

# Spring源码

---

## 模块组成及介绍

### 核心容器

由spring-beans、spring-core、spring-context和spring-expression (Spring Expression Language, SpEL) 4个模块组成。

spring-core 和 spring-beans 模块是 Spring 框架的核心模块，包含了控制反转 (Inversion of Control, IOC) 和依赖注入 (Dependency Injection, DI)。 BeanFactory 接口是 Spring 框架中的核心接口，它是工厂模式的具体实现。 BeanFactory 使用控制反转对应用程序的配置和依赖性规范与实际的应用程序代码进行了分离。但 BeanFactory 容器实例化后并不会自动实例化 Bean，**只有当 Bean 被使用时 BeanFactory 容器才会对该 Bean 进行实例化与依赖关系的装配。** spring-context 模块构架于核心模块之上，他扩展了 BeanFactory，为她添加了 Bean 生命周期控制、框架事件体系以及资源加载透明化等功能。此外该模块还提供了许多企业级支持，如邮件访问、远程访问、任务调度等， ApplicationContext 是该模块的核心接口，她的超类是 BeanFactory。与 BeanFactory 不同， ApplicationContext 容器实例化后会自动对所有的单实例 Bean 进行实例化与依赖关系的装配，使之处于待用状态。 spring-context-support 模块是对 Spring IOC 容器的扩展支持，以及 IOC 子容器。 spring-context-indexer 模块是 Spring 的类管理组件和 Classpath 扫描。 spring-expression 模块是统一表达式语言 (EL) 的扩展模块，可以查询、管理运行中的对象，同时也方便的可以调用对象方法、操作数组、集合等。它的语法类似于传统 EL，但提供了额外的功能，最出色的要数函数调用和简单字符串的模板函数。这种语言的特性是基于 Spring 产品的需求而设计，他可以非常方便地同 Spring IOC 进行交互。

### AOP 和设备支持

由 spring-aop、spring-aspects 和 spring-instrument 3 个模块组成。

spring-aop 是 Spring 的另一个核心模块，是 AOP 主要的实现模块。作为继 OOP 后，对程序员影响最大的编程思想之一，AOP 极大地开拓了人们对于编程的思路。在 Spring 中，他是以 JVM 的动态代理技术为基础，然后设计出了一系列的 AOP 横切实现，比如前置通知、返回通知、异常通知等，同时，

Pointcut 接口来匹配切入点，可以使用现有的切入点来设计横切面，也可以扩展相关方法根据需求进行切入。

spring-aspects 模块集成自 AspectJ 框架，主要是为 Spring AOP 提供多种 AOP 实现方法。spring-instrument 模块是基于 JAVA SE 中的"java.lang.instrument"进行设计的，应该算是 AOP 的一个支援模块，主要作用是在 JVM 启用时，生成一个代理类，程序员通过代理类在运行时修改类的字节，从而改变一个类的功能，实现 AOP 的功能。在分类里，我把他分在了 AOP 模块下，在 Spring 官方文档里对这个地方也有点含糊不清，这里是纯个人观点。

## 数据访问与集成

由 spring-jdbc、spring-tx、spring-orm、spring-jms 和 spring-oxm 5 个模块组成。

spring-jdbc 模块是 Spring 提供的 JDBC 抽象框架的主要实现模块，用于简化 Spring JDBC 操作。

主要是提供 JDBC 模板方式、关系数据库对象化方式、SimpleJdbc 方式、事务管理来简化 JDBC 编程，

主要实现类是 JdbcTemplate、SimpleJdbcTemplate 以及 NamedParameterJdbcTemplate。

spring-tx 模块是 Spring JDBC 事务控制实现模块。使用 Spring 框架，它对事务做了很好的封装，通过它的 AOP 配置，可以灵活的配置在任何一层；但是在很多的需求和应用，直接使用 JDBC 事务控制还是有其优势的。其实，事务是以业务逻辑为基础的；一个完整的业务应该对应业务层里的一个方法；如果业务操作失败，则整个事务回滚；所以，事务控制是绝对应该放在业务层的；但是，持久层的设计则应该遵循一个很重要的原则：保证操作的原子性，即持久层里的每个方法都应该是不可以分割的。所以，在使用 Spring JDBC 事务控制时，应该注意其特殊性。

spring-orm 模块是 **ORM 框架支持模块**，主要集成 Hibernate, Java Persistence API (JPA) 和 Java Data Objects (JDO) 用于资源管理、数据访问对象(DAO)的实现和事务策略。

spring-oxm 模块主要提供一个抽象层以支撑 OXM (OXM 是 Object-to-XML-Mapping 的缩写，它是一个 O/M-mapper，将 **java 对象映射成 XML 数据**，或者将 XML 数据映射成 java 对象)，例如：JAXB, Castor, XMLBeans, JiBX 和 XStream 等。

spring-jms 模块 (Java Messaging Service) 能够发送和接收信息，自 Spring Framework 4.1 以后，他还提供了对 spring-messaging 模块的支撑。

## Web 组件

由 spring-web、spring-webmvc、spring-websocket 和 spring-webflux 4 个模块组成。

spring-web 模块为 Spring 提供了最基础 Web 支持，主要建立于核心容器之上，**通过 Servlet 或者 Listeners 来初始化 IOC 容器**，也包含一些与 Web 相关的支持。

**spring-webmvc** 模块众所周知是一个的 Web-Servlet 模块，实现了 Spring MVC (model-view-Controller) 的 Web 应用。

**spring-websocket** 模块主要是与 Web 前端的**全双工通讯的协议**。

spring-webflux 是一个新的**非堵塞函数式 Reactive Web 框架**，可以用来建立异步的，非阻塞，事件驱动的服务，并且扩展性非常好。

## 通信报文

即 spring-messaging 模块，是从 Spring4 开始新加入的一个模块，主要职责是为 Spring 框架集

成一些基础的报文传送应用。

## 集成测试

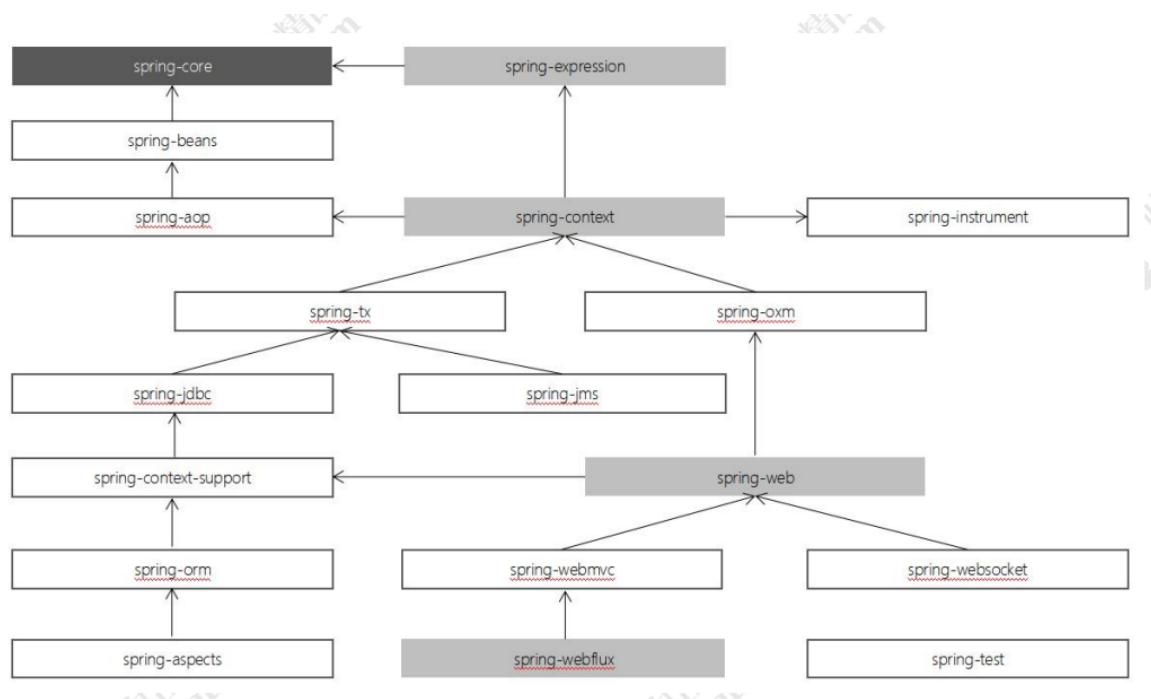
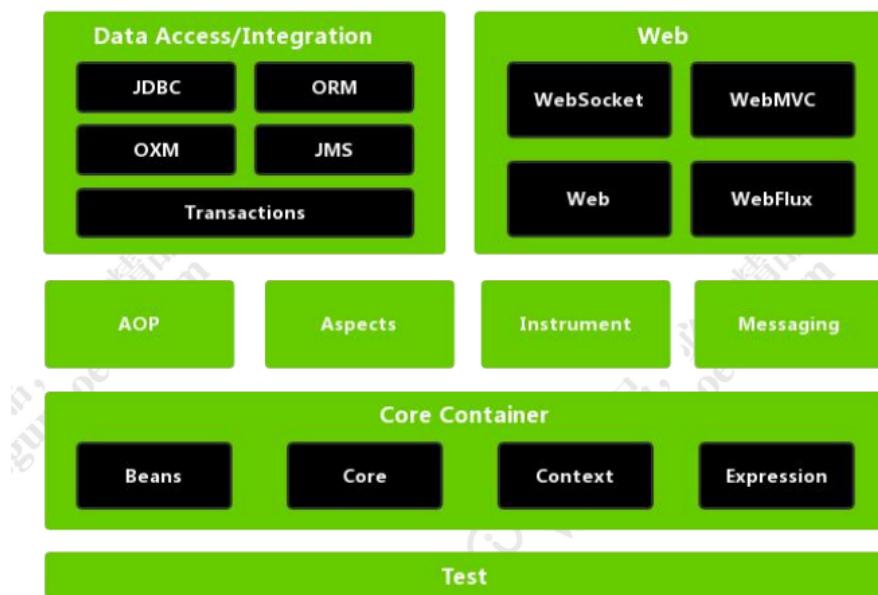
即 spring-test 模块，主要为测试提供支持的，毕竟在不需要发布（程序）到你的应用服务器或者连接

到其他企业设施的情况下能够执行一些集成测试或者其他测试对于任何企业都是非常重要的。

## 集成兼容

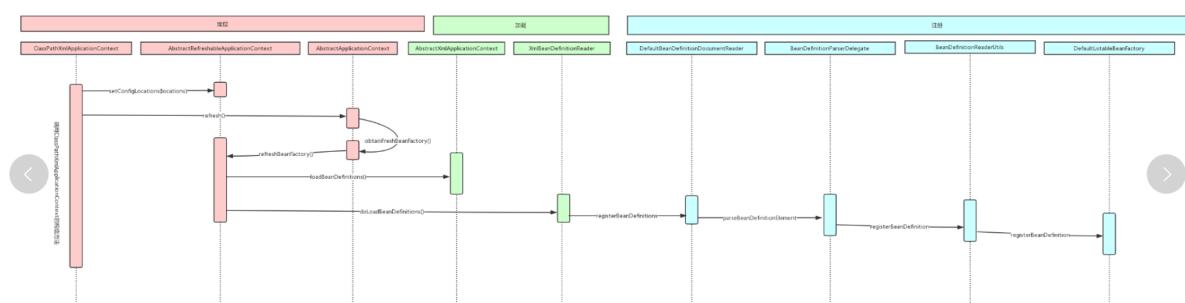
即 spring-framework-bom 模块，Bill of Materials.解决 Spring 的不同模块依赖版本不同问题

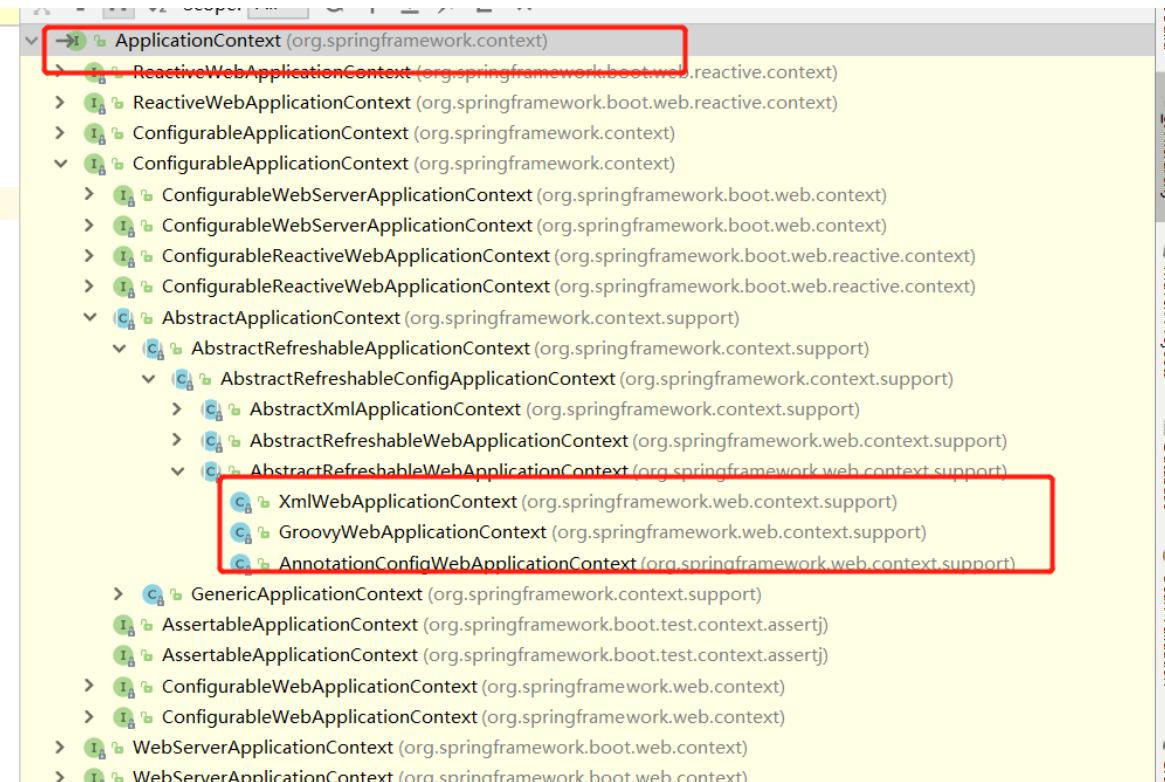
## Spring Framework 5 Runtime



## IOC时序图

定位-->加载-->注册





beanfactory是application上层接口

1、初始化的入口在容器实现中的 refresh() 调用来完成。

2、对 Bean 定义载入 IOC 容器使用的方法是 loadBeanDefinition(), 其中的大致过程如下：通过 **ResourceLoader** 来完成资源文件位置的定位，DefaultResourceLoader 是默认的实现，同时上下文本身就给出了 ResourceLoader 的实现，可以从类路径，文件系统,URL 等方式来定为资源位置。如果是 XmlBeanFactory 作为 IOC 容器，那么需要为它指定 Bean 定义的资源，也就是说 Bean 定义文件时通过抽象成 Resource 来被 IOC 容器处理的，容器通过 BeanDefinitionReader 来完成定义信息的解析和 Bean 信息的注册，往往使用的是 XmlBeanDefinitionReader 来解析 Bean 的 XML 定义文件 - 实际的处理过程是委托给 BeanDefinitionParserDelegate 来完成的，从而得到 bean 的定义信息，这些信息在 Spring 中使用 BeanDefinition 对象来表示-这个名字可以让我们想到 loadBeanDefinition(), registerBeanDefinition() 这些相关方法。它们都是为处理 BeanDefinitin 服务的，**容器解析得到 BeanDefinition 以后，需要把它在 IOC 容器中注册**，这由 IOC 实现 BeanDefinitionRegistry 接口来实现。注册过程就是在 IOC 容器内部维护的一个 HashMap 来保存得到的 BeanDefinition 的过程。这个 **HashMap 是 IOC 容器持有 Bean 信息的场所**，以后对 Bean 的操作都是围绕这个 HashMap 来实现的。然后我们就可以通过 BeanFactory 和 ApplicationContext 来享受到 Spring IOC 的服务了，在使用 IOC 容器的时候，我们注意到除了少量粘合代码，绝大多数以正确 IOC 风格编写的应用程序代码

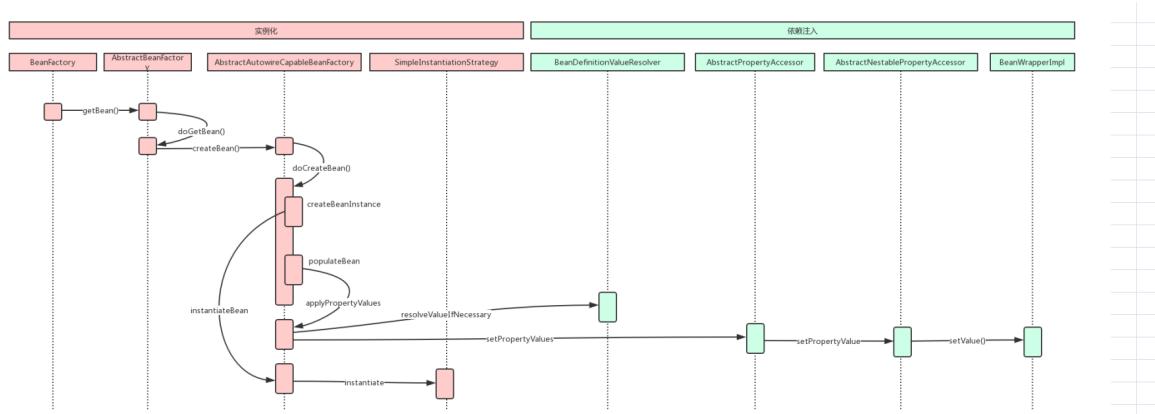
完全不用关心如何到达工厂，因为容器将把这些对象与容器管理的其他对象钩在一起。基本的策略是把工厂放到已知的地方，最好是放在对预期使用的上下文有意义的地方，以及代码将实际需要访问工厂的地方。Spring 本身提供了对声明式载入 web 应用程序用法的应用程序上下文，并将其存储在 ServletContext 中的框架实现

@compen@service@controller, 完成注册, 默认使用类名首字母小写

@AutoWrite 根据类型注入，和 @Qualifie("userService") 结合起来可以根据名字和类型注入

@Resource 默认根据名字注入，其次按照类型搜索

## DI



### 依赖注入发生的时间

- 1)、用户第一次调用 getBean() 方法时，IOC 容器触发依赖注入。
- 2)、当用户在配置文件中将元素配置了 lazy-init=false 属性，即让容器在解析注册 Bean 定义时进行预实例化，触发依赖注入

省资源用到了实例化, 取消就在注册时实例化

定位资源-->加载资源-->注册IOC(存入hashmap)-->实例化-->依赖注入

## AOP

**定义:** 以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术

**运用:** 做的一些非业务，如：日志、事务、安全等都会写在业务代码中(也也就是说，这些**非业务类横切于业务类**)，但这些代码往往是重复，复制——粘贴式的代码会给程序的维护带来不便，AOP 就实现了把这些业务需求与系统需求分开来做

具体使用:

建立一个切面类.@aspect,加入bean容器

建立pointCut方法,切入点分注注解或者方法

目标对象,设定注解或对应方法所在位置

通知:前,后,环绕通知等

## AOP 代理 (AOP Proxy)

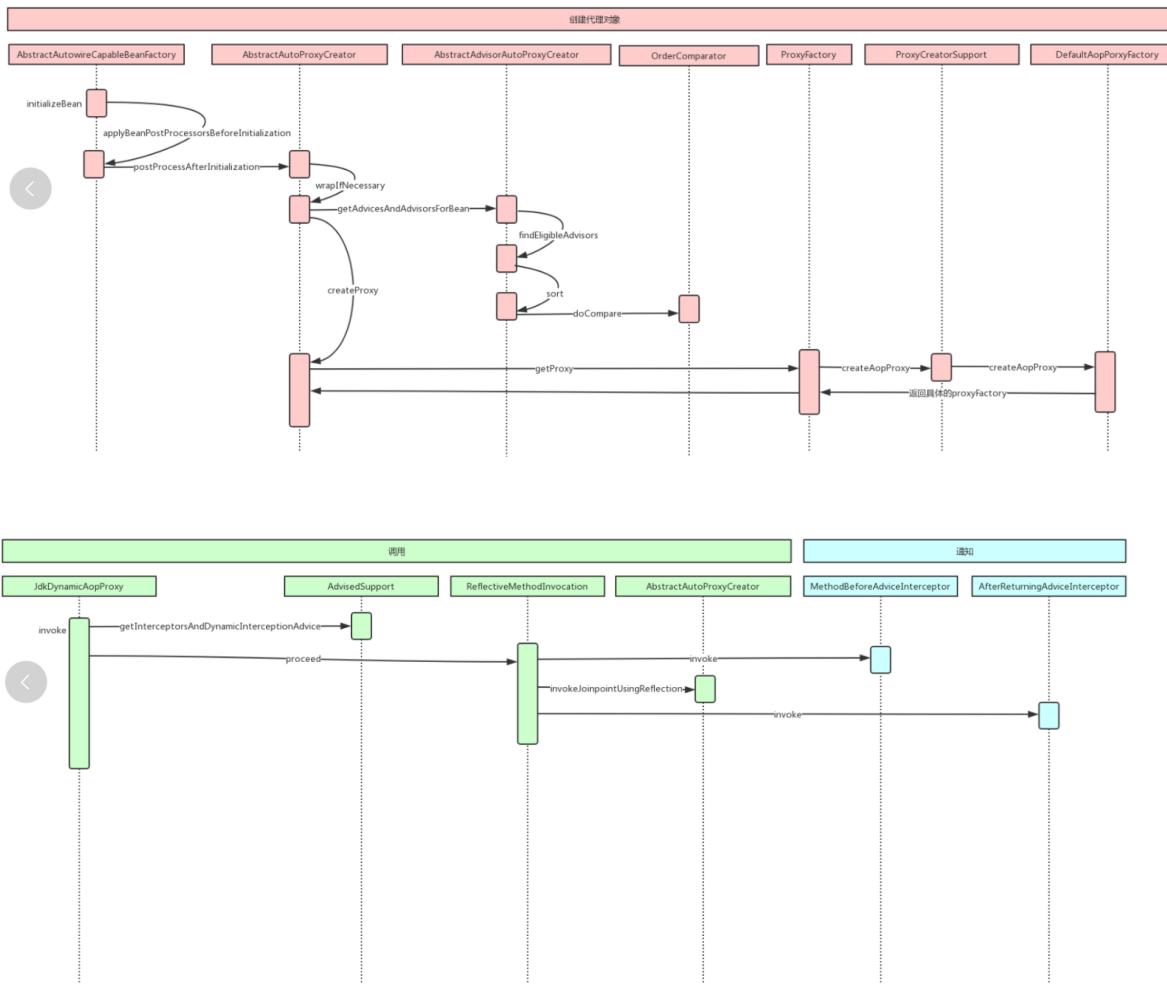
默认情况下, TargetObject 实现了接口时, 则采用 JDK 动态代理, 例如, AServicelmpl; 反之, 采用 CGLib 代理, CGLIB包的底层是通过使用一个小而快的字节码处理框架ASM, 来转换字节码并生成新的类。

例如, BServiceImpl。强制使用 CGLib 代理需要将 [aop:config](#) 的 proxy-target-class 属性设为 true。

```
1 public class LogAspect {  
2     @Pointcut("@annotation(com.zy.testAop.Action)")  
3     public void annotationPointCut() {  
4         System.out.println("annotationPointCut");  
5     }  
6     //基于注解拦截,哪里使用注解哪里拦截,更灵活  
7     @After("annotationPointCut")  
8     public void after(JoinPoint joinPoint) {  
9         MethodSignature signature =  
10            (MethodSignature)joinPoint.getSignature();  
11            Method method = signature.getMethod();  
12            Action action =  
13            method.getAnnotation(Action.class);  
14            System.out.println("注解式拦截" + action.name());  
15        }  
16        //基于方法拦截  
17        @Before("execution(*  
18            com.zy.testAop.DemoMethodService.*(..))")  
19        public void before(JoinPoint joinPoint) {  
20            MethodSignature signature =  
21            (MethodSignature)joinPoint.getSignature();  
22            Method method = signature.getMethod();  
23            System.out.println("方法规则式拦截, " +  
24            method.getName());  
25        }  
26    }
```

## 代码时序图: 创建代理对象-->调用-->通知

BeanPostProcessor 后置处理器,接入IOC容器,自定义操作自己的bean



## MVC

链路,继承httpserverlet,修改操作,加入核心操作dispatchserverlet,九大组件

multipartResolver;文件上传使用,包装成,将普通的Request 包装成

MultipartHttpServletRequest 来实现localeResolver;根据request中的参数,设置国际化

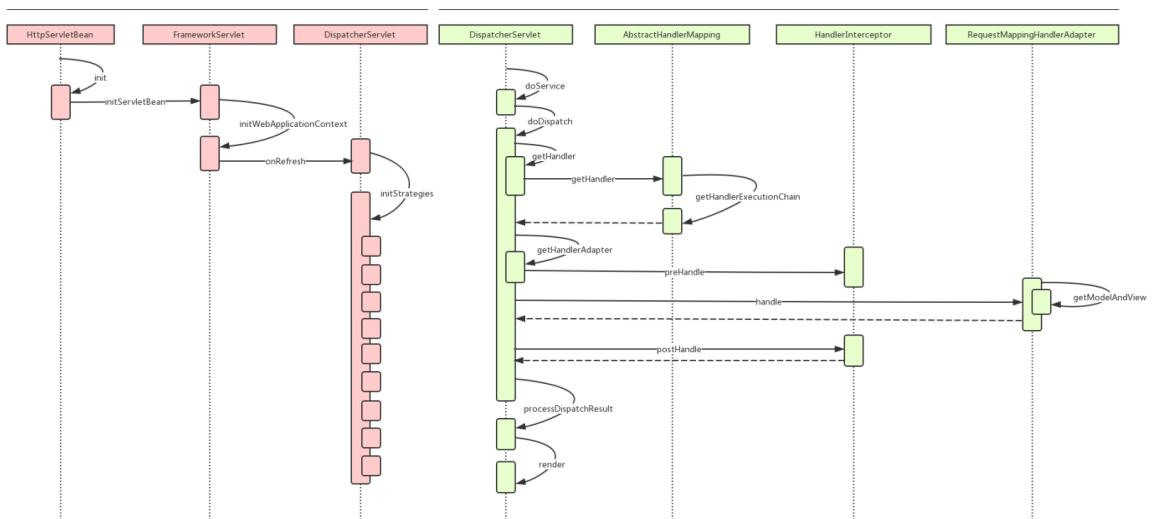
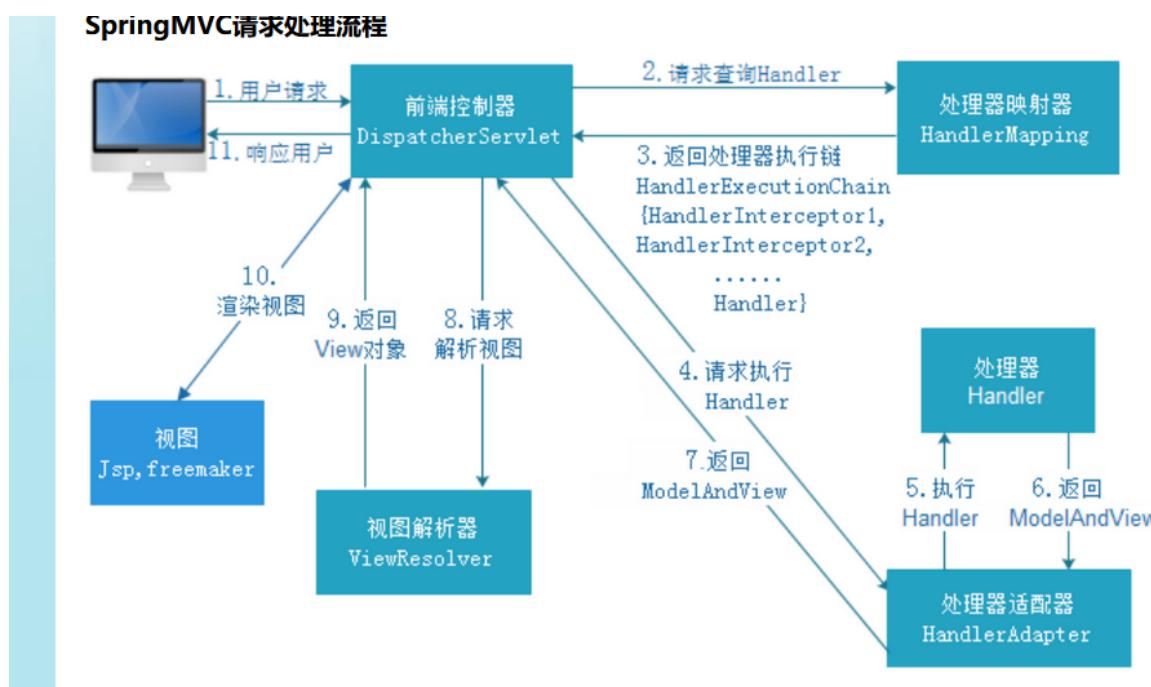
themeResolver;主题选择,加入css样式, Spring MVC 中一套主题对应一个 properties 文件

**handlerMappings;** list集合存放,controller层每一个方法对应一个handle  
**handlerAdapters;** list集合,将请求中,requestServerlet所带实参,取出去条用 handmapping

handlerExceptionResolvers;执行hand出现问题抛出,

RequestToViewNameTranslator;用于从 Request 中获取 viewName(有的 Handler 处理完成之后, 没有设置 View 也没有设置 ViewName)

flashMapManager;重定向时保存部分原来一些数据  
**viewResolvers**;handler反射invoke方法执行完毕,得到字符串之类,这里字符串与相关页面名之间的映射关系,list集合,解析



## Spring MVC 使用优化建议

### 1、Controller 如果能保持单例，尽量使用单例

减少创建对象和回收对象的开销

### 2、处理 Request 的方法中的形参务必加上@RequestParam 注解

避免 Spring MVC 使用asm 框架读取class 文件获取方法参数名的过程，  
 ps:asm 一套java字节码生成框架，它可以直接对class文件进行增删改的操作

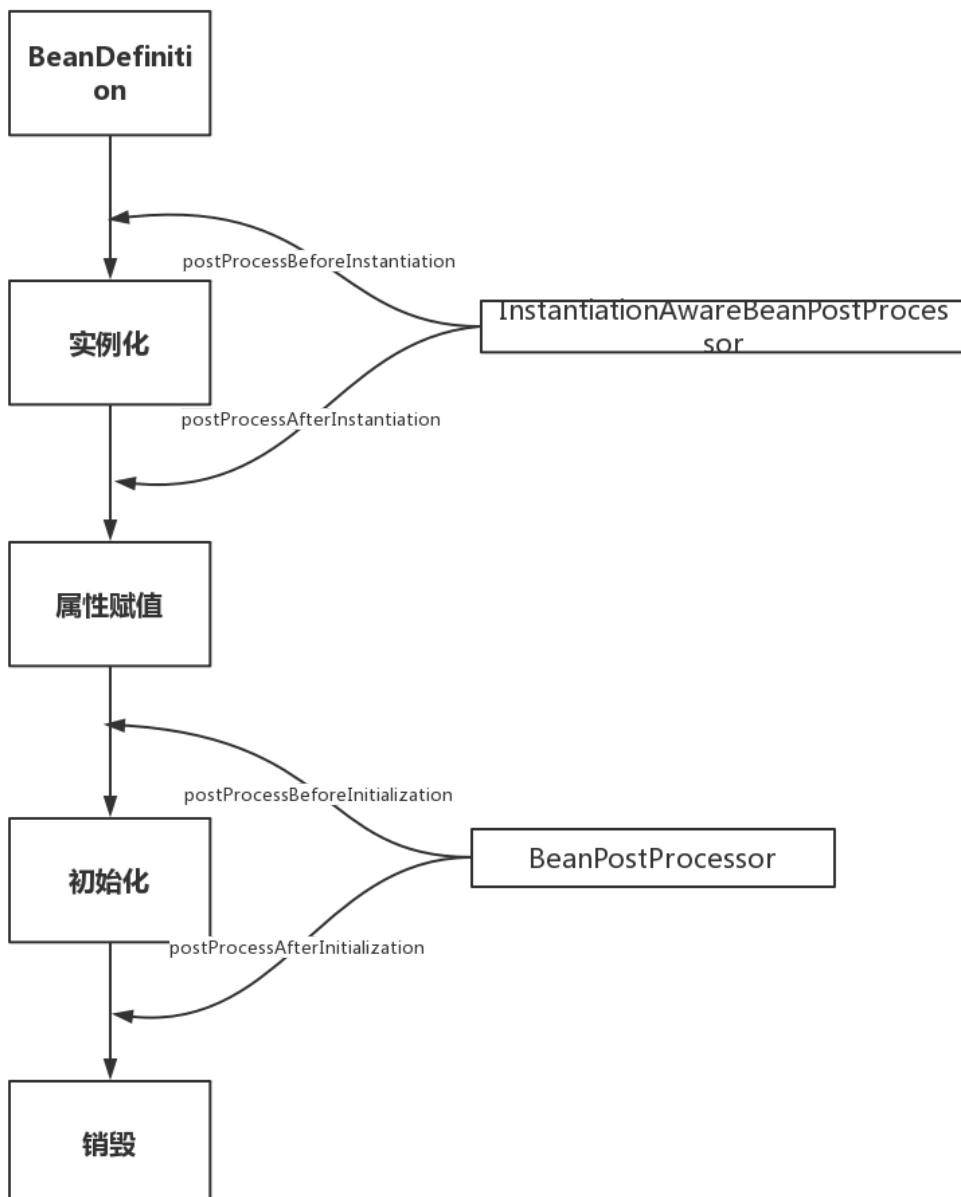
### 3、缓存 URL

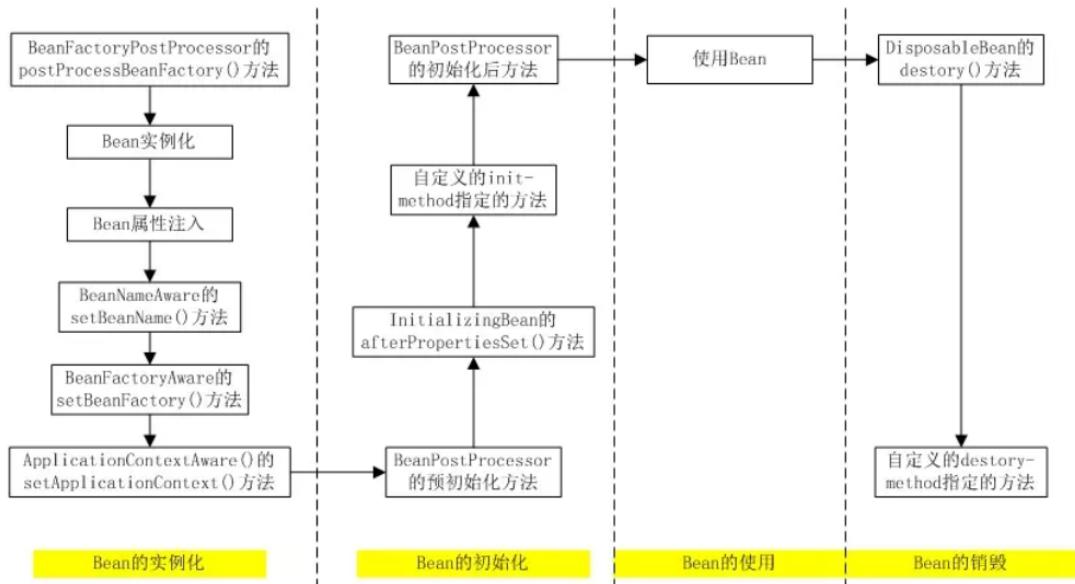
Spring MVC 并没有对处理 url 的方法进行缓存,私有方法,可以增加缓存,重新编译使用

# 生命周期

定义bean-->实例化-->属性赋值-->初始化-->销毁

spring bean是先对bean进行实例化后 再进行初始化





## 循环依赖

构造器注入循环依赖

prototype模式field属性注入循环依赖

singleton模式field属性注入（setter方法注入）循环依赖（自动解决）  
Spring容器的“三级缓存”

singletonObjects: 单例对象cache

earlySingletonObjects 提前曝光单例对象的cache

singletonFactories 单例对象工厂cache

用到第三级缓存

A首先完成了初始化的第一步，并且将自己提前曝光到singletonFactories中，此时进行初始化的第二步，发现自己依赖对象B，此时就尝试去get(B)，发现B还没有被create，所以走create流程，B在初始化第一步的时候发现自己依赖了对象A，于是尝试get(A)，尝试一级缓存singletonObjects(肯定没有，因为A还没初始化完全)，尝试二级缓存earlySingletonObjects (也没有)，尝试三级缓存singletonFactories，由于A通过ObjectFactory将自己提前曝光了，所以B能够通过ObjectFactory.getObject拿到A对象(虽然A还没有初始化完全，但是总比没有好呀)，B拿到A对象后顺利完成了初始化阶段1、2、3，完全初始化之后将自己放入到一级缓存singletonObjects中。此时返回A中，A此时能拿到B的对象顺利完成自己的初始化阶段2、3，最终A也完成了初始化，进去了二级缓存singletonObjects中，而且更加幸运的是，由于B拿到了A的对象引用，所以B现在hold住的A对象完成了初始化。

# spring JDBC

## 无框架时代

- 1 加载驱动 `Class.forName("com.mysql.jdbc.Driver")`
- 2 建立连接 `DriverManager.getConnection`
- 3、创建语句集 `psm=con.prepareStatement(sql);`
- 4、执行语句集 `rs = psm.executeQuery();`
5. 循环读取结果集，(结果集与对象映射)
- 6 关流，关连接

- 1 BeanFactory 是个 bean 工厂，是一个工厂类(接口)，它负责生产和管理 bean 的一个工厂是 ioc 容器最底层的接口，是个 ioc 容器，是 spring 用来管理和装配普通 bean 的 ioc 容器（这些 bean 成为普通 bean）。
- 2 FactoryBean 是个 bean，在 IOC 容器的基础上给 Bean 的实现加上了一个简单工厂模式和装饰模式，是一个可以生产对象和装饰对象的工厂 bean，由 spring 管理后，生产的对象是由 `getobject()` 方法决定的。

<b>Transactional</b>	<b>声明事务 (一般默认配置即可满足要求，当然也可以自定义)</b>
更多	<a href="https://www.cnblogs.com/cocoxu1992/p/10554889.html">https://www.cnblogs.com/cocoxu1992/p/10554889.html</a>

ORM 框架出现解决结果集与对象映射；

如果需要更加灵活的 SQL，可以使用 MyBatis，对于底层的编码，或者性能要求非常高的场合，可以用 JDBC。

实际上在我们的项目中，MyBatis 和 Spring JDBC 是可以混合使用的。

也根据项目的需求自己写 ORM 框架，手写 ORM 框架一样

# Mybatis 源码

MyBatis 的核心特性

- 1、使用连接池对连接进行管理

2、SQL 和代码分离，集中管理

3、结果集映射

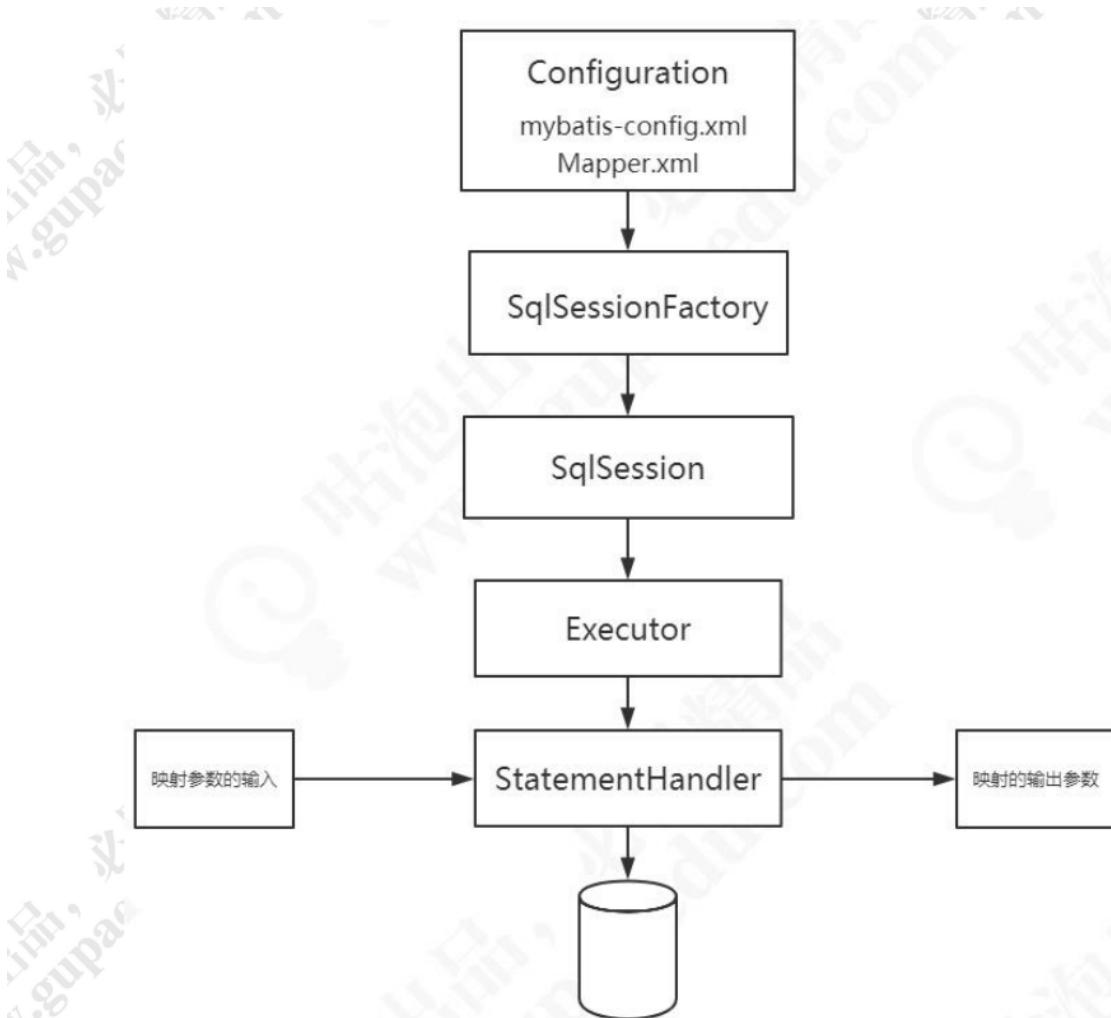
4、参数映射和动态 SQL

5、重复 SQL 的提取

6、缓存管理

7、插件机制,<https://mybatis.plus/guide>

## 工作流程图



```
1 public void testMapper() throws IOException {
2     String resource = "mybatis-config.xml";
3     InputStream inputStream =
4         Resources.getResourceAsStream(resource);
5     SqlSessionFactory sqlSessionFactory = new
6         SqlSessionFactoryBuilder().build(inputStream);
7     SqlSession session = sqlSessionFactory.openSession();
```

