【创建业务线程池, 默认线程数 8】
    this.remotingExecutor =
Executors.newFixedThreadPool(nettyServerConfig.getServerWorkerThreads());
    // 注册协议处理器 (缺省协议处理器), 【处理器是 DefaultRequestProcessor】，线程使用的是刚创建的业务的线程池
    this.registerProcessor();

    // 定时任务1: 每 10 秒钟检查 broker 存活状态, 将 IDLE 状态的 broker 移除【扫描机制, 心跳检测】
    this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
        @Override
        public void run() {
            // 扫描 brokerLiveTable 表, 将两小时没有活动的 broker 关闭,
            // 通过 next.getKey() 获取 broker 的地址, 然后【关闭服务器与broker物理
            // 节点的 channel】
            NamesrvController.this.routeInfoManager.scanNotActiveBroker();
        }
    }, 5, 10, TimeUnit.SECONDS);

    // 定时任务2: 每 10 分钟打印一遍 kv 配置。
    this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
        @Override
        public void run() {
            NamesrvController.this.kvConfigManager.printAllPeriodically();
        }
    }, 1, 10, TimeUnit.MINUTES);

    return true;
}
```

- 启动方法:

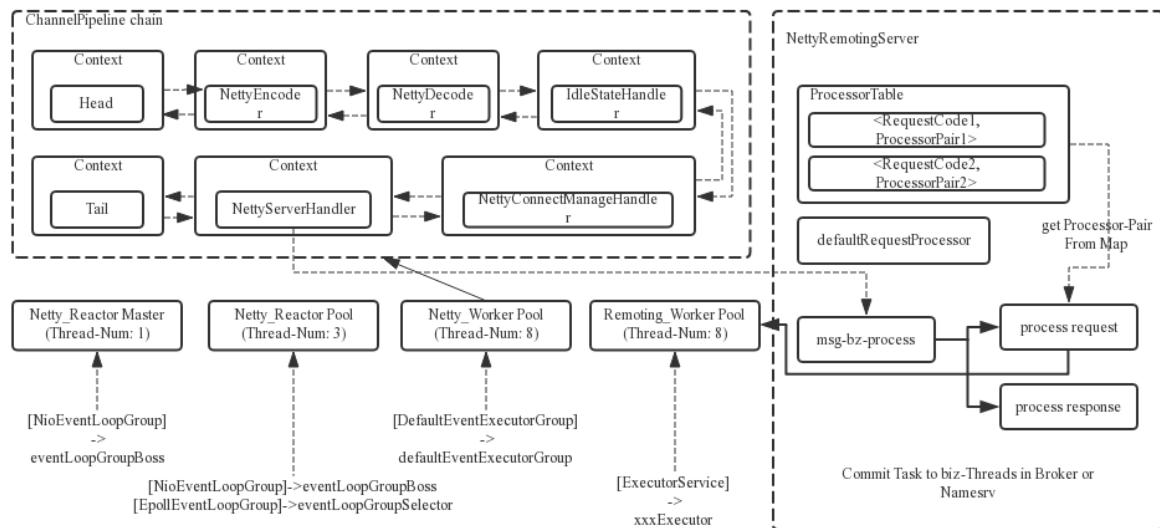
```
public void start() throws Exception {
    // 服务器网络层启动。
    this.remotingServer.start();

    if (this.filewatchService != null) {
        this.filewatchService.start();
    }
}
```

网络通信

通信原理

RocketMQ 的 RPC 通信采用 Netty 组件作为底层通信库，同样也遵循了 Reactor 多线程模型，NettyRemotingServer 类负责框架的通信服务，同时又在这之上做了一些扩展和优化

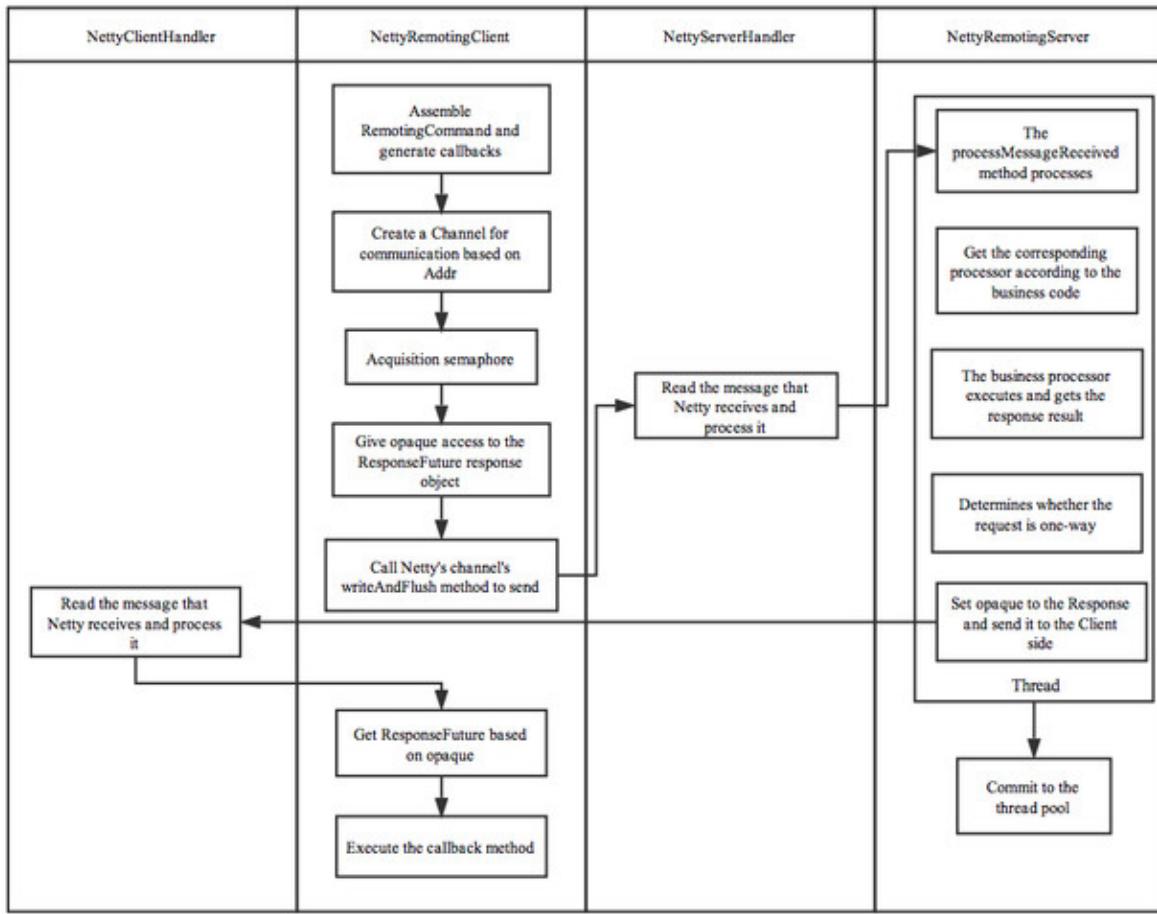


RocketMQ 基于 NettyRemotingServer 的 Reactor 多线程模型：

- 一个 Reactor 主线程 (eventLoopGroupBoss) 负责监听 TCP 网络连接请求，建立好连接创建 SocketChannel (RocketMQ 会自动根据 OS 的类型选择 NIO 和 Epoll，也可以通过参数配置)，并注册到 Selector 上，然后监听真正的网络数据
- 拿到网络数据交给 Worker 线程池 (eventLoopGroupSelector，默认设置为 3)，在真正执行业务逻辑之前需要进行 SSL 验证、编解码、空闲检查、网络连接管理，这些工作交给 defaultEventExecutorGroup (默认设置为 8) 去做
- 处理业务操作放在业务线程池中执行，根据 RemotingCommand 的**业务请求码 code**去 processorTable 这个本地缓存变量中找到对应的 processor，封装成 task 任务提交给对应的 processor 处理线程池来执行 (sendMessageExecutor，以发送消息为例)
- 从入口到业务逻辑的几个步骤中线程池一直再增加，这跟每一步逻辑复杂性相关，越复杂，需要的并发通道越宽

线程数	线程名	线程具体说明
1	NettyBoss_%d	Reactor 主线程
N	NettyServerEPOLLSelector%d%d	Reactor 线程池
M1	NettyServerCodecThread_%d	Worker 线程池
M2	RemotingExecutorThread_%d	业务 processor 处理线程池

RocketMQ 的异步通信流程：



--todo：后期对 Netty 有了更深的认知后会进行扩充，现在暂时 copy 官方文档--

官方文档：<https://github.com/apache/rocketmq/blob/master/docs/cn/design.md#2-%E9%80%9A%E4%BF%A1%E6%9C%BA%E5%88%B6>

成员属性

NettyRemotingServer 类成员变量：

- 服务器相关属性：

```

private final ServerBootstrap serverBootstrap;           // netty 服务端启动对象
private final EventLoopGroup eventLoopGroupSelector;   // netty worker
                                                       组线程池，【默认 3 个线程】
private final EventLoopGroup eventLoopGroupBoss;        // netty boss 组
                                                       线程池，【一般是 1 个线程】
private final NettyServerConfig nettyServerConfig;      // netty 服务端网络配置
private int port = 0;                                    // 服务器绑定的端口
  
```

- 公共线程池：注册处理器时如果未指定线程池，则业务处理使用公共线程池，线程数量默认是 4

```
private final ExecutorService publicExecutor;
```

- 事件监听器：Nameserver 使用 BrokerHouseKeepingService，Broker 使用 ClientHouseKeepingService

```
private final ChannelEventListener channelEventListener;
```

- 事件处理线程池：默认是 8

```
private DefaultEventExecutorGroup defaultEventExecutorGroup;
```

- 定时器：执行循环任务，并且将定时器线程设置为守护线程

```
private final Timer timer = new Timer("ServerHouseKeepingService", true);
```

- 处理器：多个 Channel 共享的处理器 Handler，多个通道使用同一个对象
- Netty 配置对象：

```
public class NettyServerConfig implements Cloneable {  
    // 服务端启动时监听的端口号  
    private int listenPort = 8888;  
    // 【业务线程池】线程数量  
    private int serverWorkerThreads = 8;  
    // 根据该值创建 remotingServer 内部的一个 publicExecutor  
    private int serverCallbackExecutorThreads = 0;  
    // netty 【worker】线程数  
    private int serverSelectorThreads = 3;  
    // 【单向访问】时的并发限制  
    private int serverOneWaySemaphoreValue = 256;  
    // 【异步访问】时的并发限制  
    private int serverAsyncSemaphoreValue = 64;  
    // channel 最大的空闲存活时间 默认是 2min  
    private int serverChannelMaxIdleTimeSeconds = 120;  
    // 发送缓冲区大小 65535  
    private int serverSocketSndBufSize = NettySystemConfig.socketSndbufSize;  
    // 接收缓冲区大小 65535  
    private int serverSocketRcvBufSize = NettySystemConfig.socketRcvbufSize;  
    // 是否启用 netty 内存池 默认开启  
    private boolean serverPooledByteBufAllocatorEnable = true;  
  
    // 默认 Linux 会启用 【epoll】  
    private boolean useEpollNativeSelector = false;  
}
```

构造方法：

- 无监听器构造：

```
public NettyRemotingServer(final NettyServerConfig nettyServerConfig) {  
    this(nettyServerConfig, null);  
}
```

- 有参构造方法:

```

public NettyRemotingServer(final NettyServerConfig nettyServerConfig,
                           final ChannelEventListener channelEventListener)
{
    // 服务器对客户端主动发起请求时并发限制。【单向请求和异步请求】的并发限制
    super(nettyServerConfig.getServerOnewaySemaphoreValue(),
          nettyServerConfig.getServerAsyncSemaphoreValue());
    // Netty 的启动器，负责组装 netty 组件
    this.serverBootstrap = new ServerBootstrap();
    // 成员变量的赋值
    this.nettyServerConfig = nettyServerConfig;
    this.channelEventListener = channelEventListener;

    // 公共线程池的线程数量，默认给的0，这里最终修改为4。
    int publicThreadNums =
        nettyServerConfig.getServerCallbackExecutorThreads();
    if (publicThreadNums <= 0) {
        publicThreadNums = 4;
    }
    // 创建公共线程池，指定线程工厂，设置线程名称前缀：NettyServerPublicExecutor_[数字]
    this.publicExecutor = Executors.newFixedThreadPool(publicThreadNums, new
        ThreadFactory(){});

    // 创建两个 netty 的线程组，一个是boss组，一个是worker组，【linux 系统默认启用
    // epoll】
    if (useEpoll()) {...} else {...}
    // SSL 相关
    loadSslContext();
}

```

启动方法

核心方法的解析:

- start(): 启动方法，创建 BootStrap，并添加 NettyServerHandler 处理器

```

public void start() {
    // Channel Pipeline 内的 handler 使用的线程资源，【线程分配给 handler 处理事
    // 件】
    this.defaultEventExecutorGroup = new DefaultEventExecutorGroup(...);

    // 创建通用共享的处理器 handler，【非常重要的 NettyServerHandler】
    prepareSharableHandlers();

    ServerBootstrap childHandler =
        // 配置工作组 boss（数量1） 和 worker（数量3） 组

```

```

        this.serverBootstrap.group(this.eventLoopGroupBoss,
this.eventLoopGroupSelector)
    // 设置服务端 ServerSocketChannel 类型, Linux 用 epoll
    .channel(useEpoll() ? EpollServerSocketChannel.class :
NioServerSocketChannel.class)
    // 设置服务端 channel 选项
    .option(ChannelOption.SO_BACKLOG, 1024)
    // 客户端 channel 选项
    .childOption(ChannelOption.TCP_NODELAY, true)
    // 设置服务器端口
    .localAddress(new
InetSocketAddress(this.nettyServerConfig.getListenPort()))
    // 向 channel pipeline 添加了很多 handler, 【包括 NettyServerHandler】
    .childHandler(new ChannelInitializer<SocketChannel>() {});

    // 客户端开启 内存池, 使用的内存池是 PooledByteBufAllocator.DEFAULT
    if (nettyServerConfig.isServerPooledByteBufAllocatorEnable()) {
        childHandler.childOption(ChannelOption.ALLOCATOR,
PooledByteBufAllocator.DEFAULT);
    }

    try {
        // 同步等待建立连接, 并绑定端口。
        ChannelFuture sync = this.serverBootstrap.bind().sync();
        InetSocketAddress addr = (InetSocketAddress)
sync.channel().localAddress();
        // 将服务器成功绑定的端口号赋值给字段 port。
        this.port = addr.getPort();
    } catch (InterruptedException e1) {}

    // housekeepingService 不为空, 则创建【网络异常事件处理器】
    if (this.channelEventListener != null) {
        // 线程一直轮询 nettyEvent 状态, 根据 CONNECT,CLOSE,IDLE,EXCEPTION 四种事
件类型
        // CONNECT 不做操作, 其余都是回调 onChannelDestroy 【关闭服务器与 Broker 物
理节点的 Channel】
        this.nettyEventExecutor.start();
    }

    // 提交定时任务, 每一秒 执行一次。扫描 responseTable 表, 将过期的数据移除
    this.timer.scheduleAtFixedRate(new TimerTask() {
        @Override
        public void run() {
            NettyRemotingServer.this.scanResponseTable();
        }
    }, 1000 * 3, 1000);
}

```

- registerProcessor(): 注册业务处理器

```

public void registerProcessor(int requestCode, NettyRequestProcessor
processor, ExecutorService executor) {
    ExecutorService executorThis = executor;
    if (null == executor) {
        // 未指定线程池资源，将公共线程池赋值
        executorThis = this.publicExecutor;
    }
    // pair 对象，第一个参数代表的是处理器， 第二个参数是线程池，默认是公共的线程池
    Pair<NettyRequestProcessor, ExecutorService> pair = new
Pair<NettyRequestProcessor, ExecutorService>(processor, executorThis);

    // key 是请求码， value 是 pair 对象
    this.processorTable.put(requestCode, pair);
}

```

- getProcessorPair(): 根据请求码获取对应的处理器和线程池资源

```

public Pair<NettyRequestProcessor, ExecutorService> getProcessorPair(int
requestCode) {
    return processorTable.get(requestCode);
}

```

请求方法

在 RocketMQ 消息队列中支持通信的方式主要有同步 (sync) 、异步 (async) 、单向 (oneway) 三种，其中单向通信模式相对简单，一般用在发送心跳包场景下，无需关注其 Response

服务器主动向客户端发起请求时，使用三种方法

- invokeSync(): 同步调用，**服务器需要阻塞等待调用的返回结果**
 - int opaque = request.getopaque(): 获取请求 ID (与请求码不同)
 - responseFuture = new ResponseFuture(...): **创建响应对象**，没有回调函数和 Once
 - this.responseTable.put(opaque, responseFuture): **加入到响应映射表中**，key 为请求 ID
 - SocketAddress addr = channel.remoteAddress(): 获取客户端的地址信息
 - channel.writeAndFlush(request).addListener(...): **将业务 Command 信息写入通道**，业务线程将数据交给 Netty，Netty 的 IO 线程接管写刷数据的操作，**监听器由 IO 线程在写刷后回调**
 - if (f.isSuccess()): 写入成功会将响应对象设置为成功状态直接 return，写入失败设置为失败状态
 - responseTable.remove(opaque): 将当前请求的 responseFuture **从映射表移除**
 - responseFuture.setCause(f.cause()): 设置错误的信息
 - responseFuture.putResponse(null): 响应 Command 设置为 null
 - responseCommand = responseFuture.waitResponse(timeoutMillis): **当前线程设置超时时间挂起，同步等待响应**
 - if (null == responseCommand): 超时或者出现异常，直接报错

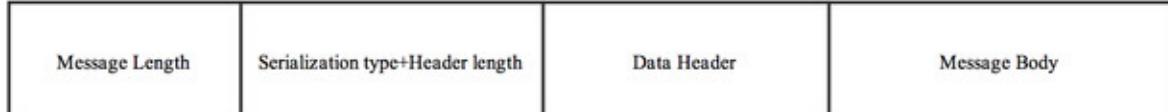
- `return responseCommand`：返回响应 Command 信息
 - `invokeAsync()`：异步调用，有回调对象，无返回值
 - `boolean acquired = this.semaphoreAsync.tryAcquire(timeoutMillis, TimeUnit.MILLISECONDS)`：获取信号量的许可证，信号量用来**限制异步请求的数量**
 - `if (acquired)`：许可证获取失败说明并发较高，会抛出异常
 - `once = new SemaphoreReleaseOnlyOnce(this.semaphoreAsync)`：Once 对象封装了释放信号量的操作
 - `costTime = System.currentTimeMillis() - beginStartTime`：计算一下耗费的时间，超时不再发起请求
 - `responseFuture = new ResponseFuture()`：创建响应对象，**包装了回调函数和 Once 对象**
 - `this.responseTable.put(opaque, responseFuture)`：加入到响应映射表中，key 为请求 ID
 - `channel.writeAndFlush(request).addListener(...)`：写刷数据
 - `if (f.isSuccess())`：写刷成功，设置 responseFuture 发生状态为 true
 - `requestFail(opaque)`：写入失败，使用 publicExecutor **公共线程池异步执行回调对象的函数**
 - `responseFuture.release()`：出现异常会释放信号量
 - `invokeOneway()`：单向调用，不关注响应结果
 - `request.markOnewayRPC()`：设置单向标记，对端检查标记可知该请是单向请求
 - `boolean acquired = this.semaphoreOneway.tryAcquire(timeoutMillis, TimeUnit.MILLISECONDS)`：获取信号量的许可证，信号量用来**限制单向请求的数量**
-

处理器类

协议设计

在 Client 和 Server 之间完成一次消息发送时，需要对发送的消息进行一个协议约定，所以自定义 RocketMQ 的消息协议。在 RocketMQ 中，为了高效地在网络中传输消息和对收到的消息读取，就需要对消息进行编解码，RemotingCommand 这个类在消息传输过程中对所有数据内容的封装，不但包含了所有的数据结构，还包含了编码解码操作

Header 字段	类型	Request 说明	Response 说明
code	int	请求操作码，应答方根据不同的请求码进行不同的处理	应答响应码，0 表示成功，非 0 则表示各种错误
language	LanguageCode	请求方实现的语言	应答方实现的语言
version	int	请求方程序的版本	应答方程序的版本
opaque	int	相当于 requestId，在同一个连接上的不同请求标识码，与响应消息中的相对应	应答不做修改直接返回
flag	int	区分是普通 RPC 还是一次性 RPC 的标志	区分是普通 RPC 还是一次性 RPC 的标志
remark	String	传输自定义文本信息	传输自定义文本信息
extFields	HashMap<String, String>	请求自定义扩展信息	响应自定义扩展信息



传输内容主要可以分为以下四部分：

- 消息长度：总长度，四个字节存储，占用一个 int 类型
- 序列化类型&消息头长度：同样占用一个 int 类型，第一个字节表示序列化类型，后面三个字节表示消息头长度
- 消息头数据：经过序列化后的消息头数据
- 消息主体数据：消息主体的二进制字节数据内容

官方文档：<https://github.com/apache/rocketmq/blob/master/docs/cn/design.md>

处理方法

NettyServerHandler 类用来处理 Channel 上的事件，在 NettyRemotingServer 启动时注册到 Netty 中，可以处理 RemotingCommand 相关的数据，针对某一种类型的**请求处理**

```
class NettyServerHandler extends SimpleChannelInboundHandler<RemotingCommand> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, RemotingCommand msg)
        throws Exception {
        // 服务器处理接受到的请求信息
        processMessageReceived(ctx, msg);
    }
}

public void processMessageReceived(ChannelHandlerContext ctx, RemotingCommand msg) throws Exception {
    final RemotingCommand cmd = msg;
    if (cmd != null) {
        // 根据请求的类型进行处理
        switch (cmd.getType()) {
            case REQUEST_COMMAND:// 客户端发起的请求，走这里
                processRequestCommand(ctx, cmd);
                break;
            case RESPONSE_COMMAND:// 客户端响应的数据，走这里【当前类本身是服务器类也是客户端类】
                processResponseCommand(ctx, cmd);
                break;
            default:
                break;
        }
    }
}
```

NettyRemotingAbstract#processRequestCommand: **处理请求的数据**

- `matched = this.processorTable.get(cmd.getCode())`: 根据业务请求码获取 Pair 对象，包含**处理器和线程池资源**
- `pair = null == matched ? this.defaultRequestProcessor : matched`: 未找到处理器则使用缺省处理器
- `int opaque = cmd.getOpaque()`: 获取请求 ID
- `Runnable run = new Runnable()`: 创建任务对象，任务在提交到线程池后开始执行
 - `doBeforeRpcHooks()`: RPC HOOK 前置处理
 - `callback = new RemotingResponseCallback()`: **封装响应客户端的逻辑**
 - `doAfterRpcHooks()`: RPC HOOK 后置处理
 - `if (!cmd.isOneWayRPC())`: 条件成立说明不是单向请求，需要结果
 - `response.setOpaque(opaque)`: 将请求 ID 设置到 response
 - `response.markResponseType()`: **设置当前请求是响应**
 - `ctx.writeAndFlush(response)`: **将响应数据交给 Netty IO 线程，完成数据写和刷**
 - `if (pair.getObject1() instanceof AsyncNettyRequestProcessor)`: Nameserver 默认使用 DefaultRequestProcessor 处理器，是一个 AsyncNettyRequestProcessor 子类
 - `processor = (AsyncNettyRequestProcessor)pair.getObject1()`: 获取处理器

- `processor.asyncProcessRequest(ctx, cmd, callback)`: 异步调用, 首先 `processRequest`, 然后 `callback` 响应客户端
 - `DefaultRequestProcessor.processRequest`: 根据业务码处理请求, 执行对应的操作
 - `ClientRemotingProcessor.processRequest`: 处理事务回查消息, 或者回执消息, 需要消费者回执一条消息给生产者
- `requestTask = new RequestTask(run, ctx.channel(), cmd)`: 将任务对象、通道、请求封装成 `RequestTask` 对象
- `pair.getObject2().submit(requestTask)`: 获取处理器对应的线程池, 将 task 提交, 从 IO 线程切换到业务线程

NettyRemotingAbstract#processResponseCommand: 处理响应的数据

- `int opaque = cmd.getOpaque()`: 获取请求 ID
- `responseFuture = responseTable.get(opaque)`: 从响应映射表中获取对应的对象
- `responseFuture.setResponseCommand(cmd)`: 设置响应的 Command 对象
- `responseTable.remove(opaque)`: 从映射表中移除对象, 代表处理完成
- `if (responseFuture.getInvokeCallback() != null)`: 包含回调对象, 异步执行回调对象
- `responseFuture.putResponse(cmd)`: 不包含回调对象, 同步调用时, 唤醒等待的业务线程

流程: 客户端 `invokeSync` → 服务器的 `processRequestCommand` → 客户端的 `processResponseCommand` → 结束

路由信息

信息管理

`RouteInfoManager` 类负责管理路由信息, `NamesrvController` 的构造方法中创建该类的实例对象, 管理服务端的路由数据

```
public class RouteInfoManager {
    // Broker 两个小时不活跃, 视为离线, 被定时任务删除
    private final static long BROKER_CHANNEL_EXPIRED_TIME = 1000 * 60 * 2;
    // 读写锁, 保证线程安全
    private final ReadwriteLock lock = new ReentrantReadWriteLock();
    // 主题队列数据, 一个主题对应多个队列
    private final HashMap<String/* topic */, List<QueueData>> topicQueueTable;
    // Broker 数据列表
    private final HashMap<String/* brokerName */, BrokerData> brokerAddrTable;
    // 集群
    private final HashMap<String/* clusterName */, Set<String/* brokerName *>> clusterAddrTable;
    // Broker 存活信息
    private final HashMap<String/* brokerAddr */, BrokerLiveInfo> brokerLiveTable;
    // 服务过滤
    private final HashMap<String/* brokerAddr */, List<String>/> filterServerTable;
}
```

路由注册

DefaultRequestProcessor REGISTER_BROKER 方法解析：

```
public RemotingCommand registerBroker(ChannelHandlerContext ctx, RemotingCommand request) {
    // 创建响应请求的对象，设置为响应类型，【先设置响应的状态码时系统错误码】
    // 反射创建 RegisterBrokerResponseHeader 对象设置到 response.customHeader 属性中
    final RemotingCommand response =
        RemotingCommand.createResponseCommand(RegisterBrokerResponseHeader.class);

    // 获取出反射创建的 RegisterBrokerResponseHeader 用户自定义header对象。
    final RegisterBrokerResponseHeader responseHeader =
        (RegisterBrokerResponseHeader) response.readCustomHeader();

    // 反射创建 RegisterBrokerRequestHeader 对象，并且将 request.extFields 中的数据写入到该对象中
    final RegisterBrokerRequestHeader requestHeader =
        request.decodeCommandCustomHeader(RegisterBrokerRequestHeader.class);

    // CRC 校验，计算请求中的 CRC 值和请求头中包含的是否一致
    if (!checksum(ctx, request, requestHeader)) {
        response.setCode(ResponseCode.SYSTEM_ERROR);
        response.setRemark("crc32 not match");
        return response;
    }

    TopicConfigSerializeWrapper topicConfigWrapper;
    if (request.getBody() != null) {
        // 【解析请求体 body】，解码出来的数据就是当前机器的主题信息
        topicConfigWrapper =
            TopicConfigSerializeWrapper.decode(request.getBody(),
                TopicConfigSerializeWrapper.class);
    } else {
        topicConfigWrapper = new TopicConfigSerializeWrapper();
        topicConfigWrapper.getDataVersion().setCounter(new AtomicLong(0));
        topicConfigWrapper.getDataVersion().setTimestamp(0);
    }

    // 注册方法
    // 参数1 集群、参数2：节点ip地址、参数3：brokerName、参数4：brokerId 注意brokerId=0 的节点为主节点
    // 参数5：ha节点ip地址、参数6当前节点主题信息、参数7：过滤服务器列表、参数8：当前服务器和客户端通信的channel
    RegisterBrokerResult result =
        this.namesrvController.getRouteInfoManager().registerBroker(..);

    // 将结果信息 写到 responseHeader 中
    responseHeader.setHaServerAddr(result.getHaServerAddr());
    responseHeader.setMasterAddr(result.getMasterAddr());
    // 获取 kv配置，写入 response body 中，【kv 配置是顺序消息相关的】
```

```

byte[] jsonValue =
this.namesrvController.getKvConfigManager().getKVListByNamespace(NamesrvUtil.NAM
ESPACE_ORDER_TOPIC);
response.setBody(jsonValue);

// code 设置为 SUCCESS
response.setCode(ResponseCode.SUCCESS);
response.setRemark(null);
// 返回 response , 【返回的 response 由 callback 对象处理】
return response;
}

```

RouteInfoManager#registerBroker: 注册 Broker 的信息

- `RegisterBrokerResult result = new RegisterBrokerResult()`: 返回结果的封装对象
- `this.lock.writeLock().lockInterruptibly()`: 加写锁后**同步执行**
- `brokerNames = this.clusterAddrTable.get(clusterName)`: 获取当前集群上的 Broker 名称列表, 是空就新建列表
- `brokerNames.add(brokerName)`: 将当前 Broker 名字加入到集群列表
- `brokerData = this.brokerAddrTable.get(brokerName)`: 获取当前 Broker 的 brokerData, 是空就新建放入映射表
- `brokerAddrsMap = brokerData.getBrokerAddrs()`: 获取当前 Broker 的物理节点 map 表, 进行遍历, 如果物理节点角色发生变化 (slave → master), 先将旧数据从物理节点 map 中移除, 然后重写放入, **保证节点的唯一性**
- `if (null != topicConfigWrapper && MixAll.MASTER_ID == brokerId)`: Broker 上的 Topic 不为 null, 并且当前物理节点是 Broker 上的 master 节点
`tcTable = topicConfigWrapper.getTopicConfigTable()`: 获取当前 Broker 信息中的主题映射表
`if (tcTable != null)`: 映射表不空就加入或者更新到 Namesrv 内
- `prevBrokerLiveInfo = this.brokerLiveTable.put(brokerAddr)`: 添加**当前节点的 BrokerLiveInfo**, 返回上一次心跳时当前 Broker 节点的存活对象数据。NamesrvController 中的定时任务会扫描映射表 brokerLiveTable

```

BrokerLiveInfo prevBrokerLiveInfo = this.brokerLiveTable.put(brokerAddr, new
BrokerLiveInfo(
    System.currentTimeMillis(), topicConfigWrapper.getDataVersion(),
    channel, haServerAddr));

```

- `if (MixAll.MASTER_ID != brokerId)`: 当前 Broker 不是 master 节点, **获取主节点的信息设置到结果对象**
- `this.lock.writeLock().unlock()`: 释放写锁

Broker

MappedFile

成员属性

MappedFile 类是最基础的存储类，继承自 ReferenceResource 类，用来**保证线程安全**

MappedFile 类成员变量：

- 内存相关：

