2、答：tomcat**生命周期管理**  
Tomcat 为了方便管理组件和容器的生命周期，定义了从创建、启动、到停止、销毁共 12 中状态，tomcat 生命周期管理了内部状态变化的规则控制，组件和容器只需实现相应的生命周期 方法即可完成各生命周期内的操作(initInternal、startInternal、stopInternal、 destroyInternal)；

比如执行初始化操作时，会判断当前状态是否 New，如果不是则抛出生命周期异常；是的 话则设置当前状态为 Initializing，并执行 initInternal 方法，由子类实现，方法执行成功则设置当 前状态为 Initialized，执行失败则设置为 Failed 状态；

Tomcat 的生命周期管理引入了事件机制，在组件或容器的生命周期状态发生变化时会通 知事件监听器，监听器通过判断事件的类型来进行相应的操作。  
事件监听器的添加可以在 server.xml 文件中进行配置;

Tomcat 各类容器的配置过程就是通过添加 listener 的方式来进行的，从而达到配置逻辑与 容器的解耦。如 EngineConfig、HostConfig、ContextConfig。  
EngineConfig：主要打印启动和停止日志  
HostConfig：主要处理部署应用，解析应用 META-INF/context.xml 并创建应用的 Context ContextConfig：主要解析并合并 web.xml，扫描应用的各类 web 资源 (filter、servlet、listener)

Servlet 是用 Java 编写的服务器端程序。其主要功能在于交互式地浏览和修改数据，生成动态 Web 内容。

1. 请求到达 server 端，server 根据 url 映射到相应的 Servlet
2. 判断 Servlet 实例是否存在，不存在则加载和实例化 Servlet 并调用 init 方法
3. Server 分别创建 Request 和 Response 对象，调用 Servlet 实例的 service 方法(service 方法 内部会根据 http 请求方法类型调用相应的 doXXX 方法)
4. doXXX 方法内为业务逻辑实现，从 Request 对象获取请求参数，处理完毕之后将结果通过 response 对象返回给调用方
5. 当 Server 不再需要 Servlet 时(一般当 Server 关闭时)，Server 调用 Servlet 的 destroy() 方 法。

load on startup

当值为 0 或者大于 0 时，表示容器在应用启动时就加载这个 servlet; 当是一个负数时或者没有指定时，则指示容器在该 servlet 被选择时才加载; 正数的值越小，启动该 servlet 的优先级越高;

single thread model

每次访问 servlet，新建 servlet 实体对象，但并不能保证线程安全，同时 tomcat 会限制 servlet 的实例数目  
最佳实践：不要使用该模型，servlet 中不要有全局变量

**请求处理过程:**

1.根据 server.xml 配置的指定的 connector 以及端口监听 http、或者 ajp 请求

2.请求到来时建立连接,解析请求参数,创建 Request 和 Response 对象,调用顶层容器 pipeline 的 invoke 方法

3.容器之间层层调用,最终调用业务 servlet 的 service 方法

4．Connector 将 response 流中的数据写到 socket 中

JSP引擎：

JSP 生命周期：

1.编译阶段:servlet 容器编译 servlet 源文  
件,生成 servlet 类

2.初始化阶段:加载与 JSP 对应的 servlet 类, 创建其实例,并调用它的初始化方法

3.执行阶段:调用与 JSP 对应的 servlet 实例的 服务方法

4.销毁阶段:调用与 JSP 对应的 servlet 实例的 销毁方法,然后销毁 servlet 实例

Jsp 解析过程：

1、代码片段:在\_jspService()方法内直接输出

2、JSP 声明: 在 servlet 类中进行输出

3、JSP 表达式:在\_jspService()方法内直接输出

4、JSP 注释:直接忽略,不输出

5、 JSP 指令:根据不同指令进行区分,include:对引入的文件进行解析;page 相关的属性会做为 JSP 的属性,影响的是解析和请求处理时的行为

6、JSP 行为:不同的行为有不同的处理方式,jsp:useBean 为例,会从 pageContext 根据 scope 的 类别获取 bean 对象,如果没有会创建 bean,同时存到相应 scope 的 pageContext 中

