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学习

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。

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：

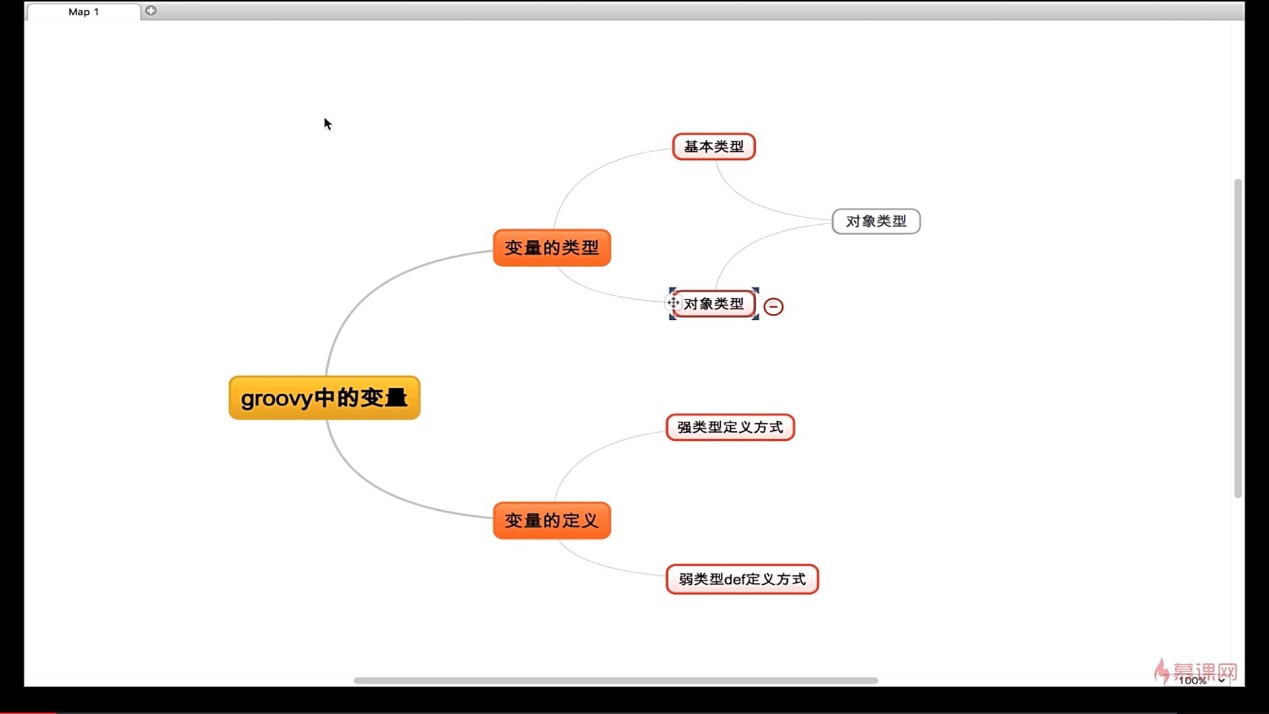
groovy在Linux上开发环境的配置：

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

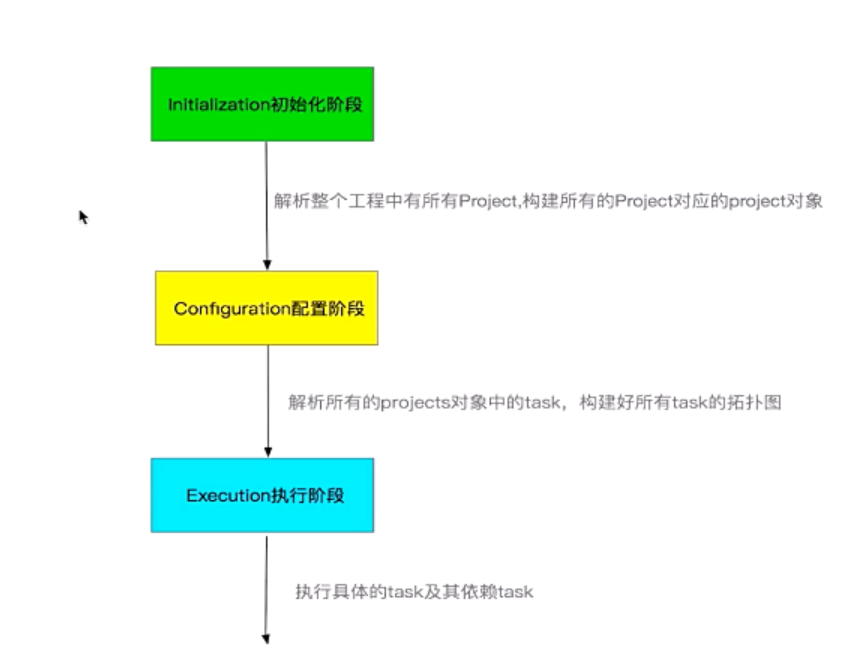
gradle的优势：灵活性，可以自行扩展脚本。

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

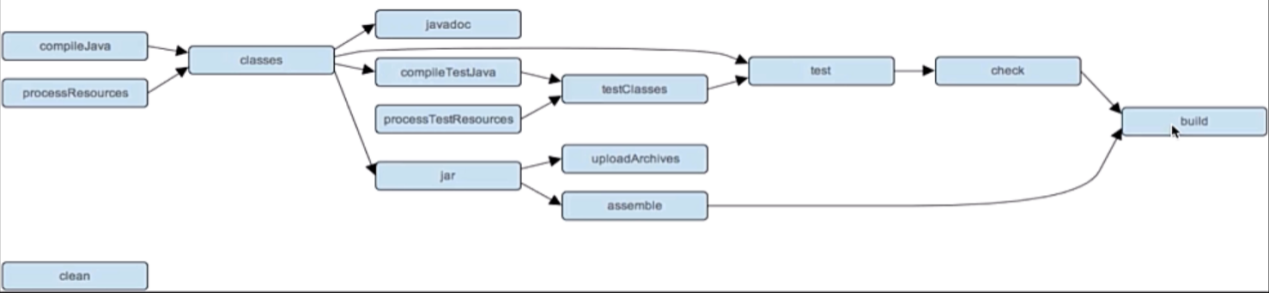
兼容性上：兼容所有的Maven、Ant

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）：



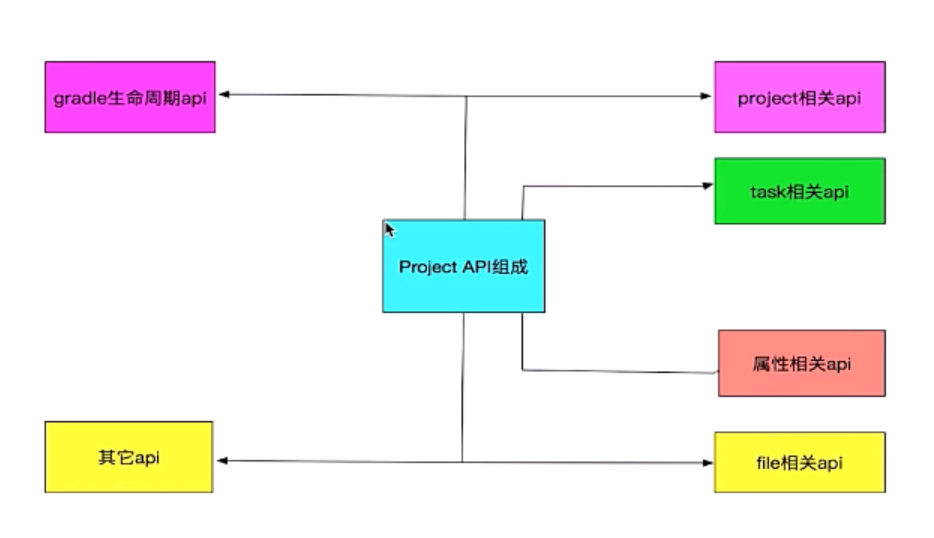
要先把其他的task执行完毕后，才能执行到build。这是在配置阶段执行完毕后执行的，配置阶段执行完后gradle在内存中生成task的依赖关系，最后才执行build