```
public static final int OS_PAGE_SIZE = 1024 * 4; // 内存页大小：默认是 4k
private AtomicLong TOTAL_MAPPED_VIRTUAL_MEMORY; // 当前进程中所有的 mappedFile 占用的总虚拟内存大小
private AtomicInteger TOTAL_MAPPED_FILES; // 当前进程中所有的 mappedFile 个数
```

- 数据位点：

```
protected final AtomicInteger wrotePosition; // 当前 mappedFile 的数据写入点
protected final AtomicInteger committedPosition; // 当前 mappedFile 的数据提交点
private final AtomicInteger flushedPosition; // 数据落盘位点，在这之前的数据是持久化的安全数据
// flushedPosition - wrotePosition 之间的数据属于脏页
```

- 文件相关：CL 是 CommitLog，CQ 是 ConsumeQueue

```
private String fileName; // 文件名称，CL和CQ文件名是【第一条消息的物理偏移量】，索引文件是【年月日时分秒】
private long fileFromOffset; // 文件名转long，代表该对象的【起始偏移量】
private File file; // 文件对象
```

MF 中以物理偏移量作为文件名，可以更好的寻址和进行判断

- 内存映射：

```
protected FileChannel fileChannel; // 文件通道
private MappedByteBuffer mappedByteBuffer; // 内存映射缓冲区，访问虚拟内存
```

ReferenceResource 类成员变量：

- 引用数量：当 `refCount <= 0` 时，表示该资源可以释放了，没有任何其他程序依赖它了，用原子类保证线程安全

```
protected final AtomicLong refCount = new AtomicLong(1); // 初始值为 1
```

- 存活状态：表示资源的存活状态

```
protected volatile boolean available = true;
```

- 是否清理：默认值 false，当执行完子类对象的 cleanup() 清理方法后，该值置为 true，表示资源已经全部释放

```
protected volatile boolean cleanupOver = false;
```

- 第一次关闭资源的时间：用来记录超时时间

```
private volatile long firstShutdownTimestamp = 0;
```

成员方法

MappedFile 类核心方法：

- appendMessage(): 提供上层向内存映射中追加消息的方法，消息如何追加由 AppendMessageCallback 控制

```
// 参数一：消息    参数二：追加消息回调
public AppendMessageResult appendMessage(MessageExtBrokerInner msg,
AppendMessageCallback cb)
```

```
// 将字节数组写入到文件通道
public boolean appendMessage(final byte[] data)
```

- flush(): 刷盘接口，参数 flushLeastPages 代表刷盘的最小页数，等于 0 时属于强制刷盘；> 0 时需要脏页（计算方法在数据位点）达到该值才进行物理刷盘；文件写满时强制刷盘

```
public int flush(final int flushLeastPages)
```

- selectMappedBuffer(): 该方法以 pos 为开始位点，到有效数据为止，创建一个切片 ByteBuffer 作为数据副本，供业务访问数据

```
public SelectMappedBufferResult selectMappedBuffer(int pos)
```

- destroy(): 销毁映射文件对象，并删除关联的系统文件，参数是强制关闭资源的时间

```
public boolean destroy(final long intervalForcibly)
```

- cleanup(): **释放堆外内存**，更新总虚拟内存和总内存映射文件数

```
public boolean cleanup(final long currentRef)
```

- warmMappedFile(): 内存预热，当要新建的 MappedFile 对象大于 1g 时执行该方法。 mappedByteBuffer 已经通过 mmap 映射，此时操作系统中只是记录了该文件和该 Buffer 的映射关系，而并没有映射到物理内存中，对该 MappedFile 的每个 Page Cache 进行写入一个字节分配内存，**将映射文件全部加载到内存**

```
public void warmMappedFile(FlushDiskType type, int pages)
```

- mlock(): 锁住指定的内存区域避免被操作系统调到 swap 空间，减少了缺页异常的产生

```
public void mlock()
```

swap space 是磁盘上的一块区域，可以是一个分区或者一个文件或者是组合。当系统物理内存不足时，Linux 会将内存中不常访问的数据保存到 swap 区域上，这样系统就可以有更多的物理内存为各个进程服务，而当系统需要访问 swap 上存储的内容时，需要通过 **缺页中断** 将 swap 上的数据加载到内存中

ReferenceResource 类核心方法：

- hold(): 增加引用计数 refCount，方法加锁

```
public synchronized boolean hold()
```

- shutdown(): 关闭资源，参数代表强制关闭资源的时间间隔

```
// 系统当前时间 - firstShutdownTimestamp 时间 > intervalForcibly 进行【强制关闭】
public void shutdown(final long intervalForcibly)
```

- release(): 引用计数减 1，当 refCount 为 0 时，调用子类的 cleanup 方法

```
public void release()
```

MapQueue

成员属性

MappedFileQueue 用来管理 MappedFile 文件

成员变量：

- 管理目录：CommitLog 是 `../store/commitlog`，ConsumeQueue 是 `../store/xxx_topic/0`

```
private final String storePath;
```

- 文件属性：

```
private final int mappedFileSize; // 目录下每个文件大小，CL文件默认 1g, CQ文件
默认 600w字节
private final CopyOnWriteArrayList<MappedFile> mappedFiles; // 目录下的每个
mappedFile 都加入该集合
```

- 数据位点:

```
private long flushedwhere = 0;      // 目录的刷盘位点, 值为 mf.fileName +
mf.wrotePosition
private long committedwhere = 0;    // 目录的提交位点
```

- 消息存储:

```
private volatile long storeTimestamp = 0; // 当前目录下最后一条 msg 的存储时间
```

- 创建服务: 新建 MappedFile 实例, 继承自 ServiceThread 是一个任务对象, run 方法用来创建实例

```
private final AllocateMappedFileService allocateMappedFileService;
```

成员方法

核心方法:

- load(): Broker 启动时, 加载本地磁盘数据, 该方法读取 storePath 目录下的文件, 创建 MappedFile 对象放入集合内

```
public boolean load()
```

- getLastMappedFile(): 获取当前正在顺序写入的 MappedFile 对象, 如果最后一个 MappedFile 写满了, 或者不存在 MappedFile 对象, 则创建新的 MappedFile

```
// 参数一: 文件起始偏移量; 参数二: 当list为空时, 是否新建 MappedFile
public MappedFile getLastMappedFile(final long startOffset, boolean
needCreate)
```

- flush(): 根据 flushedWhere 属性查找合适的 MappedFile, 调用该 MappedFile 的落盘方法, 并更新全局的 flushedWhere

```
//参数: 0 表示强制刷新, > 0 脏页数据必须达到 flushLeastPages 才刷新
public boolean flush(final int flushLeastPages)
```

- findMappedFileByOffset(): 根据偏移量查询对象

```
public MappedFile findMappedFileByOffset(final long offset, final boolean
returnFirstNotFound)
```

- deleteExpiredFileByTime(): CL 删除过期文件, 根据文件的保留时长决定是否删除

```
// 参数一：过期时间； 参数二：删除两个文件之间的时间间隔； 参数三：mf.destory传递的参数；  
参数四：true 强制删除  
public int deleteExpiredFileByTime(final long expiredTime, final int  
deleteFilesInterval, final long intervalForcibly, final boolean  
cleanImmediately)
```

- deleteExpiredFileByOffset(): CQ 删除过期文件，遍历每个 MF 文件，获取当前文件最后一个数据单元的物理偏移量，小于 offset 说明当前 MF 文件内都是过期数据

```
// 参数一：consumeLog 目录下最小物理偏移量，就是第一条消息的 offset;  
// 参数二：ConsumerQueue 文件内每个数据单元固定大小  
public int deleteExpiredFileByOffset(long offset, int unitsize)
```

CommitLog

成员属性

成员变量：

- 魔数：

```
public final static int MESSAGE_MAGIC_CODE = -626843481; // 消息的第一个字段  
是大小，第二个字段就是魔数  
protected final static int BLANK_MAGIC_CODE = -875286124; // 文件尾消息的魔法  
值
```

- MappedFileQueue：用于管理 `../store/commitlog` 目录下的文件

```
protected final MappedFileQueue mappedFileQueue;
```

- 存储服务：

```
protected final DefaultMessageStore defaultMessageStore; // 存储模块对象，上  
层服务  
private final FlushCommitLogService flushCommitLogService; // 刷盘服务，默认实  
现是异步刷盘
```

- 回调器：控制消息的哪些字段添加到 MappedFile

```
private final AppendMessageCallback appendMessageCallback;
```

- 队列偏移量字典表：key 是主题队列 id，value 是偏移量

```
protected HashMap<String, Long> topicQueueTable = new HashMap<String, Long>  
(1024);
```

- 锁相关:

```
private volatile long beginTimeInLock = 0;           // 写数据时加锁的开始时间
protected final PutMessageLock putMessageLock;       // 写锁, 两个实现类: 自旋锁和
重入锁
```

因为发送消息是需要持久化的, 在 Broker 端持久化时会获取该锁, **保证发送的消息的线程安全**

构造方法:

- 有参构造:

```
public CommitLog(final DefaultMessageStore defaultMessageStore) {
    // 创建 MappedFileQueue 对象
    // 参数1: ../store/commitlog; 参数2: 【1g】; 参数3:
    allocateMappedFileService
    this.mappedFileQueue = new MappedFileQueue(...);
    // 默认 异步刷盘, 创建这个对象
    this.flushCommitLogService = new FlushRealTimeService();
    // 控制消息哪些字段追加到 mappedFile, 【消息最大是 4M】
    this.appendMessageCallback = new DefaultAppendMessageCallback(...);
    // 默认使用自旋锁
    this.putMessageLock = ...;
}
```

成员方法

CommitLog 类核心方法:

- start(): 会启动刷盘服务

```
public void start()
```

- shutdown(): 关闭刷盘服务

```
public void shutdown()
```

- load(): 加载 CommitLog 目录下的文件

```
public boolean load()
```

- getMessage(): 根据 offset 查询单条信息, 返回的结果对象内部封装了一个 ByteBuffer, 该 Buffer 表示 [offset, offset + size] 区间的 MappedFile 的数据

```
public SelectMappedBufferResult getMessage(final long offset, final int
size)
```

- deleteExpiredFile(): 删除过期文件, 方法由 DefaultMessageStore 的定时任务调用

```
public int deleteExpiredFile()
```

- `asyncPutMessage()`: 存储消息

```
public CompletableFuture<PutMessageResult> asyncPutMessage(final MessageExtBrokerInner msg)
```

- `msg.setStoreTimestamp(System.currentTimeMillis())`: 设置存储时间，后面获取到写锁后这个事件会重写
 - `msg.setBodyCRC(UtilAll.crc32(msg.getBody()))`: 获取消息的 CRC 值
 - `topic、queueId`: 获取主题和队列 ID
 - `if (msg.getDelayTimeLevel() > 0)`: **获取消息的延迟级别，这里是延迟消息实现的关键**
 - `topic = TopicValidator.RMQ_SYS_SCHEDULE_TOPIC`: **修改消息的主题为 SCHEDULE_TOPIC_XXXX**
 - `queueId = ScheduleMessageService.delayLevel2QueueId()`: **队列 ID 为延迟级别 -1**
 - `MessageAccessor.putProperty`: **将原来的消息主题和 ID 存入消息的属性 REAL_TOPIC 中**
 - `mappedFile = this.mappedFileQueue.getLastMappedFile()`: 获取当前顺序写的 MappedFile 对象
 - `putMessageLock.lock()`: **获取写锁**
 - `msg.setStoreTimestamp(beginLockTimestamp)`: 设置消息的存储时间为获取锁的时间
 - `if (null == mappedFile || mappedFile.isFull())`: 文件写满了创建新的 MF 对象
 - `result = mappedFile.appendMessage(msg, this.appendMessageCallback)`: **消息追加，核心逻辑在回调器类**
 - `putMessageLock.unlock()`: 释放写锁
 - `this.defaultMessageStore.unlockMappedFile(..)`: 将 MappedByteBuffer 从 lock 切换为 unlock 状态
 - `putMessageResult = new PutMessageResult(PutMessageStatus.PUT_OK, result)`: 结果封装
 - `flushResultFuture = submitFlushRequest(result, msg)`: **唤醒刷盘线程**
 - `replicaResultFuture = submitReplicaRequest(result, msg)`: HA 消息同步
- `recoverNormally()`: 正常关机时的恢复方法，存储模块启动时**先恢复所有的 ConsumeQueue 数据，再恢复 CommitLog 数据**

```
// 参数表示恢复阶段 ConsumeQueue 中已知的最大的消息 offset
public void recoverNormally(Long maxPhyOffsetOfConsumeQueue)
```

- `int index = mappedFiles.size() - 3`: 从倒数第三个 file 开始向后恢复
- `dispatchRequest = this.checkMessageAndReturnsize()`: 每次从切片内解析出一条 msg 封装成 DispatchRequest 对象
- `size = dispatchRequest.getMsgSize()`: 获取消息的大小，检查 DispatchRequest 对象的状态
 情况 1：正常数据，则 `mappedFileOffset += size`
 情况 2：文件尾数据，处理下一个文件，`mappedFileOffset` 置为 0，`magic_code` 表示文件尾
- `processOffset += mappedFileOffset`: 计算出正确的数据存储位点，并设置 MappedFileQueue 的目录刷盘位点

- `this.mappedFileQueue.truncateDirtyFiles(processOffset)`: 调整 MFQ 中文件的刷盘位点
- `if (maxPhyOffsetOfConsumeQueue >= processOffset)`: 删除冗余数据, 将超过全局位点的 CQ 下的文件删除, 将包含全局位点的 CQ 下的文件重新定位
- `recoverAbnormally()`: 异常关机时的恢复方法

```
public void recoverAbnormally(long maxPhyOffsetOfConsumeQueue)
```

- `int index = mappedFiles.size() - 1`: 从尾部开始遍历 MFQ, 验证 MF 的第一条消息, 找到第一个验证通过的文件对象
- `dispatchRequest = this.checkMessageAndReturnSize()`: 每次解析出一条 msg 封装成 DispatchRequest 对象
- `this.defaultMessageStore.doDispatch(dispatchRequest)`: 重建 ConsumerQueue 和 Index, 避免上次异常停机导致 CQ 和 Index 与 CommitLog 不对齐
- 剩余逻辑与正常关机的恢复方法相似

服务线程

AppendMessageCallback 消息追加服务实现类为 DefaultAppendMessageCallback

- `doAppend()`:

```
public AppendMessageResult doAppend()
```

- `long wroteOffset = fileFromOffset + byteBuffer.position()`: 消息写入的位置, 物理偏移量 phyOffset
- `String msgId`: 消息 ID, 规则是客户端 IP + 消息偏移量 phyOffset
- `byte[] topicData`: 序列化消息, 将消息的字段压入到 msgStoreItemMemory 这个 Buffer 中
- `byteBuffer.put(this.msgStoreItemMemory.array(), 0, msgLen)`: 将 msgStoreItemMemory 中的数据写入 MF 对象的内存映射的 Buffer 中, 数据还没落盘
- `AppendMessageResult result`: 构造结果对象, 包括存储位点、是否成功、队列偏移量等信息
- `CommitLog.this.topicQueueTable.put(key, ++queueOffset)`: 更新队列偏移量

FlushRealTimeService 刷盘 CL 数据, 默认是异步刷盘类 FlushRealTimeService

- `run()`: 运行方法

```
public void run()
```

- `while (!this.isStopped())`: stopped 为 true 才跳出循环
- `boolean flushCommitLogTimed`: 控制线程的休眠方式, 默认是 false, 使用 `CountDownLatch.await()` 休眠, 设置为 true 时使用 `Thread.sleep()` 休眠
- `int interval`: 获取配置中的刷盘时间间隔

- `int flushPhysicQueueLeastPages`: 获取最小刷盘页数, 默认是 4 页, 脏页达到指定页数才刷盘
- `int flushPhysicQueueThoroughInterval`: 获取强制刷盘周期, 默认是 10 秒, 达到周期后强制刷盘, 不考虑脏页
- `if (flushCommitLogTimed)`: 休眠逻辑, 避免 CPU 占用太长时间, 导致无法执行其他更紧急的任务
- `CommitLog.this.mappedFileQueue.flush(flushPhysicQueueLeastPages)`: 刷盘
- `for (int i = 0; i < RETRY_TIMES_OVER && !result; i++)`: stopped 停止标记为 true 时, 需要确保所有的数据都已经刷盘, 所以此处尝试 10 次强制刷盘,
`result = CommitLog.this.mappedFileQueue.flush(0)`: 强制刷盘

同步刷盘类 GroupCommitService

- `run()`: 运行方法

```
public void run()
```

- `while (!this.isStopped())`: stopped 为 true 才跳出循环
`this.waitForRunning(10)`: 线程休眠 10 毫秒, 最后调用 `onwaitForEnd()` 进行请求的交换
`swapRequests()`
- `this.doCommit()`: 做提交逻辑
 - `if (!this.requestsRead.isEmpty())`: 读请求集合不为空
`for (GroupCommitRequest req : this.requestsRead)`: 遍历所有的读请求, 请求中的属性:
 - `private final long nextOffset`: 本条消息存储之后, 下一条消息开始的 offset
 - `private CompletableFuture<PutMessageStatus> flushOKFuture`: Future 对象
 - `boolean flushOK = ...`: 当前请求关注的数据是否全部落盘, 落盘成功唤醒消费者线程
`for (int i = 0; i < 2 && !flushOK; i++)`: 尝试进行两次强制刷盘, 保证刷盘成功
`CommitLog.this.mappedFileQueue.flush(0)`: 强制刷盘
`req.wakeupCustomer(flushOK ? ...)`: 设置 Future 结果, 在 Future 阻塞的线程在这里会被唤醒
`this.requestsRead.clear()`: 清理 requestsRead 列表, 方便交换时成为 requestsWrite 使用
 - `else`: 读请求集合为空
`CommitLog.this.mappedFileQueue.flush(0)`: 强制刷盘
- `this.swapRequests()`: 交换读写请求
- `this.doCommit()`: 交换后做一次提交

ConsQueue

成员属性

ConsumerQueue 是消息消费队列，存储消息在 CommitLog 的索引，便于快速定位消息

成员变量：

- 数据单元：ConsumerQueueData 数据单元的固定大小是 20 字节，默认申请 20 字节的缓冲区

```
public static final int CQ_STORE_UNIT_SIZE = 20;
```

- 文件管理：

```
private final MappedFileQueue mappedFileQueue; // 文件管理器，管理 CQ 目录下的文件
private final String storePath; // 目录，比如..../store/consumequeue/xxx_topic/
private final int mappedFileSize; // 每一个 CQ 存储文件大小，默认 20 * 30w = 600w byte
```

- 存储主模块：上层的对象

```
private final DefaultMessageStore defaultMessageStore;
```

- 消息属性：

```
private final String topic; // CQ 主题
private final int queueId; // CQ 队列，每一个队列都有一个 ConsumeQueue 对象进行管理
private final ByteBuffer byteBufferIndex; // 临时缓冲区，插新的 CQData 时使用
private long maxPhysicalOffset = -1; // 当前ConsumeQueue内存储的最大消息物理偏移量
private volatile long minLogicalOffset = 0; // 当前ConsumeQueue内存储的最小消息物理偏移量
```

构造方法：

- 有参构造：

```
public ConsumeQueue() {
    // 申请了一个 20 字节大小的 临时缓冲区
    this.byteBufferIndex = ByteBuffer.allocate(CQ_STORE_UNIT_SIZE);
}
```

成员方法

ConsumeQueue 启动阶段方法：

- load(): 第一步，加载 storePath 目录下的文件，初始化 MappedFileQueue
- recover(): 第二步，恢复 ConsumeQueue 数据
 - 从倒数第三个 MF 文件开始向后遍历，依次读取 MF 中 20 个字节的 CQData 数据，检查 offset 和 size 是否是有效数据
 - 找到无效的 CQData 的位点，该位点就是 CQ 的刷盘点和数据顺序写入点
 - 删除无效的 MF 文件，调整当前顺序写的 MF 文件的数据位点

其他方法：

- truncateDirtyLogicFiles(): CommitLog 恢复阶段调用，将 ConsumeQueue 有效数据文件与 CommitLog 对齐，将超出部分的数据文删除掉，并调整当前文件的数据位点。Broker 启动阶段先恢复 CQ 的数据，再恢复 CL 数据，但是**数据要以 CL 为基准**

```
// 参数是最大消息物理偏移量
public void truncateDirtyLogicFiles(long phyoffset)
```

- flush(): 刷盘，调用 MFQ 的刷盘方法

```
public boolean flush(final int flushLeastPages)
```

- deleteExpiredFile(): 删除过期文件，将小于 offset 的所有 MF 文件删除，offset 是 CommitLog 目录下最小的物理偏移量，小于该值的 CL 文件已经没有了，所以 CQ 也没有存在的必要

```
public int deleteExpiredFile(long offset)
```

- putMessagePositionInfoWrapper(): 向 CQ 中追加 CQData 数据，由存储主模块 DefaultMessageStore 内部的异步线程调用，负责构建 ConsumeQueue 文件和 Index 文件的，该线程会持续关注 CommitLog 文件，当 CommitLog 文件内有新数据写入，就读出来封装成 DispatchRequest 对象，转发给 ConsumeQueue 或者 IndexService

```
public void putMessagePositionInfoWrapper(DispatchRequest request)
```

- getIndexBuffer(): 转换 startIndex 为 offset，获取包含该 offset 的 MappedFile 文件，读取 [offset%maxsize, mfPos] 范围的数据，包装成结果对象返回

```
public SelectMappedBufferResult getIndexBuffer(final long startIndex)
```

IndexFile

成员属性

IndexFile 类成员属性

- 哈希:

```
private static int hashslotsize = 4;      // 每个 hash 桶的大小是 4 字节, 【用来存放索引的编号】  
private final int hashslotNum;           // hash 桶的个数, 默认 500 万
```

- 索引:

```
private static int indexSize = 20;        // 每个 index 条目的大小是 20 字节  
private static int invalidIndex = 0;       // 无效索引编号: 0 特殊值  
private final int indexNum;              // 默认值: 2000w  
private final IndexHeader indexHeader;    // 索引头
```

- 映射:

```
private final MappedFile mappedFile;       // 【索引文件使用的 MF 文件】  
private final FileChannel fileChannel;     // 文件通道  
private final MappedByteBuffer mappedByteBuffer; // 从 MF 中获取的内存映射缓冲区
```

构造方法:

- 有参构造

```
// endPhyOffset 上个索引文件 最后一条消息的 物理偏移量  
// endTimeStamp 上个索引文件 最后一条消息的 存储时间  
public IndexFile(final String fileName, final int hashslotNum, final int  
indexNum,  
                  final long endPhyOffset, final long endTimeStamp) throws  
IOException {  
    // 文件大小 40 + 500w * 4 + 2000w * 20  
    int fileTotalSize =  
        IndexHeader.INDEX_HEADER_SIZE + (hashslotNum * hashslotsize) +  
(indexNum * indexSize);  
    // 创建 mf 对象, 会在disk上创建文件  
    this.mappedFile = new MappedFile(fileName, fileTotalSize);  
    // 创建 索引头对象, 传递 索引文件mf 的切片数据  
    this.indexHeader = new IndexHeader(byteBuffer);  
    //...  
}
```

成员方法

IndexFile 类方法

- load(): 加载 IndexHeader

```
public void load()
```

- flush(): MappedByteBuffer 内的数据强制落盘

```
public void flush()
```

- isWriteFull(): 检查当前的 IndexFile 已写索引数是否 \geq indexNum, 达到该值则当前 IndexFile 不能继续追加 IndexData 了

```
public boolean iswriteFull()
```

- destroy(): 删除文件时使用的方法

```
public boolean destroy(final long intervalForcibly)
```

- putKey(): 添加索引数据, 解决哈希冲突使用头插法

```
// 参数一: 消息的 key, uniq_key 或者 keys="aaa bbb ccc" 会分别为 aaa bbb ccc 创建索引  
// 参数二: 消息的物理偏移量;    参数三: 消息存储时间  
public boolean putKey(final String key, final long phyoffset, final long storeTimestamp)
```

- `int slotPos = keyHash % this.hashslotNum`: 对 key 计算哈希后, 取模得到对应的哈希槽 slot 下标, 然后计算出哈希槽的存储位置 absSlotPos
 - `int slotValue = this.mappedByteBuffer.getInt(absslotPos)`: 获取槽中的值, 如果是无效值说明没有哈希冲突
 - `timeDiff = timeDiff / 1000`: 计算当前 msg 存储时间减去索引文件内第一条消息存储时间的差值, 转化为秒进行存储
 - `int absIndexPos`: 计算当前索引数据存储的位置, 开始填充索引数据到对应的位置
 - `this.mappedByteBuffer.putInt(absIndexPos + 4 + 8 + 4, slotValue)`: **hash 桶的原值, 头插法**
 - `this.mappedByteBuffer.putInt(absslotPos, this.indexHeader...)`: 在 slot 放入当前索引的索引编号
 - `if (this.indexHeader.getIndexCount() <= 1)`: 索引文件插入的第一条数据, 需要设置起始偏移量和存储时间
 - `if (invalidIndex == slotValue)`: 没有哈希冲突, 说明占用了一个新的 hash slot
 - `this.indexHeader`: 设置索引头的相关属性
- selectPhyOffset(): 从索引文件查询消息的物理偏移量

```
// 参数一: 查询结果全部放到该list内;  参数二: 查询key;  参数三: 结果最大数限制;  参数四  
五: 时间范围  
public void selectPhyoffset(final List<Long> phyOffsets, final String key,  
final int maxNum, final long begin, final long end, boolean lock)
```

- `if (this.mappedFile.hold())`: MF 的引用记数 +1, 查询期间 MF 资源**不能被释放**
 - `int slotvalue = this.mappedByteBuffer.getInt(absslotPos)`: 获取槽中的值, 可能是无效值或者索引编号, 如果是无效值说明查询未命中
 - `int absIndexPos`: 计算出索引编号对应索引数据的开始位点
 - `this.mappedByteBuffer`: 读取索引数据
 - `long timeRead = this.indexHeader.getBeginTimestamp() + timeDiff`: 计算出准确的存储时间
 - `boolean timeMatched = (timeRead >= begin) && (timeRead <= end)`: 时间范围的匹配
 - `phyOffsets.add(phyOffsetRead)`: 将命中的消息索引的消息偏移量加入到 list 集合中
 - `nextIndexToRead = prevIndexRead`: 遍历前驱节点
-

IndexServ

成员属性

IndexService 类用来管理 IndexFile 文件

成员变量:

- 存储主模块:

```
private final DefaultMessageStore defaultMessageStore;
```

- 索引文件存储目录: `../store/index`

```
private final String storePath;
```

- 索引对象集合: 目录下的每个文件都有一个 IndexFile 对象

```
private final ArrayList<IndexFile> indexFileList = new ArrayList<IndexFile>();
```

- 索引文件:

```
private final int hashSlotNum;           // 每个索引文件包含的 哈希桶数量 : 500w
private final int indexNum;             // 每个索引文件包含的 索引条目数量 : 2000w
```

成员方法

- `load()`: 加载 storePath 目录下的文件，为每个文件创建一个 `IndexFile` 实例对象，并加载 `IndexHeader` 信息

```
public boolean load(final boolean lastExitOK)
```

- `deleteExpiredFile()`: 删除过期索引文件

```
// 参数 offset 表示 CommitLog 内最早的消息的 phyOffset  
public void deleteExpiredFile(long offset)
```

- `this.readWriteLock.readLock().lock()`: 加锁判断
- `long endPhyOffset = this.indexFileList.get(0).getEndPhyOffset()`: 获取目录中第一个文件的结束偏移量
- `if (endPhyOffset < offset)`: 索引目录内存在过期的索引文件，并且当前的 `IndexFile` 都是过期的数据
- `for (int i = 0; i < (files.length - 1); i++)`: 遍历文件列表，删除过期的文件
- `buildIndex()`: 存储主模块 `DefaultMessageStore` 内部的异步线程调用，构建 `Index` 数据

```
public void buildIndex(DispatchRequest req)
```

- `indexFile = retryGetAndCreateIndexFile()`: 获取或者创建顺序写的索引文件对象
- `buildKey(topic, req.getUniqKey())`: 构建索引 key, `topic + # + uniqKey`
- `indexFile = putKey()`: 插入索引文件
- `if (keys != null && keys.length() > 0)`: 消息存在自定义索引 keys
`for (int i = 0; i < keyset.length; i++)`: 遍历每个索引，为每个 key 调用一次 `putKey`
- `getAndCreateLastIndexFile()`: 获取当前顺序写的 `IndexFile`，没有就创建

```
public IndexFile getAndCreateLastIndexFile()
```

HAService

HAService

Service

HAService 类成员变量：

- 主节点属性：

```
// master 节点当前有多少个 slave 节点与其进行数据同步
private final AtomicInteger connectionCount = new AtomicInteger(0);
// master 节点会给每个发起连接的 slave 节点的通道创建一个 HAConnection, 【控制
master 端向 slave 端传输数据】
private final List<HAConnection> connectionList = new LinkedList<>();
// master 向 slave 节点推送的最大的 offset, 表示数据同步的进度
private final AtomicLong push2SlaveMaxOffset = new AtomicLong(0)
```

- 内部类属性:

```
// 封装了绑定服务器指定端口, 监听 slave 的连接的逻辑, 没有使用 Netty, 使用了原生态的 NIO
去做
private final AcceptSocketService acceptSocketService;
// 控制生产者线程阻塞等待的逻辑
private final GroupTransferService groupTransferService;
// slave 节点的客户端对象, 【slave 端才会正常运行该实例】
private final HAclient haclient;
```

- 线程通信对象:

```
private final WaitNotifyObject waitNotifyObject = new WaitNotifyObject()
```

成员方法:

- start(): 启动高可用服务

```
public void start() throws Exception {
    // 监听从节点
    this.acceptSocketService.beginAccept();
    // 启动监听服务
    this.acceptSocketService.start();
    // 启动转移服务
    this.groupTransferService.start();
    // 启动从节点客户端实例
    this.haclient.start();
}
```

Accept

AcceptSocketService 类用于**监听从节点的连接**, 创建 HAConnection 连接对象

成员变量:

- 端口信息: Master 绑定监听的端口信息

```
private final SocketAddress socketAddressListen;
```

- 服务端通道:

```
private ServerSocketChannel serverSocketChannel;
```

- 多路复用器：

```
private Selector selector;
```

成员方法：

- beginAccept(): 开始监听连接, **NIO** 标准模板

```
public void beginAccept()
```

- run(): 服务启动

```
public void run()
```

- `this.selector.select(1000)`：多路复用器阻塞获取就绪的通道，最多等待 1 秒钟
- `Set<SelectionKey> selected = this.selector.selectedKeys()`：获取选择器中所有注册的通道中已经就绪好的事件
- `for (SelectionKey k : selected)`：遍历所有就绪的事件
- `if ((k.readyOps() & SelectionKey.OP_ACCEPT) != 0)`：说明 `OP_ACCEPT` 事件就绪
- `SocketChannel sc = ((ServerSocketChannel) k.channel()).accept()`：**获取到客户端连接的通道**
- `HAConnection conn = new HAConnection(HAService.this, sc)`：**为每个连接 master 服务器的 slave 创建连接对象**
- `conn.start()`：**启动 HAConnection 对象**，内部启动两个服务为读数据服务、写数据服务
- `HAService.this.addConnection(conn)`：加入到 HAConnection 集合内

Group

GroupTransferService 用来控制数据同步

成员方法：

- doWaitTransfer(): 等待主从数据同步