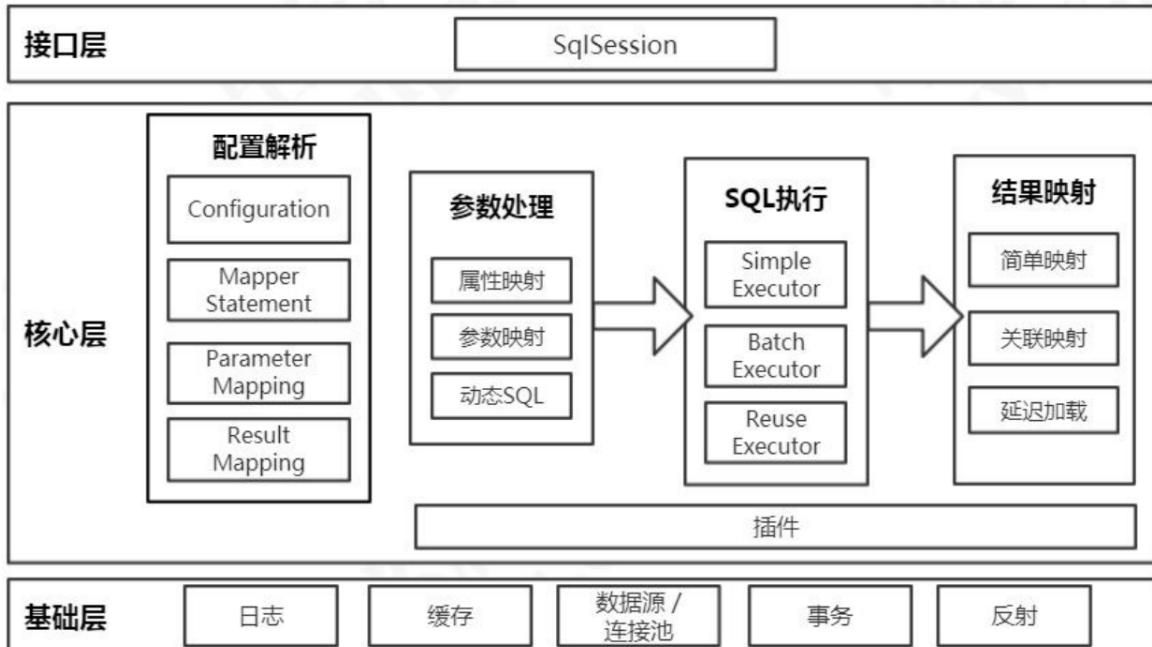
```
6   try {
7       BlogMapper mapper =
8           session.getMapper(BlogMapper.class);
9       Blog blog = mapper.selectBlogById(1);
10      System.out.println(blog);
11  } finally {
12      session.close();
13  }
14
15
16 //mybatis-config.xml
17
18 <?xml version="1.0" encoding="UTF-8" ?>
19 <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD
Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-
config.dtd">
20 <configuration>
21
22     <properties resource="db.properties"></properties>
23     <settings>
24         <!-- 打印查询语句 -->
25         <setting name="logImpl" value="STDOUT_LOGGING" />
26         <!-- 控制全局缓存（二级缓存） -->
27         <setting name="cacheEnabled" value="true"/>
28         <!-- 延迟加载的全局开关。当开启时，所有关联对象都会延迟
加载。默认 false -->
29         <setting name="lazyLoadingEnabled" value="true"/>
30         <!-- 当开启时，任何方法的调用都会加载该对象的所有属性。
默认 false，可通过select标签的 fetchType来覆盖-->
31         <setting name="aggressiveLazyLoading"
value="false"/>
32             <!-- Mybatis 创建具有延迟加载能力的对象所用到的代理工
具， 默认JAVASSIST -->
33             <!--<setting name="proxyFactory" value="CGLIB"
/>-->
34             <!-- STATEMENT级别的缓存，使一级缓存，只针对当前执行的
这一statement有效 -->
35             <!--
36                 <setting name="localCacheScope"
value="STATEMENT"/>
37                 -->
38                 <setting name="localCacheScope" value="SESSION"/>
```

```
39     </settings>
40
41     <typeAliases>
42         <typeAlias alias="blog"
43             type="com.wuzz.domain.Blog" />
44     </typeAliases>
45
46     <!--    <typeHandlers>
47         <typeHandler
48             handler="com.wuzz.type.MyTypeHandler"></typeHandler>
49         </typeHandlers>-->
50
51     <!-- 对象工厂 -->
52     <!--    <objectFactory
53         type="com.wuzz.objectfactory.GPOBJECTFACTORY">
54             <property name="wuzz" value="666"/>
55         </objectFactory>-->
56
57     <!--    <plugins>
58         <plugin
59             interceptor="com.wuzz.interceptor.SQLInterceptor">
60                 <property name="wuzz" value="betterme" />
61             </plugin>
62         <plugin
63             interceptor="com.wuzz.interceptor.MyPageInterceptor">
64                 </plugin>
65         </plugins>-->
66
67     <environments default="development">
68         <environment id="development">
69             <transactionManager type="JDBC"/><!-- 单独使用
时配置成MANAGED没有事务 -->
70             <dataSource type="POOLED">
71                 <property name="driver"
72                     value="${jdbc.driver}"/>
73                 <property name="url"
74                     value="${jdbc.url}"/>
75                 <property name="username"
76                     value="${jdbc.username}"/>
77                 <property name="password"
78                     value="${jdbc.password}"/>
79             </dataSource>
80         </environment>
81     </environments>
```

```

72      </environments>
73
74      <mappers>
75          <mapper resource="BlogMapper.xml"/>
76          <mapper resource="BlogMapperExt.xml"/>
77      </mappers>
78
79  </configuration>

```



## 缓存

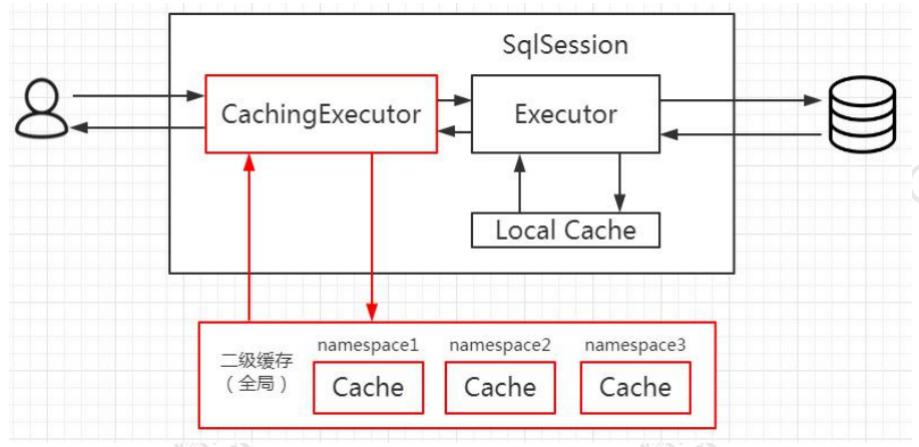
一级缓存也叫本地缓存，MyBatis 的一级缓存是在会话（`SqlSession`）层面进行缓存的。MyBatis 的一级缓存是默认开启的，不需要任何的配置。

二级缓存跨会话, mapper为单位划分，有专门的接口,可以直接用redis取代

```

1 <settings>
2   <setting name="cacheEnabled" value="true"/>
3 </settings>
4 //部分对实时性要求高的sql
5 <select id="selectBlog" resultMap="BaseResultMap"
useCache="false">

```



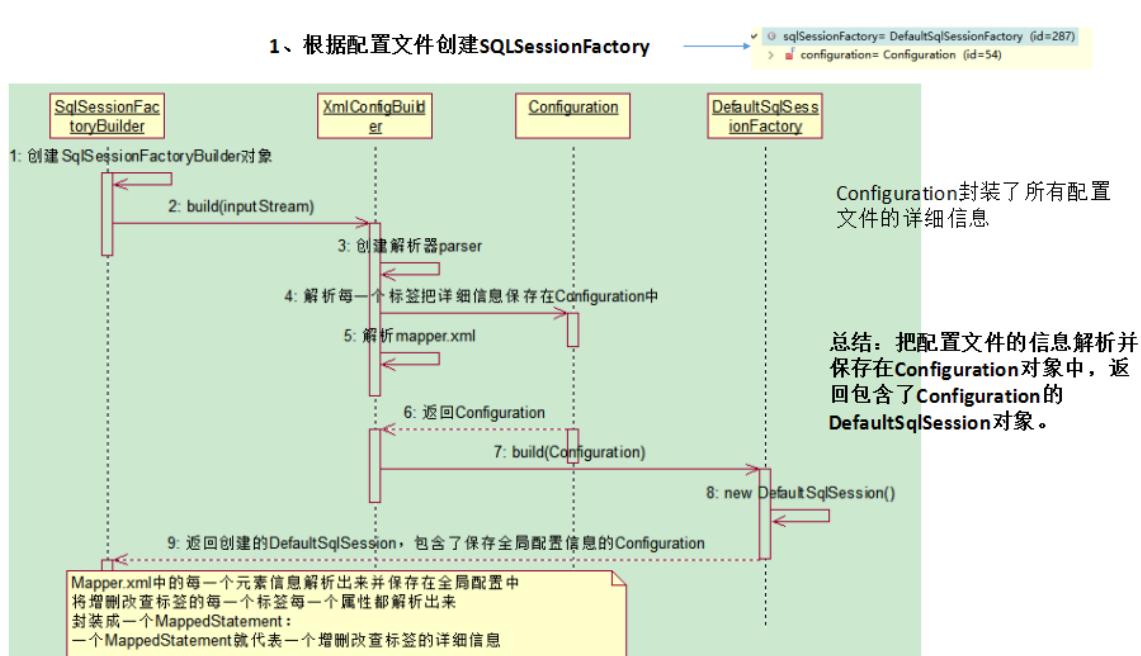
## mybatis执行过程

配置解析过程-->创建会话的过程-->**Mapper 对象的过程**-->执行 SQL

### 配置解析过程

主要完成了 config 配置文件、Mapper 文件、Mapper 接口上的注解的解析。得到一个最重要的对象 Configuration，这里面存放了全部的配置信息，它在属性里面还有各种各样的容器。

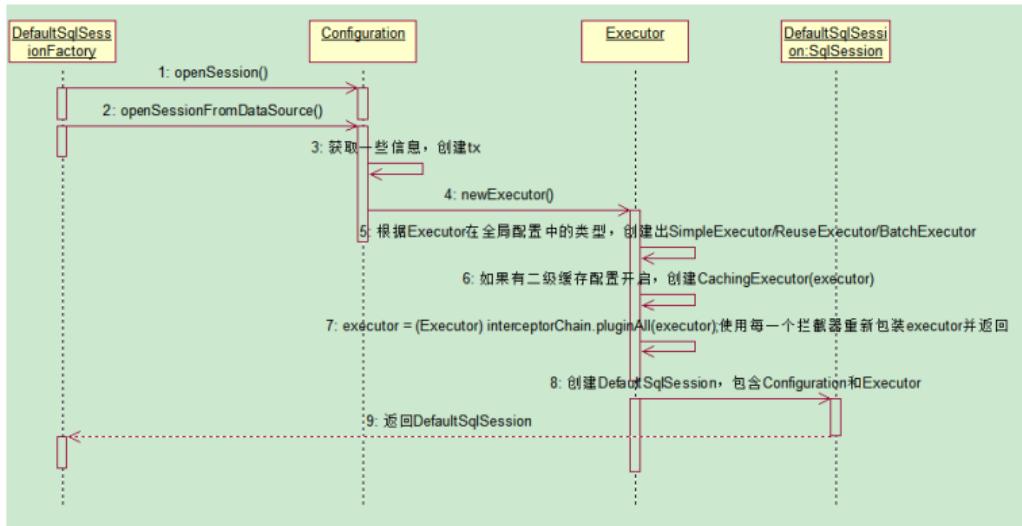
返回了一个 DefaultSqlSessionFactory，里面持有了 Configuration 的实例



### 创建会话的过程

获得了一个 DefaultSqlSession，里面包含了一个 Executor，它是 SQL 的执行者

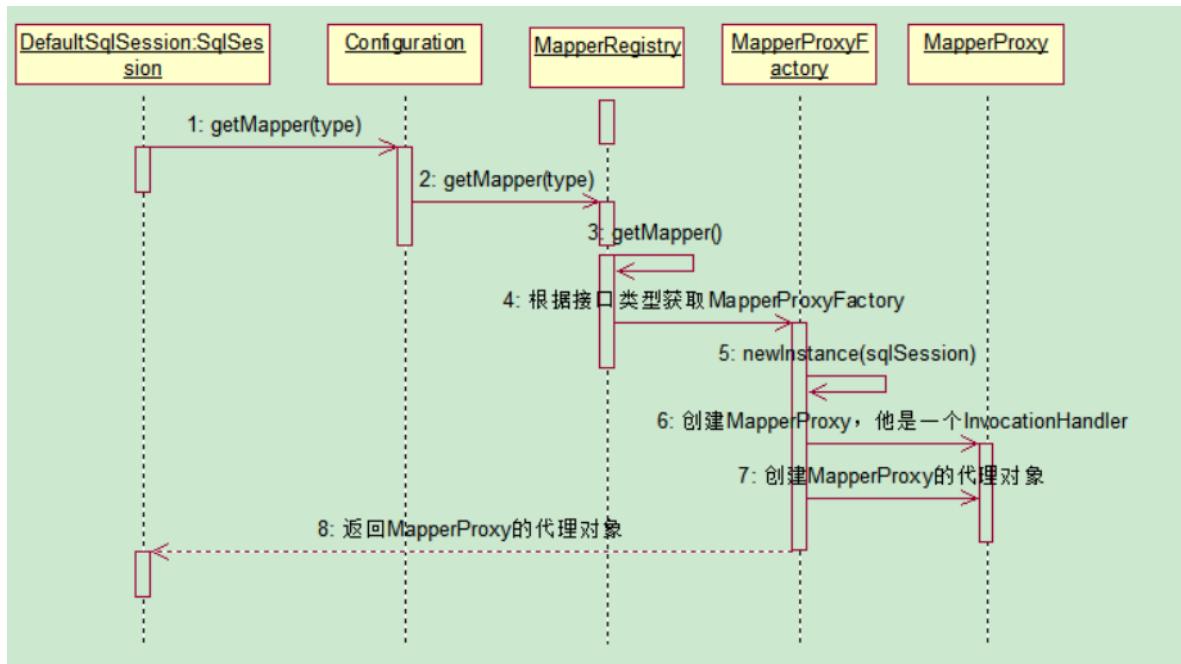
2、返回SqlSession的实现类DefaultSqlSession对象。  
他里面包含了Executor和Configuration；  
Executor会在这一步被创建



## Mapper 对象的过程

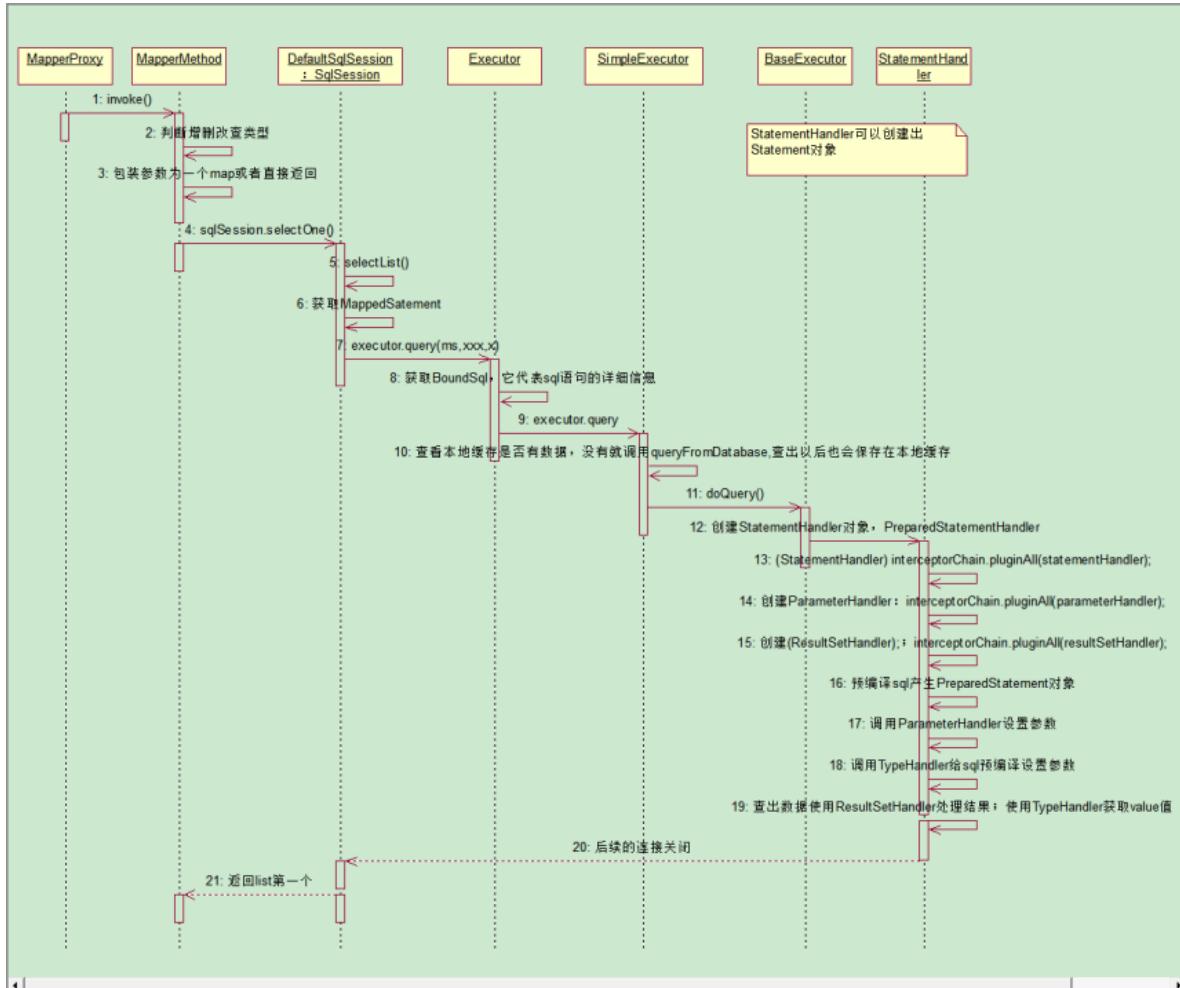
实质上是获取了一个 MapperProxy 的代理对象

mybatis中对动态代理的使用,并没有实现类,正是在**invoke**方法中,直接调用了sqlSession去执行SQL



## 执行 SQL

executor, statementHandler, parameterHandler和resultHandler对象,共同协作完成sql执行,也是插件代理的目标



## 插件原理

插件定义顺序：

插件1  
插件2  
插件3

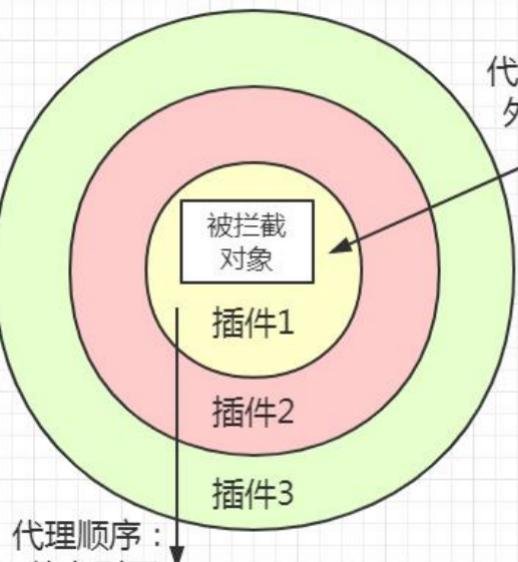
代理顺序：

插件1  
插件2  
插件3

代理执行顺序：

插件3  
插件2  
插件1

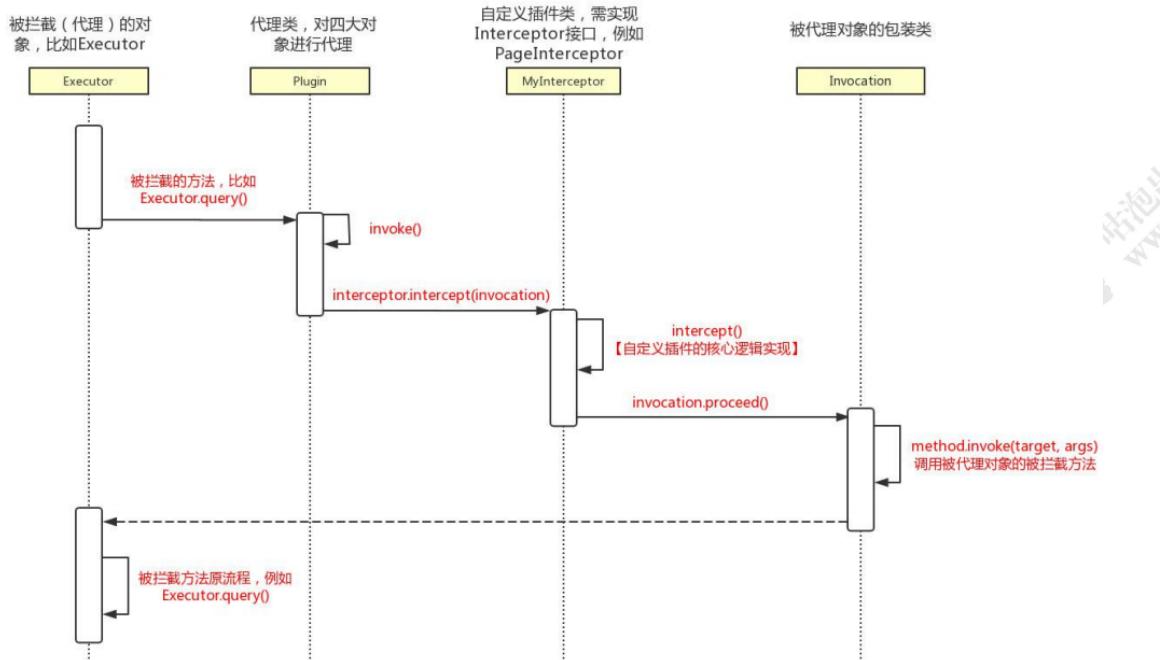
invoke()  
代理执行顺序，从外到内（相反）



- 在四大对象创建的时候
- 1、每个创建出来的对象不是直接返回的，而是
- interceptorChain.pluginAll(parameterHandler);
- 2、获取到所有的Interceptor（拦截器）（插件需要实现的接口）；

- 调用 interceptor.plugin(target); 返回 target 包装后的对象
- 3、插件机制，我们可以使用插件为目标对象创建一个代理对象；AOP（面向切面）
- 我们的插件可以为四大对象创建出代理对象；
- 代理对象就可以拦截到四大对象的每一个执行；

MyBatis插件调用流程



## 使用了设计模式

设计模式	类
工厂	SqlSessionFactory、ObjectFactory、MapperProxyFactory
建造者	XMLConfigBuilder、XMLMapperBuilder、XMLStatementBuilder
单例模式	SqlSessionFactory、Configuration、ErrorContext
代理模式	绑定: MapperProxy 延迟加载: ProxyFactory (CGLIB、JAVASSIT) 插件: Plugin Spring 集成 MyBatis: SqlSessionTemplate 的内部类 SqlSessionInterceptor MyBatis 自带连接池: PooledDataSource 管理的 PooledConnection 日志打印: ConnectionLogger、StatementLogger
适配器模式	logging 模块, 对于 Log4j、JDK logging 这些没有直接实现 slf4j 接口的日志组件, 需要适配器
模板方法	BaseExecutor 与子类 SimpleExecutor、BatchExecutor、ReuseExecutor
装饰器模式	LoggingCache、LruCache 等对 PerpetualCache 的装饰 CachingExecutor 对其他 Executor 的装饰
责任链模式	InterceptorChain

# Springboot

springboot 框架是为了能够帮助使用 spring 框架的开发者快速高效的构建一个基于 spirng 框架以及 spring 生态体系的应用解决方案。它是对“约定优于配置”这个理念下的一个最佳实践。因此它是一个服务于框架的框架，服务的范围是简化配置文件。

约定优于配置的体现主要是

1. maven 的目录结构
  - a) 默认有 resources 文件夹存放配置文件
  - b) 默认打包方式为 jar
2. spring-boot-starter-web 中默认包含 spring mvc 相关依赖以及内置的 tomcat 容器，使得构建一个 web 应用更加简单
3. 默认提供 application.properties/yml 文件
4. 默认通过 spring.profiles.active 属性来决定运行环境时读取的配置文件
5. EnableAutoConfiguration 默认对于依赖的 starter 进行自动装载

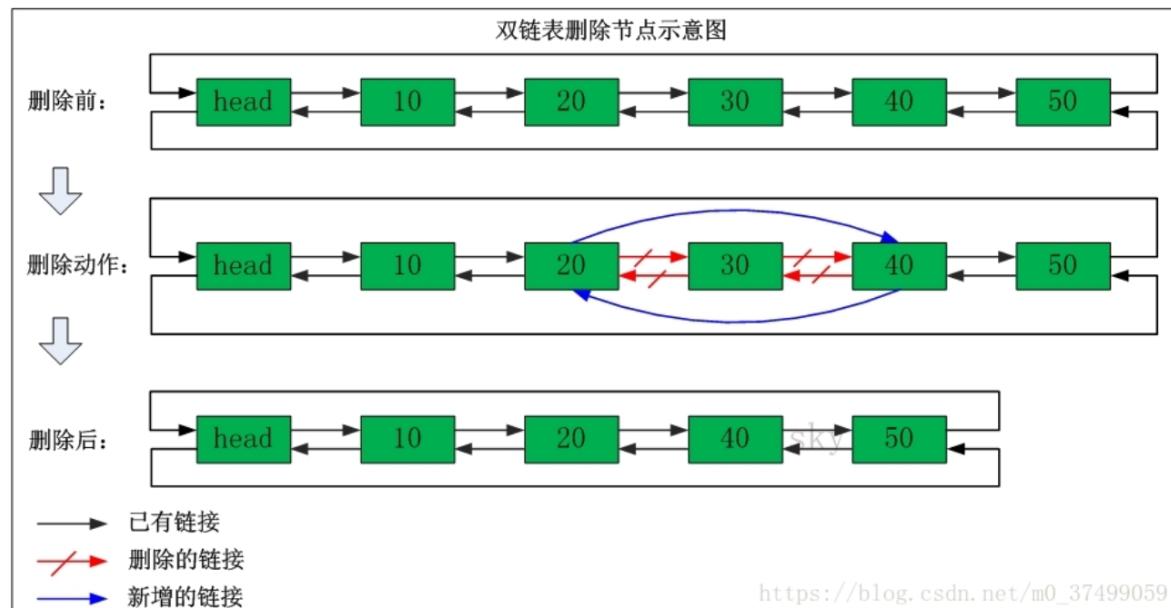
## 分页查询

[https://blog.csdn.net/weixin\\_36279318/article/details/82776632](https://blog.csdn.net/weixin_36279318/article/details/82776632)

## java基础

### 数据结构

#### 双向链表 link



```
1 //双向链表实现
2 public class DoubleLinkedList {
3     private static class Node{
```

```
4     Object value;
5     Node prev = this;
6     Node next = this;
7     Node(Object v){
8         value=v;
9     }
10    public String toString(){
11        return value.toString();
12    }
13    }
14    private Node head = new Node(null);//头结点
15    private int size;//链表大小
16
17    //添加到链表的头部
18    public boolean addFirst(Object o)
19    {
20        addAfter(new Node(o),head);
21        return true;
22    }
23    //添加到链表的尾部
24    public boolean addLast(Object o){
25        addBefore(new Node(o),head);
26        return true;
27    }
28
29    //返回链表大小
30    public int size(){
31        return size;
32    }
33
34    public String toString(){
35        StringBuffer s = new StringBuffer("[");
36        Node node=head;
37        for (int i=0;i<size;i++){
38            node=node.next;
39            if (i>0)
40                s.append(", ");
41            s.append(node.value);
42        }
43        s.append("]");
44        return s.toString();
45    }
46
```

```
47 //查找链表元素
48 private Node getNode(int index){
49     if (index<0||index>=size)
50         throw new IndexOutOfBoundsException();
51     Node node=head.next;
52     for(int i=0;i<index;i++){
53         node=node.next;
54     }
55     return node;
56 }
57
58 //在某元素前添加元素
59 private void addBefore(Node newNode,Node node){
60     newNode.prev=node.prev;
61     newNode.next=node;
62     newNode.next.prev=newNode;
63     newNode.prev.next=newNode;
64     size++;
65 }
66
67 public boolean add(Object o){
68     return addLast(o);
69 }
70
71 //将元素加到指定位置
72 public boolean add(int index,Object o){
73     addBefore(new Node(o),getNode(index));
74     return true;
75 }
76
77 //取到指定位置的元素值
78 public Object get(int index){
79     return getNode(index).value;
80 }
81
82 //在某元素后添加元素
83 private void addAfter(Node newNode,Node node){
84     newNode.prev=node;
85     newNode.next=node.next;
86     newNode.next.prev=newNode;
87     newNode.prev.next=newNode;
88     size++;
89 }
```

```
90
91     //移除特定的元素
92     private void removeNode(Node node){
93         node.next.prev=node.prev;
94         node.prev.next=node.next;
95         node.prev=null;
96         node.next=null;
97         size--;
98     }
99 }
100
101 hashMap自己简单实现
102 public interface MyMap<K,V> {
103     //大小
104     int size();
105     //是否为空
106     boolean isEmpty();
107     //根据key获取元素
108     Object get(Object key);
109     //添加元素
110     Object put(Object key,Object value);
111     interface Entry<k,v>{
112         k getKey();
113         v getValue();
114     }
115 }
116
117
118
119
120
121 /**
122 * 简单来说，HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的，
123 * 如果定位到的数组位置不含链表（当前entry的next指向null），那么对于查找，添加等操作很快，仅需一次寻址即可；
124 * 如果定位到的数组包含链表，对于添加操作，其时间复杂度为O(n)，首先遍历链表，存在即覆盖，否则新增；
125 * 对于查找操作来讲，仍需遍历链表，然后通过key对象的equals方法逐一比对查找。
126 * 所以，性能考虑，HashMap中的链表出现越少，性能才会越好
127 *
128 * @param <K>
```

```
129 * @param <V>
130 */
131 public class MyHashMap<K,V> implements MyMap{
132     //默认容量16
133     private final int DEFALUT_CAPACITY=16;
134     //内部存储结构
135     Node[] table = new Node[DEFALUT_CAPACITY];
136     //长度
137     private int size=0;
138     //keySet
139     Set<K> keySet;
140     @Override
141     public int size() {
142         return this.size;
143     }
144
145     @Override
146     public boolean isEmpty() {
147         return size==0;
148     }
149
150     @Override
151     public Object get(Object key) {
152         int hashvalue = hash(key);
153         int i=indexFor(hashvalue,table.length);
154         for (Node
node=table[i];node!=null;node=node.next){
155             if
(node.key.equals(key)&&hashvalue==node.hash){
156                 return node.value;
157             }
158         }
159         return null;
160     }
161
162     @Override
163     public Object put(Object key, Object value) {
164         //通过key,求hash值
165         int hashvalue=hash(key);
166         //通过hash,找到这个key应该放的位置
167         int i=indexFor(hashvalue,table.length);
168         //i位置已经有数据了,往链表添加元素
```

```
169     for (Node
170         node=table[i];node!=null;node=node.next){
171             Object k;
172             //且数组中有这个key,覆盖其value
173             //先判断小括号里面的两个情况,HashMap中key如果是自
174             //定义对象,一定要重写hashCode和equals方法,不然这里自己false了
175             if (node.hash==hashvalue&&
176                 ((k=node.key)==key||key.equals(k))){
177                 Object oldValue=node.value;
178                 node.value=value;
179                 //返回oldValue
180                 return oldValue;
181             }
182         }
183     }
184
185     public void addEntry(Object key,Object value,int
186     hashvalue,int i){
187         //如果超过了原数组大小,则扩大数组
188         if (++size==table.length){
189             Node[] newTable=new Node[table.length*2];
190             System.arraycopy(table,0,newTable,0,table.length);
191             table=newTable;
192         }
193         //的到i位置的数据
194         Node eNode=table[i];
195         //新增节点,将该节点的next指向前一个节点
196         table[i]=new Node(hashvalue,key,value,eNode);
197
198     //获取插入的位置
199     public int indexFor(int hashvalue,int length){
200         return hashvalue%length;
201     }
202     //获取hash值
203     public int hash(Object key){
204         return key.hashCode();
205     }
```

```
206
207     //静态内部类：Node节点实现Entry接口
208     static class Node implements MyMap.Entry{
209         int hash;//hash值
210         Object key;//key
211         Object value;//value
212         Node next;//指向下一个节点（单例表）
213         Node(int hash, Object key, Object value, Node next)
214         {
215             this.hash=hash;
216             this.key=key;
217             this.value=value;
218             this.next=next;
219         }
220         @Override
221         public Object getKey() {
222             return this.key;
223         }
224         @Override
225         public Object getValue() {
226             return this.value;
227         }
228     }
229 }
230 }
```

## ArrayList

初始10,1.5扩容,无参,有参(指定初始大小)构造

- 1) arrayList可以存放null。
- 2) arrayList本质上就是一个elementData数组。
- 3) arrayList区别于数组的地方在于能够自动扩展大小，其中关键的方法就是grow()方法。
- 4) arrayList中removeAll(collection c)和clear()的区别就是removeAll可以删除批量指定的元素，而clear是全删除集合中的元素。
- 5) arrayList由于本质是数组，所以它在数据的查询方面会很快，而在插入删除这些方面，性能下降很多，有移动很多数据才能达到应有的效果
- 6) arrayList实现了RandomAccess，所以在遍历它的时候推荐使用for循环。

## LinkedList

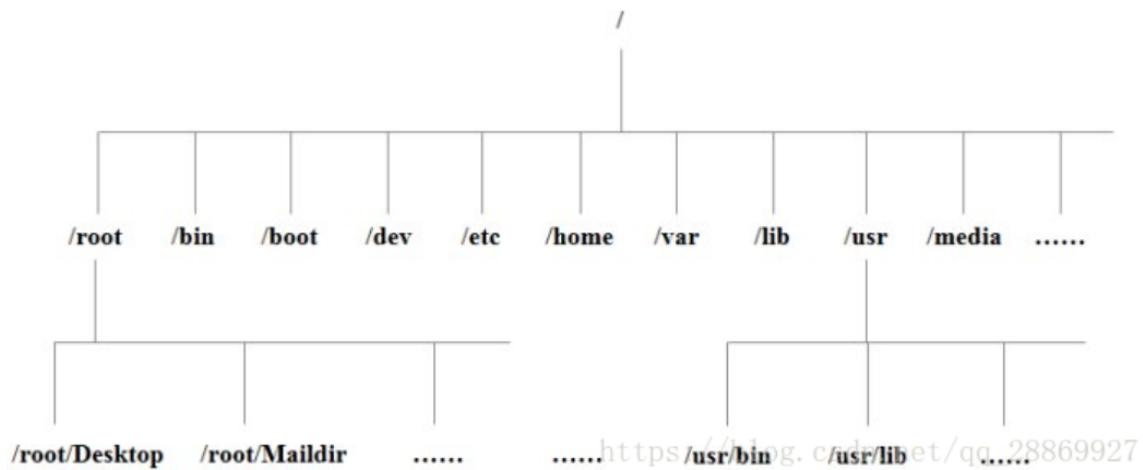
双向链表

HashMap

数组+单链表

# Linux

## 目录结构



# 并发基础知识

## 并发历史：

1. 单核流程, 读卡, 计算打印,

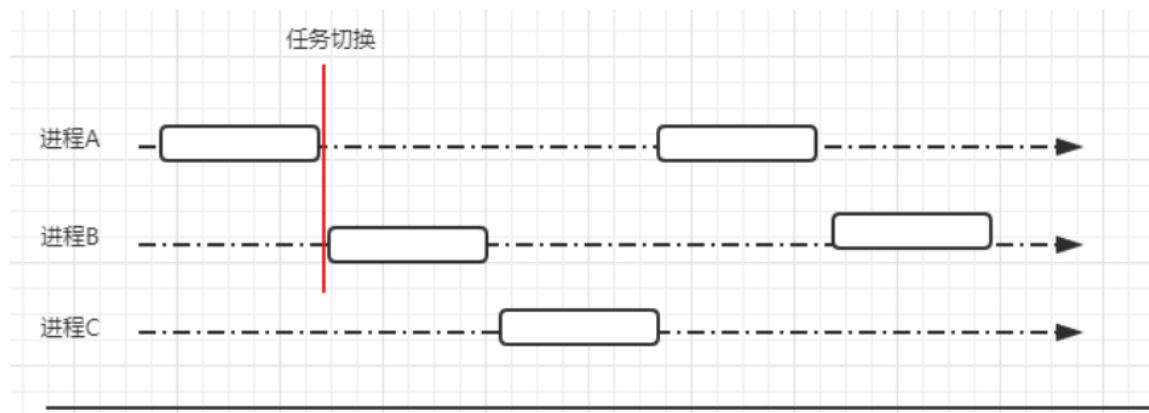
存在问题: 操作员来回调度咨询中计算机处于空闲

解决办法: 引入批处理(先将卡读成磁带)



可能存在一个io阻塞,引入进程概念,进程可以独立分配内存,切换CPU,增大利用率

对单核来说同一时间只有一个进程在跑



## 2.多核CPU时代,实现真正的并发处理

一个应用程序中可能存在多个任务处理,引入线程概念(轻量级进程)

## 3.线程的应用

在java中实现多线程的方式,

1.继承Thread类, 重写run方法

2.实现Runnable接口, 重写run方法, 实现Runnable接口的实现类的实例对象作为Thread构造函数的target

3.通过Callable和FutureTask创建线程

4.通过线程池创建线程

```

1 //继承Thread类,
2 public class MyThread extends Thread {
3     public void run() {
4         System.out.println("MyThread.run()");
5     }

```

```
6 }
7 MyThread myThread1 = new MyThread();
8 MyThread myThread2 = new MyThread();
9 myThread1.start();
10 myThread2.start();
11
12 //实现 Runnable 接口创建线程
13 //如果自己的类已经 extends 另一个类，就无法直接 extends
14 Thread，此时，可以实现一个 Runnable 接口
15 public class MyThread extends OtherClass implements
16 Runnable {
17     public void run() {
18         System.out.println("MyThread.run()");
19     }
20 }
21 //实现 Callable 接口通过 FutureTask 包装器来创建 Thread 线程，
22 带返回值
22 public class CallableDemo implements Callable<String> {
23     public static void main(String[] args) throws
24         ExecutionException, InterruptedException {
25         ExecutorService executorService=
26             Executors.newFixedThreadPool(1);
27         CallableDemo callableDemo=new CallableDemo();
28         Future<String>
29         future=executorService.submit(callableDemo);
30         System.out.println(future.get());
31         executorService.shutdown();
32     }
33     @Override
34     public String call() throws Exception {
35         int a=1;
36         int b=2;
37         System.out.println(a+b);
38         return "执行结果:"+ (a+b);
39     }
}
```

## 多线程的应用

工作中很少用到多线程开发的原因：

已经使用分布式以及异步的消息处理了,但很多开源框架底层用到了,简单来说,大牛已经帮你做了异步责任链模式例子分享,使用多线的妙用

```
1 //LinkedBlockingQueue内部由单链表实现, 只能从head取元素, 从tail添加元素。添加元素和获取元素都有独立的锁, 也就是说  
2 //LinkedBlockingQueue是读写分离的, 读写操作可以并行执行。  
3 //LinkedBlockingQueue采用可重入锁(ReentrantLock)来保证在并发情况下的线程安全。  
4  
5 Request  
6 public class Request {  
7     private String name;  
8     public String getName() {  
9         return name;  
10    }  
11    public void setName(String name) {  
12        this.name = name;  
13    }  
14    @Override  
15    public String toString() {  
16        return "Request{" +  
17            "name='" + name + '\'' +  
18            '}';  
19    } }  
20  
21 RequestProcessor  
22 public interface RequestProcessor {  
23     void processRequest(Request request);  
24 }PrintProcessor  
25 public class PrintProcessor extends Thread implements  
26 RequestProcessor{  
27     LinkedBlockingQueue<Request> requests = new  
28     LinkedBlockingQueue<Request>();  
29     private final RequestProcessor nextProcessor;  
30     public PrintProcessor(RequestProcessor nextProcessor) {  
31         this.nextProcessor = nextProcessor;  
32     }  
33     @Override  
34     public void run() {  
35         while (true) {  
36             try {  
37                 Request request=requests.take();  
38                 System.out.println("print  
39 data:"+request.getName());  
40             } catch (InterruptedException e) {  
41                 e.printStackTrace();  
42             }  
43         }  
44     }  
45 }
```

```
37     nextProcessor.processRequest(request);
38 } catch (InterruptedException e) {
39     e.printStackTrace();
40 }
41 }
42 }
43 //处理请求
44 public void processRequest(Request request) {
45     requests.add(request);
46 } }
47
48
49 SaveProcessor
50 public class SaveProcessor extends Thread implements
51 RequestProcessor{
52     LinkedBlockingQueue<Request> requests = new
53     LinkedBlockingQueue<Request>();
54     @Override
55     public void run() {
56         while (true) {
57             try {
58                 Request request=requests.take();
59                 System.out.println("begin save request
60 info:"+request);
61             } catch (InterruptedException e) {
62                 e.printStackTrace();
63             }
64         }
65     }
66 //处理请求
67 public void processRequest(Request request) {
68     requests.add(request);
69 } }
70 Main
71 public class Main {
72     PrintProcessor printProcessor;
73     protected Main(){
74         SaveProcessor saveProcessor=new SaveProcessor();
75         saveProcessor.start();
76         printProcessor=new PrintProcessor(saveProcessor);
77         printProcessor.start();
78     }
79     private void doTest(Request request){
```

```

80 printProcessor.processRequest(request);
81 }
82 public static void main(String[] args) {
83 Request request=new Request();
84 request.setName("Mic");
85 new Main().doTest(request);
86 } }

```

## 线程的生命周期

new: 初始状态

RUNABLE:运行状态

BLOCKED:阻塞状态

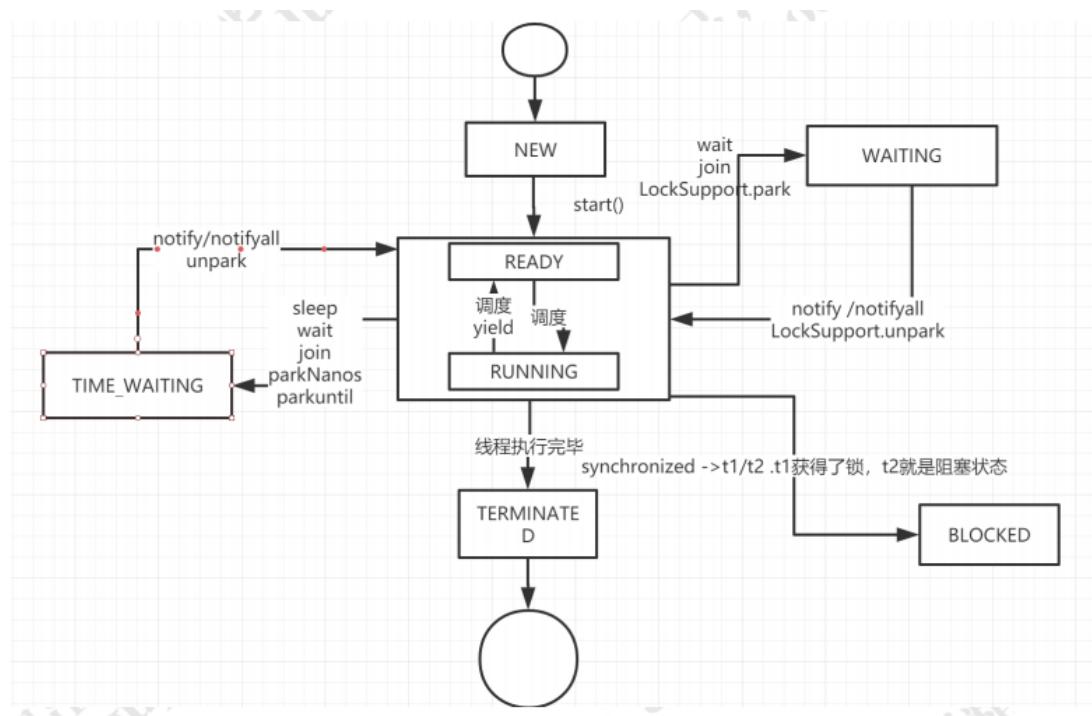
等待阻塞：运行的线程执行 wait 方法， jvm 会把当前 线程放入到等待队列

