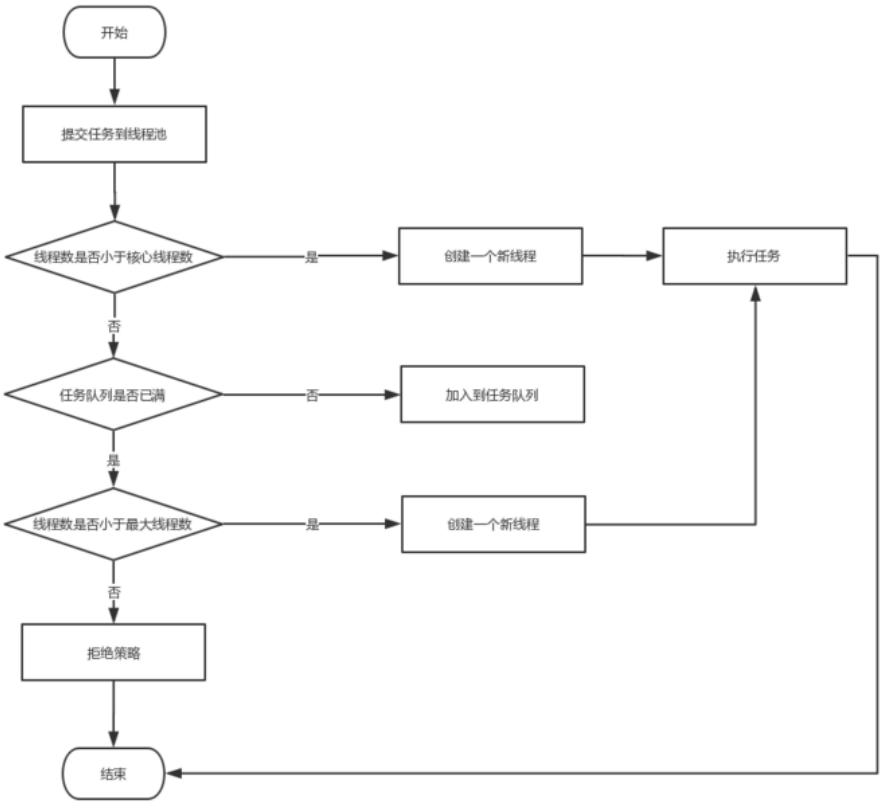
1、线程池的基本原理、参数、拒绝策略



（1）原理：

1> 将任务提交给线程池；

2> 如果线程池中的线程数小于核心线程数，则创建一个新的线程来执行任务，否则进入步骤3；

3> 提交任务时，线程池中的空闲的线程数为0，并且线程数等于核心线程数，则观察线程池中的任务队列是否已满，如果未满，则将任务添加到任务队列，否则进入步骤4；

4> 如果最大线程数大于核心线程数，并且总线程数小于最大线程数，则创建一个新的线程来执行该任务，否则进入步骤5；

5> 当任务队列已满时，就执行拒绝策略。

（2）参数说明：

参数	说明
corePoolSize	核心线程数，核心线程会一直存活，即使没有任务需要执行，当线程数小于核心线程数时，即使有线程空闲，线程池也会优先创建新线程处理。设置allowCoreThreadTimeout=true时，核心线程会超时关闭
queueCapacity	任务队列容量（阻塞队列）当核心线程数达到最大时，新任务会放在队列中排队等待执行。
maxPoolSize	最大线程数。当线程数>=corePoolSize，且任务队列已满时，线程池会创建新线程来处理任务，当线程数=maxPoolSize，且任务队列已满时，线程池会拒绝处理任务而抛出异常。
keepAliveTime	线程空闲时间，当线程空闲时间达到keepAliveTime时，线程会退出，直到线程数量=corePoolSize，如果allowCoreThreadTimeout=true，则会直到线程数量=0
allowCoreThreadTimeout	允许核心线程超时关闭
rejectedExecutionHandler	任务拒绝处理器

两种情况会拒绝处理任务：

1> 当线程数已经达到maxPoolSize，且队列已满，会拒绝新任务；

2> 当线程池被调用shundown()后，会等待线程池里的任务执行完毕，再shutdown。如果在调用shutdown()和线程池真正shundown之间提交任务，会拒绝新任务。