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’

]

}

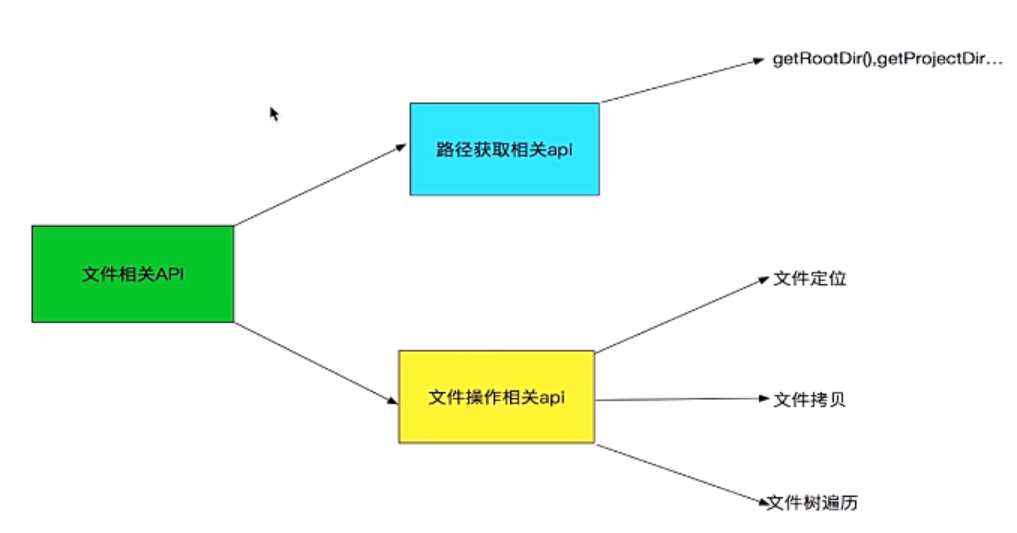
在其它地方用时：rootProject.ext.android. compileSdkVersion

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

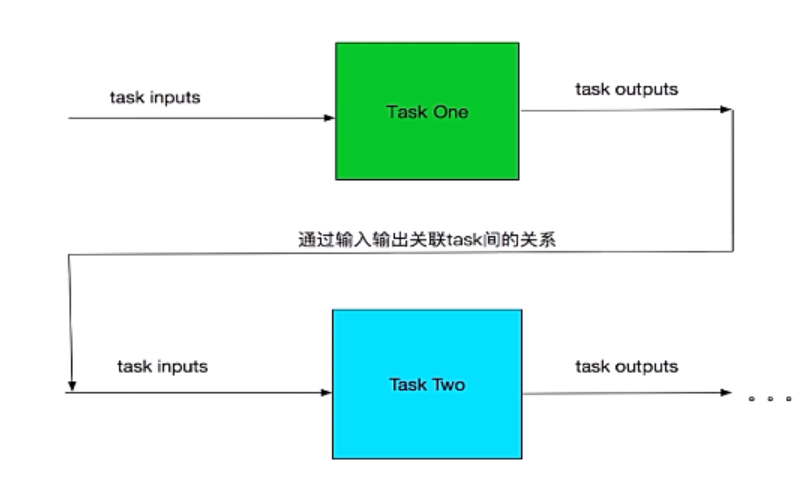
TASK\_NAME=”name ”，task的名称

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**

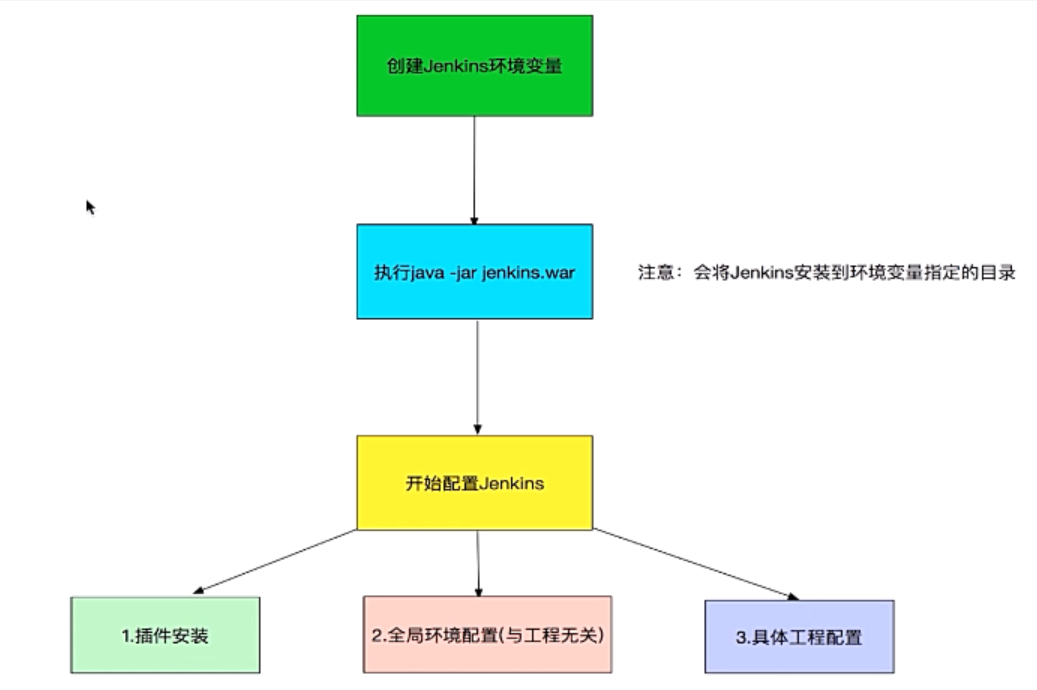
buildDir对应

**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

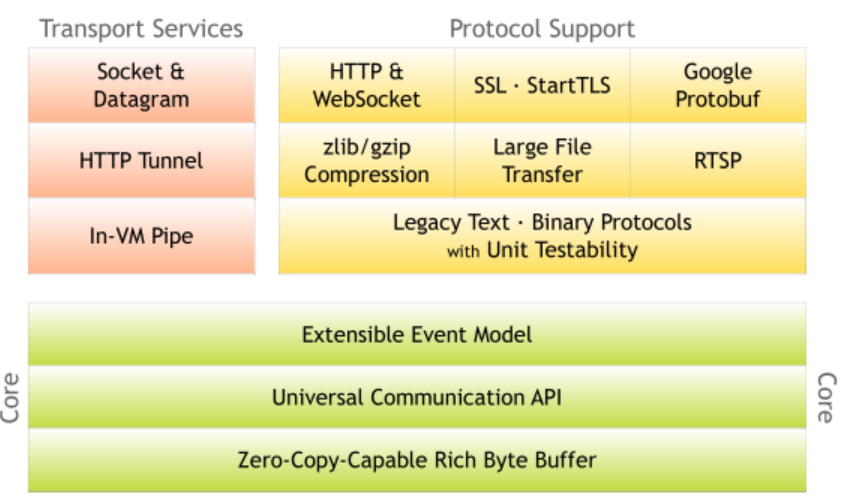
*netty是一个异步的事件驱动（一个一个阶段性的）的网络框架，它是基于一个高性能的，协议化的，服务端的和客户端的快速开发。*