```
private void doWaitTransfer()
```

- `if (!this.requestsRead.isEmpty())`：读请求不为空
- `boolean transferOK = HAService.this.push2SlaveMaxOffset... >= req.getNextOffset()`：**主从同步是否完成**
- `req.wakeupCustomer(transferOK ? ...)`：唤醒消费者
- `this.requestsRead.clear()`：清空读请求

- swapRequests(): 交换读写请求

```
private void swapRequests()
```

HAClient

成员属性

HAClient 是 slave 端运行的代码，用于和 master 服务器建立长连接，上报本地同步进度，消费服务器发来的 msg 数据

成员变量：

- 缓冲区：

```
private static final int READ_MAX_BUFFER_SIZE = 1024 * 1024 * 4;      // 默认大  
小: 4 MB  
private ByteBuffer byteBufferRead =  
    ByteBuffer.allocate(READ_MAX_BUFFER_SIZE);  
private ByteBuffer byteBufferBackup =  
    ByteBuffer.allocate(READ_MAX_BUFFER_SIZE);
```

- 主节点地址：格式为 ip:port

```
private final AtomicReference<String> masterAddress = new AtomicReference<>  
(
```

- NIO 属性：

```
private final ByteBuffer reportOffset; // 通信使用NIO, 所以消息使用块传输, 上报  
slave offset 使用  
private SocketChannel socketChannel; // 客户端与 master 的会话通道  
private Selector selector;          // 多路复用器
```

- 通信时间：上次会话通信时间，用于控制 socketChannel 是否关闭的

```
private long lastwriteTimestamp = System.currentTimeMillis();
```

- 进度信息：

```
private long currentReportedOffset = 0; // slave 当前的进度信息  
private int dispatchPosition = 0;       // 控制 byteBufferRead position 指针
```

成员方法

- run(): 启动方法

```
public void run()
```

- if (this.connectMaster()): 连接主节点，连接失败会休眠 5 秒
 - String addr = this.masterAddress.get(): 获取 master 暴露的 HA 地址端口信息
 - this.socketChannel = RemotingUtil.connect(socketAddress): 建立连接
 - this.socketChannel.register(this.selector, SelectionKey.OP_READ): 注册到多路复用器，**关注读事件**
 - this.currentReportedOffset: 初始化上报进度字段为 slave 的 maxPhyOffset
 - if (this.isTimeToReportOffset()): slave 每 5 秒会上报一次 slave 端的同步进度信息给 master
 - boolean result = this.reportSlaveMaxOffset(): **上报同步信息**，上报失败关闭连接
 - this.selector.select(1000): 多路复用器阻塞获取就绪的通道，最多等待 1 秒钟，**获取到就绪事件或者超时后结束**
 - boolean ok = this.processReadEvent(): 处理读事件
 - if (!reportSlaveMaxOffsetPlus()): 检查是否重新上报同步进度
- reportSlaveMaxOffset(): 上报 slave 同步进度

```
private boolean reportSlaveMaxOffset(final long maxoffset)
```

- 首先向缓冲区写入 slave 端最大偏移量，写完以后切换为指定置为初始状态
 - for (int i = 0; i < 3 && this.reportOffset.hasRemaining(); i++): 尝试三次写数据
 - this.socketChannel.write(this.reportOffset): **写数据**
 - return !this.reportOffset.hasRemaining(): 写成功之后 pos = limit
- processReadEvent(): 处理 master 发送给 slave 数据，返回 true 表示处理成功 false 表示 Socket 处于半关闭状态，需要上层重建 haClient

```
private boolean processReadEvent()
```

- int readSizeZeroTimes = 0: 控制 while 循环的一个条件变量，当值为 3 时跳出循环
- while (this.byteBufferRead.hasRemaining()): byteBufferRead 有空间可以去 Socket 读缓冲区加载数据
- int readSize = this.socketChannel.read(this.byteBufferRead): **从通道读数据**
- if (readSize > 0): 加载成功，有新数据
 - readSizeZeroTimes = 0: 置为 0
- boolean result = this.dispatchReadRequest(): 处理数据的核心逻辑
- else if (readSize == 0): 连续无新数据 3 次，跳出循环
- else: readSize = -1 就表示 Socket 处于半关闭状态，对端已经关闭了

- dispatchReadRequest(): 处理数据的核心逻辑, master 与 slave 传输的数据格式 {[phyOffset] [size] [data...]} , phyOffset 表示数据区间的开始偏移量, data 代表数据块, 最大 32kb, 可能包含多条消息的数据

```
private boolean dispatchReadRequest()
```

- final int msgHeadersize = 8 + 4 : 协议头大小 12
- int readSocketPos = this.byteBufferRead.position() : 记录缓冲区处理数据前的 pos 位点, 用于恢复指针
- int diff = ... : 当前 byteBufferRead 还剩多少 byte 未处理, 每处理一条帧数据都会更新 dispatchPosition
- if (diff >= msgHeadersize) : 缓冲区还有完整的协议头 header 数据
- if (diff >= (msgHeadersize + bodysize)) : 说明**缓冲区内是包含当前帧的全部数据的**, 开始处理帧数据

HAService...appendToCommitLog(masterPhyOffset, bodyData) : 存储数据到 CommitLog, 并构建 Index 和 CQ

this.byteBufferRead.position(readSocketPos) : 恢复 byteBufferRead 的 pos 指针

this.dispatchPosition += msgHeadersize + bodysize : 加一帧数据长度, 处理下一条数据使用

if (!reportSlaveMaxOffsetPlus()) : 上报 slave 同步信息
- if (!this.byteBufferRead.hasRemaining()) : 缓冲区写满了, 重新分配缓冲区
- reallocateByteBuffer(): 重新分配缓冲区

```
private void reallocateByteBuffer()
```

- int remain = READ_MAX_BUFFER_SIZE - this.dispatchPosition : 表示缓冲区尚未处理过的字节数量
- if (remain > 0) : 条件成立, 说明缓冲区**最后一帧数据是半包数据**, 但是不能丢失数据

this.byteBufferBackup.put(this.byteBufferRead) : 将半包数据拷贝到 backup 缓冲区
- this.swapByteBuffer() : 交换 backup 成为 read
- this.byteBufferRead.position(remain) : 设置 pos 为 remain , 后续加载数据 pos 从 remain 开始向后移动
- this.dispatchPosition = 0 : 当前缓冲区交换之后, 相当于是一个全新的 byteBuffer, 所以分配指针归零

HAConn

Connection

HAConnection 类成员变量：

- 会话通道： master 和 slave 之间通信的 SocketChannel

```
private final SocketChannel socketChannel;
```

- 客户端地址：

```
private final String clientAddr;
```

- 服务类：

```
private WriteSocketService writeSocketService; // 写数据服务  
private ReadSocketService readSocketService; // 读数据服务
```

- 请求位点：在 slave 上报本地的进度之后被赋值，该值大于 0 后同步逻辑才会运行，master 如果不知道 slave 节点当前消息的存储进度，就无法给 slave 推送数据

```
private volatile long slaveRequestOffset = -1;
```

- 应答位点：保存最新的 slave 上报的 offset 信息，slaveAckOffset 之前的数据都可以认为 slave 已经同步完成

```
private volatile long slaveAckOffset = -1;
```

核心方法：

- 构造方法：

```
public HAConnection(final HAService haservice, final SocketChannel  
socketChannel) {  
    // 初始化一些东西  
    // 设置 socket 读写缓冲区为 64kb 大小  
    this.socketChannel.socket().setReceiveBufferSize(1024 * 64);  
    this.socketChannel.socket().setSendBufferSize(1024 * 64);  
    // 创建读写服务  
    this.writeSocketService = new WriteSocketService(this.socketChannel);  
    this.readSocketService = new ReadSocketService(this.socketChannel);  
    // 自增  
    this.haservice.getConnectionCount().incrementAndGet();  
}
```

- 启动方法：

```
public void start() {  
    this.readSocketService.start();  
    this.writeSocketService.start();  
}
```

ReadSocket

ReadSocketService 类是一个任务对象， slave 向 master 传输的帧格式为 `[long][long][long]`，上报的是 slave 本地的同步进度，同步进度是一个 long 值

成员变量：

- 读缓冲：

```
private static final int READ_MAX_BUFFER_SIZE = 1024 * 1024;      // 默认大小  
1MB  
private final ByteBuffer byteBufferRead =  
    ByteBuffer.allocate(READ_MAX_BUFFER_SIZE);
```

- NIO 属性：

```
private final Selector selector;          // 多路复用器  
private final SocketChannel socketChannel; // master 与 slave 之间的会话  
SocketChannel
```

- 处理位点：缓冲区处理位点

```
private int processPosition = 0;
```

- 上次读操作的时间：

```
private volatile long lastReadTimestamp = System.currentTimeMillis();
```

核心方法：

- 构造方法：

```
public ReadsocketService(final SocketChannel socketChannel)
```

- `this.socketChannel.register(this.selector, SelectionKey.OP_READ)`：通道注册到多路复用器，关注读事件
- `this.setDaemon(true)`：设置为守护线程

- 运行方法：

```
public void run()
```

- `this.selector.select(1000)`：多路复用器阻塞获取就绪的通道，最多等待 1 秒钟，获取到就绪事件或者超时后结束
- `boolean ok = this.processReadEvent()`：**读数据的核心方法**，返回 true 表示处理成功 false 表示 Socket 处于半关闭状态，需要上层重建 HAConnection 对象
 - `int readSizeZeroTimes = 0`：控制 while 循环，当连续从 Socket 读取失败 3 次（未加载到数据）跳出循环

- `if (!this.byteBufferRead.hasRemaining()):` byteBufferRead 已经全部使用完, 需要清理数据并更新位点
 - `while (this.byteBufferRead.hasRemaining()):` byteBufferRead 有空间可以去 Socket 读缓冲区加载数据
 - `int readSize = this.socketChannel.read(this.byteBufferRead):` 从通道读数据
 - `if (readSize > 0):` 加载成功, 有新数据
`if ((byteBufferRead.position() - processPosition) >= 8):` 缓冲区的可读数据最少包含一个数据帧
 - `int pos = ...:` 获取可读帧数据中最后一个完整的帧数据的位点, 后面的数据丢弃
 - `long readoffset = ...byteBufferRead.getLong(pos - 8):` 读取最后一帧数据, slave 端当前的同步进度信息
 - `this.processPosition = pos:` 更新处理位点
 - `HAConnection.this.slaveAckOffset = readoffset:` 更新应答位点
 - `if (HAConnection.this.slaveRequestOffset < 0):` 条件成立给 `slaveRequestOffset` 赋值
 - `HAConnection.notifyTransferSome(slaveAckOffset):` 唤醒阻塞的生产者线程
 - `else if (readsize == 0):` 读取 3 次无新数据跳出循环
 - `else:` `readSize = -1` 就表示 Socket 处于半关闭状态, 对端已经关闭了
 - `if (interval > 20):` 超过 20 秒未发生通信, 直接结束循环
-

WriteSocket

WriteSocketService 类是一个任务对象, master 向 slave 传输的数据帧格式为 `{[phyoffset][size][data...]}{[phyoffset][size][data...]}`

- phyOffset: 数据区间的开始偏移量, 并不表示某一条具体的消息, 表示的数据块开始的偏移量位置
- size: 同步的数据块的大小
- data: 数据块, 最大 32kb, 可能包含多条消息的数据

成员变量:

- 协议头:

```
private final int headersize = 8 + 4;           // 协议头大小: 12
private final ByteBuffer byteBufferHeader; // 帧头缓冲区
```

- NIO 属性:

```
private final Selector selector;           // 多路复用器
private final SocketChannel socketChannel; // master 与 slave 之间的会话
SocketChannel
```

- 处理位点：下一次传输同步数据的位置信息，master 给当前 slave 同步的位点

```
private long nextTransferFromWhere = -1;
```

- 上次写操作：

```
private boolean lastWriteOver = true; // 上一轮数据是否传输完毕
private long lastWriteTimestamp = System.currentTimeMillis(); // 上次写操作的时间
```

核心方法：

- 构造方法：

```
public WriteSocketService(final SocketChannel socketChannel)
```

- `this.socketChannel.register(this.selector, SelectionKey.OP_WRITE)`：通道注册到多路复用器，关注写事件
- `this.setDaemon(true)`：设置为守护线程

- 运行方法：

```
public void run()
```

- `this.selector.select(1000)`：多路复用器阻塞获取就绪的通道，最多等待 1 秒钟，获取到就绪事件或者超时后结束

- `if (-1 == HAConnection.this.slaveRequestOffset)`：等待 slave 同步完数据

- `if (-1 == this.nextTransferFromWhere)`：条件成立，需要初始化该变量

```
if (0 == HAConnection.this.slaveRequestOffset) : slave 是一个全新节点，从正在顺序写的 MF 开始同步数据
```

```
long masterOffset = ... : 获取 master 最大的 offset，并计算归属的 mappedFile 文件的开始 offset
```

```
this.nextTransferFromWhere = masterOffset : 赋值给下一次传输同步数据的位置信息
```

```
this.nextTransferFromWhere = HAConnection.this.slaveRequestOffset : 大部分情况走这个赋值逻辑
```

- `if (this.lastWriteOver)`：上一次待发送数据全部发送完成

```
if (interval > 5) : 超过 5 秒未同步数据，发送一个 header 心跳数据包，维持长连接
```

- `else`：上一轮的待发送数据未全部发送，需要同步数据到 slave 节点

- `SelectMappedBufferResult selectResult`：到 CommitLog 中查询 **nextTransferFromWhere** 开始位置的数据

- `if (size > 32k)`：一次最多同步 32k 数据

```
this.nextTransferFromWhere += size : 增加 size，下一轮传输跳过本帧数据
```

```
selectResult.getByteBuffer().limit(size) : 设置 byteBuffer 可访问数据区间为 [pos, size]
```

```
this.selectMappedBufferResult = selectResult : 待发送的数据
```

- `this.byteBufferHeader.put`: 构建帧头数据
- `this.lastwriteOver = this.transferData()`: 处理数据, 返回是否处理完成
- 同步方法: 同步数据到 slave 节点, 返回 true 表示本轮数据全部同步完成, false 表示本轮同步未完成 (Header 和 Body 其中一个未同步完成都会返回 false)

```
private boolean transferData()
```

- `int writeSizeZeroTimes= 0`: 控制 while 循环, 当写失败连续 3 次时, 跳出循环
- `while (this.byteBufferHeader.hasRemaining())`: 帧头数据缓冲区有待发送的数据
- `int writeSize = this.socketChannel.write(this.byteBufferHeader)`: 向通道写帧头数据
- `if (null == this.selectMappedBufferResult)`: 说明是心跳数据, 返回心跳数据是否发送完成
- `if (!this.byteBufferHeader.hasRemaining())`: Header 写成功之后, 才进行写 Body
- `while (this.selectMappedBufferResult.getByteBuffer().hasRemaining())`: 数据缓冲区有待发送的数据
- `int writeSize = this.socketChannel.write(this.selectMappedBufferResult...)`: 向通道写帧头数据
- `if (writeSize > 0)`: 写数据成功, 但是不代表 SMBR 中的数据全部写完成
- `boolean result`: 判断是否发送完成, 返回该值

MesStore

生命周期

DefaultMessageStore 类核心是整个存储服务的调度类

- 构造方法:

```
public DefaultMessageStore()
```

- `this.allocateMappedFileService.start()`: 启动创建 MappedFile 文件服务
- `this.indexService.start()`: 启动索引服务
- `load()`: 先加载 CommitLog, 再加载 ConsumeQueue, 最后加载 IndexFile, 加载完进入恢复阶段, 先恢复 CQ, 在恢复 CL

```
public boolean load()
```

- `start()`: 核心启动方法

```
public void start()
```

- `lock = lockFile.getChannel().tryLock(0, 1, false)`: 获取文件锁, 获取失败说明当前目录已经启动过 Broker
 - `long maxPhysicalPosInLogicQueue = commitLog.getMinOffset()`: 遍历全部的 CQ 对象, 获取 CQ 中消息的最大偏移量
 - `this.reputMessageService.start()`: 设置分发服务的分发位点, 启动**分发服务**, 构建 ConsumerQueue 和 IndexFile
 - `if (dispatchBehindBytes() <= 0)`: 线程等待分发服务将分发数据全部处理完毕
 - `this.recoverTopicQueueTable()`: 因为修改了 CQ 数据, 所以再次构建队列偏移量字段表
 - `this.haService.start()`: 启动 **HA 服务**
 - `this.handleScheduleMessageService()`: 启动**消息调度服务**
 - `this.flushConsumeQueueService.start()`: 启动 CQ **消费队列刷盘服务**
 - `this.commitLog.start()`: 启动 **CL 刷盘服务**
 - `this.storeStatsService.start()`: 启动状态存储服务
 - `this.createTempFile()`: 创建 AbortFile, 正常关机时 JVM HOOK 会删除该文件, **异常宕机时该文件不会删除**, 开机数据恢复阶段根据是否存在该文件, 执行不同的恢复策略
 - `this.addScheduleTask()`: 添加定时任务
 - `DefaultMessageStore.this.cleanFilesPeriodically()`: **定时清理过期文件**, 周期是 10 秒
 - `this.cleanCommitLogService.run()`: 启动清理过期的 CL 文件服务
 - `this.cleanConsumeQueueService.run()`: 启动清理过期的 CQ 文件服务
 - `DefaultMessageStore.this.checkSelf()`: 每 10 分钟进行健康检查
 - `DefaultMessageStore.this.cleanCommitLogService.isSpaceFull()`: **磁盘预警定时任务**, 每 10 秒一次
 - `if (physicRatio > this.diskSpaceWarningLevelRatio)`: 检查磁盘是否到达 waring 阈值, 默认 90%
 - `boolean diskok = ...runningFlags.getAndMakeDiskFull()`: 设置磁盘写满标记
 - `boolean diskok = ...this.runningFlags.getAndMakeDiskOK()`: 设置磁盘可写标记
 - `this.shutdown = false`: 刚启动, 设置为 false
- `shutdown()`: 关闭各种服务和线程资源, 设置存储模块状态为关闭状态

```
public void shutdown()
```

- `destroy()`: 销毁 Broker 的工作目录

```
public void destroy()
```

服务线程

ServiceThread 类被很多服务继承，本身是一个 Runnable 任务对象，继承者通过重写 run 方法来实现服务的逻辑

- run(): 一般实现方式

```
public void run() {  
    while (!this.isstopped()) {  
        // 业务逻辑  
    }  
}
```

通过参数 stopped 控制服务的停止，使用 volatile 修饰保证可见性

```
protected volatile boolean stopped = false
```

- shutdown(): 停止线程，首先设置 stopped 为 true，然后进行唤醒，默认不直接打断线程

```
public void shutdown()
```

- waitForRunning(): 挂起线程，设置唤醒标记 hasNotified 为 false

```
protected void waitForRunning(long interval)
```

- wakeup(): 唤醒线程，设置 hasNotified 为 true

```
public void wakeup()
```

构建服务

AllocateMappedFileService 创建 MappedFile 服务

- mmapOperation(): 核心服务

```
private boolean mmapOperation()
```

- req = this.requestQueue.take(): 从 requestQueue 阻塞队列（优先级）中获取 AllocateRequest 任务
- if (...isTransientStorePoolEnable()): 条件成立使用直接内存写入数据，从直接内存中 commit 到 FileChannel 中
- mappedFile = new MappedFile(req.getFilePath(), req.getFilesize()): 根据请求的路径和大小创建对象
- mappedFile.warmMappedFile(): 判断 mappedFile 大小，只有 CommitLog 才进行文件预热

- `req.setMappedFile(mappedFile)`：将创建好的 MF 对象的赋值给请求对象的成员属性
- `req.getCountDownLatch().countDown()`：唤醒请求的阻塞线程
- `putRequestAndReturnMappedFile()`: MappedFileQueue 中用来创建 MF 对象的方法

```
public MappedFile putRequestAndReturnMappedFile(String nextFilePath, String nextNextFilePath, int filesize)
```

- `AllocateRequest nextReq = new AllocateRequest(...)`：创建 nextFilePath 的 AllocateRequest 对象，放入请求列表和阻塞队列，然后创建 nextNextFilePath 的 AllocateRequest 对象，放入请求列表和阻塞队列
- `AllocateRequest result = this.requestTable.get(nextFilePath)`：从请求列表获取 nextFilePath 的请求对象
- `result.getCountDownLatch().await(...)`：线程挂起，直到超时或者 nextFilePath 对应的 MF 文件创建完成
- `return result.getMappedFile()`：返回创建好的 MF 文件对象

ReputMessageService 消息分发服务，用于构建 **ConsumerQueue** 和 **IndexFile** 文件

- `run()`: 循环执行 `doReput` 方法，所以发送的消息存储进 CL 就可以产生对应的 CQ，每执行一次线程休眠 1 毫秒

```
public void run()
```

- `doReput()`: 实现分发的核心逻辑

```
private void doReput()
```

- `for (boolean doNext = true; this.isCommitLogAvailable() && doNext;)`：循环遍历
- `SelectMappedBufferResult result`：从 CommitLog 拉取数据，数据范围 [reputFromOffset, 包含该偏移量的 MF 的最大 Pos]，封装成结果对象
- `DispatchRequest dispatchRequest`：从结果对象读取出一条 DispatchRequest 数据
- `DefaultMessageStore.this.doDispatch(dispatchRequest)`：将数据交给分发器进行分发，用于构建 CQ 和索引文件
- `this.reputFromOffset += size`：更新数据范围

刷盘服务

FlushConsumeQueueService 刷盘 CQ 数据

- `run()`: 每隔 1 秒执行一次刷盘服务，跳出循环后还会执行一次强制刷盘

```
public void run()
```

- `doFlush()`: 刷盘

```
private void doFlush(int retryTimes)
```

- `int flushConsumeQueueLeastPages`: 脏页阈值, 默认是 2
- `if (retryTimes == RETRY_TIMES_OVER)`: 重试次数是 3 时设置强制刷盘, 设置脏页阈值为 0
- `int flushConsumeQueueThoroughInterval`: 两次刷新的时间间隔超过 60 秒会强制刷盘
- `for (ConsumeQueue cq : maps.values())`: 遍历所有的 CQ, 进行刷盘
- `DefaultMessageStore.this.getStoreCheckpoint().flush()`: 强制刷盘时将 StoreCheckpoint 瞬时数据刷盘

FlushCommitLogService 刷盘 CL 数据, 默认是异步刷盘

- `run()`: 运行方法

```
public void run()
```

- `while (!this.isStopped())`: stopped 为 true 才跳出循环
- `boolean flushCommitLogTimed`: 控制线程的休眠方式, 默认是 false, 使用 `CountDownLatch.await()` 休眠, 设置为 true 时使用 `Thread.sleep()` 休眠
- `int interval`: 获取配置中的刷盘时间间隔
- `int flushPhysicQueueLeastPages`: 获取最小刷盘页数, 默认是 4 页, 脏页达到指定页数才刷盘
- `int flushPhysicQueueThoroughInterval`: 获取强制刷盘周期, 默认是 10 秒, 达到周期后强制刷盘, 不考虑脏页
- `if (flushCommitLogTimed)`: 休眠逻辑, 避免 CPU 占用太长时间, 导致无法执行其他更紧急的任务
- `CommitLog.this.mappedFileQueue.flush(flushPhysicQueueLeastPages)`: 刷盘
- `for (int i = 0; i < RETRY_TIMES_OVER && !result; i++)`: stopped 停止标记为 true 时, 需要确保所有的数据都已经刷盘, 所以此处尝试 10 次强制刷盘,
- `result = CommitLog.this.mappedFileQueue.flush(0)`: 强制刷盘

清理服务

CleanCommitLogService 清理过期的 CL 数据, 定时任务 10 秒调用一次, **先清理 CL, 再清理 CQ**, 因为 CQ 依赖于 CL 的数据

- `run()`: 运行方法

```
public void run()
```

- `deleteExpiredFiles()`: 删除过期 CL 文件

```
private void deleteExpiredFiles()
```

- `long fileReservedTime`: 默认 72, 代表文件的保留时间
- `boolean timeup = this.isTimeToDelete()`: 当前时间是否是凌晨 4 点

- boolean spacefull = this.isSpaceToDelete(): CL 或者 CQ 的目录磁盘使用率达到阈值标准 85%
- boolean manualDelete = this.manualDeleteFileSeveralTimes > 0: 手动删除文件
- fileReservedTime *= 60 * 60 * 1000: 默认保留 72 小时
- deleteCount = DefaultMessageStore.this.commitLog.deleteExpiredFile(): 调用 MFQ 对象的删除方法

CleanConsumeQueueService 清理过期的 CQ 数据

- run(): 运行方法

```
public void run()
```

- deleteExpiredFiles(): 删除过期 CQ 文件

```
private void deleteExpiredFiles()
```

- int deleteLogicsFilesInterval: 清理 CQ 的时间间隔, 默认 100 毫秒
- long minoffset = DefaultMessageStore.this.commitLog.getMinOffset(): 获取 CL 文件中最小的物理偏移量
- if (minOffset > this.lastPhysicalMinOffset): CL 最小的偏移量大于 CQ 最小的, 说明有过期数据
- this.lastPhysicalMinOffset = minoffset: 更新 CQ 的最小偏移量
- for (Consumequeue logic : maps.values()): 遍历所有的 CQ 文件
- logic.deleteExpiredFile(minoffset): 调用 MFQ 对象的删除方法
- DefaultMessageStore.this.indexService.deleteExpiredFile(minoffset): 删除过期的索引文件

获取消息

DefaultMessageStore#getMessage 用于获取消息, 在 PullMessageProcessor#processRequest 方法中被调用 (提示: 建议学习消费者源码时再阅读)

```
// offset: 客户端拉消息使用位点;      maxMsgNums: 32;    messageFilter: 一般这里是
tagCode 过滤
public GetMessageResult getMessage(final String group, final String topic, final
int queueId, final long offset, final int maxMsgNums, final MessageFilter
messageFilter)
```

- if (this.shutdown): 检查运行状态
- GetMessageResult getResult: 创建查询结果对象
- final long maxoffsetPy = this.commitLog.getMaxOffset(): 获取 CommitLog 最大物理偏移量
- Consumequeue consumeQueue = findConsumequeue(topic, queueId): 根据主题和队列 ID 获取 ConsumeQueue 对象

- `minOffset, maxOffset` : 获取当前 ConsumeQueue 的最小 offset 和最大 offset, 判断是否满足本次 Pull 的 offset

`if (maxOffset == 0)` : 说明队列内无数据, 设置状态为 NO_MESSAGE_IN_QUEUE, 外层进行长轮询

`else if (offset < minOffset)` : 说明 offset 太小了, 设置状态为 OFFSET_TOO_SMALL

`else if (offset == maxOffset)` : 消费进度持平, 设置状态为 OFFSET_OVERFLOW_ONE, 外层进行长轮询

`else if (offset > maxOffset)` : 说明 offset 越界了, 设置状态为
OFFSET_OVERFLOW_BADLY

- `selectMappedBufferResult bufferConsumeQueue` : 查询 CQData 获取包含该 offset 的 MappedFile 文件, 如果该文件不是顺序写的文件, 就读取 [offset%maxsize, 文件尾] 范围的数据, 反之读取 [offset%maxsize, 文件名+wrotePosition尾]

先查 CQ 的原因: 因为 CQ 时 CL 的索引, 通过 CQ 查询 CL 更加快捷

- `if (bufferConsumeQueue != null)` : 只有再 CQ 删除过期数据的逻辑执行时, 条件才不成立, 一般都是成立的
- `long nextPhyFileStartOffset = Long.MIN_VALUE` : 下一个 commitLog 物理文件名, 初始值为最小值
- `long maxPhyOffsetPulling = 0` : 本次拉消息最后一条消息的物理偏移量
- `for ()` : 处理数据, 每次处理 20 字节处理字节数大于 16000 时跳出循环
- `offsetPy, sizePy, tagsCode` : 读取 20 个字节后, 获取消息物理偏移量、消息大小、消息 tagCode
- `boolean isInDisk = checkInDiskByCommitOffset(...)` : 检查消息是热数据还是冷数据, false 为热数据
 - `long memory` : Broker 系统 40% 内存的字节数, 写数据时内存不够会使用 LRU 算法淘汰数据, 将淘汰数据持久化到磁盘
 - `return (maxOffsetPy - offsetPy) > memory` : 返回 true 说明数据已经持久化到磁盘, 为冷数据
- `if (this.isTheBatchFull())` : 控制是否跳出循环
 - `if (0 == bufferTotal || 0 == messageTotal)` : 本次 pull 消息未拉取到任何东西, 需要外层 for 循环继续, 返回 false
 - `if (maxMsgNums <= messageTotal)` : 结果对象内消息数已经超过了最大消息数量, 可以结束循环了
 - `if (isInDisk)` : 冷数据
 - `if ((bufferTotal + sizePy) > ...)` : 冷数据一次 pull 请求最大允许获取 64kb 的消息
 - `if (messageTotal > ...)` : 冷数据一次 pull 请求最大允许获取 8 条消息
 - `else` : 热数据
 - `if ((bufferTotal + sizePy) > ...)` : 热数据一次 pull 请求最大允许获取 256kb 的消息
 - `if (messageTotal > ...)` : 冷数据一次 pull 请求最大允许获取 32 条消息
- `if (messageFilter != null)` : 按照消息 tagCode 进行过滤
- `selectResult = this.commitLog.getMessage(offsetPy, sizePy)` : 根据 CQ 消息物理偏移量和消息大小到 commitLog 中查询这条 msg

- `if (null == selectResult)`: 条件成立说明 commitLog 执行了删除过期文件的定时任务, 因为是先清理的 CL, 所以 CQ 还有该索引数据
 - `nextPhyFileStartOffset = this.commitLog.rollNextFile(offsetPy)`: 获取包含该 offsetPy 的下一个数据文件的文件名
 - `getResult.addMessage(selectResult)`: 将本次循环查询出来的 msg 加入到 getResult 内
 - `status = GetMessageStatus.FOUND`: 查询状态设置为 FOUND
 - `nextPhyFileStartOffset = Long.MIN_VALUE`: 设置为最小值, 跳过期 CQData 数据的逻辑
 - `nextBeginOffset = offset + (i / ConsumeQueue.CQ_STORE_UNIT_SIZE)`: 计算客户端下一次 pull 时使用的位点信息
 - `getResult.setSuggestPullingFromSlave(diff > memory)`: 选择主从节点的建议
 - `diff > memory => true`: 表示本轮查询最后一条消息为冷数据, Broker 建议客户端下一次 pull 时到 slave 节点
 - `diff > memory => false`: 表示本轮查询最后一条消息为热数据, Broker 建议客户端下一次 pull 时到 master 节点
 - `getResult.setStatus(status)`: 设置结果状态
 - `getResult.setNextBeginOffset(nextBeginOffset)`: 设置客户端下一次 pull 时的 offset
 - `getResult.setMaxOffset(maxOffset)`: 设置 queue 的最大 offset 和最小 offset
 - `return getResult`: 返回结果对象
-

Broker

BrokerStartup 启动方法