同步阻塞：运行的线程在获取对象的同步锁时，若该同 步锁被其他线程锁占用了，那么 jvm 会把当前的线程 放入到锁池中

其他阻塞：运行的线程执行 Thread.sleep 或者 t.join 方法，或者发出了 I/O 请求时，JVM 会把当前线程设置 为阻塞状态，当 sleep 结束、join 线程终止、io 处理完 毕则线程恢复

TIME\_WAITING：超时等待状态，超时以后自动返回

TERMINATED：终止状态，表示当前线程执行完毕



sleep(),睡眠,不释放锁,等待唤醒,必须加休眠时间

**wait(),等待时间,释放锁,定时或被唤醒,notify,醒来,抢锁继续工作**

park, LockSupport.park()底层是调用的Unsafe的native方法;

LockSupport.unpark()方法唤醒; **不会释放当前线程占有的锁,任意地方调用**

```
1 //案例
2 public class ThreadStatus {
3     public static void main(String[] args) {
4         //TIME_WAITING
5         new Thread(()->{
6             while(true){
7                 try {
8                     TimeUnit.SECONDS.sleep(100);
9                 } catch (InterruptedException e) {
10                     e.printStackTrace();
11                 }
12             }
13         }, "timewaiting").start();
14         //WAITING, 线程在 ThreadStatus 类锁上通过 wait 进行等待
15         new Thread(()->{
16             while(true){
17                 synchronized (ThreadStatus.class){
18                     try {
19                         ThreadStatus.class.wait();
20                     } catch (InterruptedException e) {
21                         e.printStackTrace();
22                     }
23                 }
24             }
25         }, "waiting").start();
26         //线程在 ThreadStatus 加锁后, 不会释放锁
27         new Thread(new BlockedDemo(), "BlockDemo-01").start();
28         new Thread(new BlockedDemo(), "BlockDemo-02").start();
29     }
30     static class BlockedDemo extends Thread{
31         public void run(){
32             synchronized (BlockedDemo.class){
33                 while(true){
34             }
35         }
36     }
37 }
```

```
37 try {  
38     TimeUnit.SECONDS.sleep(100);  
39 } catch (InterruptedException e) {  
40     e.printStackTrace();  
41 }  
42 }  
43 }  
44 }  
45 } }
```

线程启动用的是Start(),实际用的是native方法start0()来启动一个线程,JVM\_StartThread 方法;

在 hotspot 的源码中找到jvm.cpp 文件,那么在 JVM 层面,一定会调用 Java 中定义的 run 方法

```
> JVM_ENTRY(void, JVM_StartThread(JNIEnv* env, jobject jthread))  
>     JVMWrapper("JVM_StartThread");  
>     JavaThread *native_thread = NULL;  
>
```

## 线程的终止

stop(过时)要优雅的去中断一个线程,在线程中提供了一个 interrupt 方法,isInterrupted()来**判断是否被中断**这种通过标识位或者中断操作的方式能够使线程在终止时 有机会去清理资源,而不是武断地将线程停止

```
1 通过下面这个例子, 来实现了线程终止的逻辑  
2 public class InterruptDemo {  
3     private static int i;  
4     public static void main(String[] args) throws  
5         InterruptedException {  
6         Thread thread=new Thread(()->{  
7             while(!Thread.currentThread().isInterrupted()){//默认情  
8                 //况下  
9                 isInterrupted 返回 false、通过 thread.interrupt 变成了 true  
10                i++;  
11            }  
12            System.out.println("Num:"+i);  
13            }, "interruptDemo");  
14            thread.start();  
15            TimeUnit.SECONDS.sleep(1);  
16            thread.interrupt(); //加和不加的效果  
17        }}
```

## Thread.interrupted

上面的案例中，通过 interrupt，设置了一个标识告诉线程可以终止了，线程中还提供了静态方法 Thread.interrupted()对设置中断标识的线程复位

### 其他的线程复位

抛出 InterruptedException 异常的方法，在 InterruptedException 抛出之前，JVM 会先把线程的中断标识位清除，然后才会抛出 InterruptedException，这个时候如果调用 isInterrupted 方法，将会返回 false

### 线程的终止原理

```
public class Thread implements Runnable {
    ...
    public void interrupt() {
        if (this != Thread.currentThread())
            checkAccess();
        synchronized (blockerLock) {
            Interruptible b = blocker;
            if (b != null) {
                interrupt0();           // Just to set the interrupt flag
                b.interrupt(this);
                return;
            }
        }
        interrupt0();
    }
    ...
}
```

这个方法里面，调用了 interrupt0()，找到 jvm.cpp 文件，找到 JVM\_Interrupt 的定义

```
JVM_ENTRY(void, JVM_Interrupt(JNIEnv* env, jobject jthread))
JVMWrapper("JVM_Interrupt");
// Ensure that the C++ Thread and OS Thread structures aren't freed before we operate
oop java_thread = JNIHandles::resolve_non_null(jthread);
MutexLockerEx ml(thread->threadObj() == java_thread ? NULL : Threads_lock);
// We need to re-resolve the java_thread, since a GC might have happened during the
// acquire of the lock
JavaThread* thr = java_lang_Thread::thread(JNIHandles::resolve_non_null(jthread));
if (thr != NULL) {
    Thread::interrupt(thr);
}
JVM_END
```

Thread::interrupt(thr)这个方法，这个方法的定义在 Thread.cpp 文件中，代码如下

```
void Thread::interrupt(Thread* thread) {
    trace("interrupt", thread);
    debug_only(check_for_dangling_thread_pointer(thread));
    os::interrupt(thread);
}
```

Thread::interrupt 方法调用了 os::interrupt 方法，这个是调用平台的 interrupt 方法

以Linux为例，设置标识符true中断线程

```
void os::interrupt(Thread* thread) {
    assert(Thread::current() == thread || Threads_lock->owned_by_self(),
           "possibility of dangling Thread pointer");
    // 获取本地线程对象
    OSThread* osthread = thread->osthread();
    if (!osthread->interrupted()) { // 判断本地线程对象是否为中断
        osthread->set_interrupted(true); // 设置中断状态为true
        // More than one thread can get here with the same value of osthread,
        // resulting in multiple notifications. We do, however, want the store
        // to interrupted() to be visible to other threads before we execute unpark().
        // 这里是内存屏障，这块在后续的文章中会剖析；内存屏障的目的是使得interrupted状态对其他线程立即可见
        OrderAccess::fence();
        //_SleepEvent相当于Thread.sleep，表示如果线程调用了sleep方法，则通过unpark唤醒
        ParkEvent * const slp = thread->_SleepEvent ;
        if (slp != NULL) slp->unpark(); // 唤醒线程
    }
    // For JSR166. Unpark even if interrupt status already was set
    if (thread->is_Java_thread())
        ((JavaThread*)thread)->parker()->unpark();
    // _ParkEvent用于synchronized同步块和Object.wait()，这里相当于也是通过unpark进行唤醒
    ParkEvent * ev = thread->_ParkEvent ;
    if (ev != NULL) ev->unpark(); // 唤醒线程
}
```

对于 synchronized 阻塞的线程，被唤醒以后会继续尝试获取锁，如果失败仍然可能被 park

在调用 ParkEvent 的 park 方法之前，会先判断线程的中断状态，如果为 true，会清除当前线程的中断标识

Object.wait、Thread.sleep、Thread.join 会抛出 InterruptedException  
thread.interrupt()方法实际

Object.wait、Thread.sleep 和 Thread.join 都是属于阻塞的方法

而阻塞方法的释放会取决于一些外部的事件，但是阻塞方法可能因为等不到外部的触发事件而导致无法终止，所以它允许一个线程请求自己来停止它正在做的事情。当一个方法抛出 InterruptedException 时，它是在告诉调用者如果执行该方法的线程被中断，它会尝试停止正在做的事情并且通过抛出

`InterruptedException` 表示提前返回。所以，这个异常的意思是表示一个阻塞被其他线程中断了。然后，由于线程调用了 `interrupt()` 中断方法，那么 `Object.wait`、`Thread.sleep` 等被阻塞的线程被唤醒以后会

通过 `is_interrupted` 方法判断中断标识的状态变化，如果发现中断标识为 `true`，则先清除中断标识，然后抛出

`InterruptedException`

## 多线程原理及挑战

---

线程的合理使用能够提升程序的处理性能，主要有两个方面，

第一个是能够利用**多核 cpu** 以及超线程技术来实现线程的并行执行

第二个是线程的异步化执行相比于同步执行来说，**异步执行**能提升并发吞吐量

## 线程安全性定义

### 多线程对于共享变量访问带来的安全性问题

解决办法：锁(互斥) 强行并行转为串行，Java 提供的加锁方法就是 `Synchronized` 关键字

## `synchronized` 的基本认识

在多线程并发编程中 `synchronized` 一直是元老级角色，很多人都会称呼它为重量级锁。但是，随着 Java SE 1.6 对 `synchronized` 进行了各种优化之后，有些情况下它就并不那么重，Java SE 1.6 中为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁

`synchronized` 有三种方式来加锁，分别是

1. 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁(类似行锁)
2. 静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁(类似表锁)
3. 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。不同的修饰类型，代表锁的控制粒度(表锁和行锁)

## 锁的标识

## 在字节码对象头中有Mark word

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否偏向锁	锁标志位
无锁	对象的HashCode		分代年龄	0	01
偏向锁	线程ID	Epoch	分代年龄	1	01
轻量级锁	指向栈中锁记录的指针			00	
重量级锁	指向重量级锁的指针			10	
GC标记	空			11	

所有的 Java 对象是天生携带 monitor() 监视器，多个线程访问同步代码块时，相当于去争抢对象监视器 修改对象中的锁标识

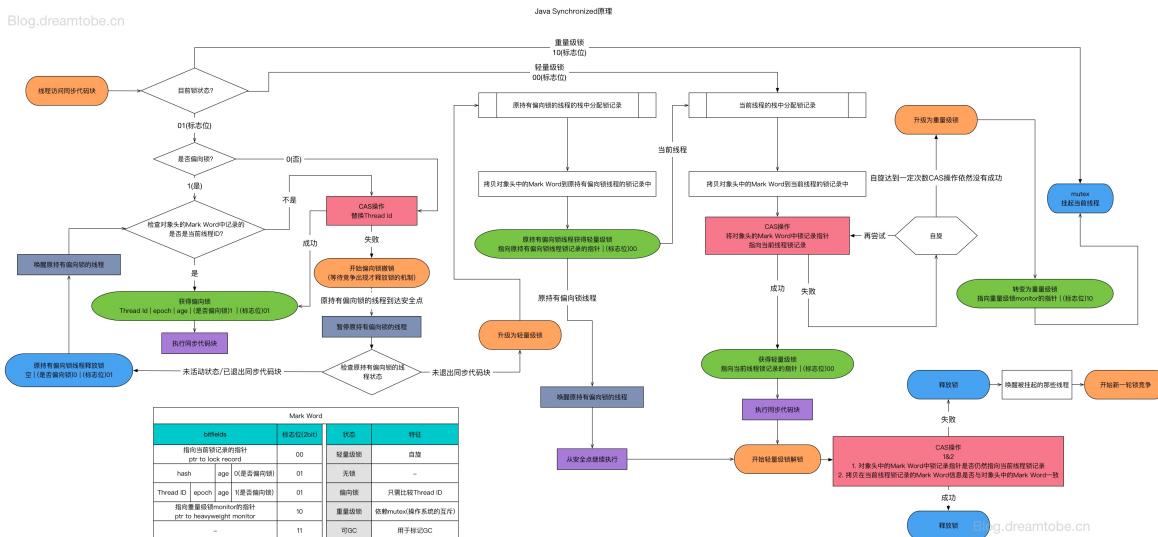
## synchronized 锁的升级

问题：使用锁能够实现数据的安全性，但是会带来性能的下降。不使用锁能够基于线程并行提升程序性能，但是却不能保证线程安全性

调查发现：

大部分情况下，加锁的代码不仅仅不存在多线程竞争，而且 **总是由同一个线程多次获得**。所以基于这样一个概率，是的 synchronized 在 JDK1.6 之后做了一些优化，默认开启偏向锁、轻量级锁的概念。因此大家会发现在 synchronized 中，锁存在四种状态

分别是：无锁0\_01、偏向锁1\_01、轻量级锁00、重量级锁10；锁的状态根据竞争激烈的程度从低到高不断升级。



## 偏向锁

## **偏向锁的基本原理**

根据概率,总是由同一个线程 多次获得,引入 偏向锁

当一个线程访问加了同步锁的代码块时,会在对象头中存储当前线程的 ID,后续这个线程进入和退出这段加了同步锁的代码块时,不需要再次加锁和释放锁。而是直接比较 对象头里面是否存储了指向当前线程的偏向锁。如果相等 表示偏向锁是偏向于当前线程的,就不需要再尝试获得锁了

## **偏向锁的获取和撤销逻辑**

1. 首先获取锁 对象的 Markword, 判断是否处于可偏向状态。  
(biased\_lock=1、且 ThreadId 为空)
2. 如果是可偏向状态, 则通过 CAS 操作, 把当前线程的 ID 写入到  
MarkWord
  - a) 如果 cas 成功, 那么 markword 就会变成这样。 表示已经获得了锁对象的偏向锁, 接着执行同步代码 块
  - b) 如果 cas 失败, 说明有其他线程已经获得了偏向锁, 这种情况说明当前锁存在竞争, 需要撤销已获得偏向锁的线程, 并且把它持有的锁升级为轻量级锁 (这个操作需要等到全局安全点, 也就是没有线程在执行字节码) 才能执行
3. 如果是已偏向状态, 需要检查 markword 中存储的 ThreadID 是否等于当前线程的 ThreadID
  - a) 如果相等, 不需要再次获得锁, 可直接执行同步代码 块
  - b) 如果不相等, 说明当前锁偏向于其他线程, 需要撤销

偏向锁并升级到轻量级锁

## **偏向锁的撤销**

偏向锁的撤销并不是把对象恢复到无锁可偏向状态 (因为 偏向锁并不存在锁释放的概念), 而是在获取偏向锁的过程 中, 发现 cas 失败也就是存在线程竞争时, 直接把被偏向 的锁对象升级到被加了轻量级锁的状态。对原持有偏向锁的线程进行撤销时, 原获得偏向锁的线程

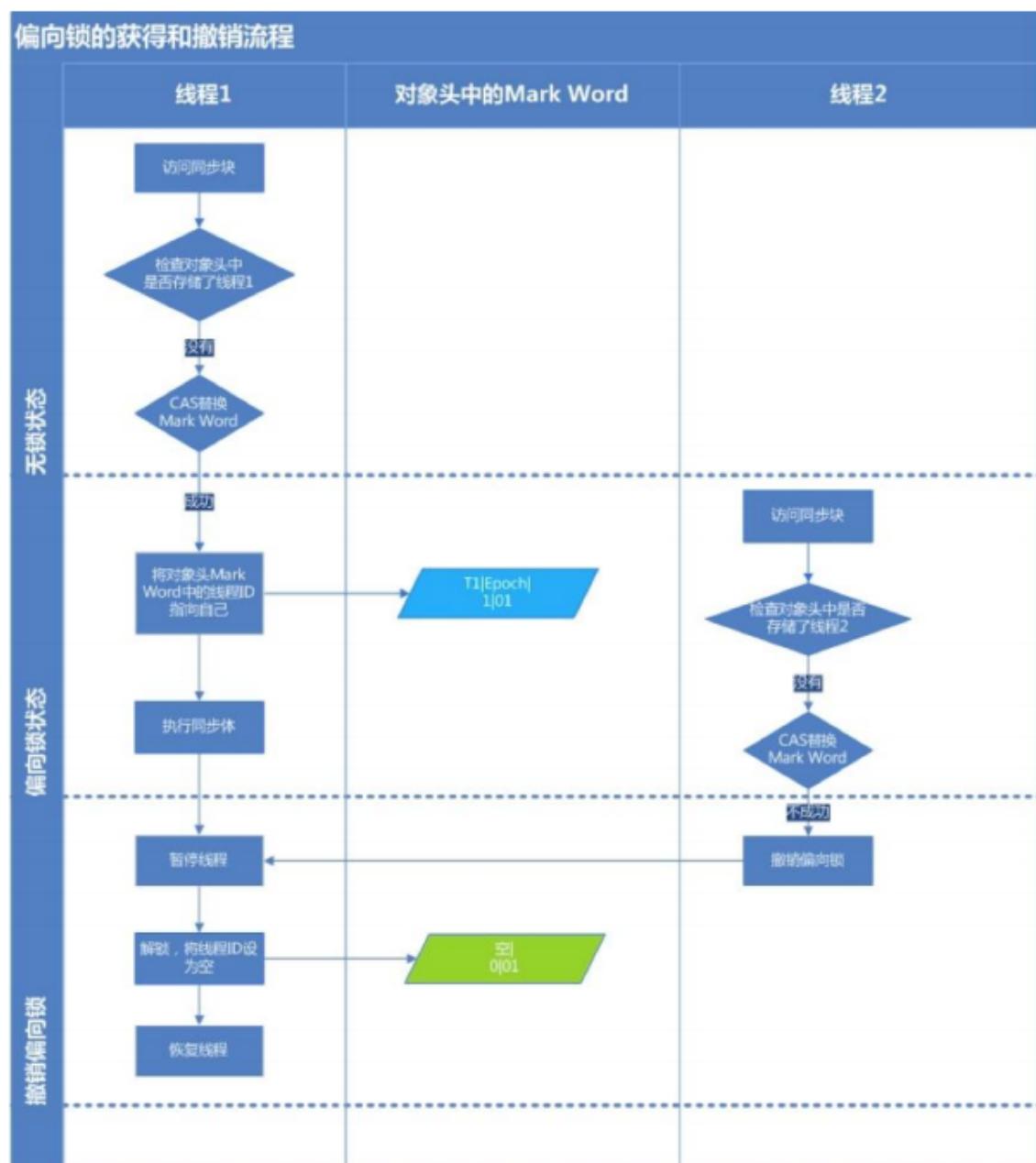
有两种情况:

1. 原获得偏向锁的线程如果已经退出了临界区, 也就是同步代码块执行完了, 那么这个时候会把对象头设置成无 锁状态并且争抢锁的线程可以基于 CAS 重新偏向但前 线程

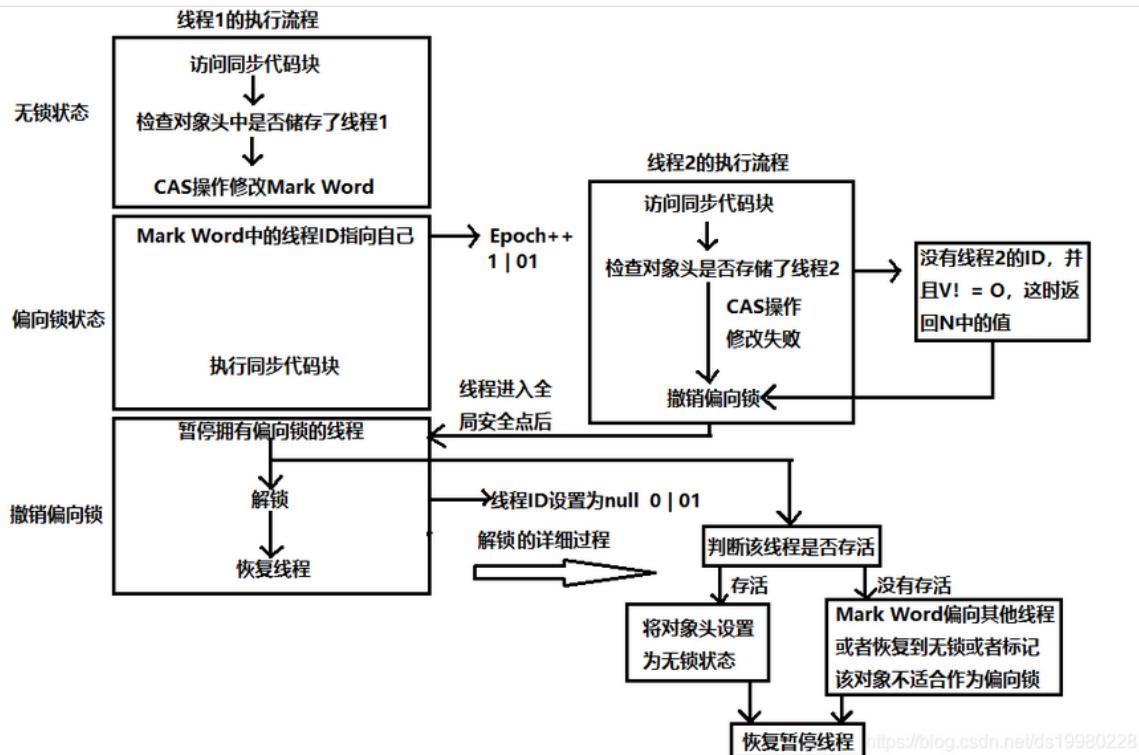
2. 如果原获得偏向锁的线程的同步代码块还没执行完，处于临界区之内，这个时候会把原获得偏向锁的线程升级为轻量级锁后继续执行同步代码块 在我们的应用开发中，绝大部分情况下一定会存在 2 个以上的线程竞争，那么如果开启偏向锁，反而会提升获取锁的资源消耗。所以可以通过 jvm 参数

UseBiasedLocking 来设置开启或关闭偏向锁

## 流程图分析



## 偏向锁的获得和撤销流程



## 轻量级锁的加锁和解锁逻辑

锁升级为轻量级锁之后，对象的 Markword 也会进行相应的变化。升级为轻量级锁的过程：

1. 线程在自己的栈桢中创建锁记录 LockRecord。
2. 将锁对象的对象头中的MarkWord复制到线程的刚刚创建的锁记录中。
3. 将锁记录中的 Owner 指针指向锁对象。
4. 将锁对象的对象头的 MarkWord 替换为指向锁记录的指针。

## CAS的操作过程

CAS操作的就是乐观锁(多线程无锁化机制)，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止

CAS的缺点：

- 1.CPU开销较大 (一般会限定次数)
- 2.不能保证代码块的原子性(只能保证一个变量的原子性)
- 3.ABA问题。CAS在操作值的时候检查值是否已经变化，没有变化的情况下才会进行更新。但是如果一个值原来是A，变成B，又变成A，那么CAS进行检查时会认为这个值没有变化，但是实际上却变化了。ABA问题的解决方法是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A -

B - A 就变成1A-2B - 3A。从Java1.5开始JDK的atomic包里提供了一个类

### CAS的操作过程和 (V, O, N) 三个值有关

	V	O	N
具体含义	内存中地址存放的实际值	预期值 (旧值)	更新后的值

下面就介绍一下操作的具体过程：

CAS在最开始的时候V和O的是相等的，N中的每次线程要进行CAS操作时要新放入的值。当要进行CAS操作时，要先判断一下V和O，若相等，说明没有V中的值还没有被其他线程更改，这时就可以将N中的值替换到V中。若不相等表明N中的值已经被其他的线程所更改，这时直接将N中的值返回即可；

当多个线程同时进行CAS操作时，只有一个线程会成功，并且更新V的值，其余的线程会失败。失败后可以选择不断的进行CAS操作，也可以直接挂起进行等待；

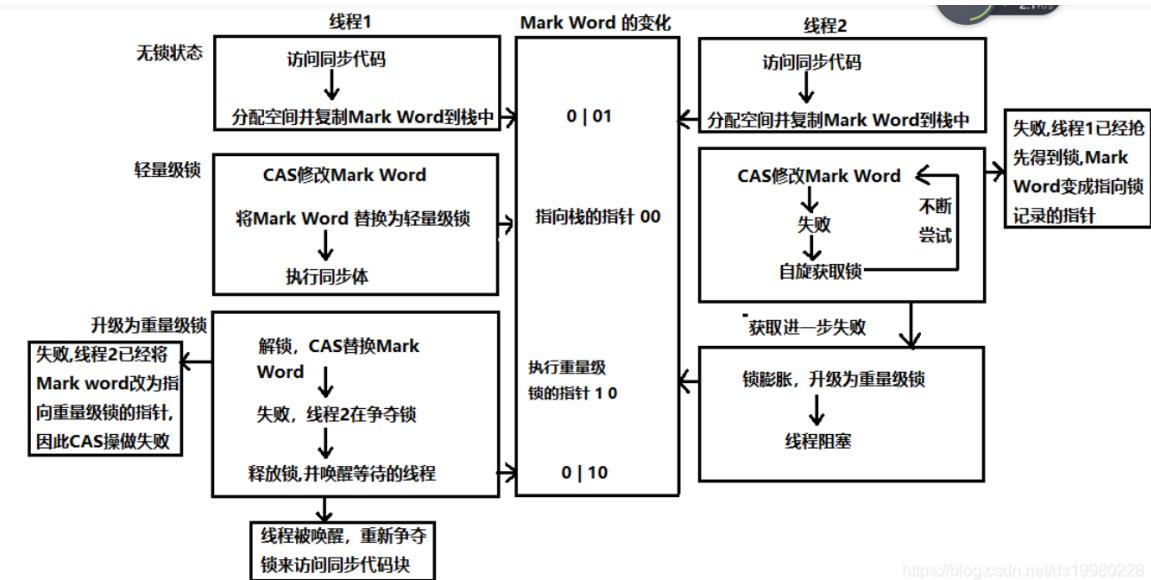
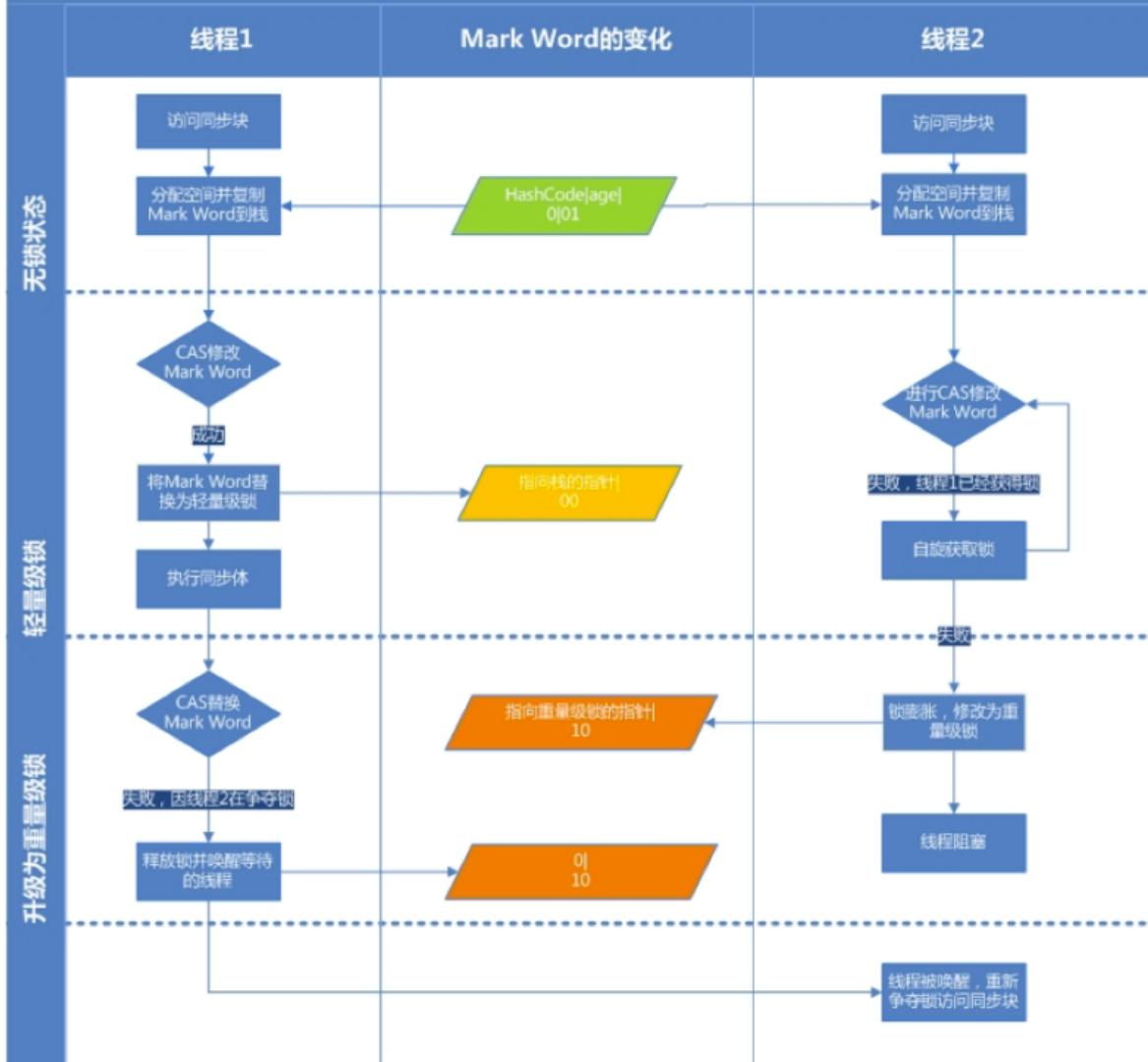
### 自旋锁

就是CAS循环访问，CAS 可以实现乐观锁，它实际上是直接利用了 CPU 层面的指令，所以性能很高。**保证原子操作**，默认情况下自旋的次数是 10 次，可以通过 preBlockSpin 来修改 在 JDK1.6 之后，引入了自适应自旋锁，根据前面的表现来增加次数，成功次数多的多给几次机会

### 轻量级锁的解锁

轻量级锁的锁释放逻辑其实就是获得锁的逆向逻辑，通过 CAS 操作把线程栈帧中的 LockRecord 替换回到锁对象的 MarkWord 中，如果成功表示没有竞争。如果失败，表示当前锁存在竞争，那么轻量级锁就会膨胀成为重量级锁

## 轻量级锁及膨胀流程图



<https://blog.csdn.net/ds19980228>

## 重量级锁的基本原理

当轻量级锁膨胀到重量级锁之后，意味着线程只能被挂起 阻塞来等待被唤醒了

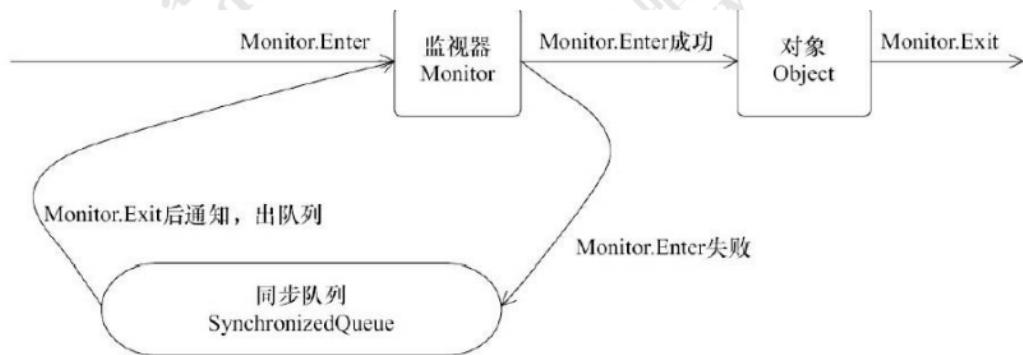
### 重量级锁的 monitor

加了同步代码块以后，在字节码中会看到一个 monitoreenter 和 monitorexit。每一个 JAVA 对象都会与一个监视器 monitor 关联，我们可以把它理解成一把锁，当一个线程想要执行一段被 synchronized 修饰的同步方法或者代码块时，该线程得先 获取到 synchronized 修饰的对象对应的 monitor。

monitoreenter 表示去获得一个对象监视器。monitorexit 表示释放 monitor 监视器的所有权，使得其他被阻塞的线程 可以尝试去获得这个监视器 monitor 依赖操作系统的 MutexLock(互斥锁)来实现的，线程被阻塞后便进入内核

(Linux) 调度状态，这个会导致系统在用户态与内核态之间来回切换，严重影响锁的性能

## 重量级锁的基本流程



任意线程对 Object (Object 由 synchronized 保护) 的访问，首先要获得 Object 的监视器。如果获取失败，线程进入同步队列，线程状态变为 BLOCKED。当访问 Object 的前驱（获得了锁的线程）释放了锁，则该释放操作唤醒阻塞在同步队列中的线程，使其重新尝试对监视器的获取。

wait：表示持有对象锁的线程 A 准备释放对象锁权限，释放 cpu 资源并进入等待状态。

notify：表示持有对象锁的线程 A 准备释放对象锁权限，通知 jvm 唤醒某个竞争该对象锁的线程 X。线程 A synchronized 代码执行结束并且释放了锁之后，线程 X 直接获得对象锁权限，其他竞争线程继续等待(即使线程 X 同步完毕，释放对象锁，其他竞争线程仍然等待，直至有新的 notify ,notifyAll 被调用)。

三种锁的总结：

偏向锁，直接字节码中被标识，没有加锁解锁开销，适用一个线程同步代码块

轻量级锁，cas 循环访问，消耗cpu，线程竞争不阻塞，适用响应速度快，及多线不同时间访问

重量级锁，不用自选

### 三种锁的对比：

锁	优点	缺点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。	如果线程间存在锁竞争，会带来额外的锁撤销的消耗。	适用于一个线程访问同步代码块
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度。	如果始终得不到锁竞争的线程使用自旋会消耗CPU。	响应时间很短，同步块执行速度非常快，适用于多个线程在不同时间段申请同一把锁
重量级锁	线程竞争不使用自旋，不会消耗CPU。	线程阻塞，响应时间缓慢。	追求吞吐量。 同步块执行速度较长。

## 可见性的本质

### 从硬件层面了解可见性的本质

CPU速度>内存>磁盘(外部io),木桶效应

最大化利用CPU性能,从硬件,操作系统,编译器优化

- 1.CPU增加高速缓存(I,II,III级),-->缓存一致性
- 2.操作系统增加进程,线程,通过CPU的时间切换提升使用效率
- 3.编译器指令优化.

这些优化带来了对应的问题,这些问题导致了线程安全的根源

## 优化原理

多核CPU发展,硬件使用了MESI协议自动优化指令代码,不清楚软件层面的依赖关系,引出内存屏障的概念防止cpu自动重排序,软件自我实现,保证可见性.

为了解决缓存不一致的问题,在CPU层面做了很多事情,主要提供了两种解决办法

1. 总线锁 :在总线上发出一个 LOCK# 信号,这个信号使得其他处理器无法通过总线来访问到共享内存中的数据,总线锁定把 CPU 和内存之间的通信锁住了,这使得锁定期间,其他处理器不能操作其他内存地址的数据开销大,不合适
2. 缓存锁:降低颗粒度,只需要保证对于被多个 CPU 缓存的同一份数据是一致的就行.缓存一致性协议来实现.如MSI, MESI, MOSI 等。最常见的就是 MESI 协议

## MESI 协议

MESI 表示缓存行的四种状态,分别是

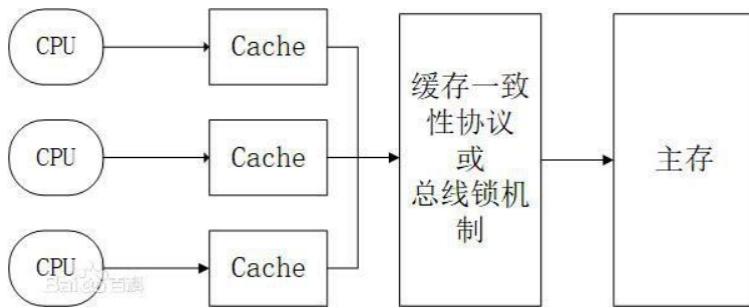
1. M(Modify) 表示共享数据只缓存在当前 CPU 缓存中，并且是被修改状态，也就是缓存的数据和主内存中的数据不一致
2. E(Exclusive) 表示缓存的独占状态，数据只缓存在当前 CPU 缓存中，并且没有被修改
3. S(Shared) 表示数据可能被多个 CPU 缓存，并且各个缓存中的数据和主内存数据一致
4. I(Invalid) 表示缓存已经失效

在 MESI 协议中，每个缓存的缓存控制器不仅知道自己的读写操作，而且也监听(snoop)其它 Cache 的读写操作

对于 MESI 协议，从 CPU 读写角度来说会遵循以下原则：

CPU 读请求：缓存处于 M、E、S 状态都可以被读取，I 状态 CPU 只能从主存中读取数据

CPU 写请求：缓存处于 M、E 状态才可以被写。对于 S 状态的写，需要将其他 CPU 中缓存行置为无效才可写



## MESI 优化带来的可见性问题

是各个 CPU 缓存行的状态是通过消息传递来进行的。如果 CPU0 要对一个在缓存中共享的变量进行写入，首先需要发送一个失效的消息给到其他缓存了该数据的 CPU。并且要等到他们的确认回执。CPU0 在这段时间内都会处于阻塞状态。

优化：在 CPU 中引入了 Store Buffer。CPU0 只需要在写入共享数据时，直接把数据写入到 store buffer 中，同时发送 invalidate 消息，然后继续去处理其他指令。当收到其他所有 CPU 发送了 invalidate acknowledge 消息时，再将 store buffer 中的数据存储至 cache line 中。最后再从缓存行同步到主内存。

新问题：

1. 数据什么时候提交是不确定的，因为需要等待其他 cpu 给回复才会进行数据同步。这里其实是一个异步操作
2. 引入了 storebufferes 后，处理器会先尝试从 storebuffer 中读取值，如果 storebuffer 中有数据，则直接从 storebuffer 中读取，否则就再从缓存行中读取

## 无法在硬件层面解决了

硬件层面无法知道软件层面数据的依赖关系

举例：

```
value = 3;
void exeToCPU0(){
    value = 10;
    isFinsh = true;
}
void exeToCPU1(){
    if(isFinsh){
        assert value == 10;
    }
}
```

exeToCPU0和exeToCPU1分别在两个独立的CPU上执行。假如 CPU0 的缓存行中缓存了 isFinish 这个共享变量，并且状态为 (E)、而 Value 可能是 (S) 状态。那么这个时候，CPU0 在执行的时候，会先把 value=10 的指令写入到 storebuffer 中。并且通知给其他缓存了该 value 变量的 CPU。在等待其他 CPU 通知结果的时候，CPU0 会继续执行 isFinish=true 这个指令。而因为当前 CPU0 缓存了 isFinish 并且是 Exclusive 状态，所以可以直接修改 isFinish=true。这个时候 CPU1 发起 read 操作去读取 isFinish 的值可能为 true，但是 value 的值不等于 10。这种情况我们可以认为是 CPU 的乱序执行，也可以认为是一种重排序，而这种重排序会带来可见性的问题。这下硬件工程师也抓狂了，我们也能理解，从硬件层面很难去知道软件层面上的这种前后依赖关系，所以没有办法通过某种手段自动去解决。

## 软件层面改造

CPU 层面提供了 memory barrier(内存屏障)的指令，从硬件层面来看这个 memroy barrier 就是 CPU flush store bufferes 中的指令。软件层面可以决定在适当的地方来插入内存屏障。

**内存屏障的作用可以通过防止 CPU 对内存的乱序访问来保证共享数据在多线程并行执行下的可见性**

JMM (java内存模型)属于语言级别的抽象内存模型，可以简单理解为对硬件模型的抽象，它定义了共享内存中多线程程序读写操作 的行为规范：在虚拟机中把共享变量存储到内存以及从内存中取出共享变量的底层实现细节 通过这些规则来规范对内存的读写操作从而保证指令的正确性，它解决了 CPU 多级缓存、处理器优化、指令重排序 导致的内存访问问题，保证了并发场景下的可见性。

注意：

JMM 并没有限制执行引擎使用处理器的寄存器或者高速缓存来提升指令执行速度，也没有限制编译器对指令进行重排序，也就是说在 JMM 中，也会存在缓存一致性问题和指令重排序问题。只是 JMM 把底层的问题抽象JVM 层面，再基于 CPU 层面提供的内存屏障指令， 以及限制编译器的重排序来解决并发问题

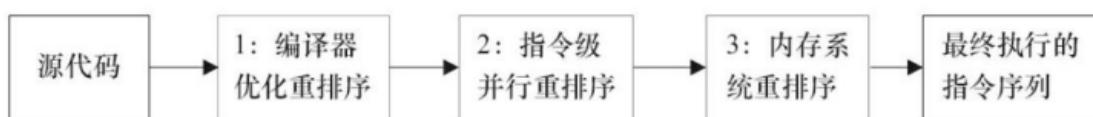
Java 内存模型底层实现可以简单的认为：通过内存屏障(memory barrier)禁止重排序，即时编译器根据具体的底层 体系架构，将这些内存屏障替换成具体的 CPU 指令。对于编译器而言，内存屏障将限制它所能做的重排序优化。而对于处理器而言，内存屏障将会导致缓存的刷新操作。比如，对于 volatile，编译器将在 volatile 字段的读写操作前后各插入一些内存屏障。

## JMM 是如何解决可见性有序性问题的

JMM 提供了一些禁用缓存以及进制重排序的方法，来解决可见性和有序性问题。这些方法大家都很熟悉： volatile、 synchronized、 final；

### JMM 如何解决顺序一致性问题

从源代码到最终执行的指令，可能会经过三种重排序。



编译器的重排序， JMM 提供了禁止特定类型的编译器重排序。

处理器重排序， JMM 会要求编译器生成指令时，会插入内存屏障来禁止处理器重排序

### JMM 层面的内存屏障

四种屏障强制规定了先后顺序(强制串行化)

为了保证内存可见性， Java 编译器在生成指令序列的适当位置会插入内存屏障来禁止特定类型的处理器的重序，

屏障类型	指令示例	备注
LoadLoad Barriers	load1 ; LoadLoad; load2	确保load1数据的装载优先于load2及所有后续装载指令的装载
StoreStore Barriers	store1; storestore;store2	确保store1数据对其他处理器可见优先于store2及所有后续存储指令的存储
LoadStore Barriers	load1;loadstore;store2	确保load1数据装载优先于store2以及后续的存储指令刷新到内存
StoreLoad Barriers	store1; storeload;load2	确保store1数据对其他处理器可见优先于load2及所有后续装载指令的装载；这条内存屏障指令是一个全能型的屏障

