学习好好学，加油！珍惜来自不易的机会

即便公司没有用到某项技术，该技术很优秀，如果只因为当前公司没有使用，就不去学习，这是非常错误的做法，不要将自己的技术能力和视野仅仅限于自己当下所处的这个环境中，应该着眼于未来，着眼于整个行业里。

日复一日，导致很多开发者就以为这就是技术的全部，同时在重复的体力劳动中也丧失了对技术细节的深入探讨。

Gradle是一款最新的，功能最强大的构建工具，

使用程序代替传统的XML配置，项目构建更加灵活。

丰富的第三方插件库。

提升自动化构建技术深度。

1.1.1 gradle

gradle相关概念介绍：

Gradle核心语法讲解及实战：

1.字符串特殊用法

2.Gradle常见数据结构（list、map，range）使用

3.Gradle面向对象特性

Gradle高级用法（重要点）

1、json我那件处理及json，model互转

2、gradle对xml文件读取和生成

3、普通文件的读取

4、网络请求json并转化为实体对象实现

5、文件下载功能实现

Gradle核心之project详解及实战

Project类核心作用 ，版本统一管理脚本编写

核心API讲解 Project类的源码

Gradle生命周期流程

Gradle核心之task详解与实战

Task定义和使用，Task执行流程

Task依赖关系与输入输出，Task继承与实现

Task修改默认构建流程，Task源码解读

综合实战：自动化生成版本说明xml文档

Gradle核心之与其它模块详解及实战

第三方依赖管理及gradle如何去处理依赖原理讲解

工程初始化核心类Setting类作用及自定义

源码管理类SourceSet讲解及实际工作中的妙用

Gradle核心之自定义plugin实战：

插件类Plugin的定义和如何使用第三方插件

Gradle如何管理插件的依赖

插件类Plugin源码解读

综合实战：将前面实现的自动化脚本封装为插件供他人使用。

Gradle程序修改默认打包流程：

java工程打包流程讲解，

将脚本嵌入到Gradle打包流程中实现我们的特定功能

打包流程核心task图解

领域特定语言DSL介绍：

DSL全称domain specific language

常见的DSL语言：XML、UML、Groovy为了解决特定领域的语言

核心思想：求专不求全

groovy语言是一种基于JVM的敏捷开发语言，结合了Python、Ruby和Smalltalk的许多强大的特性。

使用groovy可以与java完美结合，而且可以使用java所有的库，比如

groovy特性，语法上支持动态类型，闭包等新一代语言特性

无缝集成所有已经存在的java类库。

即支持面向对象也支持面向过程编程。

groovy优势，一种更加敏捷的编程语言，入门非常的容易，功能非常强大，既可以作为编程语言也可以作为脚本语言。

groovy环境搭建：

groovy在Linux上开发环境的配置：

搭建步骤：

1.安装好JDK环境

2.到官网上下载groovySDK，解压到合适的位置

3.配置Groovy的Groovy\_HOME

groovy语法详解

groovy基础语法：

groovy中的变量中的数据类型都是对象类型，int a = 0;//但是得到的a的类型是Integer类型。

变量的定义，强类型定义方式，弱类型def定义方式

强类型就是在声明变量的同时直接声明变量的类型，

弱类型可以不指定变量的类型，类似于var一个变量，

Groovy提供了一个动态类型的定义，比如：

def a = 99 //a是Integer类型的

a = “ads”//这时a是String类型的

字符串：1.String

2.Gstring

常用的三种定义方式，单引号，双引号，三引号

String方法有java.lang.String、DefaultGroovyMethods、

\*重点StringGroovyMethod，普通类型的参数，闭包类型的参数。

groovy中闭包基础详解：

groovy中闭包的使用详解：

闭包的用法，与基本类型的结合使用，与String结合使用，与数据结构结合使用，与文件结合使用

闭包中的三个重要变量：this，owner，delegate

class Person{

def static classClouser = {

println "classClouser this:"+this

println "classClouser owner:"+owner

println "classClouser ownner:"+delegate

}

def say(){

def classClouser = {

println "methodClassClouser this:"+this

println "methodClassClouser owner:"+owner

println "methodClassClouser ownner:"+delegate

}

classClouser.call()

}

}

Person p = new Person();

p.say();

Person.classClouser.call();

//在当前类中定义他们三个输出的值相同，如果在闭包找定义闭包，this指向类，

// 其它两个指向闭包定义处的类本身。

闭包的委托策略：

class Student{

String name

def pretty = {“My name is ${name}”}

String toString（）{

pretty.call()

}

}

可以定义

数据结构：groovy中列表详解

groovy中的映射（HashMap）详解

groovy中的范围：

def range = 1..10

range.each{

println i

}

面向对象：

groovy中类，接口等的定义和使用，groovy中所有的类和方法都是public类型的。

groovy中的元编程：编写代码所执行的时期：有解释执行的js，有编译执行的java，还有运行时期执行的反射。

Groovy的运行时期时很强大的。在java中调用方法时只有第一步，在编译期找有没有此方法。

groovy对json的操作

def list = [ new Person(name : ’libin ’,age:26),

new Person(name:’jinwan’,age:26)

]

def json = JsonOutput.toJson(list)//输出json字符串

println JsonOutput.prettyPrint(json)//输出标准格式

groovy对xml文件操作详解

java对xml处理用DOM和SAX来处理

groovy中如何解析一个xml格式数据

groovy中如何创建一个xml格式数据

groovy与java对比

在写法上：没有java那么多的限制

在功能上：对java已有的功能进行了极大的扩展

作用上：既可以编写应用，可以编写脚本

groovy的闭包讲解：

groovy的数据结构：

groovy面向对象（java的扩展）

gradle主要是用来构建项目的工具，一种编程框架。

gradle组成：1、groovy核心语法

2、build script block

3、gradle api

gradle的优势：灵活性，可以自行扩展脚本。

粒度性：可以看见打包，编译过程

扩展性上：可以复用已有的插件

兼容性上：兼容所有的Maven、Ant

gradle的生命周期：gradle的执行流程，

1.在初始化阶段执行的就是setting.gradle，会读出整个工程中有多少个project，为所有的project构建对应的project对象。

2.gradle build执行过程（在配置阶段生成task的有向无环图）：

要先把其他的task执行完毕后，才能执行到build。这是在配置阶段执行完毕后执行的，配置阶段执行完后gradle在内存中生成task的依赖关系，最后才执行build

gradle Project是gradle的一个入口：

项目大名是一个根工程,是一个Root project，module也被看作是project，单module包里含有build.gradle时才能被看作是一个project。java项目中，每一个project对应的就是一个jar包的生成。web项目对应的就是一个war包的生成。

一个子project对应一个输出。

所有的API组成部分：

project关于文件、属性、依赖的配置，文件的导入等。

获取父工程名称，必须在子工程中获取

def getParentProject(){

def name = this.getParent().name//返回的就是一个parentProject

println “the parent project name is ${ name }”

}

1.可以在根工程中去配置子工程：

project(‘common’){ Project project ->

apply plugin: ‘java’

group ‘com.imooc’

version ‘1.0.0-release’

dependencies{

//子工程所要依赖的jar包

}

}

//为所有工程配置

allprojects{

}

//2.为子工程配置

subprojects{

apply from: ‘../publishToMaven.gradle’

}

project相关的属性和用法：

可以专门定义一个common.gradle中包含ext{}里面定义

ext = {

android = [

compileSdkVersion : 25

buildToolsVersion : ’25.0.0’

]

}

在其它地方用时：rootProject.ext.android. compileSdkVersion

在build.gradle中引入：

apply from： this.file(‘common.gradle’)

还可以建立一个gradle.properties里面直接定义属性

文件属性操作：

其它api：依赖相关API和外部命令执行

buildscript{ ScriptHandler scriptHandler ->

//配置我们工程的仓库地址

scriptHandler.repositories{

}

//配置我们的工程“插件”依赖地址

scriptHandler.dependencies{

}

}

平时开发中禁止传递依赖，

compile（rootProject.ext.dependence.libAutoScrollViewPager）{

exclude module: ‘support-v4’//排除依赖

transitive false //禁止传递依赖

}

compile范围指的是编译范围有效，在编译和打包时都会将依赖存储进去。

provided依赖（占位编译）：在编译和测试的过程中有效，最后生成war包时不会加入，

Task定义及配置，两种创建task的方式。

Task的执行详解：只有写在doFirst和doLast中才能执行在gradle的执行阶段。

gradle的初始化阶段就是执行setting.gradle里面的内容

sourceSet类决定了

配置代码都是在配置阶段执行的，task是在执行阶段执行的，只有在doFirst或者doLast中定义的方法才能在task中执行。

Task的执行顺序：

task：dependsOn，task所依赖的task

TASK\_TYPE=”type ”，task的类型

TASK\_NAME=”name ”，task的名称

TASK\_GROUP =”group ”，task的分组

TASK\_OVERWRITE = “overwrite”，重写一个task

TASK\_ACTION=”action ”，action是task要执行的逻辑

task的输入输出

task的类型：

task types：

使用java插件可以打jar包，

gradle工程的迁移：

自动重定向路径SourceSets.main到当前src路径和res路径

gradle工程的改造：

Task创建，执行及依赖，task里面创建的是真正的脚本逻辑，

copy java文件到指定目录：

把test目录下所有文件复制到anotherDirectory目录下，然后我们执行gradle copyOne即可。

task copyOne(type:Copy){

from ‘src/main/test’

into ‘build/anotherDirectory’

}

对文件的过滤：

有时候一个目录下的文件数目过多，我们只想复制某一部分文件，比如只复制java文件或资源文件等，这时要用到copy任务的include属性，比如只复制java文件到某一指定目录：

task copyTwo(type: Copy){

form ‘src/main/test’

into ‘build/anotherDirectory’

include ‘ \* \* / \* .java’

}

发布jar文件到Maven的仓库中

apply plugin：‘maven’

publishToMaven{

repositories.mavenDeployer{

repository( url : “file://localhost/tmp/myRepo”)

}

}

新建一个工程名为InspectGradle路径为D:\study\android\projects\InspectGradleDir

rootDir对应：

D:\study\android\projects\InspectGradleDir

projectDir对应

D:\study\android\projects\InspectGradleDir\app

buildDir对应

D:\study\android\projects\InspectGradleDir\app\build

持续集成工具Jenkins

Jenkins介绍

持续集成：倡导人们多次集成，自动

Jenkins环境搭建

使用Jenkins打包

强势技能的学习：

重视基础

节奏感

思维方式的改进

效率工具的使用

主动输出

1.1.2 netty及Thrift学习

精通并发与Netty

原生的socket、http、长连接的websocket，

Netty is an asynchronous event-driven network application framework

netty是一个异步的事件驱动（一个一个阶段性的）的网络框架，它是基于一个高性能的，协议化的，服务端的和客户端的快速开发。

与传统SpringMVC框架是有很大的差别的，

在netty中一个请求过来了，不一定能处理往，它是用监听器作为回调的方式根据不同的状态来决定消息发出去了，还是在发送的过程中出现异常，都是通过监听器的方式来做进一步的判断。比如发送消息的是发送完后立刻返回，发送成功与否是通过回调来判断的。

SEDA（staged Event Driven Architacture）阶段性事件驱动：把一个请求分成若干个阶段，不同阶段可以用不同数量的线程来处理，而不是说将一个请求作为一个整体。阶段与阶段之间是通过事件驱动这种异步的通讯，每个阶段可以配置不同数量的线程。实现UDP的。

Netty是基于https的安全的支持。高性能的。

0拷贝的Byte Buffer，是提高效率的重要一环，netty 5.0已经被废弃了。

Netty介绍：

Netty架构实现

Netty模块分析

Netty Http Tunnel：1.netty对Http的一个支持

Netty对Socket的实现

Netty压缩与解压缩：

Netty对于RPC的支援：

WebSocket实现与原理分析

WebSocket连接建立方式与生命周期分解

WebSocket服务端与客户端开发

PRC框架分析（比如dubbo框架）

Apache Thrift使用方式与文件编写方式分析

Netty大文件传送支持

可扩展的事件模型

Netty统一通信API

零拷贝在Netty中的实现与支持

TCP黏包与拆包分析

NIO（非阻塞）模型在Netty中的实现

java中对NIO的实现

Channel是一个对象，可以通过它读取和写入数据。拿 NIO 与原来的 I/O 做个比较，通道就像是流，而且他们面向缓冲区的。

正如前面提到的，所有数据都通过 Buffer 对象来处理。您永远不会将字节直接写入通道中，相反，您是将数据写入包含一个或者多个字节的缓冲区。同样，您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。

通道与流的不同之处在于通道是双向的。而流只是在一个方向上移动(一个流必须是 InputStream 或者 OutputStream 的子类)， 而 通道 可以用于读、写或者同时用于读写。无论在http中的开发还是在WebSocket中的开发都离不开Channel。

因为它们是双向的，所以通道可以比流更好地反映底层操作系统的真实情况。特别是在 UNIX 模型中，底层操作系统通道是双向的

 Java IO和NIO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。 Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

Google Protobuf使用方式分析

什么是protocol buffers：它是一个语言中立，平台中立，类似于xml，更加小巧。

message Person{

required string name =1；

required int32 id= 2；

required string email = 3；

}

Person

person.getPersionByName(String name);//在rpc框架中，这个调用首先将调用序列化，之后通过将序列化后的结果传到服务端，服务端再反序列化，然后再将结果映射到自己结构当中真正完成一个代码调用的逻辑，当调用完以后，再序列化传输到客户端，客户端再反序列化。

由RPC框架根据特定的数据文件，自动生成的stub（客户端代码），skeleton（服务器端代码）。可以在java语言中调用一个方法，这个方法本身是用Python语言编写的。

Apache thrift：可以独立的使用并且不依附于netty。

rmi是EJB里面支持的一种远程调用的框架。

WebSocket（一种浏览器客户端所大量使用的一种标准）随着html5的标准一起发布的，依附于http的，一种长连接的机制，http是短连接，无状态的协议，基于请求的（一定是客户端发出的）协议。WebSocket协议支持（在受控环境中运行不受信任的代码的）客户端与（选择加入该代码的通信的）远程主机之间进行全双工通信（两边都可以是服务端，也都可以是客户端）。用于此的安全模型是Web浏览器常用的基于原始的安全模式。 协议包括一个开放的握手以及随后的TCP层上的消息帧。 该技术的目标是为基于浏览器的、需要和服务器进行双向通信的（服务器不能依赖于打开多个HTTP连接（例如，使用XMLHttpRequest或<iframe>和长轮询））应用程序提供一种通信机制。

在实现websocket连线过程中，需要通过浏览器发出websocket连线请求，然后服务器发出回应，这个过程通常称为“握手” 。在 WebSocket API，浏览器和服务器只需要做一个握手的动作，然后，浏览器和服务器之间就形成了一条快速通道。两者之间就直接可以数据互相传送，双方就是一种对等的，在此WebSocket 协议中，为我们实现即时服务带来了两大好处：

1. Header

互相沟通的Header是很小的-大概只有 2 Bytes，可以降低

2. Server Push

服务器的推送，服务器不再被动的接收到浏览器的请求之后才返回数据，而是在有新数据时就主动推送给浏览器

对于Http请求来说，header数据量太大，远远影响了效率。

对于WebSocket要加上heartBeat用于心跳检测用的，对WebSocket主动发送关闭，服务端是可以通过心跳检测感知到后，服务端也释放连接。

心跳：心跳不是服务端主动去发送信息检测客户端状态，而是在服务端保存下来所有客户端的状态信息，然后等待客户端定时来访问服务端，更新自己当前状态，如果客户端超过指定的时间没有来更新状态，则认为客户端已经宕机或者其状态异常。

心跳机制与轮询的比较：采用心跳，可以避免服务端的压力，灵活好控制，有虽然有保存客户端的socket会话，但是客户端宕机会话就失效了。所以只能等着他主动来报告状态。

基于Akka实现的RPC通信框架是Spark引以为豪的主要特性，也是与Hadoop等分布式计算框架对比过程中一大亮点，但是时代和技术都在演化，从Spark1.3.1版本开始，为了解决大数据块（如shuffle）的传输问题，Spark引入了Netty通信框架，到了1.6.0版本，Netty居然完全取代了Akka，承担Spark内部所有的RPC通信以及数据流传输。

Apache Spark是一个快速的、通用目的的一个集群框架，常用于大数据的计算。

在linux操作系统层面，网络操作即未IO操作，共有5种，1、阻塞式，非阻塞式，复用模型，信号驱动和异步五种IO模型。其中：

1. 阻塞IO操作请求发起后，从网卡等待/读取数据，内核/到用户态的拷贝，整个IO过程，用户的线程都是阻塞状态。
2. 非阻塞与阻塞的区别在于应用层不会等待网卡接收数据，即在内核数据未准备好之前，IO将返回EWOULDBLOCK，用户端通过主动轮询，主动轮询，知道内核态数据准备好，然后再主动发起内核态数据到用户拷贝读操作（阻塞）。
3. 在非阻塞IO中，每个IO的应用层代码都需要主动的去内核态轮询知道数据OK，在IO复用模型中，将轮询/事件驱动的工作交给一个单独的select/epoll的IO句柄去做，即所谓的IO复用。
4. 信号驱动IO时向内核注册信号回调函数，在数据OK的时候自动触发回调函数，进而可以在回调函数中启用数据的读取，即由内核告诉我们何时开始IO操作。
5. 异步IO是将IO数据读取到用户内存的函数注册到系统中，在内核数据OK的时候自动完成内核态到用户态的拷贝，并通知应用态数据已经读取完成，既由内核告诉我们何时IO操作已经完成。
6. java IO也经历了几次演化，从最早的BIO，到1.4版本的NIO（IO复用），到1.7版本的NIO2.0/AIO（异步IO）；基于早期BIO来实现高并发网络服务器都是依赖多线程来实现，但是线程开销大，BIO的瓶颈明显，NIO的出现解决了这一大难题，基于IO复用解决了IO高并发；但是NIO也有以下几个缺点：

①③③② API可用性较低，拿ByteBuffer来说，共用一个current指针，读写切换需要进行flip和rewind，相当麻烦。

仅仅是API，如果想在NIO上实现一个网络模型，还需要自己写很多比如线程池，解码，半包、粘包，限流等逻辑。

最后就是著名的NIO-Epoll死循环的Bug，因为这几个原因，促使了很多JAVA IO通信框架的出现，Netty就是其中一员，他也因为高度的稳定性，功能性，性能等特性，成为首选。

那么Netty和JDK NIO之间到底有什么关系，是JDK NIO的封装，Netty提供了NioEventLoopGroup、NioSocketChannel、NioServerSocketChannel的组合来完成实际IO操作，继而在此之上实现数据流Pipeline以及EventLoop线程池等功能。另外他又重写了NIO，JDK NIO底层是基于Epoll的LT模式来实现，而Netty是基于Epoll的ET模式来实现的一组IO操作EpollEventLoopGroup、EpollSocketChannel、EpollServerSocketChannel;Netty对两种实现进行了完美的封装，可以根据业务的需求来选择不同的实现。

LT:level trigger, 水平触发模式

ET:edge trigger, 边缘触发模式

相同点：都是通过epoll\_wait从EPOLL等待队列读取激活事件

区别：

1. LT模式读取激活事件后，如果还有未处理的数据。事件会重新放入EPOLL等待队列。

2. ET模式读取激活事件，直接从EPOLL等待队列移除，不管是否有未处理的数据。

事件放入EPOLL等待队列条件：

读事件

1. 从不可读变为可读

2. 有新数据到来

3. 有老数据，并且通过epoll\_ctl设置EPOLL\_CTL\_MOD(ET模式)

4. 数据未读完(LT模式)

写事件

1. 从不可写变为可写

2. 有新的可写数据出现

3. 数据可写，并且通过epoll\_ctl设置EPOLL\_CTL\_MOD(ET模式)

4. 数据可写(LT模式)

LT模式问题：如果可读或可写事件未处理，会频繁激活未处理事件

解决方法：不想处理读写事件时， 从EPOLL中移除句柄。需要处理时再加入EPOLL

ET模式问题：如果可读或可写没有全部处理，会有老数据残留。要等待新数据到来。

解决方法：1. 循环读取或写入数据，一直到EAGAIN或EWOULDBLOCK

2. 读取或者写入数据后，通过epoll\_ctl设置EPOLL\_CTL\_MOD，激活未处理事件。

Actor模型：在使用java进行并发编程时需要特别的关注锁和内存原子性等一系列线程问题，而Actor模型内部的状态由它自己维护及，即它内部数据只能由自己修改，通过消息传递来进行状态修改，所以使用Actors模型进行并发编程可以很好的避免这些问题，Actor由状态state、行为Behavior和邮箱mailBox三部分组成。

1. 状态state：Actor中的状态指的是Actor对象的变量信息，状态由Actor自己管理，避免了并发环境下锁和内存原子性等问题。
2. 行为Behavior：行为指定的是Actor中计算逻辑，通过Actor接收到信息来改变Actor的状态。
3. 邮箱mailBox：邮箱是Actor和Actor之间的通信桥梁，邮箱内部通过FIFO消息队列来存储发送方Actor消息，接收方Actor从邮箱队列中获取消息。

使用Actor模型的好处：

（1）事件模型驱动--Actor之间的通信是异步的，即使Actor在发送消息后也无需阻塞或者等待就能够处理其他事情

（2）强隔离性--Actor中的方法不能由外部直接调用，所有的一切都通过消息传递进行的，从而避免了Actor之间的数据共享，想要观察到另一个Actor的状态变化只能通过消息传递进行询问

（3）位置透明--无论Actor地址是在本地还是在远程机上对于代码来说都是一样的

（4）轻量性--Actor是非常轻量的计算单机，单个Actor仅占400多字节，只需少量内存就能达到高并发

1.netty可以作为一个rpc通讯框架实现远程过程的调用。

2.基于webSocket长连接的形式作为一个服务器。

3.netty还可以作为一个http的服务器，类似于Tomcat，jetty这种Servlet服务器。但是它没有遵循Servlet规范，因为它没有实现Servlet接口。

Channel，ChannelHandler（过滤），pipeline（管道，里面有很多的ChannelHandler）

请求路由：requestMapping，netty没有任何的支持，而SpringMVC有。

netty所能完成就是socket开发，也就是一种rpc的开发，netty支持长连接的开发，长连接可以用于消息推送，聊天，服务器端可以对客户端进行推送，都会用WebSocket。

http 1.1中新增了一个keepAlived，在几秒内，如果客户端还会向服务端返回一个响应，就不会建立新的连接。

WebSocket（HTML5规范的一部分，是http协议的升级版）可以实现浏览器和服务器之间的长连接，这种长连接是真正意义上的长连接，连接一直存在，不会断，双方是对等的，在初次时建立时发送头信息，以后就发送真正的数据本身，而不发送头信息。

长连接关闭有时候是感知不到的，所以需要加心跳。比如手机开启了飞行模式，这时服务器端是感知不到的连接关闭的。

Google Protobuf使用方式分析：用来去进行rpc这种数据传输，自定义了一种协议，可以更好的对数据编码和解码，就是序列化和反序列的过程，是rpc的一个常用的库或者说是框架，和Thrift是同一个领域的技术。

rpc库的使用原理：

RMI（Remote method invocation）：A机器调用B机器上的一个方法，跨机器的方式。我们不需要关心这种

两边调用都是序列化成字节码的形式，server反序列成特定类的特定方法的调用，然后再返回序列化结果。RMI只针对Java。开发者可以不关心具体的细节，正常调用就可以了。这种机制有一种保障来实现这种网络传输自动化，所以会存在这种代码自动生成的概念，几乎所有的rpc都存在代码生成，在自动生成的代码中会有序列化和反序列话及网络传输的过程。

客户端生成的client代码叫做stub，server端生成的代码叫做skeleton。它们底层通过socket通讯。

序列化与反序列化也就叫做：编码与解码

RPC（remote Procedure Call），远程过程调用，它的框架是跨语言的跨机器的调用。基于socket传输（效率较高），WebSocket一般基于http进行

1.首先需要去定义一个接口说明文件：server.thrift文件中描述了对象（结构体）、对象成员、接口方法等一系列信息。

2.通过rpc框架所提供的编译器，将接口说明文件编译成具体语言文件。

3.在客户端与服务器端分别引入PRC编译器所生成的文件，即可像调用本地方法一样调用远程方法。

RPC框架包含四部分：

1. 通信模块（网络通信框架）：假设通信的的为A机器与B机器，A与B之间有通信模型，在java中一般基于NIO。
2. 过程服务定位：使用给定的通信方式，与确定IP与端口方法名称确定具体的过程或方法。
3. 远程代理对象：本地调用的的方法服务其实是远程方法的本地代理，因此可能需要一个远程代理对象。

2和3可以概括为调用控制协议。

1. 序列化协议，将对象名称，方法名称，参数等对象信息进行网络传输需要转换成二进制传输，如protobuf,Arvo等。

完成RPC需要两个协议，“序列化协议”“调用控制协议”