线程池会调用rejectedExecutionHandler来处理这个任务，如果没有设置，默认时AbortPolicy，会抛出异常。

ThreadPoolExecutor类有几个内部实现类来处理这种情况：

拒绝策略	说明
AbortPolicy	直接丢弃任务，并抛出RejectedExecutionException异常
DiscardPolicy	直接丢弃任务，什么都不做，不抛异常
DiscardOldestPolicy	抛弃最老的任务，从阻塞队列中移除最早提交的任务，然后将该任务加入
CallerRunsPolicy	由提交任务的线程执行任务

（3）Executors提供4种线程池

1. newFixedThreadPool 创建一个固定长度线程池，可控制线程最大并发数，避免堵塞；
2. newCachedThreadPool 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程，线程池的规模不存在限制；
3. newScheduledThreadPool 创建一个固定长度线程池，支持定时及周期性任务执行；
4. newSingleThreadPool 创建一个单线程的线程池，它只会用唯一的工作线程来执行任务，保证所有线程按照指定顺序执行。

（4）线程池的使用（一）

1. 使用工厂类Executors中的静态方法创建线程池对象，指定线程个数

```
static ExecutorService  newFixedThreadPool(int 线程个数)  返回线程池对象
ExecutorService接口的实现类
```

2. 接口实现类对象调用方法submit(Runnable r)提交线程执行任务

```
ExecutorService es=Executors.newFixedThreadPool(2);
es.submit(new ThreadPoolRunnable()); ThreadPoolRunnable是Runnable的接口实现类
```

```
ExecutorService es=Executors.newFixedThreadPool(2);
Future<String> f=es.submit(new ThreadPoolCallable()); ThreadPoolCallable是
Callable的接口实现类
```

（5）线程池的使用（二）

1. 在application.yml中配置

```
task:
  queue:
    corePoolSize: 10
    maxPoolSize: 30
    queueCapacity: 8
    keepAlive: 60
```

2. 写配置文件读取配置信息，创建线程池

```
@Configuration
public class TaskExecutorConfig {
    @Value("${task.queue.corePoolSize}")
    private int corePoolSize;//线程池维护线程的最少数量
    @Value("${task.queue.maxPoolSize}")
    private int maxPoolSize;//线程池维护线程的最大数量
    @Value("${task.queue.queueCapacity}")
    private int queueCapacity; //缓存队列
    @Value("${task.queue.keepAlive}")
    private int keepAlive;//允许的空闲时间

    @Bean
    public Executor taskQueueExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        // 设置核心线程数
        executor.setCorePoolSize(corePoolSize);
        // 设置最大线程数
        executor.setMaxPoolSize(maxPoolSize);
        // 设置队列容量
        executor.setQueueCapacity(queueCapacity);
        // 设置默认线程名称
        executor.setThreadNamePrefix("taskQueueExecutor-");
        //对拒绝task的处理策略
        executor.setRejectedExecutionHandler(new
ThreadPoolExecutor.CallerRunsPolicy());
        // 设置线程活跃时间（秒）
        executor.setKeepAliveSeconds(keepAlive);
        executor.initialize();
        // 等待所有任务结束后再关闭线程池
        executor.setWaitForTasksToCompleteOnShutdown(true);
```

```
        return executor;
    }
}
```

3. 在Controller注入配置文件对象，获取线程池对象并提交线程

```
@Controller
public class ThreadController {
    @Autowired
    TaskExecutorConfig taskExecutorConfig;
    @RequestMapping("/threadTest")
    @ResponseBody
    public void threadTest(){
        Executor taskExecutor= taskExecutorConfig.taskQueueExecutor();
        taskExecutor.execute(new Runnable() {
            @Override
            public void run() {
                System.out.println("--------当前线程-------
"+Thread.currentThread().getName());
            }
        });
    }
}
```

（6）线程池的使用（三）--Spring xml中配置线程池

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="threadPoolTaskExecutor"
class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
    <!-- 核心线程数  -->
    <property name="corePoolSize" value="2" />
    <!-- 最大线程数 -->
    <property name="maxPoolSize" value="10" />
    <!-- 队列最大长度 >=mainExecutor.maxSize -->
    <property name="queueCapacity" value="1000" />
    <!-- 线程池维护线程所允许的空闲时间 -->
    <property name="keepAliveSeconds" value="60" />
    <!-- 线程池对拒绝任务(无线程可用)的处理策略 -->
    <property name="rejectedExecutionHandler">
        <bean class="java.util.concurrent.ThreadPoolExecutor$CallerRunsPolicy"
/>
    </property>
</bean>