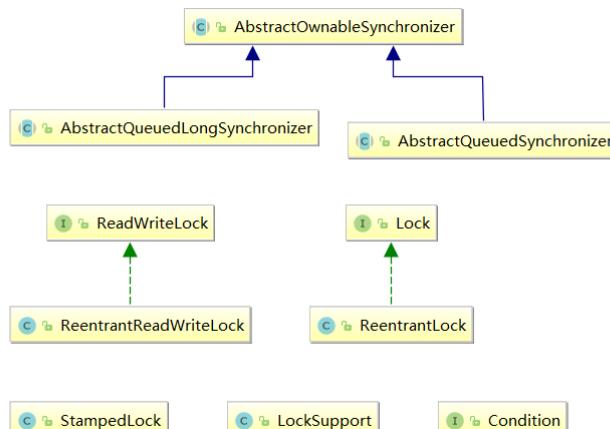
## HappenBefore

它的意思表示的是前一个操作的结果对于后续操作是可见的，所以它是一种表达多个线程之间对于内存的可见性。所以我们可以认为在 JMM 中，如果一个操作执行的结果需要对另一个操作可见，那么这两个操作必须要存在 happens-before 关系。这两个操作可以是同一个线程，也可以是不同的线程

# Java.util.concurrent并发工具包

## Lock 简介

在 Lock 接口出现之前，Java 中的应用程序对于多线程的并发安全处理只能基于 synchronized 关键字来解决。但是 synchronized 在有些场景中会存在一些短板，也就是它并不适合于所有的并发场景。但是在 Java5 以后，Lock 的出现可以解决 synchronized 在某些场景中的短板，它比 synchronized 更加灵活。  
aqs同步阻塞队列,核心方法,双向链表实现



## 常见的锁实现

ReentrantLock：表示重入锁，直接关联一次计数器增加重入次数

ReentrantReadWriteLock：重入锁的设计目的是避免线程的死锁。

ReadWriteLock 接口，维护了两个锁，一个是 ReadLock，一个是 WriteLock，他们都分别实现了 Lock 接口。读写锁是一种适合读多写少的场景下解决线程安全问题的工具

StampedLock：JDK8 引入的新的锁机制，可以简单认为是读写锁的一个改进版本，读写锁虽然通过分离读和写的功能使得读和写之间可以完全并发，但是读和写是有冲突的，如果大量的读线程存在，可能会引起写线程的饥饿。stampLock 是一种乐观的读策略，使得乐观锁完全不会阻塞写线程

## ReadWriteLock 读写锁

重入读写锁，它实现了 ReadWriteLock 接口

如果在一个整型变量上维护多种状态，就一定需要“按位切割使用”这个变量，读写锁将变量切分成了两个部分，高16位表示读，低16位表示写，划分方式如图5-8所示。

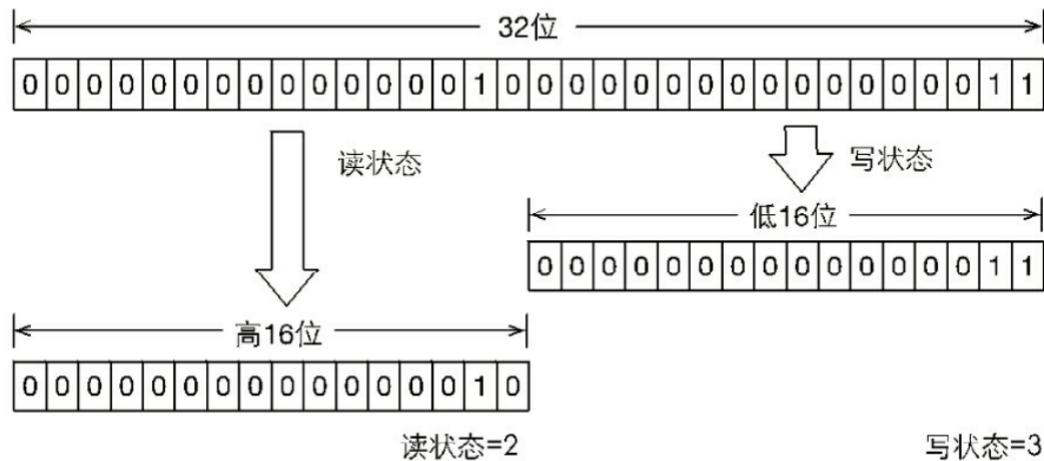


图5-8 读写锁状态的划分方式

## ReentrantLock 重入锁

重入锁，表示支持重新进入的锁，也就是说，如果当前线程 t1 通过调用 lock 方法获取了锁之后，再次调用 lock，是不会再阻塞去获取锁的，直接增加重试次数就行了。synchronized 和 ReentrantLock 都是可重入锁，**实现独占锁的功能**

```
1 //ReentrantLock 的使用案例  
2 public class AtomicDemo {
```

```
3     private static int count = 0;
4     static Lock lock = new ReentrantLock();
5
6     public static void inc() {
7         lock.lock();
8         try {
9             Thread.sleep(1);
10        } catch (InterruptedException e) {
11            e.printStackTrace();
12        }
13        count++;
14        lock.unlock();
15    }
16
17    public static void main(String[] args) throws
18        InterruptedException {
19        for (int i = 0; i < 1000; i++) {
20            new Thread(() -> {
21                AtomicDemo.inc();
22            }).start();
23            ;
24        }
25        Thread.sleep(3000);
26        System.out.println("result:" + count);
27    }
28}
29//ReentrantReadWriteLock
30public class LockDemo {
31    static Map<String, Object> cacheMap = new
HashMap<>();
32    static ReentrantReadWriteLock rwl = new
ReentrantReadWriteLock();
33    static Lock read = rwl.readLock();
34    static Lock write = rwl.writeLock();
35
36
37    public static final Object get(String key) {
38        System.out.println("开始读取数据");
39        read.lock(); //读锁
40        try {
41            return cacheMap.get(key);
42        } finally {
43            read.unlock();
44        }
45    }
46}
```

```

45     }
46
47     public static final Object put(String key,
48         Object value) {
49         write.lock();
50         System.out.println("开始写数据");
51         try {
52             return cacheMap.put(key, value);
53         } finally {
54             write.unlock();
55         }
56     }

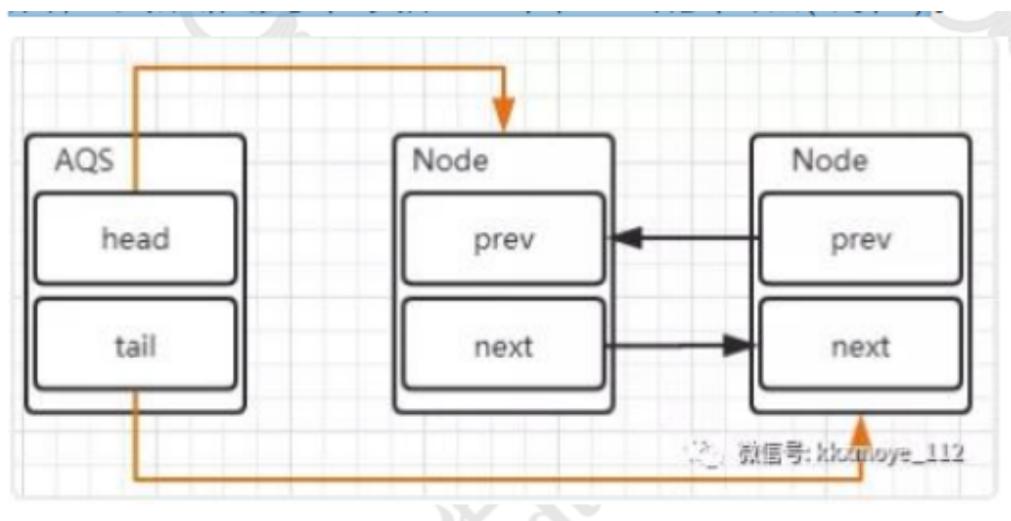
```

## ReentrantLock 的实现原理

多线程竞争重入锁时，竞争失败的线程是如何实现阻塞以及被唤醒的呢？

Lock 中，用到了一个同步队列 AQS，全称 AbstractQueuedSynchronizer  
AQS 的功能分为两种：独占和共享

AQS 队列内部维护的是一个 FIFO 的双向链表，这种结构的特点是每个数据结构都有两个指针，分别指向直接的后继节点和直接前驱节点。所以双向链表可以从任意一个节点开始很方便的访问前驱和后继。每个 Node 其实是由线程封装，当线程争抢锁失败后会封装成 Node 加入到 ASQ 队列中去；当获取锁的线程释放锁以后，会从队列中唤醒一个阻塞的节点(线程)



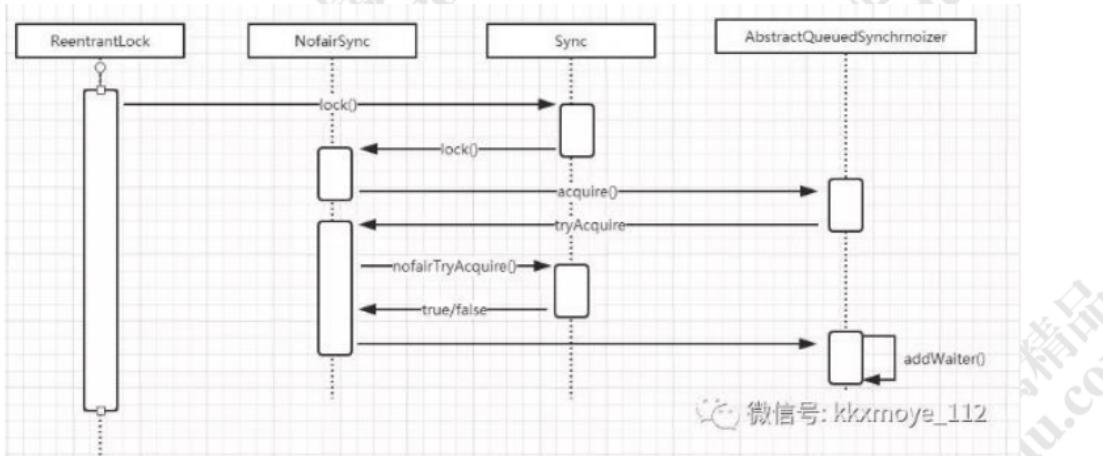
## 释放锁以及添加线程对于队列的变化

1. 新的线程封装成 Node 节点追加到同步队列中，设置 prev 节点以及修改当前节点的前置节点的 next 节点指向自己

- 通过 CAS 将 tail 重新指向新的尾部节点 head 节点表示获取锁成功的节点，当头结点在释放同步状态时，会唤醒后继节点，如果后继节点获得锁成功，会把自己设置为头结点，节点的变化过程如下

## ReentrantLock 的源码分析

调用 ReentrantLock 中的 lock() 方法，源码的调用过程我使用了时序图来展现。



获取锁的入口

这个是 reentrantLock 获取锁的入口

```

public void lock() {
    sync.lock();
}
  
```

**sync** 实际上是一个抽象的静态内部类，它继承了 AQS 来实现重入锁的逻辑，它能够实现线程的阻塞以及唤醒，但它并不具备业务功能，所以在不同的同步场景中，会继承 AQS 来实现对应场景的功能

有两个具体的实现类，分别是：

NofairSync：非公平锁表示可以存在抢占锁的功能，也就是说不管当前队列上是否存在其他线程等待，新线程都有机会抢占锁（默认方式）

FailSync：公平锁表示所有线程严格按照 FIFO 来获取锁

### NofairSync.lock

以非公平锁为例，来看看 lock 中的实现

- 非公平锁和公平锁最大的区别在于，在非公平锁中我抢占锁的逻辑是，不管有没有线程排队，我先上来 cas 去抢占一下
- CAS(compreandSet, 基于) 成功，就表示成功获得了锁
- CAS 失败，调用 acquire(1)走锁竞争逻辑

公平性锁保证了锁的获取按照FIFO原则，而代价是进行大量的线程切换。非公平性锁虽然可能造成线程“饥饿”，但极少的线程切换，保证了其**更大的吞吐量**

## CAS 乐观锁实现原理

```
protected final boolean compareAndSetState(int expect, int update) {  
    // See below for intrinsics setup to support  
    this  
    return unsafe.compareAndSwapInt(this,  
        stateOffset, expect, update);  
}
```

CAS包含3个参数：**内存值 V**    **旧的预期值 A**    **新值 B** 当且仅当V值等于A值时，将V的值改为B值，如果V值和A值不同，说明已经有其他线程做了更新，则当前线程什么都不做，最后返回当前V的真实值

通过 cas 乐观锁的方式来做**比较并替换**，这段代码的意思是，如果当前内存中的state 的值和预期值 expect 相等，则替换为 update。更新成功返回 true，否则返回 false. 这个操作是原子的，不会出现线程安全问题，这里面涉及到 Unsafe这个类的操作，以及涉及到 state 这个属性的意义。 state 是 AQS 中的一个属性，它在不同的实现中所表达的含义不一样，对于重入锁的实现来说，表示一个同步状态。它有两个含义的表示

1. 当 state=0 时，表示无锁状态
2. 当 state>0 时，表示已经有线程获得了锁，也就是 state=1，但是因为 ReentrantLock 允许重入，所以同一个线程多次获得同步锁的时候，state 会递增，比如重入 5 次，那么 state=5.而在释放锁的时候，同样需要释放 5 次直到 state=0 其他线程才有资格获得锁

JVM中的CAS操作正是利用了处理器提供的CMPXCHG指令实现的。自旋 CAS实现的基本思路就是循环进行CAS操作直到成功为止，以下代码实现了一个基于CAS线程安全的计数器 方法safeCount和一个非线程安全的计数器count

### (2) CAS实现原子操作的三大问题 及解决办法

1. ABA问题: JDK的Atomic包里提供了一个类AtomicStampedReference来解决ABA问题。加版本号

2. 循环时间长开销大, 设定次数

3. 只能保证一个变量的原子性, 将多个变脸组合使用, 。比如, 有两个共享变量 $i = 2$ ,  $j=a$ , 合并一下 $ij=2a$ , 然后用CAS来操作 $ij$

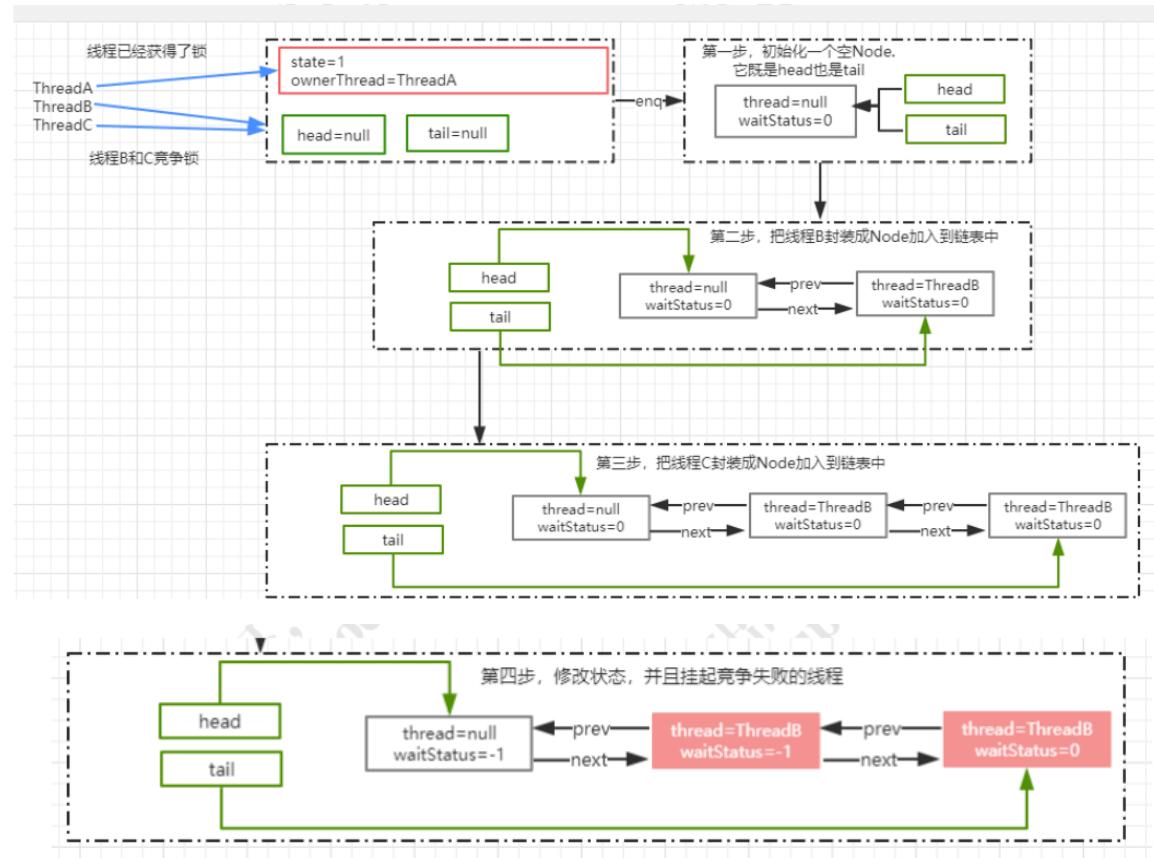
## Unsafe 类

Unsafe 类是在 sun.misc 包下, 不属于 Java 标准。但是很多 Java 的基础类库, 包括一些被广泛使用的高性能开发库都是基于 Unsafe 类开发的, 比如 Netty、Hadoop、Kafka 等; Unsafe 可认为是 Java 中留下的后门, 提供了一些低层次操作, 如直接内存访问、线程的挂起和恢复、CAS、线程同步、内存屏障

### stateOffset

stateOffset 表示 state 这个字段在 AQS 类的内存中相对于该类首地址的偏移量

## 图解分析

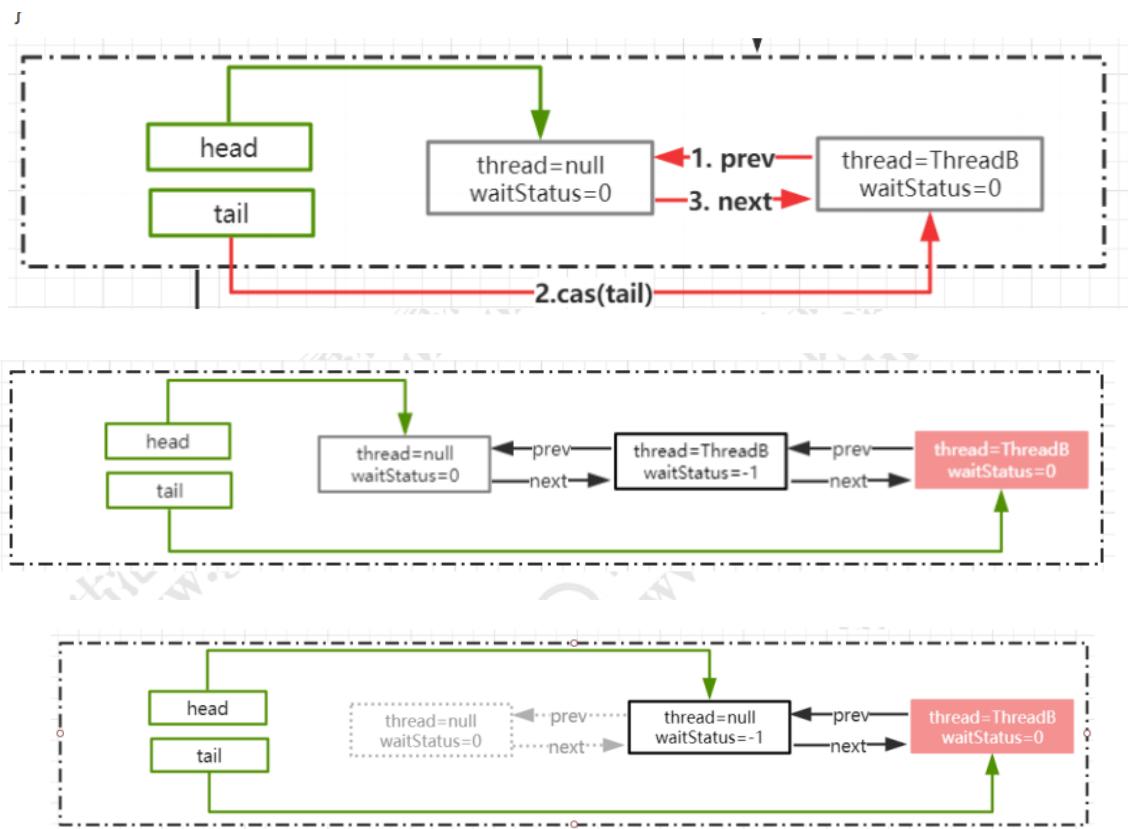


## AQS.acquireQueued

同步队列同步器,实现方式是cas自旋转

通过 addWaiter 方法把线程添加到链表后,会接着把 Node 作为参数传递给 acquireQueued 方法,去竞争锁

1. 获取当前节点的 prev 节点
2. 如果 prev 节点为 head 节点,那么它就有资格去争抢锁,调用 tryAcquire 抢占锁
3. 抢占锁成功以后,把获得锁的节点设置为 head,并且移除原来的初始化 head 节点
4. 如果获得锁失败,则根据 waitStatus 决定是否需要挂起线程
5. 最后,通过 cancelAcquire 取消获得锁的操作



## Condition

Condition 是一个多线程协调通信的工具类, **单链的等待队列**可以让某些线程一起等待某个条件 (condition), 只有满足条件时, 线程才会被唤醒

### Condition 的基本使用

```
1 public class ConditionDemoWait implements
2     Runnable {
3     private Lock lock;
4     private Condition condition;
5
6     public ConditionDemoWait(Lock lock,
```

```
7                                         Condition condition)
8 {
9     this.lock = lock;
10    this.condition = condition;
11 }
12
13 @Override
14 public void run() {
15     System.out.println("begin -
16             ConditionDemowait");
17     try {
18         lock.lock();
19         condition.await();
20         System.out.println("end -
21             ConditionDemowait");
22     } catch (InterruptedException e) {
23         e.printStackTrace();
24     } finally {
25         lock.unlock();
26     }
27 }
28 } ConditionSignal
29 public class ConditionDemoSignal implements
30     Runnable {
31     private Lock lock;
32     private Condition condition;
33
34     public ConditionDemoSignal(Lock lock,
35                                 Condition
36     condition) {
37         this.lock = lock;
38         this.condition = condition;
39     }
40
41     @Override
42     public void run() {
43         System.out.println("begin -
44             ConditionDemoSignal");
45         try {
46             lock.lock();
47             condition.signal();
48             System.out.println("end -
49                 ConditionDemoSignal");
50         }
```

```

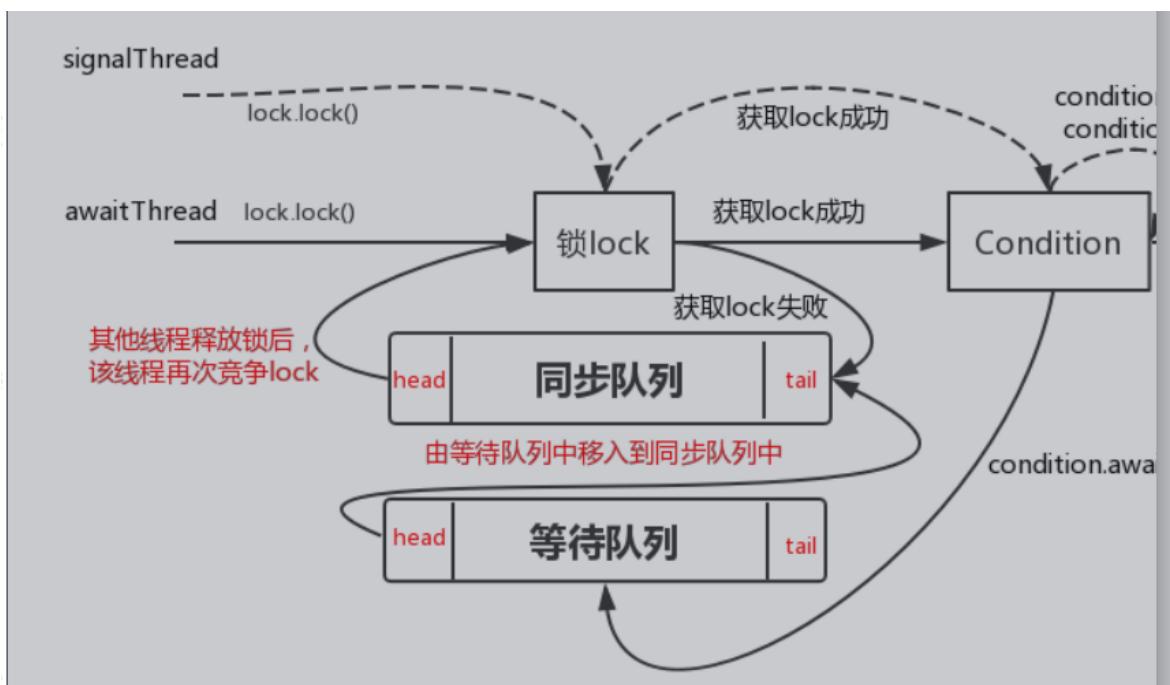
48         } finally {
49             lock.unlock();
50         }
51     }
52 }
```

condition 中两个最重要的方法，一个是 await，一个是 signal ,方法 await:把当前线程阻塞挂起 ;signal:唤醒阻塞的线程

## 对比分析

<https://www.jianshu.com/p/c330a5e7db71>

线程 awaitThread 先通过 lock.lock()方法获取锁成功后调用了 condition.await 方法进入等待队列，而另一个线程 signalThread 通过 lock.lock()方法获取锁成功后调用了 condition.signal 或者 signalAll 方法，使得线程 awaitThread 能够有机会移入到同步队列中，当其他线程释放 lock 后使得线程 awaitThread 能够有机会获取lock，从而使得线程 awaitThread 能够从 await 方法中退出执行后续操作。如果 awaitThread 获取 lock 失败会直接进入到同步队列。



等待队列相互转换,一个线程获得锁调用await方法后会被阻塞送入等待队列,艾 signal方法下被换醒去竞争锁,没抢到就会进入同步队列

**countdownlatch 是一个同步工具类**, 它允许一个或多个线程一直等待, 直到其他线程的操作执行完毕再执行

countdownlatch 提供了两个方法, 一个是 countDown, 一个是 await, countdownlatch 初始化的时候需要传入一个整数, 在这个整数倒数到 0 之

前，调用了 await 方法的程序都必须要等待，然后通过 countDown 来倒数。凡事涉及到需要指定某个人物在执行之前，要 等到前置人物执行完毕之后才执行的场景，都可以使用 CountDownLatch

```
1 public static void main(String[] args) {  
2     public static void main (String[]args) throws  
3     InterruptedException {  
4         CountDownLatch countDownLatch = new  
5             CountDownLatch(3);  
6         new Thread(() -> {  
7  
8             System.out.println("") +  
9             Thread.currentThread().g  
10                etName() + "-执行中");  
11                countDownLatch.countDown();  
12  
13                System.out.println("") +  
14                Thread.currentThread().g  
15                etName() + "-执行完毕");  
16                }, "t1").start();  
17                new Thread(() -> {  
18  
19                    System.out.println("") +  
20                    Thread.currentThread().g  
21                    etName() + "-执行中");  
22                    countDownLatch.countDown();  
23  
24                    System.out.println("") +  
25                    Thread.currentThread().g  
26                    etName() + "-执行完毕");  
27                    }, "t2").start();  
28                    new Thread(() -> {  
29  
30                        System.out.println("") +  
31                        Thread.currentThread().g  
32                        etName() + "-执行中");  
33                        countDownLatch.countDown();  
34  
35                        System.out.println("") +  
36                        Thread.currentThread().g  
37                        etName() + "-执行完毕");  
38                        }, "t3").start();  
39                        countDownLatch.await();  
40
```

```
34             System.out.println("所有线程执行完毕");
35         }
36     }
```

## Semaphore

semaphore 可以控制同时访问的线程个数，通过 acquire 获取一个许可，如果没有就等待，通过 release 释放一个许可。有点类似限流的作用。叫信号灯的原因也和他的用处有关，比如某商场就 5 个停车位，每个停车位只能停一辆车，如果这个时候来了 10 辆车，必须要等前面有空的车位才能进入。比较常见的就是用来做**限流操作了**

## CyclicBarrier

CyclicBarrier 让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续工作。

CyclicBarrier 默认的构造方法是 CyclicBarrier(int parties) 其参数表示屏障拦截的线程数量，每个线程调用 await 方法告诉 CyclicBarrier 当前线程已经到达了屏障，然后当前线程被阻塞

### 使用场景

当存在需要所有的子任务都完成时，才执行主任务，这个时候就可以选择使用 CyclicBarrier

## lock和synchronized 区别

<https://blog.csdn.net/e54332/article/details/86577071>

### 1、两者所处层面不同

synchronized 是 Java 中的一个关键字，当我们调用它时会在**虚拟机指令层面加锁**，关键字为 monitorenter 和 monitorexit

Lock 是 Java 中的一个接口，它有许多的实现类来为它提供各种功能，加锁的关键代码为大体为 Lock 和 unLock；

### 2、获锁方式

synchronized 可对实例方法、静态方法和代码块加锁，相对应的，加锁前需要获得实例对象的锁或类对象的锁或指定对象的锁。说到底就是要先获得对象的监视器（即对象的锁）然后才能够进行相关操作。

Lock的使用离不开它的实现类AQS，而它的加锁并不是针对对象的，而是针对当前线程的，并且AQS中有一个原子类state来进行加锁次数的计数

### 3、获锁失败

使用关键字synchronized加锁的程序中，获锁失败的对象会被加入到一个虚拟的等待队列中被阻塞，直到锁被释放；1.6以后加入了自旋操作

使用Lock加锁的程序中，获锁失败的线程会被自动加入到AQS的等待队列中进行自旋，自旋的同时再尝试去获取锁，等到自旋到一定次数并且获锁操作未成功，线程就会被阻塞

### 4、偏向或重入

synchronized中叫做偏向锁

当线程访问同步块时，会使用 CAS 将线程 ID 更新到锁对象的 Mark Word 中，如果更新成功则获得偏向锁，并且之后每次进入这个对象锁相关的同步块时都不需要再次获取锁了。

Lock中叫做重入锁

AQS的实现类ReentrantLock实现了重入的机制，即若线程a已经获得了锁，a再次请求锁时则会判断a是否持正有锁，然后会将原子值state+1来实现重入的计数操作

### 5、Lock独有的队列

condition队列是AQS中的一个Lock的子接口的内部现类，它一般会和ReentrantLock一起使用来满足除了加锁和解锁以外的一些附加条件，比如对线程的分组和临界数量的判断（阻塞队列）

### 6、解锁操作

synchronized：不能指定解锁操作，执行完代码块的对象会自动释放锁

Lock：可调用unlock方法去释放锁比synchronized更灵活

## ConcurrentHashMap

<https://www.jianshu.com/p/78989cd553b4>

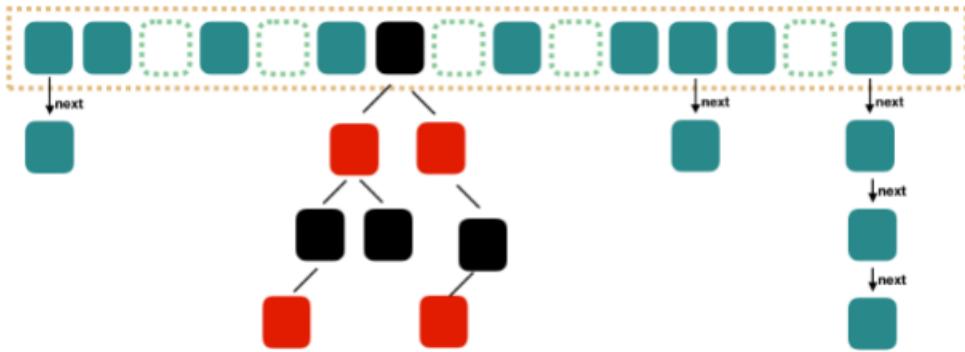
<https://yuanrengu.com/2020/ba184259.html>

在并发编程中使用HashMap可能导致程序死循环。而使用线程安全的HashTable效率又非常低下，基于以上两个原因，便有了ConcurrentHashMap的登场机会

## JDK1.8新变化

1.取消Segment分段操作,该向Node节点操作.Segment继承 ReentrantLock 来进行加锁

2.将原本数组+单向链表的数据结构变更为了数组+单向链表+红黑树的结构。



## 源码细节

增长因子默认0.75, 初始化16个节点, 在节点大于64切链表长度大于8会用红黑树取代链表, Node是ConcurrentHashMap存储结构的基本单元, 继承于HashMap中的Entry 并发控制使用Synchronized和CAS来操作.

### Synchronized加锁 在Node节点

## put过程

- 1.如果没有初始化就先调用initTable () 方法来进行初始化过程
- 2.如果没有hash冲突就直接CAS插入
- 3.如果还在进行扩容操作就先进行扩容,调用 helpTransfer () 多线程一起扩容
- 4.如果存在hash冲突, 就加锁来保证线程安全, 这里有两种情况, 一种是链表形式就直接遍历到尾端插入, 一种是红黑树就按照红黑树结构插入,
- 5.最后一个如果Hash冲突时会形成Node链表, 在链表长度超过8, Node数组超过64时会将链表结构转换为红黑树的结构, break再一次进入循环, treeifyBin () 方法进行链表转红黑树的过程
- 6.如果添加成功就调用addCount () 方法统计size, 现在调用addCount(int,int)方法计算ConcurrentHashMap的size, **在原来的基础上加一(分片处理)**

他在并发处理中使用的是乐观锁, 当有冲突的时候才进行并发处理

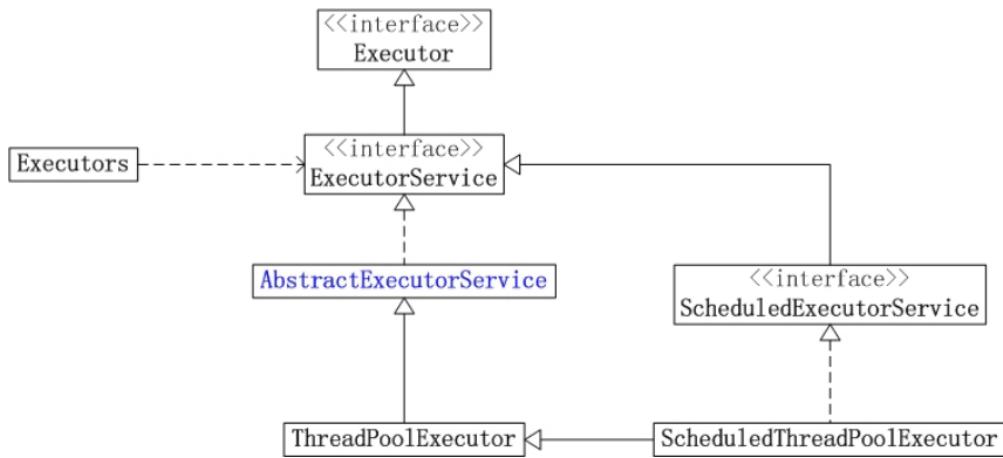
## get操作

- 1.计算hash值, 定位到该table索引位置, 如果是首节点符合就返回
- 2.如果遇到扩容的时候, 会调用标志正在扩容节点ForwardingNode的find方法, 查找该节点, 匹配就返回
- 3.以上都不符合的话, 就往下遍历节点, 匹配就返回, 否则最后就返回null

HashMap在put和get元素的时候直接取key的hashCode然后经过再次均衡后直接采用&位运算(是取模27倍)就能达到取模效果

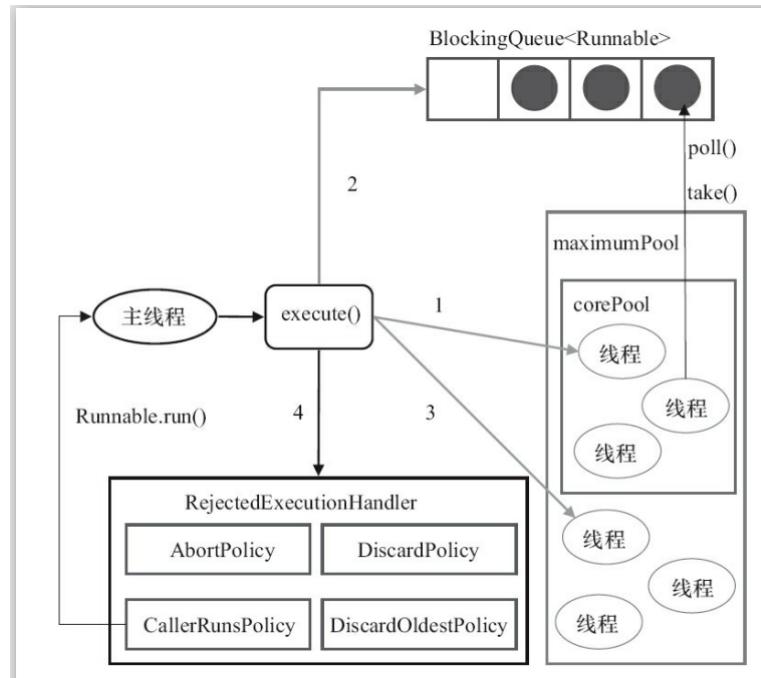
# 线程池

## 线程池架构图



解决现存创建等性能消耗

核心建议使用`ThreadPoolExecutor`类



- 1) 如果当前运行的线程少于(cpu核数)`corePoolSize`，则创建新线程来执行任务（注意，执行这一步骤需要获取全局锁）。
- 2) 如果运行的线程等于或多于`corePoolSize`，则将任务加入`BlockingQueue`，如果无法将任务加入`BlockingQueue`（队列已满），则创建新的线程来处理任务（注意，执行这一步骤需要获取全局锁）。

4) 如果创建新线程将使当前运行的线程超出maximumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法。

ThreadPoolExecutor采取上述步骤的总体设计思路，是为了在执行execute()方法时，尽可能地避免获取全局锁（那将会是一个严重的可伸缩瓶颈）。在ThreadPoolExecutor完成预热之后（当前运行的线程数大于等于corePoolSize），几乎所有的execute()方法调用都是执行步骤2，而步骤2不需要获取全局锁

## 阻塞队列原子操作分析

<https://ifeve.com/java-blocking-queue/>

BlockingQueue阻塞队列接口，

java提供了7个实现类ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列。

LinkedBlockingQueue：一个由链表结构组成的有界阻塞队列。

PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列。

DelayQueue：一个使用优先级队列实现的无界阻塞队列。

SynchronousQueue：一个不存储元素的阻塞队列。

LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列

阻塞队列提供了四种处理方法：

方法\处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除方法	remove()	poll()	take()	poll(time,unit)
检查方法	element()	peek()	不可用	不可用

1. 插入操作      add(e) : 添加元素到队列中，如果队列满了，继续插入元素会报错，IllegalStateException。

offer(e) : 添加元素到队列，同时会返回元素是否插入成功的状态，如果成功则返回 true  
put(e) : 当阻塞队列满了以后，生产者继续通过 put添加元素，队列会一直阻塞生产者线程，知道队列可用

offer(e,time,unit) : 当阻塞队列满了以后继续添加元素，生产者线程会被阻塞指定时间，如果超时，则线程直接退出

## 2. 移除操作

remove() : 当队列为空时，调用 remove 会返回 false，如果元素移除成功，则返回 true

poll() : 当队列中存在元素，则从队列中取出一个元素，如果队列为空，则直接返回 null

take() : 基于阻塞的方式获取队列中的元素，如果队列为空，则 take 方法会一直阻塞，直到队列中有新的数据可以消费

poll(time,unit) : 带超时机制的获取数据，如果队列为空，则会等待指定的时间再去获取元素返回

## ArrayBlockingQueue

作用：用在生产者消费者模型中，fifo 的队列

## 原子操作类

原子操作可以理解为：一个数，很多线程去同时修改它，不加sync同步锁，就可以保证修改结果是正确的

**Atomic正是采用了CAS算法，所以可以在多线程环境下安全地操作对象。**

volatile是Java的关键字，官方解释：volatile可以保证可见性、顺序性、一致性。

可见性：volatile修饰的对象在加载时会告知VM，对象在CPU的缓存上对多个线程是同时可见的。

顺序性：这里有JVM的内存屏障的概念，简单理解为：可以保证线程操作对象时是顺序执行的，详细了解可以自行查阅。

一致性：可以保证多个线程读取数据时，读取到的数据是最新的。（注意读取的是最新的数据，但不保证写回时不会覆盖其他线程修改的结果）

-----问题高低位，回去看视频

# 分布式架构

## 架构的本质及其演变

一个软件系统随着功能越来越多，调用量急剧增长，整个系统逐渐碎片化，越来越无序，最终无法维护和扩展，所以系统在一段时间的野蛮生长后，也需要及时干预，避免越来越无序。**架构的本质就是对系统进行有序化重构，使系统不断进化**.那架构是如何实现无序到有序的呢？**基本的手段就是分和合，先把系统打散，然后重新组合。**分的过程是把系统拆分为各个子系统 / 模块 / 组件，拆的时候，首先要解决每个组件的定位问题，然后才能划分彼此的边界，实现合理的拆分。**合就是根据最终要求，把各个分离的组件有机整合在一起，相对来说，第一步的拆分更难。**拆分的结果使开发人员能够做到业务聚焦、技能聚焦，实现开发敏捷，合的结果是系统变得柔性，可以因需而变，实现业务敏捷.