完成RPC 需要两个协议，“序列化协议”和“调用控制协议”，还需要一个网络通信框架，而Netty就是一个网络通信框架。

常见例子举例：

1.zeroC ICE，拥有自己的网络通信框架 + ICE 调用控制协议和对象序列化协议,同时也涵盖了服务组件的抽象部署等功能。

2.thrift，有自己的网络通信框架+thrift 对象序列化协议+thrift 调用控制协议

3.probuff，只是 对象序列化协议

4.XMLRPC ，jsonRPC，常见的语境是利用HTTP协议作为调用控制协议,XML 和 JSON 作为对象序列化之后的格式。

5. Netty只是网络通信框架,目的是让你用最少的代码构建出足够支撑网络通信的功能，它是基于JAVA的NIO进行封装后的产品。

一般服务与服务之间用rpc调用。

webService也是一种rpc的实现方式，决定rpc框架性能的因素：

1.编解码的效率，webService编解码的速度较rpc低

2.在数据传递上rpc都是通过webSocket进行传输的，它的效率更高，webService是通过http进行传输对的。

数据类型都是有rpc框架定义的，提供的数据类型较少，因为支持的语言多。

protocol buffers

代码生成：

使用Git作为版本控制系统：

git submodule：

thrift数据类型：

1.bool：

2.byte：有符号字节

3.i16：16位符号整数

4.i32：32位有符号整数

5.i64：64位有符号整数

6.double：64位浮点数

7.string：字符串

可以通过thrift来定义消息传输内容的本身，也提供服务端和客户端。

也可以通过netty作为载体，向服务端和客户端进行双向的传递。

Thrift是一种典型的CS（客户端/服务端）结构，客户端和服务端可以使用不同的语言开发，既然客户端和服务端能使用不同的语言开发，那么一定就要有一种中间语言关联客户端和服务端的语言，这种语言就是IDL（Interface Description Language）

Thrift容器类型：

struct ：表示一个类

service：表示服务端和客户端之间通讯用到的接口

exception：表示服务端和客户端之间通讯可能遇到的异常

thrift有自己的服务端和客户端，我们是在thrift这个框架之下编写对象的类、接口，加进去后，启动客户端，启动服务端，就能实现通讯了。

thrift工作原理：

如何实现多语言之间的通讯：

数据传输使用socket，数据再以特定的格式（string等）发送，接受方语言进行解析。

TNonblockingServerSocket socket = new TNonblockingServerSocket(8899);

TCompactProtocol：在应用层，即协议的一种对象，它会把生成的字节码尽最大的努力压缩成容量最小的字节码的组成，从一端发往另外一端。

TFramedTransport：表示的是传输层所用的一种对象，以一种什么形式，传递给了另外一端。

传输是比较底层的，协议是比较高级的，对于服务器端，它的protocol和 相对应上。

TServer的实现类有六种，其中

THsHaServer：半同步半异步的，Half的Synchronize

TthreadPoolServer线程是从线程池中取的，

Thrift架构：

YourCode就是自己写的调用，方法调用的对象，write是写入到socket里面，read是从socket里面读取出来。也是最复杂的地方，是thrift帮助我们生成的。

协议层可以传输json、xml，遵照约定的报文及数据格式。

一般rpc的模式也同上。

Thrift的协议格式：

TBinaryProtocol - 二进制格式

TCompactProtocol - 压缩的二进制格式，把二进制压缩为更为紧凑的，更加高效

TJsonProtocol - json格式，是一种文本格式

TSimpleJSONProtocol - 提供JSON只写协议，生成的文件很容易通过脚本语言解析。

TDebugProtocol-使用易懂的可读的文本格式，以便与Debug。

Thrift数据的传输方式：

TSocket - 阻塞式socket

TFrameTransport - 以frame（帧）为单位进行传输，非阻塞式服务中使用。

把数据在传输的时候分割成一帧一帧的frame

TFileTransport - 以文件形式进行传递，

TMemoryTransport - 将内存用于I/O。Java实现时内部使用了简单的ByteArrayOutPutStream。

TZlibTransport - 使用zlib进行压缩，与其他传输方式联合使用

TServer服务模型：

TSimpleServer - 简单的单线程服务模型，常用于测试

TThreadPoolServer - 多线程服务模型，使用标准的阻塞式IO。使用了一种线程池，一个请求过来后，会从线程池中取出一个线程，用来服务请求，请求处理完后，把线程交还给线程池。

TNonblockingServer -引入了线程池去处理，使用非阻塞IO，必须使用

THsHaServer - THsHa引入了线程池去处理，其模型把读写任务放到线程池去处理，Half-sync/Half-async的处理模型，Half-aysnc是在处理IO事件上比如创建连接读和写上，Half-sync用于handler（真正的处理器）对rpc的同步处理。THsHaServer继承了TNoblockingServer，所以也需要TFramedTransport

## 1.1.2.1 NIO知识点

阻塞   传统的 IO 流都是阻塞式的。也就是说，当一个线程调用 read() 或 write()时，该线程被阻塞，直到有一些数据被读取或写入，该线程在此期间不能执行其他任务。因此，在完成网络通信进行 IO 操作时，由于线程会阻塞，所以服务器端必须为每个客户端都提供一个独立的线程进行处理，当服务器端需要处理大量客户端时，性能急剧下降。

非阻塞  Java NIO 是非阻塞模式的。当线程从某通道进行读写数据时，若没有数据可用时，该线程可以进行其他任务。线程通常将非阻塞 IO 的空闲时间用于在其他通道上执行 IO 操作，所以单独的线程可以管理多个输入和输出通道。因此，NIO 可以让服务器端使用一个或有限几个线程来同时处理连接到服务器端的所有客户端。

thrift也支持常量定义，使用const关键字:

const i32 MAX\_RETRIES\_TIME = 10

const string MY\_WEBSITE = “http://facebook.com”

格式是：namespace 语言名 路径，相当于package

Thrift提供两个关键字：required、optional，分别用于表示对应的字段是必填的还是可选的。

struct People{

1：required string name；//必须输入

2 : optional i32 age;//非必输

}

注释：# 、//、/\*\*/都是注释

javaIO：流的概念：

java程序通过流来完成输入/输出。流是生产或者消费信息的抽象。流通过java的输入/输出系统与物理设备链接。尽管与它们链接的物理设备不尽相同，所有流的行为具有同样的方式。这样相同的输入、输出类和方法适用于所有类型的外部设备。这意味着一个输入流能够抽象多种不同类型dev输入：从磁盘文件，从键盘或者从网络套接字。同样，一个输出流可以输出到控制台，磁盘文件或者相连的网络。流是处理输入/输出的一个洁净的方法，例如它不需要代码理解键盘和网络的不同java中流的实现是在java.io包定义的类层次结构内部。

InputStream和outputStream是最底层的输入流和输出流。

在I/O库中Decorator（装饰）模式的运用，也称为包装（wrapper）模式

装饰模式对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案。

装饰模式以对客户透明的方式动态的给一个对象附加更多的责任。装饰模式在不创造更多子类的情况下，将对象的功能加以扩展。

装饰模式 VS 继承

装饰模式：

用来扩展特定对象的功能

不需要子类

动态，运行时分配职责，防止由于子类而导致的复杂和混乱，更多的灵活性，对于一个给定的对象，同时可能有不同的装饰对象，客户端可以通过它的需要选择合适的装饰对象发送消息。

装饰模式的使用性：

1.想要透明并且动态的给对象增加新的职责、方法而又不影响其他对象。

2.给对象增加的职责在未来可能会发生改变

3.用子类扩展功能不实际的情况下

volatile的作用：1、内存可见性。2、防止指令重排序

传统的IO是同步的，是阻塞的，

NIO是异步的，非阻塞的，是jdk1.4时出来的

java.io与java.nio的比较：

java.io中最核心的概念是流（Stream），面向流的编程，对于流而言，只可能输出流要么是输入流。

java.nio中有三个核心概念：

1.Selector，选择器

2.Channel，通道

3.Buffer，缓冲区

在java.nio中是面向块，block或者是缓存区buffer来编程的。

selector（选择器）

channel（通道） channel channel

Buffer Buffer Buffer

selector是通过event回调事件在channel中来回切换的，可以把channel类似的理解为io中的stream，buffer就是一块内存，实际上就是一个数组，数据的读写都是同过Buffer来实现的。数据不是直接从channel中读取的，而是先把数据从channel读取到buffer中，然后从buffer中取数据。这个buffer不仅可以从中读取到数据，据而且还可以往里面写数。buffer是双向的。在由读变成写，或者由写变成读时，必须要调用buffer.flip（）进行一次反转。

buffer除了是数组之外，buffer还提供了对于数据结构化访问方式， 并且可以追踪到系统的读写过程，系统可以得到当前读取位置。

java中的8中原生类型都有各自对应的Buffer类型，例如，IntBuffer，LongBuffer，ByteBuffer及CharBuffer。

Channel指的是可以想去写入数据或者从中读取数据的对象，是通过buffer来进行的，永远都不会出现直接向Channel写入数据或者读取数据的情况。

Buffer中三种重要状态属性的含义：position，limit与capacity，如何向buffer中读，如何向buffer中写及为什么要调用buffer.flip()进行反转。

public final Buffer flip() {

limit = position;

position = 0;

mark = -1;

return this;

}

public final Buffer clear() {

position = 0;

limit = capacity;

mark = -1;

return this;

}

关于Buffer的Scattering与Gathering

Scattering：分散开，将来自一个Channel里面的数据读取到多个buffer里面。如果第一个没有读满是不会读取到第二个的。

Gathering：把分散开的汇聚到一起，往外面写的时候也可以传入一个buffer数组，它会按照顺序把buffer数据写入到channel中。

这样可以把消息协议分开放到buffer数组中。

这样就不用取解析某种协议，直接取固定长度数组的数据即可。

传统的IO网络编程：

ServerSocket serverSocket = ...

serverSocket.bind(8899);//这里的8899只是用于客户端连接的端口，在真正

两边交互时，就会重新分配端口号

serverSocket.accept();//阻塞方法，等待客户端和服务端连接，当连接上后

返回一个客户端的socket对象

socket.getInputStream();显然是不能这样写的，因为当有很多要连接时，如果一个连接处理的业务多，就会一直阻塞在这里，其它连接就进不来

//所以，一般都是

While(true){

Socket socket = serverSocket.accept();//阻塞方法

//把socket交给一个线程后，程序继续执行

new Thread(socket){//由于模型简单，所以服务端在接受到大量的业务时，产生的线程数过多是会对CPU负荷过大，

run(){

Socket.getIntStream();

}

}

}

这种编程模型比较简单，所以会出现线程问题。

对于nio，服务端可以用一个线程处理多个socket连接及数据发送，即一个线程可以处理n多个客户端。

Node采用的全部都是异步的处理方式。

定义一个Selector，它针对的就是客户端的连接，一个选择器可以监听到客户端各种各样的事件，异步编程有各种各样的Event，可以在不同的回调事件里面编写逻辑。

实现Closeable类：用于资源自动关闭，close()方法会被自动调用。

Selector:A multiplexor object：它是一个双向通信，它由系统范围的选择器提供者，也可以自定义一个选择器提供者。

将channel通道注册到selector上，注册之后的结果通过SelectorKey去标识的，一个Selector里面维护了三种selectorKey的集合set，

第一种：key set全部的key的集合，里面包含了各种事件比如连接事件、接受连接、读、写等事件。

第二种selected-key set里面包含了各种事件比如我们所感兴趣的连接事件、接受连接或者读、写等事件。是key set的子集。

第三种cancelled-key set原来感兴趣，现在不感兴趣的。是key set的子集。

可以同过SelectableChannel的register方法把Selector注册到Channle

同时也将会把一个key添加到key的集合当中，

无论是调用selector的cancelled-key的cancel方法

或者是关闭这个channel，会使得这个key添加到cancelle-key集合中。

Channel一个key会在下一次调用select方法时，取消与之关联channel的注册。这时这个key从selector的key sets中移除出去。

selector一旦创建好后，每个事件一旦产生，

Set<SelectionKey> selectionKeySet = selector.selectedKeys();selectionKeySet 就有了，它携带了与某个channel所关联的特性，可以根据selectionKey.channel()来获取这个特性，获取到channel对象后，就能源源不断的获取到数据。这个模型叫做事件模型。

Selector会调用select方法，这是一个阻塞方法，会收集各个channel的event事件所在的set<selectionKey>集合。不能直接把key放置到selectionKey集合当中，都是由于某个event事件被触发后，才将key放入到selectionKey集合当中，但是可以调用remove方法直接移除key。可以根据selectionKey得到客户端相应的事件：

SelectionKey.isAcceptable();//客户端和服务端是否连接上

SelectionKey.isReadable();//是否有数据发送过来

Selection操作：通过select方法获取到各种Evnet事件，然后根据事件对象获取相应的事件信息。它涉及了三个步骤：

Selector selector = Selector.open();

ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();//打开ServerSocketChannel

serverSocketChannel.configureBlocking(false);//将其设置成异步的

ServerSocket serverSocket = serverSocketChannel.socket();//打开socket

InetSocketAddress address = new InetSocketAddress();

serverSocket.bind(address);//开启socket服务

Selector selector = Selector.open();//打开selector

//将selector注册到channel上

serverSocketChannel.register(selector, SelectionKey.OP\_ACCEPT);

While(true){

selector.select();//这是一个阻塞事件，当客户端发送消息后直接在这里接收

Set<SelectionKey> selectionKeySet = selector.selectedKeys();//里面有所有已注册到channel的事件

for(SelectionKey selectionKey : selectionKeySet){

If(selectionKey.isAcceptable){

ServerSocketChannel serverChannel= (ServerSocketChannel)selectionKey.channel();//之前注册的什么类型，就强转成什么类型

SocketChannel socketChannel = serverChannel.accept();

socketChannel.configureBlocking(false);

socketChannel.register(selector,SelectionKey.OP\_READ);

}else if(selectionKey.isReadable()){

SocketChannel socketChannel = selectionKey.channel();

While(true){

ByteBuffer byteBuffer = ByteBuffer.allocate(521);

Int read = socketChannel.read(byteBuffer );

If(read <=0 ){

Break;

}

byteBuffer.flip();

//写出byteBuffer里面的字符

Chatset charset = Charset.forName(“UTF-8”);

String receiveMessage = String.valueOf(charset.decode(byteBuffer).array());

socketChannel.write(byteBuffer);

}

socketChannel .read();

}

}

}

//编解码

//解码，将字节数组转为字符串，就能展现出来

MapperByteBuffer inputData = inputFileChannel.map(FileChannel.MapMode.READ\_ONLY, 0 , inputLength);

Charset charset = Charset.forName(“utf-8”);

CharsetDecoder decoder = Charset.newDecoder();

CharBuffer charBuffer = decoder.decode(inputData);

String str = String.valueOf(charBuffer.array( ));

一个byte等于8个bit

ASCII码

7 bit来表示一个字符，共计可以表示128中字符

ISO-8859-1：8 bit来表示一个字符，即用一个完整的字节（byte）来表示一个字符，共计可以表示256个字符。

gb2312用两个字节表示一个汉字。

GBK中文占俩个字节

UTF-8(变长的字节表示形式)中文占三个字节，字母占一个字节。

Unicode中文占两个字节

BOM，一个不可见的字符，要特别注意。

EventLoopGroup bossGroup = new NioEventLoopGroup( );

EventLoopGroup workerGroup = new NioEventLoopGroup( );

ServerBootstrap serverBootstrap = new ServerBootstrap();//一个辅助类，用于简化netty服务器的创建工作

serverBootstrap.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class).childHandler(new MyServerHandler( ));

ChannelFuture channelFuture = serverBootstrap.bind(8893).sync();

channelFuture.channel().closeFuture().sync();

在MyServerHanlder类中会继承SimpleChannelInboundHandler<HttpObject>或者

ChannelInboundHandlerAdapter，Inbound叫做入栈，就是说请求从外面进来了。

ByteBuf的性能比ByteBuffer的性能要高。

ChannelOutboundHandlerAdapter，outbound叫做出栈，把消息发送出去

RPC框架用最少的线程做更多的事情，它是可以跨语言的，之所以可以跨语言，

在于它的消息协议的制定，因为消息本身是跨语言的，只要用不同的语言既定的格式把消息解析出来即可。

netty是一个高性能的网络框架和零拷贝有关

从磁盘上的文件读取到内存然后再以流的形式字节数组出去。

直接

传统IO拷贝操作



read()调用后，由用户空间切换到了内核空间，让后通过DMA方式（一种内存访问的机制）把磁盘上的数据读到内核缓冲区中，接着又将内核缓冲区中的数据拷贝到了用户空间的一个缓冲区中。发生了两次上下文切换。

write()方法调用时，写入socket时，由用户空间切换到内核空间，然后将用户空间里面的数据再次拷贝到另外一个内核缓冲区，之后将内核缓冲区中的数据拷贝到写引擎（一个发送数据的队列）的缓冲区（换句话说，就是再次从内核缓冲区拷贝到socket缓冲区），接着将数据从socket缓冲区中发送给远端对应的服务器。有四次数据拷贝过程，以上有四次上下文的切换，以及两次户空间到内核空间，内核空间到用户空间。用户空间就是一个中转的媒介。如果对数据没有修改时，拷贝就是无用的。

① 发出read系统调用：导致用户空间到内核空间的上下文切换(第一次上下文切换)。通过DMA引擎将文件中的数据从磁盘上读取到内核空间缓冲区(第一次拷贝: hard drive ——> kernel buffer)。  
② 将内核空间缓冲区的数据拷贝到用户空间缓冲区(第二次拷贝: kernel buffer ——> user buffer)，然后read系统调用返回。而系统调用的返回又会导致一次内核空间到用户空间的上下文切换(第二次上下文切换)。  
③ 发出write系统调用：导致用户空间到内核空间的上下文切换(第三次上下文切换)。将用户空间缓冲区中的数据拷贝到内核空间中与socket相关联的缓冲区中(即，第②步中从内核空间缓冲区拷贝而来的数据原封不动的再次拷贝到内核空间的socket缓冲区中。)(第三次拷贝: user buffer ——> socket buffer)。  
④ write系统调用返回，导致内核空间到用户空间的再次上下文切换(第四次上下文切换)。通过DMA引擎将内核缓冲区中的数据传递到协议引擎(第四次拷贝: socket buffer ——> protocol engine)，这次拷贝是一个独立且异步的过程。

Q：你可能会问独立和异步这是什么意思？难道是调用会在数据被传输前返回？  
A：事实上调用的返回并不保证数据被传输；它甚至不保证传输的开始。它只是意味着将我们要发送的数据放入到了一个待发送的队列中，在我们之前可能有许多数据包在排队。除非驱动器或硬件实现优先级环或队列，否则数据是以先进先出的方式传输的。

**总的来说，传统的I/O操作进行了4次用户空间与内核空间的上下文切换，以及4次数据拷贝。其中4次数据拷贝中包括了2次DMA拷贝和2次CPU拷贝。**

Q: 传统I/O模式为什么将数据从磁盘读取到内核空间缓冲区，然后再将数据从内核空间缓冲区拷贝到用户空间缓冲区了？为什么不直接将数据从磁盘读取到用户空间缓冲区就好？  
A: 传统I/O模式之所以将数据从磁盘读取到内核空间缓冲区而不是直接读取到用户空间缓冲区，是为了减少磁盘I/O操作以此来提高性能。因为OS会根据局部性原理在一次read()系统调用的时候预读取更多的文件数据到内核空间缓冲区中，这样当下一次read()系统调用的时候发现要读取的数据已经存在于内核空间缓冲区中的时候只要直接拷贝数据到用户空间缓冲区中即可，无需再进行一次低效的磁盘I/O操作(注意：磁盘I/O操作的速度比直接访问内存慢了好几个数量级)。  
Q: 既然系统内核缓冲区能够减少磁盘I/O操作，那么我们经常使用的BufferedInputStream缓冲区又是用来干啥的？  
A: BufferedInputStream的作用是会根据情况自动为我们预取更多的数据到它自己维护的一个内部字节数据缓冲区中，这样做能够减少系统调用的次数以此来提供性能。

**总的来说内核空间缓冲区的一大用处是为了减少磁盘I/O操作，因为它会从磁盘中预读更多的数据到缓冲区中。而BufferedInputStream的用处是减少“系统调用”。**

**DMA**  
DMA(Direct Memory Access) ———— 直接内存访问 ：DMA是允许外设组件将I/O数据直接传送到主存储器中并且传输不需要CPU的参与，以此将CPU解放出来去完成其他的事情。  
而用户空间与内核空间之间的数据传输并没有类似DMA这种可以不需要CPU参与的传输工具，因此用户空间与内核空间之间的数据传输是需要CPU全程参与的。所有也就有了通过零拷贝技术来减少和避免不必要的CPU数据拷贝过程。

InputStream和OutputStream在底层的反应。

零拷贝完全依赖底层操作系统，如果操作系统提供了就有，Linux和Unix都提供了，避免的两次不必要的拷贝。

零拷贝



操作系统意义上的零拷贝，没有用户空间和内核空间之间的拷贝。

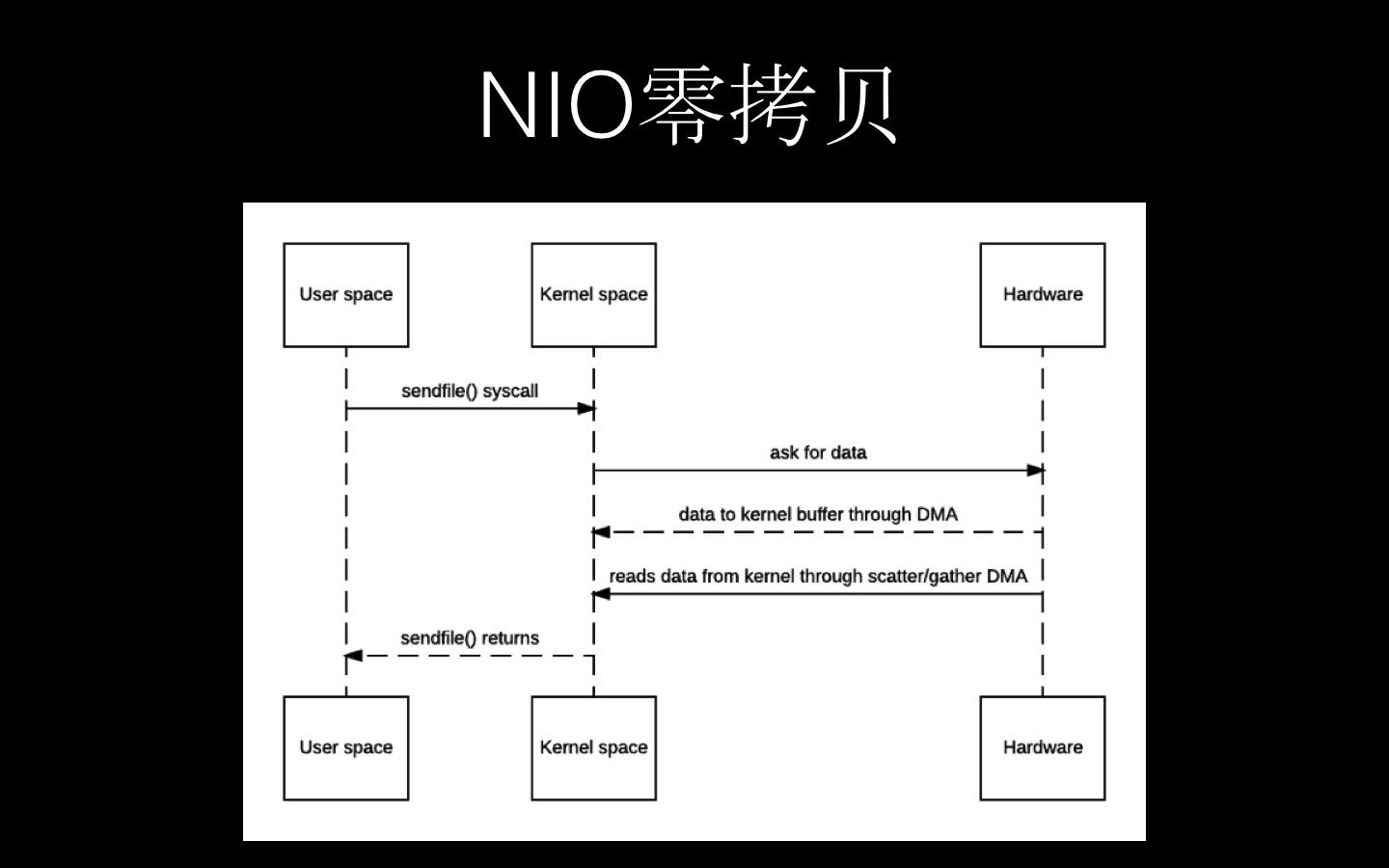
User space发出sendfile()命令，内核空间去向硬件请求数据，把数据从磁盘上拷贝到内核空间缓冲区，然后从内核空间拷贝到socket缓冲区中，最后从socket拷贝到协议引擎中。数据的第三次复制发生在 DMA 引擎将数据从socket缓冲区传到协议引擎时。

① 发出sendfile系统调用，导致用户空间到内核空间的上下文切换(第一次上下文切换)。通过DMA引擎将磁盘文件中的内容拷贝到内核空间缓冲区中(第一次拷贝: hard drive ——> kernel buffer)。然后再将数据从内核空间缓冲区拷贝到内核中与socket相关的缓冲区中(第二次拷贝: kernel buffer ——> socket buffer)。  
② sendfile系统调用返回，导致内核空间到用户空间的上下文切换(第二次上下文切换)。通过DMA引擎将内核空间socket缓冲区中的数据传递到协议引擎(第三次拷贝: socket buffer ——> protocol engine)