```
public static void main(String[] args) {
    start(createBrokerController(args));
}
public static BrokerController start(BrokerController controller) {
    controller.start(); // 启动
}
```

BrokerStartup#createBrokerController: 构造控制器, 并初始化

- `final BrokerController controller()`: 创建实例对象
- `boolean initResult = controller.initialize()`: 控制器初始化
 - `this.registerProcessor()`: 注册了处理器, 包括发送消息、拉取消息、查询消息等核心处理器
 - `initialTransaction()`: 初始化了事务服务, 用于进行事务回查

BrokerController#start: 核心启动方法

- `this.messageStore.start()`: 启动存储服务
- `this.remotingServer.start()`: 启动 Netty 通信服务
- `this.filewatchservice.start()`: 启动文件监听服务

- `startProcessorByHa(messageStoreConfig.getBrokerRole())`: 启动事务回查
 - `this.scheduledExecutorService.scheduleAtFixedRate()`: 每隔 30s 向 NameServer 上报 Topic 路由信息, 心跳机制
`BrokerController.this.registerBrokerAll(true, false, brokerConfig.isForceRegister())`
-

Producer

生产者类

生产者类

DefaultMQProducer 是生产者的默认实现类

成员变量:

- 生产者实现类:

```
protected final transient DefaultMQProducerImpl defaultMQProducerImpl
```

- 生产者组: 发送事务消息, Broker 端进行事务回查 (补偿机制) 时, 选择当前生产者组的下一个生产者进行事务回查

```
private String producerGroup;
```

- 默认主题: isAutoCreateTopicEnable 开启时, 当发送消息指定的 Topic 在 Namesrv 未找到路由信息, 使用该值创建 Topic 信息

```
private String createTopicKey = TopicValidator.AUTO_CREATE_TOPIC_KEY_TOPIC;  
// 值为【TBW102】，Just for testing or demo program
```

- 消息重投: 系统特性消息重试部分详解了三个参数的作用

```
private int retryTimesWhenSendFailed = 2; // 同步发送失败后重试的发送次数,  
加上第一次发送, 一共三次  
private int retryTimesWhenSendAsyncFailed = 2; // 异步  
private boolean retryAnotherBrokerWhenNotStoreOK = false; // 消息未存储成功,  
选择其他 Broker 重试
```

- 消息队列:

```
private volatile int defaultTopicQueueNums = 4; // 默认 Broker 创建的队列数
```

- 消息属性:

```
private int sendMsgTimeout = 3000; // 发送消息的超时限制
private int compressMsgBodyOverHowmuch = 1024 * 4; // 压缩阈值, 当 msg body 超过 4k 后使用压缩
private int maxMessageSize = 1024 * 1024 * 4; // 消息体的最大限制, 默认 4M
private TraceDispatcher traceDispatcher = null; // 消息轨迹
```

构造方法:

- 构造方法:

```
public DefaultMQProducer(final String namespace, final String producerGroup,
    RPCHook rpcHook) {
    this.namespace = namespace;
    this.producerGroup = producerGroup;
    // 创建生产者实现对象
    defaultMQProducerImpl = new DefaultMQProducerImpl(this, rpcHook);
}
```

成员方法:

- start(): 启动方法

```
public void start() throws MQClientException {
    // 重置生产者组名, 如果传递了命名空间, 则 【namespace%group】
    this.setProducerGroup(withNamespace(this.producerGroup));
    // 生产者实现对象启动
    this.defaultMQProducerImpl.start();
    if (null != traceDispatcher) {
        // 消息轨迹的逻辑
        traceDispatcher.start(this.getNameServerAddr(),
            this.getAccessChannel());
    }
}
```

- send(): **发送消息**:

```
public SendResult send(Message msg){
    // 校验消息
    validators.checkMessage(msg, this);
    // 设置消息 Topic
    msg.setTopic(withNamespace(msg.getTopic()));
    return this.defaultMQProducerImpl.send(msg);
}
```

- request(): 请求方法, **需要消费者回执消息**

```
public Message request(final Message msg, final MessageQueue mq, final long
    timeout) {
    msg.setTopic(withNamespace(msg.getTopic()));
    return this.defaultMQProducerImpl.request(msg, mq, timeout);
}
```

实现者类

DefaultMQProducerImpl 类是默认的生产者实现类

成员变量：

- 实例对象：

```
private final DefaultMQProducer defaultMQProducer; // 持有默认生产者对象，用来获取对象中的配置信息  
private MQClientInstance mqClientFactory; // 客户端实例对象，生产者启动后需要注册到该客户端对象内
```

- 主题发布信息映射表：key 是 Topic, value 是发布信息

```
private final ConcurrentHashMap<String, TopicPublishInfo> topicPublishInfoTable  
= new ConcurrentHashMap<String, TopicPublishInfo>();
```

- 异步发送消息：相关信息

```
private final BlockingQueue<Runnable> asyncSenderThreadPoolQueue; // 异步发送消息，异步线程池使用的队列  
private final ExecutorService defaultAsyncSenderExecutor; // 异步发送消息默认使用的线程池  
private ExecutorService asyncSenderExecutor; // 异步消息发送线程池，指定后就不使用默认线程池了
```

- 定时器：执行定时任务

```
private final Timer timer = new Timer("RequestHouseKeepingService", true);  
// 守护线程
```

- 状态信息：服务的状态，默认创建状态

```
private ServiceState serviceState = ServiceState.CREATE_JUST;
```

- 压缩等级：ZIP 压缩算法的等级，默认是 5，越高压缩效果好，但是压缩的更慢

```
private int zipCompressLevel = Integer.parseInt(System.getProperty...,  
"5"));
```

- 容错策略：选择队列的容错策略

```
private MQFaultStrategy mqFaultStrategy = new MQFaultStrategy();
```

- 钩子：用来进行前置或者后置处理

```

ArrayList<SendMessageHook> sendMessageHookList;           // 发送消息的钩子，留给
用户扩展使用
ArrayList<CheckForbiddenHook> checkForbiddenHookList;   // 对比上面的钩子，可以
抛异常，控制消息是否可以发送
private final RPCHook rpcHook;                           // 传递给
NettyRemotingClient

```

构造方法：

- 默认构造：

```

public DefaultMQProducerImpl(final DefaultMQProducer defaultMQProducer) {
    // 默认 RPC HOOK 是空
    this(defaultMQProducer, null);
}

```

- 有参构造：

```

public DefaultMQProducerImpl(final DefaultMQProducer defaultMQProducer,
RPCHook rpcHook) {
    // 属性赋值
    this.defaultMQProducer = defaultMQProducer;
    this.rpcHook = rpcHook;

    // 创建【异步消息线程池任务队列】，长度是 5w
    this.asyncSenderThreadPoolQueue = new LinkedBlockingQueue<Runnable>
(50000);
    // 创建默认的异步消息任务线程池
    this.defaultAsyncSenderExecutor = new ThreadPoolExecutor(
        // 核心线程数和最大线程数都是 系统可用的计算资源（8核16线程的系统就是 16）...
)
}

```

实现方法

- start(): 启动方法，参数默认是 true，代表正常的启动路径

```

public void start(final boolean startFactory)

```

- `this.serviceState = ServiceState.START_FAILED`: 先修改为启动失败，成功后再修改，这种思想很常见
- `this.checkConfig()`: 判断生产者组名不能是空，也不能是 default_PRODUCER
- `if (!getProducerGroup().equals(MixAll.CLIENT_INNER_PRODUCER_GROUP))`: 条件成立说明当前生产者不是内部产生者，内部生产者是处理消息回退的这种情况使用的生产者
`this.defaultMQProducer.changeInstanceNameToPID()`: 修改生产者实例名称为当前进程的 PID

- o `this.mQclientFactory = ...`: 获取当前进程的 MQ 客户端实例对象，从 factoryTable 中获取 key 为 客户端 ID，格式是 `ip@pid`，一个 JVM 进程只有一个 PID，也只有一个 **MQClientInstance**
- o `boolean registerOK = mQclientFactory.registerProducer(...)`: 将生产者注册到 RocketMQ 客户端实例内
- o `this.topicPublishInfoTable.put(...)`: 添加一个主题发布信息，key 是 **TBW102**，value 是一个空对象
- o `mQclientFactory.start()`: 启动 RocketMQ 客户端实例对象
- o `this.mQclientFactory.sendHeartbeatToAllBrokerwithLock()`: RocketMQ **客户端实例向已知的 Broker 节点发送一次心跳** (也是定时任务)
- o `this.timer.scheduleAtFixedRate()`: request 发送的消息需要消费着回执信息，启动定时任务每秒一次删除超时请求
 - 生产者 msg 添加信息关联 ID 发送到 Broker
 - 消费者从 Broker 拿到消息后会检查 msg 类型是一个需要回执的消息，处理完消息后会根据 msg 关联 ID 和客户端 ID 生成一条响应结果消息发送到 Broker，Broker 判断为回执消息，会根据客户端ID 找到 channel 推送给生产者
 - 生产者拿到回执消息后，读取出来关联 ID 找到对应的 RequestFuture，将阻塞线程唤醒
- `sendDefaultImpl()`: 发送消息

```
//参数1: 消息; 参数2: 发送模式 (同步异步单向); 参数3: 回调函数, 异步发送时需要; 参数4: 发送超时时间, 默认 3 秒
private SendResult sendDefaultImpl(Msg msg, CommunicationMode communicationMode,
sendCallback, timeout) {}
```

- o `this.makeSureStateOK()`: 校验生产者状态是运行中，否则抛出异常
- o `topicPublishInfo = this.tryToFindTopicPublishInfo(msg.getTopic())`: **获取当前消息主题的发布信息**
 - `this.topicPublishInfoTable.get(topic)`: 先尝试从本地主题发布信息映射表获取信息，获取不到继续执行
 - `this.mQclientFactory.update...FromNameServer(topic)`: 然后从 Namesrv 更新该 Topic 的路由数据
 - `this.mQclientFactory.update...FromNameServer(...)`: **路由数据是空，获取默认 TBW102 的数据**

`return topicPublishInfo`: 返回 TBW102 主题的发布信息
- o `String[] brokersSent = new String[timesTotal]`: 下标索引代表第几次发送，值代表这次发送选择 Broker name
- o `for (; times < timesTotal; times++)`: 循环发送，**发送成功或者发送尝试次数达到上限，结束循环**
- o `String lastBrokerName = null == mq ? null : mq.getBrokerName()`: 获取上次发送失败的 BrokerName
- o `mqSelected = this.selectOneMessageQueue(topicPublishInfo, lastBrokerName)`: 从发布信息中选择一个队列，生产者的**负载均衡策略**，参考系统特性章节
- o `brokersSent[times] = mq.getBrokerName()`: 将本次选择的 BrokerName 存入数组

- o `msg.setTopic(this.defaultMQProducer.withNamespace(msg.getTopic()))`: 产生重投, 重投消息需要加上标记
- o `sendResult = this.sendKernelImpl`: 核心发送方法
- o `switch (communicationMode)`: 异步或者单向消息直接返回 null, 异步通过回调函数处理, 同步发送进入逻辑判断
`if (sendResult.getSendStatus() != SendStatus.SEND_OK)`: 服务端 Broker 存储失败, 需要重试其他 Broker
- o `throw new MQClientException()`: 未找到当前主题的路由数据, 无法发送消息, 抛出异常
- `sendKernelImpl()`: 核心发送方法

```
//参数1: 消息; 参数2: 选择的队列; 参数3: 发送模式(同步异步单向); 参数4: 回调函数, 异步发送时需要; 参数5: 主题发布信息; 参数6: 剩余超时时间限制
private SendResult sendKernelImpl(Msg msg, mq, communicationMode, sendcallback, topicPublishInfo, timeout)
```

- o `brokerAddr = this.mQClientFactory(...)`: 获取指定 BrokerName 对应的 master 节点的地址, master 节点的 ID 为 0, 集群模式下, 发送消息要发到主节点
- o `brokerAddr = MixAll.brokerVIPChannel()`: Broker 启动时会绑定两个服务器端口, 一个是普通端口, 一个是 VIP 端口, 服务器端根据不同端口创建不同的 NioSocketChannel
- o `byte[] prevBody = msg.getBody()`: 获取消息体
- o `if (!(msg instanceof MessageBatch))`: 非批量消息, 需要重新设置消息 ID
`MessageClientIDSetter.setUniqID(msg)`: msg id 由两部分组成, 一部分是 ip 地址、进程号、ClassLoader 的 hashCode, 另一部分是时间差(当前时间减去当月一号的时间)和计数器的值
- o `if (this.tryToCompressMessage(msg))`: 判断消息是否压缩, 压缩需要设置压缩标记
- o `hasCheckForbiddenHook, hasSendMessageHook`: 执行钩子方法
- o `requestHeader = new SendMessageRequestHeader()`: 设置发送消息的消息头
- o `if (requestHeader.getTopic().startsWith(MixAll.RETRY_GROUP_TOPIC_PREFIX))`: 重投的发送消息
- o `switch (communicationMode)`: 异步发送一种处理方式, 单向和同步同样的处理逻辑
`sendResult = this.mQClientFactory.getMQClientAPIImpl().sendMessage()`: 发送消息
 - `request = RemotingCommand.createRequestCommand()`: 创建一个 RequestCommand 对象
 - `request.setBody(msg.getBody())`: 将消息放入请求体
 - `switch (communicationMode)`: 根据不同的模式 invoke 不同的方法
- `request()`: 请求方法, 消费者回执消息, 这种消息是异步消息
 - o `requestResponseFuture = new RequestResponseFuture(correlationId, timeout, null)`: 创建请求响应对象
 - o `getRequestFutureTable().put(correlationId, requestResponseFuture)`: 放入 RequestFutureTable 映射表中

- `this.sendDefaultImpl(msg, CommunicationMode.ASYNC, new SendCallback())`: 发送异步消息，有回调函数
- `return waitResponse(msg, timeout, requestResponseFuture, cost)`: 用来挂起请求的方法

```
public Message waitResponseMessage(final long timeout) throws
InterruptedException {
    // 请求挂起
    this.countDownLatch.await(timeout, TimeUnit.MILLISECONDS);
    return this.responseMsg;
}
```

- 当消息被消费后，客户端处理响应时通过消息的关联 ID，从映射表中获取消息的 RequestResponseFuture，执行下面的方法唤醒挂起线程

```
public void putResponseMessage(final Message responseMsg) {
    this.responseMsg = responseMsg;
    this.countDownLatch.countDown();
}
```

路由信息

TopicPublishInfo 类用来存储路由信息

成员变量：

- 顺序消息：

```
private boolean orderTopic = false;
```

- 消息队列：

```
private List<MessageQueue> messageQueueList = new ArrayList<>();
// 主题全部的消息队列
private volatile ThreadLocalIndex sendwhichQueue = new ThreadLocalIndex();
// 消息队列索引
```

```
// 【消息队列类】
public class MessageQueue implements Comparable<MessageQueue>, Serializable {
    private String topic;
    private String brokerName;
    private int queueId;// 队列 ID
}
```

- 路由数据：主题对应的路由数据

```
private TopicRouteData topicRouteData;
```

```
public class TopicRouteData extends RemotingSerializable {
    private String orderTopicConf;
    private List<QueueData> queueDatas;      // 队列数据
    private List<BrokerData> brokerDatas;    // Broker 数据
    private HashMap<String/* brokerAddr */, List<String> /* Filter Server */>
filterServerTable;
}
```

```
public class QueueData implements Comparable<QueueData> {
    private String brokerName; // 节点名称
    private int readQueueNums; // 读队列数
    private int writeQueueNums; // 写队列数
    private int perm;          // 权限
    private int topicSynFlag;
}
```

```
public class BrokerData implements Comparable<BrokerData> {
    private String cluster;     // 集群名
    private String brokerName; // Broker节点名称
    private HashMap<Long/* brokerId */, String/* broker address */>
brokerAddrs;
}
```

核心方法：

- selectOneMessageQueue(): **选择消息队列**使用

```
// 参数是上次失败时的 brokerName，可以为 null
public MessageQueue selectOneMessageQueue(final String lastBrokerName) {
    if (lastBrokerName == null) {
        return selectOneMessageQueue();
    } else {
        // 遍历消息队列
        for (int i = 0; i < this.messageQueueList.size(); i++) {
            // 【获取队列的索引, +1】
            int index = this.sendWhichQueue.getAndIncrement();
            // 获取队列的下标位置
            int pos = Math.abs(index) % this.messageQueueList.size();
            if (pos < 0)
                pos = 0;
            // 获取消息队列
            MessageQueue mq = this.messageQueueList.get(pos);
            // 与上次选择的不同就可以返回
            if (!mq.getBrokerName().equals(lastBrokerName)) {
                return mq;
            }
        }
    }
    return selectOneMessageQueue();
}
```

公共配置

公共的配置信息类

- ClientConfig 类

```
public class ClientConfig {  
    // Namesrv 地址配置  
    private String namesrvAddr =  
        NameServerAddressutils.getNameServerAddresses();  
    // 客户端的 IP 地址  
    private String clientIP = Remotingutil.getLocalAddress();  
    // 客户端实例名称  
    private String instanceName = System.getProperty("rocketmq.client.name",  
        "DEFAULT");  
    // 客户端回调线程池的数量，平台核心数，8核16线程的电脑返回16  
    private int clientCallbackExecutorThreads =  
        Runtime.getRuntime().availableProcessors();  
    // 命名空间  
    protected String namespace;  
    protected AccessChannel accessChannel = AccessChannel.LOCAL;  
  
    // 获取路由信息的间隔时间 30s  
    private int pollNameServerInterval = 1000 * 30;  
    // 客户端与 broker 之间的心跳周期 30s  
    private int heartbeatBrokerInterval = 1000 * 30;  
    // 消费者持久化消费的周期 5s  
    private int persistConsumerOffsetInterval = 1000 * 5;  
    private long pullTimeDelayMilliswhenException = 1000;  
    private boolean unitMode = false;  
    private String unitName;  
    // vip 通道，broker 启动时绑定两个端口，其中一个是 vip 通道  
    private boolean vipChannelEnabled = Boolean.parseBoolean();  
    // 语言，默认是 Java  
    private LanguageCode language = LanguageCode.JAVA;  
}
```

- NettyClientConfig

```
public class NettyClientConfig {  
    // 客户端工作线程数  
    private int clientworkerThreads = 4;  
    // 回调处理线程池 线程数：平台核心数  
    private int clientCallbackExecutorThreads =  
        Runtime.getRuntime().availableProcessors();  
    // 单向请求并发数，默认 65535  
    private int clientOnewaySemaphoreValue =  
        NettySystemConfig.CLIENT_ONEWAY_SEMAPHORE_VALUE;  
    // 异步请求并发数，默认 65535
```

```

    private int clientAsyncSemaphoreValue =
NettySystemConfig.CLIENT_ASYNC_SEMAPHORE_VALUE;
    // 客户端连接服务器的超时时间限制 3秒
    private int connectTimeoutMillis = 3000;
    // 客户端未激活周期, 60s (指定时间内 ch 未激活, 需要关闭)
    private long channelNotActiveInterval = 1000 * 60;
    // 客户端与服务器 ch 最大空闲时间 2分钟
    private int clientChannelMaxIdleTimeSeconds = 120;

    // 底层 Socket 写和收 缓冲区的大小 65535 64k
    private int clientSocketSndBufSize = NettySystemConfig.socketSndbufSize;
    private int clientSocketRcvBufSize = NettySystemConfig.socketRcvbufSize;
    // 客户端 netty 是否启动内存池
    private boolean clientPooledByteBufAllocatorEnable = false;
    // 客户端是否超时关闭 Socket 连接
    private boolean clientCloseSocketIfTimeout = false;
}

```

客户端类

成员属性

MQClientInstance 是 RocketMQ 客户端实例，在一个 JVM 进程中只有一个客户端实例，**既服务于生产者，也服务于消费者**

成员变量：

- 配置信息：

```

private final int instanceIndex;           // 索引一般是 0, 因为客户端实例一般都
是一个进程只有一个
private final String clientId;            // 客户端 ID ip@pid
private final long bootTimestamp;          // 客户端的启动时间
private ServiceState serviceState;        // 客户端状态

```

- 生产者消费者的映射表：key 是组名

```

private final ConcurrentMap<String, MQProducerInner> producerTable
private final ConcurrentMap<String, MQConsumerInner> consumerTable
private final ConcurrentMap<String, MQAdminExtInner> adminExtTable

```

- 网络层配置：

```
private final NettyClientConfig nettyClientConfig;
```

- 核心功能的实现：负责将 MQ 业务层的数据转换为网络层的 RemotingCommand 对象，使用内部持有的 NettyRemotingClient 对象的 invoke 系列方法，完成网络 IO（同步、异步、单向）

```
private final MQClientAPIImpl mqClientAPIImpl;
```

- 本地路由数据：key 是主题名称，value 路由信息

```
private final ConcurrentHashMap<String, TopicRouteData> topicRouteTable = new
ConcurrentHashMap<>();
```

- 锁信息：两把锁，锁不同的数据

```
private final Lock lockNamesrv = new ReentrantLock();
private final Lock lockHeartbeat = new ReentrantLock();
```

- 调度线程池：单线程，执行定时任务

```
private final ScheduledExecutorService scheduledExecutorService;
```

- Broker 映射表：key 是 BrokerName

```
// 物理节点映射表，value: Long 是 brokerID，【ID=0 的是主节点，其他是从节点】，String
是地址 ip:port
private final ConcurrentHashMap<String, HashMap<Long, String>> brokerAddrTable;
// 物理节点版本映射表，String 是地址 ip:port，Integer 是版本
ConcurrentMap<String, HashMap<String, Integer>> brokerVersionTable;
```

- 客户端的协议处理器**：用于处理 IO 事件

```
private final ClientRemotingProcessor clientRemotingProcessor;
```

- 消息服务：

```
private final PullMessageService pullMessageService;           // 拉消息服务
private final RebalanceService rebalanceService;             // 消费者负载均衡服
务
private final ConsumerStatsManager consumerStatsManager;    // 消费者状态管理
```

- 内部生产者实例：处理消费端**消息回退**，用该生产者发送回退消息

```
private final DefaultMQProducer defaultMQProducer;
```

- 心跳次数统计：

```
private final AtomicLong sendHeartbeatTimesTotal = new AtomicLong(0)
```

构造方法：

- MQClientInstance 有参构造：

```
public MQClientInstance(ClientConfig clientConfig, int instanceIndex, String
clientId, RPCHook rpcHook) {
    this.clientConfig = clientConfig;
    this.instanceIndex = instanceIndex;
    // Netty 相关的配置信息
    this.nettyClientConfig = new NettyClientConfig();
```

```

// 平台核心数
this.nettyClientConfig.setClientCallbackExecutorThreads(...);
this.nettyClientConfig.setUseTLS(clientConfig.isUseTLS());
// 【创建客户端协议处理器】
this.clientRemotingProcessor = new ClientRemotingProcessor(this);
// 创建 API 实现对象
// 参数一：客户端网络配置
// 参数二：客户端协议处理器，注册到客户端网络层
// 参数三：rpcHook，注册到客户端网络层
// 参数四：客户端配置
this.mQClientAPIImpl = new MQClientAPIImpl(this.nettyClientConfig,
this.clientRemotingProcessor, rpcHook, clientConfig);

//...
// 内部生产者，指定内部生产者的组
this.defaultMQProducer = new
DefaultMQProducer(MixAll.CLIENT_INNER_PRODUCER_GROUP);
}

```

- MQClientAPIImpl 有参构造：

```

public MQClientAPIImpl(nettyClientConfig, clientRemotingProcessor, rpcHook,
clientConfig) {
    this.clientConfig = clientConfig;
    topAddressing = new TopAddressing(MixAll.getWSAddr(),
clientConfig.getUnitName());
    // 创建网络层对象，参数二为 null 说明客户端并不关心 channel event
    this.remotingClient = new NettyRemotingClient(nettyClientConfig, null);
    // 业务处理器
    this.clientRemotingProcessor = clientRemotingProcessor;
    // 注册 RpcHook
    this.remotingClient.registerRPCHook(rpcHook);
    // ...
    // 注册回退消息的请求码

    this.remotingClient.registerProcessor(RequestCode.PUSH_REPLY_MESSAGE_TO_CLIENT,
this.clientRemotingProcessor, null);
}

```

成员方法

- start(): 启动方法
 - synchronized (this)：加锁保证线程安全，保证只有一个实例对象启动
 - this.mQClientAPIImpl.start()：启动客户端网络层，底层调用 RemotingClient 类
 - this.startScheduledTask()：启动定时任务
 - this.pullMessageService.start()：启动拉取消息服务
 - this.rebalanceService.start()：启动负载均衡服务
 - this.defaultMQProducer...start(false)：启动内部生产者，参数为 false 代表不启动实例

- startScheduledTask(): 启动定时任务，调度线程池是单线程
 - `if (null == this.clientConfig.getNamesrvAddr())`: Namesrv 地址是空，需要两分钟拉取一次 Namesrv 地址
 - 定时任务 1: 从 Namesrv 更新客户端本地的路由数据，周期 30 秒一次

```
// 获取生产者和消费者订阅的主题集合，遍历集合，对比从 namesrv 拉取最新的主题路由数据
// 和本地数据，是否需要更新
MQClientInstance.this.updateTopicRouteInfoFromNameServer();
```

- 定时任务 2: 周期 30 秒一次，两个任务
 - 清理下线的 Broker 节点，遍历客户端的 Broker 物理节点映射表，将所有主题数据都不包含的 Broker 物理节点清理掉，如果被清理的 Broker 下所有的物理节点都没有了，就将该 Broker 的映射数据删除掉
 - 向在线的所有 Broker 发送心跳数据，同步发送的方式，返回值是 Broker 物理节点的版本号，更新版本映射表

```
MQClientInstance.this.cleanOfflineBroker();
MQClientInstance.this.sendHeartbeatToAllBrokerWithLock();
```

```
// 心跳数据
public class HeartbeatData extends RemotingSerializable {
    // 客户端 ID ip@pid
    private String clientID;
    // 存储客户端所有生产者数据
    private Set<ProducerData> producerDataSet = new
    HashSet<ProducerData>();
    // 存储客户端所有消费者数据
    private Set<ConsumerData> consumerDataSet = new
    HashSet<ConsumerData>();
}
```

- 定时任务 3: 消费者持久化消费数据，周期 5 秒一次

```
MQClientInstance.this.persistAllConsumerOffset();
```

- 定时任务 4: 动态调整消费者线程池，周期 1 分钟一次

```
MQClientInstance.this.adjustThreadPool();
```

- updateTopicRouteInfoFromNameServer(): 更新路由数据，通过加锁保证当前实例只有一个线程去更新
 - `if (isDefault && defaultMQProducer != null)`: 需要默认数据

`topicRouteData = ...getDefaultTopicRouteInfoFromNameServer()`: 从 Namesrv 获得默认的 TBW102 的路由数据
 - `topicRouteData = ...getTopicRouteInfoFromNameServer(topic)`: 需要从 Namesrv 获得路由数据（同步）
 - `old = this.topicRouteTable.get(topic)`: 获取客户端实例本地的该主题的路由数据

- o `boolean changed = topicRouteDataIsChange(old, topicRouteData)`: 对比本地和最新下拉的数据是否一致
 - o `if (changed)`: 不一致进入更新逻辑
 - `this.brokerAddrTable.put(...)`: 更新客户端 broker 物理节点映射表
 - `Update Pub info`: 更新生产者信息
 - `publishInfo = topicRouteData2TopicPublishInfo(topic, topicRouteData)`: 将主题路由数据转化为发布数据，会创建消息队列 MQ，放入发布数据对象的集合中
 - `impl.updateTopicPublishInfo(topic, publishInfo)`: 生产者将主题的发布数据保存到它本地，方便发送消息使用
 - `update sub info`: 更新消费者信息，创建 MQ 队列，更新订阅信息，用于负载均衡
 - `this.topicRouteTable.put(topic, cloneTopicRouteData)`: 将数据放入本地路由表
-

网络通信

成员属性

NettyRemotingClient 类负责客户端的网络通信

成员变量：

- Netty 服务相关属性：

```
private final NettyClientConfig nettyClientConfig;           // 客户端的网络层配置
private final Bootstrap bootstrap = new Bootstrap();         // 客户端网络层启动对象
private final EventLoopGroup eventLoopGroupWorker;          // 客户端网络层 Netty IO 线程组
```

- Channel 映射表：

```
private final ConcurrentMap<String, ChannelWrapper> channelTables; // key 是服务器的地址，value 是通道对象
private final Lock lockChannelTables = new ReentrantLock();           // 锁，控制并发安全
```

- 定时器：启动定时任务

```
private final Timer timer = new Timer("ClientHouseKeepingService", true)
```

- 线程池：

```
private ExecutorService publicExecutor;           // 公共线程池
private ExecutorService callbackExecutor;         // 回调线程池，客户端发起异步请求，服务器的响应数据由回调线程池处理
```

- 事件监听器：客户端这里是 null

```
private final ChannelEventListener channelEventListener;
```

构造方法

- 无参构造：

```
public NettyRemotingClient(final NettyClientConfig nettyClientConfig) {
    this(nettyClientConfig, null);
}
```

- 有参构造：

```
public NettyRemotingClient(NettyClientConfig, ChannelEventListener) {
    // 父类创建了2个信号量，1、控制单向请求的并发度，2、控制异步请求的并发度
    super(nettyClientConfig.getClientOnewaySemaphoreValue(),
          nettyClientConfig.getClientAsyncSemaphoreValue());
    this.nettyClientConfig = nettyClientConfig;
    this.channelEventListener = channelEventListener;

    // 创建公共线程池
    int publicThreadNums =
        nettyClientConfig.getClientCallbackExecutorThreads();
    if (publicThreadNums <= 0) {
        publicThreadNums = 4;
    }
    this.publicExecutor = Executors.newFixedThreadPool(publicThreadNums,);

    // 创建 Netty IO 线程，1个线程
    this.eventLoopGroupWorker = new NioEventLoopGroup(1, );

    if (nettyClientConfig.isUseTLS()) {
        sslContext = TlsHelper.buildSslContext(true);
    }
}
```

成员方法

- start(): 启动方法

```
public void start() {
    // channel pipeline 内的 handler 使用的线程资源，默认 4 个
    this.defaultEventExecutorGroup = new DefaultEventExecutorGroup();
    // 配置 netty 客户端启动类对象
    Bootstrap handler =
        this.bootstrap.group(this.eventLoopGroupWorker).channel(NioSocketChannel.class)
            //...
```