### 架构的分类

架构一般可分业务架构、应用架构、技术架构

1. 业务架构从概念层面帮助开发人员更好的理解系统，比如业务流程、业务模块、输入输出、业务域
2. 应用架构从逻辑层面帮助开发落地系统，如数据交互关系、应用形式、交互方式，是的整个系统逻辑上更容易理解，步入大家熟知的 SOA 就属于应用架构的范畴
3. 技术架构主要解决技术平台选型、如操作系统、中间件、设备、多机房、水平扩展、高可用等问题需要注意的是，系统或者架构首先都是为人服务的，系统的有序度高，用用逻辑合理，业务概念清晰是第一位。现在大家讨论更多的是技术架构，如高并发设计，分布式事务处理等，只是因为这个不需要业务上下文背景，比较好相互沟通。具体架构设计时，首先要关注业务架构和应用架构，这个架构新手要特别注意。也是面试时候的痛点！

垂直伸缩：表示通过升级或者增加单台机器的硬件来支撑访问量以及数据量增长的方式，垂直伸缩的好处在于技术难度比较低，运营和改动成本也相对较低。但是缺点是机器性能是有瓶颈的，同时升级高性能的小型机或者大型机，成本是非常大的

### 分布式架构的定义

分布式系统是指位于网络计算机上的组件仅通过传递消息来通信和协调目标系统。这里面有两个重要因素

1. 组件是分布在网络计算机上
2. 组件之间**仅仅通过消息传递来通信并协调行动**分布式系统其实也可以认为是一种**去中心化**的实现思路，对于用户来说是无感知的

## 分布式架构的意义

从单机单用户到单机多用户，再到底现在的网络时代，应用系统发生了很多的变化，为什么要用分布式系统呢？

1. 升级单机处理能力的性价比越来越低
2. 单机处理能力存在瓶颈
3. 对于稳定性和可用性的要求

演变流程：

单体架构-->集群-->soa>微服务

单体架构：所有都在一台机器上，容量有限，

集群：水平拆分，

session问题，

解决办法，复制session，访问数量更大，集中处理，使用Redis处理

数据库跟着拆分，读写分离，根据业务来，然后水平拆分，同一张表分在不同的库

微服务：重点解决服务直接的调用，维护，通信

影响性能的因素：性

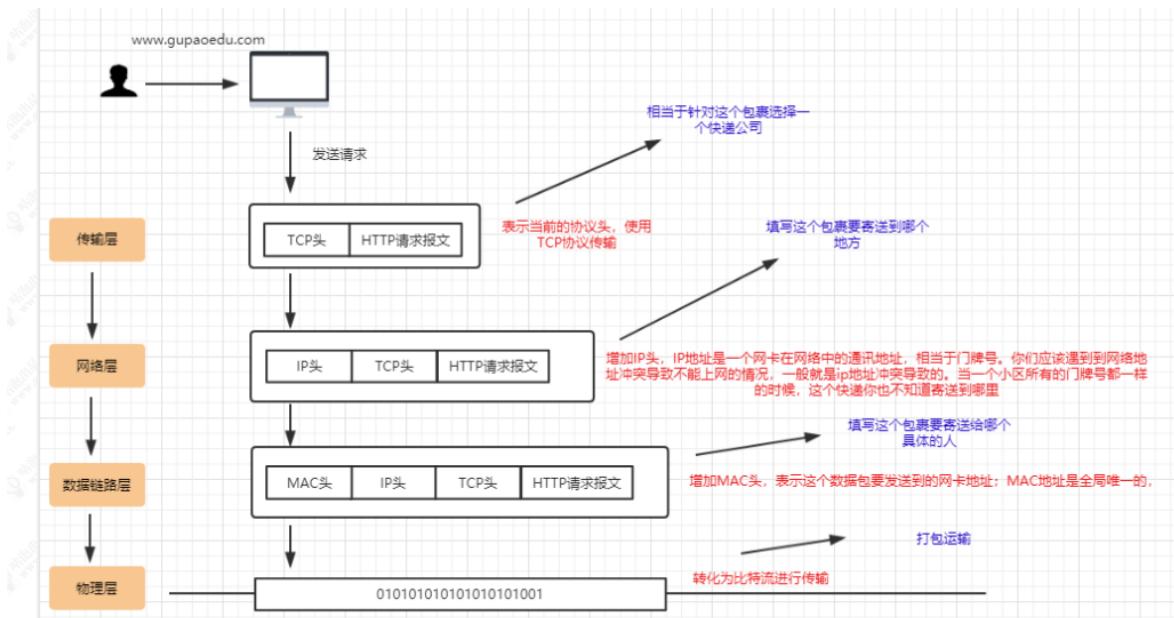
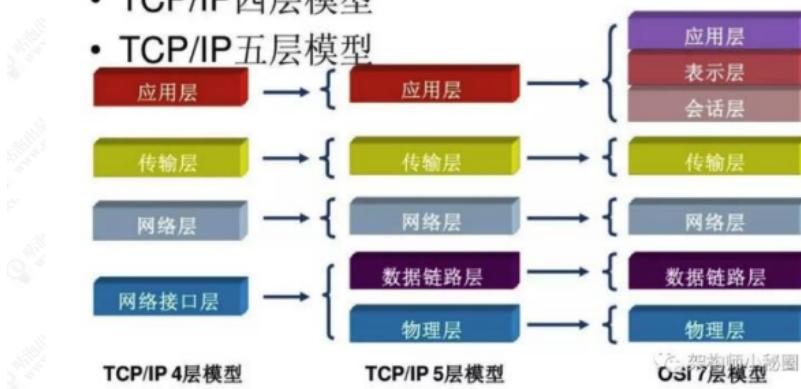
**能瓶颈只会是：**CPU、文件IO、网络IO、内存、等因素。

## 网络通信原理

物理层-->数据链路层-->网络层-->传输层-->**会话层**-->**表示层**-->**应用层**(统称应用层)

## 分层模型

- OSI七层模型
- TCP/IP四层模型
- TCP/IP五层模型

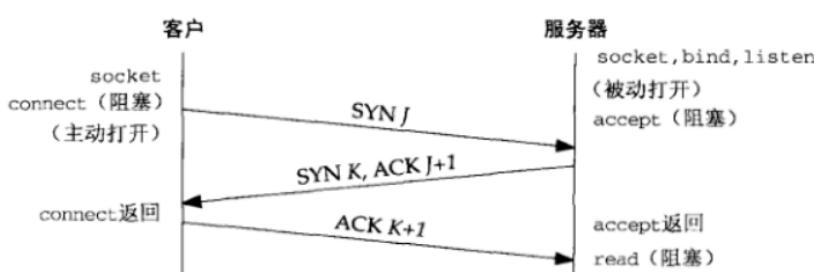


TCp/Ip分层管理:

应用层, 传输层, 网络层, 数据链路层

确认连接, 三次握手; 断开连接: 四次挥手

TCP三次握手如图:



ACK (Acknowledge character) 即是确认字符

Java 应用程序中如何使用 **socket 套接字**来建立一个基于 tcp 协议的通信流程

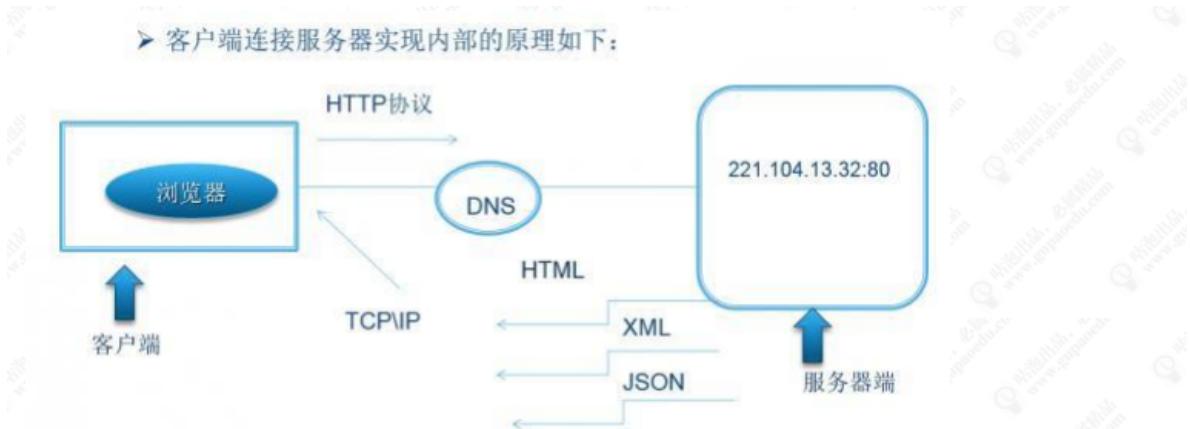
每个 TCP Socket 的内核中都有一个发送缓冲区和一个接收缓冲区，TCP 的全双工的工作模式及 TCP 的滑动窗口就是依赖于这两个独立的 Buffer 和该 Buffer 的填充状态

阻塞:通过 socket.accept 去接收一个客户端请求, accept 是一个阻塞的方法, 意味着 TCP 服务器一次 只能处理一个客户端请求, 当一个客户端向一个已经被其他客户端占用的服务器发送连接请求时, 虽然在连接建立后可以向服务端发送数据, 但是在服务端处理完之前的请求之前, 却不会对新的客户端做出响应, 这种类型的服务器称为“迭代服务器”。迭代服务器是按照顺序处理客户端请求, 也就是服务端必须要处理完前一个请求才能对下一个客户端的请求进行响应.

通过java多线程解决,线程池技术

## Http协议原理

基于tcp/ip的应用层协议



**URL统一资源定位符 ip:端口/路径**

### 常见错误码

	类别	原因短语
1XX	Informational (信息性状态码)	接收的请求正在处理
2XX	Success (成功状态码)	请求正常处理完毕
3XX	Redirection (重定向状态码)	需要进行附加操作以完成请求
4XX	Client Error (客户端错误状态码)	服务器无法处理请求
5XX	Server Error (服务器错误状态码)	服务器处理请求出错

GET：一般是用于客户端发送一个 URI 地址去获取服务端的资源（一般用于查询操作），Get不支持的传输数据有限制，具体限制由浏览器决定

POST：一般用户客户端传输一个实体给到服务端，让服务端去保存（一般用于创建操作）

PUT：向服务器发送数据，一般用于更新数据的操作

DELETE：客户端发起一个 Delete 请求要求服务端把某个数据删除（一般用于删除操作）

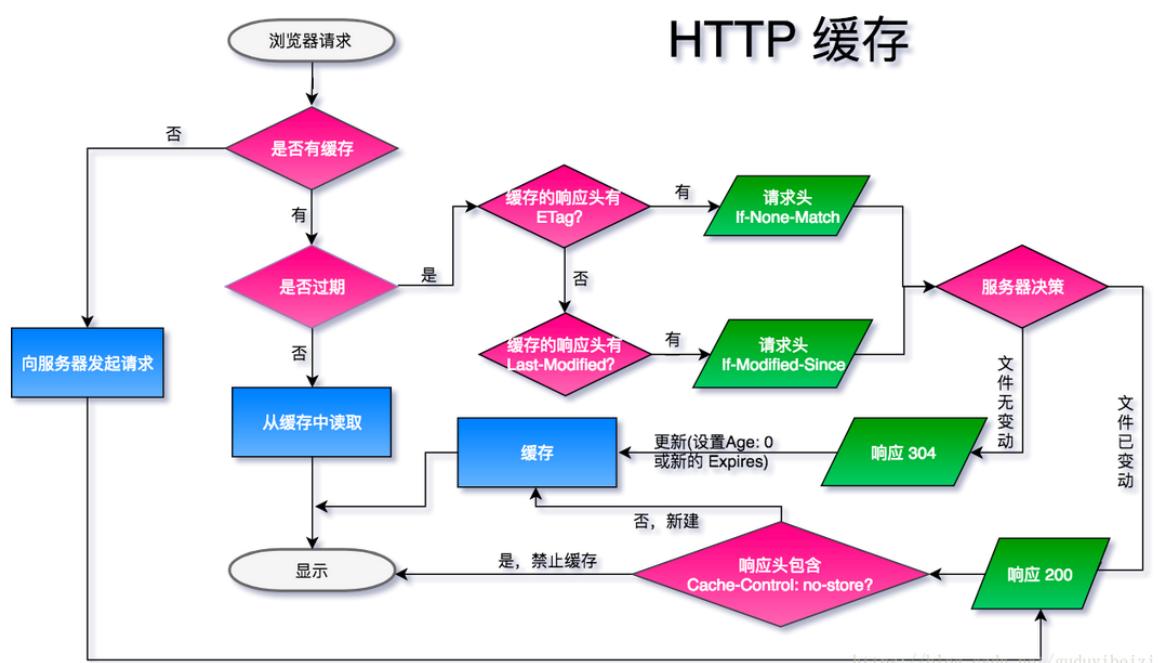
HEAD:获得报文首部、

OPTIONS：询问支持的方法、

TRACE：追踪路径、

CONNECT：用隧道协议连接代理

## HTTP 缓存机制详解



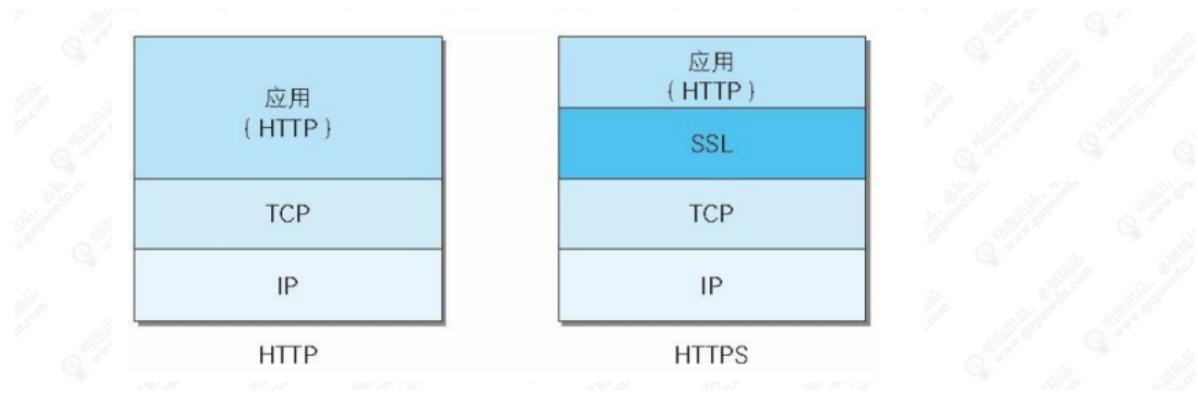
<https://blog.csdn.net/guduyibeizi/article/details/81814577>

详细格式参考：<https://www.cnblogs.com/breka/articles/9791664.html>

## Http 无状态协议

HTTP 协议是无状态的，什么是无状态呢？就是说 HTTP 协议本身不会对请求和响应之间的通信状态做保存

每个请求都是完全独立的，每个请求包含了处理这个请求所需的数据



## 序列化序列化

java对象在JVM虚拟机关闭后以字节形式保存,供其在网络之间传输.对象实现 **Serializable** 接口,会设置 **serialVersionUID** 作为唯一标识,若序列化文件和java类id不一致会报错;Transient 关键字在变量声明前加上该关键字, 避免序列化,0.null

### 分布式架构下常见序列化技术

java内置序列化存在的问题.

- 1.序列化的数据比较大, 传输效率低
- 2.其他语言无法识别和对接

常见序列化工具:**XML 序列化框架介绍**,webservice;XML 序列化的好处在于**可读性好, 方便阅读和调试**。但是序列化以后的字节码文件比较大, 而且效率不高, **适用于对性能不高, 而且 QPS 较低的企业级内部系统之间的数据交换的场景**, 同时 XML 又具有语言无关性, 所以还可以用于异构系统之间的数据交换和协议

**JSON 序列化框架**,Jackson , 阿里开源的 FastJson, Google 的 GSON;Jackson 与 fastjson 要比 GSON 的性能要好, 但是 Jackson、 GSON 的稳定性要比 Fastjson 好。而 fastjson 的优势在于提供的 api 非常容易使用

### Hessian 序列化框架:

一个支持跨语言传输的二进制序列化协议;Dubbo 采用的就是 Hessian 序列化来实现, 只不过 Dubbo 对 Hessian 进行了重构, 性能更高

### Protobuf 序列化的原理

Protocol Buffer 的性能好, 主要体现在 序列化后的数据体积小 & 序列化速度快, 最终使得 传输效率高, 其原因如下:

序列化速度快的原因：

- a. 编码 / 解码 方式简单 (只需要简单的数学运算 = 位移等等)
- b. 采用 Protocol Buffer 自身的框架代码 和 编译器 共同完成 序列化后的数据量体积小 (即数据压缩效果好) 的原因：

- a. 采用了独特的编码方式，如 Varint、Zigzag 编码方式等等
- b. 采用 T - L - V 的数据存储方式：减少了分隔符的使用 & 数据存储得紧凑

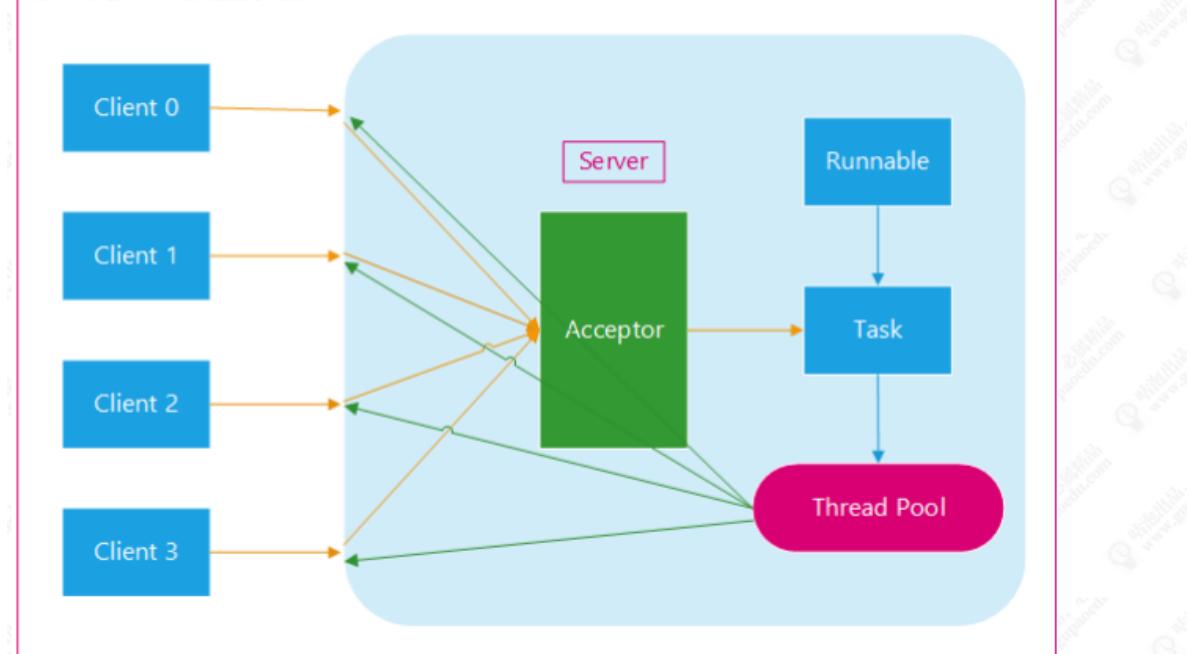
### 选型建议

1. 对性能要求不高的场景，可以采用基于 XML 的 SOAP(xml(xsd)+http) 协议 ,webservice 使用了此协议()
2. 对性能和间接性有比较高要求的场景，那么 Hessian、Protobuf、Thrift(Facebook 使用)、Avro 都可以。
3. 基于前后端分离，或者独立的对外的 api 服务，选用 JSON 是比较好的，对于调试、可读性都很不错
4. Avro 设计理念偏于动态类型语言，那么这类的场景使用 Avro 是可以的各个序列化技术的性能比较

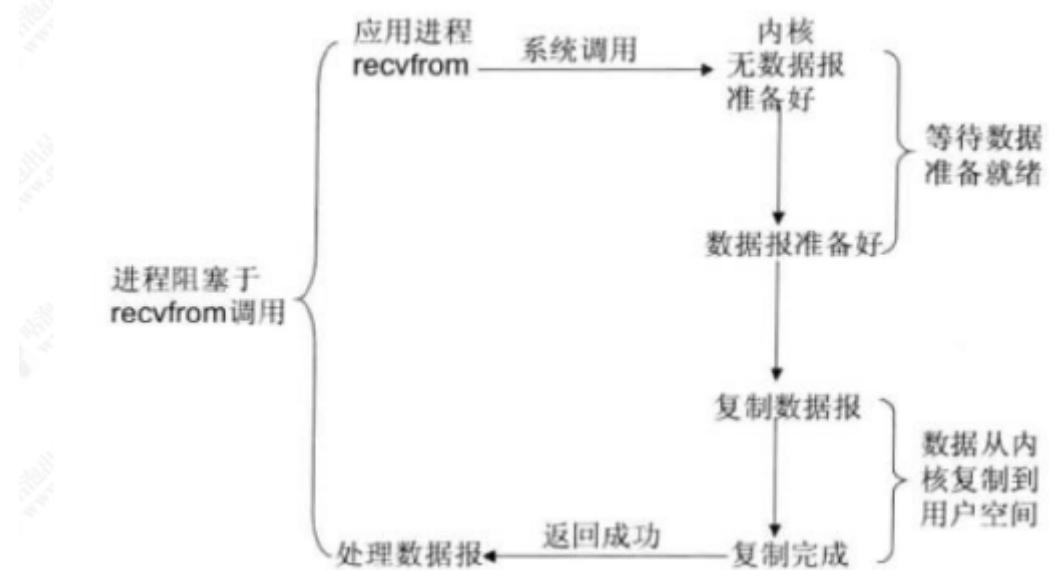
这个地址有针对不同序列化技术进行性能比较：<https://github.com/eishay/jvmserializers/wiki>

## IO模型

## 伪异步IO模型图

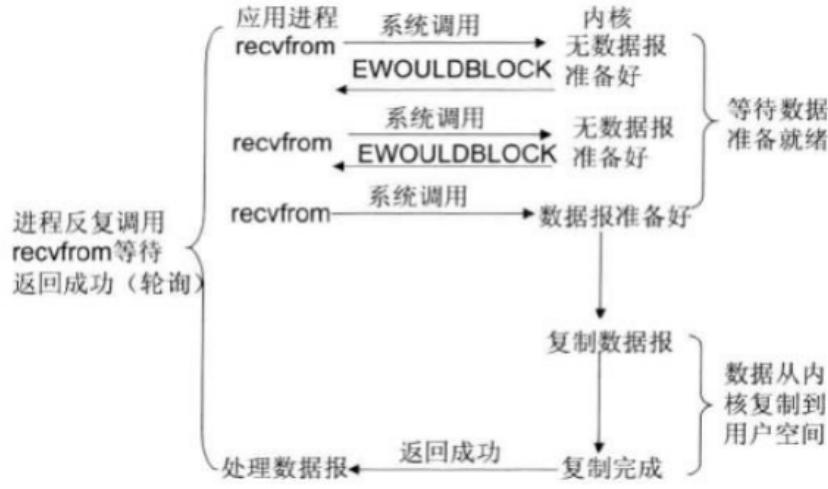


## 阻塞IO



## 非阻塞IO

nio



## I/O 复用模型

前面异步IO还是通过轮询查看状态,先通过一个线程监听所有文件描述符;常见的 IO 多路复用方式有【select、poll、epoll】

**select 本质还是轮询**,支持线程有限默认1024,开的越多监听越麻烦

**epoll事件驱动**,数据准备就绪会通知当前进程具体哪个线程的id准备好了

**阻塞**:往往需要**等待缓冲区中的数据准备好**过后才处理其他的事情,否则一直等待在那里。

**非阻塞**:当我们的进程访问我们的数据缓冲区的时候,如果数据没有准备好则**直接返回,不会等待**。如果数据已经准备好了,也直接返回

**同步**:是**应用程序要直接参与 I/O 读写的操作**。

**异步**:所有的**I/O 读写交给操作系统去处理**,应用程序只需要等待通知。

同步方式在处理 I/O 事件的时候,必须阻塞在某个方法上面等待我们的 I/O 事件完成(阻塞 I/O 事件或者通过轮询 I/O 事件的方式),对于异步来说,所有的 I/O 读写都交给了操作系统。这个时候,我们可以去做其他的事情,并不需要去完成真正的 I/O 操作,当操作完成 I/O 后,会给我们的应用程序一个通知。

**同步**:阻塞到 I/O 事件,阻塞到 read 或则 write。这个时候我们就完全不能做自己的事情。让读写方法加入到线程里面,然后阻塞线程来实现,对线程的性能开销比较大

属性	同步阻塞 IO(BIO)	伪异步 IO	非阻塞 IO ( NIO )	异步 IO(AIO)
客户端数:IO 线程数	1:1	M:N(M>=N)	M:1	M:0
阻塞类型	阻塞	阻塞	非阻塞	非阻塞
同步	同步	同步	同步(多路复用)	异步
API 使用难度	简单	简单	复杂	一般
调试难度	简单	简单	复杂	复杂

## 分布式常用算法

### 一致性hash算法

一致性hash算法是分布式中一个常用且好用的**分片算法、或者数据库分库分表算法**。避免单点故障、提升处理效率、横向扩展等原因，分布式系统已经成为居家旅行必备的部署模式，所以也产出了几种数据分片的方法：

1.取模，2.划段，3.一致性hash

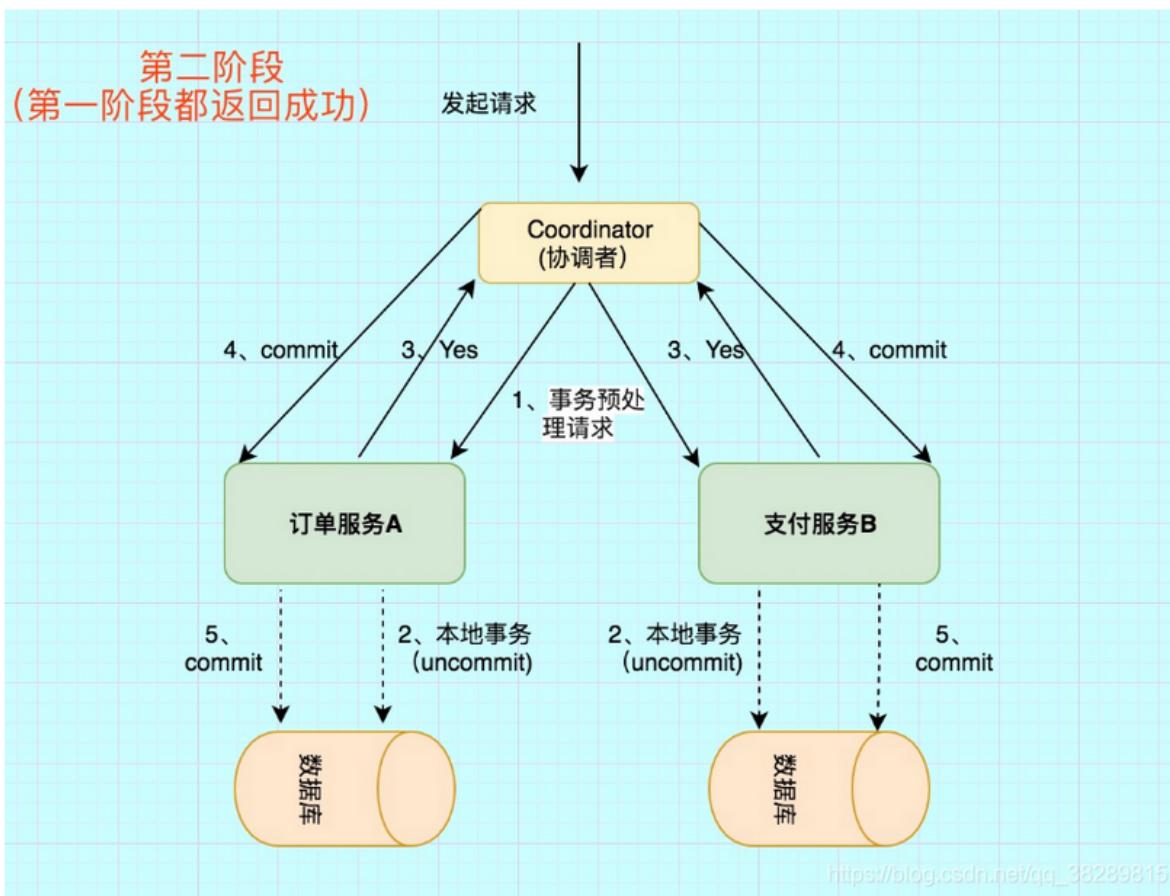
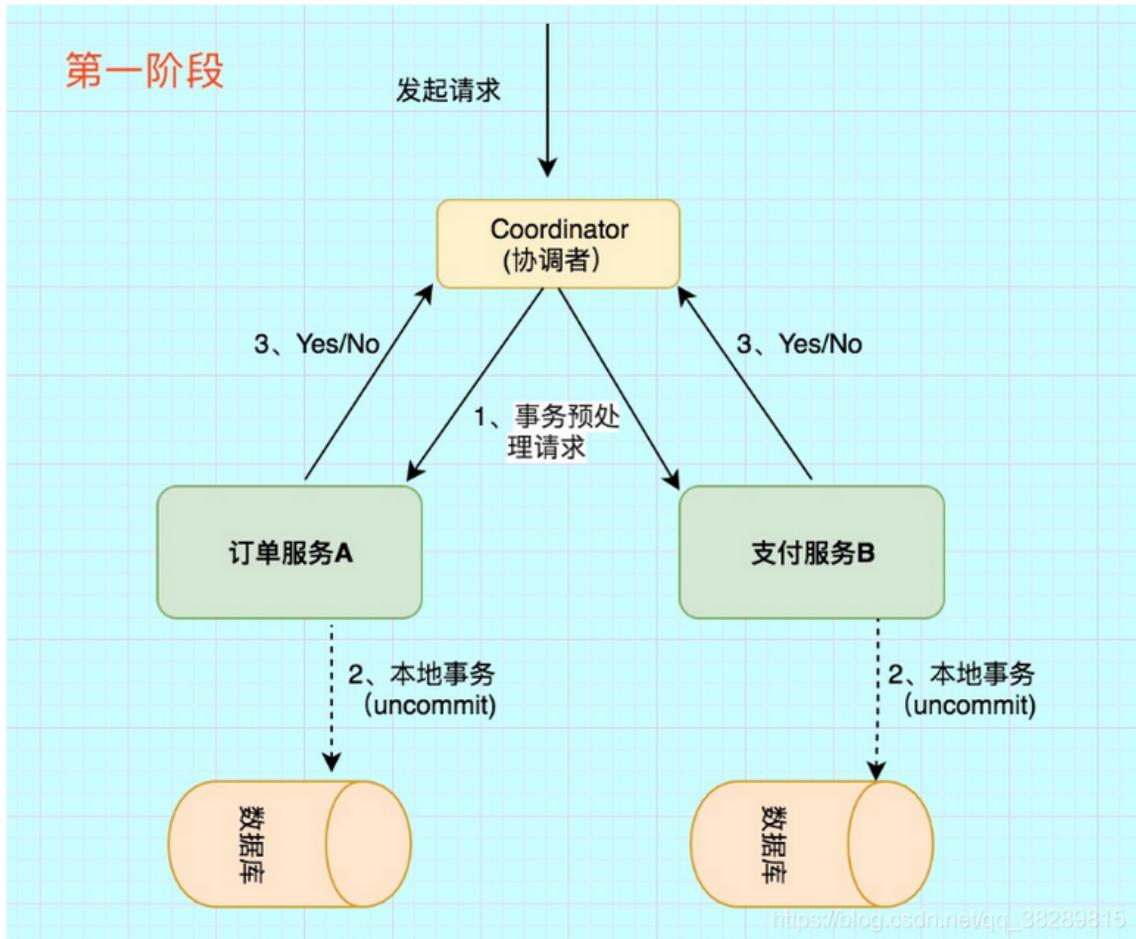
前两种需要固定的节点数，即节点数不能变，不能某一个节点挂了或者实时增加一个节点，变了分片规则就需要改变，需要迁移的数据也多。

那么一致性hash是怎么解决这个问题的呢？

一致性hash：对节点和数据，都做一次hash运算，然后比较节点和数据的hash值，数据值和节点最相近的节点作为处理节点。为了分布得更均匀，通过使用**虚拟节点**的方式，每个节点计算出n个hash值，均匀地放在hash环上这样数据就能比较均匀地分布到每个节点。可以动态增加删除节点而不影响其他节点

<https://blog.csdn.net/u011305680/article/details/79721030>

**cap,推出BASE理论,基本可用,软状态,最终一致性,只无法实现强一致性的,使用2pc提交**



优点:原理简单易实现,

缺点:单点问题,脑裂;性能问题,系统阻塞状态,

无论是在第一阶段的过程中，还是在第二阶段，**所有的参与者资源和协调者资源都是被锁住的**

## 2PC 可能面临的故障

### (1) 协调者正常，参与者宕机

由于协调者无法收集到所有参与者的反馈，会陷入阻塞情况。

解决方案：引入超时机制，如果协调者在超过指定的时间还没有收到参与者的反馈，事务就失败，向所有节点发送终止事务请求。

### (2) 协调者宕机，参与者正常

无论处于哪个阶段，由于协调者宕机，无法发送提交请求，所有处于执行了操作但是未提交状态的参与者都会陷入阻塞情况。

解决方案：引入协调者备份，同时协调者需记录操作日志。当检测到协调者宕机一段时间后，协调者备份取代协调者，并读取操作日志，向所有参与者询问状态。

### (3) 协调者和参与者都宕机

①发生在第一阶段：因为第一阶段，所有参与者都没有真正执行 Commit，所以只需重新在剩余的参与者中重新选出一个协调者，新的协调者在重新执行第一阶段和第二阶段就可以了。

②发生在第二阶段 并且 挂了的参与者在挂掉之前没有收到协调者的指令。也就是上面图片中的第 4 步挂了，这是可能协调者还没有发送第 4 步就挂了。这种情形下，新的协调者重新执行第一阶段和第二阶段操作。

**无法解决:**③发生在第二阶段 并且 有部分参与者已经执行完 commit 操作。就好比这里 订单服务A 和 支付服务B 都收到协调者发送的 Commit 信息，开始真正执行本地事务 Commit，但突发情况，A Commit 成功，B 却挂了。这个时候目前来讲数据是不一致的。虽然这个时候可以再通过手段让他和协调者通信，再想办法把数据搞成一致的，但是，这段时间内他的数据状态已经是不一致的！

3pc

**引入超时机制，同时在 协调者 和 参与者 中都引入超时机制。**

通过 CanCommit、PreCommit、DoCommit 三个阶段的设计，相较于 2PC 而言，多设置了一个缓冲阶段保证了在最后提交阶段之前各参与节点的状态是一致的。以上就是 3PC 相对于 2PC 的一个提高（相对缓解了 2PC 中的前两个问题），但是 3PC 依然没有完全解决数据不一致的问题。

Cheap Paxos 算法、

Paxos 中存在三种角色 Proposer（提议者）、Acceptor（决策者）、Learner（议案学习者），整个过程（一个实例或称一个事务或一个 Round）分为两个段；

### **phase1 (准备阶段)**

\1. Proposer(提议者)向超过半数 ( $n/2+1$ ) Acceptor 发起 prepare 消息(发送编号)

\2. 如果 prepare 符合协议规则 Acceptor 回复 promise 消息，否则拒绝

### **phase2 (决议阶段或投票阶段)**

\1. 如果超过半数 Acceptor 回复 promise，Proposer 向 Acceptor 发送 accept 消息

\2. Acceptor 检查 accept 消息是否符合规则，消息符合则批准 accept 请求

Raft 算法、ZAB 协议等等。

## **Netty 与 NIO 之前世今生**

1、掌握 NIO 的核心组件 **Buffer、Selector、Channel**。

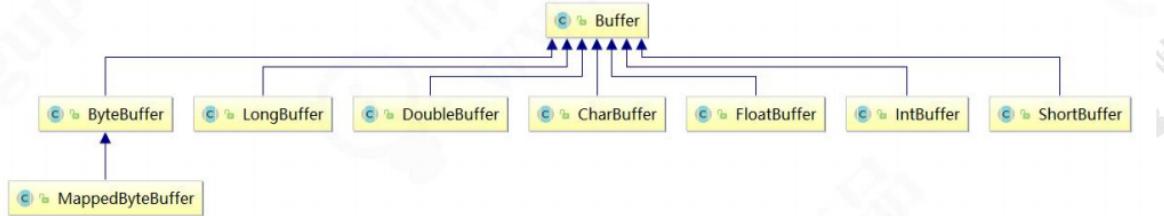
connect 连接内外部, Buffer 读写数据, selector, 事件驱动

2、何谓多路复用？

3、Netty 支持的功能与特性。

### **buffer**

缓冲区实际上是一个容器对象，更直接的说，其实就是一个数组，在 NIO 库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的；在写入数据时，它也是写入到缓冲区中的；任何时候访问 NIO 中的数据，都是将它放到缓冲区中。而在面向流 I/O 系统中，所有数据都是直接写入或者直接将数据读取到 Stream 对象中。在 NIO 中，所有的缓冲区类型都继承于抽象类 Buffer，最常用的就是 ByteBuffer，对于 Java 中的基本类型，基本都有一个具体 Buffer 类型与之相对应，它们之间的继承关系如下图所示



`ByteBuffer.allocate(10);`指定缓冲区数据组大小

`buffer.slice();`分片创建**子缓冲区**,`buffer.asReadOnlyBuffer();`只读缓冲区数据和原缓冲区数据共享

`ByteBuffer.allocateDirect(1024);`**直接缓冲区**;直接缓冲区是为**加快 I/O 速度**, 使用一种特殊方式为其分配内存的缓冲区, JDK 文档中的描述为: 给定一个直接字节缓冲区, Java 虚拟机将尽最大努力直接对它执行本机 I/O 操作也就是说, 它会在每一次调用底层操作系统的本机 I/O 操作之前(或之后), 尝试**避免将缓冲区的内容拷贝到一个中间缓冲区中或者从一个中间缓冲区中拷贝数据**

**MappedByteBuffer 内存映射文件** 不需要将数据读取到OS内核缓冲区, 而是直接将进程的用户私有地址空间中的一部分区域与文件对象建立起映射关系, 就好像直接从内存中读、写文件一样

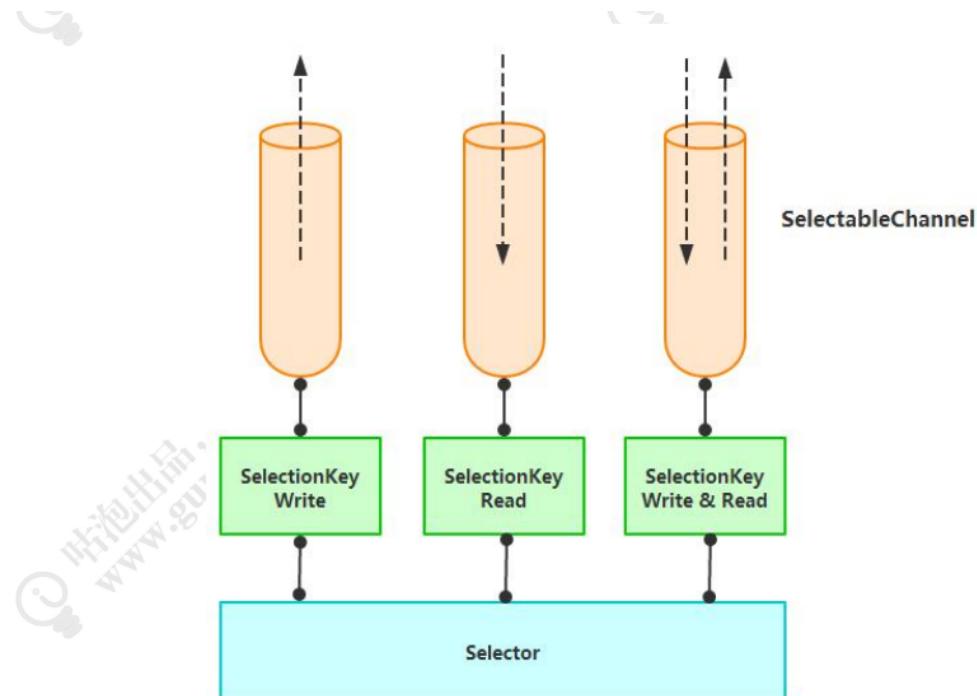
ps:直接缓冲区是指物理内存,非直接缓冲区是jvm内存

```

1 public class MappedBuffer {
2     static private final int start = 0;
3     static private final int size = 1024;
4     static public void main( String args[] ) throws
Exception {
5         RandomAccessFile raf = new RandomAccessFile(
"E://test.txt", "rw" );
6         FileChannel fc = raf.getChannel();
7         //把缓冲区跟文件系统进行一个映射关联
8         //只要操作缓冲区里面的内容, 文件内容也会跟着改变
9         MappedByteBuffer mbb = fc.map(
FileChannel.MapMode.READ_WRITE, start, size );
10        mbb.put( 0, (byte)97 );
11        mbb.put( 1023, (byte)122 );
12        raf.close();
13    }
14 }
```

## 选择器 Selector

NIO 中非阻塞 I/O 采用了基于 Reactor 模式的工作方式，I/O 调用不会被阻塞，相反是注册感兴趣的特定 I/O 事件，如可读数据到达，新的套接字连接等等，在发生特定事件时，系统再通知我们。NIO 中实现非阻塞 I/O 的核心对象就是 Selector，Selector 就是注册各种 I/O 事件地方，而且当那些事件发生时，就是这个对象告诉我们所发生的事件，如下图所示：



当有读或写等任何注册的事件发生时，可以从 Selector 中获得相应的 SelectionKey，同时从 SelectionKey 中可以找到发生的事件和该事件所发生的具体的 SelectableChannel，以获得客户端发送过来的数据