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。

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。

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。

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：

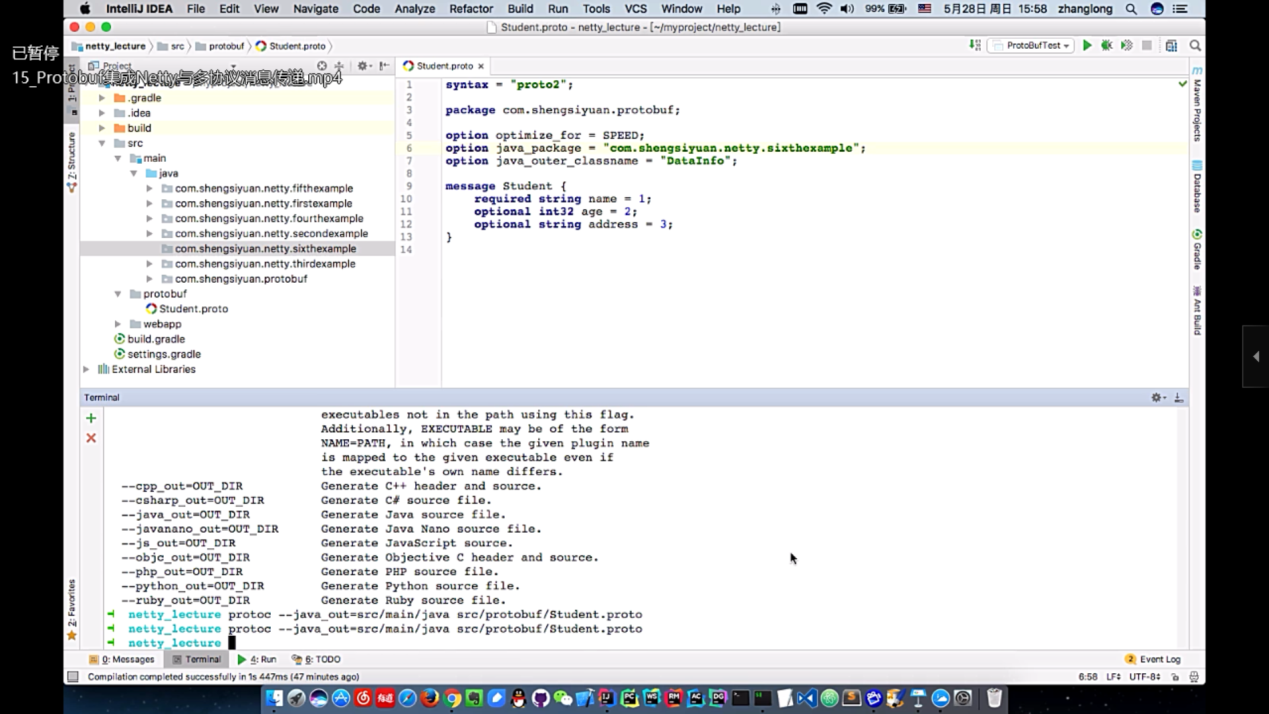
1. 编解码的效率，webService编解码的速度较rpc低
2. 在数据传递上rpc都是通过webSocket进行传输的，它的效率更高，webService是通过http进行传输对的。

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：

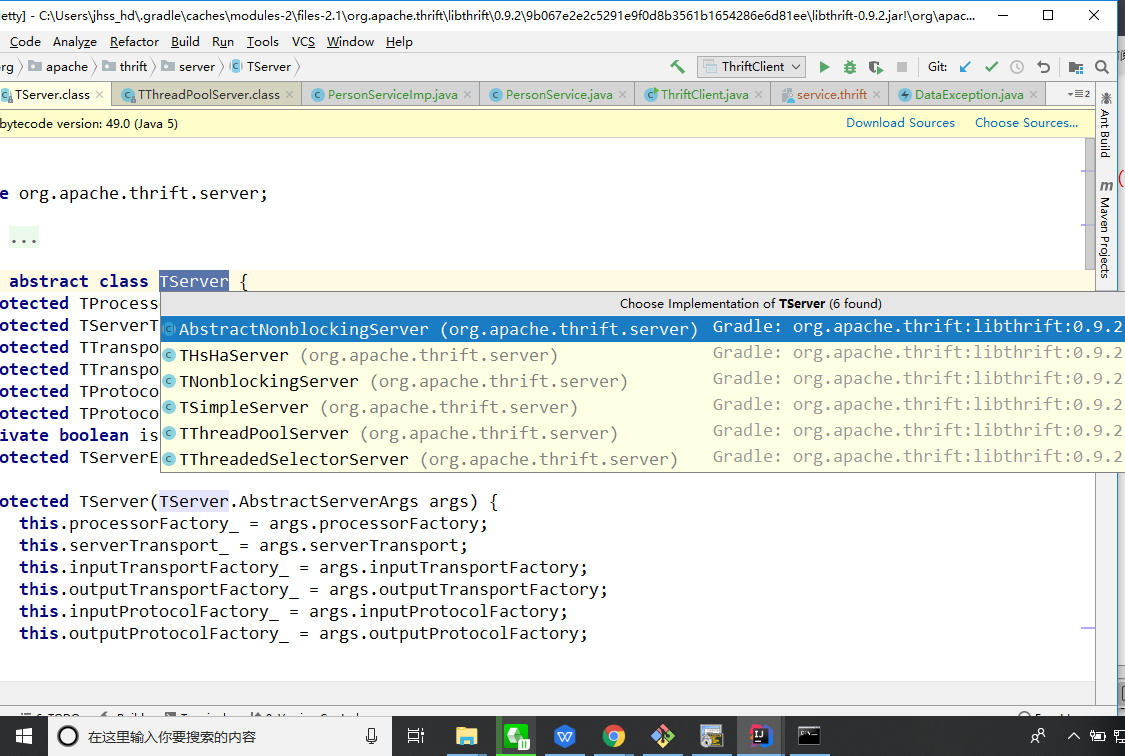
数据传输使用socket，数据再以特定的格式（string等）发送，接受方语言进行解析。