```

    .handler(new ChannelInitializer<SocketChannel>() {
        @Override
        public void initChannel(SocketChannel ch) throws Exception {
            ChannelPipeline pipeline = ch.pipeline();
            // 加几个handler
            pipeline.addLast(
                // 服务端的数据，都会来到这个
                new NettyClientHandler());
        }
    });
    // 注意 Bootstrap 只是配置好客户端的元数据了，【在这里并没有创建任何 channel 对象】
    // 定时任务 扫描 responseTable 中超时的 ResponseFuture，避免客户端线程长时间阻塞
    this.timer.scheduleAtFixedRate(() -> {
        NettyRemotingClient.this.scanResponseTable();
    }, 1000 * 3, 1000);
    // 这里是 null，不启动
    if (this.channelEventListener != null) {
        this.nettyEventExecutor.start();
    }
}

```

- 单向通信：

```

public RemotingCommand invokeSync(String addr, final RemotingCommand
request, long timeoutMillis) {
    // 开始时间
    long beginStartTime = System.currentTimeMillis();
    // 获取或者创建客户端与服务端（addr）的通道 channel
    final Channel channel = this.getAndCreateChannel(addr);
    // 条件成立说明客户端与服务端 channel 通道正常，可以通信
    if (channel != null && channel.isActive()) {
        try {
            // 执行 rpcHook 拓展点
            doBeforeRpcHooks(addr, request);
            // 计算耗时，如果当前耗时已经超过 timeoutMillis 限制，则直接抛出异常，不再
            // 进行系统通信
            long costTime = System.currentTimeMillis() - beginStartTime;
            if (timeoutMillis < costTime) {
                throw new RemotingTimeoutException("invokeSync call
timeout");
            }
            // 参数1：客户端-服务端通道channel
            // 参数二：网络层传输对象，封装着请求数据
            // 参数三：剩余的超时限制
            RemotingCommand response = this.invokeSyncImpl(channel, request,
...);
            // 后置处理
            doAfterRpcHooks(RemotingHelper.parseChannelRemoteAddr(channel),
request, response);
            // 返回响应数据
            return response;
        } catch (RemotingSendRequestException e) {}
    } else {
        this.closeChannel(addr, channel);
    }
}

```

```
        throw new RemotingConnectException(addr);
    }
}
```

延迟消息

消息处理

BrokerStartup 初始化 BrokerController 调用 `registerProcessor()` 方法将 SendMessageProcessor 注册到 NettyRemotingServer 中，对应的请求 ID 为 `SEND_MESSAGE = 10`，NettyServerHandler 在处理请求时通过 CMD 会获取处理器执行 `processRequest`

```
// 参数一：处理通道的事件；    参数二：客户端
public RemotingCommand processRequest(ChannelHandlerContext ctx, RemotingCommand request) {
    RemotingCommand response = null;
    response = asyncProcessRequest(ctx, request).get();
    return response;
}
```

SendMessageProcessor#asyncConsumerSendMsgBack：异步发送消费者的回调消息

- `final RemotingCommand response`：创建一个服务器响应对象
- `final ConsumerSendMsgBackRequestHeader requestHeader`：解析出客户端请求头信息，几个核心字段：
 - `private Long offset`：回退消息的 CommitLog offset
 - `private Integer delayLevel`：延迟级别，一般是 0
 - `private String originMsgId, originTopic`：原始的消息 ID，主题
 - `private Integer maxReconsumeTimes`：最大重试次数，默认是 16 次
- `if (...)`：鉴权，是否找到订阅组配置、Broker 是否支持写请求、订阅组是否支持消息重试
- `String newTopic = MixAll.getRetryTopic(...)`：**获取消费者组的重试主题**，规则是 `%RETRY%GroupName`
- `int queueIdInt = Math.abs()`：**重试主题下的队列 ID 是 0**
- `TopicConfig topicConfig`：获取重试主题的配置信息
- `MessageExt msgExt`：根据消息的物理 offset 到存储模块查询，内部先查询出这条消息的 size，然后再根据 offset 和 size 查询出整条 msg
- `final String retryTopic`：获取消息的原始主题
- `if (null == retryTopic)`：条件成立说明**当前消息是第一次被回退**，添加 `RETRY_TOPIC` 属性
- `msgExt.setWaitStoreMsgOK(false)`：异步刷盘
- `if (msgExt... >= maxReconsumeTimes || delayLevel < 0)`：消息重试次数超过最大次数，不支持重试
- `newTopic = MixAll.getDLQTopic()`：**获取消费者的死信队列**，规则是 `%DLQ%GroupName`

- `queueIdInt, topicConfig`: 死信队列 ID 为 0, 创建死信队列的配置
 - `if (0 == delayLevel)`: 说明延迟级别由 Broker 控制
`delayLevel = 3 + msgExt.getReconsumeTimes()`: **延迟级别默认从 3 级开始, 每重试一次, 延迟级别 +1**
 - `msgExt.setDelayTimeLevel(delayLevel)`: **将延迟级别设置进消息属性**, 存储时会检查该属性, 该属性值 > 0 会**将消息的主题和队列修改为调度主题和调度队列 ID**
 - `MessageExtBrokerInner msgInner`: 创建一条空消息, 消息属性从 offset 查询出来的 msg 中拷贝
 - `msgInner.setReconsumeTimes()`: 重试次数设置为原 msg 的次数 +1
 - `UtilAll.isBlank(originMsgId)`: 判断消息是否是初次返回到服务器
 - true: 说明 msgExt 消息是第一次被返回到服务器, 此时使用该 msg 的 id 作为 originMessageId
 - false: 说明原始消息已经被重试不止 1 次, 此时使用 offset 查询出来的 msg 中的 originMessageId
 - `CompletableFuture putMessageResult = ...asyncPutMessage(msgInner)`: 调用存储模块存储消息
- `DefaultMessageStore#asyncPutMessage`:
- `PutMessageResult result = this.commitLog.asyncPutMessage(msg)`: **将新消息存储到 CommitLog 中**
-

调度服务

DefaultMessageStore 中有成员属性 ScheduleMessageService, 在 start 方法中会启动该调度服务

成员变量:

- 延迟级别属性表:

```
// 存储延迟级别对应的 延迟时间长度 (单位: 毫秒)
private final ConcurrentMap<Integer /* level */, Long/* delay timeMillis */>
delayLevelTable;
// 存储延迟级别 queue 的消费进度 offset, 该 table 每 10 秒钟, 会持久化一次, 持久化到本地磁盘
private final ConcurrentMap<Integer /* level */, Long/* offset */>
offsetTable;
```

- 最大延迟级别:

```
private int maxDelayLevel;
```

- 模块启动状态:

```
private final AtomicBoolean started = new AtomicBoolean(false);
```

- 定时器: 内部有线程资源, 可执行调度任务

```
private Timer timer;
```

成员方法：

- load(): 加载调度消息，**初始化 delayLevelTable 和 offsetTable**

```
public boolean load()
```

- start(): 启动消息调度服务

```
public void start()
```

- `if (started.compareAndSet(false, true))`: 将启动状态设为 true
- `this.timer`: 创建定时器对象
- `for (... : this.delayLevelTable.entrySet())`: 为每个延迟级别创建一个延迟任务提交到 timer，周期执行，这样就可以**将延迟消息得到及时的消费**
- `this.timer.scheduleAtFixedRate()`: 提交周期型任务，延迟 10 秒执行，周期为 10 秒，持久化延迟队列消费进度任务
- `ScheduleMessageService.this.persist()`: 持久化消费进度

调度任务

DeliverDelayedMessageTimerTask 是一个任务类

成员变量：

- 延迟级别：延迟队列任务处理的延迟级别

```
private final int delayLevel;
```

- 消费进度：延迟队列任务处理的延迟队列的消费进度

```
private final long offset;
```

成员方法：

- run(): 执行任务

```
public void run() {
    if (isStarted()) {
        this.executeOnTimeout();
    }
}
```

- executeOnTimeout(): 执行任务

```
public void executeOnTimeout()
```

- `consumeQueue cq`：获取出该延迟队列任务处理的**延迟队列 ConsumeQueue**
 - `selectMappedBufferResult bufferCQ`：根据消费进度查询出 SMBR 对象
 - `for (; i < bufferCQ.getSize(); i += ConsumeQueue.CQ_STORE_UNIT_SIZE)`：每次读取 20 个字节的数据
 - `offsetPy, sizePy`：延迟消息的物理偏移量和消息大小
 - `long tagsCode`：延迟消息的交付时间，在 ReputMessageService 转发时根据消息的 DELAY 属性是否 >0，会在 tagsCode 字段存储交付时间
 - `long deliver... = this.correctDeliverTimestamp(..)`：**校准交付时间**，延迟时间过长会调整为当前时间立刻执行
 - `long countdown = deliverTimestamp - now`：计算差值
 - `if (countdown <= 0)`：**消息已经到达交付时间了**
`MessageExt msgExt`：根据物理偏移量和消息大小获取这条消息
`MessageExtBrokerInner msgInner`：**构建一条新消息**，将原消息的属性拷贝过来
 - `long tagsCodeValue`：不再是交付时间了
 - `MessageAccessor.clearProperty(msgInner, DELAY..)`：清理新消息的 DELAY 属性，避免存储时重定向到延迟队列
 - `msgInner.setTopic()`：**修改主题为原始的主题 %RETRY%GroupName**
 - `String queueIdstr`：修改队列 ID 为原始的 ID`PutMessageResult putMessageResult`：**将新消息存储到 CommitLog**，消费者订阅的是目标主题，会再次消费该消息
 - `else`：**消息还未到达交付时间**
`ScheduleMessageService.this.timer.schedule()`：创建该延迟级别的任务，延迟 countDown 毫秒之后再执行
`ScheduleMessageService.this.updateoffset()`：更新延迟级别队列的消费进度
 - `PutMessageResult putMessageResult`
 - `bufferCQ == null`：说明通过消费进度没有获取到数据
 - `if (offset < cqMinOffset)`：如果消费进度比最小位点都小，说明是过期数据，重置为最小位点
 - `ScheduleMessageService.this.timer.schedule()`：重新提交该延迟级别对应的延迟队列任务，延迟 100 毫秒后执行
-

事务消息

生产者类

TransactionMQProducer 类发送事务消息时使用

成员变量：

- 事务回查线程池资源：

```
private ExecutorService executorService;
```

- 事务监听器：

```
private TransactionListener transactionListener;
```

核心方法：

- start(): 启动方法

```
public void start()
```

- `this.defaultMQProducerImpl.initTransactionEnv()`: 初始化生产者实例和回查线程池资源
- `super.start()`: 启动生产者实例
- sendMessageInTransaction(): 发送事务消息

```
public TransactionSendResult sendMessageInTransaction(final Message msg,
final Object arg) {
    msg.setTopic(NamespaceUtil.wrapNamespace(this.getNamespace(),
msg.getTopic()));
    // 调用实现类的发送方法
    return this.defaultMQProducerImpl.sendMessageInTransaction(msg, null,
arg);
}
```

- `TransactionListener transactionListener = getCheckListener()`: 获取监听器
- `if (null == localTransactionExecuter && null == transactionListener)`: 两者都为 null 抛出异常
- `MessageAccessor.setProperty(msg,`
`MessageConst.PROPERTY_TRANSACTION_PREPARED, "true")`: **设置事务标志**
- `sendResult = this.send(msg)`: 发送消息，同步发送
- `switch (sendResult.getSendStatus())`: **判断发送消息的结果状态**
- `case SEND_OK`: 消息发送成功
 - `msg.setTransactionId(transactionId)`: **设置事务 ID 为消息的 UNIQ_KEY 属性**
 - `localTransactionState = ...executeLocalTransactionBranch(msg, arg)`: **执行本地事务**
- `case SLAVE_NOT_AVAILABLE`: 其他情况都需要回滚事务
 - `localTransactionState = LocalTransactionState.ROLLBACK_MESSAGE`: **事务状态设置为回滚**
- `this.endTransaction(sendResult, ...)`: 结束事务
 - `EndTransactionRequestHeader requestHeader`: 构建事务结束头对象
 - `this.mqClientFactory.getMQClientAPIImpl().endTransactionOneway()`: 向 Broker 发起事务结束的单向请求

接受消息

SendMessageProcessor 是服务端处理客户端发送来的消息的处理器，`processRequest()` 方法处理请求

核心方法：

- `asyncProcessRequest()`：处理请求

```
public CompletableFuture<RemotingCommand>
asyncProcessRequest(ChannelHandlerContext ctx,
                    RemotingCommand request) {
    final SendMessageContext mqtraceContext;
    switch (request.getCode()) {
        // 回调消息回退
        case RequestCode.CONSUMER_SEND_MSG_BACK:
            return this.asyncConsumerSendMsgBack(ctx, request);
        default:
            // 解析出请求头对象
            SendMessageRequestHeader requestHeader =
                parseRequestHeader(request);
            if (requestHeader == null) {
                return CompletableFuture.completedFuture(null);
            }
            // 创建上下文对象
            mqtraceContext = buildMsgContext(ctx, requestHeader);
            // 前置处理器
            this.executeSendMessageHookBefore(ctx, request, mqtraceContext);
            // 判断是否是批量消息
            if (requestHeader.isBatch()) {
                return this.asyncSendBatchMessage(ctx, request,
                    mqtraceContext, requestHeader);
            } else {
                return this.asyncSendMessage(ctx, request, mqtraceContext,
                    requestHeader);
            }
    }
}
```

- `asyncSendMessage()`：异步处理发送消息

```
private CompletableFuture<RemotingCommand>
asyncSendMessage(ChannelHandlerContext ctx, RemotingCommand request,
                 SendMessageContext mqtraceContext, SendMessageRequestHeader requestHeader)
```

- `RemotingCommand response`：创建响应对象
- `MessageExtBrokerInner msgInner = new MessageExtBrokerInner()`：创建 msgInner 对象，并赋值相关的属性，主题和队列 ID 都是请求头中的
- `String transFlag`：获取事务属性

- o `if (transFlag != null && Boolean.parseBoolean(transFlag))`: 判断事务属性是否是 true, 走事务消息的存储流程
 - `putMessageResult = ...asyncPrepareMessage(msgInner)`: 事务消息处理流程

```
public CompletableFuture<PutMessageResult>
asyncPutHalfMessage(MessageExtBrokerInner messageInner) {
    // 调用存储模块, 将修改后的 msg 存储进 Broker(CommitLog)
    return
store.asyncPutMessage(parseHalfMessageInner(messageInner));
}
```

TransactionalMessageBridge#parseHalfMessageInner:

- `MessageAccessor.putProperty(...)`: 将消息的原主题和队列 ID 放入消息的属性中
- `msgInner.setSysFlag(...)`: 消息设置为非事务状态
- `msgInner.setTopic(TransactionMessageUtil.buildHalfTopic())`: 消息主题设置为半消息主题
- `msgInner.setQueueId(0)`: 队列 ID 设置为 0
- o `else`: 普通消息存储

回查处理

ClientRemotingProcessor 是客户端用于处理请求, 创建 MQClientAPIImpl 时将该处理器注册到 Netty 中, `processRequest()` 方法根据请求的命令码, 进行不同的处理, 事务回查的处理命令码为 `CHECK_TRANSACTION_STATE`

Broker 端有定时任务发送回查请求

成员方法:

- `checkTransactionState()`: 检查事务状态

```
public RemotingCommand checkTransactionState(ChannelHandlerContext ctx,
RemotingCommand request)
```

- o `final CheckTransactionStateRequestHeader requestHeader`: 解析出请求头对象
- o `final MessageExt messageExt`: 从请求 body 中解析出服务器回查的事务消息
- o `String transactionId`: 提取 UNIQ_KEY 字段属性值赋值给事务 ID
- o `final String group`: 提取生产者组名
- o `MQProducerInner producer = this...selectProducer(group)`: 根据生产者组获取生产者对象
- o `String addr = RemotingHelper.parseChannelRemoteAddr()`: 解析出要回查的 Broker 服务器的地址
- o `producer.checkTransactionState(addr, messageExt, requestHeader)`: 生产者的事务回查

- `Runnable request = new Runnable()`: 创建回查事务状态任务对象
 - 获得生产者的 TransactionCheckListener 和 TransactionListener, 选择一个不为 null 的监听器进行事务状态回查
 - `this.processTransactionState()`: 处理回查状态
 - `EndTransactionRequestHeader thisHeader`: 构建 EndTransactionRequestHeader 对象
 - `DefaultMQProducerImpl...endTransactionOneway()`: 向 Broker 发起结束事务单向请求, **二阶段提交**
 - `this.checkExecutor.submit(request)`: 提交到线程池运行

参考图: <https://www.processon.com/view/link/61c8257e0e3e7474fb9dc0>

参考视频: <https://space.bilibili.com/457326371>

事务提交

EndTransactionProcessor 类是服务端用来处理客户端发来的提交或者回滚请求

- `processRequest()`: 处理请求

```
public RemotingCommand processRequest(ChannelHandlerContext ctx,
RemotingCommand request)
```

- `EndTransactionRequestHeader requestHeader`: 从请求中解析出 EndTransactionRequestHeader
- `if (MessageSysFlag.TRANSACTION_COMMIT_TYPE)`: **事务提交**
 - `result = this.brokerController...commitMessage(requestHeader)`: 根据 commitLogOffset 提取出 halfMsg 消息
 - `MessageExtBrokerInner msgInner`: 根据 result 克隆出一条新消息
 - `msgInner.setTopic(msgExt.getUserProperty(...))`: **设置回原主题**
 - `msgInner.setQueueId(Integer.parseInt(msgExt.getUserProperty(..)))`: **设置回原队列 ID**
 - `MessageAccessor.clearProperty()`: 清理上面的两个属性
 - `MessageAccessor.clearProperty(msgInner, ...)`: **清理事务属性**
- `RemotingCommand sendResult = sendFinalMessage(msgInner)`: 调用存储模块存储至 Broker
- `this.brokerController...deletePrepareMessage(result.getPrepareMessage())`: **向删除 (OP) 队列添加消息**, 消息体的数据是 halfMsg 的 queueOffset, 表示半消息队列指定的 offset 的消息已被删除
 - `if (this...putOpMessage(msgExt, TransactionalMessageUtil.REMOVETAG))`: 添加一条 OP 数据
 - `MessageQueue messageQueue`: 新建一个消息队列, OP 队列

- `return addRemoveTagInTransactionop(messageExt, messageQueue)`：添加数据
 - `Message message`：创建 OP 消息
 - `writeOp(message, messageQueue)`：写入 OP 消息
 - `else if (MessagesSysFlag.TRANSACTION_ROLLBACK_TYPE)`：**事务回滚**
`this.brokerController...deletePrepareMessage(result.getPrepareMessage())`：
也需要向 OP 队列添加消息
-

Consumer

消费者类

默认消费

DefaultMQPushConsumer 类是默认的消费者类

成员变量：

- 消费者实现类：

```
protected final transient DefaultMQPushConsumerImpl  
defaultMQPushConsumerImpl;
```

- 消费属性：

```
private String consumerGroup; // 消费者组  
private MessageModel messageModel = MessageModel.CLUSTERING; // 消费模式,  
默认集群模式
```

- 订阅信息：key 是主题，value 是过滤表达式，一般是 tag

```
private Map<String, String > subscription = new HashMap<String, String>()
```

- 消息监听器：**消息处理逻辑**，并发消费 MessageListenerConcurrently，顺序（分区）消费 MessageListenerOrderly

```
private MessageListener messageListener;
```

- 消费位点：当从 Broker 获取当前组内该 queue 的 offset 不存在时，consumeFromWhere 才有效，默认值代表从队列的最后 offset 开始消费，当队列内再有一条新的 msg 加入时，消费者才会去消费

```
private ConsumeFromwhere consumeFromwhere =  
ConsumeFromwhere.CONSUME_FROM_LAST_OFFSET;
```

- 消费时间戳：当消费位点配置的是 CONSUME_FROM_TIMESTAMP 时，并且服务器 Group 内不存在该 queue 的 offset 时，会使用该时间戳进行消费

```
private String consumeTimestamp =  
utilAll.timeMillisToHumanString3(System.currentTimeMillis() - (1000 * 60 *  
30)); // 消费者创建时间 - 30秒, 转换成 格式: 年月日小时分钟秒, 比如 20220203171201
```

- 队列分配策略：主题下的队列分配策略，RebalanceImpl 对象依赖该算法

```
private AllocateMessageQueueStrategy allocateMessageQueueStrategy;
```

- 消费进度存储器：

```
private OffsetStore offsetStore;
```

核心方法：

- start(): 启动消费者

```
public void start()
```

- shutdown(): 关闭消费者

```
public void shutdown()
```

- registerMessageListener(): 注册消息监听器

```
public void registerMessageListener(MessageListener messageListener)
```

- subscribe(): 添加订阅信息，**将订阅信息放入负载均衡对象的 subscriptionInner 中**

```
public void subscribe(String topic, String subExpression)
```

- unsubscribe(): 删除订阅指定主题的信息

```
public void unsubscribe(String topic)
```

- suspend(): 停止消费

```
public void suspend()
```

- resume(): 恢复消费

```
public void resume()
```

默认实现

DefaultMQPushConsumerImpl 是默认消费者的实现类

成员变量：

- 客户端实例：整个进程内只有一个客户端实例对象

```
private MQClientInstance mqClientFactory;
```

- 消费者实例：门面对象

```
private final DefaultMQPushConsumer defaultMQPushConsumer;
```

- **负载均衡**：分配订阅主题的队列给当前消费者，20 秒钟一个周期执行 Rebalance 算法（客户端实例触发）

```
private final RebalanceImpl rebalanceImpl = new RebalancePushImpl(this);
```

- 消费者信息：

```
private final long consumerStartTimestamp; // 消费者启动时间
private volatile ServiceState serviceState; // 消费者状态
private volatile boolean pause = false; // 是否暂停
private boolean consumeOrderly = false; // 是否顺序消费
```

- **拉取消息**：封装拉消息的 API，服务器 Broker 返回结果中包含下次 Pull 时推荐的 BrokerId，根据本次请求数据的冷热程度进行推荐

```
private PullAPIWrapper pullAPIWrapper;
```

- **消息消费服务**：并发消费和顺序消费

```
private ConsumeMessageService consumeMessageService;
```

- 流控：

```
private long queueFlowControlTimes = 0; // 队列流控次数，默认每1000次流控，进行一次日志打印
private long queueMaxSpanFlowControlTimes = 0; // 流控使用，控制打印日志
```

- HOOK：钩子方法

```
// 过滤消息 hook
private final ArrayList<FilterMessageHook> filterMessageHookList;
// 消息执行hook，在消息处理前和处理后分别执行 hook.before hook.after 系列方法
private final ArrayList<ConsumeMessageHook> consumeMessageHookList;
```

核心方法：

- start()：加锁保证线程安全

```
public synchronized void start()
```

- `this.checkConfig()`: 检查配置，包括组名、消费模式、订阅信息、消息监听器等
- `this.copySubscription()`: 拷贝订阅信息到 RebalanceImpl 对象
 - `this.rebalanceImpl.getSubscriptionInner().put(topic, subscriptionData)`: 将订阅信息加入 rbl 的 map 中
 - `this.messageListenerInner = ...getMessageListener()`: 将消息监听器保存到实例对象
 - `switch (this.defaultMQPushConsumer.getMessageMode())`: 判断消费模式，广播模式下直接返回
 - `final String retryTopic`: 创建当前**消费者组重试的主题名**，规则`%RETRY%ConsumerGroup`
 - `SubscriptionData subscriptionData = FilterAPI.buildSubscriptionData()`: 创建重试主题的订阅数据对象
 - `this.rebalanceImpl.getSubscriptionInner().put(retryTopic, subscriptionData)`: 将创建的重试主题加入到 rbl 对象的 map 中，**消息重试时会加入到该主题，消费者订阅这个主题之后，就有机会再次拿到该消息进行消费处理**
- `this.mQClientFactory = ...getOrCreateMQClientInstance()`: 获取客户端实例对象
- `this.rebalanceImpl`: 初始化负载均衡对象，**设置队列分配策略对象到属性中**
- `this.pullAPIWrapper = new PullAPIWrapper()`: 创建拉消息 API 对象，内部封装了查询推荐主机算法
- `this.pullAPIWrapper.registerFilterMessageHook(filterMessageHookList)`: 将过滤 Hook 列表注册到该对象内，消息拉取下来之后会执行该 Hook，**再进行一次自定义的消息过滤**
- `this.offsetStore = new RemoteBrokerOffsetStore()`: 默认集群模式下创建消息进度存储器
- `this.consumeMessageService = ...`: 根据消息监听器的类型创建消费服务
- `this.consumeMessageService.start()`: 启动消费服务
- `boolean registerOK = mQClientFactory.registerConsumer()`: **将消费者注册到客户端实例中**，客户端提供的服务：
 - 心跳服务：把订阅数据同步到订阅主题的 Broker
 - 拉消息服务：内部 PullMessageService 启动线程，基于 PullRequestQueue 工作，消费者负载均衡分配到队列后会向该队列提交 PullRequest
 - 队列负载服务：每 20 秒调用一次 `consumer.doRebalance()` 接口
 - 消息进度持久化
 - 动态调整消费者、消费服务线程池
- `mQClientFactory.start()`: 启动客户端实例
- `this.updateTopic`: 从 nameserver 获取主题路由数据，生成主题集合放入 rbl 对象的 table
- `this.mQClientFactory.checkClientInBroker()`: 检查服务器是否支持消息过滤模式，一般使用 tag 过滤，服务器默认支持
- `this.mQClientFactory.sendHeartbeatToAllBrokerWithLock()`: 向所有已知的 Broker 节点，**发送心跳数据**
- `this.mQClientFactory.rebalanceImmediately()`: 唤醒 rbl 线程，触发负载均衡执行

负载均衡

实现方式

MQClientInstance#start 中会启动负载均衡服务 RebalanceService：

```
public void run() {
    // 检查停止标记
    while (!this.isStopped()) {
        // 休眠 20 秒，防止其他线程饥饿，所以【每 20 秒负载均衡一次】
        this.waitForRunning(waitInterval);
        // 调用客户端实例的负载均衡方法，底层【会遍历所有消费者，调用消费者的负载均衡】
        this.mqClientFactory.doRebalance();
    }
}
```

RebalanceImpl 类成员变量：

- 分配给当前消费者的处理队列：处理消息队列集合，**ProcessQueue** 是 MQ 队列在消费者端的快照

```
protected final ConcurrentMap<MessageQueue, ProcessQueue> processQueueTable;
```

- 消费者订阅主题的队列信息：

```
protected final ConcurrentMap<String/* topic */, Set<MessageQueue>>
topicSubscribeInfoTable;
```

- 订阅数据：

```
protected final ConcurrentMap<String/* topic */, SubscriptionData>
subscriptionInner;
```

- 队列分配策略：

```
protected AllocateMessageQueueStrategy allocateMessageQueueStrategy;
```

成员方法：

- doRebalance(): 负载均衡方法，以每个消费者实例为粒度进行负载均衡

```
public void doRebalance(final boolean isOrder) {
    // 获取当前消费者的订阅数据
    Map<String, SubscriptionData> subTable = this.getSubscriptionInner();
    if (subTable != null) {
        // 遍历所有的订阅主题
        for (final Entry<String, SubscriptionData> entry :
            subTable.entrySet()) {
```

```

    // 获取订阅的主题
    final String topic = entry.getKey();
    // 按照主题进行负载均衡
    this.rebalanceByTopic(topic, isorder);
}
}

// 将分配到当前消费者的队列进行过滤，不属于当前消费者订阅主题的直接移除
this.truncateMessageQueueNotMyTopic();
}

```

集群模式下：

- `Set<MessageQueue> mqSet = this.topicSubscribeInfoTable.get(topic)`：订阅的主题下的全部队列信息
- `cidAll = this...findConsumerIdList(topic, consumerGroup)`：从服务器获取消费者组下的全部消费者 ID
- `collections.sort(mqAll)`：主题 MQ 队列和消费者 ID 都进行排序，**保证每个消费者的视图一致性**
- `allocateResult = strategy.allocate()`：**调用队列分配策略**，给当前消费者进行分配 MessageQueue（下一节）
- `boolean changed = this.updateProcessQueueTableInRebalance(...)`：**更新队列处理集合**，mqSet 是 rbl 算法分配到当前消费者的 MQ 集合
 - `while (it.hasNext())`：遍历当前消费者的全部处理队列
 - `if (mq.getTopic().equals(topic))`：该 MQ 是本次 rbl 分配算法计算的主题
 - `if (!mqSet.contains(mq))`：该 MQ 经过 rbl 计算之后，**被分配到其它 Consumer 节点**
- `if (pq.setDropped(true))`：将删除状态设置为 true
- `if (this.removeUnnecessaryMessageQueue(mq, pq))`：删除不需要的 MQ 队列
 - `this...getOffsetStore().persist(mq)`：在 MQ 归属的 Broker 节点持久化消费进度
 - `this...getOffsetStore().removeOffset(mq)`：删除该 MQ 在本地的消费进度
 - `if (this.defaultMQPushConsumerImpl.isConsumeOrderly() &&`：是否是**顺序消费**和**集群模式**
- `if (pq.getLockConsume().tryLock(1000, ...))`：获取锁成功，说明顺序消费任务已经停止消费工作
- `return this.unlockDelay(mq, pq)`：**释放锁 Broker 端的队列锁，向服务器发起 oneway 的解锁请求**
 - `if (pq.hasTempMessage())`：队列中有消息，延迟 20 秒释放队列分布式锁，确保全局范围内只有一个消费任务运行中
 - `else`：当前消费者本地该消费任务已经退出，直接释放锁
- `else`：顺序消费任务正在消费一批消息，不可打断，增加尝试获取锁的次数
- `it.remove()`：从 processQueueTable 移除该 MQ
- `else if (pq.isPullExpired())`：说明当前 MQ 还是被当前 Consumer 消费，此时判断一下是否超过 2 分钟未到服务器拉消息，如果条件成立进行上述相同的逻辑
- `for (MessageQueue mq : mqSet)`：开始处理当前主题**新分配到当前节点的队列**

- `if (isOrder && !this.lock(mq))`: 顺序消息为了保证有序性，需要获取队列锁
- `ProcessQueue pq = new ProcessQueue()`: 为每个新分配的消息队列创建快照队列
- `long nextOffset = this.computePullFromWhere(mq)`: 从服务端获取新分配的 MQ 的消费进度
- `ProcessQueue pre = this.processQueueTable.putIfAbsent(mq, pq)`: 保存到处理队列集合
- `PullRequest pullRequest = new PullRequest()`: 创建拉取请求对象
- `this.dispatchPullRequest(pullRequestList)`: 放入 PullMessageService 的本地阻塞队列内，用于拉取消息工作
- `lockAll()`: 续约锁，对消费者的所有队列进行续约