### NIO代码示例

使用 NIO 中非阻塞 I/O 编写服务器处理程序，大体上可以分为下面三个步骤：

1. 向 Selector 对象注册感兴趣的事件。
2. 从 Selector 中获取感兴趣的事件。
3. 根据不同的事件进行相应的处理。

```
1 package com.gupaoedu.vip.netty.io.nio;
2
3 import com.sun.org.apache.bcel.internal.generic.Select;
4
5 import java.io.IOException;
6 import java.net.InetSocketAddress;
7 import java.net.SocketAddress;
```

```
8 import java.nio.ByteBuffer;
9 import java.nio.channels.SelectionKey;
10 import java.nio.channels.Selector;
11 import java.nio.channels.ServerSocketChannel;
12 import java.nio.channels.SocketChannel;
13 import java.util.Iterator;
14 import java.util.Set;
15
16 /**
17 * NIO的操作过于繁琐，于是才有了Netty
18 * Netty就是对这一系列非常繁琐的操作进行了封装
19 *
20 * Created by Tom.
21 */
22 public class NIOServerDemo {
23
24     private int port = 8080;
25
26     //准备两个东西
27     //轮询器 Selector 大堂经理
28     private Selector selector;
29     //缓冲区 Buffer 等候区
30     private ByteBuffer buffer =
31         ByteBuffer.allocate(1024);
32
33     //初始化完毕
34     public NIOServerDemo(int port){
35         //初始化大堂经理，开门营业
36         try {
37             this.port = port;
38             ServerSocketChannel server =
39                 ServerSocketChannel.open();
40             //我得告诉地址
41             //IP/Port
42             server.bind(new
43                 InetSocketAddress(this.port));
44             //BIO 升级版本 NIO，为了兼容BIO，NIO模型默认是采
45             //用阻塞式
46             server.configureBlocking(false);
47
48             //大堂经理准备就绪，接客
49             selector = Selector.open();
50         } catch (IOException e) {
51             e.printStackTrace();
52         }
53     }
54
55     //处理客户请求
56     private void handle(SelectionKey key) throws IOException {
57         if(key.isAcceptable()){
58             ServerSocketChannel server = (ServerSocketChannel)key.channel();
59             SocketChannel client = server.accept();
60             client.configureBlocking(false);
61             client.register(selector, SelectionKey.OP_READ);
62         }
63         if(key.isReadable()){
64             SocketChannel client = (SocketChannel)key.channel();
65             ByteBuffer buffer = client.read(buffer);
66             if(buffer.remaining() == 0){
67                 client.close();
68             }
69         }
70     }
71
72     //启动大堂经理
73     public void start() throws IOException {
74         selector.select();
75         Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
76         while(iterator.hasNext()){
77             handle(iterator.next());
78             iterator.remove();
79         }
80     }
81 }
```

```
47         //在门口翻牌子，正在营业
48         server.register(selector,
49             SelectionKey.OP_ACCEPT);
50     } catch (Exception e) {
51         e.printStackTrace();
52     }
53 }
54
55     public void listen(){
56         System.out.println("listen on " + this.port +
57 ".");
58     try {
59         //轮询主线程
60
61         while (true){
62             //大堂经理再叫号
63             selector.select();
64             //每次都拿到所有的号子
65             Set<SelectionKey> keys =
66             selector.selectedKeys();
67             Iterator<SelectionKey> iter =
68             keys.iterator();
69             //不断地迭代，就叫轮询
70             //同步体现在这里，因为每次只能拿一个key，每次
71             //只能处理一种状态
72             while (iter.hasNext()){
73                 SelectionKey key = iter.next();
74                 iter.remove();
75                 //每一个key代表一种状态
76                 //没一个号对应一个业务
77                 //数据就绪、数据可读、数据可写 等等等等
78                 process(key);
79             }
80         }
81     }
82
83     //具体办业务的方法，坐班柜员
```

```
84     //每一次轮询就是调用一次process方法，而每一次调用，只能干一件事
85     //在同一时间点，只能干一件事
86     private void process(SelectionKey key) throws
87         IOException {
88         //针对于每一种状态给一个反应
89         if(key.isAcceptable()){
90             ServerSocketChannel server =
91                 (ServerSocketChannel)key.channel();
92             //这个方法体现非阻塞，不管你数据有没有准备好
93             //你给我一个状态和反馈
94             SocketChannel channel = server.accept();
95             //一定一定要记得设置为非阻塞
96             channel.configureBlocking(false);
97             //当数据准备就绪的时候，将状态改为可读
98             key =
99                 channel.register(selector,SelectionKey.OP_READ);
100        }
101        else if(key.isReadable()){
102            //key.channel 从多路复用器中拿到客户端的引用
103            SocketChannel channel =
104                (SocketChannel)key.channel();
105            int len = channel.read(buffer);
106            if(len > 0){
107                buffer.flip();
108                String content = new
109                    String(buffer.array(),0,len);
110                key =
111                    channel.register(selector,SelectionKey.OP_WRITE);
112                //在key上携带一个附件，一会再写出去
113                key.attach(content);
114                System.out.println("读取内容：" +
115                    content);
116            }
117        }
118        else if(key.iswritable()){
119            SocketChannel channel =
120                (SocketChannel)key.channel();
121            String content = (String)key.attachment();
122            channel.write(ByteBuffer.wrap(("输出：" +
123                content).getBytes()));
124        }
125    }
126}
```

```

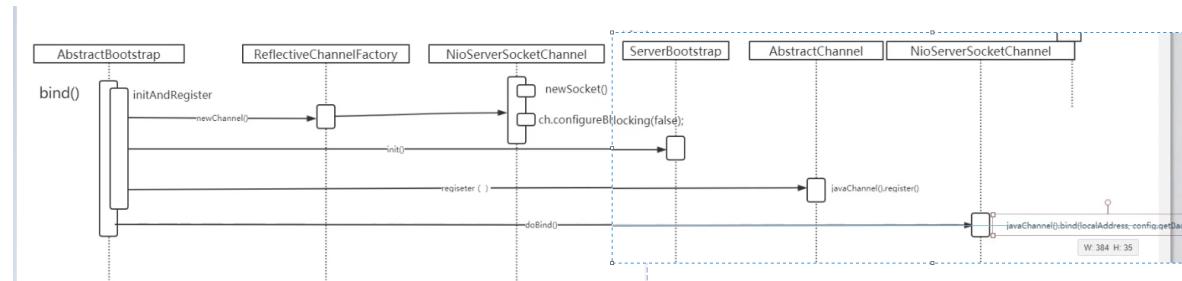
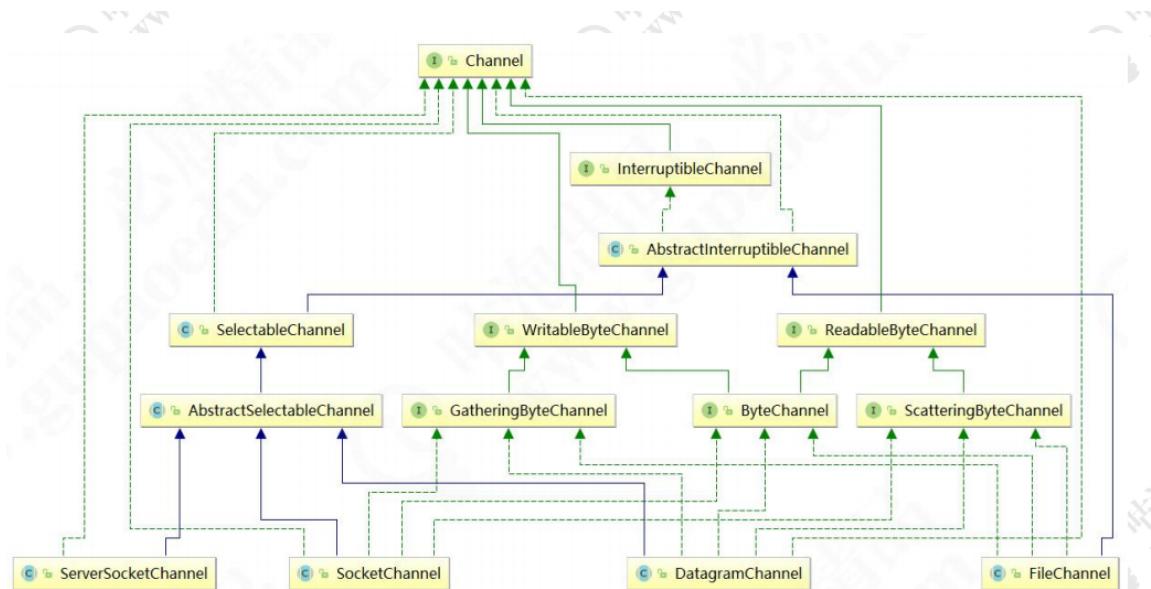
117             channel.close();
118     }
119 }
120
121 public static void main(String[] args) {
122     new NIOServerDemo(8080).listen();
123 }
124
125
126
127
128 }
129

```

## 通道 Channel

通道是一个对象，通过它可以读取和写入数据，当然了所有数据都通过 Buffer 对象来处理。我们永远不会将字节直接写入通道中，相反是将数据写入包含一个或者多个字节的缓冲区。同样不会直接从通道中读取字节，而是将数据从通道读入缓冲区，再从缓冲区获取这个字节

在 NIO 中，提供了多种通道对象，而所有的通道对象都实现了 Channel 接口。它们之间的继承关系如下图所示：



## 1. 使用 NIO 读取数据

任何时候读取数据，都不是直接从通道读取，而是从通道读取到缓冲区。所以使用 NIO 读取数据可

以分为下面三个步骤：

1. 从 FileInputStream 获得 Channel
2. 创建 Buffer
3. 将数据从 Channel 读取到 Buffer 中

下面是一个简单的使用 NIO 从文件中读取数据的例子：

## 2. 使用 NIO 写入数据

使用 NIO 写入数据与读取数据的过程类似，同样数据不是直接写入通道，而是写入缓冲区，可以分为下面三个步骤：

1. 从 FileInputStream 获得 Channel。
2. 创建 Buffer。
3. 将数据从 Channel 写入到 Buffer 中

```
1 public class FileInputStreamDemo {  
2  
3  
4     static public void main( String args[] ) throws  
Exception {  
5         FileInputStream fin = new  
FileInputStream("E://test.txt");  
6  
7         // 获取通道  
8         FileChannel fc = fin.getChannel();  
9  
10        // 创建缓冲区  
11        ByteBuffer buffer = ByteBuffer.allocate(1024);  
12  
13        // 读取数据到缓冲区  
14        fc.read(buffer);  
15  
16        buffer.flip();  
17  
18        while (buffer.remaining() > 0) {  
19            byte b = buffer.get();  
20            System.out.print(((char)b));  
}
```

```
21     }
22
23     fin.close();
24 }
25 }
26 package com.gupaoedu.vip.netty.io.nio.channel;
27 import java.io.*;
28 import java.nio.*;
29 import java.nio.channels.*;
30
31 public class FileOutputStreamDemo {
32     static private final byte message[] = { 83, 111, 109,
33     101, 32, 98, 121, 116, 101, 115, 46 };
34
35     static public void main( String args[] ) throws
36     Exception {
37         FileOutputStream fout = new FileOutputStream(
38             "E://test.txt" );
39
40         FileChannel fc = fout.getChannel();
41
42         ByteBuffer buffer = ByteBuffer.allocate( 1024 );
43
44         for (int i=0; i<message.length; ++i) {
45             buffer.put( message[i] );
46         }
47
48         buffer.flip();
49         fc.write( buffer );
50     }
51 }
```

## .IO 多路复用

Reactor模式与Observer模式在某些方面极为相似：当一个主体发生改变时，所有依属体都得到通知。不过，观察者模式与单个事件源关联，而反应器模式则与多个事件源关联。

select, poll, epoll都是IO多路复用的机制。

select, 同步, Windows, 轮询

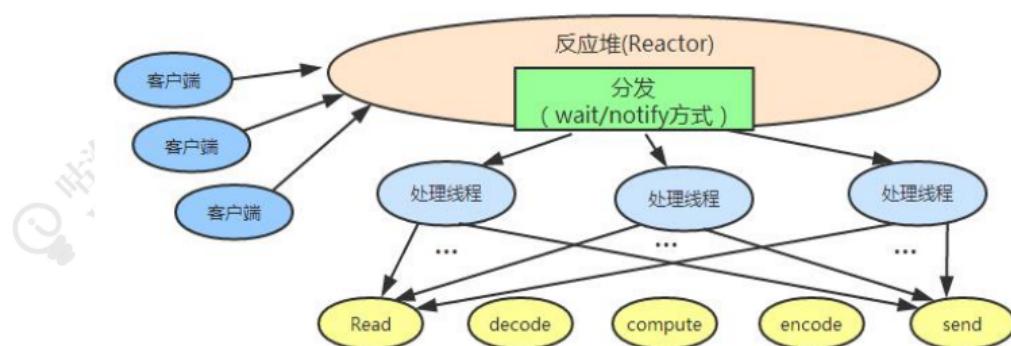
poll ,同步,Linux ,轮询

epoll,同步,Linux系统,事件机制

IO模型	相对性能	关键思路	操作系统	JAVA支持
select	较高	Reactor	windows/Linux	支持,Reactor 模式(反应器设计模式)。Linux 操作系统的 kernels 2.4 内核版本之前， 默认使用 select；而目前 windows 下对同步 IO 的支持，都是 select 模型。
poll	较高	Reactor	Linux	Linux 下的 JAVA NIO 框架，Linux kernels 2.6 内核版本之前使用 poll 进行支持。也是使用的 Reactor 模式。
epoll	高	Reactor/Proactor	Linux	Linux kernels 2.6 内核版本及以后使用 epoll 进行支持；Linux kernels 2.6 内核版本之前使用 poll 进行支持；另外一定注意，由于 Linux 下没有 Windows 下的 IOCP 技术提供真正的 异步 IO 支持，所以 Linux 下使用 epoll 模拟异步 IO。
kqueue	高	Proactor	Linux	目前 JAVA 的版本不支持。

## 反应堆 Reactor

咕泡出品，必属精品 www.gupaoedu.com



# Netty框架

---

## Netty 支持的功能与特性

按照定义来说，Netty 是一个异步、事件驱动的用来做高性能、高可靠性的网络应用框架。主要的优点有：

1. 框架设计优雅，底层模型随意切换适应不同的网络协议要求。
2. 提供很多标准的协议、安全、编码解码的支持。
3. 解决了很多 NIO 不易用的问题。
4. 社区更为活跃，在很多开源框架中使用，如 Dubbo、RocketMQ、Spark 等。

Netty 支持的功能或者特性：

1. 底层核心有：Zero-Copy-Capable Buffer，非常易用的灵拷贝 Buffer（这个内容很有意思，稍后专门来说）；统一的 API；标准可扩展的时间模型

2. 传输方面的支持有：管道通信（具体不知道干啥的，还请老司机指教）；  
Http 隧道；TCP 与 UDP

3. 协议方面的支持有：基于原始文本和二进制的协议；解压缩；大文件传输；  
流媒体传输；protobuf 编解码；安全认证；http 和 websocket

**用途：**开发RPC框架

### **Netty 采用 NIO 而非 AIO 的理由**

1. Netty 不看重 Windows 上的使用，在 Linux 系统上，AIO 的底层实现仍使用 EPOLL，没有很好实现 AIO，因此在性能上没有明显的优势，而且被 JDK 封装了一层不容易深度优化

2. Netty 整体架构是 reactor 模型，而 AIO 是 proactor 模型，混合在一起会非常混乱，把 AIO 也改造成 reactor 模型看起来是把 epoll 绕个弯又绕回来

3. AIO 还有个缺点是接收数据需要预先分配缓存，而不是 NIO 那种需要接收时才需要分配缓存，所以对连接数量非常大 但流量小的情况，内存浪费很多

4. Linux 上 AIO 不够成熟，处理回调结果速度跟不到处理需求，比如外卖员太少，顾客太多，供不应求，造成处理速度有瓶颈（待验证）

Netty 基本上是作为架构的技术底层而存在的，主要完成**高性能的网络通信**。

## **RPC框架**

分布式架构下服务数量逐渐增加，为了提高管理效率，RPC 框架应运而生。

**RPC 用于提高业务复用及整合的**，分布式服务框架下 RPC 是关键

RPC 服务治理框架主要有 Dubbo 和 Spring CloudDubbo 核

心模块主要有四个：Registry 注册中心、Provider 服务端、Consumer 消费端、Monitor 监控中心

简单PRC手写,远程调用java类,自定义协议就是定义一个java类  
类似代理类,远程代理名字的由来

主要模块包括：

api：主要用来定义对外开放的功能与服务接口。

protocol：主要定义自定义传输协议的内容。

registry：主要负责保存所有可用的服务名称和服务地址。

provider：实现对外提供的所有服务的具体功能。

consumer：客户端调用。

monitor：完成调用链监控

```

1 @Data
2 public class InvokerProtocol implements Serializable {
3     private String className;//类名
4     private String methodName;//函数名称
5     private Class<?>[] parames;//参数类型
6     private Object[] values;//参数列表 }

```

协议中主要包含的信息有类名、函数名、形参列表和实参列表，通过这些信息就可以定位到一个具体的业务逻辑实现

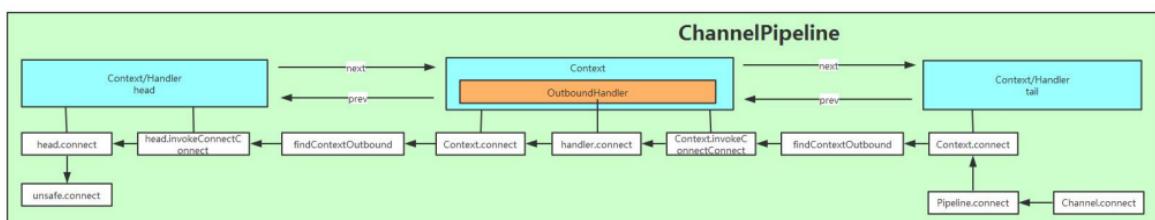
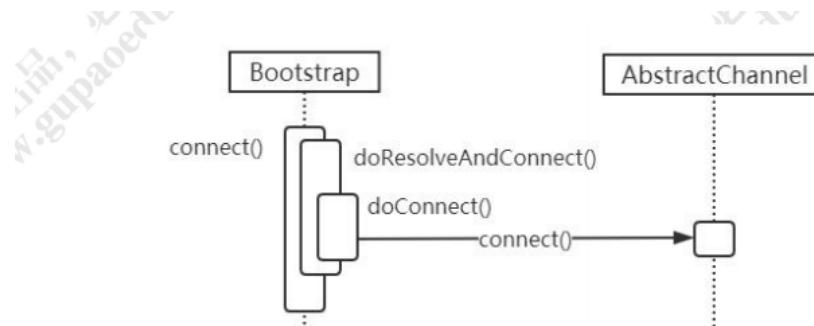
header 是一个 OutBoundHandler，而 tail 是一个 InboundHandler， inbound 类似于事件回调（响应请求的事件），而 outbound 类似于主动触发（发起请求的事件）。注意，如果我们捕获了一个事件，并且想让这个事件继续传递下去，那么需要调用 Context 对应的传播方法 fireXXX

## Outbound 事件传播方式

Outbound 事件都是请求事件(request event)，即请求某件事情的发生，然后通过 Outbound 事件进行通知。

Outbound 事件的传播方向是 tail -> customContext -> head。

Bootstrap 的 connect() 方法时，就会触发一个 Connect 请求事件，此调用会触发如下调用链



## Outbound 事件总结:

- 1、Outbound 事件是请求事件 (由 connect()发起一个请求，并最终由 unsafe 处理这个请求)
- 2、Outbound 事件的发起者是 Channel。
- 3、Outbound 事件的处理者是 unsafe。
- 4、Outbound 事件在 Pipeline 中的传输方向是 tail -> head。
- 5、在 ChannelHandler 中处理事件时，如果这个 Handler 不是最后一个 Handler，则需要调用 ctx 的方法 (如：ctx.connect()方法)将此事件继续传播下去。如果不这样做，那么此事件的传播会提前终止。
- 6、Outbound 事件流：Context.OUT\_EVT() ->  
Connect.findContextOutbound() -> nextContext.invokeOUT\_EVT() ->  
nextHandler.OUT\_EVT() -> nextContext.OUT\_EVT()

## Inbound 事件总结:

- 1、Inbound 事件是通知事件，当某件事情已经就绪后，通知上层。
- 2、Inbound 事件发起者是 unsafe。
- 3、Inbound 事件的处理者是 Channe，如果用户没有实现自定义的处理方法，那么 Inbound 事件默认的处理者是TailContext，并且其处理方法是空实现。
- 4、Inbound 事件在 Pipeline 中传输方向是 head -> tail。
- 5、在 ChannelHandler 中处理事件时，如果这个 Handler 不是最后一个 Handler，则需要调用 ctx.fireIN\_EVT()事件 (如：ctx.fireChannelActive()方法) 将此事件继续传播下去。如果不这样做，那么此事件的传播会提前终止。
- 6、Outbound 事件流：Context.fireIN\_EVT() ->  
Connect.findContextInbound() -> nextContext.invokeIN\_EVT() ->  
nextHandler.IN\_EVT() -> nextContext.fireIN\_EVT().

## Netty是如何解决Jdk空轮循bug的?

- 1 NioEventLoop#select(boolean oldWakenUp)中有个selectCnt变量，会在每次做 Selector.select(timeoutMillis)时对该变量进行递增。
- 2 如果**select后没有有效的key,并且任务队列没有任务,则用当前时间减去select开始时间,计算得到select毫秒.**
- 3 如果select耗时 >= select时设置的超时时间,没有发生空轮循.selectCnt重置

为1,执行下一次select

#### 4 如果select耗时 < select时设置的超时时间,则发生空轮循

判断selectCnt是否  $\geq \text{SELECTOR\_AUTO\_REBUILD\_THRESHOLD}$ (默认512,  
可通过属性io.netty.selectorAutoRebuildThreshold设置),

如果大于等于该值则重新构建Selector,将OldSelector上的keys注册到新的  
Selector,selectCnt重置为1,执行下一次select

#### 1、EventLoopGroup创建

CPU核数 \* 2 默认值  
线程的命名 eventLoop-1-xxx  
mpscQueue EventLoopGroup  
execute()  
EventLoop取值 判断是否2的幂, 采用位运算, 不是2的幂采用取模运算

#### 2、EventLoop执行

selector 记录每一次轮询的耗时, 然后, 也记录了轮询次数  
空轮训次数大于512次, 把所有key.cancel() rebuildSelector()

process

每一次执行会判断inEventLoop() 看是不是Netty创建的线程, 加入执行队列  
如果不是新建一个Runnable接口进行包装, 再存入队列  
next().register(OP\_READ)

select 优化 HashSet直接替换成了数组

#### 3、客户端的接入

doReadMessage()方法  
pipeline.fireChannelRead()  
ChannelHandler channelRead()

unsafe.read()

I

Nagle算法是以他的发明人John Nagle的名字命名的, 它用于自动连接许多的小缓冲器消息; 这一过程(称为nagling)通过减少必须发送包的个数来增加网络软件系统的效率

**反斜线、转义和引用** 反斜线字符('\'')用于引用转义构造, 如上表所定义的, 同时还用于引用其他将被解释为非转义构造的字符。因此, 表达式 \\ 与单个反斜线匹配, 而 \{} 与左括号匹配。在不表示转义构造的任何字母字符前使用反斜线都是错误的; 它们是为将来扩展正则表达式语言保留的。可以在非字母字符前使用反斜线, 不管该字符是否非转义构造的一部分。

根据 [Java Language Specification](#) 的要求, Java 源代码的字符串中的反斜线被解释为 [Unicode 转义](#)或其他字符转义。因此必须在字符串字面值中使用两个反斜线, 表示正则表达式受到保护, 不被 Java 字节码编译器解释。例如, 当解释为正则表达式时, 字符串字面值 "\b" 与单个退格字符匹配, 而 "\\b"

与单词边界匹配。字符串字面值 `"\(\text{hello}\)"` 是非法的，将导致编译时错误；要与字符串 `(hello)` 匹配，必须使用字符串字面值 `"\\(\text{hello}\\)"`。

**组和捕获** 捕获组可以通过从左到右计算其开括号来编号。例如，在表达式 `((A)(B(C)))` 中，存在四个这样的组： 1 `((A)(B(C)))` 2 `\A` 3 `(B(C))` 4 `(C)`

`^`:匹配输入字符串的开始位置

`\s` 空白字符: `[ \t\n\x0B\f\r]` `\S` 非空白字符: `[^\s]`

## 高效的Reactor线程模型

常见的Reactor线程模型有三种，分别如下：

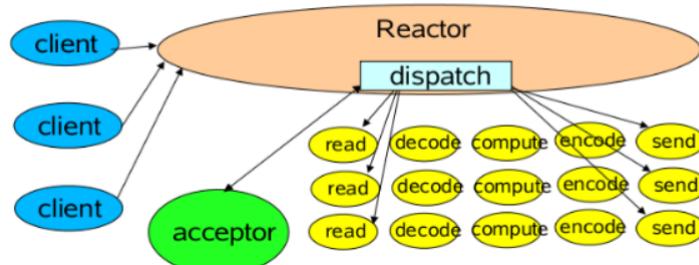
1. Reactor单线程模型；
2. Reactor多线程模型；
3. 主从Reactor多线程模型；

### Reactor单线程模型

Reactor单线程模型，指的是所有的I/O操作都在同一个NIO线程上面完成， NIO线程的职责如下：

1. 作为NIO服务端，接收客户端的TCP连接；
2. 作为NIO客户端，向服务端发起TCP连接；
3. 读取通信对端的请求或者应答消息；
4. 向通信对端发送消息请求或者应答消息；

Reactor线程是个多面手，负责多路分离套接字，Accept新连接，并分派请求到处理器链中。该模型适用于处理器链中业务处理组件能快速完成的场景。不过，这种单线程模型不能充分利用多核资源，所以实际使用的不多。



## 存在问题

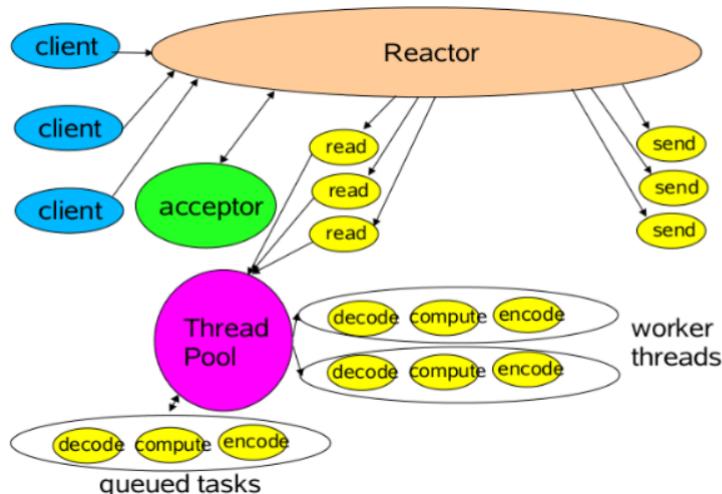
对于一些小容量应用场景，可以使用单线程模型，但是对于高负载、大并发的应用却不合适，主要原因如下：

1. 一个NIO线程同时处理成百上千的链路，性能上无法支撑。即便NIO线程的CPU负荷达到100%，也无法满足海量消息的编码、解码、读取和发送；
2. 当NIO线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往进行重发，这更加重了NIO线程的负载，最终导致大量消息积压和处理超时，NIO线程会成为系统的性能瓶颈；
3. 可靠性问题。一旦NIO线程意外跑飞，或者进入死循环，会导致整个系统通讯模块不可用，不能接收和处理外部信息，造成节点故障。

## Reactor多线程模型

Reactor多线程模型与单线程模型最大区别就是有一组NIO线程处理I/O操作，它的特点如下：

1. 有一个专门的NIO线程--acceptor新城用于监听服务端，接收客户端的TCP连接请求；
2. 网络I/O操作--读、写等由一个NIO线程池负责，线程池可以采用标准的JDK线程池实现，它包含一个任务队列和N个可用的线程，由这些NIO线程负责消息的读取、解码、编码和发送；
3. 1个NIO线程可以同时处理N条链路，但是1个链路只对应1个NIO线程，防止发生并发操作问题。



不足：在绝大多数场景下，Reactor多线程模型都可以满足性能需求；但是，在极特殊应用场景中，一个NIO线程负责监听和处理所有的客户端连接可能会存在性能问题。例如百万客户端并发连接，或者服务端需要对客户端的握手信息进行安全认证，认证本身非常损耗性能。这类场景下，单独一个Acceptor线程

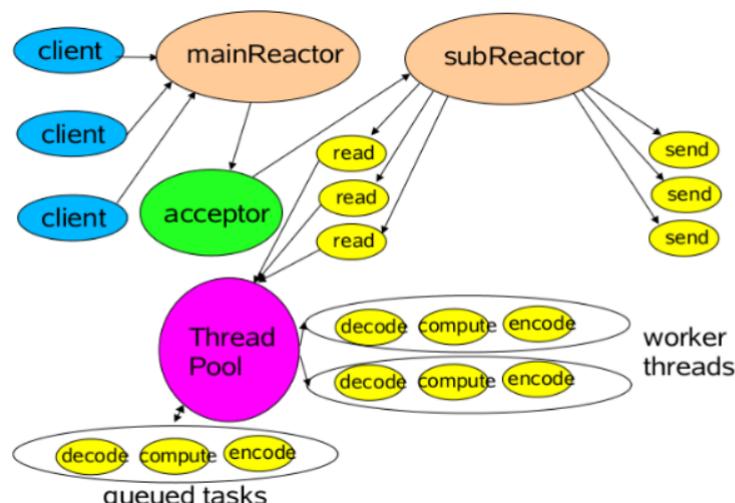
可能会存在性能不足问题，为了解决性能问题，产生了第三种Reactor线程模型--主从Reactor多线程模型。

### 主从Reactor多线程模型

特点是：服务端用于接收客户端连接的不再是1个单独的NIO线程，而是一个独立的**NIO线程池**。Acceptor接收到客户端TCP连接请求处理完成后（可能包含接入认证等），将新创建的SocketChannel注册到I/O线程池（sub reactor线程池）的某个I/O线程上，由它负责SocketChannel的读写和编解码工作。

Acceptor线程池只用于客户端的登录、握手和安全认证，一旦链路建立成功，就将链路注册到后端subReactor线程池的I/O线程上，有I/O线程负责后续的I/O操作。

第三种模型比起第二种模型，是将Reactor分成两部分，mainReactor负责监听server socket，accept新连接，并将建立的socket分派给subReactor。subReactor负责多路分离已连接的socket，读写网络数据，对业务处理功能，其扔给worker线程池完成。通常，subReactor个数上可与CPU个数等同。



NioEventLoopGroup 与 Reactor 线程模型的对应自由切换

```
1 /**
2  * Netty单线程模型服务端代码示例
3  * @param port
4  */
5 public void bind(int port) {
6     EventLoopGroup reactorGroup = new
7 NioEventLoopGroup();
8     try {
9         ServerBootstrap b = new ServerBootstrap();
10        b.group(reactorGroup, reactorGroup)
11            .channel(NioServerSocketChannel.class)
```

```
11         .childHandler(new
12             ChannelInitializer<SocketChannel>() {
13                 @Override
14                 protected void
15                     initChannel(SocketChannel ch) throws Exception {
16                         ch.pipeline().addLast("http-
17                             codec", new HttpServerCodec());
18
19                         ch.pipeline().addLast("aggregator", new
20                             HttpObjectAggregator(65536));
21
22                         ch.pipeline().addLast("http-
23                             chunked", new ChunkedWriteHandler());
24
25                         //后面代码省略
26                     }
27
28                 });
29
30     /**
31      * Netty多线程模型代码
32      * @param port
33      */
34     public void bind2(int port) {
35         EventLoopGroup acceptorGroup = new
36             NioEventLoopGroup(1);
37         NioEventLoopGroup ioGroup = new
38             NioEventLoopGroup();
39
40         try {
41             ServerBootstrap b = new ServerBootstrap();
42             b.group(acceptorGroup, ioGroup)
43                 .channel(NioServerSocketChannel.class)
44                 .childHandler(new
45                     ChannelInitializer<SocketChannel>() {
46                         @Override
47                         protected void
48                             initChannel(SocketChannel ch) throws Exception {
```

```
43         ch.pipeline().addLast("http-
44         codec", new HttpServerCodec());
45
46         ch.pipeline().addLast("aggregator", new
47         HttpObjectAggregator(65536));
48             ch.pipeline().addLast("http-
49             chunked", new ChunkedWriteHandler());
50                 //后面代码省略
51             }
52         });
53
54     }
55
56     Channel ch = b.bind(port).sync().channel();
57     ch.closeFuture().sync();
58 } catch (InterruptedException e) {
59     e.printStackTrace();
60 } finally {
61     acceptorGroup.shutdownGracefully();
62     ioGroup.shutdownGracefully();
63 }
64 /**
65 * Netty主从线程模型代码
66 * @param port
67 */
68 public void bind3(int port) {
69     EventLoopGroup acceptorGroup = new
70     NioEventLoopGroup();
71     NioEventLoopGroup ioGroup = new
72     NioEventLoopGroup();
73     try {
74         ServerBootstrap b = new ServerBootstrap();
75         b.group(acceptorGroup, ioGroup)
76             .channel(NioServerSocketChannel.class)
77             .childHandler(new
78         ChannelInitializer<SocketChannel>() {
79             @Override
80             protected void
81         initChannel(SocketChannel ch) throws Exception {
82                 ch.pipeline().addLast("http-
83                 codec", new HttpServerCodec());
84
85                 ch.pipeline().addLast("aggregator", new
86                 HttpObjectAggregator(65536));
87             }
88         });
89     }
90 }
```

```
75                     ch.pipeline().addLast("http-
    chunked", new ChunkedWriteHandler());
76                         //后面代码省略
77                     }
78                 });
79
80             Channel ch = b.bind(port).sync().channel();
81             ch.closeFuture().sync();
82         } catch (InterruptedException e) {
83             e.printStackTrace();
84         } finally {
85             acceptorGroup.shutdownGracefully();
86             ioGroup.shutdownGracefully();
87         }
88     }
```

## Reactor线程模型NioEventLoopGroup

### 创建阶段

1.NioEventLoopGroup构造方法创建Threadfactory创建,每个线程独立的名字

2.newChild()方法

3.会被放到线程队列里面去MpscQueue(线程池内部执行队列)

4newchooser() 获取一个线程(封装成EventLoop),执行阶段会调用  
Event.execute

### 执行阶段

无锁化穿行(线程安全)

1.bind()

2调用EventLoopde.execute,inEventLoop()现在拿到的这个EventLoop 是不是  
Netty自己创建的线程,每个线程的任务,一定是Netty自己创建的(保证线程的安  
全)

3.selector关联,轮询所以得事件

select() 方法(封装selectionKey)

processSelectedKeys

新连接的处理阶段

Nagle算法,延迟发送,缓冲一定数据再发送减少网络开销

NioSocketChannel Op\_READ

PipeLine 最终放到全部是channleHandler线程池下去执行,p.next()

零拷贝:

1.接受和发送bytebuffer使用堆外内存直接进入socket读写(省略操作系统内核).

提供组合buffer对象,可以聚合多个bytebuffer对象

transferTo()直接将文件换从区的数据发送到目标Chanel

内存池

pooled与unpooled

unsafe与非unsafa(直接调用计算机读写)

heap和direct(堆内存和堆外内存)

## Netty高性能的原因

1.volatile的大量/正确使用

2.CAS和原子类的广泛使用

3.线程安全容器的使用;

4通过读写锁提升并发性能

影响序列化性能的关键因素:

1.序列化后的码流大小(带宽)

2.序列化与反序列化得性能(cpu性能)

3.是否支持跨语言(异构系统的开发)

灵活的TCP参数配置能力

扩展了大量的参数

## 涉及的设计模式

迭代器模式要点回顾：

1. 实现迭代器接口
2. 实现对容器中的各个对象逐个访问的方法。

```
1 public class CompositeByteBuf extends
AbstractReferenceCountedByteBuf implements
Iterable<ByteBuf>
2 {
3     protected byte _getByte(int index) {
4         CompositeByteBuf.Component c =
this.findComponent(index);
5         return c.buf.getByte(index - c.offset);
6     }
7 }
```

**ChannelPipeline责任链：**是指多个对象都有机会处理同一个请求，从而避免请求的发送者和接收者之间的耦合关系。然后，将这些对

象连成一条链，并且沿着这条链往下传递请求，直到有一个对象可以处理它为止。在每个对象处理过程中，每个对象

只处理它自己关心的那一部分，不相关的可以继续往下传递，直到链中的某个对象不想处理，可以将请求终止或丢弃。

责任链模式要点回顾：

- 1、需要有一个顶层责任处理接口（ChannelHandler）。**责任链顺序,在初始化划得时候任务都加入了一个list集合**
- 2、需要有动态创建链、添加和删除责任处理器的接口（ChannelPipeline）。
- 3、需要有上下文机制（ChannelHandlerContext）。

4、需要有责任终止机制（不调用 ctx.fireXXX()方法，则终止传播）。

```
1 abstract class AbstractChannelHandlerContext extends  
2 DefaultAttributeMap implements ChannelHandlerContext,  
3 ResourceLeakHint {  
4     private AbstractChannelHandlerContext findContextInbound()  
5     {  
6         AbstractChannelHandlerContext ctx = this;  
7         do {ctx = ctx.next; }  
8         while(!ctx.inbound);  
9         return ctx; }  
10    }
```

## Netty 答疑

1. Netty 定位：

A、作为开源码框架的底层框架（TCP 通信）

SpringBoot 内置的容器(Tomcat/Jerry) Zookeper 数据交互 Dubbo 多协议 RPC 的支持

B、直接做服务器（消息推送服务，游戏后台）

2、Netty 如何确定要使用哪些编码器和解码器

很简单，看 API 文档 Netty 自带的编解码器可以解决 99% 的业务需求 1% 自己编解码

3、Netty 中大文件上传的那个 handler 是怎么做到防止内存撑爆的

ByteBuf 分片，直接缓冲区，0 拷贝，提高内存的利用率 加内存

4、Tomcat NIO 方式的调优线程，本质上是对 netty 的调优吗 8.5 之后开始用 Netty？

5、责任链模式能否用在，一个操作出口参数为另一个操作的入口 执行顺序有关系，有先后 API 设计 callable(上一次调用的结果)，msg (皮球)

6、Netty 里面 Pooled 缓冲区和 Unpooled 缓冲区内存分配（在录播中讲解）

7、Linux 底层 IO 模型，主从，多路复用的思想（录制一个基于硬件层面 IO 模型，显得更加专业）

8、Selector 客户端与服务端之间是什么关系？

客户端：CONNECT READ WRITE 服务端：ACCEPT READ WRITE

## 面试题

### Netty 定位：

- A、作为开源码框架的底层框架 (TCP 通信)
- B、直接做服务器 (消息推送服务，游戏后台)

### Netty 如何确定要使用哪些编码器和解码器

Netty 自带的编解码器可以解决 99% 的业务需求 1% 自己编解码

### Netty 中大文件上传的那个 handler 是怎么做到防止内存撑爆的

ByteBuf 分片，直接缓冲区，0 拷贝，提高内存的利用率，加内存

责任链模式能否用在，一个操作出口参数为另一个操作的入口, API 设计  
callable(上一次调用的结果)， msg (皮球)

## zookeeper

---

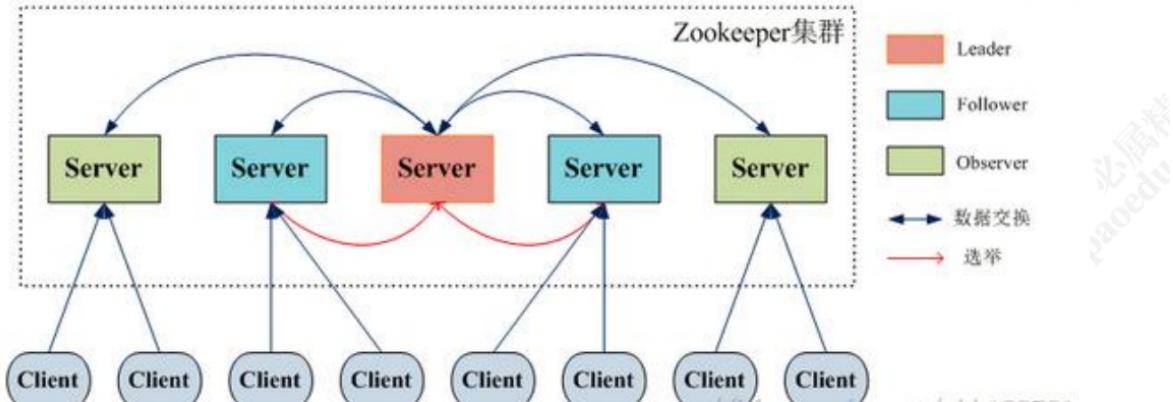
用 docker 安装 zookeeper 集群

[https://blog.csdn.net/qq\\_38270106/article/details/88789737?utm\\_medium=distribute.pc\\_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-3.control&depth\\_1-utm\\_source=distribute.pc\\_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-3.control](https://blog.csdn.net/qq_38270106/article/details/88789737?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-3.control&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-3.control)

Zookeeper 作为一个分布式协调框架，内部存储了一些分布式系统运行时的状态的数据，比如 master 选举、比如分布式锁。对这些数据的操作会直接影响到分布式系统的运行状态。因此，为了保证 zookeeper 中的数据的安全性，避免误操作带来的影响。Zookeeper 提供了一套 ACL 权限控制机制来保证数据的安全。

ZooKeeper 主要服务于分布式系统，可以用 ZooKeeper 来做：统一配置管理、统一命名服务、分布式锁、集群管理。

使用分布式系统就无法避免对节点管理的问题(需要实时感知节点的状态、对节点进行统一管理等等)，而由于这些问题处理起来可能相对麻烦和提高了系统的复杂性，ZooKeeper 作为一个能够通用解决这些问题的中间件就应运而生了



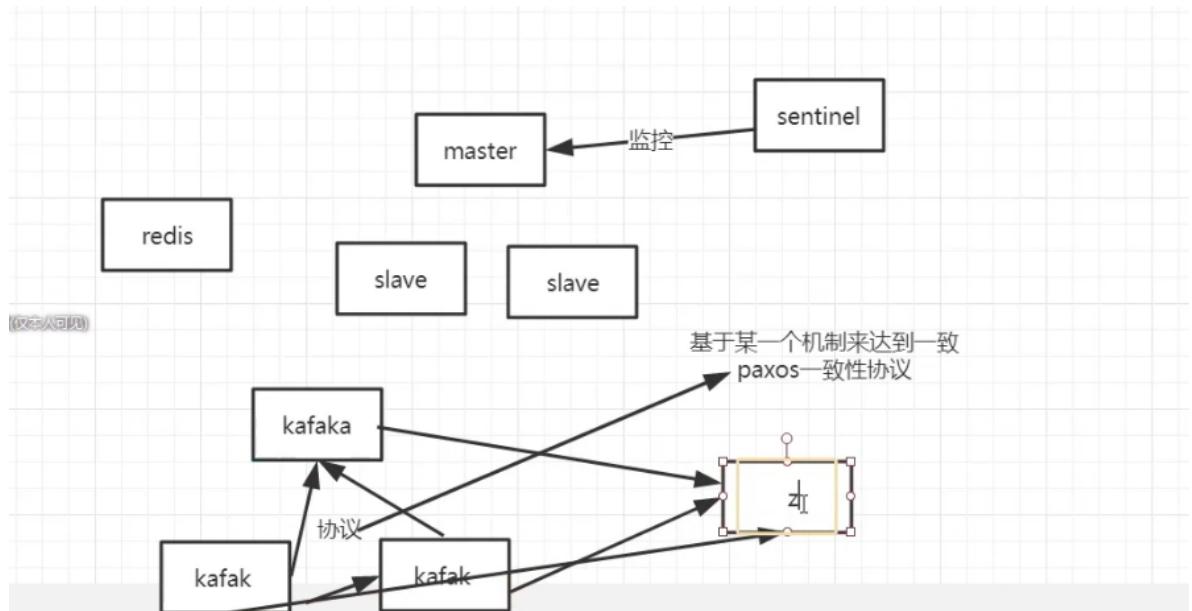
zookeeper 的视图结构和标准的文件系统非常类似，每一个节点称之为 ZNode，是 zookeeper 的最小单元。每个 znode 上都可以保存数据以及挂载子节点。构成一个层次化的树形结构

持久节点 (PERSISTENT)

临时节点(EPHEMERAL)

临时有序节点(EPHEMERAL)

curator



1. zookeeper实现分布式锁,

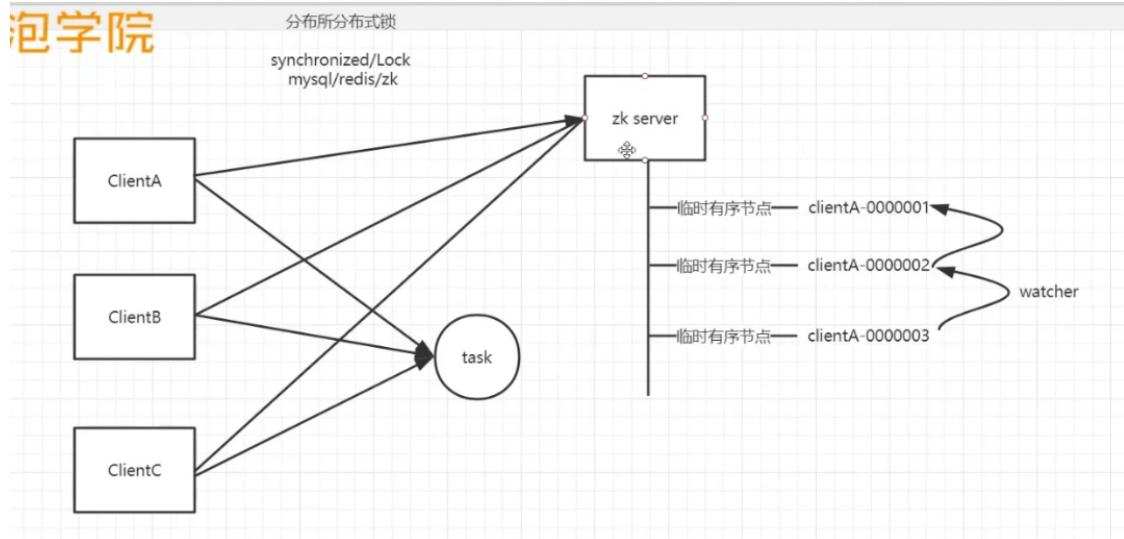
利用 zookeeper 节点的特性来实现独占锁，就是同级节点的唯一性,惊群效应不适合高并发舍去