</beans>
```

（7）Executors.newFixedThreadPool(2)和 new ThreadPoolTaskExecutor()创建线程池的区别

newFixedThreadPool的底层是通过ThreadPoolExecutor来实现的，ThreadPoolTaskExecutor是spring core包中的，而ThreadPoolExecutor是JDK中的JUC。ThreadPoolTaskExecutor是对ThreadPoolExecutor进行了封装处理。

（8）若希望在第一批的线程完成后再执行第二批的线程，该怎么实现。

```
public void batchThreadTest() throws InterruptedException {
        //线程池10个线程
        ExecutorService executorService = Executors.newFixedThreadPool(10);
//用new ThreadPoolTaskExecutor()的方法设置线程池不能保证10个任务完成后，运行下面10个任务，？？？
        //第一批十个任务，需要收集任务吗？？？如何收集？？？
        List<StartAgent> agentsStart = new ArrayList<>();
        for(int i=0;i<10;i++){
            agentsStart.add(new StartAgent());
        }
        //第二批十个任务
        List<StartAgent> agentsStart2 = new ArrayList<>();
        for(int i=0;i<10;i++){
            agentsStart.add(new StartAgent());
        }
        List<List<StartAgent>> task = new ArrayList<>();
        task.add(agentsStart);
        task.add(agentsStart2);
        //记录任务执行时间
        long t1 = System.currentTimeMillis();
        CountDownLatch c ;
        //循环任务组
        for(List<StartAgent> startList : task){
            //定义线程阻塞为10
            c = new CountDownLatch(10);
            for(StartAgent agent : startList){
                agent.setCountDownLatch(c);
                executorService.submit(agent);
            }
            //阻塞，等待十个任务都执行后，才继续下一批10任务
            //c.await();
        }
        executorService.shutdown();
    }
}
```

```
public class StartAgent implements Runnable{
    private CountDownLatch countDownLatch;

    @Override
    public void run() {
        try {
            System.out.println("开始启动节点：" +
Thread.currentThread().getName());
            //模拟每个任务执行3秒钟
            Thread.sleep(3000);
            System.out.println(Thread.currentThread().getName() + "执行完毕");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            //注意一定要在finally调用countDown，否则产生异常导致没调用到countDown造成程序死锁
            countDownLatch.countDown();
        }
```