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。

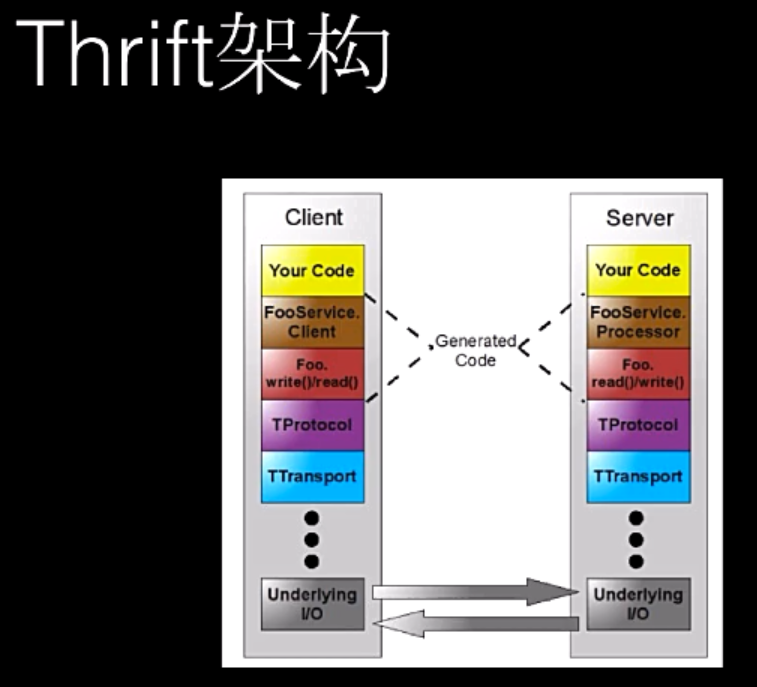


TServer的实现类有六种，其中

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

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selectd-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。

Cannel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方法目的是将数据从磁盘取出来放到内存中，Kemel space（内核空间缓冲区），User space（用户空间缓冲区）

write方法目的是将内存中的数据写入到网络中，下面的缓冲区时socket的缓冲区。上面是同步的传统IO操作。以上有四次上下文的切换，以及两次不必要的拷贝。切换：由用户空间到内核空间，再由内核空间到用户空间，再由用户空间到内核空间，内核空间到用户空间。