2. 利用有序节点来实现分布式锁

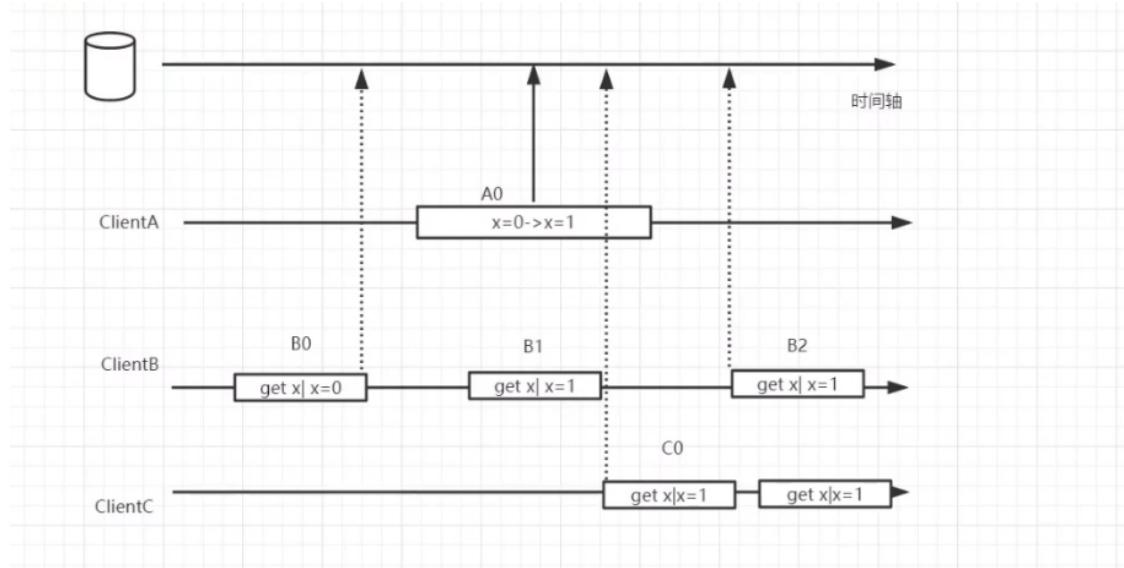
每个客户端都往**指定的节点下注册一个临时有序节点**,越早创建的节点,节点的顺序编号就越小,最小的节点设置为获得锁。

当比自己小的节点删除以后,客户端会收到watcher事件,此时再次判断自己的节点是不是所有子节点中最小的,如果是则获得锁,否则就不断重复这个过程每个客户端只需要监控一个节点.curator框架去练zookeeper加锁更加方便

<https://blog.csdn.net/lovexiaotaozi/article/details/83382128>



### 3. Zookeeper数据的同步流程



### 4. Zab协议(强一致性)

ZAB (Zookeeper Atomic Broadcast) 协议是为分布式协调服务 ZooKeeper 专门设计的一种支持崩溃恢复的原子广播协议

- . 崩溃恢复,leader挂掉后启动
- . 原子广播,半数follower和leader同步数据时提交(类似2pc提交)  
leader负责事务

```

case LOOKING: //第一次进入到这个case
    // If notification > current, replace and send messages out
    if (n.electionEpoch > logicalclock.get()) { //
        logicalclock.set(n.electionEpoch);
        recvset.clear(); //清空
        //收到票据之后，当前的server要听谁的。
        //可能是听server1的、也可能是听server2，也可能是听server3
        //zab leader选举算法
        if(totalOrderPredicate(n.leader, n.xqid, n.peerEpoch,
            getInitId(), getInitLastLoggedXqid(), getPeerEpoch())) {
            //把自己的票据更新成对方的票据，那么下一次，发送的票据就是新的票据
            updateProposal(n.leader, n.xqid, n.peerEpoch);
        } else {
            //收到的票据小于当前的节点的票据，下一次发送票据，仍然发送自己的
            updateProposal(getInitId(),
                getInitLastLoggedXqid(),
                getPeerEpoch());
        }
        //继续发送通知
        sendNotifications();
    } else if (n.electionEpoch < logicalclock.get()) { //说明当前的数据已经过期了
        if(LOG.isDebugEnabled()){

```

## 5. 分布式数据原子性

zookeeper 为数据节点引入了版本的概念，每个数据节点都有三类版本信息，对数据节点任何更新操作都会引起版本号的变化,类似cas机制

版本类型	说 明
version	当前数据节点数据内容的版本号
cversion	当前数据节点子节点的版本号
aversion	当前数据节点 ACL 变更版本号

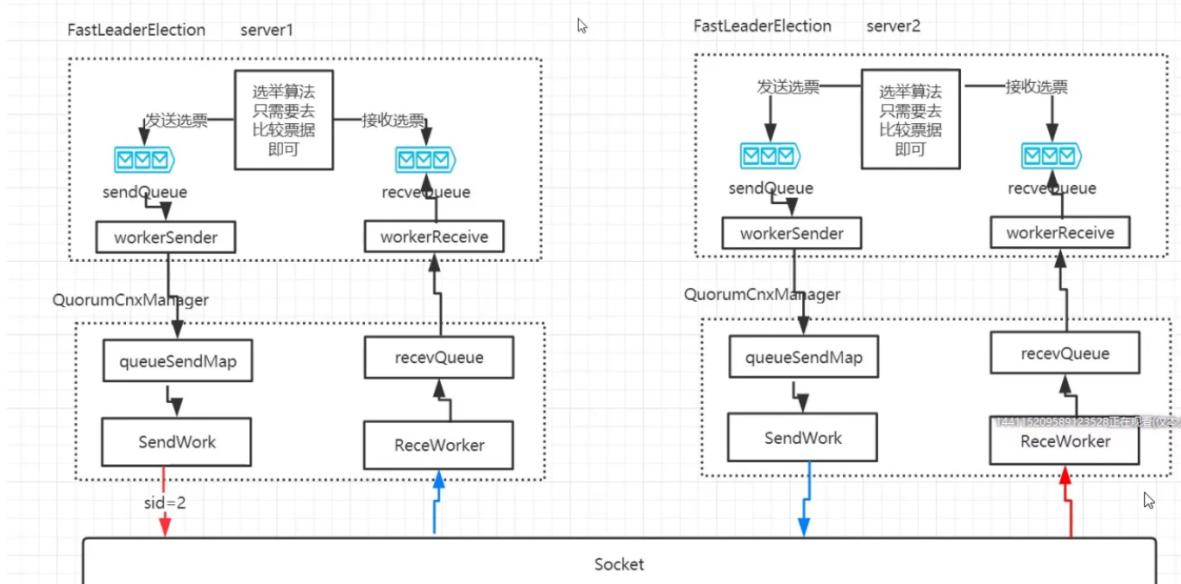
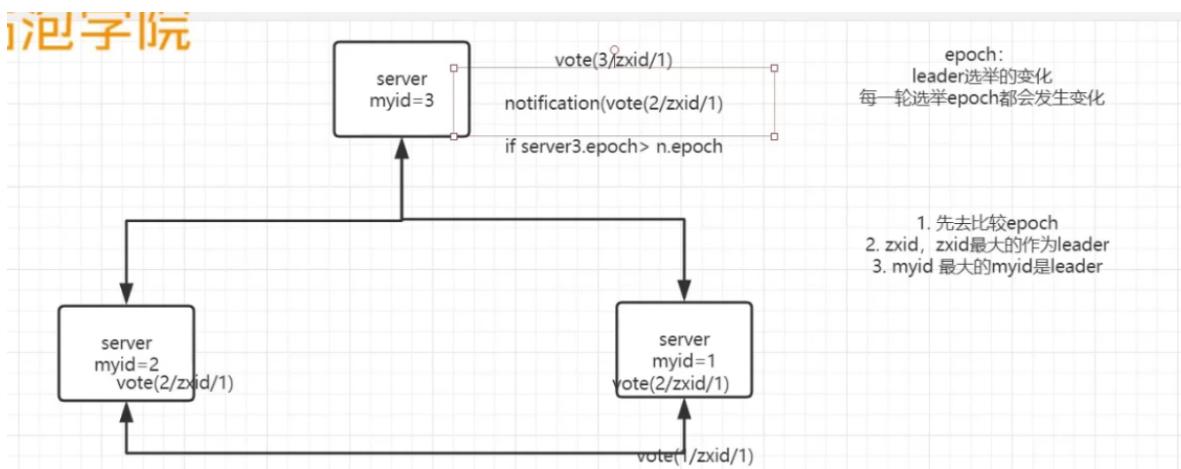
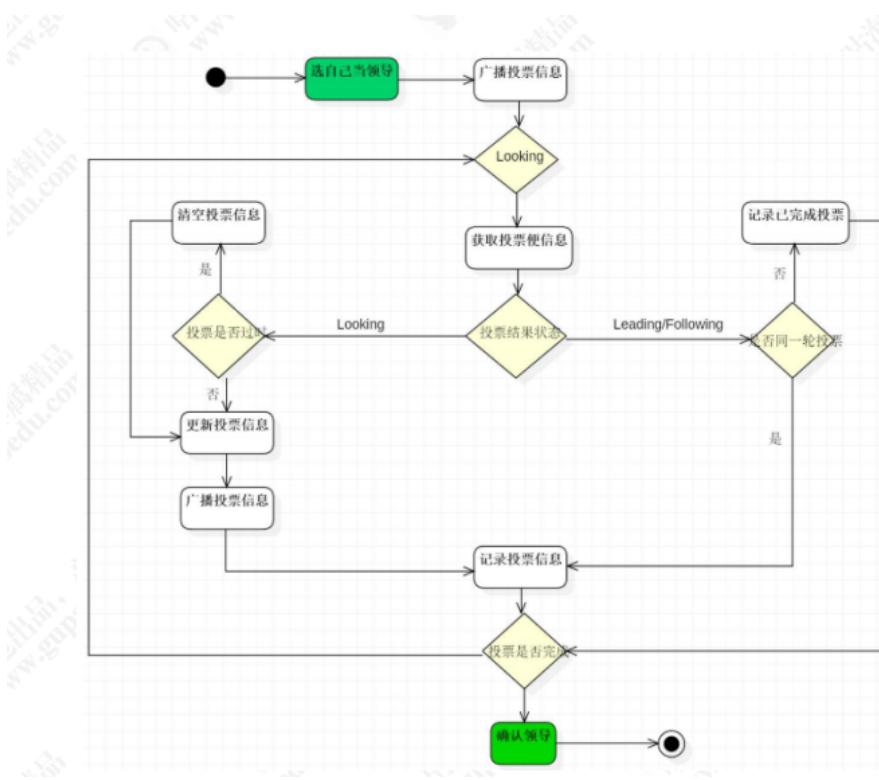
## 6. Leader选举原理及源码分析

s1,s2,s3

根据myid和zid(事物)来选,初始化时(1,0),(2,0),(3,0),选举最小的myid(1,0)  
崩溃恢复的时候再选,选zid最大也就是数据最新的follower,选举中所有节点  
**加锁停止服务,**

zxid 是 64 位，高 32 位是 epoch 编号，每经过一次 Leader 选举产生一个新的 leader，新的 leader 会将 epoch 号+1，低 32 位是消息计数器，每接收到一条消息这个值+1，**新 leader 选举后这个值重置为 0.**这样设计的好处在与老的 leader 挂了以后重启，它不会被选举为 leader，因此此时它的 zxid 肯定小于当前新的 leader。

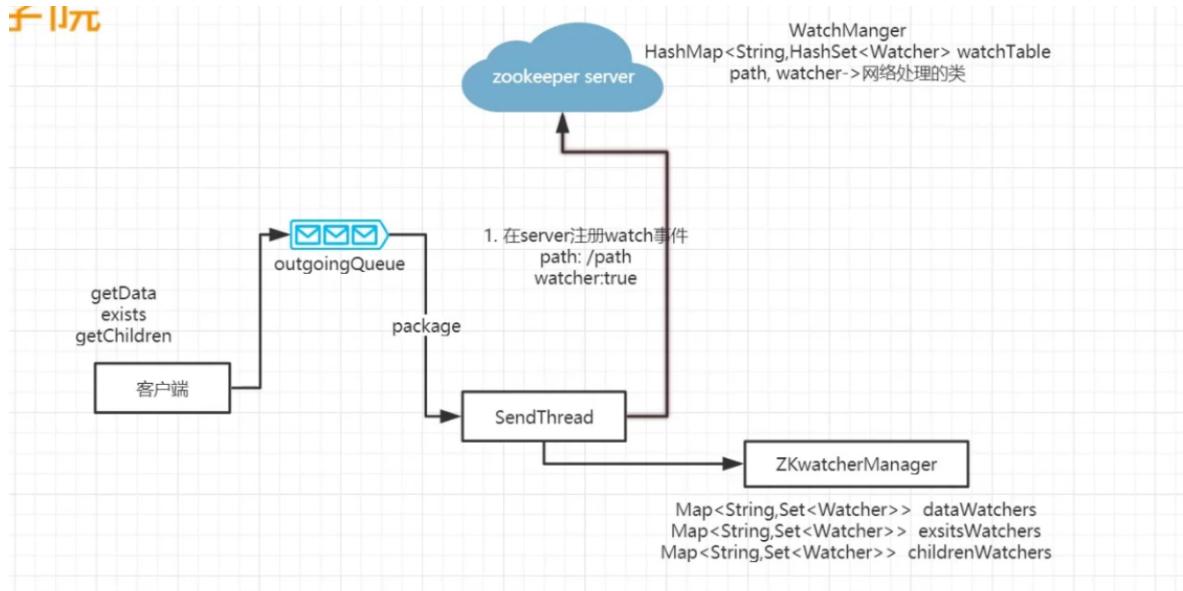
主写从读



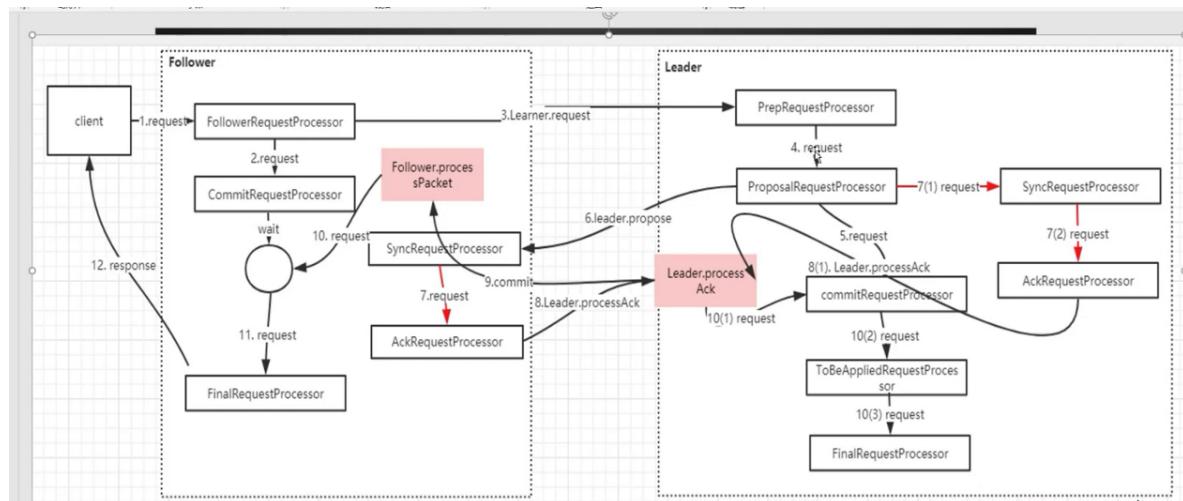
5.watch 监听,注册一次消费就没有了,在客户端存储,

ZooKeeper 的 Watcher 机制，总的来说可以分为三个过程：客户端注册 Watcher、服务器处理 Watcher 和客户端回调 Watcher

客户端注册 watcher 有 3 种方式，getData、exists、getChildren；



集群模式下的watch图



1. zookeeper的背景 (chubby的分布式锁和leader选举)
2. zookeeper实现服务的注册和发现(临时节点、 watcher)
3. 安装
4. 基本的操作，如何基于java客户端api来访问zookeeper server实现对于zk server的数据的crud
5. 数据有特性 (临时性、有序性、持久化、同级节点的唯一性)
6. zookeeper基于这些特性能解决什么问题？ leader选举（顺序临时节点）、分布式锁、服务的注册发现
7. zookeeper 的原理 (zab协议) -> 崩溃恢复 (leader选举)，原子广播 (数据同步)
8. 通过源码来分析leader选举的原理 (myid/zxid/epoch)
9. watcher的机制，实现节点的事件监听 (节点的修改，删除、增加事件) -> 配置中心
10. 分析了watcher的核心源码
11. zookeeper的jute的序列化
12. zookeeper顺序一致性
13. zookeeper的网络通信 (NIO)

leader(负责事务请求)->写比较多的情况下性能瓶颈

cp -> 在很多需要ap的场景中，无法使用<sup>I</sup>

## zookeeper选举详细过程

[https://blog.csdn.net/weixin\\_39618279/article/details/114734332](https://blog.csdn.net/weixin_39618279/article/details/114734332)

# Nginx

## | 简介

根据前面的对比，我们可以了解到Nginx是一个http服务器。是一个使用c语言开发的高性能的**http服务器**及反向代理服务器。Nginx是一款高性能的http服务器/反向代理服务器及电子邮件 (IMAP/POP3) 代理服务器。由俄罗斯的程序设计师Igor Sysoev所开发，官方测试nginx能够支撑5万并发链接，并且cpu、内存等资源消耗却非常低，运行非常稳定。

## 反向代理: 代理服务器端

### Nginx的应用场景

- 1、 http服务器。Nginx是一个http服务可以独立提供http服务。可以做网页静态服务器。
- 2、 虚拟主机。可以实现在一台服务器虚拟出多个网站。例如个人网站使用的虚拟主机,基于端口的，不同的端口,基于域名的，不同域名
- 3、 反向代理，负载均衡。当网站的访问量达到一定程度后，单台服务器不能满足用户的请求时，需要用多台服务器集群可以使用nginx做反向代理。并且多台服务器可以平均分担负载，不会因为某台服务器负载高宕机而某台服务器闲置的情况。

# 配置文件解读

\*可以设置浏览器缓存,对于图片设置一段时间内不能重复访问服务器

\*根据数据类型可以做压缩处理,减少网络带宽

[https://blog.csdn.net/wangbin\\_0729/article/details/82109693](https://blog.csdn.net/wangbin_0729/article/details/82109693)

## Nginx日志配置处理

### 一、介绍日志文件格式

日志文件的记录是有格式的,我们可以按系统默认的格式去记录,也可以按我们自定义的格式去记录。可以用log\_format指令来设置Nginx服务器的日志文件的记录格式。

### 二、日志格式说明

```
1 log_format main '$remote_addr - $remote_user
2   [$time_local] "$request"
3
4
5           '$status $body_bytes_sent
6           "$http_referer"
7
8
9           '"$http_user_agent"
  "$http_x_forwarded_for";
```

main: 日志格式

\$remote\_addr: 客户端ip地址

\$remote\_user: 客户端用户名

\$request: 请求的url

\$status: 请求状态 (比如: 正在请求中, 请求完成等状态)

\$body\_bytes\_sent: 发送给客户端的字节数 (服务端返回给客户端数据的大小)

\$http\_referer: 原网页的url (从哪个url打开的所访问的页面)

\$http\_user\_agent: 客户端的浏览器记录的信息(比如是google浏览器, 那么就记录google浏览器的类型等基本信息)

\$http\_x\_forwarded\_for: 客户端ip地址 (类似于\$remote\_addr)

通常情况下用以上配置即可，可以将main换成combined(默认的日志格式)，其他不变即可。

### 三、Nginx日志文件存储路径配置

#### 1、说明

日志文件在记录的时候，需要存储到磁盘上，存储的路径是可配置的，通过access\_log指令来配置Nginx的日志文件的存储路径。

#### 2、修改配置

```
access_log logs/access.log main;
```

logs/access.log：这个为日志文件的存放路径，从安装目录开始，意思绝对路径为：/usr/local/nginx/logs/access.log

main：为日志格式，通常为combined

若不想记录Nginx的日志，则将access\_log路径配成off即可，如下：

```
access_log off;
```

### 四、Nginx日志文件的切割

#### 1、说明：

为了使Nginx的日志文件存储更合理、有序，我们需要将日志文件进行分开存储，比如我们可以按时间来分开，今天的日志文件存储到一个文件中，明天的日志文件则存储到另一个新的文件中等等。这时候就需要用到了Nginx的日志文件的切割。

有两种方式：

#### 1、手动进行切割

步骤：分两步骤。

(1) 进入logs目录。执行命令

```
mv access.log xxx (随便起个名) .log
```

意思是说将以前的日志文件重命名为一个新的名字的日志文件。

(2) 执行命令

```
kill -USR1 主进程号(需要先用ps -ef|grep nginx命令找到master的进程号)
```

kill -USR1 主进程号 这个命令在信号控制那章节有提到。进行日志切割的命令。

操作完(1)(2)两步骤后，在logs目录下ls一下，会发现又自动多出了一个日志文件(这个日志文件的命名来源于nginx.conf文件里的access\_log的路径)，并且里面无内容。这个文件就是切割出来的新文件，再有日志会往这里面写，而不会操作老日志文件。

#### 2、自动进行切割

步骤：

(1) 首先创建个sh文件(称为批处理日志文件)，进入nginx的logs目录运行命令

```
touch cutlog.sh (文件名自定义，后缀.sh即可)
```

(2) 编辑cutlog.sh文件。输入如下内容：

```
vi cutlog.sh
```

```
1 D=$(date +%Y%m%d)
2
3
4
5 LOGS_PATH=/usr/local/nginx/logs
6
7
8
9 mv /usr/local/nginx/logs/access.log
10      ${LOGS_PATH}/${D}.log
11
12
13 kill -USR1 $(cat /usr/local/nginx/nginx.pid)
```

说明：

•

D=\$(date +%Y%m%d), 声明个变量，名为D（自定义），值为date (Linux自带的，类似于函数，用于获取当前时间，并且格式为年月日)

\${D}: 引用上面变量名为D的变量

\$(cat /usr/local/nginx/nginx.pid): cat命令意思是查看。nginx.pid: 存储的是nginx的主进程号。连起来的意思就是查看nginx主进程号，带上\$(xx)就是说拿到xx

•

(3) 定时执行某个文件，输入如下命令：(注意：运行crontab命令需要先用yum进行安装crontab，否则会出现command not found)

```
crontab -e
```

crontab: 定时执行某个文件

crontab -e: 编辑定时执行的内容

(4) 执行完 (3) 后会出现编辑器，输入如下内容：

```
23 59 * * * /bin/bash /usr/local/nginx/logs/cutlog.sh
```

含义：在每天23点59分定时执行cutlog.sh文件，这样就实现了每天定时切割日志文件了。

原理：只是将手动切割写成个脚本。

# Double

soa理念-->微服务

两者的本质还是对远程服务的复用,RPC框架

官方参考文档

Dubbo 分布式服务治理解决方案,不仅仅是远程服务通信，并且还解决了服务路由、负载、降级、容错等功能。(利用mock属性,缺少熔断)

Dubbo 能够支持的注册中心有：consul、etcd、**nacos**、sofa、**zookeeper**、**redis**、multicast

Dubbo 能够支持多协议,集成 Webservice 协议, REST 协议,http,9中协议

## 添加新的协议支持

```
1 添加新的协议支持
2 <dubbo:protocol name="rest" port="8888" server="jetty"/>
3 提供新的服务
4 @Path("/users")：指定访问 UserService 的 URL 相对路径
   是/users，即
5 http://localhost:8080/users
6 @Path("/register")：指定访问 registerUser()方法的 URL 相对路
   径是/register,
7 再结合上一个 @Path 为 UserService 指定的路径，则调用
8 UserService.register()的完整路径为
9 http://localhost:8080/users/register
10 @POST：指定访问 registerUser()用 HTTP POST 方法
11 @Consumes({MediaType.APPLICATION_JSON})：指定
   registerUser()接收
12 JSON 格式的数据。REST 框架会自动将 JSON 数据反序列化为 User 对
   象
13 （注解可以放在接口上，客户端需要根据接口的注解来进行解析和调用）
14 @Path("/user")
15 public interface UserService {
16     @GET
17     @Path("/register/{id}")
18     void register(@PathParam("id") int id);
19 }
```

客户端配置

```
20 <dubbo:protocol name="rest" port="8888" server="jetty"/>
21 <dubbo:reference id="userService"
22 interface="com.gupaoedu.practice.UserService"
23 protocol="rest"/>
24 在服务接口获得上下文的方式
25 第一种
26 笔记仅仅提供与课后复习，为了达到更好的学习效果，请自己进行梳理，转载请注明《咕泡学院》
27 HttpServletRequest
28 request=
29 (HttpServletRequest)RpcContext.getContext().getRequest();
30 第二种
31 通过注解的方式
32 @GET
33 @Path("/register/{id}")
34 void register(@PathParam("id") int id, @Context
35 HttpServletRequest request);
36 dubbo 监控
37 https://github.com/apache/dubbo-admin
38 Dubbo 里面提供了一种基于终端操作的方法来实现服务治理
39 使用 telnet localhost 20880 连接到服务对应的端口
```

## 负载均衡

默认就集成了负载均衡的算法和实现,提供了 4 中负载均衡实现

**RandomLoadBalance**,权重随机算法

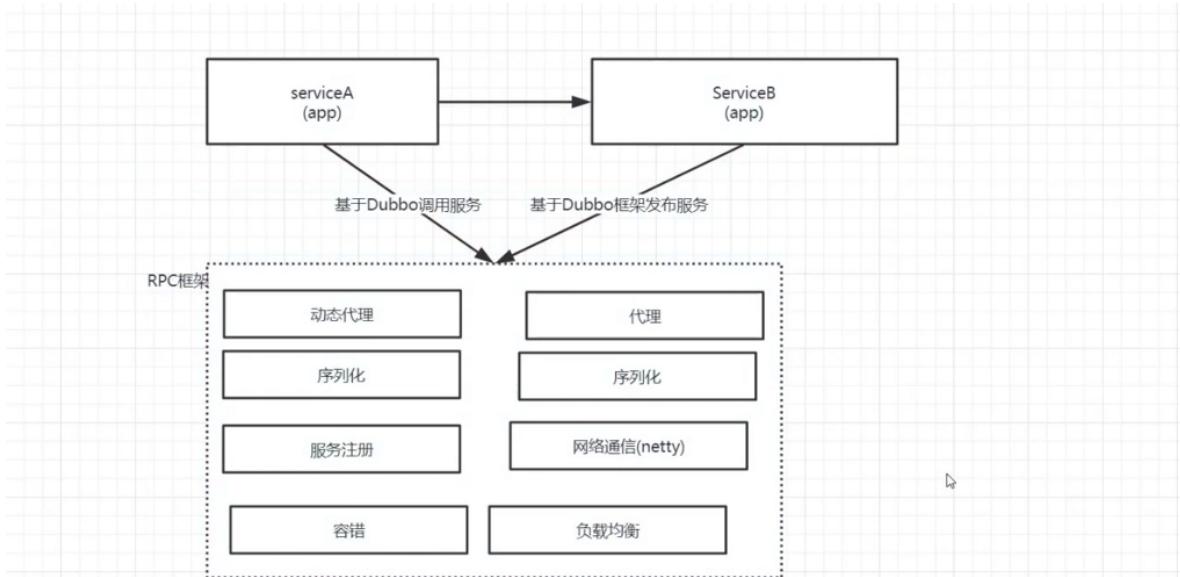
**LeastActiveLoadBalance** 最少活跃调用数算法，活跃调用数越小，表明该服务提供者效率越高，

每个服务提供者对应一个活跃数 active。初始情况下，所有服务提供者活跃数均为 0。每收到一个请求，活跃数加 1，完成请求后则将活跃数减 1。在服务运行一段时间后，**活跃数字越小代表效率越高**

**ConsistentHashLoadBalance** 一致性hash

当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动

**RoundRobinLoadBalance** 加权轮询算法



## 容错机制

`@Service(loadbalance = "random", cluster = "failsafe")`

**Failover Cluster**, 重试, 默认机制, 可通过 `retries="2"` 来设置重试次数(不含第一次), 查询语句容错策略

**Failsafe Cluster** 快速失败, 只发起一次调用, 失败立即报错; 增删改建议使  
幂等性: 就是用户对于同一操作发起的一次请求或者多次请求的结果是一致的, 不会因为多次点击而产生了副作用

**Failsafe Cluster** 失败安全, 出现异常时, 直接忽略; 通常用于写入审计日志等操作

**Fallback Cluster** 失败自动恢复, 后台记录失败请求, 定时重发; 通常用于消息通知操作

**Forking Cluster** 并行调用多个服务器, 只要一个成功即返回; 通常用于实时性要求较高的读操作, 但需要浪费更多服务资源; 可通过 `forks="2"` 来设置

**Broadcast Cluster** 广播调用所有提供者, 逐个调用, 任意一台报错则报错

## 6种容错

重试? -> failover (默认的情况) -?三次, retries=2()

不希望重试? 快速失败 failfast

失败之后, 记录日志. fallback

失败安全, failsafe

并行调用多个服务, forking

广播出去->任何一台报错就失败.(更新所有节点的缓存)

## 服务降级,通过mock操作,在客户端配置

```
public class DubboController {  
    //Dubbo提供的注解  
    @Reference(loadbalance = "roundrobin", timeout = 1, cluster = "failfast", mock = "com.gupaoedu.dubbo.client.SayHelloServiceMock")  
    ISayHelloService sayHelloService; //dubbo://
```

## 服务降级

当某个**非关键服务**出现错误时, 可以通过降级功能来临时屏蔽这个服务

场景:

人工降级, 在大促之前通过降级开关关闭哪些推荐内容、评价等对主流程没有影响的功能

故障降级, 比如调用的远程服务挂了, 网络故障、或者 RPC 服务返回异常。那么可以直接降级, 降级的方案比如设置默认值、采用兜底数据 (广告)

限流降级, 在秒杀这种流量比较集中并且流量特别大的情况下, 当达到阀值时, 后续的请求被降级, 比如进入排队页面, 比如跳转到错误页

Dubbo 中提供了一个 mock 的配置来实现服务降级

1.在客户端创建mock类

2.在服务端配置mock

```
@Reference(  
    loadbalance = "random",  
    mock =  
    "com.springboot.practice.springbootdubboclient.MockSayHelloService",  
    timeout = 1000,  
    cluster = "failfast")  
IHelloService helloService;
```

Dubbo 缺省会在启动时检查依赖的服务是否可用，不可用时会抛出异常，阻止 Spring 初始化完成，以便上线时，能及早发现问题，默认 check="true"。

可以通过 check="false" 关闭检查，比如，测试时，有些服务不关心，或者出现了循环依赖，必须有一方先启动。

➤ registry、reference、consumer 都可以配置 check 这个属性

## 多版本支持

当一个接口实现，出现不兼容升级时，可以用版本号过渡，版本号不同的服务相互间

不引用。

可以按照以下的步骤进行版本迁移：

- \1. 在低压力时间段，先升级一半提供者为新版本
- \2. 再将所有消费者升级为新版本
- \3. 然后将剩下的一半提供者升级为新版本

## 配置中心

2.7 版本中，可以用zookeeper做配置中心

**dubboadmin 添加配置可视化界面**

Dubbo2.7 中对元数据进行了改造，简单来说，就是把属于**服务治理的数据发布到注册中心**，其他的配置数据统一发布到元数据中心。这样一来大大降低了注册

中心的负载。

Dubbo+zookeeper+SpringBoot+Redis+MQ微服务架构实战开发\_哔哩哔哩  
( °- °)つ口 干杯~-bilibili <https://www.bilibili.com/video/BV1VJ411g7Zq>

## SPI 扩展点

spi主要是解决是为了解决因为类上层加载器无法去加载下层加载器这个问题

java spi标准

**实现 SPI 需要遵循的标准**

1. 需要在 classpath 下创建一个目录，该目录命名必须是：META-INF/service
2. 在该目录下创建一个 properties 文件，该文件需要满足以下几个条件
  - 2.1 文件名必须是扩展的接口的全路径名称
  - 2.2 文件内部描述的是该扩展接口的所有实现类
  - 2.3 文件的编码格式是 UTF-8
3. 通过 java.util.ServiceLoader 的加载机制来发现

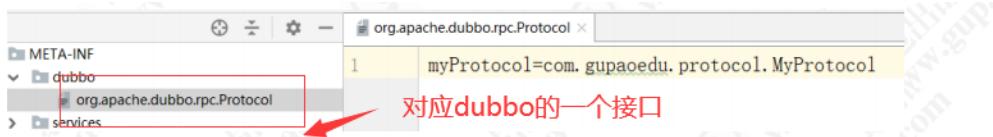


## SPI 的缺点

如果你在 META-INF/service 下的文件里面加了 N 个实现类，那么 JDK 启动的时候都会一次性全部加载。那么如果有的扩展点实现初始化很耗时或者如果有某些实现类并没有用到，那么会很浪费资源

## Dubbo 优化后的 SPI 机制

1. 需要在 resource 目录下配置 META-INF/dubbo 或者 META-INF/dubbo/internal 或者 META-INF/services，并基于 SPI 接口去创建一个文件
2. 文件名称和接口名称保持一致，文件内容和 SPI 有差异，内容是 KEY 对应 Value



### 3. 自定义接口的实现

```

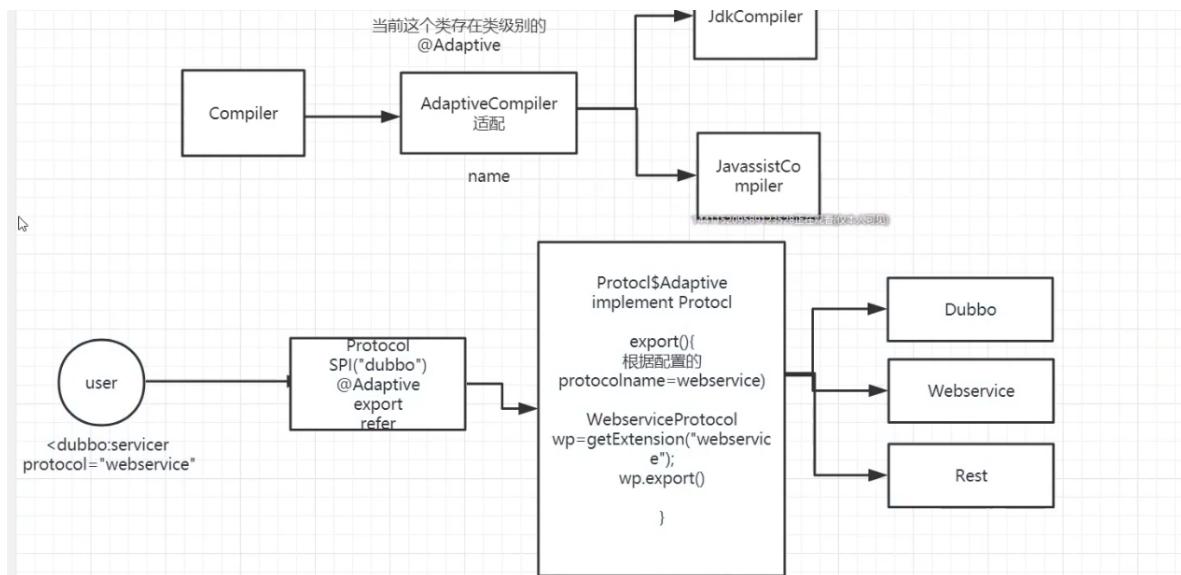
1 Protocol1
protocol=ExtensionLoader.getExtensionLoader(Protocol.class)
    .getExtension("myProtocol");
2 System.out.print(protocol.getDefaultPort), 输出结果
  
```

## dubbo增加了自适应扩展点

类级别 @Adaptive 放在类上，说明当前类是一个确定的自适应扩展点的类

方法级别 方法级别，那么需要生成一个动态字节码，来进行转发。

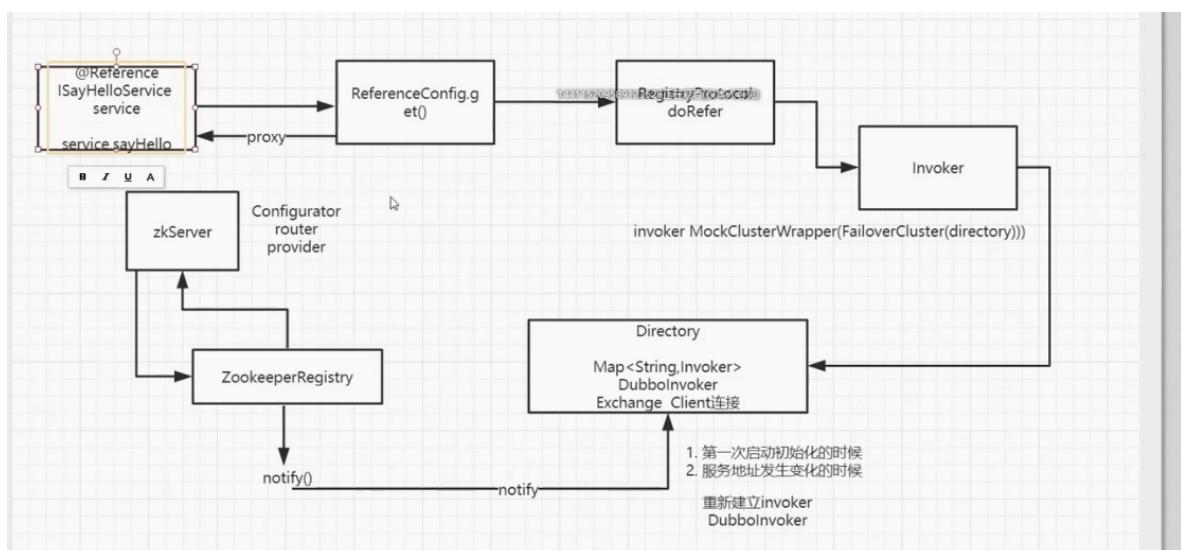
**@Activate 自动激活扩展点**, 可配置加载条件 激活扩展点 需要加入指定rule, 类似 springboot 的 condition

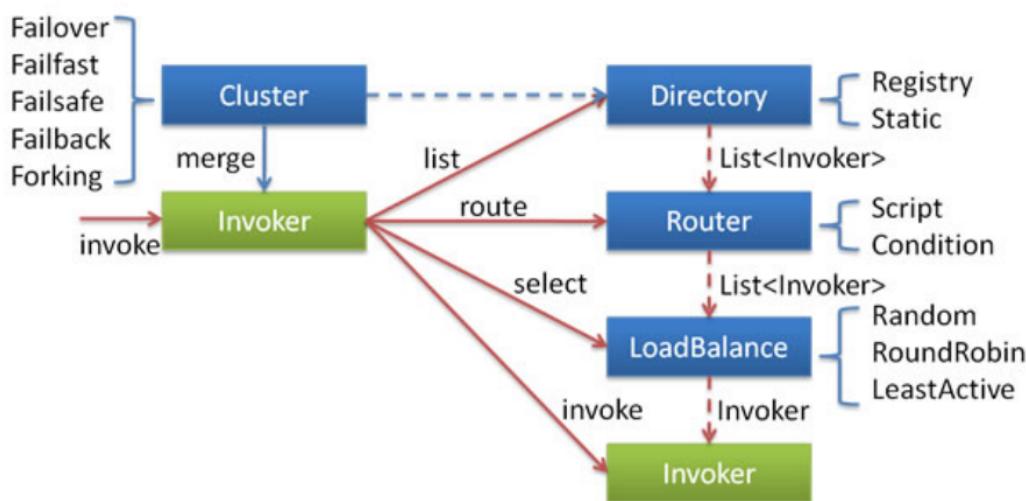
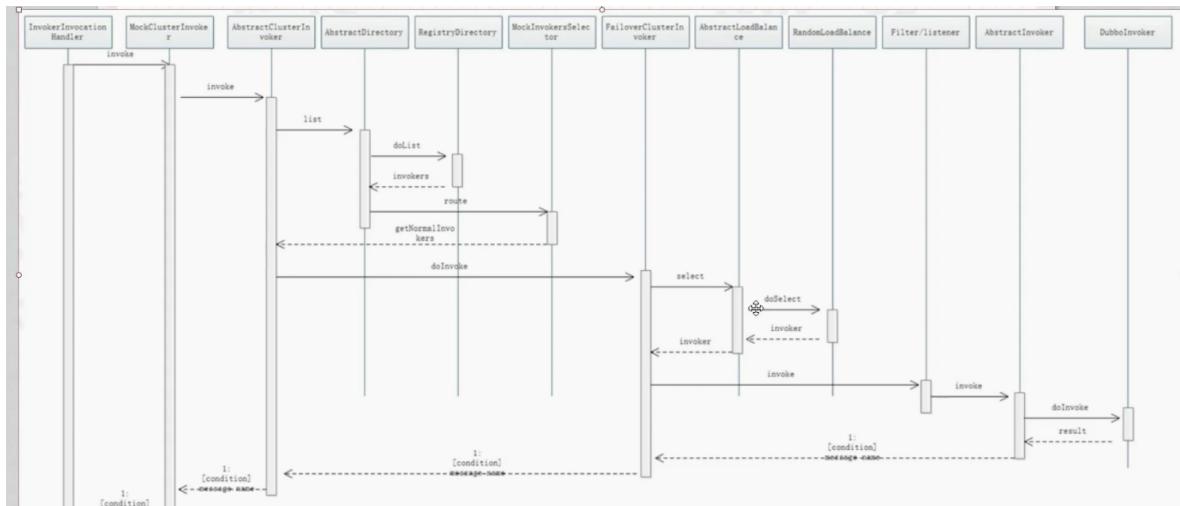


举个例子，我们可以看看 org.apache.dubbo.Filter 这个类，它有非常多的实现，比如说 CacheFilter，这个缓存过滤器，配置信息如下

group 表示客户端和和服务端都会加载，value 表示 url 中有 cache\_key 的时候

`@Activate(group = {CONSUMER, PROVIDER}, value = CACHE_KEY)`





这里的 Invoker 是 Provider 的一个可调用 Service 的抽象，Invoker 封装了 Provider 地址及 Service 接口信息

Directory 代表多个 Invoker，可以把它看成 List，但与 List 不同的是，它可能是动态变化的，比如注册中心推送变更

Cluster 将 Directory 中的多个 Invoker 伪装成一个 Invoker，对上层透明，伪装过程包含了容错逻辑，调用失败后，重试另一个 Router 负责从多个 Invoker 中按路由规则选出子集，比如读写分离，应用隔离等 LoadBalance 负责从多个 Invoker 中选出具体的一个用于本次调用，选的过程包含了负载均衡算法，调用失败后，需要重选

路由器的一个作用是**连通不同的网络**，另一个作用是**选择信息传送的线路**。选择通畅快捷的近路，能大大提高通信速度，减轻网络系统通信负荷，节约网络系统资源，提高网络系统畅通率，从而让网络系统发挥出更大的效益来

- 1 路由器的一个作用是连通不同的网络，另一个作用是选择信息传送的线路。选择通畅快捷的近路，能大大提高通信速度，减轻网络系统通信负荷，节约网络系统资源，提高网络系统畅通率，从而让网络系统发挥出更大的效益来

# 总结

服务的发布做了什么事情

1. 基于spring进行解析配置文件到config
2. 各种逻辑判断,保证配置信息安全性
3. 组装url(registry-->zookeeper"-->.dubbo://-->injvm)
4. 构建一个invoke(动态代理)
5. registryProtocol.exprot
6. 各种wrapper(fitter/qos(质量检测)/litter)
7. DubboProtocol 发布服务
8. 启动一个nettyserver

适配-->这对不同协议做不同的适配-->url  
(adaptiveExtension/Extension)

- 启动的时候 -> 注册端口监听、注册服务地址到注册中心
- 生成一个invoker的代理类，代理服务端的实现
- wrapper。对invoker进行包装、filter
- 从注册中心获得url地址->生成invoker ->