```
public void lockAll()
```

- `HashMap<String, Set<MessageQueue>> brokerMqs`: 将分配给当前消费者的全部 MQ 按照 BrokerName 分组
- `while (it.hasNext())`: 遍历所有的分组
- `final Set<MessageQueue> mqs`: 获取该 Broker 上分配给当前消费者的 queue 集合
- `FindBrokerResult findBrokerResult`: 查询 Broker 主节点信息
- `LockBatchRequestBody requestBody`: 创建请求对象，填充属性
- `Set<MessageQueue> lockOKMQSet`: 以组为单位向 Broker 发起批量续约锁的同步请求，返回成功的队列集合
- `for (MessageQueue mq : lockOKMQSet)`: 遍历续约锁成功的 MQ

`processQueue.setLocked(true)`: 分布式锁状态设置为 true，表示允许顺序消费

`processQueue.setLastLockTimestamp(System.currentTimeMillis())`: 设置上次获取锁的时间为当前时间

`for (MessageQueue mq : mqs)`: 遍历当前 Broker 上的所有队列集合

`if (!lockOKMQSet.contains(mq))`: 条件成立说明续约锁失败

`processQueue.setLocked(false)`: 分布式锁状态设置为 false，表示不允许顺序消费

队列分配

AllocateMessageQueueStrategy 类是队列的分配策略

- 平均分配: AllocateMessageQueueAveragely 类

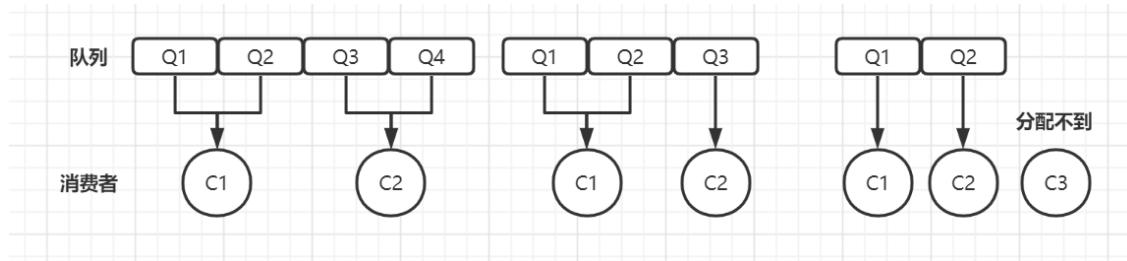
```
// 参数一: 消费者组                                     参数二: 当前消费者id
// 参数三: 主题的全部队列, 包括所有 broker 上该主题的 mq   参数四: 全部消费者id集合
public List<MessageQueue> allocate(String consumerGroup, String currentCID,
List<MessageQueue> mqAll, List<String> cidAll) {
    // 获取当前消费者在全部消费者中的位置, 【全部消费者是已经排序好的, 排在前面的优先分配更多的队列】
```

```

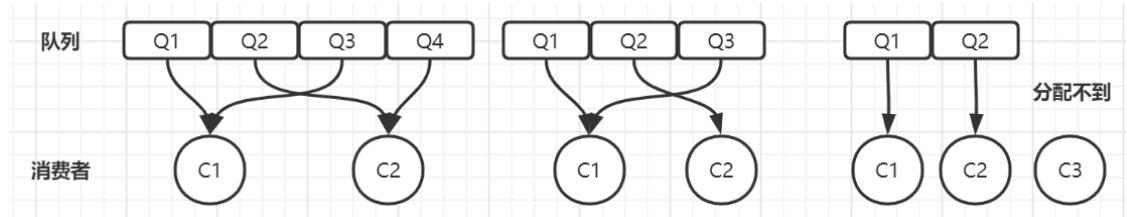
int index = cidAll.indexOf(currentCID);
// 平均分配完以后，还剩余的待分配的 mq 的数量
int mod = mqAll.size() % cidAll.size();
// 首先判断整体的 mq 的数量是否小于消费者的数量，小于消费者的数量就说明不够分的，先分
一个
int averageSize = mqAll.size() <= cidAll.size() ? 1 :
    // 成立需要多分配一个队列，因为更靠前
    (mod > 0 && index < mod ? mqAll.size() / cidAll.size() + 1 :
    mqAll.size() / cidAll.size());
// 获取起始的分配位置
int startIndex = (mod > 0 && index < mod) ? index * averageSize : index
* averageSize + mod;
// 防止索引越界
int range = Math.min(averageSize, mqAll.size() - startIndex);
// 开始分配，【挨着分配，是直接就把当前的 消费者分配完成】
for (int i = 0; i < range; i++) {
    result.add(mqAll.get((startIndex + i) % mqAll.size()));
}
return result;
}

```

队列排序后：Q1 → Q2 → Q3，消费者排序后 C1 → C2 → C3



- 轮流分配：AllocateMessageQueueAveragelyByCircle



- 指定机房平均分配：AllocateMessageQueueByMachineRoom，前提是 Broker 的命名规则为 机
房名@BrokerName

拉取服务

实现方式

MQClientInstance#start 中会启动消息拉取服务：PullMessageService

```

public void run() {
    // 检查停止标记, 【循环拉取】
    while (!this.isStopped()) {
        try {
            // 从阻塞队列中获取拉消息请求
            PullRequest pullRequest = this.pullRequestQueue.take();
            // 拉取消息, 获取请求对应的使用当前消费者组中的哪个消费者, 调用消费者的
            pullMessage 方法
            this.pullMessage(pullRequest);
        } catch (Exception e) {
            log.error("Pull Message Service Run Method exception", e);
        }
    }
}

```

DefaultMQPushConsumerImpl#pullMessage:

- `ProcessQueue processQueue = pullRequest.getProcessQueue()`: 获取请求对应的快照队列, 并判断是否是删除状态
- `this.executePullRequestLater()`: 如果当前消费者不是运行状态, 则拉消息任务延迟 3 秒后执行, 如果是暂停状态延迟 1 秒

• 流控的逻辑:

`long cachedMessageCount = processQueue.getMsgCount().get()`: 获得消费者本地该 queue 快照内缓存的消息数量, 如果大于 1000 条, 进行流控, 延迟 50 毫秒

`long cachedMessagesizeInMiB`: 消费者本地该 queue 快照内缓存的消息容量 size, 超过 100m 消息未被消费进行流控

`if(processQueue.getMaxSpan() > 2000)`: 消费者本地缓存消息第一条消息最后一条消息跨度超过 2000 进行流控

- `SubscriptionData subscriptionData`: 本次拉消息请求订阅的主题数据, 如果调用了 `unsubscribe(主题)` 将会获取为 null
- `PullCallback pullCallback = new Pullcallback()`: **拉消息处理回调对象**
 - `pullResult = ...processPullResult()`: 预处理 PullResult 结果, 将服务器端指定 MQ 的拉消息下一次的推荐节点保存到 pullFromWhichNodeTable 中, **并进行消息过滤**
 - `case FOUND`: 正常拉取到消息

`pullRequest.setNextOffset(pullResult.getNextBeginOffset())`: 更新 pullRequest 对象下一次拉取消息的位点

`if (pullResult.getMsgFoundList() == null...)`: 消息过滤导致消息全部被过滤掉, 需要立马发起下一次拉消息

`boolean ... = processQueue.putMessage()`: 将服务器拉取的消息集合**加入到消费者本地**的 processQueue 内

`DefaultMQPushConsumerImpl...submitConsumeRequest()`: **提交消费任务, 分为顺序消费和并发消费**

`DefaultMQPushConsumerImpl.executePullRequestImmediately(pullRequest)`: 将更新过 nextOffset 字段的 PullRequest 对象, 再次放到 pullMessageService 的阻塞队列中, **形成闭环**

- `case NO_NEW_MSG || NO_MATCHED_MSG`: 表示本次 pull 没有新的可消费的信息

- `pullRequest.setNextOffset()`: 更新 pullRequest 对象下一次拉取消息的位点
 - `Default..executePullRequestImmediately(pullRequest)`: 再次拉取请求
 - `case OFFSET_ILLEGAL`: 本次 pull 时使用的 offset 是无效的, 即 `offset > maxOffset || offset < minOffset`
 - `pullRequest.setNextOffset()`: 调整 `pullRequest.nextOffset` 为正确的 offset
 - `pullRequest.getProcessQueue().setDropped(true)`: 设置该 processQueue 为删除状态, 如果有该 queue 的消费任务, 消费任务会马上停止
 - `DefaultMQPushConsumerImpl.this.executeTaskLater()`: 提交异步任务, 10 秒后去执行
 - `DefaultMQPushConsumerImpl...updateOffset()`: 更新 offsetStore 该 MQ 的 offset 为正确值, 内部直接替换
 - `DefaultMQPushConsumerImpl...persist()`: 持久化该 messageQueue 的 offset 到 Broker 端
 - `DefaultMQPushConsumerImpl...removeProcessQueue()`: 删除该消费者该 messageQueue 对应的 processQueue
 - 这里没有再次提交 pullRequest 到 pullMessageService 的队列, 那该队列不再拉消息了吗?
 - 负载均衡 rbl 程序会重建该队列的 processQueue, 重建完之后会为该队列创建新的 PullRequest 对象
 - `int sysFlag = PullSysFlag.buildSysFlag()`: 构建标志对象, sysFlag 高 4 位未使用, 低 4 位使用, 从左到右 0000 0011
 - 第一位: 表示是否提交消费者本地该队列的 offset, 一般是 1
 - 第二位: 表示是否允许服务器端进行长轮询, 一般是 1
 - 第三位: 表示是否提交消费者本地该主题的订阅数据, 一般是 0
 - 第四位: 表示是否为类过滤, 一般是 0
 - `this.pullAPIWrapper.pullKernelImpl()`: 拉取消息的核心方法
-

封装对象

PullAPIWrapper 类封装了拉取消息的 API

成员变量:

- 推荐拉消息使用的主机 ID:

```
private ConcurrentMap<MessageQueue, AtomicLong/* brokerId */>
pullFromWhichNodeTable
```

成员方法:

- `pullKernelImpl()`: 拉消息
 - `FindBrokerResult findBrokerResult`: 本地查询指定 BrokerName 的地址信息, 推荐节点或者主节点

- o `if (null == findBrokerResult)`: 查询不到, 就到 Namesrv 获取指定 topic 的路由数据
 - o `if (findBrokerResult.isSlave())`: 成立说明 findBrokerResult 表示的主机为 slave 节点, **slave 不存储 offset 信息**
`sysFlagInner = PullSysFlag.clearCommitOffsetFlag(sysFlagInner)`: 将 sysFlag 标记位中 CommitOffset 的位置为 0
 - o `PullMessageRequestHeader requestHeader`: 创建请求头对象, 封装所有的参数
 - o `PullResult pullResult =`
`this.mQClientFactory.getMQClientAPIImpl().pullMessage()`: 调用客户端实例的方法, 核心逻辑就是**将业务数据转化为 RemotingCommand 通过 NettyRemotingClient 的 IO 进行通信**
 - `RemotingCommand request`: 创建网络层传输对象 RemotingCommand 对象, **请求 ID 为 PULL_MESSAGE = 11**
 - `return this.pullMessageSync(...)`: 此处是**异步调用, 处理结果放入 ResponseFuture 中**, 参考服务端小节的处理器类 `NettyServerHandler#processMessageReceived` 方法
 - `RemotingCommand response = responseFuture.getResponseCommand()`: 获取服务器端响应数据 response
 - o `PullResult pullResult`: 从 response 内提取出来拉消息结果对象, 将响应头 `PullMessageResponseHeader` 对象中信息**填充到 PullResult 中**, 列出两个重要的字段:
 - o `private Long suggestWhichBrokerId`: 服务端建议客户端下次 Pull 时选择的 BrokerID
 - o `private Long nextBeginOffset`: 客户端下次 Pull 时使用的 offset 信息
 - `pullCallback.onSuccess(pullResult)`: 将 PullResult 交给拉消息结果处理回调对象, 调用 onSuccess 方法
-

拉取处理

处理器

BrokerStartup#createBrokerController 方法中创建了 BrokerController 并进行初始化, 调用 `registerProcessor()` 方法将处理器 PullMessageProcessor 注册到 NettyRemotingServer 中, 对应的请求 ID 为 `PULL_MESSAGE = 11`, NettyServerHandler 在处理请求时通过请求 ID 会获取处理器执行 `processRequest` 方法

```
// 参数一: 服务器与客户端 netty 通道; 参数二: 客户端请求; 参数三: 是否允许服务器端长轮询, 默认 true
private RemotingCommand processRequest(final Channel channel, RemotingCommand request, boolean brokerAllowSuspend)
```

- `RemotingCommand response`: 创建响应对象, 设置为响应类型的请求, 响应头是 `PullMessageResponseHeader`
- `final PullMessageResponseHeader responseHeader`: 获取响应对象的 header
- `final PullMessageRequestHeader requestHeader`: 解析出请求头 `PullMessageRequestHeader`

- `response.setOpaque(request.getOpaque())`: 设置 opaque 属性，客户端根据该字段获取 **ResponseFuture** 进行处理
- 进行一些鉴权的逻辑：是否允许长轮询、提交 offset、topicConfig 是否是空、队列 ID 是否合理
- `ConsumerGroupInfo consumerGroupInfo`: 获取消费者组信息，包含全部的消费者和订阅数据
- `subscriptionData = consumerGroupInfo.findSubscriptionData()`: 获取指定主题的订阅数据
- `if (!ExpressionType.isTagType())`: 表达式匹配
- `MessageFilter messageFilter`: 创建消息过滤器，一般是通过 tagCode 进行过滤
- `DefaultMessageStore.getMessage()`: **查询消息的核心逻辑，在 Broker 端查询消息** (存储端笔记详解了该源码)
- `response.setRemark()`: 设置此次响应的状态
- `responseHeader.set...`: 设置响应头对象的一些字段
- `switch (this.brokerController.getMessageStoreConfig().getBrokerRole())`: 如果当前主机节点角色为 slave 并且从节点读并未开启的话，直接给客户端一个状态 `PULL_RETRY_IMMEDIATELY`，并设置为下次从主节点读
- `if (this.brokerController.getBrokerConfig().isSlaveReadEnable())`: 消费太慢，**下次从另一台机器拉取**
- `switch (getMessageResult.getStatus())`: 根据 getMessageResult 的状态设置 response 的 code

```

public enum GetMessageStatus {
    FOUND, // 查询成功
    NO_MATCHED_MESSAGE, // 未查询到到消息，服务端过滤 tagCode
    MESSAGE_WAS_Removing, // 查询时赶上 CommitLog 清理过期文件，导致查询失败，立刻尝试
    OFFSET_FOUND_NULL, // 查询时赶上 ConsumerQueue 清理过期文件，导致查询失败，【进行长轮询】
    OFFSET_OVERFLOW_BADLY, // pullRequest.offset 越界 maxOffset
    OFFSET_OVERFLOW_ONE, // pullRequest.offset == CQ.maxOffset, 【进行长轮询】
    OFFSET_TOO_SMALL, // pullRequest.offset 越界 minOffset
    NO_MATCHED_LOGIC_QUEUE, // 没有匹配到逻辑队列
    NO_MESSAGE_IN_QUEUE, // 空队列，创建队列也是因为查询导致，【进行长轮询】
}

```

- `switch (response.getCode())`: 根据 response 状态做对应的业务处理
 - `case ResponseCode.SUCCESS`: 查询成功
 - `final byte[] r = this.readGetMessageResult()`: 本次 pull 出来的全部消息导入 byte 数组
 - `response.setBody(r)`: 将消息的 byte 数组保存到 response body 字段
 - `case ResponseCode.PULL_NOT_FOUND`: 产生这种情况大部分原因是 `pullRequest.offset == queue.maxOffset`，说明已经没有需要获取的消息，此时如果直接返回给客户端，客户端会立刻重新请求，还是继续返回该状态，频繁拉取服务器导致服务器压力大，所以此处**需要长轮询**
 - `if (brokerAllowSuspend && hasSuspendFlag)`: brokerAllowSuspend = true，当长轮询结束再次执行 processRequest 时该参数为 false，所以**每次 Pull 请求至多在服务器端长轮询控制一次**

- o `PullRequest pullRequest = new PullRequest()`: 创建长轮询 PullRequest 对象
 - o `this.brokerController...suspendPullRequest(topic, queueId, pullRequest)`: 将长轮询请求对象交给长轮询服务
 - `String key = this.buildKey(topic, queueId)`: 构建一个 `topic@queueId` 的 key
 - `ManyPullRequest mpr = this.pullRequestTable.get(key)`: 从拉请求表中获取对象
 - `mpr.addPullRequest(pullRequest)`: 将 `PullRequest` 对象放入到长轮询的请求集合中
 - o `response = null`: 响应设置为 null 内部的 callBack 就不会给客户端发送任何数据, 不进行通信, 否则就又开始重新请求
 - `boolean storeOffsetEnable`: 允许长轮询、sysFlag 表示提交消费者本地该队列的 offset、当前 broker 节点角色为 master 节点三个条件成立, 才在 Broker 端存储消费者组内该主题的指定 queue 的消费进度
 - `return response`: 返回 response, 不为 null 时外层 processRequestCommand 的 callback 会将数据写给客户端
-

长轮询

`PullRequestHoldService` 类负责长轮询, `BrokerController#start` 方法中调用了 `this.pullRequestHoldService.start()` 启动该服务

核心方法:

- `run()`: 核心运行方法

```
public void run() {
    // 循环运行
    while (!this.isStopped()) {
        if (this.brokerController.getBrokerConfig().isLongPollingEnable()) {
            // 服务器开启长轮询开关: 每次循环休眠5秒
            this.waitForRunning(5 * 1000);
        } else {
            // 服务器关闭长轮询开关: 每次循环休眠1秒
            this.waitForRunning(...);
        }
        // 检查持有的请求
        this.checkHoldRequest();
        // ....
    }
}
```

- `checkHoldRequest()`: 检查所有的请求
 - o `for (String key : this.pullRequestTable.keySet())`: 处理所有的 `topic@queueId` 的逻辑
 - o `String[] kArray = key.split(TOPIC_QUEUEID_SEPARATOR)`: key 按照 @ 拆分, 得到 topic 和 queueId

- o `long offset = this...getMaxOffsetInQueue(topic, queueId)`: 到存储模块查询该 ConsumeQueue 的**最大 offset**
 - o `this.notifyMessageArriving(topic, queueId, offset)`: 通知消息到达
 - `notifyMessageArriving()`: 通知消息到达的逻辑, ReputMessageService 消息分发服务也会调用该方法
 - o `ManyPullRequest mpr = this.pullRequestTable.get(key)`: 获取对应的的 manyPullRequest 对象
 - o `List<PullRequest> requestList`: 获得该队列下的所有 PullRequest, 并进行遍历
 - o `List<PullRequest> replayList`: 当某个 pullRequest 不超时, 并且对应的 `CQ.maxOffset <= pullRequest.offset`, 就将该 PullRequest 再放入该列表
 - o `long newestoffset`: 该值为 CQ 的 maxOffset
 - o `if (newestOffset > request.getPullFromThisOffset())`: 请求对应的队列内可以 pull 消息了, 结束长轮询
 - o `boolean match`: 进行过滤匹配
 - o `this.brokerController...executeRequestWhenWakeup()`: 将满足条件的 pullRequest 再次提交到线程池内执行
 - `final RemotingCommand response`: 执行 processRequest 方法, 并且**不会触发长轮询**
 - `channel.writeAndFlush(response).addListene()`: 将结果数据发送给客户端
 - o `if (System.currentTimeMillis() >= ...)`: 判断该 pullRequest 是否超时, 超时后的也是重新提交到线程池, 并且不进行长轮询
 - o `mpr.addPullRequest(replayList)`: 将未满足条件的 PullRequest 对象再次添加到 ManyPullRequest 属性中
-

结果类

GetMessageResult 类成员信息:

```
public class GetMessageResult {
    // 查询消息时, 最底层都是 mappedFile 支持的查询, 查询时返回给外层一个
    SelectMappedBufferResult,
    // mappedFile 每查询一次都会 refCount++, 通过SelectMappedBufferResult持有
    mappedFile, 完成资源释放的句柄
    private final List<SelectMappedBufferResult> messageMapedList =
        new ArrayList<SelectMappedBufferResult>(100);

    // 该List内存储消息, 每一条消息都被转成 ByteBuffer 表示了
    private final List<ByteBuffer> messageBufferList = new ArrayList<ByteBuffer>
(100);
    // 查询结果状态
    private GetMessageStatus status;
    // 客户端下次再向当前Queue拉消息时, 使用的 offset
    private long nextBeginOffset;
    // 当前queue最小offset
    private long minOffset;
```

```
// 当前queue最大offset  
private long maxOffset;  
// 消息总byte大小  
private int bufferTotalSize = 0;  
// 服务器建议客户端下次到该 queue 拉消息时是否使用 【从节点】  
private boolean suggestPullingFromSlave = false;  
}
```

队列快照

成员属性

ProcessQueue 类是消费队列的快照

成员变量：

- 属性字段：

```
private final AtomicLong msgCount = new AtomicLong(); // 队列中消息数量  
private final AtomicLong msgSize = new AtomicLong(); // 消息总大小  
private volatile long queueOffsetMax = 0L; // 快照中最大 offset  
private volatile boolean dropped = false; // 快照是否移除  
private volatile long lastPullTimestamp = current; // 上一次拉消息的时间  
private volatile long lastConsumeTimestamp = current; // 上一次消费消息的时间  
private volatile long lastLockTimestamp = current; // 上一次获取锁的时间
```

- 消息容器：key 是消息偏移量，val 是消息

```
private final TreeMap<Long, MessageExt> msgTreeMap = new TreeMap<Long,  
MessageExt>();
```

- 顺序消费临时容器：

```
private final TreeMap<Long, MessageExt> consumingMsgOrderlyTreeMap = new  
TreeMap<Long, MessageExt>();
```

- 锁：

```
private final ReadwriteLock lockTreeMap; // 读写锁  
private final Lock lockConsume; // 重入锁，【顺序消费使用】
```

- 顺序消费状态：

```
private volatile boolean locked = false; // 是否是锁定状态  
private volatile boolean consuming = false; // 是否是消费中
```

成员方法

核心成员方法

- `putMessage()`: 将 Broker 拉取下来的 msgs 存储到快照队列内，返回为 true 表示提交顺序消费任务，false 表示不提交

```
public boolean putMessage(final List<MessageExt> msgs)
```

- `this.lockTreeMap.writeLock().lockInterruptibly()`: 获取写锁
 - `for (MessageExt msg : msgs)`: 遍历 msgs 全部加入 msgTreeMap, key 是消息的 queueOffset
 - `if (!msgTreeMap.isEmpty() && !this.consuming)`: **消息容器中存在未处理的消息，并且不是消费中的状态**
 - `dispatchToConsume = true`: 代表需要提交顺序消费任务
 - `this.consuming = true`: 设置为顺序消费执行中的状态
 - `this.lockTreeMap.writeLock().unlock()`: 释放写锁
- `removeMessage()`: 移除已经消费的消息，参数是已经消费的消息集合，并发消费使用

```
public long removeMessage(final List<MessageExt> msgs)
```

- `long result = -1`: 结果初始化为 -1
 - `this.lockTreeMap.writeLock().lockInterruptibly()`: 获取写锁
 - `this.lastConsumeTimestamp = now`: 更新上一次消费消息的时间为现在
 - `if (!msgTreeMap.isEmpty())`: 判断消息容器是否是空，**是空直接返回 -1**
 - `result = this.queueOffsetMax + 1`: 设置结果，**删除完后消息容器为空时返回**
 - `for (MessageExt msg : msgs)`: 将已经消费的消息全部从 msgTreeMap 移除
 - `if (!msgTreeMap.isEmpty())`: 移除后容器内还有待消费的消息，**获取第一条消息 offset 返回**
 - `this.lockTreeMap.writeLock().unlock()`: 释放写锁
- `takeMessages()`: 获取一批消息，顺序消费使用

```
public List<MessageExt> takeMessages(final int batchsize)
```

- `this.lockTreeMap.writeLock().lockInterruptibly()`: 获取写锁
 - `this.lastConsumeTimestamp = now`: 更新上一次消费消息的时间为现在
 - `for (int i = 0; i < batchsize; i++)`: 从头节点开始获取消息
 - `result.add(entry.getValue())`: 将消息放入结果集合
 - `consumingMsgOrderlyTreeMap.put()`: 将消息加入顺序消费容器中
 - `if (result.isEmpty())`: 条件成立说明顺序消费容器本地快照内的消息全部处理完了，**当前顺序消费任务需要停止**
 - `consuming = false`: 消费状态置为 false
 - `this.lockTreeMap.writeLock().unlock()`: 释放写锁
- `commit()`: 处理完一批消息后调用，顺序消费使用

```
public long commit()
```

- `this.lockTreeMap.writeLock().lockInterruptibly()`: 获取写锁
- `Long offset = this.consumingMsgOrderlyTreeMap.lastKey()`: 获取顺序消费临时容器最后一条数据的 key
- `msgCount, msgSize`: 更新顺序消费相关的字段
- `this.consumingMsgOrderlyTreeMap.clear()`: 清空顺序消费容器的数据
- `return offset + 1`: 消费者下一条消费的位点
- `this.lockTreeMap.writeLock().unlock()`: 释放写锁
- `cleanExpiredMsg()`: 清除过期消息

```
public void cleanExpiredMsg(DefaultMQPushConsumer pushConsumer)
```

- `if (pushConsumer.getDefaultMQPushConsumerImpl().isConsumeOrderly())`: 顺序消费不执行过期清理逻辑
- `int loop = msgTreeMap.size() < 16 ? msgTreeMap.size() : 16`: 最多循环 16 次
- `if (!msgTreeMap.isEmpty() &&`: 如果容器中第一条消息的消费开始时间与当前系统时间差值 > 15min, 则取出该消息
- `else`: 直接跳出循环, 因为快照队列内的消息是有顺序的, 第一条消息不过期, 其他消息都不过期
- `pushConsumer.sendMessageBack(msg, 3)`: 消息回退到服务器, 设置该消息的延迟级别为 3
- `if (!msgTreeMap.isEmpty() && msg.getQueueOffset() == msgTreeMap.firstKey())`: 条件成立说明消息回退期间, 该目标消息并没有被消费任务成功消费
- `removeMessage(Collections.singletonList(msg))`: 从 treeMap 将该回退成功的 msg 删除

并发消费

成员属性

`ConsumeMessageConcurrentlyService` 负责并发消费服务

成员变量:

- 消息监听器: 封装处理消息的逻辑, 该监听器由开发者实现, 并注册到 `defaultMQPushConsumer`

```
private final MessageListenerConcurrently messageListener;
```

- 消费属性:

```
private final BlockingQueue<Runnable> consumeRequestQueue; // 消费任务队列
private final String consumerGroup; // 消费者组
```

- 线程池:

```
private final ThreadPoolExecutor consumeExecutor; // 消费任务线程池, 默认 20
private final ScheduledExecutorService scheduledExecutorService; // 调度线程池, 延迟提交消费任务
private final ScheduledExecutorService cleanExpireMsgExecutors; // 清理过期消息任务线程池, 15min 一次
```

成员方法

ConsumeMessageConcurrentlyService 并发消费核心方法

- start(): 启动消费服务, DefaultMQPushConsumerImpl 启动时会调用该方法

```
public void start() {
    // 提交“清理过期消息任务”任务, 延迟15min之后执行, 之后每15min执行一次
    this.cleanExpireMsgExecutors.scheduleAtFixedRate(() ->
        cleanExpireMsg(),
        15, 15,
        TimeUnit.MINUTES);
}
```

- cleanExpireMsg(): 清理过期消息任务

```
private void cleanExpireMsg()
```

- Iterator<Map.Entry<MessageQueue, ProcessQueue>> it: 获取分配给当前消费者的队列
 - while (it.hasNext()): 遍历所有的队列
 - pq.cleanExpiredMsg(this.defaultMQPushConsumer): 调用队列快照 ProcessQueue 清理过期消息的方法
- submitConsumeRequest(): 提交消费请求

```
// 参数一: 从服务器 pull 下来的这批消息
// 参数二: 消息归属 mq 在消费者端的 processQueue, 提交消费任务之前, msgs已经加入到该pq 内了
// 参数三: 消息归属队列
// 参数四: 并发消息此参数无效
public void submitConsumeRequest(List<MessageExt> msgs, ProcessQueue
processQueue, MessageQueue messageQueue, boolean dispatchToConsume)
```

- final int consumeBatchSize: 一个消费任务可消费的消息数量, 默认为 1
- if (msgs.size() <= consumeBatchSize): 判断一个消费任务是否可以提交
ConsumeRequest consumeRequest: 封装为消费请求
this.consumeExecutor.submit(consumeRequest): 提交消费任务, 异步执行消息的处理
- else: 说明消息较多, 需要多个消费任务
for (int total = 0; total < msgs.size();): 将消息拆分成多个消费任务

- processConsumeResult(): 处理消费结果

```
// 参数一：消费结果状态； 参数二：消费上下文； 参数三：当前消费任务
public void processConsumeResult(status, context, consumeRequest)
```

- switch (status) : 根据消费结果状态进行处理
- case CONSUME_SUCCESS : 消费成功
 - `if (ackIndex >= consumeRequest.getMsgs().size())`: 消费成功的话, ackIndex 设置成 `消费消息数 - 1` 的值, 比如有 5 条消息, 这里就设置为 4
 - `ok, failed`: ok 设置为消息数量, failed 设置为 0
- case RECONSUME_LATER : 消费失败
 - `ackIndex = -1`: 设置为 -1
- switch (this.defaultMQPushConsumer.getMessageModel()): 判断消费模式, 默认是 **集群模式**
- for (int i = ackIndex + 1; i < msgs.size(); i++) : 当消费失败时 ackIndex 为 -1, i 的起始值为 0, 该消费任务内的**全部消息**都会尝试回退给服务器
- MessageExt msg : 提取一条消息
- boolean result = this.sendMessageBack(msg, context) : **发送消息回退, 同步发送**
- if (!result) : 回退失败的消息, 将**消息的重试属性加 1**, 并加入到回退失败的集合
- if (!msgBackFailed.isEmpty()): 回退失败集合不为空
 - `consumeRequest.getMsgs().removeAll(msgBackFailed)`: 将回退失败的消息从当前消费任务的 msgs 集合内移除
 - `this.submitConsumeRequestLater()`: **回退失败的消息会再次提交消费任务**, 延迟 5 秒钟后再次尝试消费
- `long offset = ...removeMessage(msgs)`: 从 pq 中删除已经消费成功的消息, 返回 offset
- `this...getOffsetStore().updateOffset()`: 更新消费者本地该 mq 的**消费进度**

消费请求

ConsumeRequest 是 ConsumeMessageConcurrentlyService 的内部类, 是一个 Runnable 任务对象
成员变量:

- 分配到该消费任务的消息:

```
private final List<MessageExt> msgs;
```

- 消息队列:

```
private final ProcessQueue processQueue; // 消息处理队列
private final MessageQueue messageQueue; // 消息队列
```

核心方法：

- run(): 执行任务