拷贝：由内核空间将数据拷贝到用户空间，再由用户空间拷贝到内核空间。

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。

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。

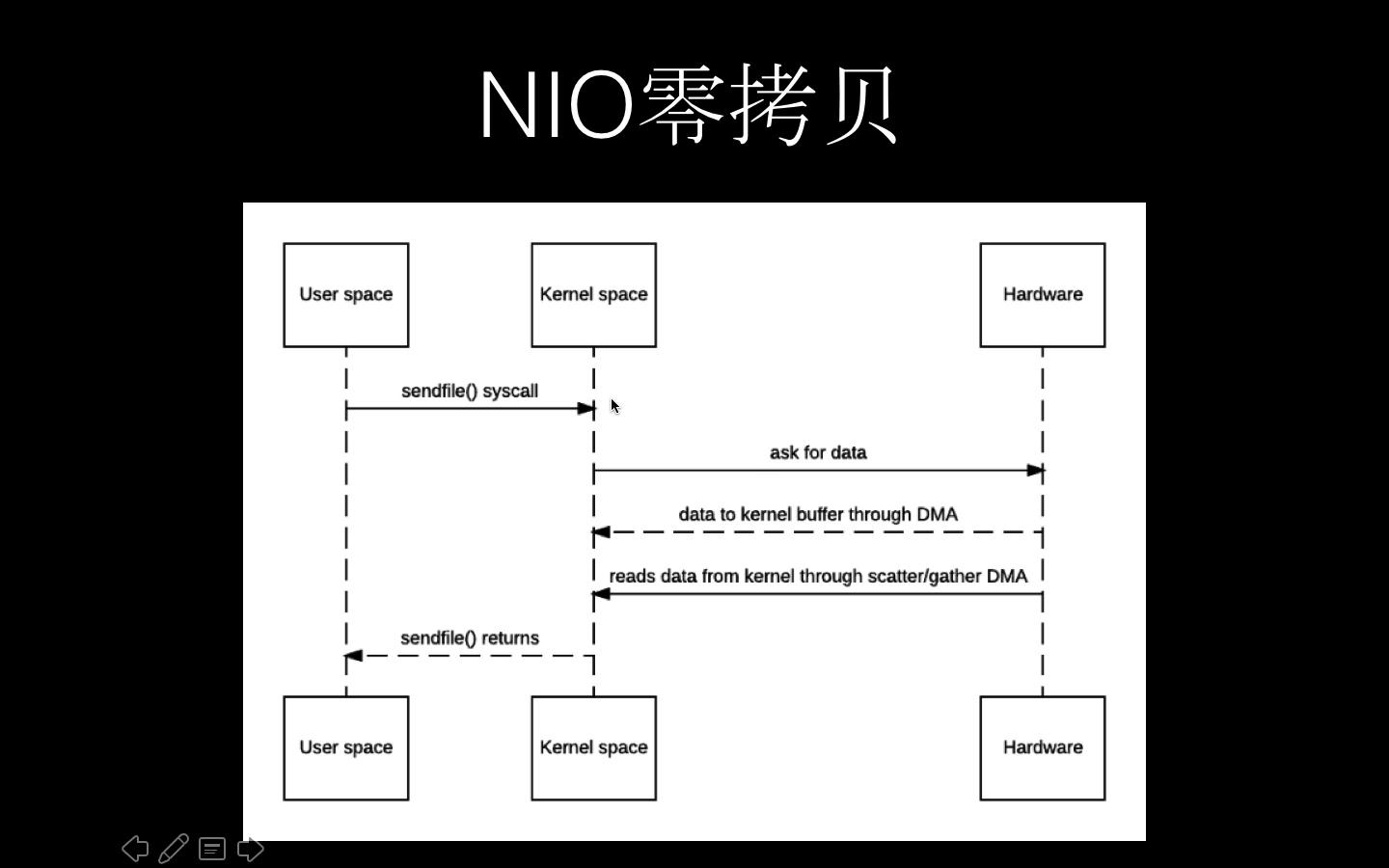
零拷贝



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，然后硬件中的数据就读到了内核空间的缓冲区中。

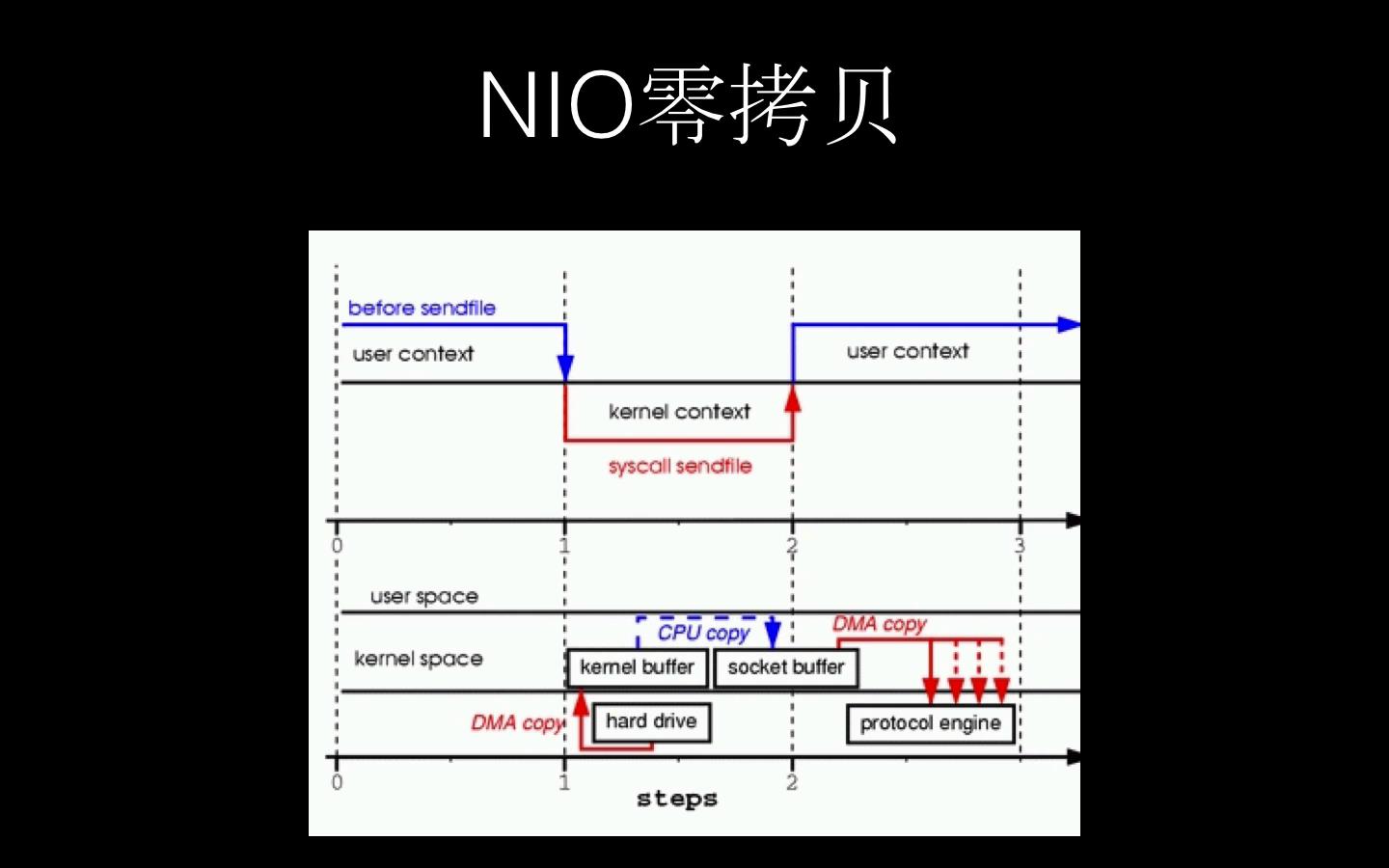
改进后的零拷贝



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。

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。

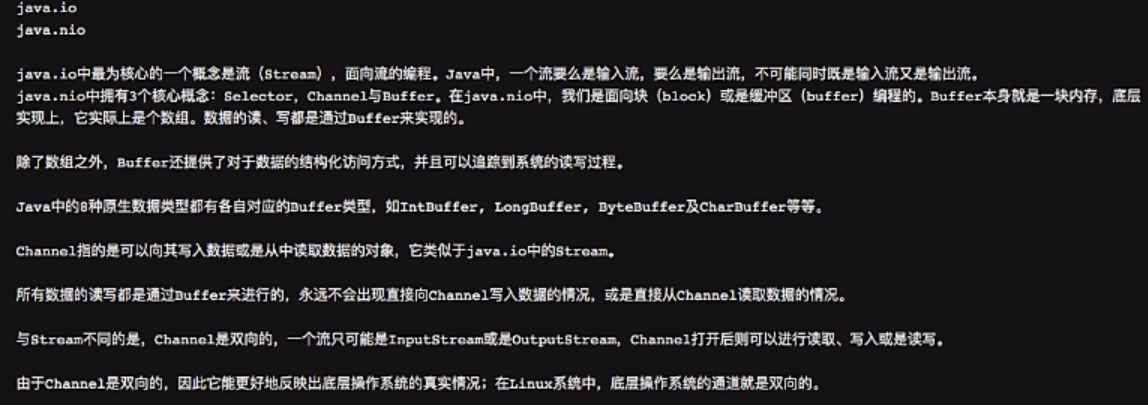


在kemel buffer --》socket buffer的过程：

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。

NIO:



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。

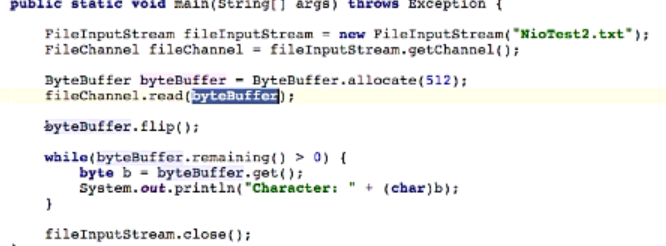
数据来自于Channel，把数据从Channel中读到buffer中，再读取buffer。

通过

bufer.put()