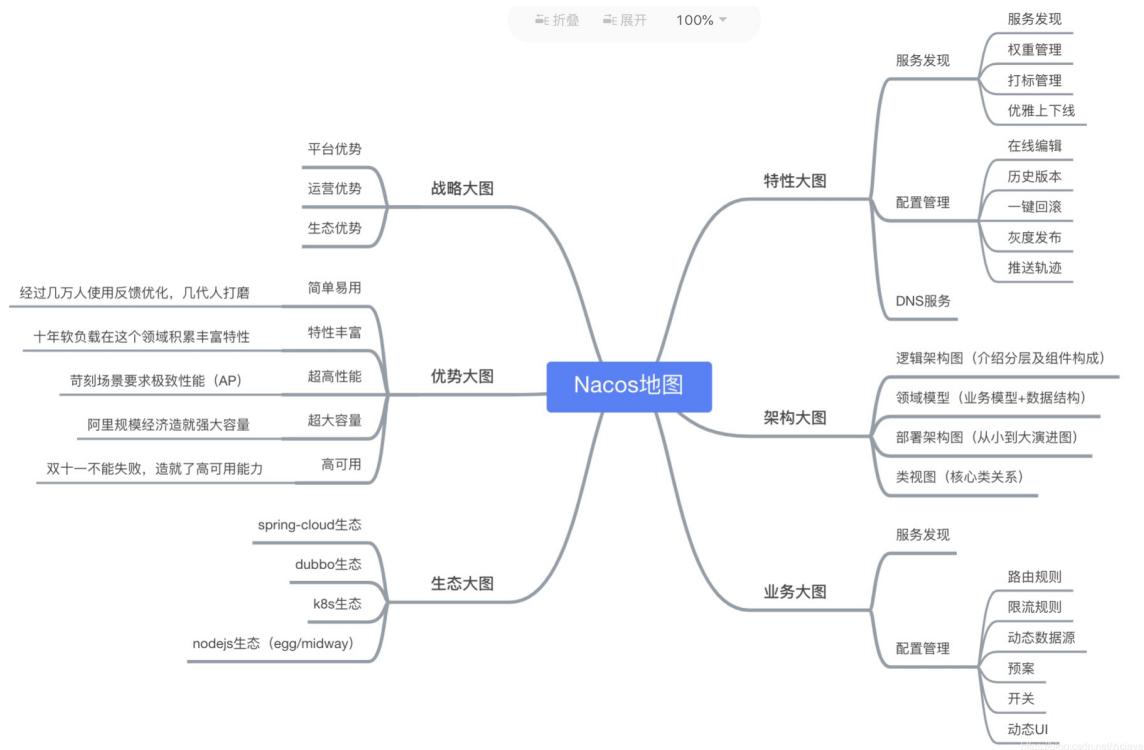
客户端

- 生成proxy\$0 ; ->handler
- Directory->StaticDirectory (静态) | RegistryDirectory(动态)->动态刷新目标服务地址的变化
- Mock (降级、测试mock)
- 参数组装 (RpcInvocation)
- 容错(failover)->默认是重试
- 路由
- 负载均衡-> loadbalance (路由机制(directory中拿到list))-> 筛选出唯一一个目标地址
- nettyClient进行远程通信

数据传输建议包100kb

## Nacos

可以做配置中心和注册中心,这里主要讲了配置中心,上面zookeeper主要讲了注册中心,



	Nacos	Eureka	Consul	CoreDNS	Zookeeper
一致性协议	CP+AP	AP	CP	—	CP
健康检查	TCP/HTTP/MYSQL/Client Beat	Client Beat	TCP/HTTP/gRPC/Cmd	—	Keep Alive
负载均衡策略	权重/ metadata/Selector	Ribbon	Fabio	RoundRobin	—
雪崩保护	有	有	无	无	无
自动注销实例	支持	支持	支持	不支持	支持
访问协议	HTTP/DNS	HTTP	HTTP/DNS	DNS	TCP
监听支持	支持	支持	支持	不支持	支持
多数据中心	支持	支持	支持	不支持	不支持
跨注册中心同步	支持	不支持	支持	不支持	不支持
SpringCloud集成	支持	支持	支持	不支持	支持
Dubbo集成	支持	不支持	支持	不支持	支持
K8S集成	支持	不支持	支持	支持	不支持

## 基本使用

启动服务端

nacos-config-spring-boot-starter ,坐标

配置

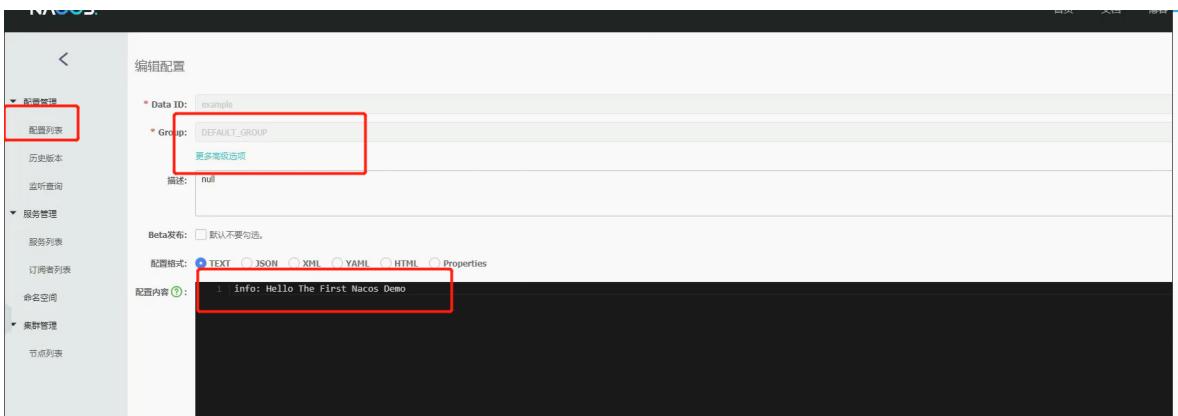
nacos.config.server-addr=

注解配置controller

```
/*
 * NacosPropertySource(dataId = "example", groupId = "DEFAULT_GROUP", autoRefreshed = true)
 */
@RestController
public class NacosConfigController {

    /**
     * 当前的info这个属性，回去nacos-server找到对应的info这个属性
     * 高可用性
     * hello Nacos表示本地属性（降级属性）
     */
    @NacosValue(value = "${info:hello Nacos}", autoRefreshed = true)
    private String info;

    @GetMapping("/get")
    public String get() {
        return info;
    }
}
```



就是类似springboot里的配置文件,只不过这里在页面可以动态实时更改,

响应的也会有权限问题,数据存储问题,

传输协议: open api ,sdk

sdk使用



## 实现一个配置中心设计思路

服务器端的配置保存 (持久化)

数据库

服务器端提供访问api

rpc、http (openapi)

ps,既然两种方式都可以实现远程调用,选择原则

速度来看，RPC要比http更快，虽然底层都是socket，但是http协议的信息往往比较臃肿

难度来看，RPC实现较为复杂，http相对比较简单

灵活性来看，http更胜一筹，因为它不关心实现细节，跨平台、跨语言。

因此，两者都有不同的使用场景：

如果对效率要求更高，并且开发过程使用统一的技术栈，那么用RPC还是不错的。

如果需要更加灵活，跨语言、跨平台，显然http更合适

## 数据变化之后如何通知到客户端

zookeeper (session manager)

zookeeper采用的是推拉结合的模式

- \1. 客户端订阅znode节点
- \2. 被订阅的节点发生变化后，zookeeper服务端向客户端发生数据变更的watcher事件通知
- \3. 客户端接收到watcher通知后，主动从服务端拉取变更的数据

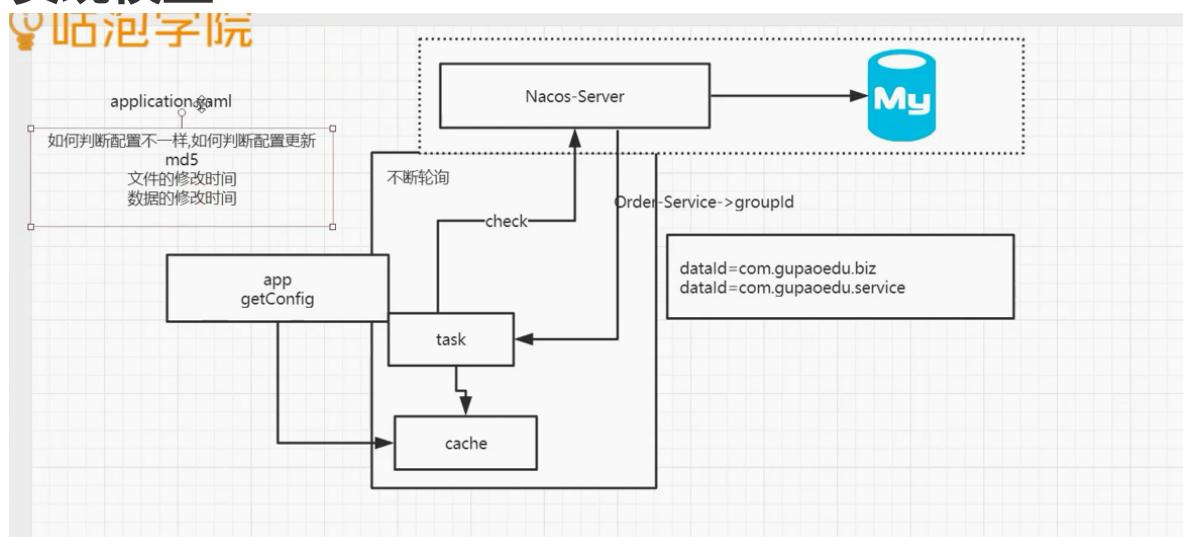
push (服务端主动推送到客户端) 、 pull(客户端主动拉去数据) ? -> 长轮训(pull数据量很大会怎么办)

## 客户端如何去获得远程服务的数据 ()

### 安全性 ()

刷盘（本地缓存）->

## 实现模型



客户端会缓存到本地,动态监控,每次批次3000,异步动态比较的是MD5加密后的值

```
//做了分批处理

public void checkConfigInfo() {
    // 分任务
    int listenerSize = cacheMap.get().size();
    // 向上取整为批数
    int longingTaskCount = (int) Math.ceil(listenerSize / ParamUtil.getPerTaskConfigSize());
    if (longingTaskCount > currentLongingTaskCount) {
        for (int i = (int) currentLongingTaskCount; i < longingTaskCount; i++) {
            // 要判断任务是否在执行 这块需要好好想想。 任务列表现在是无序的。变化过程可能有问题
            executorService.execute(new LongPollingRunnable(i));
        }
        currentLongingTaskCount = longingTaskCount;
    }
}
```

检查本地配置

监听->

```
cacheData.isUseLocalConfigInfo()
```

检查缓存的md5

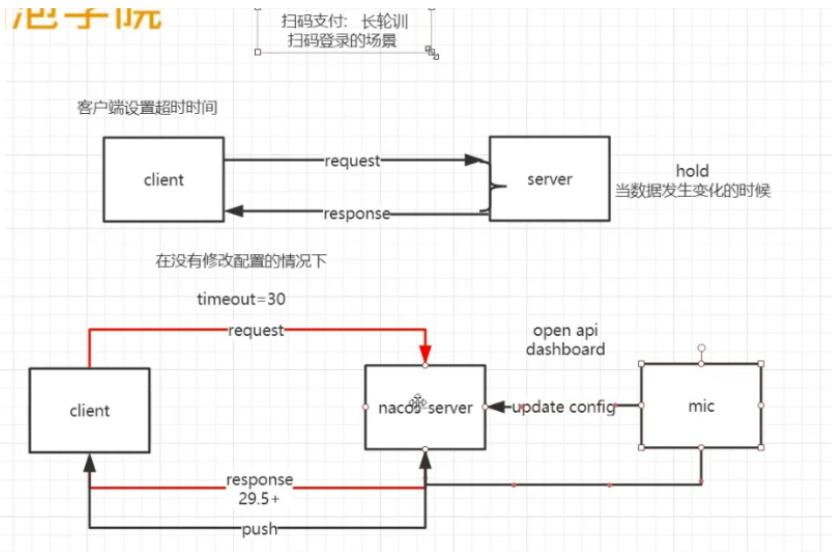
检查文件的更新时间

```
checkLocalConfig(cacheData);
if (cacheData.isUseLocalConfigInfo()) {
    cacheData.checkListenerMd5();
```

groupid+dataid+tenant

简单总结一下刚刚分析的整个过程。

客户端发起**长轮训请求**服务端收到请求以后，先比较服务端缓存中的数据是否相同，如果不同，则直接返回；如果相同，则通过schedule**延迟29.5s之后**再执行比较为了保证当服务端在29.5s之内发生数据变化能够及时通知给客户端，服务端采用事件订阅的方式来监听服务端本地数据变化的事件，一旦收到事件，则触发DataChangeTask的通知，并且遍历allStubs队列中的ClientLongPolling，把结果写回到客户端，就完成了一次数据的推送。如果DataChangeTask任务完成了数据的“推送”之后，ClientLongPolling中的调度任务又开始执行了怎么办呢？很简单，只要在进行“推送”操作之前，先将原来等待执行的调度任务取消掉就可以了，这样就防止了推送操作写完响应数据之后，调度任务又去写响应数据，这时肯定会报错的。所以，在ClientLongPolling方法中，最开始的一个步骤就是删除订阅事件。所以总的来说，**Nacos采用推+拉的形式**，来解决最开始关于长轮训时间间隔的问题。当然，30s这个时间是可以设置的，而之所以定30s，应该是一个经验值。



## 集群选举问题

Nacos支持集群模式，很显然。而一旦涉及到集群，就涉及到主从，那么nacos是一种什么样的机制来实现的集群呢？

nacos的集群类似于zookeeper，它分为leader角色和follower角色，那么从这个角色的名字可以看出来，这个集群存在选举的机制。因为如果自己不具备选举功能，角色的命名可能就是master/slave了，当然这只是我基于这么多组件的命名的一个猜测

### raft(共识算法)

图解<http://thesecretlivesofdata.com/raft/>

Nacos集群采用**raft算法**来实现，它是相对zookeeper的选举算法较为简单的一种。选举算法的核心在 RaftCore 中，包括数据的处理和数据同步

raft算法演示地址在Raft中，节点有三种角色： Leader：负责接收客户端的请求；Candidate：用于选举Leader的一种角色；Follower：负责响应来自Leader或者选举分为两个节点

服务启动的时候

leader挂了的时候

所有节点启动的时候，都是follower状态。如果在一段时间内如果没有收到leader的心跳（可能是没有leader，也可能是leader挂了），那么follower会变成Candidate。然后发起选举，选举之前，会增加term(周期,朝代)，这个term和zookeeper中的epoch的道理是一样的。follower会投自己一票，并且给其他节点发送票据vote，等到其他节点回复

在这个过程中，可能出现几种情况收到过半的票数通过，则成为leader被告知其他节点已经成为leader，则自己切换为follower 一段时间内没有收到过半的投票，则重新发起选举

### **约束条件在任一term中，单个节点最多只能投一票**

选举的几种情况

第一种情况，赢得选举之后，leader会给所有节点发送消息，避免其他节点触发新的选举

第二种情况，比如有三个节点A B C。A B同时发起选举，而A的选举消息先到达C，C给A投了一

票，当B的消息到达C时，已经不能满足上面提到的第一个约束，即C不会给B投票，而A和B显然都

不会给对方投票。A胜出之后，会给B,C发心跳消息，节点B发现节点A的term不低于自己的term，

知道已经有Leader了，于是转换成follower

第三种情况，没有任何节点获得majority投票，可能是平票的情况。加入总共有四个节点

(A/B/C/D) , Node C、Node D同时成为了candidate，但Node A投了NodeD一票，NodeB投

了Node C一票，这就出现了平票 split vote的情况。这个时候大家都在等啊等，直到超时后重新发

起选举。如果出现平票的情况，那么就延长了系统不可用的时间,因此raft引入了randomized election timeouts(150-300 ms)来尽量避免平票情况

两个超时,心跳超时判断主节点挂了,变成候选人发送投票,,

选举超时,平票,进行下一轮操作

### **数据的处理**

对于事务操作，请求会转发给leader,非事务操作上，可以任意一个节点来处理

如果当前的节点不是leader，则转发给leader节点处理,如果是，则向所有节点发送onPublish数据保存到日志

## sentinel限流

### 限流的基本认识

<https://github.com/alibaba/Sentinel/wiki>

#### 场景分析

一个互联网产品，打算搞一次大促来增加销量以及曝光。公司的架构师基于往期的流量情况做了一个活动流量的预估，然后整个公司的各个技术团队开始按照这个目标进行设计和优化，最终在大家不懈的努力之下，达到了链路压测的目标流量峰值。到了活动开始那天，大家都在盯着监控面板，看着流量像洪水一样涌进来。由于前期的宣传工作做得很好，使得这个流量远远超过预估的峰值，后端服务开始不稳定，CPU、内存各种爆表。部分服务开始出现无响应的情况。最后，整个系统开始崩溃，用户无法正常访问服务。最后导致公司巨大的损失

#### 引入限流

在10.1黄金周，各大旅游景点都是人满为患。所有有些景点为了避免出现踩踏事故，会采取限流措施。那在架构场景中，是不是也能这么做呢？针对这个场景，能不能够设置一个最大的流量限制，如果超过这个流量，我们就拒绝提供服务，从而使得我们的服务不会挂掉。当然，限流虽然能够保护系统不被压垮，但是对于被限流的用户，就会很不开心。所以限流其实是一种有损的解决方案。但是相比于全部不可用，有损服务是最好的一种解决办法

#### 限流的作用

除了前面说的限流使用场景之外，限流的设计还能防止恶意请求流量、恶意攻击。所以，限流的基本原理是通过对并发访问/请求进行限速或者一个时间窗口内的请求进行限速来保护系统，一旦达到限制速率则可以拒绝服务（定向到错误页或者告知资源没有了）、排队或等待(秒杀、下单)、降级（返回兜底数据或默认数据或默认数据，如商品详情页库存默认有货）一般互联网企业常见的限流有：限制总并发数（如数据库连接池、线程池）、限制瞬时并发数（nginx limit\_conn模块，用来限制瞬时并发连接数）、限制时间窗口内的平均速率（如Guava的RateLimiter、nginx的limit\_req模块，限制每秒的平均速率）；其他的还有限制远程接口调用速率、限制MQ的消费速率。另外还可以根据网络连接数、网络流量、CPU或内存负载等来限流。有了限流，就意味着在处理

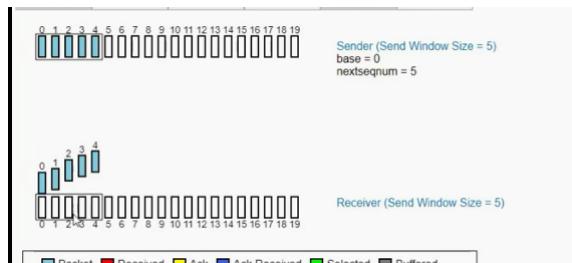
高并发的时候多了一种保护机制，不用担心瞬间流量导致系统挂掉或雪崩，最终做到有损服务而不是不服务



## 常见限流算法

### 滑动窗口

基于tcp的



### 漏桶

桶本身具有一个恒定的速率往下漏水，而上方时快时慢的会有水进入桶内。当桶还未满时，上方的水可以加入。一旦水满，上方的水就无法加入。

在桶满水之后，常见的两种处理方式为：

- 1) 暂时拦截住上方水的向下流动，等待桶中的一部分水漏走后，再放行上方水。
- 2) 溢出的上方水直接抛弃。

### 特点

1. 漏水的速率是固定的
2. 即使存在突然注水量变大的情况，漏水的速率也是固定的

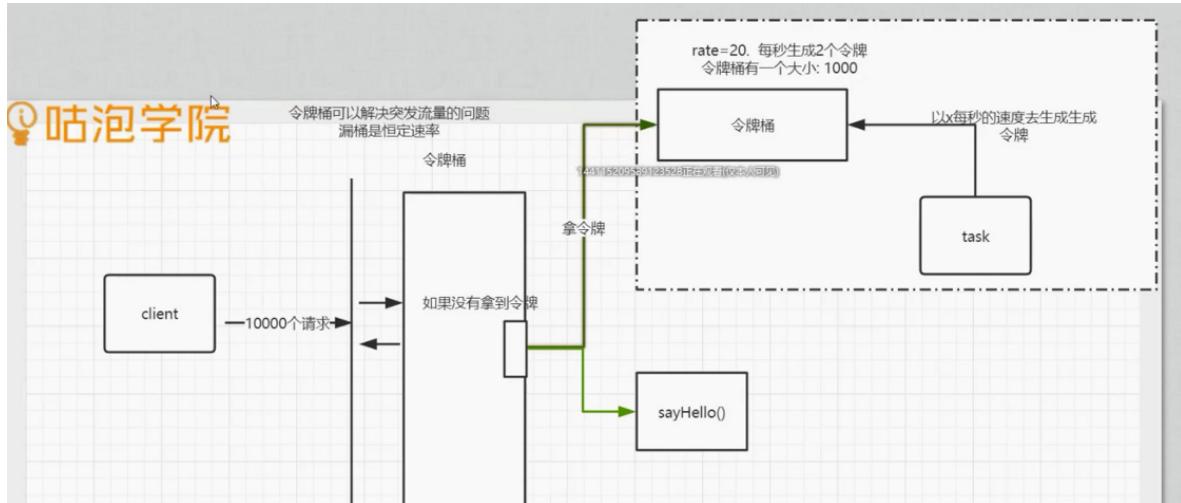
### 令牌桶(能够解决突发流量)

令牌桶算法是网络流量整形 (Traffic Shaping) 和速率限制 (Rate Limiting) 中最常使用的一种算法。

典型情况下，令牌桶算法用来控制发送到网络上的数据的数目，并允许突发数据的发送。

令牌桶是一个存放固定容量令牌（token）的桶，按照固定速率往桶里添加令牌；令牌桶算法实际上由三

部分组成：两个流和一个桶，分别是令牌流、数据流和令牌桶



## 令牌流与令牌桶

系统会以一定的速度生成令牌，并将其放置到令牌桶中，可以将令牌桶想象成一个缓冲区（可以用队列

这种数据结构来实现），当缓冲区填满的时候，新生成的令牌会被扔掉。这里有两个变量很重要：

第一个是生成令牌的速度，一般称为 rate。比如，我们设定 rate = 2，即每秒钟生成 2 个令牌，也就是每 1/2 秒生成一个令牌；

第二个是令牌桶的大小，一般称为 burst。比如，我们设定 burst = 10，即令牌桶最大只能容纳 10 个令牌。

## 常用算法比较

算法	确定参数	空间复杂度	时间复杂度	限制突发流量	平滑限流	分布式环境下实现难度
固定窗口	计数周期T、周期内最大访问数N	低O(1) (记录周期内访问次数及周期开始时间)	低O(1)	否	否	低
滑动窗口	计数周期T、周期内最大访问数N	高O(N) (记录每个小周期中的访问数量)	中O(N)	是	相对实现。滑动窗口的格子划分的越多，那么滑动窗口的滚动就越平滑	中
漏桶	漏桶流出速度r、漏桶容量N	低O(1) (记录当前漏桶中容量)	高O(N)	是	是	高
令牌桶	令牌产生速度r、令牌桶容量N	低O(1) (记录当前令牌桶中令牌数)	高O(N)	是	是	高

## 基本使用

引入依赖,定义限流规则,根据规则使用



```

<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-core</artifactId>
    <version>1.6.3</version>
</dependency>

private static void initFlowRules() {
    List<FlowRule> rules=new ArrayList<>(); //限流规则的集合
    FlowRule flowRule=new FlowRule();
    flowRule.setResource("doTest");//资源(方法名称、接口)
    flowRule.setGrade(RuleConstant.FLOW_GRADE_QPS); //限流的阈值的类型
    flowRule.setCount(10);
    rules.add(flowRule);
    FlowRuleManager.loadRules(rules);
}

public static void main(String[] args) {
    initFlowRules(); //初始化一个规则

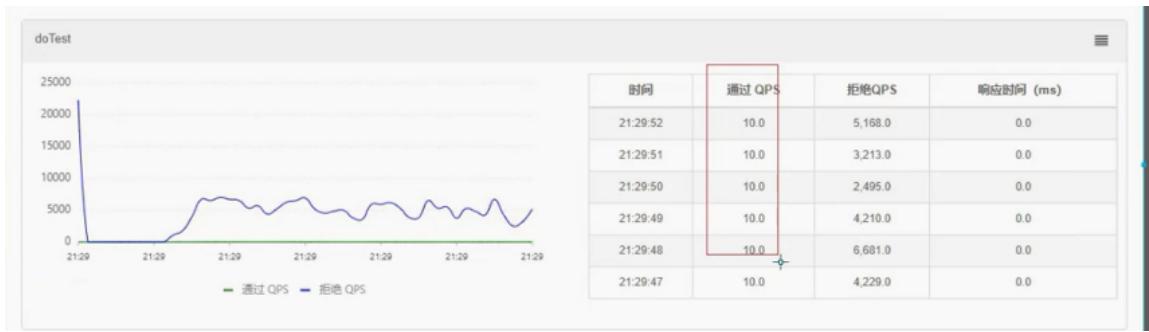
    while(true){
        Entry entry=null;
        try{
            entry= SphU.entry( name: "doTest");
            System.out.println("Hello Word");
        }catch (BlockException e){//如果被限流了，那么会抛出这个异常
            e.printStackTrace();
        }finally {
            if(entry!=null){
                entry.exit();//释放
            }
        }
    }
}

```

启动对应的控制台,方便监控GitHub上面下载

//定义新的端口和地址,起名字,监控自己

```
java -jar -Dserver.port=8888 -  
Dcsp.sentinel.dashboard.server=localhost:8888 -Dproject.name=sentinel-  
dashboard,xxx
```

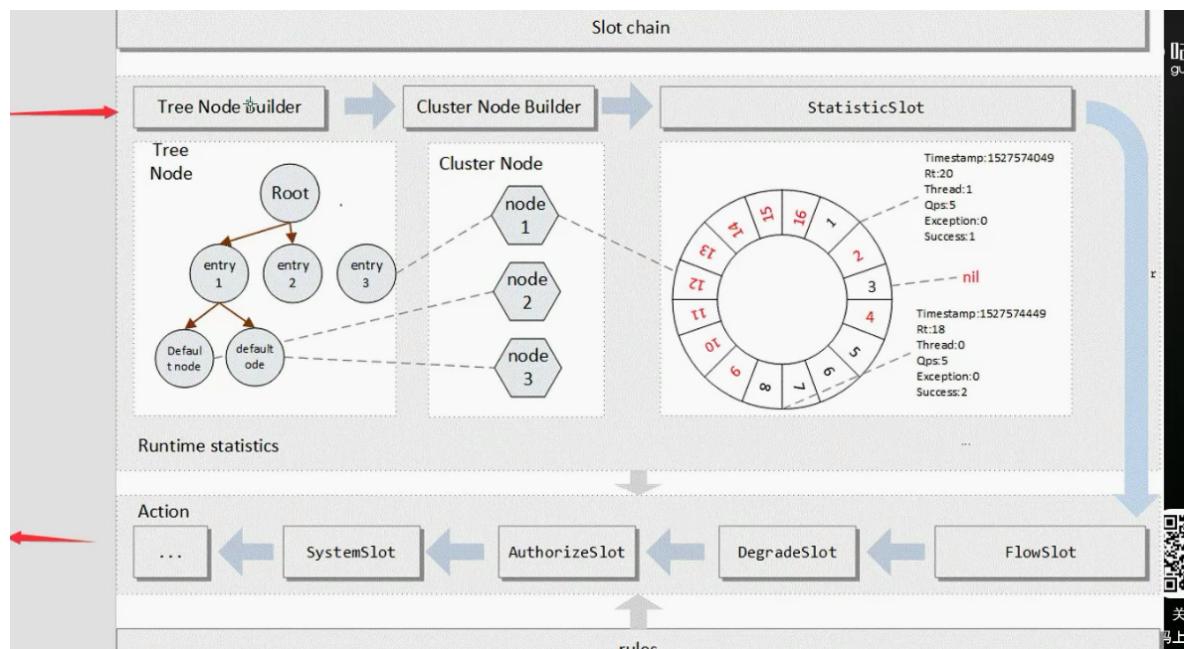


## sentinel实现限流原理

1.sentinel使用滑动窗口算法

2利用职责链的调用

3.spi扩展点



## 压力测试

[https://blog.csdn.net/weixin\\_39089928/article/details/87369101](https://blog.csdn.net/weixin_39089928/article/details/87369101)

Jmeter ,Apache提供烦人免费的工具,参看上面操作

## sentinel 熔断

发生在客户端,服务——一定时间出现一定的错误,客户端停止服务一段时间

## 基本使用

安装控制台 `Sentinel Dashboard`

```
1 project:
2   name: 在控制台显示的项目名
3 spring:
4   cloud:
5     sentinel:
6       transport:
7         dashboard: 192.168.1.154:8088
8 依赖
9 <dependency>
10   <groupId>org.springframework.cloud</groupId>
11   <artifactId>spring-cloud-starter-alibaba-
12     sentinel</artifactId>
13   <version>0.2.0.RELEASE</version>
14 </dependency>
15 持久化限流规则
16 在项目的resources下新增目录META-
17 INF\services\com.alibaba.csp.sentinel.init.InitFunc, 并填
18 写以上类的完成类名如
19 com.sumscope.study.springboot2.service.FileDataSourceInit
20 即可, 这样, 在您的classpath目录下通过
21 DegradeRule.json, FlowRule.json, SystemRule.json分别来定义降
22 级规则, 流控规则和系统规则, 比如我们定义一个流控规则, 让test资源
23 QPS为1, 即1秒钟最多调用1次, 如下:
24
25
26
27
28 [
```

```
16
17
18 [
19   {
20     "resource": "test",
21     "controlBehavior": 0,
22     "count": 1,
23     "grade": 1,
24     "limitApp": "default",
25     "strategy": 0
26   }
27 ]
28 ]
```

# Kafka 做消息中间件

## 基本概念

以kafka一开始设计的目标就是作为一个分布式、高吞吐量的消息系统，所以适合运用在大数据传输场景。这里主要研究kafka作为分布式消息中间件

特点:吞吐量、内置分区、冗余及容错性的优点(kafka每秒可以处理几十万消息)

## 消息中间件设计思路

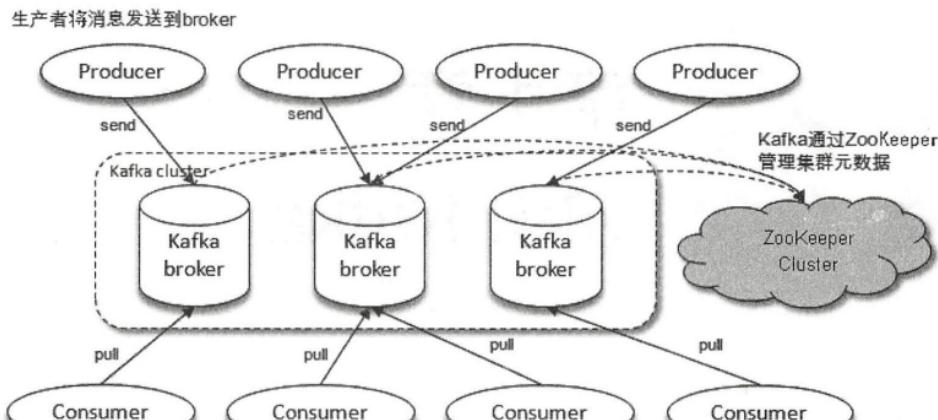
### 基本需求

- 实现消息的发送和接收。NIO通信(序列化/反序列化) ->dubbo->序列化；avro (avro) ; protobuf() / zk (jute) ...
- 实现消息存储的需求 (持久化、非持久化) 数据库的存储、文件存储(磁盘；顺序读写、页缓存、持久化的时机、零拷贝)、内存
- 是否支持跨语言 (多语言生态)
- 消息的确认 (确认机制) -> 业务逻辑需求
- 支持集群(master选举)->自己实现选举、第三方的实现 (zk)

### 高级需求

- 消息是否支持有序(业务逻辑)
- 是否支持事务消息 (业务逻辑) ->最终一致性
- 是否能够支持高并发和大数据的存储->
- 是否支持可靠性存储
- 是否支持多协议.
- 是否要收费.

Consumer 负责从 Broker 订阅并消费消息。



消费者采用拉 (pull) 模式订阅并消费消息

图 1-1 Kafka 体系结构

**消息系统：**afka 和传统的消息系统（也称作消息中间件）都具备系统解码、冗余存储、流量峰、缓冲、异步通信、扩展性、可恢复性等功能。与此同时，Kafka 供了大多数消息系统难以实现的**消息顺序性保障及回溯消费的功能**

**存储系统：**Kafka 把消息持久化到磁盘，相比于其他基于内存存储的系统而言，有效地降低了数据丢失的风险也正是得益于 Kafka 的消息持久化功能和多副本机制，我们可以把 Kafka 作为长期的数据存储系统来使用，只需要把对应的数据保留策略设置为“永久”或启用主题的日志压缩功能即可

**流式处理平台：**Kafka 不仅为每个流行的流式处理框架提供了可靠的数据来源，还供了一个完整的流式处理类库，比如窗口、连接、变换和聚合等各类操

**topic:**主体,逻辑上的同一,broker不同节点存储topic数据.默认分到一个partition上面,可以设置参数修改

**GroupID :**消费者的组织,在同一GroupId下只能消费一次,指定对应的分片,是靠coodiantor操作

**broker:**节点,kafka实例,多个组成集群

**Partition:**分区,一个broker可以有多个Partition

**Replica机制 :**保证数据的可靠,每个分区上有多个副本,leader和follower

**batch.size:**生产者发送多个消息到broker上的同一个分区时，异步发送,为了减少网络请求带来的性能开销，通过批量的方式来提交消息，可以通过这个参数来控制批量提交的字节数大小，默认大小是16384byte,也就是**16kb**,意味着当一批消息大小达到指定的batch.size的时候会统一发送

## Kafka安装

1.下载二进制压缩包,2.0内置zk,搭建集群建议使用外部zk

2.解压,先启动zookeeper,内置了zookeeper,可以用外部的,使用外部的需要修改配置文件,填写zookeeperip地址,修改监听地址

advertised.listeners=PLAINTEXT:你的IP地址,修改brokerid,每个节点唯一

3.java语言开发的,直接在bin目录下启动脚本

进入bin目录启动 `./kafka-server-start.sh ..//config/server.properties`

创建topic `./kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test3`

参考博客

<https://blog.csdn.net/sculpta/article/details/107871345>

## Springboot+kafka

```
1 <dependency>
2   <groupId>org.springframework.kafka</groupId>
3   <artifactId>spring-kafka</artifactId>
4   <version>2.2.0.RELEASE</version>
5 </dependency>
6 //生产者
7 @Component public class KafkaProducer
8 {
9     @Autowired
10    private KafkaTemplate<String, String>
11    kafkaTemplate;
12    public void send(){
13
14        kafkaTemplate.send("test", "msgKey", "msgData");
15    }
16 //消费者
17 @Component public class KafkaConsumer {
18     @KafkaListener(topics = {"test"})
19     public void listener(ConsumerRecord record){
20         Optional<?>
21         msg=Optional.ofNullable(record.value());
22         if(msg.isPresent()){
23             System.out.println(msg.get());
24         }
25     }
26     spring.kafka.bootstrap-
servers=192.168.13.102:9092,192.168.13.103:9092,192.168.1
3.104:9092 //卡夫卡地址
27     spring.kafka.producer.key-
serializer=org.apache.kafka.common.serialization.StringSe
rializer //key序列化
```

```
27     spring.kafka.producer.value-
  serializer=org.apache.kafka.common.serialization.StringSe
  rializer //value序列化
28     spring.kafka.consumer.group-id=test-consumer-group
29     spring.kafka.consumer.auto-offset-reset=earliest
30     auto.offset.reset
31 这个参数是针对新的groupid中的消费者而言的，当有新groupid的消费者
  来消费指定的topic时，对于
32 该参数的配置，会有不同的语义
33 auto.offset.reset=latest情况下，新的消费者将会从其他消费者最后
  消费的offset处开始消费Topic下的
34 消息
35 auto.offset.reset= earliest情况下，新的消费者会从该topic最早
  的消息开始消费
36 auto.offset.reset=none情况下，新的消费者加入以后，由于之前不存
  在offset，则会直接抛出异常。
37     spring.kafka.consumer.enable-auto-commit=true //自动提交
38     spring.kafka.consumer.key-
  deserializer=org.apache.kafka.common.serialization.String
  Deserializer spring.kafka.consumer.value-
  deserializer=org.apache.kafka.common.serialization.String
  Deserializer
```

## kafka消息分发策略

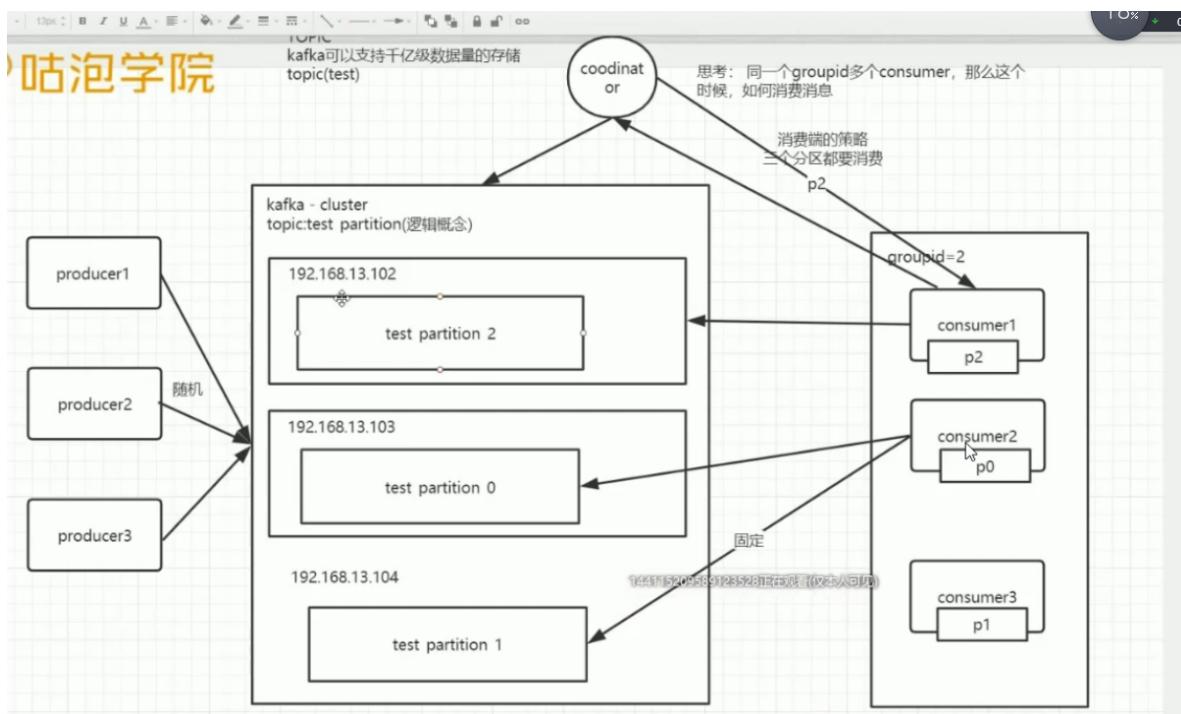
消息是kafka中最基本的数据单元，在kafka中，一条消息由key、value两部分构成，在发送一条消息时，我们可以指定这个key，那么producer会根据key和partition机制来判断当前这条消息应该发送并存储到哪个partition中。我们可以根据需要进行扩展producer的partition机制。

### 消息默认的分发机制

默认情况下，**kafka采用的是hash取模的分区算法**。如果Key为null，则会随机分配一个分区。这个随机是在这个参数“metadata.max.age.ms”的时间范围内随机选择一个。对于这个时间段内，如果key为null，则只会发送到唯一的分区。这个值默认情况下是10s更新一次。关于Metadata，这个之前没讲过，简单理解就是Topic/Partition和broker的映射关系，每一个topic的每一个partition，需要知道对应的broker列表是什么，leader是谁、follower是谁。这些信息都是存储在Metadata这个类里面

### consumer和partition的数量建议

1. 如果consumer比partition多，是浪费，因为kafka的设计是在一个**partition上是不允许并发的**，所以consumer数不要大于partition数
2. 如果consumer比partition少，一个consumer会对应于多个partitions，这里主要合理分配consumer数和partition数，否则会导致partition里面的数据被取的不均匀。**最好partiton数目是consumer数目的整数倍**，所以partition数目很重要，比如取24，就很容易设定consumer数目
3. 如果consumer从多个partition读到数据，不保证数据间的顺序性，kafka**只保证在一个partition上数据是有序的**，但多个partition，根据你读的顺序会有不同
4. 增减consumer，broker，partition会导致rebalance，所以rebalance后consumer对应的partition会发生变化



## 分区分配策略

存在三种分区分配策略，一种是**Range(默认)**、另一种是**RoundRobin (轮询)**、**StickyAssignor(粘性)**。在消费端中的ConsumerConfig中，通过这个属性来指定分区分配策略 `PARTITION_ASSIGNMENT_STRATEGY_CONFIG`

### RangeAssignor (范围分区)

Range策略是对每个主题而言的，首先对同一个主题里面的分区按照序号进行排序，并对消费者按照字母顺序进行排序

### RoundRobinAssignor (轮询分区)

轮询分区策略是把所有partition和所有consumer线程都列出来，然后按照hashcode进行排序。最后通过轮询算法分配partition给消费线程。如果所有consumer实例的订阅是相同的，那么partition会均匀分布。

### StrickyAssignor 粘性分配策略

分区的分配尽可能的均匀;分区的分配尽可能和上次分配保持相同  
当两者发生冲突时，第一个目标优先于第二个目标

Kafka只保证对ISR集合中的所有副本保证完全同步，处于ISR内部的follower都是可以和leader进行同步的，一旦出现故障或延迟，就会被踢出ISR

## coordinator

消费者向kafka集群中的任意一个broker发送一个GroupCoordinatorRequest请求，服务端会返回一个负载最小的broker节点的id，并将该broker设置为coordinator

在rebalance之前，需要保证coordinator是已经确定好了的，整个rebalance的过程分为两个步骤，Join和Sync

**join:** 表示加入到consumer group中，在这一步中，所有的成员都会向coordinator发送joinGroup的请求。一旦所有成员都发送了joinGroup请求，那么coordinator会选择一个**consumer担任leader角色**，并把组成员信息和订阅信息发送消费者

leader选举算法比较简单，如果消费组内没有leader，那么第一个加入消费组的消费者就是消费者leader，如果这个时候leader消费者退出了消费组，那么重新选举一个leader，这个**选举很随意，类似于随机算法**,consumerleader选用号策略发个broker,broker再分别发给所有consumer确定好统一策略避免冲突,减少并发

## 如何保存消费端的消费位置

kafka通过offset保证消息在分区内的顺序，offset的顺序不跨分区，即kafka只保证在同一个分区内的消息是有序的；对于应用层的消费来说，每次消费一个消息并且提交以后，会保存当前消费到的最近的一个offset.

在kafka中，提供了一个**consumer\_offsets** 的一个topic，把offset信息写入到这个topic中。 consumer\_offsets——按保存了每个consumer group某一时刻提交的offset信息。 \_\_consumer\_offsets 默认有50个分区。