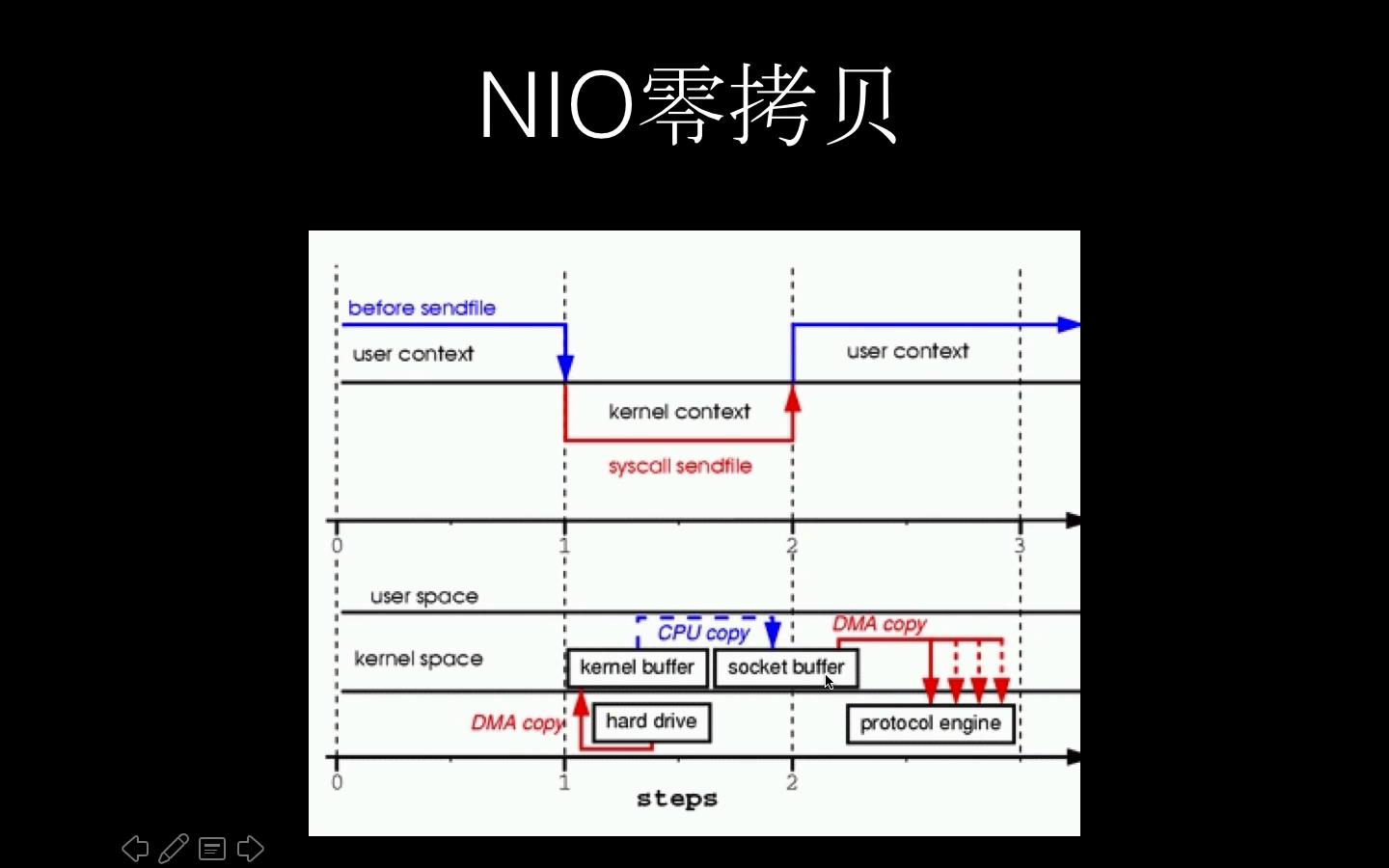
**总的来说，通过sendfile实现的零拷贝I/O只使用了2次用户空间与内核空间的上下文切换，以及3次数据的拷贝。其中3次数据拷贝中包括了2次DMA拷贝和1次CPU拷贝。**

Q：但通过是这里还是存在着一次CPU拷贝操作，即，kernel buffer ——> socket buffer。是否有办法将该拷贝操作也取消掉了？  
A：有的。但这需要底层操作系统的支持。从Linux 2.4版本开始，操作系统底层提供了scatter/gather这种DMA的方式来从内核空间缓冲区中将数据直接读取到协议引擎中，而无需将内核空间缓冲区中的数据再拷贝一份到内核空间socket相关联的缓冲区中。

改进后的零拷贝



将数据从硬件拷贝到内核空间后，可以直接利用scatter和gather直接把内核空间中的数据写入到socket buffer中。这种一次性的复制就可以完成所有的操作。



DMA copy，先将磁盘上的文件拷贝到kemel buffer（内核缓冲区）中，然后又将其中的文件描述符：kemel buffer的内存地址在什么地方，有多长（要读多少数据）拷贝到socket buffer中，而不是整个读到的内容，只是两个指针。

协议引擎（protocol engine）直接根据socket中的文件描述符把kemel buffer中的数据去取出来，发送给服务器端。从socket buffer和kemel buffer中读取，gather操作。

① 发出sendfile系统调用，导致用户空间到内核空间的上下文切换(第一次上下文切换)。通过DMA引擎将磁盘文件中的内容拷贝到内核空间缓冲区中(第一次拷贝: hard drive ——> kernel buffer)。  
② 没有数据拷贝到socket缓冲区。取而代之的是只有相应的描述符信息会被拷贝到相应的socket缓冲区当中。该描述符包含了两方面的信息：a)kernel buffer的内存地址；b)kernel buffer的偏移量。  
③ sendfile系统调用返回，导致内核空间到用户空间的上下文切换(第二次上下文切换)。DMA gather copy根据socket缓冲区中描述符提供的位置和偏移量信息直接将内核空间缓冲区中的数据拷贝到协议引擎上(第二次拷贝: kernel buffer ——> protocol engine)，这样就避免了最后一次CPU数据拷贝。

**总的来说，带有DMA收集拷贝功能的sendfile实现的I/O只使用了2次用户空间与内核空间的上下文切换，以及2次数据的拷贝，而且这2次的数据拷贝都是非CPU拷贝。这样一来我们就实现了最理想的零拷贝I/O传输了，不需要任何一次的CPU拷贝，以及最少的上下文切换。**

优点：1.减少甚至完全避免不必要的CPU拷贝，从而让CPU解脱出来去执行其他的任务2.减少内存带宽的占用3.通常零拷贝技术还能够减少用户空间和操作系统内核空间之间的上下文切换

内存映射文件，直接操作内存来修改磁盘上的文件。在访问文件或者修改文件时通常是在User space上进行的，将磁盘上的文件映射到内核空间上，可以让程序直接修改内核上的文件，底层操作系统会提供相应的支持，这样就可以通过修改user space来修改磁盘上的文件了。

通过MappedByteBuffer类。

在kemel buffer --》socket buffer的过程：

将kemel buffer中的文件描述符写到socket buffer的文件描述符中，第一个是位置，第二个是长度，现在不用将文件复制到socket中，而是将文件描述符放置到socket中即可。protocol engine从两个地方拷贝文件，实际从kemel buffer中读取数据，从socket buffer中读取文件描述符，是一种gather操作。

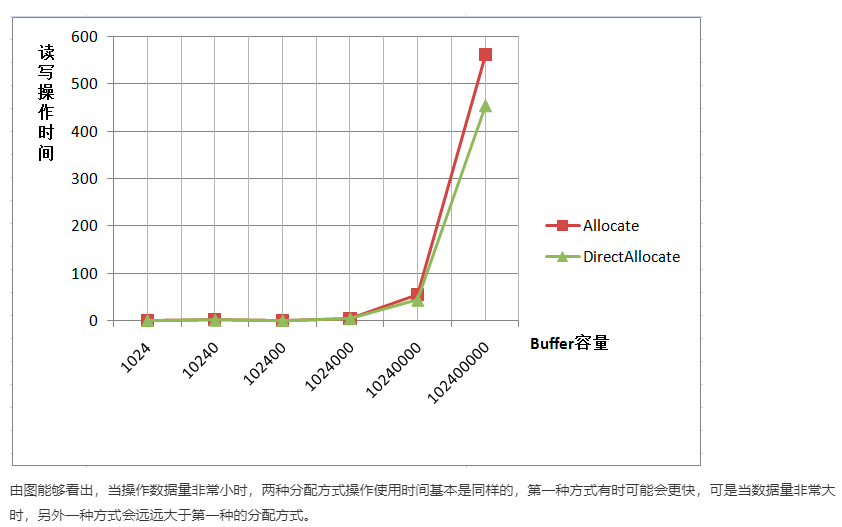
在Linux2.4中会满足条件。

EventLoopGroup bossGroup = new EventLoopGroup(1);//说明只用一个线程用来监听。

jdk在nio中有一个空轮询的bug，看netty是如果解决的。workaround绕过这个bug的。

字节缓存区有两种形式：1、直接字节缓存区，JVM会尽最大的努力直接在此缓冲区上执行本机I/O操作。也就是说，在每次调用基础操作系统的一个本机I/O操作之前或之后，虚拟机都会尽量避免将缓冲区的内容复制到中间缓冲区中，

直接字节缓存区可以通过调用allocateDirect工厂方法创建。所以，建议将直接缓冲区主要分配给那些易受基础系统的本机 I/O 操作影响的大型、持久的缓冲区。一般情况下，最好仅在直接缓冲区能在程序性能方面带来明显好处时分配它们。为什么要提供两种方式呢？这与Java的内存使用机制有关。第一种分配方式产生的内存开销是在JVM中的，而另外一种的分配方式产生的开销在JVM之外，以就是系统级的内存分配。当Java程序接收到外部传来的数据时，首先是被系统内存所获取，然后在由系统内存复制复制到JVM内存中供Java程序使用。所以在另外一种分配方式中，能够省去复制这一步操作，效率上会有所提高。可是系统级内存的分配比起JVM内存的分配要耗时得多，所以并非不论什么时候allocateDirect的操作效率都是最高的。



对于java来说默认使用了BIG\_ENDIAN方式存储，通过final ByteOrder order()返回当前的字节顺序。

final ByteBuffer order(ByteOrder byteOrder)设置字节顺序，

字节序，又称端序，尾序，Endianness。字节序时指存放多字节数据的字节的顺序，整数在内存中额存放方式和网络传输的传输顺序。

一般情况字节序指示了UCS-2字符的那个字节存储在低地址。如果LsByte在MSByte的前面，即LSB为低地址，则该字节是小端序；反之则是大端序。在网络编程中，字节序是一个必须要被考虑的因素，因为不同的处理器体系可能采用不同的字节序。在多平台的代码编程中，字节序可能会导致难以察觉的bug

网络序：网络传输一般采用大端序，也被称之为网络字节序，或网络序。IP协议中定义大端序为网络字节序。

capacity：作为一个内存块，Buffer有一个固定大小值，capacity 个 byte、long，char 等类型。一旦 Buffer 满了，需要将其清空，通过读数据或者清除数据才能继续写数据往里写数据

position：写数据时，position表示当前位置。初始的position值为0。当

数据写入时，position会向前移动到下一个可插入数据的Buffer单元。

当读取数据时，也是从某个特定位置读。当将Buffer从写模式切换到读模式，position会被重置为0。

在写模式下，Buffer的limit表示最多能往Buffer里面写多少数据。在写模式下，limit等于Buffer的capacity

当切换Buffer到读模式时，limit表示你能读到多少数据。

rewind()方法：Buffer.rewind()将position设回0，能够重读Buffer中的所有数据。limit数据不变，仍然表示从Buffer中读取多少个元素。

一旦读完Buffer中的数据，需要让Buffer准备好再次被写入。可以通过clear()或者compact( )方法来完成。

如果调用的是clear（）方法，position将被设回0，limit被设置成capacity的值。换句话说，Buffer被清空了。

public final Buffer clear() {

position = 0;

limit = capacity;

mark = -1;

return this;

}

public ByteBuffer compact() {  
 System.arraycopy(hb, ix(position()), hb, ix(0), remaining());  
 position(remaining());//limit - position;  
 limit(capacity());//capacity大小  
 discardMark();//mark = -1  
 return this;  
}

clear（）方法将position设置回0，limit将设置成capacity的值。Buffer中的数据并未清除，只是开始覆盖原来的数据。

如果Buffer中有一些未读的数据，调用clear（）方法，数据将被遗忘，不再有任何标记哪些数据被读过，哪些数据没有读过。

如果Buffer中仍有未读的数据，且后续还需要这些数据，但是此时想要先写一些数据，就是用compact()方法。

compact（）方法将所有未读的数据拷贝到Buffer起始处。然后将position设到最后一个未读取元素正后面。limit属性依然clear()方法一样，Buffer 准备好写数据了，但是不会覆盖未读的数据。

mark()与reset()方法

mark()方法可以标记Buffer中的一个特定的position。之后可以通过调用Buffer.reset()方法恢复到这个position。

ByteBuffer.wrap（）用于转换一个byte数组，用于读取byte数组里面的数据

直接缓冲区：

操作系统在内存区域中进行I/O操作。这些内存区域就操作系统方面而言，是相连的字节序列。只有字节缓冲区有资格参与I/O操作。在JVM中，字节数组可能不会在内存中连续存储，或者无用存储单元收集可能随时对其进行移动。

字节数组是对象，而数据存储在对象中的方式在不同的JVM实现中都各有不同。

出于这一原因，引入了直接缓冲区的概念。直接缓冲区被用于与通道和固有 I/O 例程交互。它们通过使用固有代码来告知操作系统直接释放或填充内存区域，对用于通道直接或原始存取的内存区域中的字节元素的存储尽了最大的努力

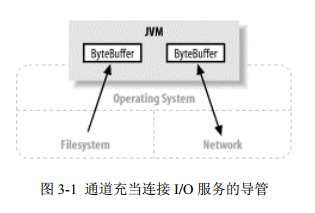
直接字节缓冲区通常是 I/O 操作最好的选择。在设计方面，它们支持 JVM 可用的最高效  
I/O 机制。非直接字节缓冲区可以被传递给通道，但是这样可能导致性能ᦏ耗。通常非直接缓  
冲不可能成为一个本地 I/O 操作的目标。如果您向一个通道中传递一个非直接 ByteBuffer  
对象用于写入，通道可能会在每次调用中隐含地进行下面的操作：  
1.创建一个临时的直接 ByteBuffer 对象。  
2.将非直接缓冲区的内容复制到临时缓冲中。  
3.使用临时缓冲区执行低层次 I/O 操作。  
4.临时缓冲区对象离开作用域，并最终成为被回᭦的无用数据。  
这可能导致缓冲区在每个 I/O 上复制并产生大量对象，而这种事都是我们极力避免的。不过，依靠工具，事情可以不这么糟糕。运行时间可能会缓存并重新使用直接缓冲区或者执行其他一些聪明的技巧来提高吞吐量。如果您仅仅为一次使用而创建了一个缓冲区，区别并不是很明显。另一方面，如果您将在一段高性能㝊本中重复使用缓冲区，分配直接缓冲区并重新使用它们会使您游刃有余。直接缓冲区时 I/O 的最佳选择，但可能比创建非直接缓冲区要花费更高的成本。直接缓冲区使用的内存是通过调用本地操作系统方面的代码分配的，绕过了标准 JVM堆栈。建立和销毁直接缓冲区会明显比具有堆栈的缓冲区更加破费，这取决于主操作系统以及 JVM 实现。  
直接缓冲区的内存区域不受无用存储单元收集支配，因为它们位于标准 JVM 堆栈之外。使用直接缓冲区或非直接缓冲区的性能权㺑会因JVM，操作系统，以及代码设计而产生巨大差异。通过分配堆栈外的内存，您可以使您的应用程序依赖于JVM未涉及的其它力量。当加入其他的移动部分时，确定您正在达到想要的效果。我以一条旧的软件行业格言建议您：先使其工作，再加快其运行。不要一开始就过多担心优化问题；首先要注重正确性。 JVM实现可能会执行缓冲区缓存或其他的优化， 5这会在不需要您参与许多不必要工作的情况下为您ᨀ供所需的性能。

直接 ByteBuffer 是通过调用具有所需容量的 ByteBuffer.allocateDirect()函数产生的，就像我们之前所涉及的 allocate()函数一样。注意用一个 wrap()函数所创建的被包装的缓冲区总是非直接的。

Channel是nio的主要创新，它们既不是一个扩展也不是一项增强，而是全新、极好的java I/O实例，与其直接连接。Channel用于在字节缓存区和通道另外一侧的实体，通常是一个文件或者套接字，之间有效地传输数据。

缓冲区好比一个载体，接着将缓冲写到通道中，然后另一方接收。

通道与操作系统的文件描述符和文件句柄有着一对一的关系。虽然通道比文件描述符更为广义，多数通道都是连接到开放的文件描述符。Channel类提供维持平台独立性所需的抽象过程，不过仍然会模拟现代操作系统本身的I/O性能。



I/O可以大概分为两类：File I/O和Stream I/O。它们是文件通道和套接字通道。有一个FileChannel类和三个socket通道类：SocketChannel、ServerSocketChannel和DatagramChannel。

通道可以有多种方式创建。

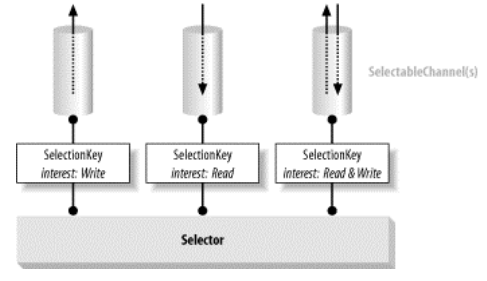
Socket他通道可以直接创建新socket通道的工厂方法。但是一个FileChannel对象只能通过一个打开的RandomAccessFile、FileInputStream或者FileOutputStream对象上调用getChannel()方法来获取。

Scatter/Gather

通道提供了一种被称为Scatter/Gather的重要功能。Scatter/Gather是一个简单却又强大的概念。它是指在多个缓冲区上实现一个简单的I/O操作。对于writer而言，数据是从几个缓冲区按顺序抽取，称为gather，并沿着通道发送。

gather过程的效果就好比全部缓冲区的内容被连接起来，并在发送数据前放到一个大的缓冲区，将每个缓冲区填满直至通道中的数据或者缓冲区的最大空间被消耗完。

大多数现代操作系统都支持本地矢量I/O。当您在一个通道上请求一个Scatter/Gather操作时，该请求会被翻译为适当的本地调用来直接填充或者抽取缓冲区。减少或避免了缓冲区拷贝和系统调用。Scatter/Gather应该使用直接的ByteBuffers以从本地I/O中获取最大性能优势。



调用可选择通道的 register( )方法会将它注册到一个选择器上。如果您试图注册一个处于阻塞状态的通道， register( )将抛出未检查的 IllegalBlockingModeException 异常。此外，通道  
一旦被注册，就不能回到阻ຎ状态。试图这么做的话，将在调用 configureBlocking( )方法时将抛出IllegalBlockingModeException 异常。  
并且，理所当然地，试图注册一个已经关闭的 SelectableChannel 实例的话，也将抛出ClosedChannelException 异常，就像方法原型指示的那样

NioEventLoopGroup中bossGroup的线程数一个都是设置1。作用就是接受客户端的连接，当连接的数据到来后，它不做任何处理，将处理逻辑给workGroup，作用就是转发请求。workGroup是真正完成用户请求处理的业务逻辑。

EventLoopGroup就是一个死循环，

如果没有设置时，线程数是超线程处理后核心数乘2.

1. 自己定一个任务
2. 创建线程是另外一个逻辑

实现线程本身的创建和线程要执行的任务的定义解耦。

NioEventLoopGroup -> MultithreadEventLoopGroup中含有

DefaultThreadFactory：线程工厂，用于专门创建线程

ThreadPerTaskExecutor中用到了两种设计模式：

1. 代理模式

2、命令模式，需要你帮我做一件事情，不用管事情是如何定义的，你只要执行就可以。

**private final** ThreadFactory **threadFactory**;  
**public** ThreadPerTaskExecutor(ThreadFactory threadFactory) {  
 **if** (threadFactory == **null**) {  
 **throw new** NullPointerException(**"threadFactory"**);  
 }  
 **this**.**threadFactory** = threadFactory;  
}  
@Override  
**public void** execute(Runnable command) {  
 **threadFactory**.newThread(command).start();  
}

ServerChannel是一个标识接口（Mark interface），如同一个类实现了该接口，就具备了一些特性供我们使用。

ServerBootstrp是Bootstrap启动类的一个子类（单并不是直接继承过来）。

ServerBootstrap.bind()后，才会启动

Callable其实和Runnable是类似的，只是Callable返回了执行结果。

Future<String> future = executor.submit(new Callable<String>{

public String call(){

.......//处理的逻辑

}

});

displayOtherThings();//继续执行主线程的逻辑

future.get( ) //获取到业务执行完返回的结果。

但是一般也不知道何时执行完毕，所以也不知道何时调用future.get（）方法，

所以在io.netty中的Future类有一个addListener方法，可以监听完成状态。

从而获取完成任务的返回结果。

Future<V> addListener(GenericFutureListener<? **extends** Future<? **super** V>> listener);

一般见到Listener时，都会用到观察者模式，

ChannelFuture返回完成状态信息及完成结果。

它还允许您添加{@link ChannelFutureListener}，以便在I / O操作完成时得到通知。

{@link ChannelHandler}中的事件处理程序方法通常由一个I / O线程调用。如果由I / O线程调用的事件处理程序\*方法调用{@link #await（）}，则它等待的I / O操作可能永远不会完成，因为{@link #await（）}可以阻止它正在等待的I / O \*操作，这是一个死锁，所以需要使用addListener()方法。

比如说在事件处理器中调用了await方法，然而事件处理器又是有I/O线程所调用的。所以在handlerAdded()/handlerRemoved()方法里面不要调用await方法。

channelRead方法是接收到对端消息。

I/O超时和等待超时是两个不同的概念。

等待超时

*\* f.awaitUninterruptibly(10, TimeUnit.SECONDS);*

连接超时

*b.option({****@link*** *ChannelOption}.CONNECT\_TIMEOUT\_MILLIS, 10000)*

ChannelPipeline：把所有的已有配置类连接在一起

pipeline.addLast(**new** ServerBootstrapAcceptor(  
 channel, currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));

Proactor模式：操作是多路复用的，另外一个是异步的，比如I/O操作时是在内核空间完成的还是在用户空间完成的。

Responsive：响应式设计，前端一套代码可以在PC上手机上显示的效果一样。

Reactor模式：反应器模式，netty整体架构时Reactor，

分发模式：

1. 将一个完整处理过程分解为一个一个细小的任务。
2. 在每个I/O任务执行相关动作且不产生阻塞，CPU不会进行等待，当I/O完成时，CPU继续完成该任务。
3. IO经常作为一种触发器来实现的，当读操作完成时，就会产生一个读事件，系统功能就会被触发。

java.nio包就很好的实现了上述机制：

1. 非阻塞的读和写
2. 通过感知IO事件分发任务的执行，所以结合一系列基于事件驱动模式的设计，给高性能IO服务的架构与设计带来丰富的可扩展性。

基于事件驱动模式的设计：

1. 可以节省一定的性能资源，事件驱动模式下不需要为每个客户端建立一个连接。更少的线程开销。
2. 更少的上下文切换和更少的锁互斥。
3. 分发可能变的更慢，必须手工的将动作绑定到事件上。