buffer.filp();//状态的反转

buffer.get()；

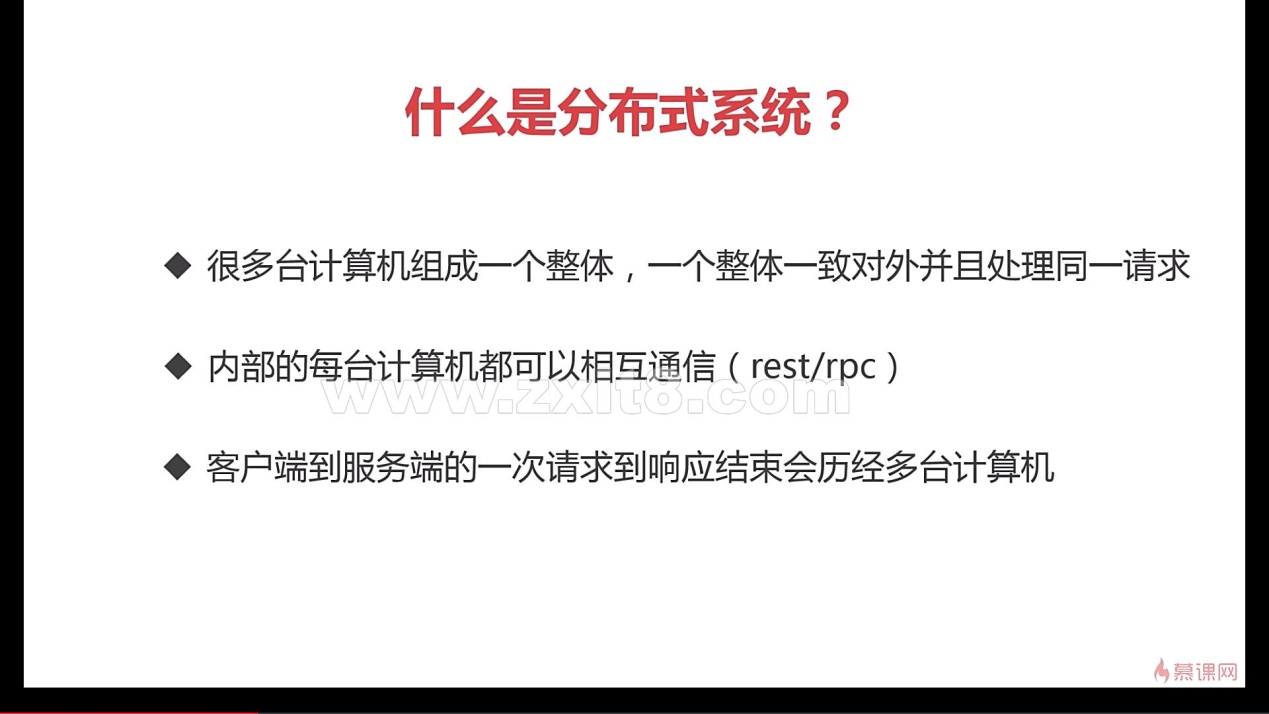


对于服务器这种，关于心跳，heartBeat，服务器端比如zookeeper形成的集群，它们之间都会通过tcp方式进行节点与节点之间的通讯。主节点负责读写，从节点只负责读。因为主机器发生异常时，通过异步的方式把，节点与节点之间是最终一致性，而不是实时一致性。经过了几毫秒后，最终一致。节点有可能挂掉，所以需要每过几秒钟A向B发送一个心跳包，可以设置连续发送几次。这是长连接中最重要的一个保证网络是否中断的措施。服务端掉用closed方法，然后继续尝试连接。

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。

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’