7、 HTML:在\_jspService()方法内直接输出

8、JSP 隐式对象:在\_jspService()方法会进行声明,只能在方法中使用;

**Connector**

Connector的实现模式有三种，分别是BIO、NIO、APR，可以在server.xml中指定。

1.JIO：用java.io编写的TCP模块，阻塞IO

2.NIO：用java.nio编写的TCP模块，非阻塞IO，（IO多路复用）

3.APR：全称Apache Portable Runtime，使用JNI的方式来进行读取文件以及进行网络传输

**Comet**

Comet是一种用于web的推送技术，能使服务器实时地将更新的信息传送到客户端，而无须客户端发出请求  
在WebSocket出来之前，如果不适用comet，只能通过浏览器端轮询Server来模拟实现服务器端推送。  
Comet支持servlet异步处理IO，当连接上数据可读时触发事件，并异步写数据(阻塞)。

**异步Servlet**

传统流程：

首先，Servlet 接收到请求之后，request数据解析；

接着，调用业务接口的某些方法，以完成业务处理；

最后，根据处理的结果提交响应，Servlet 线程结束

异步处理流程：

客户端发送一个请求

Servlet容器分配一个线程来处理容器中的一个servlet

servlet调用request.startAsync()，保存AsyncContext, 然后返回

任何方式存在的容器线程都将退出，但是response仍然保持开放

业务线程使用保存的AsyncContext来完成响应（线程池）

客户端收到响应

Servlet 线程将请求转交给一个异步线程来执行业务处理，线程本身返回至容器，此时 Servlet 还没有生成响应数据，异步线程处理完业务以后，可以直接生成响应数据（异步线程拥有 ServletRequest 和 ServletResponse 对象的引用）

**为什么web应用中支持异步？**

推出异步，主要是针对那些比较耗时的请求：比如一次缓慢的数据库查询，一次外部REST API调用, 或者是其他一些I/O密集型操作。这种耗时的请求会很快的耗光Servlet容器的线程池，继而影响可扩展性。

Note：从客户端的角度来看，request仍然像任何其他的HTTP的request-response交互一样，只是耗费了更长的时间而已

4、答：

Server端：三个核心类。分别是

NettyServerListener：服务启动监听器

ServerChannelHandlerAdapter：通道适配器，主要用于多线程共享

RequestDispatcher：请求分排器

**在pom.xml中添加以下依赖**：

<dependency>

<groupId>io.netty</groupId>

<artifactId>netty-all</artifactId>

<version>5.0.0.Alpha2</version>

</dependency>

<dependency>

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

<artifactId>spring-boot-configuration-processor</artifactId>

<optional>true</optional>

</dependency>

**让SpringBoot的启动类实现CommandLineRunner接口并重写run方法,比如我的启动类是CloudApplication.java**

@SpringBootApplication

public class CloudApplication implements CommandLineRunner {

public static void main(String[] args) {

SpringApplication.run(CloudApplication.class, args);

}

public void run(String... strings) {

}

}

**创建类NettyServerListener.java**

// 读取yml的一个配置类

import com.edu.hart.modules.constant.NettyConfig;

// Netty连接信息配置类

import com.edu.hart.modules.constant.NettyConstant;

//

import com.edu.hart.rpc.util.ObjectCodec;

import io.netty.bootstrap.ServerBootstrap;

import io.netty.channel.ChannelFuture;

import io.netty.channel.ChannelInitializer;

import io.netty.channel.ChannelPipeline;

import io.netty.channel.EventLoopGroup;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.SocketChannel;

import io.netty.channel.socket.nio.NioServerSocketChannel;

import io.netty.handler.codec.LengthFieldBasedFrameDecoder;

import io.netty.handler.codec.LengthFieldPrepender;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Component;

import javax.annotation.PreDestroy;

import javax.annotation.Resource;

/\*\*

\* 服务启动监听器

\*

\*/

@Component

public class NettyServerListener {

private static final Logger LOGGER = LoggerFactory.getLogger(NettyServerListener.class);