而且通常实现的复杂度也会增加，相关功能必须分解成简单的非阻塞操作

1. Reactor模式会分配适当的handler处理程序来回调IO事件。
2. 每个handler执行非阻塞的操作。

Channels支持非阻塞读写的socket连接

Buffers用于被Channels读写的字节数组对象

Selectors用于判断已经向selector注册过的channel拥有IO事件SelectionKeys负责IO事件的状态与绑定

## 1.1.2.1 Mina

协议栈的基础：TCP/UDP

TCP：三次握手，四次挥手，以及滑动窗口协议。

UDP：报文协议，如何封装报文的。

Mina：Apache开发的一个开源的网络通信框架，基于javaNIO实现的。

编码与解码，当

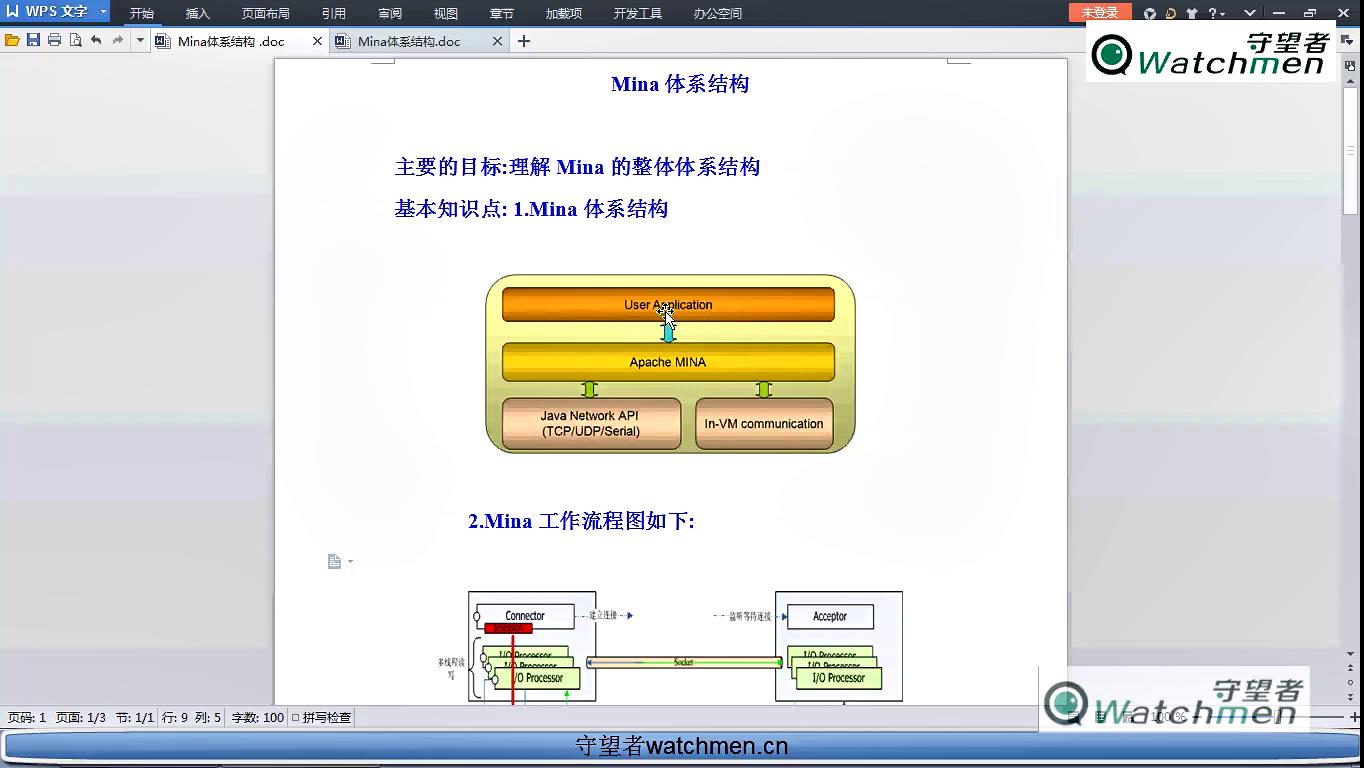
对象到网络中时，需要编码成二进制。

将网络中的二进制解码成java对象或者基本数据类型

练习：

总结：1、NioSocketAccept

1. 设置编码解码过滤器，协议类的滤器
2. 设置一些session属性
3. 处理类Handler
4. 绑定一个端口

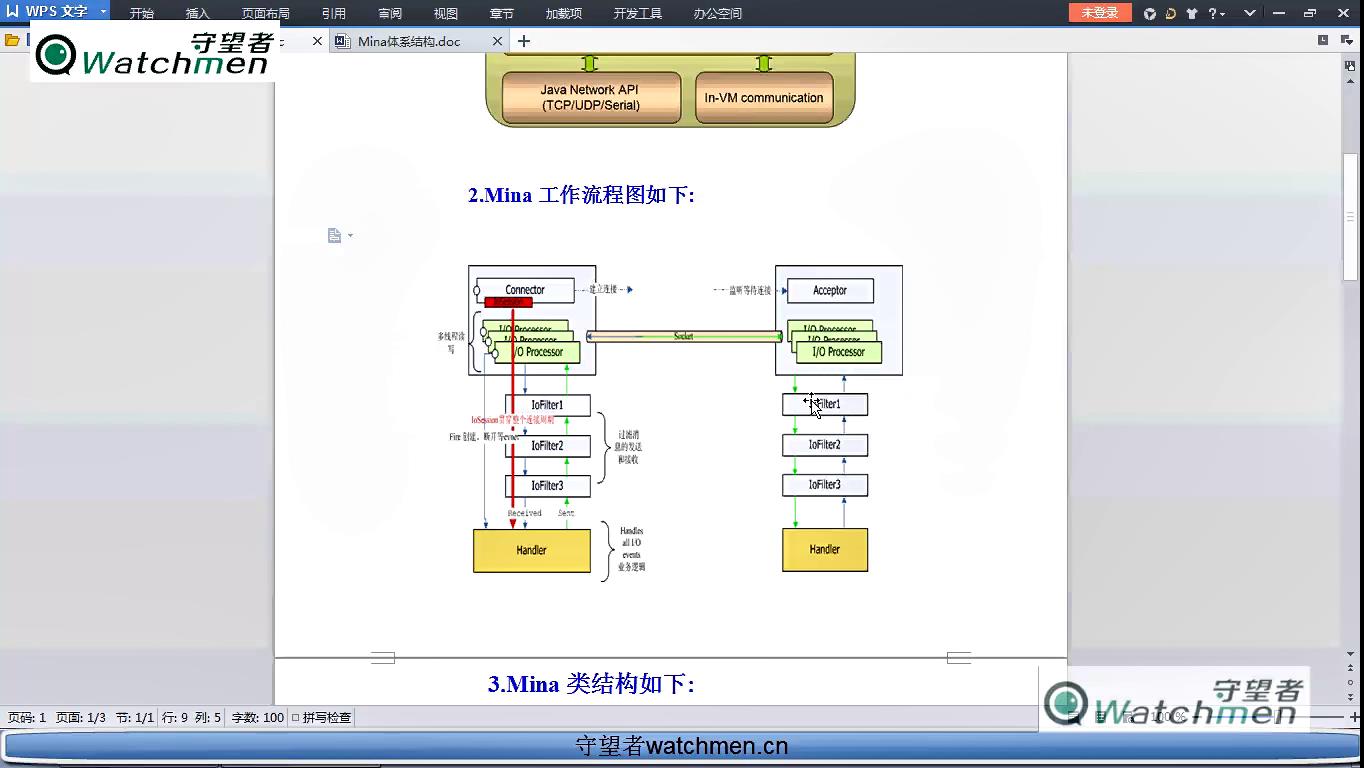


1. mina在程序上所处的地位

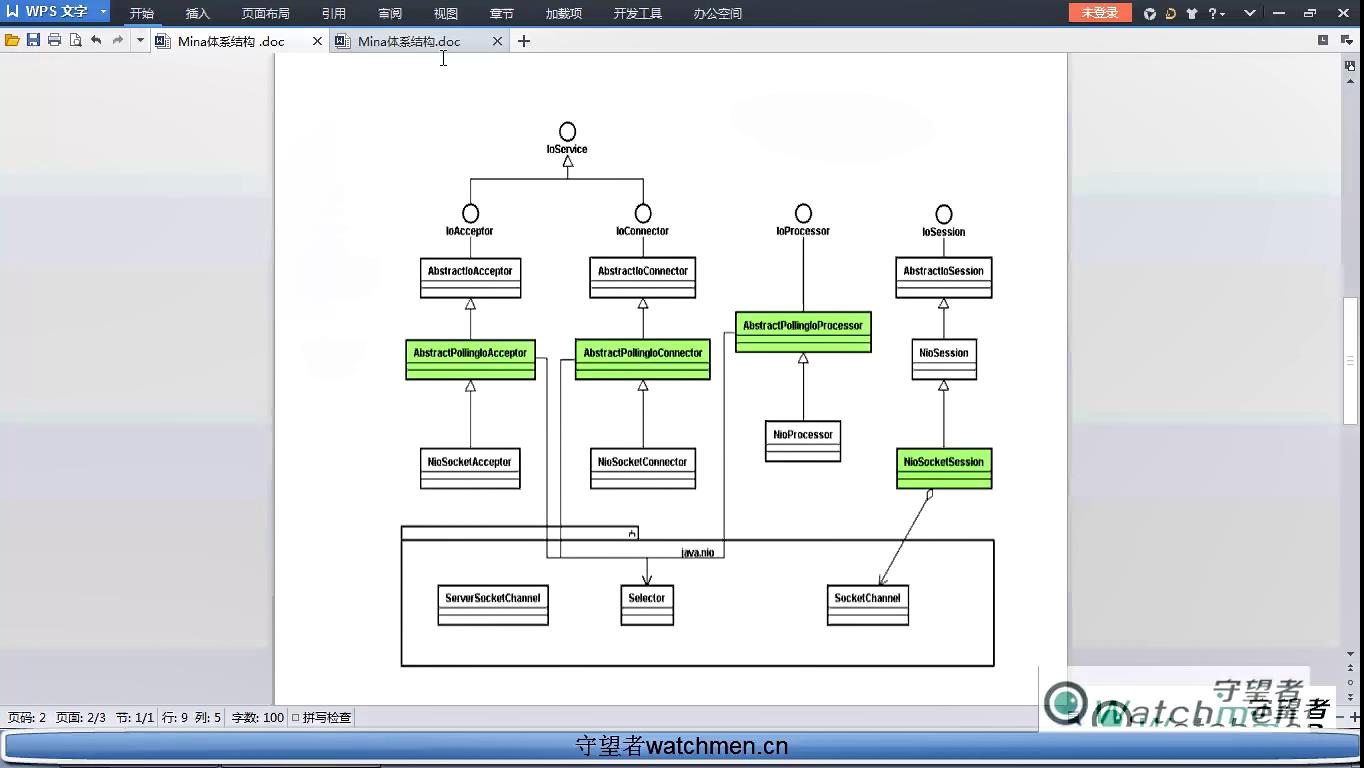
主要屏蔽了网络通信的一些细节，对socket进行了封装，并且是NIO的一个实现架构，可以快速开发网络通信。常用于游戏的开发，中间件等服务端程序的开发。

1. IOService接口，子接口IOAcceptor、IoConnector

用于描述客户端和服务端接口，其子类分别用于描述客户端和服务端。IoProcess的子类IoProcessor<T extends IoSession>它是多线程环境处理连接，下面有多个IoFilter提供数据的过滤工作：包括编解码、日志等信息的过滤，Handler就是业务对象。自定义的Handler需要实现IoHandlerAdapter



IoSession用于客户端和服务端之间的连接的描述及数据的接收和发送



长连接：通信双方长期保持一个连接状态不断开，比如qq,当我们登录上qq时，就去连接腾讯服务器，建立连接后就不断开，除非发生了异常。

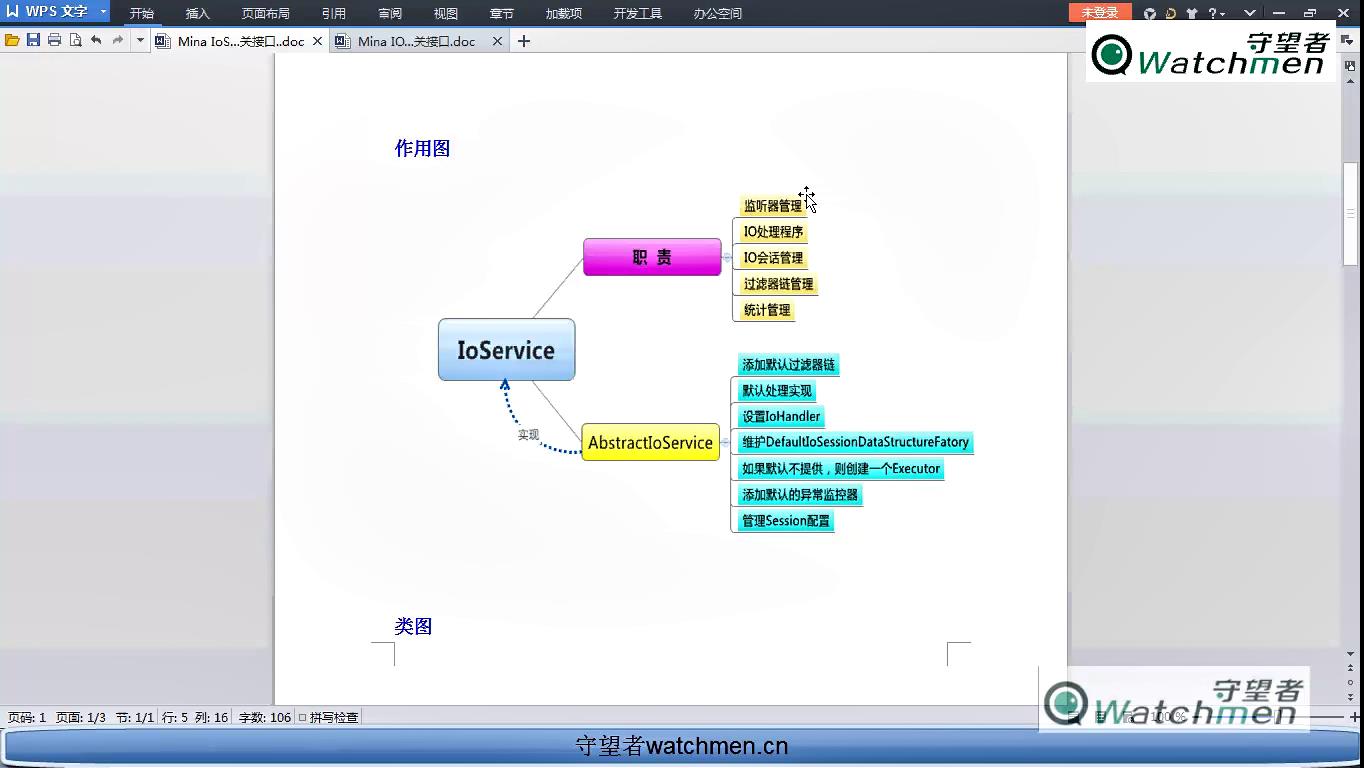
短连接：

通信双方不是保持一个长期的连接状态，比如http协议，当客户端发起http请求，服务器处理http请求，当服务器处理完成后，返回客户端数据后就断开连接。对于下次的连接请求需要重新发起。

IOService：实现了对网络通信的客户端和服务端之间的抽象，用于描述客户端的子接口IOConnector。用于描述服务端的子接口：IoAcceptor。

1. IoService的作用

IoService可以管理网络通信的客户端和服务端，并且可以管理连接双方的session会话，同时可以添加过滤器



IOAcceptor和IOConnector对应的抽象类。

IoService常用API定义一些抽象的接口，可以获得过滤器：

1. getFilterChain()获得过滤器链
2. setHandler（IHandler handler）设置真正业务handler
3. getSessionConfig()得到会话的配置信息
4. dispose()关闭连接时调用的方法

IOFilter：对应用程序和网络这块的传输，就是二进制数据和读写之间的相互转化，有相应的解码和编码器。

在完成自定义过滤器：

就是在handler处理之前，需要调用相应的过滤器进行过滤。

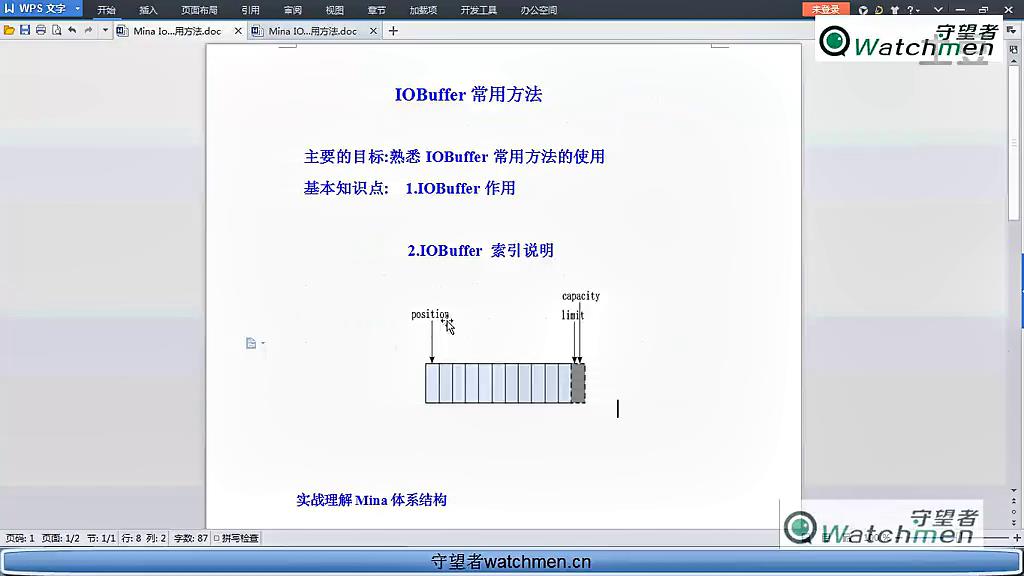
IOBuffer的索引属性：

Capacity:代表当前缓冲区的大小

Position:理解成当前读写位置，是一个可读数据单位的位置：

Position<=Capacity的时候可以完成数据的读写操作。

Limit：下一个不可以被读写缓冲区单元的位置。Limit<=Cap



定义的解码器：因为往往不是通过一个字符串就可以传输所有的信息，我们传输的是自定的协议包。并且能在应用程序和网络通信中存在对象和二进制流之间传化关系。所以我们需要结合业务编写自定义的编解码器。

常用的自定义协议的方法：

1. 定长的方式

Aa,bb,cc,ok等这样的通讯方式

1. 定界符.helloworld|watchmen通过特殊的符号来区分消息。这样会出现粘包，半包的现象。

半包helloworld只接收到hello带来了不准确的消息。这样就应该丢弃。

1. 自定义协议包。

先定义包头，用于描述包的信息，包有多大。数据包的版本号，以及整个数据包的长度。（包头+包体）

包体：实际数据

全是一个流。

1. 自定义协议：

包头（length，flag（描述版本信息））

包体（content）

1. 通过客户端不断发送指定数目的定义数据包，在服务端解析，这个过程要解决半包的问题

可以将Channel理解为stream。

buffer本身就是一块内存，实际上是一个数组，数据的读写都是通过buffer来实现的。

Inbuffer buffer = IntBuffere.allocate(10);//给buffer分配大小

数据的对于写都是有buffer实现的。

数据来自于Channel，把数据从Channel中读到buffer中，再读取buffer。

通过

buffer.put()

buffer.filp();//状态的反转

buffer.get()；

对于服务器这种，关于心跳，heartBeat，服务器端比如zookeeper形成的集群，它们之间都会通过tcp方式进行节点与节点之间的通讯。主节点负责读写，从节点只负责读。因为主机器发生异常时，通过异步的方式把，节点与节点之间是最终一致性，而不是实时一致性。经过了几毫秒后，最终一致。节点有可能挂掉，所以需要每过几秒钟A向B发送一个心跳包，可以设置连续发送几次。这是长连接中最重要的一个保证网络是否中断的措施。服务端掉用closed方法，然后继续尝试连接。

如果客户端已经建立好了一个长连接，客户端开了飞行模式，这时服务端是感知不到连接关闭，是不会调用handlerRemoved方法的。

这时服务端向客户端继续推送消息时，经过一段时间后，服务器是会抛出异常的。

Channel ——Socket

EventLoop——控制流、多线程处理、并发

ChannelFuture——异步通知

Channel接口：

基本的I/O操作，bing()、connect（）、read（）和write（）依赖于底层网络传输依赖于底层网络传输所提供的原始。在基于Java的网络编程中，其基本的构造是class Socket。Netty的Channel接口所提供的API，大大地降低了直接使用Socket类的复杂性。此外，Channel也是拥有许多预定义的、专门化实现的广泛类层次结构的根，下面是一个简短的部分清单：

EmbeddedChannel；

LocalServerChannel；

NioDatagramChannel；

NioSctpChannel；

NioSocketChannel；

EventLoop接口：

EventLoop定义了Netty的核心抽象，用于处理连接的生命周期中所发生的事件。

一个EventLoopGroup包含一个或者多个EventLoop；

一个EventLoop在它的生命周期内只和一个Thread绑定；

所有由EventLoop处理得I/O事件都将在它专有的Thread上处理

一个Channel在它的生命周期内只注册一个EventLoop

一个EventLoop可能会被分匹配给一个或者多个Channel。

在这种设计中，一个给定Channel的I/O操作都是由相同的Thread执行的，实际上消除了对于同步的需要

Channel接口：

NIO中的Channel与Netty中的Channel不是一个东西

Netty重新设计了Channel接口，并且给予了很多不同的实现，Channel除了NIO中Channel所包含的网络I/O操作，主动建立、关闭连接及获取双方网络地址的功能外，还包含了Netty框架的功能，例如Channel的EventLoop/Pipeline等。

* Channel接口是能与一个网络套接字(或组件)进行I/0操作(读取\写入\连接\绑定)的纽带.
* 通过Channel可以****获取连接的状态****(是否连接/是否打开),****配置通道的参数****(设置缓冲区大小等),进行****I/O操作****

Channel的基本方法

id():返回此通道的全局唯一标识符.

isActive():如果通道处于活动状态并连接,则返回true.

isOpen():如果通道打开并且可能稍后激活,则返回true.

isRegistered():如果通道注册了EventLoop，则返回true。

config():返回关于此通道的配置.

localAddress():返回此通道绑定的本地地址.

pipeline():返回分派的ChannelPipeline.

remoteAddress():返回此通道连接到的远程地址.

flush():请求通过ChannelOutboundInvoker将所有挂起的消息输出.

ChannelFuture接口：

Netty所有的I/O操作都是异步的。因为一个操作可能不会立即返回，所以我们需要一种用于在之后的某个时间点确定其结果的方法。其addListener()方法注册了一个ChannelFutureListener，以便在某个操作完成时，无论是否成功，都得到通知。

由于Netty中所有的I/O操作都是异步的，因此Netty为了解决调用者如何获取异步操作结果的问题而专门设计了ChannelFuture接口。

ChannelFuture有两种状态：未完成（uncompleted）和完成（complete）

当令Channel开始一个I/O操作时，会创建一个新的ChannelFuture去异步完成操作。

未完成

Completed successfully |

+---------------------------+

+----> isDone() = true |

+--------------------------+ | | isSuccess() = true |

| Uncompleted | | +===========================+

+--------------------------+ | | Completed with failure |

| isDone() = false | | +---------------------------+

| isSuccess() = false |----+----> isDone() = true |

| isCancelled() = false | | | cause() = non-null |

| cause() = null | | +===========================+

+--------------------------+ | | Completed by cancellation |

| +---------------------------+

+----> isDone() = true |

| isCancelled() = true |

GenericFutureListener监听接口：

虽然可以通过ChannelFuture的get()方法获取异步操作的结果，但完成时间是无法预测的，若不设置超时时间，则有可能导致线程长时间被阻塞；如不能精确的设置超时时间则可能导致I/O操作中断，Netty建议通过****GenericFutureListener****接口执行异步操作结束后的回调.

public void channelRead(ChannelHandlerContext ctx, Object msg){

ChannelFuture future = ctx.channel().close();

future.addListener(new ChannelFutureListener(){

public void opereationComplete(ChannelFuture future){

}

});

}

另外,****ChannelFuture****允许添加一个或多个(移除一个或多个)****GenericFutureListener****监听接口,方法名:addListener(), addListeners(), removeListener(), removeListeners().