```
            try {
                countDownLatch.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        public void setCountDownLatch(CountDownLatch countDownLatch) {
            this.countDownLatch = countDownLatch;
        }
    }
```

`countDownLatch`：使一个线程等待其他线程各自执行完毕后再执行。是通过一个计数器来实现的，计数器的初始值是线程的数量，每当一个线程执行完毕后，计数器的值就-1，当计数器的值为0时，表示所有线程都执行完毕，然后在闭锁上等待的线程就可以恢复工作了。

（9）合理设置核心线程数量

1. 当线程池的核心线程数量过大或过小的影响：当线程池核心线程数量过大时，线程与线程之间会争取CPU资源，这样就会导致上下文切换，过多的上下文切换会增加线程的执行时间，影响整体执行效率；当线程池中的核心线程数量过少时，如果同一时间有大量任务需要处理，可能会导致大量任务在任务队列中排队等待执行，甚至会出现队列满了之后任务无法执行的情况，或者大量任务堆积在任务队列导致内存溢出。
2. 任务性质

   CPU密集型（计算密集型）：系统的IO读写效率高于CPU效率，大部分情况是CPU有许多运算需要处理，使用率很高，但IO执行很快。

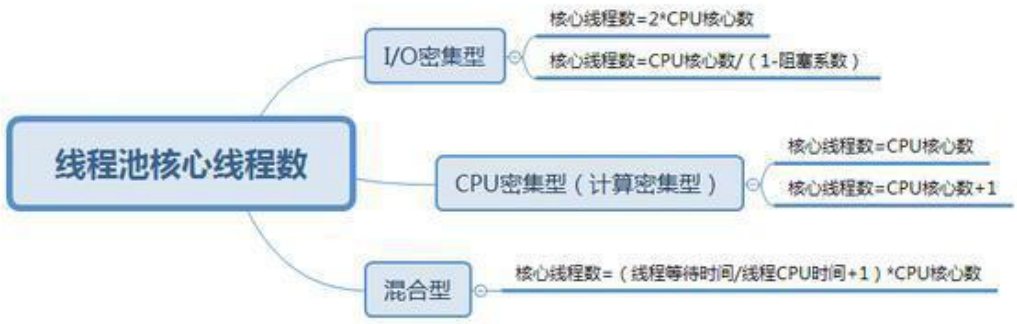
   I/O密集型：系统的CPU性能比磁盘读写效率要高很多，大多数情况是CPU在等I/O的读写操作，此时CPU的使用率并不高。

   混合型任务：既包含CPU密集型又包含I/O密集型。
3. 合理设置核心线程数

   CPU密集型：由于CPU密集型任务的性质，导致CPU的使用率很高，如果线程池中的核心线程数量过多，会增加上下文切换的次数，带来额外的开销，因此，一般情况下，线程池的核心线程数量等于CPU核心数+1。（注：这里核心线程数不是等于CPU核心数，是因为考虑CPU密集型任务由于某些原因而暂停，此时有额外的线程能确保CPU这个时刻不会浪费，但同时也会增加一个CPU上下文切换，因此核心线程数是等于CPU核心数？还是CPU核心数+1？可以根据实际情况来确定）

   I/O密集型任务：由于I/O密集型任务使用率并不是很高，可以让CPU在等待I/O操作的时候去处理别的任务，充分利用CPU。因此，一般情况下，线程池的核心线程数等于2*CPU核心数。（注：有些公司会考虑所需要的CPU阻塞系数，即核心线程数=CPU核心数/(1-阻塞系数)）。

   混合型任务：由于包含2种类型的任务，故混合型任务的线程数与线程时间有关。一般情况下，线程池的核心线程数=(线程等待时间/线程CPU时间+1)*CPU核心数。在某种特定的情况下，还可以将任务分为I/O密集型任务和CPU密集型任务，分别让不同的线程池去处理，但有一个前提--分开后2种任务的执行时间相差不太大。



2、连接池

（1）基本原理：在内部对象池中，维护一定的数据库连接，并对外暴露数据库连接获取和返回的方法。

（2）连接池的作用

1. 资源重用：由于数据库连接得到重用，避免了频繁创建、释放连接引起的大量性能开销。在减少系统消耗的基础上，增进了系统环境的平稳性（减少内存碎片，数据库临时进程、线程的数量）；
2. 更快的系统响应速度：数据库连接池在初始化过程中，往往已经创建了若干数据库连接置于池内备用，此时连接池的初始化操作均已完成。对于业务请求处理而言，直接利用现有可用连接，避免了数据库连接初始化和释放过程的时间开销，从而缩减了系统整体响应时间。
3. 新的资源分配手段：对于多应用共享同一数据库的系统而言，可在应用层通过数据库连接的配置，实现数据库连接技术；
4. 统一的连接管理，避免数据库连接泄漏：在较为完备的数据库连接池实现中，可根据预先的连接占用超时设定，强制收回被占用的连接，从而避免了常规数据库连接操作中可能出现的资源泄漏。

（3）连接池的实现（一）---c3p0

1. 导入c3p0的包，写c3p0-config.xml配置文件（要放在指定的位置）

```
<!-- 自定义的c3p0-config.xml -->
<?xml version="1.0" encoding="UTF-8"?>

<c3p0-config>
  <default-config>
    <property name="driverClass">com.mysql.jdbc.Driver</property>
    <property name="jdbcUrl">jdbc:mysql://localhost:3306/jdbc</property>
    <property name="user">root</property>
    <property name="password">java</property>

    <property name="initialPoolSize">10</property>
    <property name="maxIdleTime">30</property>
    <property name="maxPoolSize">100</property>
    <property name="minPoolSize">10</property>
  </default-config>

  <named-config name="mySource">
    <property name="driverClass">com.mysql.jdbc.Driver</property>
    <property
name="jdbcUrl">jdbc:mysql://localhost:3306/bookstore</property>
    <property name="user">root</property>
    <property name="password">xxxx</property>
```

```
    <property name="initialPoolSize">10</property>
    <property name="maxIdleTime">30</property>
    <property name="maxPoolSize">100</property>
    <property name="minPoolSize">10</property>
  </named-config>
</c3p0-config>
```

```
ComboPooledDataSource ds=new ComboPooledDataSource(); 加载默认的配置
ComboPooledDataSource ds=new ComboPooledDataSource("mySource");加载name为
mySource的配置
```

也可用将配置信息写在properties文件中，

```
ComboPooledDataSource ds = new ComboPooledDataSource();
Properties props = new Properties();
props.load(类
名.class.getClassLoader().getResourceAsStream("c3p0.properties"));
ds.setDriverClass(props.getProperty("driverClass"));
ds.setJdbcUrl(props.getProperty("jdbcUrl"));
```

（4）连接池的实现（二）----DBCP

1.导入DBCP的包，写properties的配置文件，对文件的key有要求，driver\urll\username\password

2.读配置文件，加载输入流