1. 设置用户邮箱

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

1. 查看设置

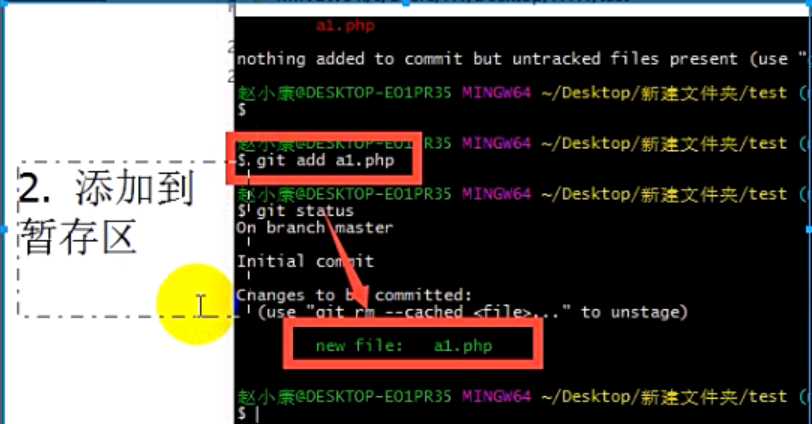
git config -list

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）

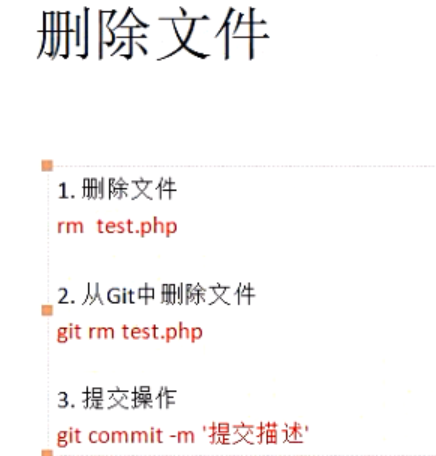
git init，后生成了一个.git文件

在git add 文件名后，



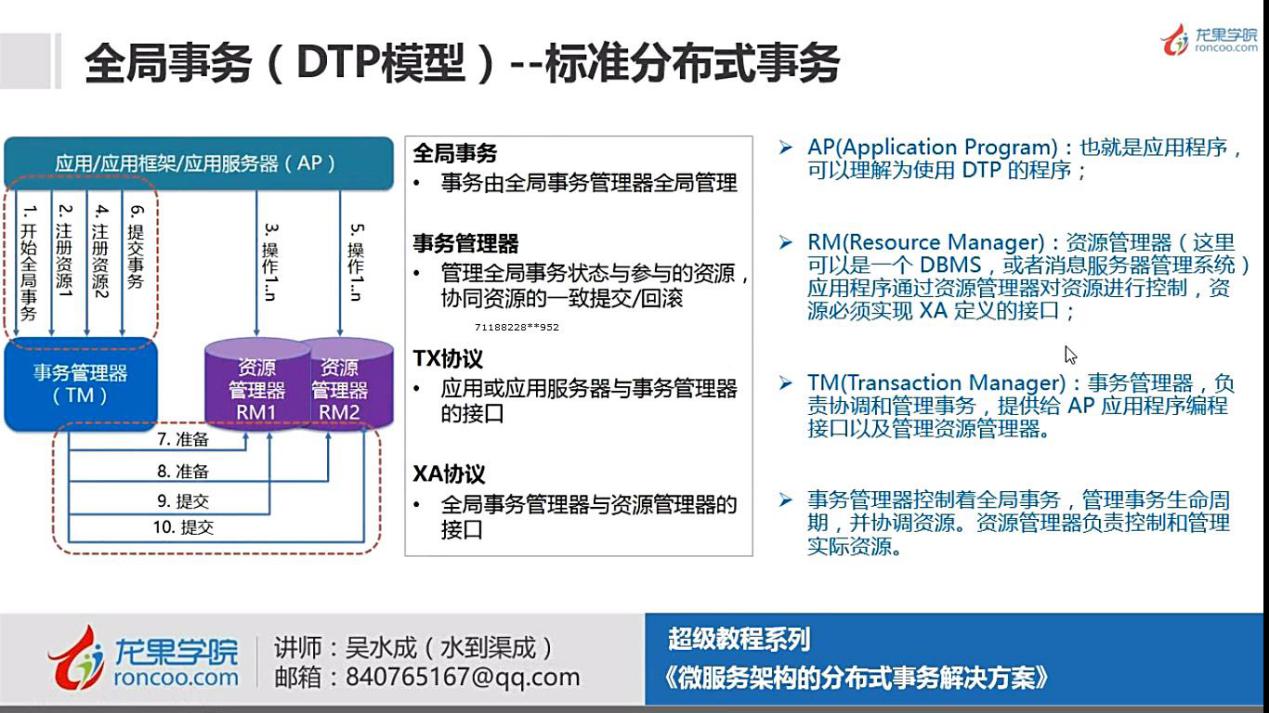
提交到Git仓库：

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：

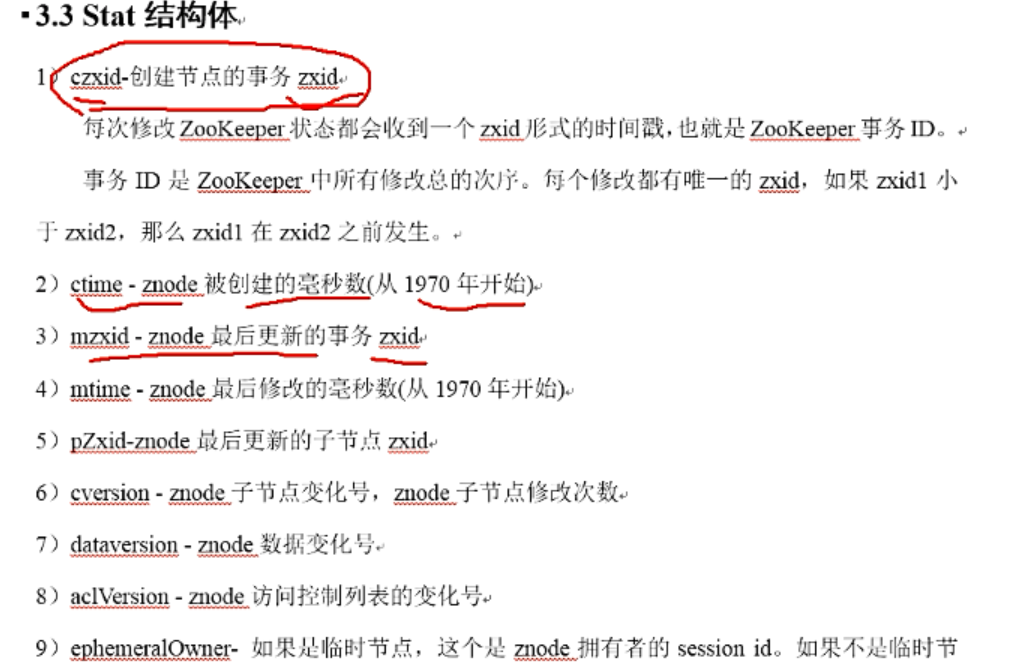
get /path watch

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

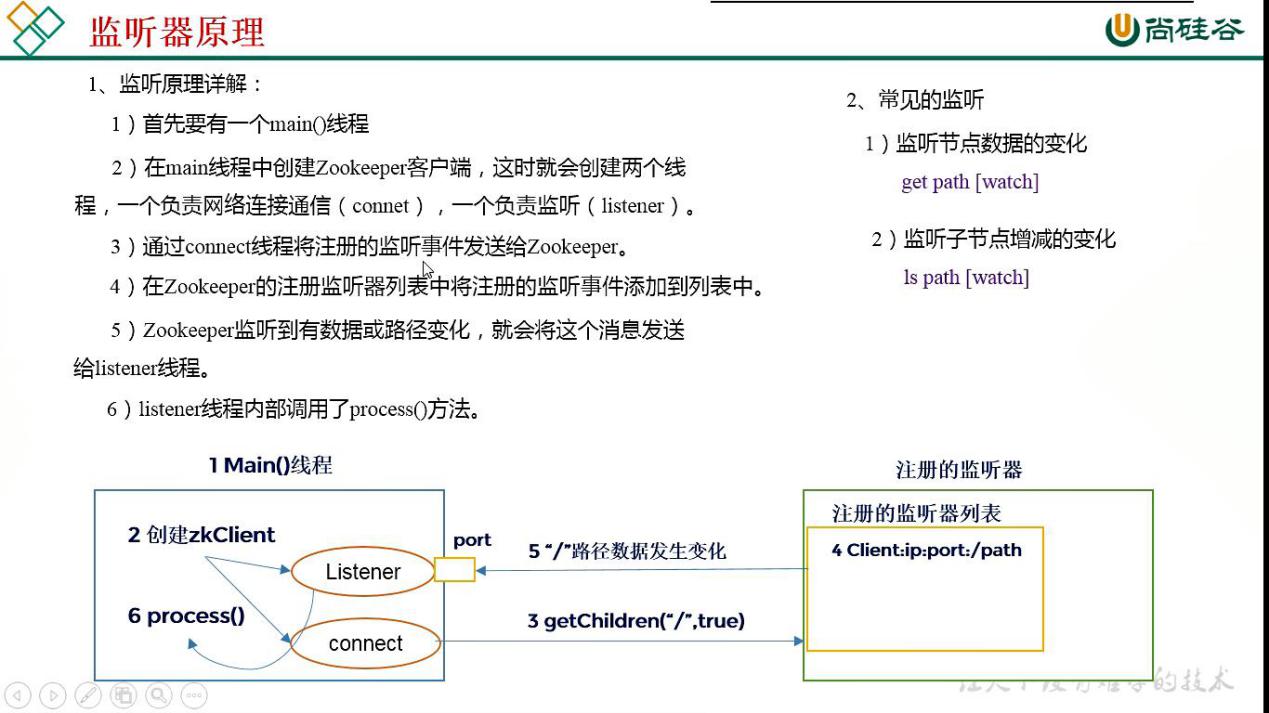
递归删除：rmr /san/suguo

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：

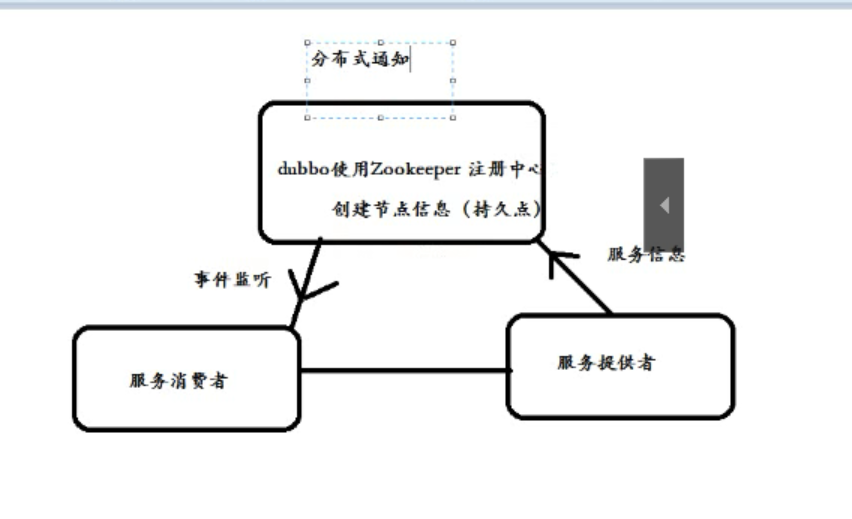
1. 某分布式系统中，主节点可以有多台，可以动态上下线，任意一台客户端都能实时感知到主节点服务器的上下线。

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

### 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。

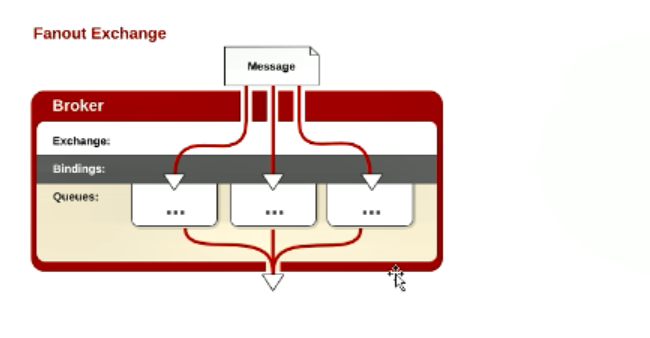
KafKa是一个分布式发布订阅消息系统。

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。

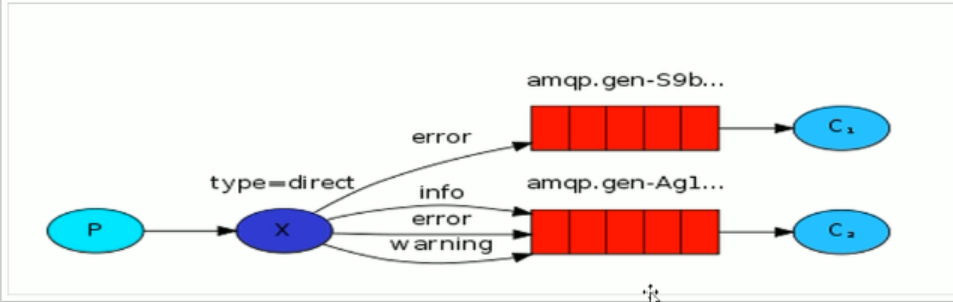
在rabbitMQ中，信息会从你的应用程序出来，来到Rabbit的队列，所有信息可以只存储在一个队列中，队列可以存储很多信息，因为它基本上是一个无限制的缓冲区