监听器对Future的扩展起到了很灵活的作用，当某个计算完毕，会触发相应的时间，得到Future的结果，因为jdk的get方法我们知道什么时候去掉，调早了需要等待，调晚了浪费了一段时间，还有isDone里边有2种情况，无法区分到底是正常的io完毕返回的true还是被取消之后返回的true，所以到了netty的Future里边加了一个isSuccess()方法，只有正常的io处理结束isSuccess()才返回true。

zookeeper是一个中间件，为分布式系统提供协调服务：

zookeeper的特性：一致性：数据一致性，数据按照顺序分批入库

原子性：事务要么成功要么失败。

单一视图：

可靠性：每次对zk的操作斗殴会保存在服务端。

实时性：客户端可以读取到zk服务端的最新数据

tickTime：用于计算的事件单元。比如session超时：N\*tickTime

initLimit：用于集群，允许从节点连接并同步到主节点（master）

的初始化连接时间，以tickTime的倍数来表示。

syncLimit：用于集群，master主节点与从节点之间发送消息，请求与应答时间。（心跳机制）。

dataDir：必须配置。zookeeper的日志位置

clientPort：连接服务器的客户端，端口默认为2181。

./zkServer.sh restart

Git学习：

仓库（Repository）

工作区 –》 暂存区 –》Git仓库

git status：查看文件所在位置

git add hello.php，把工作区的文件提交到暂存区

git commit –m “添加描述”，把暂存区的命令提交到Git仓库中生成一个新的版本

Git基础设置：

Git安装完成之后，需要进行一些配置

1、设置用户名

git –config –global user.name ‘libinjin’

2、设置用户邮箱

git config –global user.name ‘libin@youguu.com’

3、查看设置

git config -list

初始化一个新的Git仓库：

1、创建文件夹

2、在文件夹内初始化Git（创建Git仓库）

git init，后生成了一个.git文件

在git add 文件名后，

提交到Git仓库：

git commit –m ‘add a1.php’

将本地仓库同步到git远程仓库中：

git push

1.1.3 Zookeeper

Zookeeper = 文件系统+通知机制

1、Zookeeper是一个领导者（Leader）和多个跟随者（Follower）组成的集群。

2、集群汇总只要有半数以上节点存活，Zookeeper集群就能正常服务。所以一般服务器器的个数都是单数。

3、全局数据一致，每个Server保存一份相同的数据副本，Client无论连接到哪个Server上，得到的数据都是一致的。

4、更新请求顺序进来，来自同一个Client的更新请求按顺序依次执行。

5、数据更新的原子性，是典型的事务特定。

6、实现性，在一定时间范围内，Client能取到最新的数据，也是就每台Server上同步副本的速度非常快。因为数据量很小。

Zookeeper数据模型是一个典型的树形，每个Znode默认能够存储1MB大小的数据，每个Znode都可以通过其路径唯一标识。

应用场景：提供的服务包括，统一命名服务、统一配置管理、统一集群管理、服务器节点动态上线，这时客户端可以根据watcher监听得到服务器的状态，软负载均衡等。

Zookeeper虽然在配置文件中并没有指定Master和Slave。但是，Zookeeper工作时，是有一个节点作为Leader，其他则为Follower，Leader是通过内部的选举机制临时产生的。一般上来先选自己，当发现自己票数没有过半，就会把这一票投给id大的server，这是假如有5台服务器，Master技术第三台Server。

持久性：客户端和服务器端断开连接后，创建的节点不删除

持久化目录节点：客户端与Zookeeper断开连接后，该节点依然存在，

持久化顺序编号目录节点：客户端与Zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号。在分布式系统中，顺序号可以被用于为所有的事件进行全局排序，这样客户端可以通过顺序号推断事件的顺序。

短暂性：客户端和服务器端断开连接后，创建的节点自己删除。利用这种机制，可以实现服务器上线上下线通知，如果服务器掉了就能证明该服务器下线了。

xsync：同步分发，

配置zookeeper集群：在data目录中创建一个myid文件，编辑myid在文件中添加对应的server对应的编号。然后：

查看节点： ls /

创建普通节点： create /san “suguo”

创建临时节点：create –e /san/suoguo “liubie”

创建有编号的节点：create –s /sanguo/weiguo “caocao”

修改某个节点的值：set /san “lib”

节点值的变化监听：

get /path watch

一次修改会在一次监听中监听到，但是下次修改需要重新监听。

删除节点：delete /san/suguo

递归删除：rmr /san/suguo

查看节点的状态：stat /sanguo

监听器原理详解：

1、首先要有一个main线程

2、在main线程中创建Zookeeper客户端，这时就会创建两个线程，一个负责网络通信（connect），一个负责监听（listen）。

3、在main线程中创建zkClient端有Listener和connect，用客户端注册监听，然后服务端数据有变化，通过listener在process里面写的逻辑进行处理，在zookeeper的注册监听列表中将监听事件添加到列表中。zookeeper监听到有数据或者路径变化，就会将这个消息发送给Listenr线程。

4、常见的监听有，监听节点数据的变化，监听子节点增减的变化。

5、get path watch:监听节点数据的变化

6、ls path watch:监听子节点增减的变化

Client向zookeeper的Server1上写入数据，发送一个写请求。

如果Server1不是Leader，nameServer1会把接收到的请求进一步转发给Leader，因为每个zookeeper的Server里面有一个是Leader，这个Leader会将写请求广播给各个Server，比如Server和Server2，各个Server写成功后就会通知Leader。

监听服务器节点动态上下线：

1、某分布式系统中，主节点可以有多台，可以动态上下线，任意一台客户端都能实时感知到主节点服务器的上下线。

请简述zookeeper的选举机制：半数机制

zookeeper的监听原理：

zookeeper的部署方式有哪几种，集群中的角色有哪些，集群中最少需要几台机器？

1、部署方式有单点模式、集群模式

2、角色有leader和Follower

3、集群中最少需要3台及3台以上的机器

Zookeeper在dubbo中的应用。

Zookeeper是一个高性能，分布式的，开源分布式应用协调服务，它提供了简单原始的功能，分布式应用可以基于它实现更高级的服务，比如数据同步，配置管理，集群管理，命名空间。它被设计为易于编程，使用文件系统目录树作为数据模型。服务端跑在java上，提供java和C的客户端API。

分布式系统

1.多台计算机构成

2.计算机之间通过网络进行通信

3、彼此进行交互

4、共同目标

zookeeper介绍—协调者

协调：多个节点一起完成的一个动作

举例：集群成员管理

锁

选主

同步

发布/订阅

中间件，提供协调服务

作用于分布式系统，发挥其优势，可以为大数据提供服务

支持java，提供了客户端api，可以对节点进行增删改查，可以做触发器。

zookeeper的特性：1、一致性

2、原子性，事务要么都成功要么都失败，所有节点的原子性，不会几台机子成功，几台失败。

3、单一视图：客户端连接集群中的任一zk节点，数据都是一致的

4、每次对zookeeper的操作状态都会保存在服务端

5、客户端可以读取到zk服务端的最新数据

tickTime：用于计算的时间单元。比如session超时：N\*tickeTime

initLimit：用于集群，允许从节点连接并同步到master节点的初始化连接时间，以tickTime的倍数来表示

syncLimit：用于集群，master主节点与从节点之间发送消息，请求和应答时间长度。（心跳机制）超过一定时间，从节点就会被抛弃。

dataDir:必须配置的，zookeeper一些配置文件

dataLogDir：日志目录，如果不配置会和dataDir公用

Zookeeper基本数据模型：

1、是一个树形结构，类似于前端开发中的tree.js组件

2、每一个节点都被称之为znode，它可以所有子节点，也可以有数据

3、每个节点分为临时节点和永久节点，临时节点在客户端断开后消失

4、每个zk节点都有各自的版本号，可以通过命令行来显示节点信息，也就是节点的详情

5、每当节点数据发生变化时，该节点的版本号会累积，也叫做乐观锁。

6、删除、修改过时节点，版本号不匹配则会报错

7、每个zk节点存储的数据不宜过大，几K即可

8、节点可以设置权限acl，可以通过权限来限制用户的访问

客户端连接

查看znode结构

zk的作用体现一：

master节点选举，主节点挂了以后，从节点就会接手工作，并且保证这个节点是唯一的，这也就是所谓的首脑模式，从而保证我们的集群是高可用的。

统一配置文件管理，只需要部署一台服务器，则可以把相同的配置文件同步更新到其他所有服务器，此操作在云计算中用的特别多，假设修改了redis统一配置

zk的作用四：

提供了分布式锁，分布式环境中不同进程之间争夺资源，类似于多线程中的锁、

五、集群管理，集群中保证数据的强一致性，客户端不管是读取的主节点，还是从节点，读取到的数据都是一致的。

客户端与服务端之间的连接存在会话：

每个会话都会设置一个超时时间

心跳结束，session则过期，则临时节点znode会被抛弃

心跳机制：客户端向服务端的ping包请求，说明该客户端活着，不让服务端删除自己。

zk常用命令：

create命令：create -e /imooc/tmp imooc-data

get：查看，set：设置，delete：删除

ls / :查看命令

zk特性：watcher机制：

针对每个节点的操作，都会有一个监督者，wathcer相当于一个触发器，可以监听节点增删改操作。

当监控的某个对象（znode）发送了变化，则触发watcher事件

zk中的watcher是一次性的，触发后立即销毁，可以用Apache的一种把它做成永久性的。

watcher机制二：父节点，子节点增删改都会触发其watcher。

1.子节点创建事件

2.节点删除事件

3.节点数据变化事件

2.stat /lin watch

watcher使用场景：

1、统一资源配置

ACL（access control lists）权限控制：

1、可以对节点设置相关读写等权限，目的是为了保障数据安全性

2、权限permissions可以指定不同的权限范围及角色

ACL命令行：

getACL：获取某个节点的acl权限信息

setAcl：设置某个节点的acl权限信息

addauth：输入认证授权信息，注册时输入明文密码（登录）但是在zk的系统里，密码是以加密的形式存在的

Zookeeper从设计模式角度来理解，是一个基于观察者模式设计的分布式服务管理框架，它负责存储和管理有用的数据，然后接受观察者的注册，一旦这些数据的转态发生变化，Zookeeper就将负责通知已经在Zookeeper上注册的那些观察者做出相应的反应。

zookeeper中项目组件：

1. recipes：zookeeper典型应用场景的实现，它的实现是基于Curator Framework。
2. Framework，Zookeeper API的高层封装，大大简化了Zookeeper客户端编程，添加了Zookeeper连接管理，重试机制。
3. Utilities：为Zookeeper提供的各种实用程序。
4. Client：Zookeeper Client的封装，用于取代原生Zookeeper客户端。

1.1.4 RabbitMQ

MQ称为Message Queue，消息队列是应用程序和应用程序之间的通信方法。

RabbitMQ是一个开源的，在AMQP基础上完整的，可复用的企业消息系统。

AMQP是一个提供统一消息服务的应用层标准高级消息队列协议。

KafKa是一个分布式发布订阅消息系统。

同步变异步可以使用mq，还可以使用线程，在订单处理时，调用线程池来处理不同的请求，流量削锋，mq使用场景。

在rabbitMQ中，信息会从你的应用程序出来，来到Rabbit的队列，所有信息可以只存储在一个队列中，队列可以存储很多信息，因为它基本上是一个无限制的缓冲区

多个生产者可以将消息发送到一个队列中，多个消费者也可以只从同一个队列中取值。

fanout模式：

路由消息模式：

通配符模式：

自动声明：如果交换机不存在的时候，自动声明一个交换机，如果存在，就不用声明了。在使用的时候才去声明，前提是不存在时：auto-declare = true

消息非持久化设置：durable = “false”，关键在于服务重启后，是否还会存在。默认在Spring容器中的是持久化的。

# 1.1.5 Git学习

一、在本地拉取项目

1.先切换到将本地文件所在位置进入该文件夹后，执行

git init

命令把这个目录变成Git可以管理的仓库。

2.安装完成后，还需要最后一步设置，在命令行输入：

3.$ git config --global user.name "Your Name"

4.$ git config --global user.email "email@example.com"

2.然后执行git add readme.txt把想要管理的文件放入到仓库中需要两步。

1）git add readme.txt

2）git commit -m “wrote a readme file”,告诉git把文件提交到仓库中

-m 后面输入的是本次提交的说明

二、把本地项目上传到远程GitHub上：

1）先在本地项目文件夹中git init

2）然后将代码git add 并commit

3）在GitHub上建立自己的repository

4）将本地git项目上传到github上事先新建好的repository中：

5）进入工程文件夹所在目录(即前面创建本地仓库的位置)，右键Git Init Here，出现.git文件，是有关配置等功能的，不用管。然后到git bash here，依次输入以下命令：

git remote add origin git@github.com:github用户名/repository名.git

git pull git@github.com:github用户名/repository名.git

add commit push：

输入命令：git add .

add后面加了一个点，是想要提交所有文件，如果想提交指定的文件，可以写文件名，执行完增加命令后，要执行提交命令，如下：

commit：

输入命令：git commit –m “自定义项目名\_v1.0版本”

push：

输入命令：git push git@github.com:github用户名/自定义项目名.git

如果中途出现：

执行：git pull --rebase origin master让后继续push

3.当文件更改后，执行git diff查看文件修改的地方

5.利用git log命令显示最近到最远的提交历史git log --pretty=oneline aat.tx得到提交历史版本号及修改说明

6.版本回退，git reset --hard HEAD^这是将版本回退到上一个版本，如果要到上上一个执行git reset --hard HEAD^^，版本回退的前提是没有push到远程端。

7.版本前进,git reset --hard + 版本号码前五位，当你关闭了电脑后，再次查看可以git -reflog来查看所有提交过的版本号

工作区和暂存区：

工作区：它就是你在电脑里面能看到的目录，比如我的antherGit文件夹就是一个工作区。

版本库（Repository）

工作区有一个

隐藏目录 .git，这个不算工作区，而是Git的版本库

Git的版本库里面存了很多东西，其中最重要的成为stage（或者index）的暂存区，还有Git为我们自动创建的第一个分支master，以及master的一个指针叫做HEAD。

git add命令实际上就是把提交的所有修改放到暂存区（Stage），然后执行git commit就可以一次性把暂存区的所有修改提交到分支上去。

提交完后，工作区就是干净的。

git status

On branch master

nothing to commit, working tree clean

现在的版本库变成了这样，暂存区就没有任何内容了

删除文件：

在Git中，删除本地工作区文件后，如果该文件在版本库中存在，可以利用

git checkout test.txt将文件恢复到工作区中。同时也可以把文件恢复到直接最近add或者commit的状态。

命令git checkout -- readme.txt意思就是，把readme.txt文件在工作区的修改全部撤销，这里有两种情况：

一种是readme.txt自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；

一种是readme.txt已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次git commit或git add时的状态。

8.添加远程库后就能把本地文件上传到远程库中：

先需要在本地生成公钥和私钥，在连接远程库的途中需要输入：ssh-keygen -t rsa -C "libin@youguu.com"

git remote add origin git@github.com:libinjin/trade.git

上传到远程：git push -u origin master

8.克隆一个远程库到本地：

git clone git@github.com:libinjin/myReposity.git

切换到该库上：

cd myReposity/

9.创建分支：git branch dev 切换到该分支上：git checkout

git checkout –b dev//创建新的分支并切换到dev分支上

10.merge合并

把dev合并到master上，先要git checkout master上，然后执行：

git merge dev，合并dev到master上，当前在master上

在实际执行中，要执行：

git merge --no-ff dev

git merge –-abort//取消本次合并

删除某个分支：git branch –d <name>

11.查看本地和远程的所有分支

git branch –al

12.如果想要把一个文件拉取到本地某个分支上，先看本地有没有该分支，如果没有就要先创建一个分支，必须把以前的代码提交完后才能创建新的分支。

git checkout dev//创建本地分支dev

13.将远程分支文件下载到本地分支上

git pull origin dev

14.在本地给远程端新建一个分支

例如把本地分支dev创建到远程端：git push origin dev

15.查看两个分支的差异：

git diff branch1 branch2 --stat   //显示出所有有差异的文件列表

git diff branch1 branch2 文件名(带路径)   //显示指定文件的详细差异

git diff branch1 branch2                   //显示出所有有差异的文件的详细

16.Bug分支：在需要修改另外一个bug时，可在master上打一个分支

当手中的工作进行了一半，如果该工作现场并没有提交，可以先把这个工作现场stash（储藏）起来，但是这个工作现场需要先被add到stage区，然后恢复现场后继续工作。执行：git stash

当提交完以后，可以执行git stash list查看，就看不到任何stash内容了。

当修改完成后，可以将stage区的文件放出来，执行：git stash apply恢复以后，stash内容并不删除，但是你需要用git stash drop来删除。

另一种方式：git stash pop，恢复的同时把stash内容也删除了

17.推送分支

推送分支，就是把该分支上的所有本地提交到远程库。推送时，要指定本地分支，这样Git就会把该分支推送到远程对应的分支上。

git push origin master

本地master分支时主分支，因此需要时刻与远程同步。

dev分支时开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步。

18．当本地在执行push时，发现和远程端有冲突，报错：![rejected] master->master(fetch first)

error:failed to push some refs to 'https://github.com/xxx/xxx.git'

这时，先把远程端的代码与本地同步，同步完以后再提交到远程。

先执行git fetch,在本地把代码对比完后，再merge

git merge --no –ff origin/master

当在本地解决了冲突代码后，执行git commit -a

执行git push，将代码push到远程端。

19．tag标签（为了确定某一次的commit）

在发布一个新版本时，我们通常先在版本库中打一个标签（tag），这样就唯一确定打标签时刻的版本。Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像对不对？但是分支可以移动，标签不能移动），所以，创建和删除标签都是瞬间完成的。

Git有commit，为什么还要引入tag？

“请把上周一的那个版本打包发布，commit号是6a5819e...”

“一串乱七八糟的数字不好找！”

如果换一个办法：

“请把上周一的那个版本打包发布，版本号是v1.2”

“好的，按照tag v1.2查找commit就行！”

所以，tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。

打一个新标签：git tag v1.0

查看所打分支：git tag

默认标签就是打在最新提交的commit上的。有时候，如果忘了打标签，可以补打标签。

1）先查看历史提交版本：

git log --pretty=oneline --abbrev-commit

2）查看tag的详细信息：git show v0.1

标签总是和某个commit挂钩。如果这个commit既出现在master分支，又出现在dev分支，那么在这两个分支上都可以看到这个标签

创建一个标签：git tag –a <tagname> -m “标签信息”

删除本地tag，git tag –d v0.9

生成一个远程标签时，先要在本地生成这个标签，然后push到远程：

git push origin v1.0

        如上所示，使用命令 git push origin --delete Chapater6   可以删除远程分支Chapater6

           使用命令，git branch -d Chapater8 可以删除本地分支（在主分支中）

1.1.6 sql知识点

1.Spring实现数据库的读写分离

在大型的电子商务系统，在数据库层面大都采用读写分离技术，就是一个Master数据库，多个Slave数据库。

Master库负责数据更新和实时数据查询，Slave库负责非实时数据查询。因为在实际的应用中，数据库都是读多写少，读取数据的频率高，更新数据的频率相对较少，而读取数据通常耗时比较长，占用数据库服务器CPU较多。所以把查询从主库中抽取出来，采用多个从库，使用负载均衡，减轻每个从库的查询压力。

(1)采用读写分离技术目标：有效减轻Master库的压力，又可以把用户查询数据的请求分发到不同的Slave库，从而保证系统的健壮性。

实现方式：

1.定义两个数据库连接，一个是MasterDataSource，另外一个是SlaveDataSource。更新数据时我们读取MasterDataSource，查询数据时我们读取SlaveDataSource。

2.第二种方式动态数据源切换，就是在程序运行时，把数据源动态织入到程序中，从而选择读取主库还是从库，主要使用的技术是：annotation，spring AOP，反射。

@PostConstruct

被@PostConstruct修饰的方法在服务器架子Servlet的时候运行，并且只会被服务器执行一次。PostConstruct在构造函数之后执行，init（）方法之前执行。PreDestroy（）方法执行之后执行。

假设有一张表，表的数据有10万条数据，其中有一条数据是nickname=’css ’,如果要拿这些数据的话需要执行select \* from awared where nickname=’css ’一般情况下，在没有建立索引的时候，mysql需要扫描全表10万条数据中找，如果在nickname上建立索引，那么mysql只需要扫描一行数据及为我们找到该条数据。

mysql的索引分为单列索引（主键索引，唯一索引，普通索引）和组合索引。

单列索引：一个索引只包含一个列，一个表可以有多个单列索引。

组合索引：一个组合索引包含两个或者两个以上的列。

CREATE TABLE awared(

id int NOT NULL AUTO\_INCREMENT COMMENT ‘用户id’

aty\_id varchar(100) NOT NULL DEFAULT ‘’COMMENT ‘用户昵称’

)

单列索引：普通索引，这时最基本的索引

建立索引的方式一：CREATE INDEX account\_index ON awared(‘aty\_id’)

方式二：ALTER TABLE award ADD INDEX accouont\_index(‘aty\_id’)

唯一索引与普通索引类似，，不同的是唯一索引要求所有的类的值是唯一的，这一点和主键索引一样，但是他允许有空值。

其sql格式是CREATE UNIQUE INDEX indexName ON TableName

ALTER TABLE tableName add UNIQUE(column\_list)

(1)主键索引

它是一种特殊的唯一索引，不允许有空值。一般是在建表的时候同时创建主键索引：

代码如下:

CREATE TABLE user(

id int unsigned not null auto\_increment,

name varchar(50) not null,

email varchar(40) not null,

primary key (id)

);

(2)普通索引

这是最基本的索引，它没有任何限制。创建方式：

代码如下:

create index idx\_name on user(

name(20)

);

mysql支持前缀索引，一般姓名不会超过20个字符，所以我们这里建立索引的时候限定了长度20，这样可以节省索引文件大小

(3)唯一索引

它与前面的普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。创建方式：

复制代码 代码如下:

CREATE UNIQUE INDEX idx\_email ON user(

email

);

(4)复合索引

代码如下:

CREATE TABLE test (

    id INT NOT NULL,

    last\_name CHAR(30) NOT NULL,

    first\_name CHAR(30) NOT NULL,

    PRIMARY KEY (id),

    INDEX name (last\_name,first\_name)

);

name索引是一个对last\_name和first\_name的索引。索引可以用于为last\_name，或者为last\_name和first\_name在已知范围内指定值的查询。因此，name索引用于下面的查询：

SELECT \* FROM test WHERE last\_name='Widenius';

SELECT \* FROM test WHERE last\_name='Widenius' AND first\_name='Michael';

但是不能用于SELECT \* FROM test WHERE first\_name='Michael';这是因为MySQL组合索引为“最左前缀”的结果,简单的理解就是只从最左面的开始组合。

table：这是表的名字。

type：连接操作的类型。下面是MySQL文档关于ref连接类型的说明：

“对于每个来自于前面的表的行组合，所有有匹配索引值的行将从这张表中读取。如果联接只使用键的最左边的前缀，或如果键不是

UNIQUE或PRIMARY KEY（换句话说，如果联接不能基于关键字选择单个行的话），则使用ref。如果使用的键仅仅匹配少量行，该联接

类型是不错的。” 在本例中，由于索引不是UNIQUE类型，ref是我们能够得到的最好连接类型。 如果EXPLAIN显示连接类型是“ALL”，而且你并不想从表里面选择出大多数记录，那么MySQL的操作效率将非常低，因为它要扫描整个表。你可以加 入更多的索引来解决这个问题。预知更多信息，请参见MySQL的手册说明。

possible\_keys：

可能可以利用的索引的名字。这里的索引名字是创建索引时指定的索引昵称；如果索引没有昵称，则默认显示的是索引中第一个列的名字

（在本例中，它是“idx\_name”）。

Key：

它显示了MySQL实际使用的索引的名字。如果它为空（或NULL），则MySQL不使用索引。

key\_len：

索引中被使用部分的长度，以字节计。

ref：

它显示的是列的名字（或单词“const”），MySQL将根据这些列来选择行。在本例中，MySQL根据三个常量选择行。

rows：

MySQL所认为的它在找到正确的结果之前必须扫描的记录数。显然，这里最理想的数字就是1。 本例中未索引前遍历的记录数为1041，而建立索引后为1

Extra：

这里可能出现许多不同的选项，其中大多数将对查询产生负面影响。在本例中，MySQL只是提醒我们它将用using where，using index子句限制搜索结果集。

ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT=utf-8

在这个sql语句中，ENGINE=InnoDB是默认的

utf-8虽然在my.ini里面设置过了，但设置的是mysql的语言编码，而这里创建的时候不设置，就会出现乱码问题，二者的作用域不一样的，在创建表单的时候，这个charset会作用到这个表上面，它代表mysql遍历数据库数据表时设定字符集为utf-8。

mysql中engine=inndb和engine=myisam的区别

1、ISAM是一个定义明确且历经时间考验的数据表格管理方法，它在设计时就考虑到数据库被查询的次数要远大于更新的次数。因此，ISAM执行读取操作的速度很快，而且不占用大量的内存和存储资源。ISAM的两个主要不足之处在于，它不支持事务处理，也不能够容错，如果你的

2、InnoDB:这种类型是事务安全的.它与BDB类型具有相同的特性,它们还支持外键.InnoDB表格速度很快.具有比BDB还丰富的特性,因此如果需要一个事务安全的存储引擎,建议使用它.如果你的数据执行大量的INSERT或UPDATE,出于性能方面的考虑，应该使用InnoDB表，对于支持事物的InnoDB类型的标，影响速度的主要原因是AUTOCOMMIT默认设置是打开的，而且程序没有显式调用BEGIN 开始事务，导致每插入一条都自动Commit，严重影响了速度。可以在执行sql前调用begin，多条sql形成一个事物（即使autocommit打 开也可以），将大大提高性能。

#查看索引

show index from `share\_assistance`;

查看表信息，里面包括使用的引擎类型，字符编码，表结构等

mysql> show create table t1;--t1是表名

explain select \* share\_assistance，查看执行详细情况

如果分别在vc\_Name,vc\_City,i\_Age上建立单列索引，让该表有3个单列索引，查询时和上述的组合索引效率一样吗，大不一样，远远低于组合索引，虽然有三个索引，但mysql只能用到其中的那个它认为似乎最有效的单列索引。。

建立这样的组合索引，其实是相当于分别建立了

vc\_Name,vc\_City,i\_Age ,vc\_Name vc\_City,vc\_Name i\_Age

这样的三个组合索引为什么没有vc\_City,i\_Age等这样的组合索引，是因为mysql组合索引“最左前缀”的结果。简单的理解就是只从最左面的开始组合。并不是只要包含这三列的查询都会用到该组合索引。

使用索引的优点：

1、可以通过建立唯一索引或者主键索引，保证数据库表中每一行数据的唯一性

2、建立索引可以大大提高索引的数据，以及减少表的检索行数。

3、在表连接的连接条件可以加速表与表直接的相连。

4、在分组和排序字句进行数据检索，可以减少查询时间中分组和排序时所消耗的时间

5、建立索引，在查询中使用索引可以提高性能。

使用索引需要注意的地方：

1、在经常需要搜索的列上，可以加快索引的速度

2、主键列上可以确保列的唯一性

3、在表与表的连接条件上加上索引，可以加快连接查询的速度。

4、在经常需要排序order by，分组group by和distinct列上加索引可以加快排序查询的时间，单独order by用不了索引，索引考虑加where或加limit。

5、在一些where之后的< <= > >=Between in以及某个情况下的like建立字段的索引B-TREE

6、like语句的如果你对nickname字段建立了一个索引，当查询的时候的语句是nickname like ‘%ABC%’那么这个索引将不会起到作用，而nickname like ‘ABC%’会用到索引。

7、索引不会包含null列，如果列中包含null值豆浆不会被包含在索引中，复合索引中如果有一列含有null值，那么这个组合索引将会失效。

8、使用短索引，如果你的一个字段是Char(32)或者int(32)，在创建索引的时候指定前缀长度比如前10个字符（前提是多数值是唯一的），那么短索引可以提高查询速度，并且可以减少磁盘的空间，也可以减少I/O操作。

9、不要在列上进行计算，这样会使得mysql索引失效也会进行全表扫描

10、选择越小的数据类型越好，因为通常越小的数据类型在磁盘，内存，CPU，缓存中占用的空间很少，处理起来更快。

(六)什么情况下不创建索引

1.查询中很少使用到的列 不应该创建索引,如果建立了索引然而还会降低mysql的性能和增大了空间需求.