```
public void run()
```

- if (this.processQueue.isDropped()): 条件成立说明该 queue 经过 rbl 算法分配到其他的 consumer
- MessageListenerConcurrently listener: 获取消息监听器
- ConsumeConcurrentlyContext context: 创建消费上下文对象
- defaultMQPushConsumerImpl.resetRetryAndNamespace(): 重置重试标记
 - final String groupTopic: 获取当前消费者组的重试主题 %RETRY%GroupName
 - for (MessageExt msg : msgs): 遍历所有的消息
 - String retryTopic = msg.getProperty(...): 原主题，一般消息没有该属性，只有被重复消费的消息才有
 - if (retryTopic != null && groupTopic.equals(...)): 条件成立说明该消息是被重复消费的消息
 - msg.setTopic(retryTopic): 将被重复消费的消息主题修改回原主题
- if (ConsumeMessageConcurrentlyService...hasHook()): 前置处理
- boolean hasException = false: 消费过程中，是否向外抛出异常
- MessageAccessor.setConsumeStartTimeStamp(): 给每条消息设置消费开始时间
- status = listener.consumeMessage(collections.unmodifiableList(msgs), context): **消费消息**
- if (ConsumeMessageConcurrentlyService...hasHook()): 后置处理
- ...processConsumeResult(status, context, this): **处理消费结果**

顺序消费

成员属性

ConsumeMessageOrderlyService 负责顺序消费服务

成员变量：

- 消息监听器：封装处理消息的逻辑，该监听器由开发者实现，并注册到 defaultMQPushConsumer

```
private final MessageListenerOrderly messageListener;
```

- 消费属性：

```
private final BlockingQueue<Runnable> consumeRequestQueue; // 消费任务队列
private final String consumerGroup; // 消费者组
private volatile boolean stopped = false; // 消费停止状态
```

- 线程池:

```
private final ThreadPoolExecutor consumeExecutor; // 消费任务线程池
private final ScheduledExecutorService scheduledExecutorService; // 调度线程池, 延迟提交消费任务
```

- 队列锁: 消费者本地 MQ 锁, 确保本地对于需要顺序消费的 MQ 同一时间只有一个任务在执行

```
private final MessageQueueLock messageQueueLock = new MessageQueueLock();
```

```
public class MessageQueueLock {
    private ConcurrentHashMap<MessageQueue, Object> mqLockTable = new ConcurrentHashMap<MessageQueue, Object>();
    // 获取本地队列锁对象
    public Object fetchLockObject(final MessageQueue mq) {
        Object objLock = this.mqLockTable.get(mq);
        if (null == objLock) {
            objLock = new Object();
            Object prevLock = this.mqLockTable.putIfAbsent(mq, objLock);
            if (prevLock != null) {
                objLock = prevLock;
            }
        }
        return objLock;
    }
}
```

已经获取了 Broker 端该 Queue 的独占锁, 为什么还要获取本地队列锁对象? (这里我也没太懂, 先记录下来, 本地多线程?)

- Broker queue 占用锁的角度是 Client 占用, Client 从 Broker 的某个占用了锁的 queue 拉取下来消息以后, 将消息存储到消费者本地的 ProcessQueue 中, 快照对象的 consuming 属性置为 true, 表示本地的队列正在消费处理中
- ProcessQueue 调用 takeMessages 方法时会获取下一批待处理的消息, 获取不到会修改 consuming = false, 本消费任务马上停止。
- 如果此时 Pull 再次拉取一批当前 ProcessQueue 的 msg, 会再次向顺序消费服务提交消费任务, 此时需要本地队列锁对象同步本地线程

成员方法

- start(): 启动消费服务, DefaultMQPushConsumerImpl 启动时会调用该方法

```
public void start()
```

- this.scheduledExecutorService.scheduleAtFixedRate(): 提交锁续约任务, 延迟 1 秒执行, 周期为 20 秒钟
- ConsumeMessageOrderlyService.this.lockMQPeriodically(): 锁续约任务

- `this.defaultMQPushConsumerImpl.getRebalanceImpl().lockAll()`: 对消费者的
所有队列进行续约
- `submitConsumeRequest()`: 提交消费任务请求

```
// 参数: true 表示创建消费任务并提交, false不创建消费任务, 说明消费者本地已经有消费任务在  
执行了
public void submitConsumeRequest(..., final boolean dispathToConsume) {
    if (dispathToConsume) {
        // 当前进程中不存在 顺序消费任务, 创建新的消费任务, 【提交到消费任务线程池】
        ConsumeRequest consumeRequest = new ConsumeRequest(processQueue,
messageQueue);
        this.consumeExecutor.submit(consumeRequest);
    }
}
```

- `processConsumeResult()`: 消费结果处理

<code>// 参数1: msgs 本轮循环消费的消息集合</code>	参数2: <code>status</code> 消费状态
<code>// 参数3: context 消费上下文</code>	参数4: 消费任务
<code>// 返回值: boolean 决定是否继续循环处理pq内的消息</code>	
<code>public boolean processConsumeResult(final List<MessageExt> msgs, final ConsumeOrderlyStatus status, final ConsumeOrderlyContext context, final ConsumeRequest consumeRequest)</code>	

- `if (context.isAutoCommit())`: 默认自动提交
- `switch (status)`: 根据消费状态进行不同的处理
 - `case SUCCESS`: 消费成功
 - `commitOffset = ...commit()`: 调用 pq 提交方法, 会将本次循环处理的消息从顺序消费 map 删
除, 并且返回消息进度
 - `case SUSPEND_CURRENT_QUEUE_A_MOMENT`: 挂起当前队列
 - `consumeRequest.getProcessQueue().makeMessageToConsumeAgain(msgs)`: 回滚消息
 - `for (MessageExt msg : msgs)`: 遍历所有的消息
 - `this.consumingMsgOrderlyTreeMap.remove(msg.getQueueOffset())`: 从顺序消
费临时容器中移除
 - `this.msgTreeMap.put(msg.getQueueOffset(), msg)`: 添加到消息容器
 - `this.submitConsumeRequestLater()`: 再次提交消费任务, 1 秒后执行
 - `continueConsume = false`: 设置为 false, 外层会退出本次的消费任务
 - `this.defaultMQPushConsumerImpl.getOffsetStore().updateOffset(...)`: 更新本地
消费进度

消费请求

ConsumeRequest 是 ConsumeMessageOrderlyService 的内部类，是一个 Runnable 任务对象

核心方法：

- run(): 执行任务

```
public void run()
```

- final Object objLock : 获取本地锁对象
- synchronized (objLock) : 本地队列锁，确保每个 MQ 的消费任务只有一个在执行，**确保顺序消费**
- if(.. || (this.processQueue.isLocked() && !this.processQueue.isLockExpired())) : 当前队列持有分布式锁，并且锁未过期，持锁时间超过 30 秒算过期
- final long beginTime : 消费开始时间
- for (boolean continueConsume = true; continueConsume;) : 根据是否继续消费的标记判断是否继续
- final int consumeBatchSize : 获取每次循环处理的消息数量，一般是 1
- List<MessageExt> msgs = this...takeMessages(consumeBatchsize) : 到处理队列获取一批消息
- if (!msgs.isEmpty()) : 获取到了待消费的消息
 final ConsumeOrderlyContext context : 创建消费上下文对象
 this.processQueue.getLockConsume().lock() : 获取 lockConsume 锁，与 RBL 线程同步使用
 status = messageListener.consumeMessage(...) : 监听器处理消息
 this.processQueue.getLockConsume().unlock() : 释放 lockConsume 锁
 if (null == status) : 处理消息状态返回 null，设置状态为挂起当前队列
 continueConsume = ...processConsumeResult() : 消费结果处理
- else : 获取到的消息是空
 continueConsume = false : 结束任务循环
- else : 当前队列未持有分布式锁，或者锁过期
 ConsumeMessageOrderlyService.this.tryLockLaterAndReconsume() : 重新提交任务，根据是否获取到队列锁，选择延迟 10 毫秒或者 300 毫秒

生产消费

生产流程：

- 首先获取当前消息主题的发布信息，获取不到去 Namesrv 获取（默认有 TBW102），并将获取到的路由数据转化为发布数据，**创建 MQ 队列在多个 Broker 组**（一组代表一主多从的 Broker 架构），客户端实例同样更新订阅数据，创建 MQ 队列，放入负载均衡服务 topicSubscribeInfoTable 中
- 然后从发布数据中选择一个 MQ 队列发送消息
- Broker 端通过 SendMessageProcessor 对发送的消息进行持久化处理，存储到 CommitLog。将重试次数过多的消息加入**死信队列**，将延迟消息的主题和队列修改为调度主题和调度队列 ID
- Broker 启动 ScheduleMessageService 服务会为每个延迟级别创建一个延迟任务，让延迟消息得到有效的处理，将到达交付时间的消息修改为原始主题的原始 ID 存入 CommitLog，消费者就可以进行消费了

消费流程：

- 消息消费队列 ConsumerQueue 存储消息在 CommitLog 的索引，消费者通过该队列来读取消息实体内容，一个 MQ 就对应一个 CQ
- 首先通过负载均衡服务，将分配到当前消费者实例的 MQ 创建 PullRequest，并放入 PullMessageService 的本地阻塞队列内
- PullMessageService 循环从阻塞队列获取请求对象，发起拉消息请求，并创建 PullCallback 回调对象，将正常拉取的消息提交到**消费任务线程池**，并设置请求的下一次拉取位点，重新放入阻塞队列，形成闭环
- 消费任务服务对消费失败的消息进行回退，通过内部生产者实例发送回退消息，回退失败的消息会再次提交消费任务重新消费
- Broker 端对拉取消息的请求进行处理（processRequestCommand），查询成功将消息放入响应体，通过 Netty 写回客户端，当 `pullRequest.offset == queue.maxOffset` 说明该队列已经没有需要获取的消息，将请求放入长轮询集合等待有新消息
- PullRequestHoldService 负责长轮询，每 5 秒遍历一次长轮询集合，将满足条件的 PullRequest 再次提交到线程池内处理

Zookeeper

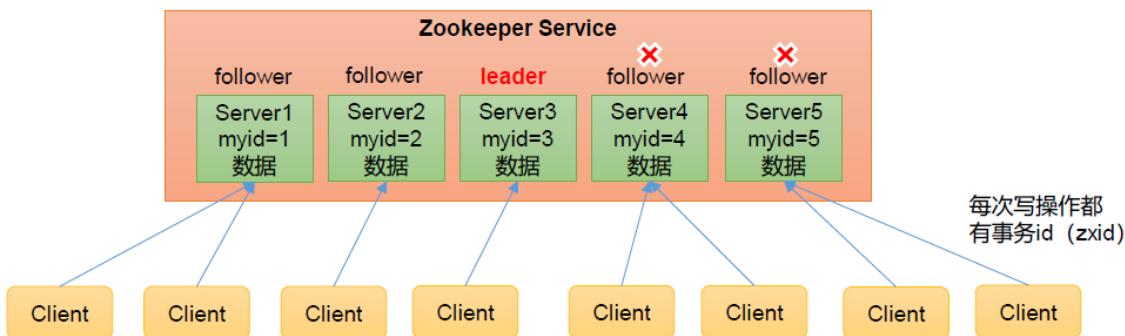
基本介绍

框架特征

Zookeeper 是 Apache Hadoop 项目子项目，为分布式框架提供协调服务，是一个树形目录服务

Zookeeper 是基于观察者模式设计的分布式服务管理框架，负责存储和管理共享数据，接受观察者的注册监控，一旦这些数据的状态发生变化，Zookeeper 会通知观察者

- Zookeeper 是一个领导者 (Leader)，多个跟随者 (Follower) 组成的集群
- 集群中只要有半数以上节点存活就能正常服务，所以 Zookeeper 适合部署奇数台服务器
- **全局数据一致**，每个 Server 保存一份相同的数据副本，Client 无论连接到哪个 Server，数据都是一致
- 更新的请求顺序执行，来自同一个 Client 的请求按其发送顺序依次执行
- **数据更新原子性**，一次数据更新要么成功，要么失败
- 实时性，在一定的时间范围内，Client 能读到最新数据
- 心跳检测，会定时向各个服务提供者发送一个请求（实际上建立的是一个 Socket 长连接）



参考视频：<https://www.bilibili.com/video/BV1to4y1C7gw>

应用场景

Zookeeper 提供的主要功能包括：统一命名服务、统一配置管理、统一集群管理、服务器节点动态上下线、软负载均衡、分布式锁等

- 在分布式环境中，经常对应用/服务进行统一命名，便于识别，例如域名相对于 IP 地址更容易被接收

```
/service/www.baidu.com      # 节点路径  
192.168.1.1 192.168.1.2    # 节点值
```

如果在节点中记录每台服务器的访问数，让访问数最少的服务器去处理最新的客户端请求，可以实现负载均衡

```
192.168.1.1 10 # 次数  
192.168.1.1 15
```

- 配置文件同步可以通过 Zookeeper 实现，将配置信息写入某个 ZNode，其他客户端监视该节点，当节点数据被修改，通知各个客户端服务器
 - 集群环境中，需要实时掌握每个集群节点的状态，可以将这些信息放入 ZNode，通过监控通知的机制实现
 - 实现客户端实时观察服务器上下线的变化，通过心跳检测实现
-

基本操作

安装搭建

安装步骤：

- 安装 JDK
- 拷贝 apache-zookeeper-3.5.7-bin.tar.gz 安装包到 Linux 系统下，并解压到指定目录
- conf 目录下的配置文件重命名：

```
mv zoo_sample.cfg zoo.cfg
```

- 修改配置文件：

```
vim zoo.cfg
# 修改内容
dataDir=/home/seazean/software/zookeeper-3.5.7/zkData
```

- 在对应目录创建 zkData 文件夹：

```
mkdir zkData
```

Zookeeper 中的配置文件 zoo.cfg 中参数含义解读：

- tickTime = 2000：通心跳时间，Zookeeper 服务器与客户端心跳时间，单位毫秒
 - initLimit = 10：Leader 与 Follower 初始通信时限，初始连接时能容忍的最多心跳次数
 - syncLimit = 5：Leader 与 Follower 同步通信时限，LF 通信时间超过 syncLimit * tickTime，Leader 认为 Follower 下线
 - dataDir：保存 Zookeeper 中的数据目录，默认是 tmp 目录，容易被 Linux 系统定期删除，所以建议修改
 - clientPort = 2181：客户端连接端口，通常不做修改
-

操作命令

服务端

Linux 命令：

- 启动 ZooKeeper 服务: `./zkserver.sh start`
- 查看 ZooKeeper 服务: `./zkServer.sh status`
- 停止 ZooKeeper 服务: `./zkServer.sh stop`
- 重启 ZooKeeper 服务: `./zkServer.sh restart`
- 查看进程是否启动: `jps`

客户端

Linux 命令：

- 连接 ZooKeeper 服务端：

```
./zkcli.sh          # 直接启动  
./zkcli.sh -server ip:port # 指定 host 启动
```

客户端命令：

- 基础操作：

```
quit              # 停止连接  
help              # 查看命令帮助
```

- 创建命令：`/` 代表根目录

```
create /path value      # 创建节点, value 可选  
create -e /path value   # 创建临时节点  
create -s /path value   # 创建顺序节点  
create -es /path value  # 创建临时顺序节点, 比如node10000012 删除12后也会继续从  
13开始, 只会增加
```

- 查询命令：

```
ls /path           # 显示指定目录下子节点  
ls -s /path       # 查询节点详细信息  
ls -w /path       # 监听子节点数量的变化  
stat /path        # 查看节点状态  
get -s /path      # 查询节点详细信息  
get -w /path      # 监听节点数据的变化
```

```

# 属性，分为当前节点的属性和子节点属性
czxid: 节点被创建的事务ID，是ZooKeeper中所有修改总的次序，每次修改都有唯一的zxid，谁小谁先发生
ctime: 被创建的时间戳
mzxid: 最后一次被更新的事务ID
mtime: 最后修改的时间戳
pzxid: 子节点列表最后一次被更新的事务ID
cversion: 子节点的变化号，修改次数
dataversion: 节点的数据变化号，数据的变化次数
aclversion: 节点的访问控制列表变化号
ephemeralOwner: 用于临时节点，代表节点拥有者的 session id，如果为持久节点则为0
dataLength: 节点存储的数据的长度
numChildren: 当前节点的子节点数量

```

- 删除命令：

```

delete /path          # 删除节点
deleteall /path       # 递归删除节点

```

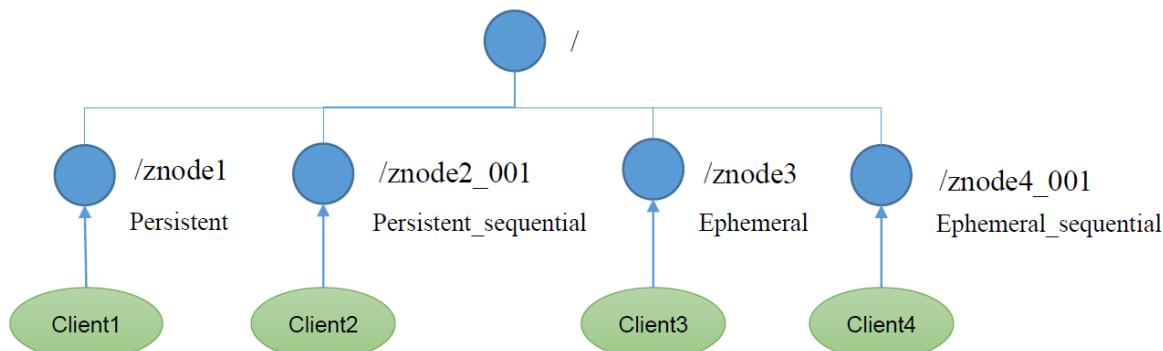
数据结构

ZooKeeper 是一个树形目录服务，类似 Unix 的文件系统，每一个节点都被称为 ZNode，每个 ZNode 默认存储 1MB 的数据，节点上会保存数据和节点信息，每个 ZNode 都可以通过其路径唯一标识

节点可以分为四大类：

- PERSISTENT: 持久化节点
- EPHEMERAL: 临时节点，客户端和服务端断开连接后，创建的节点删除
- PERSISTENT_SEQUENTIAL: 持久化顺序节点，创建 znode 时设置顺序标识，节点名称后会附加一个值，**顺序号是一个单调递增的计数器**，由父节点维护
- EPHEMERAL_SEQUENTIAL: 临时顺序节点

注意：在分布式系统中，顺序号可以被用于为所有的事件进行全局排序，这样客户端可以通过顺序号推断事件的顺序



代码实现

添加 Maven 依赖:

```
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.5.7</version>
</dependency>
```

实现代码:

```
public static void main(String[] args) {
    // 参数一: 连接地址
    // 参数二: 会话超时时间
    // 参数三: 监听器
    ZooKeeper zkClient = new ZooKeeper("192.168.3.128:2181", 20000, new
watcher() {
    @Override
    public void process(WatchedEvent event) {
        System.out.println("监听处理函数");
    }
});
}
```

集群介绍

相关概念

Zookeepe 集群三个角色:

- Leader 领导者: 处理客户端**事务请求**, 负责集群内部各服务器的调度
- Follower 跟随者: 处理客户端非事务请求, 转发事务请求给 Leader 服务器, 参与 Leader 选举投票
- Observer 观察者: 观察集群的最新状态的变化, 并将这些状态进行同步; 处理非事务性请求, 事务性请求会转发给 Leader 服务器进行处理; 不会参与任何形式的投票。只提供非事务性的服务, 通常用于在不影响集群事务处理能力的前提下, 提升集群的非事务处理能力 (提高集群读的能力, 但是也降低了集群选主的复杂程度)

相关属性:

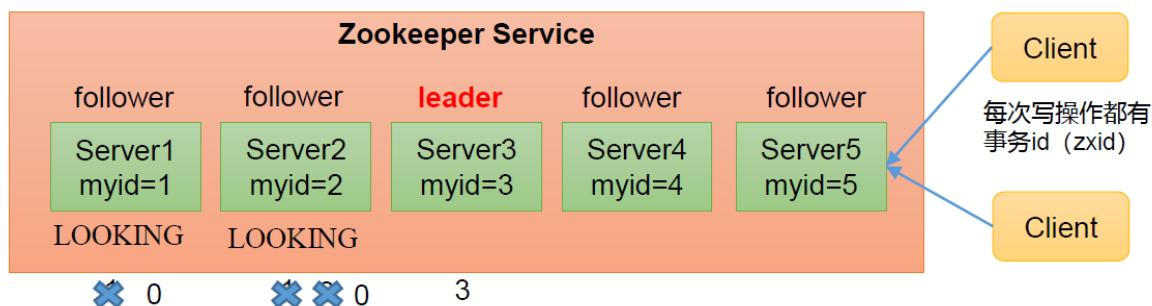
- SID: 服务器 ID, 用来唯一标识一台集群中的机器, 和 myid 一致
- ZXID: 事务 ID, 用来标识一次服务器状态的变更, 在某一时刻集群中每台机器的 ZXID 值不一定完全一致, 这和 ZooKeeper 服务器对于客户端更新请求的处理逻辑有关

- Epoch: 每个 Leader 任期的代号, 同一轮选举投票过程中的该值是相同的, 投完一次票就增加
 - 选举机制: 半数机制, 超过半数的投票就通过
 - 第一次启动选举规则: 投票过半数时, 服务器 ID 大的胜出
 - 第二次启动选举规则:
 - EPOCH 大的直接胜出
 - EPOCH 相同, 事务 ID 大的胜出 (事务 ID 越大, 数据越新)
 - 事务 ID 相同, 服务器 ID 大的胜出
-

初次选举

选举过程:

- 服务器 1 启动, 发起一次选举, 服务器 1 投自己一票, 票数不超过半数, 选举无法完成, 服务器 1 状态保持为 LOOKING
- 服务器 2 启动, 再发起一次选举, 服务器 1 和 2 分别投自己一票并**交换选票信息**, 此时服务器 1 会发现服务器 2 的 SID 比自己投票推举的 (服务器 1) 大, 更改选票为推举服务器 2。投票结果为服务器 1 票数 0 票, 服务器 2 票数 2 票, 票数不超过半数, 选举无法完成, 服务器 1、2 状态保持 LOOKING
- 服务器 3 启动, 发起一次选举, 此时服务器 1 和 2 都会更改选票为服务器 3, 投票结果为服务器 3 票数 3 票, 此时服务器 3 的票数已经超过半数, 服务器 3 当选 Leader, 服务器 1、2 更改状态为 FOLLOWING, 服务器 3 更改状态为 LEADING
- 服务器 4 启动, 发起一次选举, 此时服务器 1、2、3 已经不是 LOOKING 状态, 不会更改选票信息, 交换选票信息结果后服务器 3 为 3 票, 服务器 4 为 1 票, 此时服务器 4 更改选票信息为服务器 3, 并更改状态为 FOLLOWING
- 服务器 5 启动, 同 4 一样



再次选举

ZooKeeper 集群中的一台服务器出现以下情况之一时，就会开始进入 Leader 选举：

- 服务器初始化启动
- 服务器运行期间无法和 Leader 保持连接

当一台服务器进入 Leader 选举流程时，当前集群可能会处于以下两种状态：

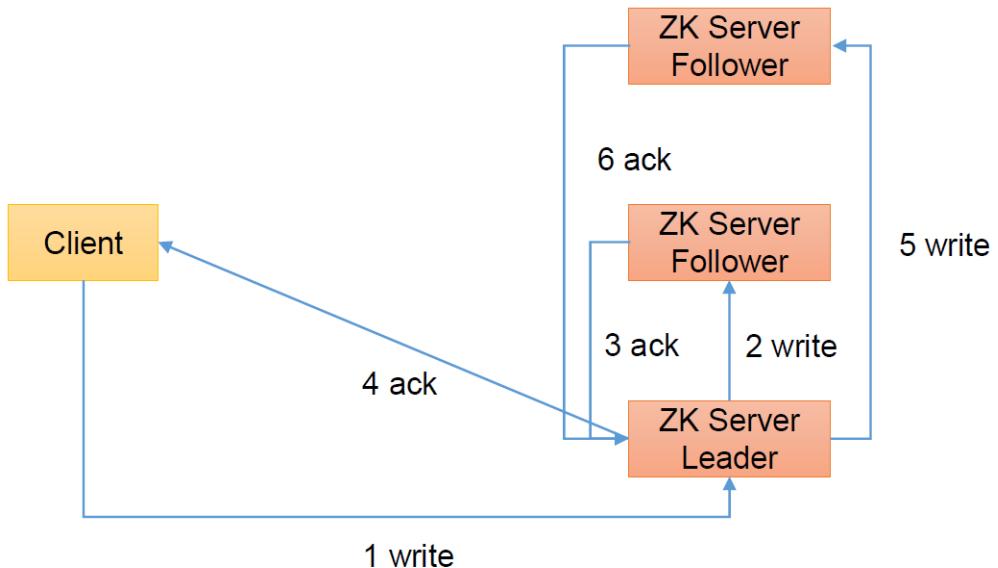
- 集群中本来就已经存在一个 Leader，服务器试图去选举 Leader 时会被告知当前服务器的 Leader 信息，对于该服务器来说，只需要和 Leader 服务器建立连接，并进行状态同步即可
- 集群中确实不存在 Leader，假设服务器 3 和 5 出现故障，开始进行 Leader 选举，SID 为 1、2、4 的机器投票情况

(EPOCH, ZXID, SID): (1, 8, 1), (1, 8, 2), (1, 7, 4)

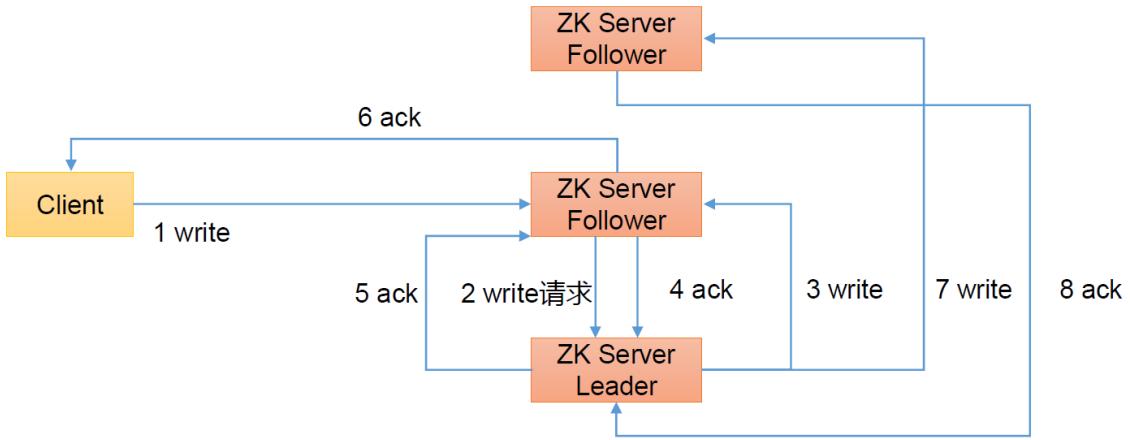
根据选举规则，服务器 2 胜出

数据写入

写操作就是事务请求，写入请求直接发送给 Leader 节点：Leader 会先将数据写入自身，同时通知其他 Follower 写入，**当集群中有半数以上节点写入完成**，Leader 节点就会响应客户端数据写入完成



写入请求直接发送给 Follower 节点：Follower 没有写入权限，会将写请求转发给 Leader，Leader 将数据写入自身，通知其他 Follower 写入，当集群中有半数以上节点写入完成，Leader 会通知 Follower 写入完成，由 Follower 响应客户端数据写入完成



底层协议

Paxos

Paxos 算法：基于消息传递且具有高度容错特性的一致性算法

优点：快速正确的在一个分布式系统中对某个数据值达成一致，并且保证不论发生任何异常，都不会破坏整个系统的一致性

缺陷：在网络复杂的情况下，可能很久无法收敛，甚至陷入活锁的情况

ZAB

算法介绍

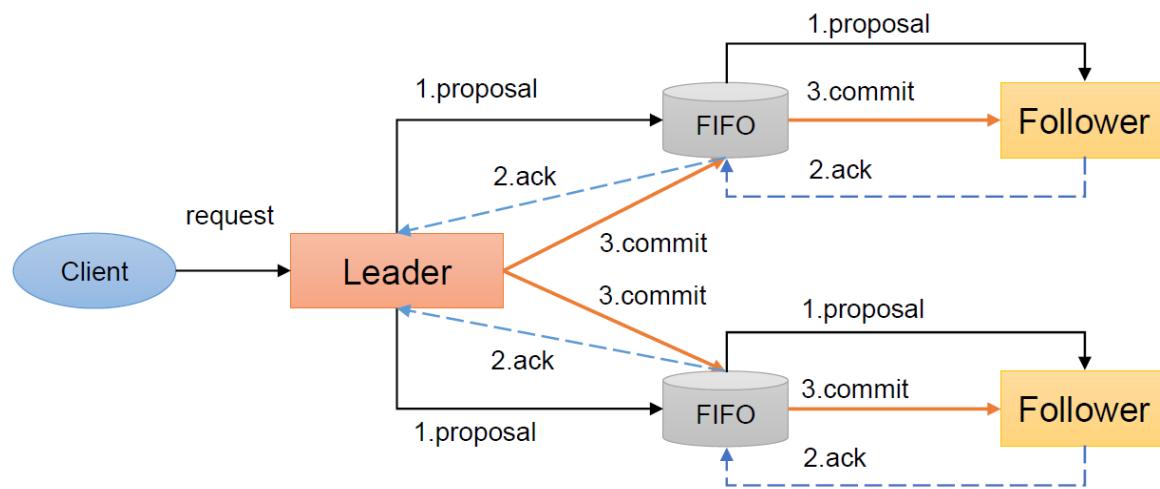
ZAB 协议借鉴了 Paxos 算法，是为 Zookeeper 设计的支持崩溃恢复的原子广播协议，基于该协议 Zookeeper 设计为只有一台客户端（Leader）负责处理外部的写事务请求，然后 Leader 将数据同步到其他 Follower 节点

Zab 协议包括两种基本的模式：消息广播、崩溃恢复

消息广播

ZAB 协议针对事务请求的处理过程类似于一个**两阶段提交**过程：广播事务阶段、广播提交操作

- 客户端发起写操作请求，Leader 服务器将请求转化为事务 Proposal 提案，同时为 Proposal 分配一个全局的 ID，即 ZXID
- Leader 服务器为每个 Follower 分配一个单独的队列，将广播的 Proposal **依次放到队列中去**，根据 FIFO 策略进行消息发送
- Follower 接收到 Proposal 后，将其以事务日志的方式写入本地磁盘中，写入成功后向 Leader 反馈一个 ACK 响应消息
- Leader 接收到超过半数以上 Follower 的 ACK 响应消息后，即认为消息发送成功，可以发送 Commit 消息
- Leader 向所有 Follower 广播 commit 消息，同时自身也会完成事务提交，Follower 接收到 Commit 后，将上一条事务提交



两阶段提交模型可能因为 Leader 宕机带来数据不一致：

- Leader 发起一个事务 Proposal 后就宕机，Follower 都没有 Proposal
- Leader 收到半数 ACK 宕机，没来得及向 Follower 发送 Commit

崩溃恢复

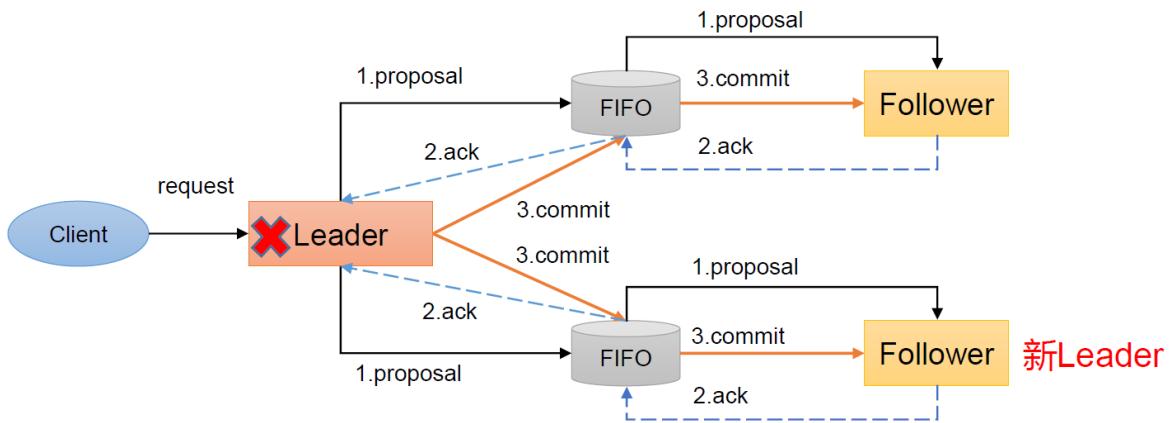
Leader 服务器出现崩溃或者由于网络原因导致 Leader 服务器失去了与**过半 Follower**的联系，那么就会进入崩溃恢复模式，崩溃恢复主要包括两部分：Leader 选举和数据恢复

Zab 协议崩溃恢复要求满足以下两个要求：

- 已经被 Leader 提交的提案 Proposal，必须最终被所有的 Follower 服务器正确提交
- 丢弃已经被 Leader 提出的，但是没有被提交的 Proposal

Zab 协议需要保证选举出来的 Leader 需要满足以下条件：

- 新选举的 Leader 不能包含未提交的 Proposal，即新 Leader 必须都是已经提交了 Proposal 的 Follower 节点
- 新选举的 Leader 节点含有**最大的 ZXID**，可以避免 Leader 服务器检查 Proposal 的提交和丢弃工作



数据恢复阶段：

- 完成 Leader 选举后，在正式开始工作之前（接收事务请求提出新的 Proposal），Leader 服务器会首先确认事务日志中的所有 Proposal 是否已经被集群中过半的服务器 Commit
- Leader 服务器需要确保所有的 Follower 服务器能够接收到每一条事务的 Proposal，并且能将所有已经提交的事务 Proposal 应用到内存数据中，所以只有当 Follower 将所有尚未同步的事务 Proposal 都从 Leader 服务器上同步，并且应用到内存数据后，Leader 才会把该 Follower 加入到真正可用的 Follower 列表中

异常处理

Zab 的事务编号 zxid 设计：

- zxid 是一个 64 位的数字，低 32 位是一个简单的单增计数器，针对客户端每一个事务请求，Leader 在产生新的 Proposal 事务时，都会对该计数器加 1，而高 32 位则代表了 Leader 周期的 epoch 编号
- epoch 为当前集群所处的代或者周期，每次 Leader 变更后都会在 epoch 的基础上加 1，Follower 只服从 epoch 最高的 Leader 命令，所以旧的 Leader 崩溃恢复之后，其他 Follower 就不会继续追随
- 每次选举产生一个新的 Leader，就会从新 Leader 服务器上取出本地事务日志中最大编号 Proposal 的 zxid，从 zxid 中解析得到对应的 epoch 编号，然后再对其加 1 后作为新的 epoch 值，并将低 32 位数字归零，由 0 开始重新生成 zxid

Zab 协议通过 epoch 编号来区分 Leader 变化周期，能够有效避免不同的 Leader 错误的使用了相同的 zxid 编号提出了不一样的 Proposal 的异常情况

Zab 数据同步过程：数据同步阶段要以 Leader 服务器为准

- 一个包含了上个 Leader 周期中尚未提交过的事务 Proposal 的服务器启动时，这台机器加入集群中会以 Follower 角色连上 Leader
- Leader 会根据自己服务器上最后提交的 Proposal 和 Follower 服务器的 Proposal 进行比对，让 Follower 进行一个回退或者前进操作，到一个已经被集群中过半机器 Commit 的最新 Proposal （源码解析部分详解）

CAP

CAP 理论指的是在一个分布式系统中，Consistency（一致性）、Availability（可用性）、Partition Tolerance（分区容错性）不能同时成立，ZooKeeper 保证的是 CP

- ZooKeeper 不能保证每次服务请求的可用性，在极端环境下可能会丢弃一些请求，消费者程序需要重新请求才能获得结果
- 进行 Leader 选举时**集群都是不可用**

CAP 三个基本需求，因为 P 是必须的，因此分布式系统选择就在 CP 或者 AP 中：

- 一致性：指数据在多个副本之间是否能够保持数据一致的特性，当一个系统在数据一致的状态下执行更新操作后，也能保证系统的数据仍然处于一致的状态
- 可用性：指系统提供的服务必须一直处于可用的状态，即使集群中一部分节点故障，对于用户的每一个操作请求总是能够在有限的时间内返回结果
- 分区容错性：分布式系统在遇到任何网络分区故障时，仍然能够保证对外提供服务，不会宕机，除非是整个网络环境都发生了故障

监听机制

实现原理

ZooKeeper 中引入了 Watcher 机制来实现了发布/订阅功能，客户端注册监听目录节点，在特定事件触发时，ZooKeeper 会通知所有关注该事件的客户端，保证 ZooKeeper 保存的任何的数据的任何改变都能快速的响应到监听应用程序

监听命令：**只能生效一次**，接收一次通知，再次监听需要重新注册