```
Properties props = new Properties();
props.load(类名.class.getClassLoader().getResourceAsStream("dbcp.properties"));
```

3.创建连接池

```
DataSource ds=BasicDataSourceFactory.createDataSource(props);
```

（4）连接池的实现（三）---druid

```
//数据源配置
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setUrl("jdbc:mysql://127.0.0.1/db_student?
serverTimezone=UTC");
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver"); //这个可以缺省
的，会根据url自动识别
        dataSource.setUsername("root");
        dataSource.setPassword("abcd");

        //下面都是可选的配置
        dataSource.setInitialSize(10);  //初始连接数，默认0
        dataSource.setMaxActive(30);  //最大连接数，默认8
        dataSource.setMinIdle(10);  //最小闲置数
        dataSource.setMaxWait(2000);  //获取连接的最大等待时间，单位毫秒
        dataSource.setPoolPreparedStatements(true); //缓存PreparedStatement，默认
false
        dataSource.setMaxOpenPreparedStatements(20); //缓存PreparedStatement的最大
数量，默认-1（不缓存）。大于0时会自动开启缓存PreparedStatement，所以可以省略上一句代码
```

```
        //获取连接
        Connection connection = dataSource.getConnection();
```

（5）使用mybatis框架，可用直接在application.yml或bootstrap.yml中直接配置？？？？

```
spring:
    datasource:
        type: com.alibaba.druid.pool.DruidDataSource
        driverClassName: com.mysql.cj.jdbc.Driver
        url: jdbc:mysql://localhost:3306/test?
useUnicode=true&characterEncoding=UTF-
8&serverTimezone=Asia/Shanghai&rewriteBatchedStatements=true
        username: root
        # 1.配置生成的password
        password: 123456
        #   Druid数据源配置
        # 初始连接数
        initialSize: 5
        # 最小连接池数量
        minIdle: 10
        # 最大连接池数量
        maxActive: 20
        # 配置获取连接等待超时的时间
        maxWait: 60000
        # 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒
        timeBetweenEvictionRunsMillis: 60000
        。。。。。。
```

（6）多数据源如何配置，产生冲突时如何处理

多数据源的典型使用场景：

1. 业务复杂，数据分布在不同的数据库中；
2. 读写分离，一些规模较小的公司，没有专门的中间件团队搭建读写分离基础设施，因此需要业务开发人员自行实现读写分离。在读写分离中，主库和从库的数据是一致的（不考虑主从延迟）。数据更新操作（insert、update、delete）都是在主库上进行，主库将数据变更信息同步给从库，在查询时，可以在从库上进行，从而分担主库的压力。

   读写分离时，可以用com.baomidou.dynamic.datasource

```
spring:
  datasource:
    druid:
      stat-view-servlet:
        enabled: true
        loginUsername: admin
        loginPassword: 123456
    dynamic:
      druid:
        initial-size: 5
        min-idle: 5
        maxActive: 20
        maxWait: 60000
        。。。。
      datasource:
        # 主库数据源
        master:
```

```
            driver-class-name: com.mysql.cj.jdbc.Driver
            url: jdbc:mysql://localhost:3306/ry-cloud?
useUnicode=true&characterEncoding=utf8&zeroDateTimeBehavior=convertToNull&us
eSSL=true&serverTimezone=GMT%2B8
            username: root
            password: 123
          # 从库数据源
          slave:
            username: root
            password: 123
            url: jdbc:mysql://localhost:3306/ry-cloud0?
useUnicode=true&characterEncoding=utf8&zeroDateTimeBehavior=convertToNull&us
eSSL=true&serverTimezone=GMT%2B8
            driver-class-name: com.mysql.cj.jdbc.Driver
```

```
/**
 * 主库数据源
 */
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@DS("master")
public @interface Master
{
}
```

```
/**
 * 从库数据源
 */
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@DS("slave")
public @interface Slave
{
}
```

```
@Slave
public List<SysDept> selectDeptList(SysDept dept)
{
    return deptMapper.selectDeptList(dept);
}
```

3、hashcode()和equals()的关系

（1）hashcode用于在散列存储结构（如：HashTable、HashMap等）中确定对象的存储地址，可用提高查找的快捷性；

（2）equals判断两个变量或实例所指向的同一内存空间的值是不是相同，比较的包括引用和引用指向的内存中的值；

（3）如果两个对象的hashcode相同，equals不一定返回true；如果两个对象的equals方法返回true，hashcode一定相同；

（4）如果对象的equals方法被重写，那么对象的hashcode方法也尽量重写。

4、读取一个文件的方法

（1）按字节读取文件内容