2.很少数据的列也不应该建立索引,比如 一个性别字段 0或者1,在查询中,结果集的数据占了表中数据行的比例比较大,mysql需要扫描的行数很多,增加索引,并不能提高效率

3.定义为text和image和bit数据类型的列不应该增加索引,

4.当表的修改(UPDATE,INSERT,DELETE)操作远远大于检索(SELECT)操作时不应该创建索引,这两个操作是互斥的关系

索引被用来快速找出在一个列上用一特定值的行，没有索引，mysql不得不首先以第一条记录开始读完整个表直到它找出相关的行。

mysql中的key和index的区别：

1、key时数据库的物理结构，它包含两层意义和作用：一是约束，偏重于约束和规范数据库的结构完整性，二是索引，辅助查询用的。

包括primary key，unique key，foreign key等。

primary key有两个作用，一是约束作用（constraint）,用来规范一个存储主键和唯一性，但同时也在此key上建立了一个主键索引；

unique key也有两个作用：一是约束作用constraint，规范数据的唯一性，但同时也在这个key上建立了一个唯一索引。

unique约束：唯一标识数据库表中的每条记录。unique和primary key约束均为列或列集合提供了唯一性的保证。每个 表可以有多个UNIQUE约束，但是每个表只能有一个primary key约束。

foreign key也有两个作用，一是约束作用（constraint），规范数据的引用完整性，但同时也在这个key上建立了一个索引。

可见，mysql的key是同时具有constraint和index的意义，这点和其他数据库表现的可能有区别。

mysql中key值（PRI，UNI，MUL）的含义：

PRI主键约束；UNI唯一约束；MUL可以重复

MySQL 中Index 与Key 的区别

Key即键值，是关系模型理论中的一部份，比如有主键（Primary Key)，外键（Foreign Key）等，用于数据完整性检否与唯一性约束等。

而Index则处于实现层面，比如可以对表的任意列建立索引，那么当建立索引的列处于SQL语句中的Where条件中时，就可以得到快速的数据定位，从而快速检索。

至于Unique Index，则只是属于Index中的一种而已，建立了Unique Index表示此列数据不可重复，猜想MySQL对Unique Index类型的索引可以做进一步特殊优化吧。

UNIQUE KEY的用途：主要是用来防止数据插入的时候重复的。

数据库优化的方法：

1、选取最合适的字段属性

在创建表的时候，尽量把表中的字段宽度设置的尽可能小。尽量把字段设置为not null，这样在将来执行查询的时候数据库不用去比较null

2、使用外键，锁定表的方法可以维护数据的完整性，但是它却不能保证数据的关联性。这时可以使用外键。

3、使用索引，索引是提高数据库性能的常用方法，索引应建立在那些将用于join，where判断和order by排序的字段上，尽量不要对数据库中某个含有大量重复的值的字段建立索引。

主键自动建立唯一索引，频繁作为查询条件的字段创建索引，查询中与其它表关联的字段创建索引，查询中排序的字段，查询中统计或者分组字段添加索引。

添加primary key（主键索引），添加唯一索引unique

添加index（普通索引）：alert table news add index index\_name

建立测试表t\_mobiles

Create table ‘t\_mobiles’(

‘id’ bigInt(20) NOT NULL AUTO\_INCREMENT,

‘userId’varchar(255) character set utf8,

‘mobile’varchar(24) not null default ‘’comment ’手机号’

‘createTime’datetime DEFAULT NULL COMMIT ‘创建时间’

PRIMARY KEY(‘id’),

KEY ‘联合索引’(‘userId’, ’mobile’)

)ENGINE=InnoDB AUTO\_INCREMENT=71185 DEFAULT CHARSET=utf8 COMMENT=’手机短息详情 ’

通过Explain查看执行计划来观察索引利用情况。

在创建复合索引时，应该仔细考虑列的顺序。对索引中的所有列执行搜索或仅对前几列执行搜索时，复合索引非常有用，仅对后面的任意列执行搜索时，复合索引则没有用处。

多个单索引在多条件查询时只会生效第一个索引，所以在多条件联合查询时最好建立联合索引，最左前缀原则，以最左边的为起点任何连续的索引都能匹配上，

注意：1.如果第一字段是范围查找，则需要单独建立一个索引。

2.在建立联合索引时，要根据业务需求，where子句中使用最频繁的一列放在最左边。

当同时存在联合索引和单列索引（字段有重复的），这时涉及到MySQL本身的查询优化器的策略，当一个表有多个索引可以走时，MySQL根据查询语句的成本来选择走哪个索引。

联合索引本质：当创建（a,b,c）联合索引时，相当于创单列索引，建了（a）单列索引，（a,b）联合索引以及（a,b,c）联合索引。想要索引生效的话，只能使用a和a,b和a,b,c三种组合。

联合索引比对每列分别建立索引更有优势，因为索引建立的越多就越占磁盘空间，在更新数据的时候速度会更慢。另外建立多列索引时，顺序也是需要注意的，应该将严格的索引放在前面，这样筛选的力度会更大，效率会更高。

ENGINE=InnoDB：将mysql数据库引擎设置为InnoDB。InnoDB表示带行锁定和外键的事务安全表。

4、优化查询语句，在查询时，不要使用 select \*。尽量避免使用in或者not in可以替换成between如果是子查询使用exists，还有在like开头添加%，避免进行null值的判断，避免在where条件中等号的左侧进行表达式、函数操作，避免使用or，尽量使用union代替手动创建临时表。以上都会优化都会避免数据库引擎放弃索引进行全表扫描。

5、使用view视图，简化查询，封装数据，提高查询效率。

6、对查询进行优化，赢尽量避免全表扫描，首先应考虑在where及order by涉及的列上建立索引。

7、应尽量避免在where子句中对字段进行null判断，否则将会导致引擎放弃使用索引而进行全表扫描，比如，select id from t where num is null可以在num上设置默认值0，确保表中的num列没有null值，然后查询select id from t where num=0;

8、应尽量避免在where后面使用!=或者<>操作符，避免在where子句中使用or来连接条件，如：select id from t where num=10 or num=20可以这样写：select id from t where num=10 union all select id from t where num=20

9、 in和not in也要慎重使用，比如select id from t where num in(1,2,3);

对于连续的数值，能用between就不要使用in了，select id from t where num between 1 and 3;

10、避免在where子句中对字段进行表达式操作，这样将导致引擎放弃使用索引而进行全表扫描。如：select id from t where num/2=100

应改为：select id from t where num=100\*2

11、应尽量避免在where子句中对字段进行函数操作，比如：select id from t where substring(name,1,3)=’abc ’,应改为select id from t where name like ‘abc%’

12、在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

13、尽量避免大事务操作，提高系统并发能力。

14、尽量避免像客户端返回大数据量，如果数据量过大，应考虑相应需求是否合理。

1.1.7 SpringBoot

Spring的发展：

Spring1.x时代：在Spring1.x时代，都是通过xml文件配置bean，随着项目的不断扩大，需要将xml配置分放到不同的配置文件中，需要频繁的在java类和xml配置文件中切换。

Spring2.x时代：

随着JDK 1.5带来的注解支持，Spring2.x可以使用注解对Bean进行申明和注入，大大的减少了xml配置文件，同时也大大简化了项目的开发。

最佳实践：

应用的基本配置用xml，比如：数据源、资源文件等；

业务开发用注解，比如：Service中注入bean等；

Spring3.x到Spring4.x

从Spring3.x开始提供了java配置方式，使用java配置方式可以更好的理解你配置的Bean，现在我们就处于这个时代，并且Spring4.x和Spring boot都推荐使用java配置方式。

Spring的java配置方式：

java配置方式就是为了取代之前xml配置的方式。

1、@Configuration

2、@Bean作用域方法上，相对于xml配置中的<bean>

SpringBoo使用“习惯优于配置”（项目中存在）大量的配置，此外还内置一个习惯性的配置，让你无需手动配置的理念让你的项目快速运行起来。使用Spring Boot很容易创建一个独立运行（运行jar，内嵌Servlet容器，内嵌了一个Tomcat）、准生产级别的基于Spring框架的项目，使用Spring Boot你可以不用或者只需要很少的Spring配置。尤其是集成第三方时非常的方便。

SpringBoot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程，该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。

SpringBoot的特定：

创建独立的Spring应用程序

嵌入的Tomcat无需部署war文件

简化Maven配置

自动配置Spring

提供生产就绪型功能，如指标，健康检查和外部配置

开箱即用，没有代码生产，也无需XML配置

SpringBoot特性理解：1、为基于Spring的开发提供更快的入门体验。开箱即用，没有代码生成，也无需XML配置。

2、同时也可以修改默认值来满足特定的需求。

3、提供了一些大型项目中常见的非功能特性，如嵌入式服务器、安全、指标、健康检测、外部配置等。

4、Spring Boot并不是对Spring功能上的增强，而是提供了一种快速使用Spring的方式。

二、配置fastjson（支持两种方法）

第一种方法就是：

启动类继承extends WebMvcConfigurerAdapter

覆盖方法configureMessageConverters

SpringBoot的优点：

优点：1、快速构建项目

2、对主流开发框架的无配置集成

3、项目可独立运行，无需外部依赖Servlet容器，减少部署过程

4、提供运行时的应用监控

5、极大提高了开发、部署效率

6、与云计算的天然集成、微服务（可以把一些子系统拆分成一个一个jar工程去运行）

7、SpringBoot是Spring家族的全新框架，它用来简化Spring应用程序的创建和开发过程，SpringBoot可以简化SSM框架的开发过程。它抛弃了对web.xml的配置，及对Spring、Mybatis的配置，采用了大量默认配置简化开发过程。

8、采用springBoot可以非常容易快速的创建基于Spring框架的应用程序，配置简单了，部署简单了，监控简单了。化繁为简。SpringBoot就是一个简化开发过程的一个框架。

Spring Boot的应用：

Spring boot必须设置一个parent。因为该parent包含了大量默认的配置，大大的简化了我们的开发。

导入spring boot的web支持

<dependency>

<groupId></groupId>

<artifactId></artifactId>

<>

</dependency>

添加spring boot的插件

SpringBoot核心：

入口类和@SpringBootApplication，Spring Boot的项目一般都会有\*Application的入口类，入口类中会有main方法，这是一个标准的java应用程序的入口方法。

SpringBootApplication:这是SpringBoot的核心注解，它是一个组合注解。

@SpringBootConfiguration：这是Spring Boot项目的配置注解，这也是一个组合注解。

在Spring Boot项目中推荐使用@SpringBootConfiguration替代@Configuration（类似于Spring容器的xml配置）。

2.@EnableAutoConfiguration：启用自动配置，该注解会开Spring Boot根据项目中依赖的jar包自动配置项目的配置项：

如：我们添加了spring-boot-starter-web的依赖，项目就会引入SpringMVC的依赖，Spring Boot会自动配置tomcat和SpringMVC。

里面有一个@ComponentScan会默认扫描含有SpringBootApplication注解的包下的所有类及该包的子包。

@ComponentScan():

在SpringBoot项目中可以使用全局配置文件application.properties或者是application.xml，在resources目录下或者类路径下的/config下，一般我们放到resources下。

webSocket：服务端和客户端一种长连接的形式进行通信。并且是服务器端向客户端推送的形式发送数据。

@ConditionalOnClass({ JedisConnection.class, RedisOperations.class，Jedis.class}):说明在必须有里面的类才能实例化该类

\*/

@Configuration

@ConditionalOnClass(GenericObjectPool.class)

protected static class RedisConnectionConfiguration { }

SpringBoot的web开发

web开发的自动配置类：org.springframework.boot.autoconfigure.web.webMvcAutoConfiguration

@ConditionOnMissingBean：当容器里面已经指定了该类就不在创建该类，当容器没有创建该类时就可以创建该类。

SpringMVC的配置：

在SpringMVC中已经不需要<mvc:annotation-driven/>，springboot中已经做了自动配置。

<context:component-scan base-package=”com.taotao.controller”/>springboot会自动扫描@Controller的类，所以不需要配置了。

@Bean可以将第三方法的框架整合到SpringBoot里面

SpringBoot的热部署：

Spring-boot-devtools是一个为开发者服务的一个模块，其中最重要的功能就是自动应用代码更改到最新的App上面去。原理是在发现代码有更改之后，重新启动应用，但是速度比手动停止后再启动还要快。

其原理是使用了两个ClassLoader，一个ClassLoader加载那些不改变的类第三方jar包，另外一个ClassLoader加载会更改的类，称为restart ClassLoader

这样在有代码更改的时候，原来的restart ClassLoader被丢弃，重新创建一个restart ClassLoader，由于需要加载的类相当比较少，所以实现了较快的重新启动时间。

1.当我们改变了方法的返回值，是能够进行热部署的

2.当重新创建了一个方法，是能够进行热部署的

3.当重新创建了一个类时，是能够进行热部署的

devtools会监听classpath下的文件变动，并且会立即重新启动应用，因为其采用的虚拟机机制，该项重启是很快的。

devtools可以实现页面热部署，即页面修改后会立即生效，这个可以直接在application.properties文件中配置

spring.thymeleaf.cache=false来实现，

什么是JPA：

JPA的全称是java Persistence API，是通过jdk 5.0或者XML描述对象关系表

的映射关系，并将运行期间的实体对象持久化到数据库中。

SpringBoot的特性：1、一个能够快速创建基于Spring的应用程序，

2、能够直接使用java main方法启动内嵌的Tomcat，Jetty服务器运行SpringBoot程序，不需要war包。

3、约定的starter POM来简化Maven配置，让Maven的配置变的更简单。

4、根据项目的Maven依赖配置，SpringBoot自动配置Spring、SpringMVC等

5、提供了程序的健康检查等功能。

6、基本可以完全不使用XML配置文件，采用注解配置。

SpringBoot四大核心：1、自动配置：针对很多Spring应用程序和常见的应用功能，SpringBoot能自动提供相关配置。

2、起步依赖：告诉SpringBoot需要什么功能，它就能引入需要的依赖库。

3、Actuator：健康检查，让你能够深入运行中的SpringBoot应用程序，一探SpringBoot程序的内部信息。

4、命令行界面：这是SpringBoot的可选特性，主要针对Groovy语言使用

SpringBoot开发环境：如果使用SpringBoot2.0，需要Spring 5.0，jdk 8.0,如果是eclipse，推荐安装Spring Tool Suite（STS）插件；Maven使用推荐 3.2+

第一个SpringBoot程序解析：1.Maven添加父级Spring-boot-starter-parent依赖的配置后，当前项目就是SpringBoot项目。

2、Spring-boot-starter-parent是一个特殊的starter依赖，它用来提供相关的Maven默认依赖，使用它后，常用的jar包依赖可以省去version配置。

3、如果不想使用某个默认的依赖版本，可以通过pom.xml文件的属性配置覆盖依赖项，比如覆盖Spring版本

<properties>

<spring.version>5.0.0.RELEASE</spring.version>

</properties>

4、@SpringBootApplication注解是SpringBoot项目的核心注解，主要作用是开启Spring自动配置的。

5、编写一个App.java，用于启动SpringBoot应用，

SpringBoot的核心配置文件：1、SpringBoot的核心配置文件用于配置SpringBoot程序，有两种格式的配置文件：

1、properties文件，application.properties是核心配置文件

2、.yml文件

配置实例

多环境配置文件

2、Springboot自定义配置文件：boot.name = 北京动力节点

boot.location=北京大兴

用@Value（“${boot.name}”）

springBoot下的SpringMVC：SpringBoot下的SpringMVC和之前的SpringMVC使用是完全一样的：

@Controller

@RestController：Spring 4以后新增的，是@Controller与@ResponseBody的组合注解，用于返回字符串或者json数据

@GetMapping：@RequestMapping（value=“/boot”,method=RequestMethod.GET）等价于@GetMapping(“/boot”)

@PostMapping，可以做一个增加或者修改时的操作

@PutMapping，做一个修改操作

@DeleteMapping，做一个删除操作，可以用GetMapping代替

SpringBoot和JSP的整合：

1、在开发JSP页面时在pom.xml中配置依赖项

<!--引入SpringBoot内嵌的Tomcat对JSP的解析包 -->

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

</dependency>

<!—Servlet依赖的jar包-->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api<artifactId>

</dependency>

<!—jsp依赖的jar包-->

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>javax.servlet.jsp-api<artifactId>

<version>2.3.1</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl<artifactId>

</dependency>

2、在application.properties文件配置spring mvc的视图为jsp

spring.mvc.view.prefix=/

spring.mvc.view.suffix=.jsp

3、在src/main下创建一个webapp目录，然后子该目录下新建jsp页面

4、在build中添加配置，为了能把静态资源打包到.class中

SpringBoot集成MyBatis的开发：

1、加载mybatis整合springboot

2、在SpringBoot的核心配置文件中配置mybatis的Mapper.xml文件所在的位置：mybatis.mapper-locations=classpath:com/djbc/springboot/mapper/\*.xml

3、在SpringBoot的核心配置文件中配置数据源

4、在mybatis的Mapper接口类上面添加@Mapper注释或者在运行的主类上添加@MapperScan（“ ”）注解包扫描

SpringBoot实现事务支持：

1、在入口类中使用注解@EnableTransactionManagement开启事务支持

2、在访问数据库的Service方法上添加注解@Transactionl即可。

SpringBoot实现Restful API:

认识Restful，SpringBoot开发Restful:

restFull是一种互联网架构设计风格，它并不是标准，它只是提出了一组客户端和服务器交互时的架构理念和设计原则，基于这种理念和原则设计的接口可以更简洁，更有层次，任何的技术都可以实现这种里面。比如访问一个http接口：

http://localhost：8080/api/order?id=121&status=1

采用RESTFUL俸给的地址为：http://localhost:8080/api/order/1012/1

具体实现：1、用@PathVariable，获取url中的数据，该注解只是实现RESTFull最主要的一个注解。

2、增加post方法

3、删除delete方法

4、修改put方法

5、查询get方法

SpringBoot热部署添加配置：

SpringBoot集成Redis的步骤：

1、添加配置jar依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

2、在SpringBoot核心配置文件中application.properties中配置redis的连接信息。

spring.redis.host=localhost

spring.redis.port=6379

spring.redis.password=123456

3、配置了上面的步骤，SpringBoot将自动配置redisTemplate，在需要操作redis的类中注入redisTemplate。在redis中存值也是序列化的结构

SpringBoot和Dubbo的集成：

1、加入springboot与dubbo集成的起步依赖

2、在SpringBoot的核心配置文件application.properties中配置dubbo的信息。

spring.dubbo.appname=springboot-dubbo-provider

spring.dubbo.registry=zookeeper://localhost:2181

3、由于zookeeper做注册中心，所以添加zookeeper的客户端jar包

4、 编写dubbo的接口实现类

在实现类里面添加注解：

@Component

@Service(version=“1.0.0”,timeout = 10000)等价于<dubbo:service interface=””/>//这个注解是dubbo的，不是spring的

5、在main方法上添加：@EnableDubboConfiguration//开启dubbo的自动配置

6、在消费这边：用@Reference//<dubbo:reference id=”” interface=””/>

springBoot配置拦截器：

**1.1.8 微服务**

分布式计算：

分类：同步，异步

通讯协议：Http、自定义协议（ProtoBuffer，thrift自定义协议）

通讯协议：Http、自定义协议

传输介质：文本（二进制+编码转化过来的）、二进制Binary

微服务面临的挑战：

1. 注册与发现（所有注册的节点需要被注册中心发现，节点有主动被注册，有被动被注册）
2. 路由（有一个RPC服务，1.为了负载均衡2.服务层可以根据协议进行相应的路由）
3. 可靠性（进程里面的调用是很可靠的，网络上十个节点有九个是可用的）
4. 延迟，由于远程的操控（网络延迟），以及序列化，反序列化这个过程，所以有延迟
5. 热点（这个功能点有很多处调用，所以用负载均衡来解决热点的问题）
6. 短路（为了保护大多数功能不受影响）
7. 伸缩（和容器机器部署有关）
8. 异步（springCloud中叫做消息，异步）
9. 监控
10. 配置（分布式及本地配置，文件修改，要做相应的更新）
11. 数据同步（数据和业务上数据的一致性）
12. 安全（服务和服务之间是否有权限调用，单点登录）

元编程：

在java编程中，元编程是一种新型的编程模式，目的是为了减少代码行数，达到事半功倍的效果。

主要模式：1、注解驱动

2、反射驱动

3、表达式驱动

4、Lambda

5、Script On JVM（Groovy、JavaScript等）

# 1.1.9 MySQL的分区

什么是分区：分区就是将一个表分解成多个区块进行操作和保存，从而降低每次操作的数据，而对应用来说都是透明的，从逻辑上来看是只有一个表（这里跟分库分表的访问不一样），但在物理上这个表可能是由多个物理分区组成的，每个分区都是独立的对象，可以进行独立处理。

分区能干什么：

1、分区后，分割的数据能够有多个不同的物理文件路径，即放在

了不同的磁盘路径下。

2、可以存储更多的数据，突破系统单个文件最大限制，可以扩容。

3、提升性能，提高每个分区的读写速度，提高分区范围查询的速度。

4、可以通过删除相关分区来快速删除数据

5、通过跨多个磁盘来分散数据查询，从而提高磁盘I/O的性能。

6、涉及到SUM（）和COUNT（）函数时，会分区全表扫描的。

7、可以对分区中的表进行备份和恢复独立的分区，这对大数据量很有好处。

分区能支持的引擎：MyISAM（不支持事务）,InnoDB等，不支持MERGE和CSV等。同一个分区表中的所有分区必须是同一个存储引

从MySLQ5.1开始引入分区功能。

分区类型：

1、RANGE分区：基于属于一个给定连续区间的列值，把多行分配给分区。

2、LIST分区：类似于按RANGE分区，LIST是列值匹配一个离散集合中的某个值进行选择。

3、HASH分区：基于用户定义的表达式的返回值来进行选择的分区，该表达式使用将要插入到表中的这些行的列值进行计算，这个函数必须产生非负整数值。通过HASH运算来决定应该放在哪个分区中。

4、KEY分区：类似于按HASH分区，由MySQL服务器提供其自身的哈希函数。

但是不论什么类型的分区，都要注意一下问题：

1、如果表中存在primary key或者unique key时，分区的列必须是primary key或者unique key的一个组成部分，也就是说，分区函数的列只能从pk或者uk这些key中取子集。

2、如果表中不存在任何的primary key或者unique key，则可以指定任意一个列作为分区列

3、mysql5.5版本前的Range、List、Hash分区要求分区键必须是int，5.5及以上版本，支持分整型的Range和List分区。

create table tbl\_users2(

uuid int not null,

customerId varchar(20),

name varchar(100)

);

按照UUID 进行List型分区：

PARTITION BY List(uuid)(

PARTITION p0 VALUES in(1,3,5),

PRRTITION p1 VALUES in(7,9,10),

PRRTITION p1 VALUES in(7,9,10)

)

PARTITION BY RANGE(UUID)(

PARTITION p0 VALUES LESS THAN(5),

PARTITION P1 VALUES LESS THAN(10),

PARTITION P2 VALUES LESS THAN(11),

PARTITION P3 VALUES LESS THAN MAXVALUE

)

SELECT \* FROM tb\_user2 partition(p2);//产看在p2分区上的

1、如果试图操作的列值不在分区值列表中时，那么会失败并报错。要注意的是，LIST分区没有类似如“VALUES LESS THAN MAXVALUE”这样的含义其他值在内的定义，将要匹配的任何值都必须在列表中找到。

2、LIST分区除了能和RANGE分区结合起来生成一个复合的子分区，与HASH和KEY分区结合起来生成复合的子分区也是可以的。

3、Hash分区：Hash分区主要用来确保数据在预先确定数目的分区中平均分布。在RANGE和LIST分区中，必须明确指定一个给定的列值或列值集合以指定应该保存在哪个分区中。而在HASH分区中，MySQL自动完成这些工作，要做的只是基于将要被哈希的列值指定一个表达式，以及指定备份区的表将要被分割成的分区数量，如：