```
ls -w /path          # 监听【子节点数量】的变化  
get -w /path        # 监听【节点数据】的变化
```

工作流程：

- 在主线程中创建 Zookeeper 客户端，这时就会创建**两个线程**，一个负责网络连接通信 (connect)，一个负责监听 (listener)
- 通过 connect 线程将注册的监听事件发送给 Zookeeper
- 在 Zookeeper 的注册监听器列表中将注册的**监听事件添加到列表中**
- Zookeeper 监听到有数据或路径变化，将消息发送给 listener 线程
- listener 线程内部调用 process() 方法

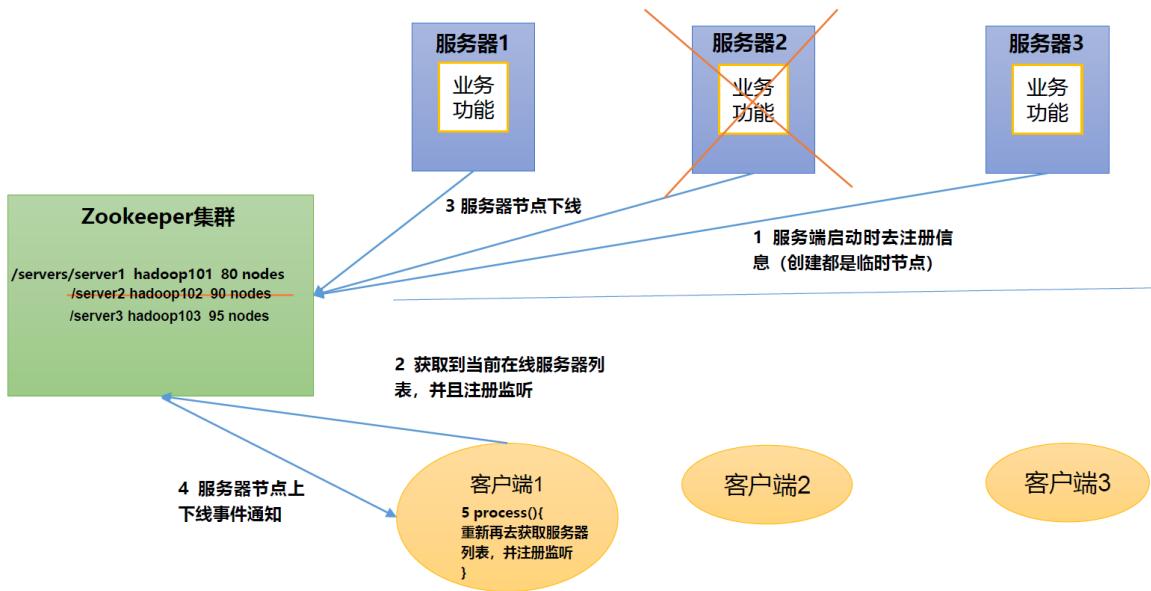
Curator 框架引入了 Cache 来实现对 ZooKeeper 服务端事件的监听，三种 Watcher：

- NodeCache：只是监听某一个特定的节点
- PathChildrenCache：监控一个 ZNode 的子节点
- TreeCache：可以监控整个树上的所有节点，类似于 PathChildrenCache 和 NodeCache 的组合

监听案例

整体架构

客户端实时监听服务器动态上下线



代码实现

客户端：先启动客户端进行监听

```
public class Distributeclient {  
    private String connectString = "192.168.3.128:2181";  
    private int sessionTimeout = 20000;  
    private ZooKeeper zk;  
  
    public static void main(String[] args) throws Exception {  
        Distributeclient client = new Distributeclient();  
  
        // 1 获取zk连接  
        client.getConnect();  
  
        // 2 监听/servers下面子节点的增加和删除  
        client.getServerList();  
  
        // 3 业务逻辑  
        client.business();  
    }  
}
```

```

private void business() throws InterruptedException {
    Thread.sleep(Long.MAX_VALUE);
}

private void getServerList() throws KeeperException, InterruptedException {
    ArrayList<String> servers = new ArrayList<>();
    // 获取所有子节点, true 代表触发监听操作
    List<String> children = zk.getChildren("/servers", true);

    for (String child : children) {
        // 获取子节点的数据
        byte[] data = zk.getData("/servers/" + child, false, null);
        servers.add(new String(data));
    }
    System.out.println(servers);
}

private void getConnect() throws IOException {
    zk = new ZooKeeper(connectString, sessionTimeout, new Watcher() {
        @Override
        public void process(WatchedEvent event) {
            getServerList();
        }
    });
}
}

```

服务端：启动时需要 Program arguments

```

public class DistributeServer {
    private String connectString = "192.168.3.128:2181";
    private int sessionTimeout = 20000;
    private ZooKeeper zk;

    public static void main(String[] args) throws Exception {
        DistributeServer server = new DistributeServer();

        // 1 获取 zookeeper 连接
        server.getConnect();

        // 2 注册服务器到 zk 集群, 注意参数
        server.register(args[0]);

        // 3 启动业务逻辑
        server.business();
    }

    private void business() throws InterruptedException {
        Thread.sleep(Long.MAX_VALUE);
    }

    private void register(String hostname) throws KeeperException,
    InterruptedException {
        // OPEN_ACL_UNSAFE: ACL 开放
        // EPHEMERAL_SEQUENTIAL: 临时顺序节点
    }
}

```

```

        String create = zk.create("/servers/" + hostname, hostname.getBytes(),
                                  ZooDefs.Ids.OPEN_ACL_UNSAFE,
                                  CreateMode.EPHEMERAL_SEQUENTIAL);
        System.out.println(hostname + " is online");
    }

    private void getConnect() throws IOException {
        zk = new ZooKeeper(connectString, sessionTimeout, new Watcher() {
            @Override
            public void process(WatchedEvent event) {
            }
        });
    }
}

```

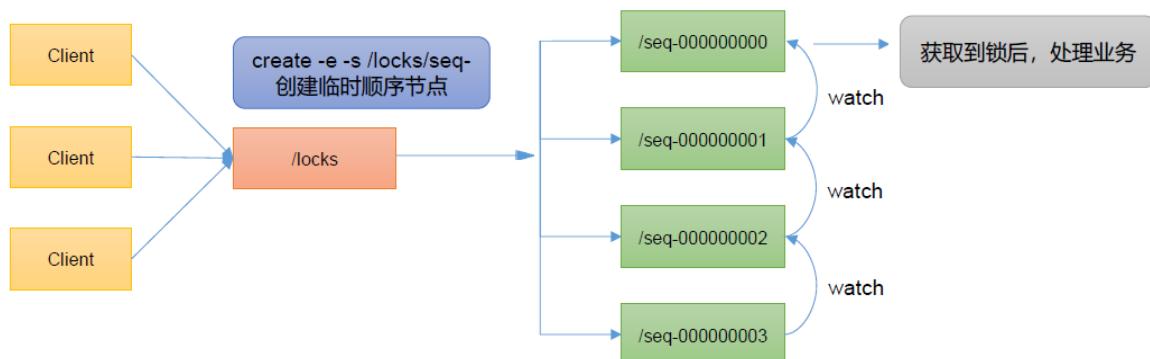
分布式锁

实现原理

分布式锁可以实现在分布式系统中多个进程有序的访问该临界资源，多个进程之间不会相互干扰

核心思想：当客户端要获取锁，则创建节点，使用完锁，则删除该节点

1. 客户端获取锁时，在 /locks 节点下创建**临时顺序节点**
 - 使用临时节点是为了防止当服务器或客户端宕机以后节点无法删除（持久节点），导致锁无法释放
 - 使用顺序节点是为了系统自动编号排序，找最小的节点，防止客户端饥饿现象，保证公平
2. 获取 /locks 目录的所有子节点，判断自己的**子节点序号是否最小**，成立则客户端获取到锁，使用完锁后将该节点删除
3. 反之客户端需要找到比自己小的节点，**对其注册事件监听器，监听删除事件**
4. 客户端的 Watcher 收到删除事件通知，就会重新判断当前节点是否是子节点中序号最小，如果是则获取到了锁，如果不是则重复以上步骤继续获取到比自己小的一个节点并注册监听



Curator

Curator 实现分布式锁 API，在 Curator 中有五种锁方案：

- InterProcessSemaphoreMutex：分布式排它锁（非可重入锁）
- InterProcessMutex：分布式可重入排它锁
- InterProcessReadWriteLock：分布式读写锁
- InterProcessMultiLock：将多个锁作为单个实体管理的容器
- InterProcessSemaphoreV2：共享信号量

```
public class CuratorLock {  
  
    public static CuratorFramework getCuratorFramework() {  
        // 重试策略对象  
        ExponentialBackoffRetry policy = new ExponentialBackoffRetry(3000, 3);  
        // 构建客户端  
        CuratorFramework client = CuratorFrameworkFactory.builder()  
            .connectString("192.168.3.128:2181")  
            .connectionTimeoutMs(2000) // 连接超时时间  
            .sessionTimeoutMs(20000) // 会话超时时间 单位ms  
            .retryPolicy(policy) // 重试策略  
            .build();  
  
        // 启动客户端  
        client.start();  
        System.out.println("zookeeper 启动成功");  
        return client;  
    }  
  
    public static void main(String[] args) {  
        // 创建分布式锁1  
        InterProcessMutex lock1 = new InterProcessMutex(getCuratorFramework(),  
            "/locks");  
  
        // 创建分布式锁2  
        InterProcessMutex lock2 = new InterProcessMutex(getCuratorFramework(),  
            "/locks");  
  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                lock1.acquire();  
                System.out.println("线程1 获取到锁");  
  
                Thread.sleep(5 * 1000);  
  
                lock1.release();  
                System.out.println("线程1 释放锁");  
            }  
        }).start();  
  
        new Thread(new Runnable() {  
            @Override
```

```
public void run() {
    lock2.acquire();
    System.out.println("线程2 获取到锁");

    Thread.sleep(5 * 1000);

    lock2.release();
    System.out.println("线程2 释放锁");

}
}.start();
}

}
```

```
<dependency>
<groupId>org.apache.curator</groupId>
<artifactId>curator-framework</artifactId>
<version>4.3.0</version>
</dependency>
<dependency>
<groupId>org.apache.curator</groupId>
<artifactId>curator-recipes</artifactId>
<version>4.3.0</version>
</dependency>
<dependency>
<groupId>org.apache.curator</groupId>
<artifactId>curator-client</artifactId>
<version>4.3.0</version>
```

源码解析

服务端

服务端程序的入口 QuorumPeerMain

```
public static void main(String[] args) {
    QuorumPeerMain main = new QuorumPeerMain();
    main.initializeAndRun(args);
}
```

initializeAndRun 的工作：

- 解析启动参数
- 提交周期任务，定时删除过期的快照
- 初始化通信模型，默认是 NIO 通信

```

// QuorumPeerMain#runFromConfig
public void runFromConfig(QuorumPeerConfig config) {
    // 通信信组件初始化， 默认是 NIO 通信
    ServerCnxnFactory cnxnFactory = ServerCnxnFactory.createFactory();
    // 初始化NIO 服务端socket， 绑定2181 端口， 可以接收客户端请求
    cnxnFactory.configure(config.getClientPortAddress(),
    config.getMaxClientCnxns(), false);
    // 启动 zk
    quorumPeer.start();
}

```

- 启动 zookeeper

```

// QuorumPeer#start
public synchronized void start() {
    if (!getView().containsKey(myid)) {
        throw new RuntimeException("My id " + myid + " not in the peer
list");
    }
    // 冷启动数据恢复， 将快照中数据恢复到 DataTree
    loadDataBase();
    // 启动通信工厂实例对象
    startServerCnxnFactory();
    try {
        adminServer.start();
    } catch (AdminServerException e) {
        LOG.warn("Problem starting AdminServer", e);
        System.out.println(e);
    }
    // 准备选举环境
    startLeaderElection();
    // 执行选举
    super.start();
}

```

选举机制

环境准备

QuorumPeer#startLeaderElection 初始化选举环境：

```

synchronized public void startLeaderElection() {
    try {
        // Looking 状态， 需要选举
        if (getPeerState() == ServerState.LOOKING) {
            // 选票组件：myid (serverid), zxid, epoch
            // 开始选票时，serverid 是自己，【先投自己】
    }
}

```

```

        currentVote = new Vote(myid, getLastLoggedZxid(),
getCurrentEpoch());
    }
}

if (electionType == 0) {
    try {
        udpSocket = new DatagramSocket(getQuorumAddress().getPort());
        // 响应投票结果线程
        responder = new ResponderThread();
        responder.start();
    } catch (SocketException e) {
        throw new RuntimeException(e);
    }
}

// 创建选举算法实例
this.electionAlg = createElectionAlgorithm(electionType);
}

```

```

// zk总的发送和接收队列准备
protected Election createElectionAlgorithm(int electionAlgorithm){
    // 负责选举过程中的所有网络通信，创建各种队列和集合
    QuorumCnxManager qcm = createCnxnManager();
    QuorumCnxManager.Listener listener = qcm.listener;
    if(listener != null){
        // 启动监听线程，调用 client = ss.accept()阻塞，等待处理请求
        listener.start();
        // 准备好发送和接收队列准备
        FastLeaderElection fle = new FastLeaderElection(this, qcm);
        // 启动选举线程，【workerSender 和 workerReceiver】
        fle.start();
        le = fle;
    }
}

```

选举源码

当 Zookeeper 启动后，首先都是 Looking 状态，通过选举让其中一台服务器成为 Leader

执行 `super.start()` 相当于执行 `QuorumPeer#run()` 方法

```

public void run() {
    case LOOKING:
        // 进行选举，选举结束返回最终成为 Leader 胜选的那张选票
        setCurrentVote(makeLEStrategy().lookForLeader());
}

```

FastLeaderElection 类：

- `lookForLeader`: 选举

```

public Vote lookForLeader() {
    // 正常启动中其他服务器都会向我发送一个投票，保存每个服务器的最新合法有效的投票
    HashMap<Long, Vote> recvset = new HashMap<Long, Vote>();
    // 存储合法选举之外的投票结果
    HashMap<Long, Vote> outofelection = new HashMap<Long, Vote>();
    // 一次选举的最大等待时间，默认值是0.2s
    int notTimeout = finalizewait;
    // 每发起一轮选举，logicalclock++，在没有合法的epoch 数据之前，都使用逻辑时钟代替
    synchronized(this){
        // 更新逻辑时钟，每进行一次选举，都需要更新逻辑时钟
        logicalclock.incrementAndGet();
        // 更新选票(serverid, zxid, epoch)
        updateProposal(getInitId(), getInitLastLoggedzxid(),
        getPeerEpoch());
    }
    // 广播选票，把自己的选票发给其他服务器
    sendNotifications();
    // 一轮一轮的选举直到选举成功
    while ((self.getPeerState() == ServerState.LOOKING) && (!stop)){ }
}

```

- sendNotifications: 广播选票

```

private void sendNotifications() {
    // 遍历投票参与者，给每台服务器发送选票
    for (long sid : self.getCurrentAndNextConfigVoters()) {
        // 创建发送选票
        ToSend notmsg = new ToSend(...);
        // 把发送选票放入发送队列
        sendqueue.offer(notmsg);
    }
}

```

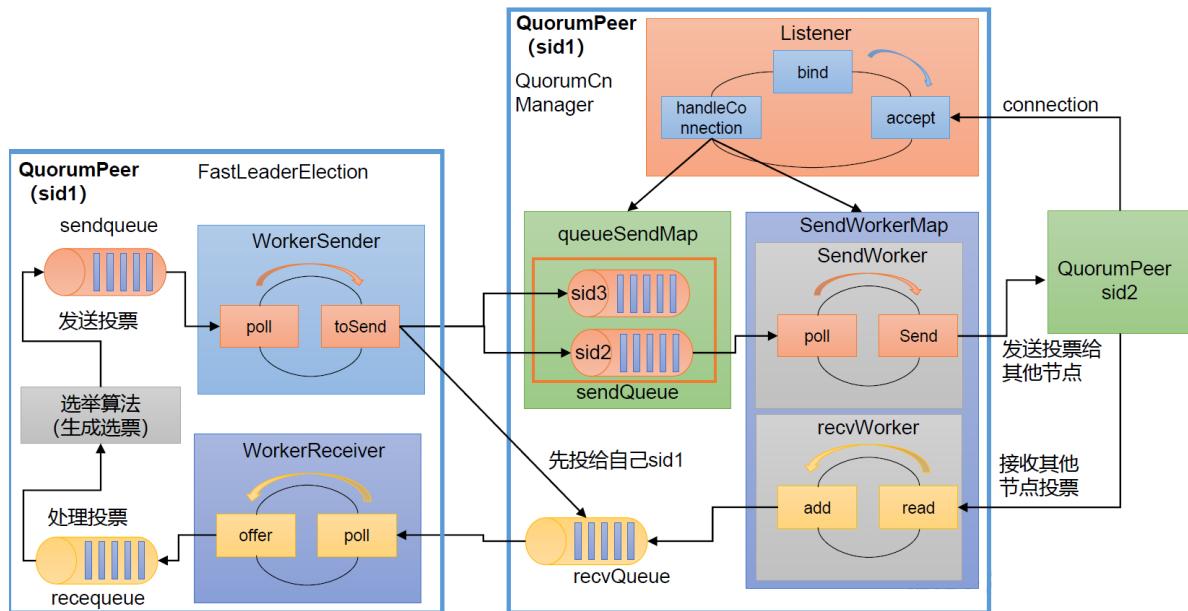
FastLeaderElection 中有 WorkerSender 线程：

- `ToSend m = sendqueue.poll(3000, TimeUnit.MILLISECONDS)`：阻塞获取要发送的选票
- `process(m)`：处理要发送的选票
 - `manager.toSend(m.sid, requestBuffer)`：发送选票
 - `if (this.mysid == sid)`：如果消息的接收者 `sid` 是自己，直接进入自己的 `RecvQueue`（自己投自己）
 - `else`：如果接收者是其他服务器，创建对应的发送队列或者复用已经存在的发送队列，把消息放入该队列
 - `connectOne(sid)`：建立连接
 - `sock.connect(electionAddr, cnxTO)`：建立与 `sid` 服务器的连接
 - `initiateConnection(sock, sid)`：初始化连接
 - `startConnection(sock, sid)`：创建并启动发送器线程和接收器线程
 - `dout = new DataOutputStream(buf)`：获取 `Socket` 输出流，向服务器发送数据

- `din = new DataInputStream(new BIS(sock.getInputStream()))`: 通过输入流读取对方发送过来的选票
- `if (sid > self.getId())`: 接收者 sid 比我的大，没有资格给对方发送连接请求的，直接关闭自己的客户端
- `Sendworker sw`: 初始化发送器，并启动发送器线程，线程 run 方法
 - `while (running && !shutdown && sock != null)`: 连接没有断开就一直运行
 - `ByteBuffer b = pollSendQueue()`: 从发送队列 SendQueue 中获取发送消息
 - `lastMessageSent.put(sid, b)`: 更新对于 sid 这台服务器的最近一条消息
 - `send(b)`: 执行发送
- `Recvworker rw`: 初始化接收器，并启动接收器线程
 - `din.readFully(msgArray, 0, length)`: 输入流接收消息
 - `addToRecvQueue(new Message(messagg, sid))`: 将消息放入接收消息 recvQueue 队列

FastLeaderElection 中有 WorkerReceiver 线程

- `response = manager.pollRecvQueue()`: 从 RecvQueue 中阻塞获取出选举投票消息 (其他服务器发送过来的)

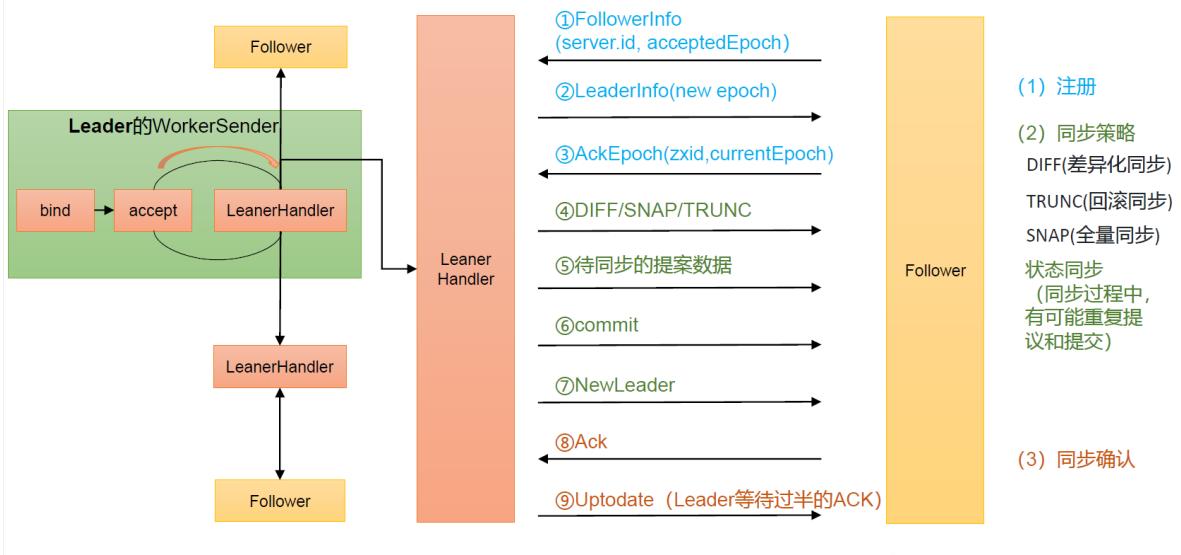


状态同步

选举结束后，每个节点都需要根据角色更新自己的状态，Leader 更新状态为 Leader，其他节点更新状态为 Follower，整体流程：

- Follower 需要让 Leader 知道自己的状态 (sid, epoch, zxid)
- Leader 收到信息，根据信息构建新的 epoch，要返回对应的信息给 Follower，Follower 更新自己的 epoch

- Leader 需要根据 Follower 的状态，确定何种方式的数据同步 DIFF、TRUNC、SNAP，就是要以 Leader 服务器数据为准
 - DIFF: Leader 提交的 zxid 比 Follower 的 zxid 大，发送 Proposal 给 Follower 提交执行
 - TRUNC: Follower 的 zxid 比 leader 的 zxid 大，Follower 要进行回滚
 - SNAP: Follower 没有任何数据，直接全量同步
- 执行数据同步，当 Leader 接收到超过半数 Follower 的 Ack 之后，进入正常工作状态，集群启动完成

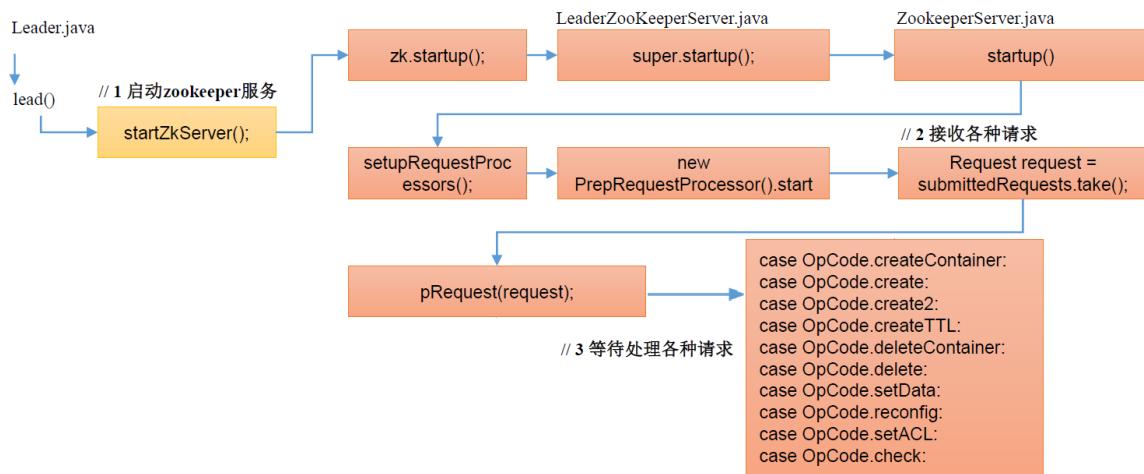


核心函数解析：

- Leader 更新状态入口: `Leader.lead()`
 - `zk.loadData()` : 恢复数据到内存
 - `cnxAcceptor = new LearnerCnxAcceptor()` : 启动通信组件
 - `s = ss.accept()` : 等待其他 Follower 节点向 Leader 节点发送同步状态
 - `LearnerHandler fh` : 接收到 Follower 的请求，就创建 LearnerHandler 对象
 - `fh.start()` : 启动线程，通过 switch-case 语法判断接收的命令，执行相应的操作
- Follower 更新状态入口: `Follower.followerLeader()`
 - `QuorumServer leaderServer = findLeader()` : 查找 Leader
 - `connectToLeader(addr, hostname)` : 与 Leader 建立连接
 - `long newEpochZxid = registerWithLeader(Leader.FOLLOWERINFO)` : 向 Leader 注册

主从工作

Leader: 主服务的工作流程



Follower: 从服务的工作流程，核心函数为 `Follower#followLeader()`

- `readPacket(qp)`: 读取信息
- `processPacket(qp)`: 处理信息

```

protected void processPacket(QuorumPacket qp) throws Exception{
    switch (qp.getType()) {
        case Leader.PING:
            break;
        case Leader.PROPOSAL:
            break;
        case Leader.COMMIT:
            break;
        case Leader.COMMITANDACTIVATE:
            break;
        case Leader.UPTODATE:
            break;
        case Leader.REVALIDATE:
            break;
        case Leader.SYNC:
            break;
        default:
            break;
    }
}

```

客户端