/\*\*

\* 创建bootstrap

\*/

ServerBootstrap serverBootstrap = new ServerBootstrap();

/\*\*

\* BOSS

\*/

EventLoopGroup boss = new NioEventLoopGroup();

/\*\*

\* Worker

\*/

EventLoopGroup work = new NioEventLoopGroup();

/\*\*

\* 通道适配器

\*/

@Resource

private ServerChannelHandlerAdapter channelHandlerAdapter;

/\*\*

\* NETT服务器配置类

\*/

@Resource

private NettyConfig nettyConfig;

/\*\*

\* 关闭服务器方法

\*/

@PreDestroy

public void close() {

LOGGER.info("关闭服务器....");

//优雅退出

boss.shutdownGracefully();

work.shutdownGracefully();

}

/\*\*

\* 开启及服务线程

\*/

public void start() {

// 从配置文件中(application.yml)获取服务端监听端口号

int port = nettyConfig.getPort();

serverBootstrap.group(boss, work)

.channel(NioServerSocketChannel.class)

.option(ChannelOption.SO\_BACKLOG, 100)

.handler(new LoggingHandler(LogLevel.INFO));

try {

//设置事件处理

serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>() {

@Override

protected void initChannel(SocketChannel ch) throws Exception {

ChannelPipeline pipeline = ch.pipeline();

pipeline.addLast(new LengthFieldBasedFrameDecoder(nettyConfig.getMaxFrameLength()

, 0, 2, 0, 2));

pipeline.addLast(new LengthFieldPrepender(2));

pipeline.addLast(new ObjectCodec());

pipeline.addLast(channelHandlerAdapter);

}

});

LOGGER.info("netty服务器在[{}]端口启动监听", port);

ChannelFuture f = serverBootstrap.bind(port).sync();

f.channel().closeFuture().sync();

} catch (InterruptedException e) {

LOGGER.info("[出现异常] 释放资源");

boss.shutdownGracefully();

work.shutdownGracefully();

}

}

}

**创建类ServerChannelHandlerAdapter.java - 通道适配器**

// 记录调用方法的元信息的类

import com.edu.hart.rpc.entity.MethodInvokeMeta;

import io.netty.channel.ChannelHandler.Sharable;

import io.netty.channel.ChannelHandlerAdapter;

import io.netty.channel.ChannelHandlerContext;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Component;

import javax.annotation.Resource;

/\*\*

\* 多线程共享

\*/

@Component

@Sharable

public class ServerChannelHandlerAdapter extends ChannelHandlerAdapter {

/\*\*

\* 日志处理

\*/

private Logger logger = LoggerFactory.getLogger(ServerChannelHandlerAdapter.class);

/\*\*

\* 注入请求分排器

\*/

@Resource

private RequestDispatcher dispatcher;

@Override

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {

cause.printStackTrace();

ctx.close();

}

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) {

MethodInvokeMeta invokeMeta = (MethodInvokeMeta) msg;

// 屏蔽toString()方法

if (invokeMeta.getMethodName().endsWith("toString()")

&& !"class java.lang.String".equals(invokeMeta.getReturnType().toString()))

logger.info("客户端传入参数 :{},返回值：{}",

invokeMeta.getArgs(), invokeMeta.getReturnType());

dispatcher.dispatcher(ctx, invokeMeta);

}

}

RequestDispatcher.java

// 封装的返回信息枚举类

import com.edu.hart.modules.communicate.ResponseCodeEnum;

// 封装的返回信息实体类

import com.edu.hart.modules.communicate.ResponseResult;

// 封装的连接常量类

import com.edu.hart.modules.constant.NettyConstant;

// 记录元方法信息的实体类

import com.edu.hart.rpc.entity.MethodInvokeMeta;

// 对于返回值为空的一个处理

import com.edu.hart.rpc.entity.NullWritable;

// 封装的返回信息实体工具类

import com.edu.hart.rpc.util.ResponseResultUtil;

import io.netty.channel.ChannelFuture;

import io.netty.channel.ChannelFutureListener;

import io.netty.channel.ChannelHandlerContext;

import org.springframework.beans.BeansException;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ApplicationContextAware;

import org.springframework.stereotype.Component;

import java.lang.reflect.Method;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