PARTITION BY HASH(uuid)

PATITIONS 3;//将UUID除以3后按照余数分区

子分区：子分区时分区表中每个分区的再次分割，适合保存非常大量的数据。

mysql分区事项：1、最大分区数目不能超过1024，一般对单表的分区数不要超过150个。

3、如果含有唯一索引或者主键，则分区列必须包含在所有的唯一索引或者主键在内

4、不支持主键

5、不支持全文索引，对分区表的分区键创建索引没那么这个索引也将被分区。

一般不建议对分区再进行子分区。

分库分表：

数据库的复制(读写分离)能解决访问问题，但并不能解决大规模的并发写入问题，由于无法进行分布式部署，而一台服务器的（CPU、磁盘、内存、IO等）是有限的，最终数据库所能承载的数据量、数据处理能力都将遭遇瓶颈，所以对这台master服务器进行分库分表操作（把写入多的master进行拆分）。

复制：一台机器成master，另外一台机器成slave，用于做主从，slave会从master复制数据，可以一主多从。一般写要比读少。读会很多，这样就把读的压力分散到多台上了。高并发大规模的写入还是要靠master写入的。所以就要

1、解决磁盘系统的最大文件限制，比如常见的有：

FAT16(最大分区2GB)

2、减少增量数据写入时的锁对查询的影响，减少长时间查询造成的表锁，影响写入操作等锁竞争的情况，节省排队的时间开支，增加吞吐量。

3、由于单表数量下降，常见的查询操作由于减少了需要扫描的记录，使得单表单词查询所需的检索行数变少，减少了磁盘IO，时延变短。

什么是分库，分库又叫垂直切分，就是把原本存储于一个库的表拆分存储到多个库上，通常是将表按照功能模块、关系密切程度划分出来，部署到不同的库上。

如果数据库是因为表太多而造成海量数据，并且项目的各项业务逻辑划分清晰，之间没有太多的关联。低耦合，那么规则简单明了、容易实施。

分库的优点是：实现简单，库与库之间界限分明，便于维护，缺点是不利于频繁跨库操作，单表数据量大的问题解决不了。

什么是分表：分表又叫做水平切分，是按照一定的业务规则或逻辑，讲一个表中的数据拆分成多份，分别存储在多个表结构一样的表中，这多个表可以存在一到多个库中，分表有分成垂直分表和水平分表。

单库的分区就是变相的在一个库中进行的分表。

垂直分表：将本来可以在同一个表的内容，人为划分为多个表。所谓的本来，是指按照关系型数据库的第三范式要求，是应该在同一表的。把一张表的多个字段分到表一，另外多个字段分到表二，等等。

水平分表，也被成为数据分片：是把一个表复制成同样表结构的不同表，然后把数据按照一定的规则划分，分别存储到这些表中，从而保证单表的容量不会太大，提升性能，当然这些结构一样的表，可以放到一到多个库中。

1、大数量时，先考虑读写分离。

2、如果数据量继续增大时，考虑垂直分库，把一个库分成多个库

数据库中有表1/2/3/4 --》数据库1存表1、数据库2存表2、数据库3存表3/4,= --如果库3数据量又大，给其做slave，

如果还存不下时，就进行单库单表的操作。一个库就只放一张表。--》再多，进行分区(相当于单表水平分)。--》再多进行水平分表，进入多个数据库中。

分表的优点：能解决分库的不足点，但是缺点是实现起来比较复杂，特别是分表规则的划分，程序的编写，以及后期的数据库拆分移植维护。

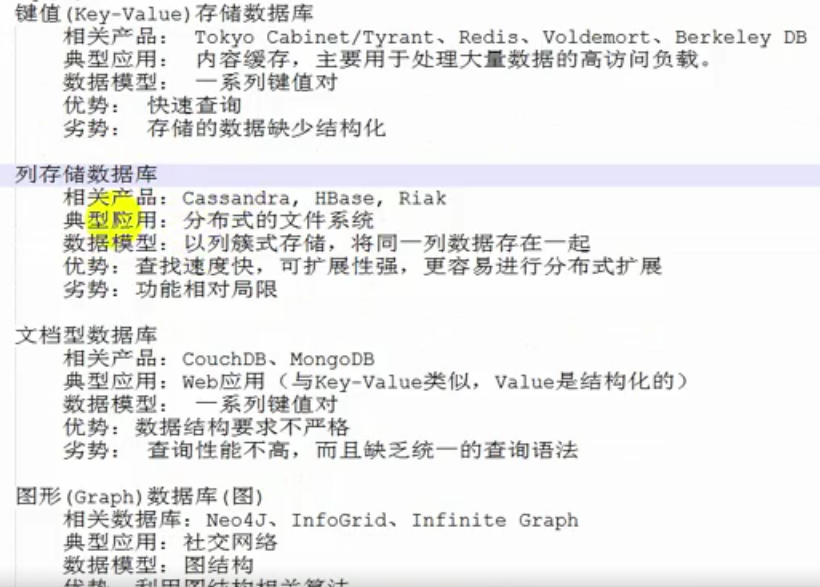
一般都是先分库再分表，两者结合使用

如何分库：分析业务功能，以及表间的聚合关系，把关系紧密的表放在一起（主表和子表不能主外键不能跨库参照）。

数据库读写分离：使绝大部分数据读操作访问都可以不通过数据库就可以完成，但仍有一部分读操作（缓存访问不命中，缓存过期）和全部的写操作需要访问数据库，在网站的用户达到一定规模后，数据库因为负载压力过高而成为网站的瓶颈，这就需要实现数据库读写分离。

给主数据库写数据，从数据库不知道时，这时需要进行主从复制。

分布式文件系统：HDFS



优势：快速建立各个节点之间的关系。

redis主要应用于高速内容缓冲，同时支持数据持久化和事务管理。

redis支持多种数据类型：string，hash,list,set,sortedset

redis是c语言写的，所以需要先编译。

# 1.2.0 Redis

sortedset：概述sortedset中的每一个成员都会有一个分数score与之关联，Redis正是通过分数来为集合中的成员进行从大到小的排序。

在sortedset中添加、删除或更新一个成员都是非常快速的操作，其时间复杂度为集合中的成员数量的对数。由于sorted-set。

添加元素：zadd key score

按照分数进行排序：zrange mysort 0 -1 withscores

按照分数进行倒叙：zrevrange key start stop withscores

按照分数排序：zrangebyscore mysort 0 -1

keys的通用操作：

keys \*

del key1

exists key：判断key是否存在

选择库：select 1

移库：move key 1：将当前库的key移植到1号库中

dbsize：返回当前数据库中key的数目。

info：获取服务器的信息和统计

flushdb：删除当前选择数据库中的所有key

flushall：删除所有数据库中的所有key

消息订阅及发布：

subscribe channel：订阅频道，

psubscribe channel\*：批量订阅频道，例如：psubscribe s\*，订阅以“s”开头的频道

publish channel content：在指定的频道中发布消息。

redis中的事务：

multi：开启事务用于标记事务的开始，其后执行的命令都将被存入命令队列，直到执行exec时，这些命令才会被原子的执行，类似于关系型数据库中的：begin Transaction

exec：提交事务，类似于commit

discard：事务回滚，类似于rollback

在redis中如何保证数据的安全性：

redis持久化，redis的高性能是由于其将所有数据都存储在了内存中，为了使redis在重启之后仍能保证数据不丢失，需要将数据从内存中同步到硬盘中，这一过程就是持久化。

redis中支持两种方式的持久化，一种是RDB方式，一种是AOF方式，可以单独使用其中一种或者将两种结合使用。

1.RDB持久化（默认）：快照的形式，该机制是将内存中的数据集快照写入磁盘。

2.AOF持久化：该机制将日志的形式记录在服务器上所处理的每一个写操作，在Redis服务器启动之初会读取该文件来重新构建数据库，以保证启动后数据库中的数据是完整的。

3.无持久化

我们可以通过重置的方式禁用Redis服务器的持久化功能。这样就可以视其为一个加强版的memcached了。

数据多的时候用RDB。RDB是在某个时间点存一次快照，可能会丢失数据，如果一段时间内服务器挂了，该段时间内的操作的数据就没有保存上。

在Linux上安装的软件远程访问就必须要开放端口。

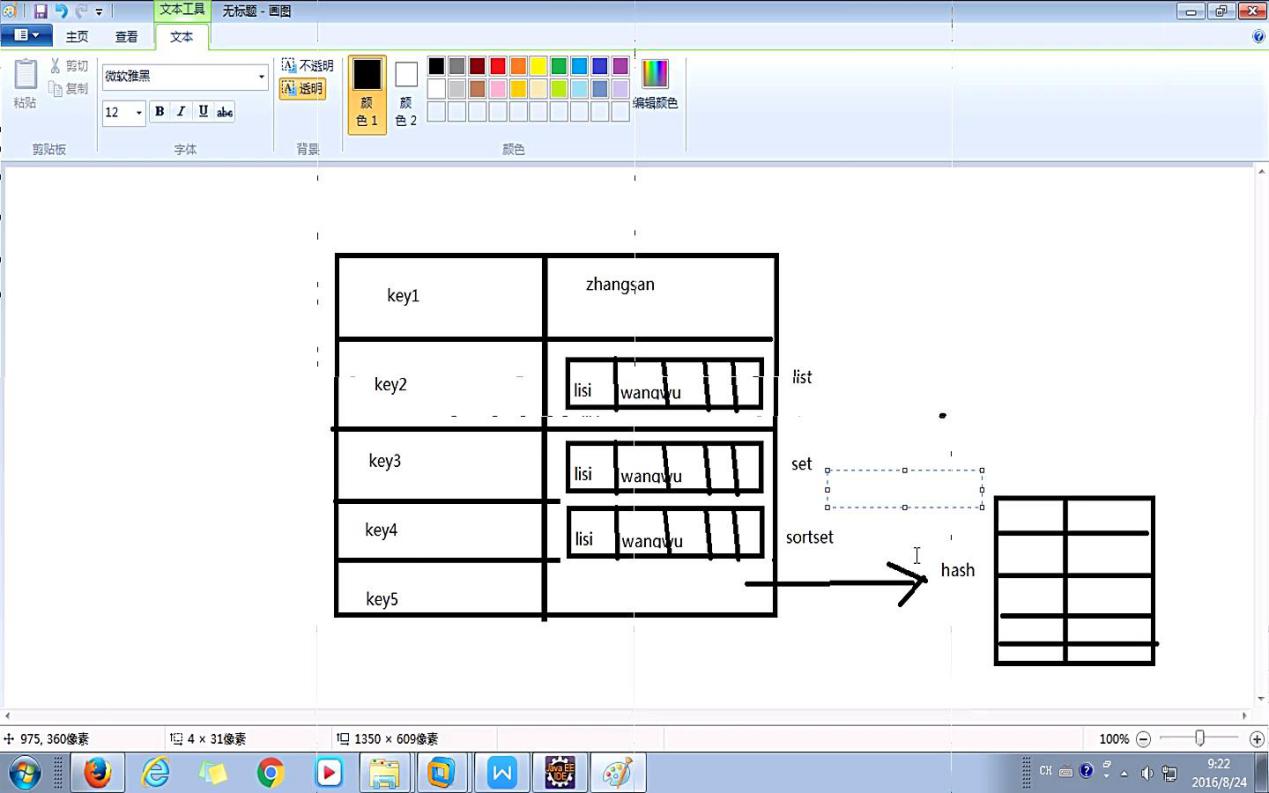
/sbin/iptables –I INPUT –p tcp –dport 6379 –j ACCEPT

#将配置永久保存

/etc/rc.d/init.d/iptables save

Redis是用C语言开发的一种高性能键值对（key-value）数据库，通过key进行数据的取值。它通过提供许多种键值数据类型来适应不同场景下的存储需求，目前为止Redis支持的键值数据类型如下：

1. 字符串类型string
2. 散列类型Map
3. 列表类型List
4. 集合类型set
5. 有序集合类型sortset



关于key的定义：1、key不要太长，也不要太短、有一个统一的命名规范。

字符串类型是Redis中最为基础的数据存储类型，它在Redis中是二级制安全的，这就是说存入和取出的数据相同，在Redis字符串类型的Value最多可以容纳的数据长度是512M

命令：set key value

取值：get key

getset key “”

删除 key：del key

Hash:

命令：hset myhashKey1 name1 tom

gset myhashKey1 name1

hmset myhashKey1 age 15 address beijing //设置多个

hmget myhashKey1 age addr //取出多个

取出全部：hgetall myhashKey

删除：hdel myhashKey age 删除某个key下的filed

删除整个key：del key

List：

Ipush key values[ value1 value2…]：在指定的key所关联的list的头部插入所有的values，如果该key不存在，该命令在插入的之前创建一个与该key关联的空链表，之后再向该链表的头部插入数据。插入成功，返回元素的个数。

Irange key start end：获取链表中从start到end的元素的值，

lpop key:返回并弹出指定的key关联的链表中的第一个元素，即头部元素。出栈，pop后该元素从list中弹出。

lpushx key value：仅当参数中指定的key存在时，向关联的list的头部插入value，如果不存在，将不进行插入。

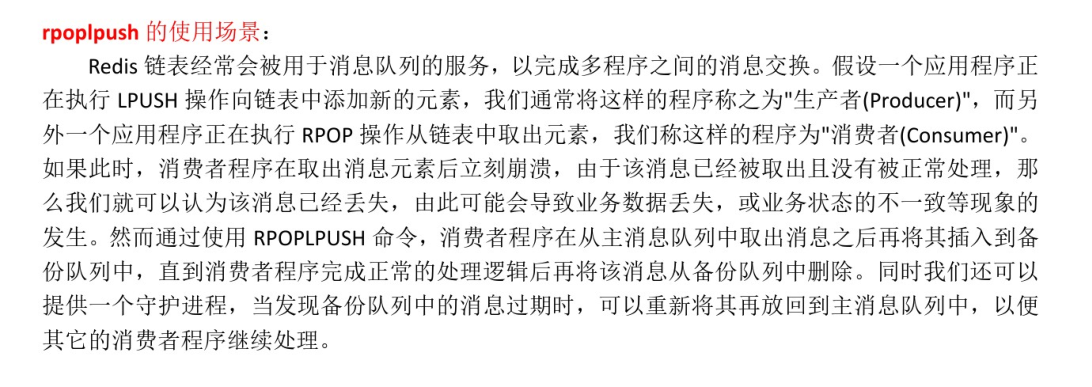
rpushx key value：在该list的尾部添加元素。

lrem key count value：删除count个值为value的元素，如果count大于0，从头到尾遍历并删除count个值为value的元素。

lset key index value：设置链表中的index下标的元素值为value

linsert key before | after pivot value：在元素pivot之前或者之后添加元素value

rpoplpush resource destination：将链表中的尾部元素弹出并添加到头部，循环操作



添加/删除元素

sadd key values [value1 value2 value3]：向set中添加数据，如果该key的值已有则不会重复添加。

srem key value1：删除set中指定的成员

smembers key：获取set中所有的成员

sismember key member：判断member这个值是否存在，如果存在返回1,如果改值不存在或者该key本身就不存在，无论集合中有多少元素都可以急速的返回结果。

两个set取差集：

sdiff key1 key2

sdiff key1 key2：两个集合取差集，属于A并且不属于B的集合构成的元素，顺序不一样，得到的值也就不一样。

集合的交集运算：

sinter key1 key2

使用场景：

可以使用redis的set数据类型追踪一些唯一性的数据，可以用set存储session（每个用户的空间），如果是不同步的数据，就存到sort里面。

redis可以统一管理session会话，可实现单点登录。

Sorted-Set和Set类型极为相似，它们都是字符串的集合，都不允许重复的成员出现在一个set职工，它们之间的主要差别是sorted-set中的成员都会有一个分数score与之关联，redis正是通过分数来为集合中的成员进行从大到小的排序。

唯一、有效

为每个元素都加了一个权重，在排序的时候按照权重去排序。

添加：zadd mysort 100 a 50 b 80 c

查看：zrange mysort 0 -1 查看全部

zrange mysort 0 -1 withscores //根据分数给mysort排序查看

zrevrange mysort 0 -1 withscores//根据分数给mysort倒序排列

zrangebyscore mysort 0 100 withscores limit 0 2//取前两

key的使用：

key pattern：获取所有与pattern匹配的key，返回所有与该key匹配的keys。\*表示任意一个或者多个字符，？表示任意一个字符

exists myset：判断该key是否存在。

rename key newkey

expire key：设置过期时间

ttl key：获取该key所剩的超时时间，如果没有设置超时，返回-1，如果有则返回-2。

type key：获取指定key的类型。该命令以字符串的格式返回。

返回的字符串为string、list、set、hash和zset，如果key不存在返回none。

选择库：select 1 //默认是选择的0库

移库：move mylist 1//把0库的key等于mylist移到1号库

flushdb，删除当前选择数据库中的所有key

flushall，删除所有数据库中的所有key

消息订阅与发布：

subscribe channel：订阅频道，例：subscribe mychat，订阅mychat这个频道

publish channel content，发布消息，当发布了内容后，订阅该频道立即

redis中的连接：

redis-cli -h 192.168.3.101（ip地址） -p 6379 （端口号）

数据持久化：

RDB方式：

save the DB on disk

save 900 1

save 500

持久化文件名称：dump.rdb

持久化文件目录：dir ./

当触及rdb持久化策略时，此时将会fork（创建）一个子进程

1. 同时会拍摄当前数据库的快照信息（SNAPSHOT）
2. 父进程继续处理服务端其他业务，由子进程处理SNAPSHOT的持久化存储
3. 当子进程持久化结束，则自动销毁。

rdb缺点：可能会丢失最后一次的数据。

aof模式：将每次写操作存储到aof持久化文件中，默认不开启。

redis.conf文件中的配置 appendonly 修改为yes

持久化方式二：

appendonly yes

将用户每一步的操作都记录下来

rdb和aop同时设置时，也会按照AOP来记录、优先读取aof，aof中不能有异常，会导致启动报错。

主从复制（读写分离）

1. 为了避免单点故障(当数据库挂了，仍然后其他数据库提供服务)：设置一个主数据库（Master）和n个从数据库（Slave），所有的写操作都在主数据库执行，读操作在从数据库执行。为保证数据的一致性，将数据库的数据会复制到从数据库中。

复制时不会阻塞主数据对连接的处理，采用多线程处理，主从复制采用优化的二进制协议，效率非常高。

1. 设置主从复制
2. 使用命令

slaveof master ip port （在内存中设置，重启后就不存在了）

1. 使用配置文件

slaveof 192.168.2.101 6379 （写在redis.conf配置文件中，永久保存）

查看主从配置：info replication

# 1.2.1 Elasticsearch

高度伸缩的开源全文本搜索和分析引擎，能够快速地存储、搜索、分析大数据，并且接近于实时。通常用于复杂的搜索功能和需求。

常见使用场景：

1. 文本搜索
2. 日志搜索
3. 数据分析

包括索引、映射、搜索、聚合，Elasticsearch的集群、分词。

Elasticsearch是一个基于Lucene构建的开源、分布式、restful接口全文搜索引擎。Elasticsearch还是一个分布式文档数据库，其中每个字段均是被索引的数据且可被搜索，它能够扩展至数以百计的服务器存储以及处理PB级的数据。他可以在很短的时间内存储、搜索和分析大量的数据。他通常作为具有复杂搜索场景下的核心发动机。

Elasticsearch就是为高可用和可扩展而生的。可以通过购置性能更强的服务器来完成，称为垂直扩展或者向上扩展，或增加更多的服务器来完成，称为水平扩展或者向外扩展。

尽管ES能够利用更强劲的硬件，垂直扩展毕竟还是有它的极限。真正的可扩展性来自于水平扩展，通过向集群中添加更多的节点来分担负载，增加可靠性。

当有大量数据，千万条以上的记录时，你有商业智能分析的需求，希望快速调查、分析和可视化。在这种情况下，可以用Elasticsearch来存储你的数据，然后用Kibana建立自定义的仪表板或者任何你熟悉的语言开发展示界面，用它的聚合功能来执行复杂的商业智能与数据查询。

GitHub的搜索是基于Elasticsearch构建的，共有40-50个索引库，分别用于索引网站需要跟踪的各种数据。虽然只索引项目的的主分支master，但这个数据量依然巨大，包括20亿个索引文档，30TB的索引文件。

Elasticsearch的优点：

横向可扩展性：只需要增加一台服务器，做些配置，启动一下Elasticsearch进程既可以并入集群中。

分片机制提供更好的分布性：同一个索引分为多个分片（sharding）,这点类似于HDFS的块机制，分而治之的方式可提升处理效率。

全文搜索：

全文搜索是指计算机搜索程序通过扫描文章中的每一个词，对每一个词建立一个索引，指明该词在文章中出现的次数和位置，当用户查询时，搜索程序就根据事先建立的索引进行查找，并将查找的结果反馈给用户。这个过程类似于通过字典中的搜索字表查字的过程。

大数据技术栈：Hadoop，Spark，Storm，Hive，HBASE，Zookeeper，Elasticsearch，数据仓库，数据挖掘，

Elasticsearch：

什么是搜索，

如果用数据做搜索会怎么样



用数据库搜索性能会很差

什么是全文检索和Lucene

倒排索引：

生化危机电影 --》生化 危机 电影

生化危机海报

生化危机文章

生化危机新闻

关键词 ids

生化 1,2,3,4

危机 1,2,3,4

电影 1

海报 2

文章 3

新闻 4

数据库中扫描100万次每次还要匹配关键词。

用es进行检索出有该关键词的就直接返回，速度非常快。

全文检索：

Lucene：是一个jar包，里面包含了封装好的各种建立倒排索引，以及进行搜索的代码，包括各种算法，在java开发时，引入Lucene的jar包，然后就能直接进行开发了，用Lucene可以将已有的数据建立索引，Lucene会在本地磁盘上面，给我们组织索引的数据结构。

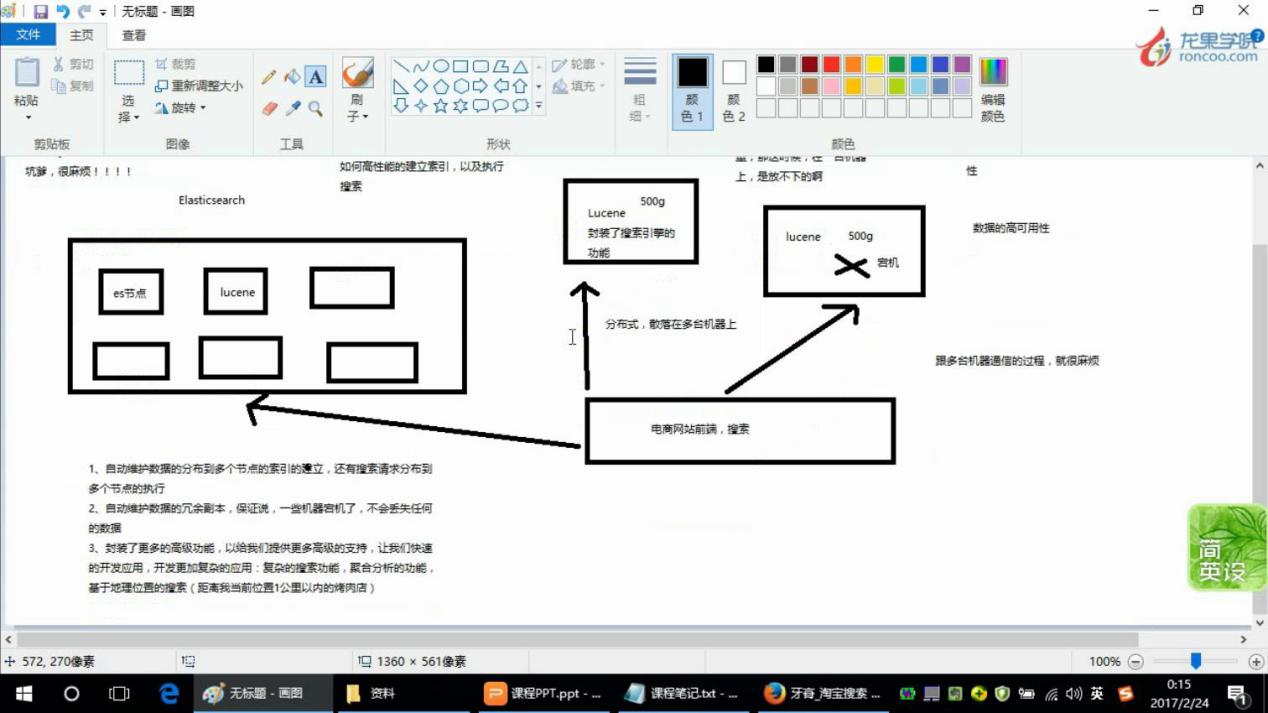
什么是Elasticsearch

部署在单台机器上时，假设磁盘就500G空间，当数据量很大时，有1T的数据量，那时候，在一台机器放不下，就需要做分布式，散落在多台机器上，跟多台机器通信的过程就很麻烦，保证数据的高可用性，及管理给那块放，从那块取，如果一台机器宕机，会丢失数据。