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。

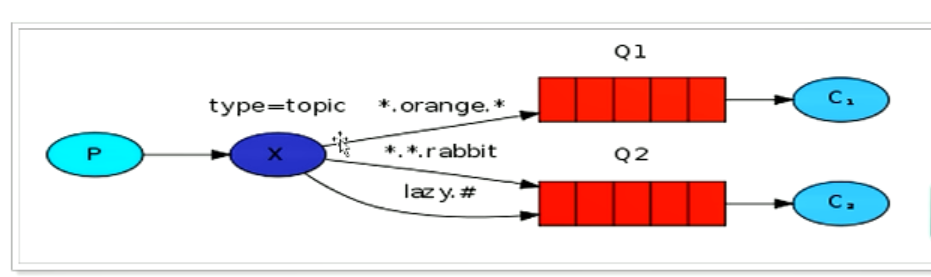
fanout模式：



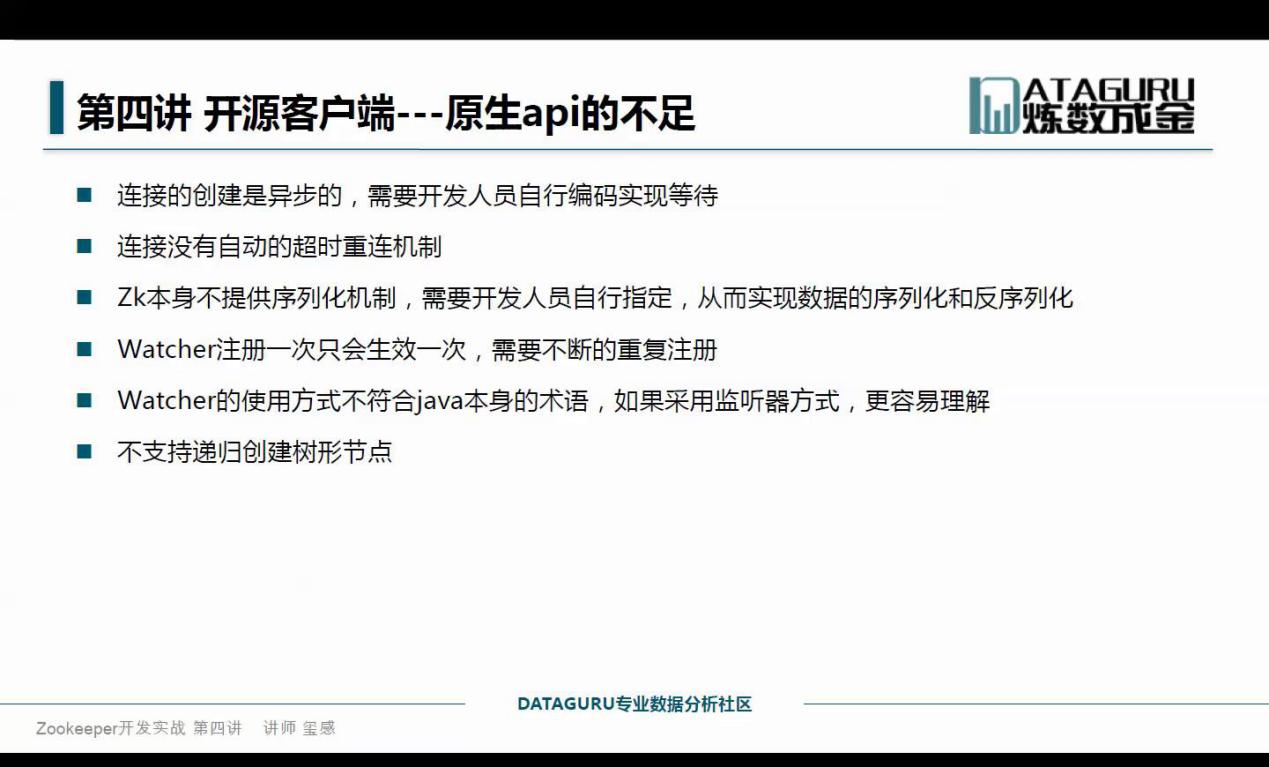
路由消息模式：



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

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。

1. 安装完成后，还需要最后一步设置，在命令行输入：
2. $ git config --global user.name "Your Name"
3. $ 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

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](mailto: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：

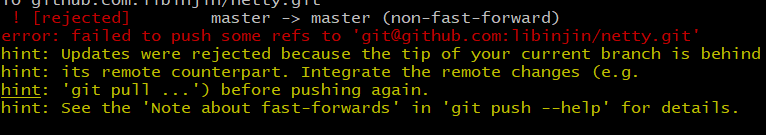
commit：

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

sdf

3.当文件更改后，执行git diff查看文件修改的地方

1. 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
2. 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。
3. 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：

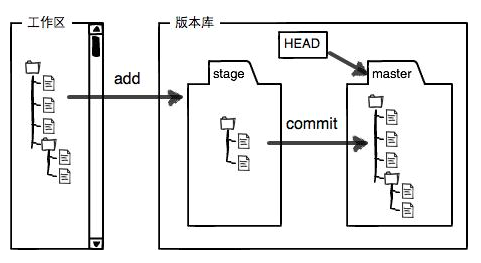
工作区：它就是你在电脑里面能看到的目录，比如我的antherGit文件夹就是一个工作区。

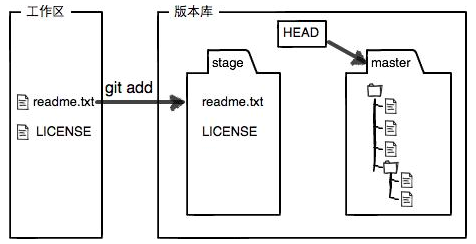
版本库（Repository）

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。

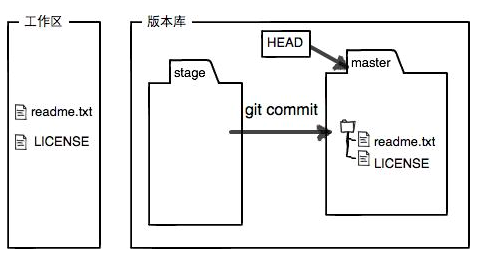
提交完后，工作区就是干净的。

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。

1. 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](mailto: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](mailto:git@github.com:libinjin/myReposity.git)

切换到该库上：

cd myReposity/

1. 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

1. 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. 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啊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。

hotfix\_v2.6.16

groovy讲解之groovy初探

为什么选择groovy及其优势