/\*\*

\* 请求分排器

\*/

@Component

public class RequestDispatcher implements ApplicationContextAware {

private ExecutorService executorService = Executors.newFixedThreadPool(NettyConstant.getMaxThreads());

private ApplicationContext app;

/\*\*

\* 发送

\*

\* @param ctx

\* @param invokeMeta

\*/

public void dispatcher(final ChannelHandlerContext ctx, final MethodInvokeMeta invokeMeta) {

executorService.submit(() -> {

ChannelFuture f = null;

try {

Class<?> interfaceClass = invokeMeta.getInterfaceClass();

String name = invokeMeta.getMethodName();

Object[] args = invokeMeta.getArgs();

Class<?>[] parameterTypes = invokeMeta.getParameterTypes();

Object targetObject = app.getBean(interfaceClass);

Method method = targetObject.getClass().getMethod(name, parameterTypes);

Object obj = method.invoke(targetObject, args);

if (obj == null) {

f = ctx.writeAndFlush(NullWritable.nullWritable());

} else {

f = ctx.writeAndFlush(obj);

}

f.addListener(ChannelFutureListener.CLOSE);

} catch (Exception e) {

ResponseResult error = ResponseResultUtil.error(ResponseCodeEnum.SERVER\_ERROR);

f = ctx.writeAndFlush(error);

} finally {

f.addListener(ChannelFutureListener.CLOSE);

}

});

}

/\*\*

\* 加载当前application.xml

\*

\* @param ctx

\* @throws BeansException

\*/

public void setApplicationContext(ApplicationContext ctx) throws BeansException {

this.app = ctx;

}

}

application.yml文件中对于netty的一个配置

netty:

port: 2222

NettyConfig.java

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.stereotype.Component;

/\*\*

\* 读取yml配置文件中的信息

\*/

@Component

@ConfigurationProperties(prefix = "netty")

public class NettyConfig {

private int port;

public int getPort() {

return port;

}

public void setPort(int port) {

this.port = port;

}

}

NettyConstanct.java

import org.springframework.stereotype.Component;

/\*\*

\* Netty服务器常量

\*/

@Component

public class NettyConstant {

/\*\*

\* 最大线程量

\*/

private static final int MAX\_THREADS = 1024;

/\*\*

\* 数据包最大长度

\*/

private static final int MAX\_FRAME\_LENGTH = 65535;

public static int getMaxFrameLength() {

return MAX\_FRAME\_LENGTH;

}

public static int getMaxThreads() {

return MAX\_THREADS;

}

}

至此，netty服务端算是与SpringBoot整合完毕

8答：实现java.io.Serializable 接口的类是可序列化的，序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间；== 比较的是变量(栈)内存中存放的对象的(堆)内存地址，用来判断两个对象的地址是否相同，即是否是指相同一个对象；equals用来比较的是两个对象的内容是否相等，由于所有的类都是继承自java.lang.Object类的，所以适用于所有对象，如果没有对该方法进行覆盖的话，调用的仍然是Object类中的方法，而Object中的equals方法返回的却是==的判断。基本数据类型之间的比较需要用双等号（==），因为他们比较的是值

9、答：实体类为未来参与排序等集合操作，一般要重写compare方法，一般注意是按照升序排列还是降序排列

10、答：由于浏览器的同源策略，出于防范跨站脚本的攻击，禁止客户端脚本（js）对不同域的服务进行跨站调用

11、答：接口幂等任意多次执行所产生的影响均与一次执行的影响相同，解决方案：1）唯一索引，防止新增脏数据，2）token机制，防止页面重复提交；3）插入更新；4）多版本控制；5）状态机

12、答：在类上加@Transactional(对整个类起作用), 在方法上加@Transactional(只对整个方法起作用）,不生效原因：

jdk是代理接口，私有方法必然不会存在在接口里，所以就不会被拦截到；

cglib是子类，private的方法照样不会出现在子类里，也不能被拦截