如果数据量很大时，超过单台机器的承受范围，你就必须用多台机器进行数据的存储和搜索，如果我们自己实现，就很麻烦。所以Elasticsearch就应用而生，

可以建立一个一个es的节点（封装了一个一个的Lucene），作为一个集群，统一对外提供服务。

1. 可以自动维护数据的分布到多个节点的索引的建立，还有搜索请求分布到多个节点的执行。
2. 自动维护数据冗余副本，保证说，一些机器宕机了，不会丢失任何的数据
3. 封装了更多的高级功能，给我们提供了更多高级的支持，让我们快速的开发应用，开发更加复杂的应用，复杂的搜索功能，聚合分析的功能，基于地理位置的搜索（距离我当前位置1公里以内的烤肉店）。



Elasticsearch的功能：

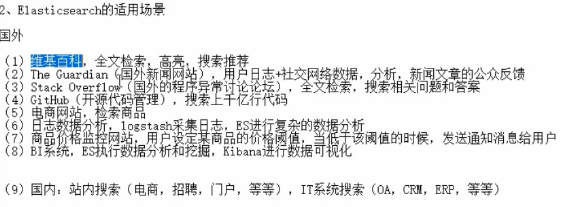
1、分布式的搜索引擎和数据分析引擎

2、全文检索，结构化检索select \* from emp where name like %sdf%

数据分析：分析每一个商品分类下有多少个商品，select category\_id,count(\*)

from produts group by category\_id查询每种商品的数据

1. 对海量数据进行近实时的处理，近实时，检索数据花费时间长的叫做离线批处理，实时就是秒级别的。

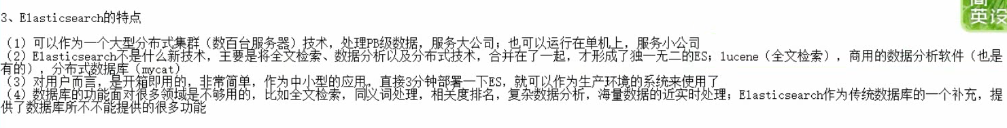


日志数据分析：logstash采集日志，ES进行复杂的数据分析，ELK技术，Elasticsearch+logstash+kibana

商品价格监控网站，类似于股票预警，发一条消息。

BI系统，商业智能，Business Intelligence,比如分析一下某某区域最近3年的用户消费金额的趋势以及用户

群体的组成构成，产出相关的数张报表。ES执行数据分析和挖掘，Kibana进行数据可视化。



Elasticsearch的核心概念：

1. Near RealTime（NRT），近实时。
2. Cluster，集群，包含多个节点，每个节点属于哪个集群是通过一个配置（即集群名称）来决定的。
3. Node：节点，集群中的一个节点，节点也有一个名称、
4. Document：文档，es中的最小数据单元，一个document可以是一条任意的json格式的数据，每个index下的type中，都可以存储多个document。

product Document

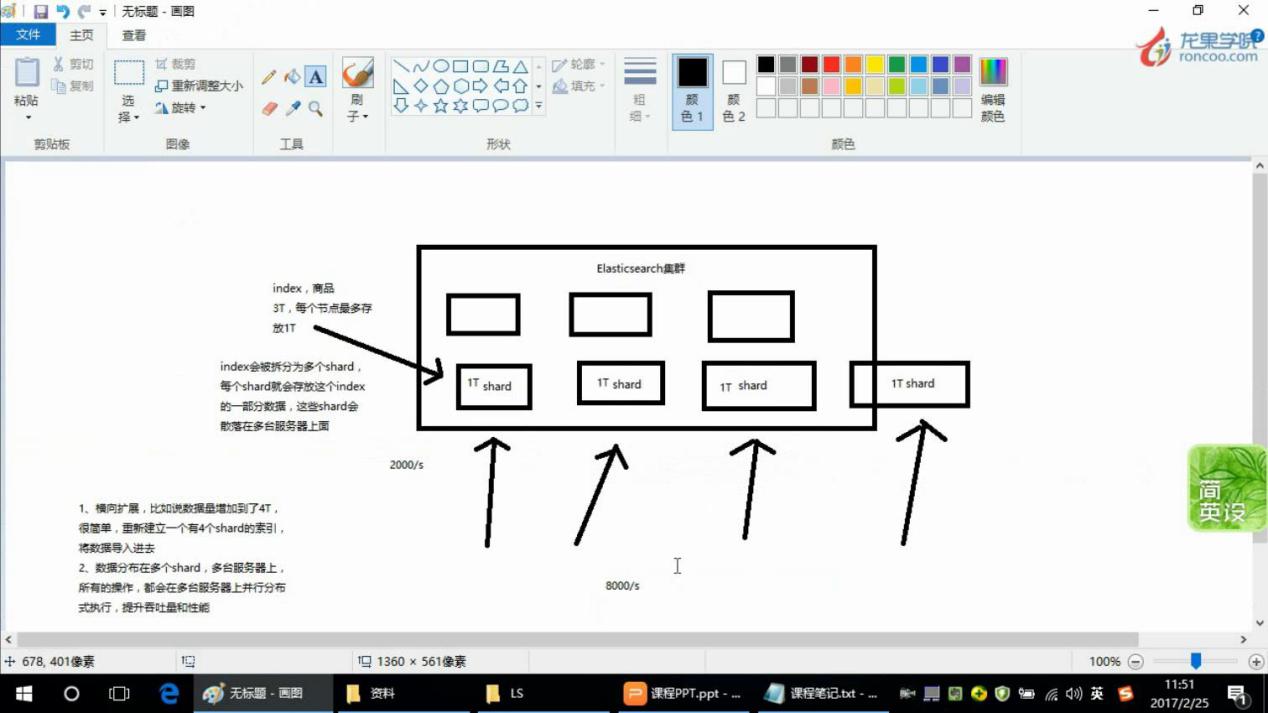
{

“product\_id”:“1”,

“product\_name” : “”,

“product\_desc” : “”

}

1. Index，索引，包含一堆有相似结构文档数据，比如可以有一个客户索引，商品分类索引，订单索引，索引有一个名称。一个Index中可包含多个document。里面可能存放了所有的商品数据，包括所有的商品document
2. type，每个索引有多个type，定义了不同种类的document
3. 

shard为了分片，缓解一台上面的访问压力，进行横向扩展，每个shard分布在多台机器上。

replica：任何一个服务器随时可能故障或宕机，此时shard可能会丢失，因此可以为每个shard创建多个replica副本，其可以在shard出故障的时候提供服务，1.高可用性的体现，一个shard宕机，数据不丢，服务继续提供

2.提升了搜索的吞吐量

应用系统的数据结构是面向对象的，复杂的。

对象数据存储到关系型数据库中，只能拆解开来，变成扁平的多张表，每次查询的时候还得还原对象格式，相当麻烦。

ES是面向文档的，与面向对象的数据结构一样。基于这种文档数据结构，es可以提供复杂的索引，全文索引，分析聚合等功能。

如何快速了解集群的健康状态：

green：每个索引的primary shard和replica shared都是active状态。

快速查看索引：

创建索引：

PUT /test\_index?pretty

PUT /索引名称/类型/id

PUT /ecommerce/product/5

{

"name": "jiaqiang bbb",

"desc": "a good man",

"age": 26,

"tags": [

"qingxin"

]

}

PUT也可以修改，修改的时候必须把所有的field都传入，不管是否想要修改，如果没有传入就会被覆盖，就会丢失。必须全量的修改，不能增量的修改。

POST才可以单个的修改：

POST /ecommerce/product/3/\_update（这是修改的API）

{

“doc”:{

“name”: “libin"

}

}

6种搜索方式：

1. query string search
2. query DSL
3. query filter
4. full-text-search
5. phrase search
6. highlight search（高亮）

1.GET /ecommerce/product/\_search

GET /ecommerce/product/\_search?q=desc:a&sort=age:desc

按照desc分组并按照age排序

返回结果：

{

"took": 12,

"timed\_out": false,//是否超时

"\_shards": {

"total": 5,//总共拆分成了5个分片，所以对于搜索请求，会打到所有的primary shard

"successful": 5,

"failed": 0

},

"hits": {//查询的document的数据

"total": 2,

"max\_score": 1,//score的含义就是document对于一个search的相关度的匹配分数，分数越高，越相关，就越匹配

"hits": [

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "AWtp2eAgWnQAd4P6vh4l",

"\_score": 1,

"\_source": {

"doc": {

"name": "xiaoqiang"

}

}

},

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "3",

"\_score": 1,

"\_source": {

"name": "aaaa",

"desc": "a good man",

"age": 21,

"tags": [

"qingxin"

]

}

}

]

}

}

1. Domain Specified Language,特定领域的语言DSL

http request body，请求体，可以用json的格式来构建查询语法，比较方便，可以构建各种复杂的语法，比query string search强大的多。

查询所有的商品：

GET /ecommerce/product/\_search

{

“query”:{

“match\_all”:{}

}

}

}

GET /ecommerce/product/\_search

{

“query”:{

“match”:{

“name”: ”yagao”

}

}

}

2.根据name等于aaaa进行查询并且根据年龄排序

GET /ecommerce/product/\_search

{

"query":{

"match":{

"name":"aaaa"

}

}，

“sort”：[

{“age”: ”desc”}

]

}

GET /\_cat/indices?v 查看所有的索引

分页查询：

GET /ecommerce/product/\_search

{

"query":{

"match":{

"name":"bbb" //select \* from like name %bbb%

}

}

, "from": 1,

"size": 3

}

GET /ecommerce/product/\_search

{

"query":{

"match\_all": {//select \* from order by age asc limit 1,2

}

},"sort": [

{

"age": "asc"

}

]

, "from": 1,

"size": 2

}

查询只包含商品的名称及价格字段

GET /ecommerce/product/\_search

{

“query”: {

“match\_all”:{} select

},

“\_source”:[“name”, “price”] //name,price from

}

1. 搜索商品名称包含yagao，而且售价大于25元的商品

GET /ecommerce/product/\_search

{

“query”: {

“bool”:{

“must”:{//必须包含的 where name = ‘’

“match”: “yaogao”

}

},

“filter”:{

“range”:{

“price”: { “gt ”: 25 }// and price>25

}

}

}

}

GET /secretary/info/\_search

{

"query": {

"term": {

"code": "600001"

}

}

}

GET /secretary/info/\_search //进行多匹配

{

"query": {

"multi\_match": {

"query": "你好",

"fields": ["answer","question"]

}

},

"from": 0,

"size": 2

}

1. full-text-search（全文检索）

producer这个字段，会被拆解，建立倒排索引

special id:4

yagao id:4

producer id:1,2,3,4

gaolujie id:1

zhonghua 3

jiajieshi 2

因为关键字在id等于4时出现了两次，所以

get /ecommerce/products/\_search

{

"query":{

"match": {

“producer”: ”yagao producer”

} //select \* from where producer like %yagao producer%模糊查询

}

}

1. phrase search（短语搜索）和全文索引刚好相反，必须全部匹配上才行

get /ecommerce/products/\_search

{

“query”:{

“match\_phrase”:{

“producer”: ”yagao producer”select \* from where producer like %yagao producer%，必须全部匹配上

}

}

}

1. 高亮，把字标红

GET /ecommerce/products/\_search

{

"query": {

"match": {

"producer": "yagao producer"

}

},

"highlight": {

"fields": {

"producer":{}//把匹配出的该字段标红

}

}

}

计算每个tag下的商品数量group by tags sum(\*)分组，每组的数量。

GET /ecommerce/products/\_search

{

“size”: 0,//得到的结果包含原始数据，所以去掉原始数据

“aggs”:{

“group\_by\_tags”:{//自定义分组名称

“terms”:{

“field”:”tags”

}

}

}

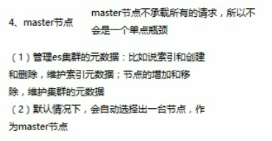
按照指定的价格范围区间进行分组，然后再每组内按照tag进行分组，最后再计算每组的平均价格。

Elasticsearch对复杂分布式机制的透明隐藏特性：

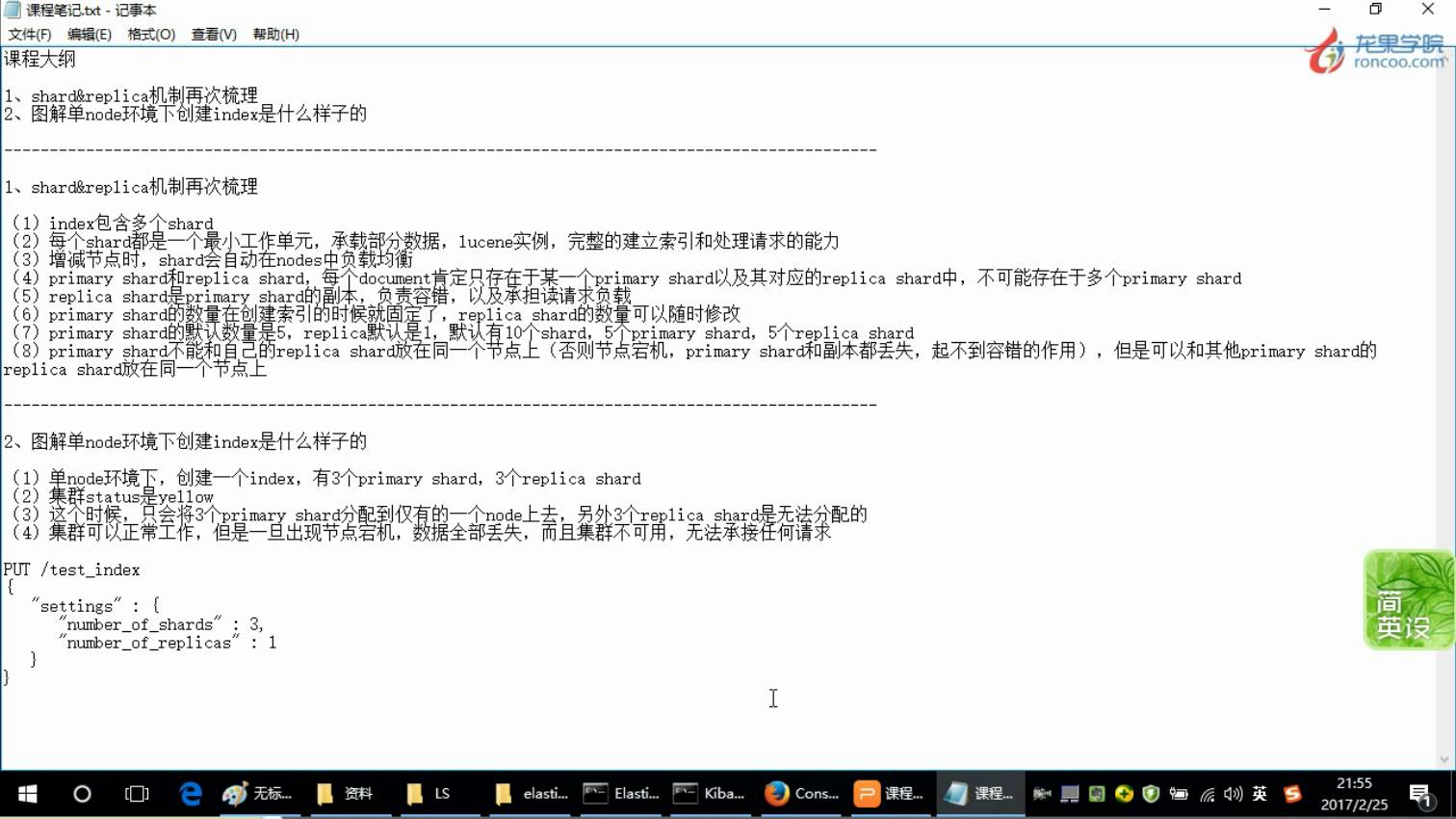
Elasticsearch是一套分布式的系统，分布式是为了应对大数量，隐藏了复杂的分布式机制，

分片机制，我们之前随便就将一些document插入到es集群中去，我们没有在意数是如何分到哪个shard中去的。cluster discovery集群发现机制，shard负载均衡，shard副本，请求路由，集群扩容，shard重新分配

master节点要做的：



1. 节点对等的分布式架构



每个shard可以看做一个Lucene单元。在增加节点时，node会自动负载均衡。

每个shard都分为primary shard和replica shard。

一个document只存在一个shard里面。

replica主要是用来容错的，是和primary shard不能放在同一个节点中的。

1. \_index元数据
2. 代表一个document存放在哪个index中
3. 类似的数据放在一个索引，非类似的数据放不同索引
4. index中包含了很多类似的document
5. 索引必须是小写的，不能下划线开头，不能好友逗号

类似的数据建一个索引，因为type为product的可能在查找数据，sale类型的需要分析数据，这样就会造成product读的慢。在自己独立的shard中，与其他数据不在一个shared中，就不能互相影响。

1. \_type元数据

逻辑上对index中有些许不同的几类数据进行分类，因为一批相同的数据，可能有很多相同的fields，比如商城里面有电子产品，生活用品。

1. \_id元数据

代表document的唯一标识，与index和type在一起，可以唯一标识和定位一个document

可以手动指定一个id，也可以让es自动生成一个id

1、根据应用情况，

2、自动生成id

POST /test\_index/test\_type

{

“test\_content”: ”my test”

}

自动生成的id，长度为20,个字符，URL安全，base64编码，GUID算法保证分布式系统并行生成时不可能发生冲突

定制\_source

GET /ecommerce/products/3?\_source=tags

GET /ecommerce/products/3?\_source=tags,name

1、document的全量替换PUT，类似于insert into on duplicate key update

但是这个是全量的，不是增量的。

2、document的强制创建

1）PUT，有时候只想创建不想替换，

PUT /index/type/id?op\_type=create

PUT /test\_index/test\_type/4/\_create

{

"test\_field": "my test"

}

Elasticsearch中并发冲突问题：

悲观锁并发的控制方案：

创建于关系型数据库中，比如：牙膏100件，有线程A，线程B，当A线程读取时，B线程是读不到数据的。当A线程取出完一件时，B线程才能读取到数据，

悲观锁就是在各种情况下都上锁，上锁之后只有一个线程可以操作这条数据。当然，不同的场景下，上的锁不同，有行级锁，表级锁，读锁，写锁。通过锁控制线程之间并发执行导致数据不准确。并发效率低，同一时间只能有一条线程操作数据。

ES中使用的是乐观锁控制方案。

写的时候，会判断当前数据的version版本号，是不是和数据库中的版本号一致，都是version=1。

在线程A修改了数据后，就会将version版本号加一，然后此时线程B去判断当前线程数据的版本号，version=1，与数据库中的版本号等于2不一致，说明已经修改过，此时用户B不会用99去更新。而是重新去es中读取最新的数据版本，99件再次减一。

1.select version from table;

2.update set num = 99，version = version + 1 where id= #{id} and version = #{version}

但是乐观锁会导致死锁。

事务A执行：

update set num = 99 where id = 1

update set num = 99 where id = 2

事务B执行：

update set num = 99 where id = 2

update set num = 99 where id = 1

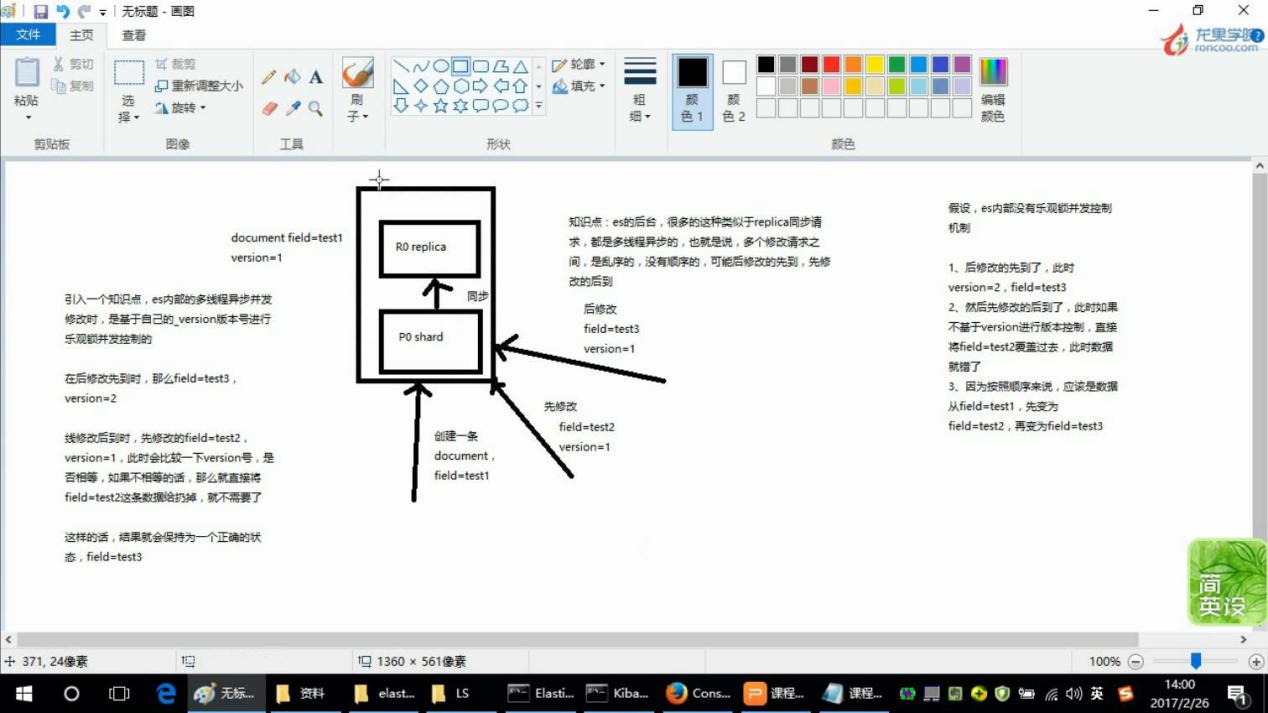
这时会互相等待对方释放锁，造成死锁。

解决办法：净两条执行顺序一致即可。

事务A在执行完第一条update的时候，刚好事务B也执行完第一条update

此时， 事务A中order表中的id = 1的行被锁住， 事务B中order表中id = 2的行被锁住，两个事务继续往下执行

事务A中第二条update执行需要order表中id = 2的行数据，而事务B中第二条update执行需要id = 1的行数据， 两条update往下执行的条件都需要对方事务中已经被锁住的行，于是陷入无限等待，形成死锁。



# 1.2.2 Docker

Docker是一个轻量级容器技术，类似于虚拟机技术。Docker是直接运行在当前操作系统（Linux）之上，而不是运行在虚拟机中，但是也实现了虚拟机技术的资源隔离，性能远远高于虚拟机技术。

Docker支持将软件编译成一个镜像（image），在这个镜像里做好对软件的各种设置，然后发布这个镜像，使用者可以运行这个镜像，运行中的镜像称之为容器container，容器的启动是非常快的，一般都是以秒为单位。这个有点像我们平时安装的ghost操作系统，系统安装好后软件都有了，虽然完全不是一种东西，但是思路是类似的。

目前各大主流云计算平台都支持Docker容器技术，包括阿里云、百度云平台（资源隔离通过Docker实现）、Cloud Foundry。

hadoop集群搭建：

1. 单节点：
2. 伪分布式
3. 完全分布式
4. HA（高可用）
5. 联邦模式

三、准备机器

1.至少3台机器

1、配置静态ip

2、配置主机名称

3、配置主机映射（ip和hostname）

4、创建同名的用户配置sudoer权限，

配置/etc/sudoers文件，将同一个用户名配置

hadoop01 ALL=(ALL) ALL

这样，hadoop用户就具有了sudoer权限，意味着hadoop用户可以通过sudo + 命令来执行一些原来只有root才可以执行的命令

5、同步时间

6、配置ssh免密登录（利用普通用户）

1）生成密钥对：ssh-keygen ,需要回车三次

2)把公钥发送给其他机器：ssh-copy-id ip

用ssh登录，ssh + ip ,退出：exit

7、selinux：安全secrity-enhanced linux安全增强的linux

vim /etc/selinux/config

三种情况：

enforce：强制的，必须严格遵循的

permissive：宽容的，如果不遵循会一直打印警告信息

disable：禁用的

8、关闭防火墙

搭建Hadoop集群：

1. 上传到linux系统
2. 查看目录结构

bin：二进制可执行文件（Hadoop/hdfs/mapred/yarn）

sbin：开启/关闭集群

etc：Hadoop配置文件

Hadoop-env.sh/core-site.xml/hdfs-site.xml/yarn-site.xml/mapred-sit.xml/slaves

运行方法：

bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.7.jar grep input output 'hdfs'

1. 伪分布式

以进程分离方式运行，hdfs，yarn集群中需要进程都可以启动

所有的进程都在一个节点上运行。

完全分布式：

hdfs集群：1nn + n dn + 1 2nn

yarn集群：1 rm + n nm

规划：

hadp01: nn dn nm

hadp02: rm dn nm

hadp03:2nn dn nm

1. 上传hadoop软件包
2. 解包解压缩
3. 配置

hadoop-env.sh中配置export JAVA\_HOME

core-sit.xml

1. 分发到每个节点
2. 启动集群
3. 测试

groovy讲解之groovy初探

为什么选择groovy及其优势