```
FileInputStream fis=new FileInputStream("D:\\a.doc");
byte[] b=new byte[1024];
int len=0;
while((len=fis.read(b))!=-1){
    System.out.println(new String(b,0,len));
}
fis.close();
```

（2）按字符读取文件内容

```
FileReader fr=new FileReader("D:\\a.txt");
char[] c=new char[1024];
int len=0;
while((len=fr.read(c))!=-1){
    System.out.println(new String(c,0,len));
}
fr.close();
```

（3）按行读取文件内容

```
FileReader fr=new FileReader("D:\\a.txt");
BufferedReader br=new BufferedReader(fr);
String line=null;
while((line=br.readLine())!=null){
    System.out.println(line);
}
br.close();
fr.close();
```

（4）随机读取文件内容

```
RandomAccessFile raf=new RandomAccessFile("D:\\a.txt","r");
// 文件长度，字节数
long fileLength = raf.length();
// 读文件的起始位置
int beginIndex = (fileLength > 4) ? 4 : 0;能设置文件读取的起始位置
// 将读文件的开始位置移到beginIndex位置。
raf.seek(beginIndex);
byte[] b=new byte[10];
int len=0;
while((len=raf.read(b))!=-1){
    System.out.println(new String(b,0,len));
}
raf.close();
```

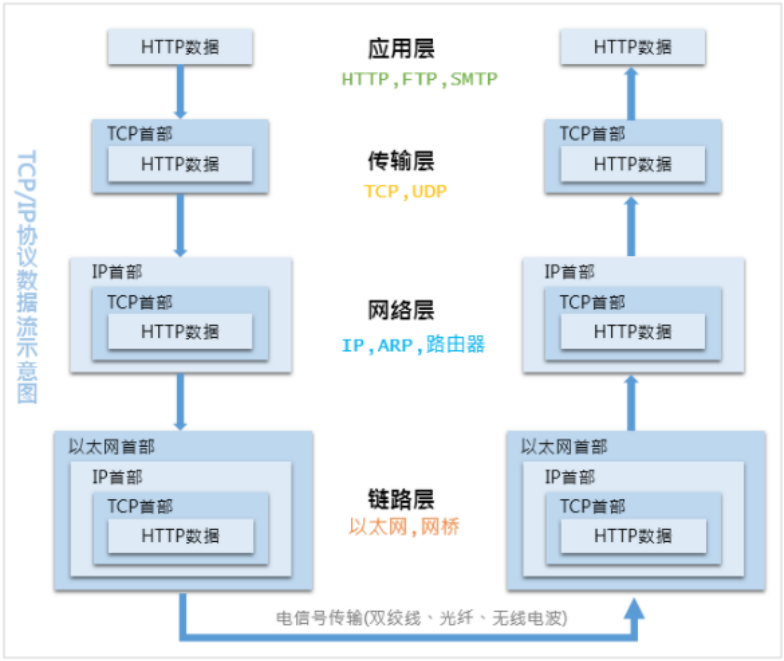
5、TCP/IP协议

（1）TCP/IP协议栈是一系列网络协议的总和，是构成网络通信的核心骨架。它定义了电子设备如何连入因特网，以及数据如何在它们之间进行传输。

（2）TCP/IP协议采用4层结构，分别是：**应用层**，**传输层**，**网络层**，**链路层**。

（3）一个主机的数据要经过哪些过程才能发送到对方的主机上：

0. 物理介质：物理介质就是把电脑连接起来的物理手段，常见的有光纤、双绞线、无线电波等，它决定了电信号（0和1）的传输方式，物理介质的不同决定了电信号的传输带宽、速率、传输距离以及抗干扰性等。TCP/IP协议栈分为四层，每一层由特定的协议与对方进行通信，而协议之间的通信最终都要转化为0和1的电信号，通过物理介质进行传输才能到达对方的电脑，因此，物理介质是网络通信的基石。



当通过http发起一个请求时，应用层、传输层、网络层和链路层的相关协议依次对该请求进行包装并携带对应的**首部**，最终在链路层生成**以太网数据包**，以太网数据包通过物理介质传输给对方主机，对方接收到数据包以后，然后再一层一层采用对应的协议进行拆包，最后把应用层数据交给应用程序处理。

网络通信就好比送快递，商品外面的一层层包裹就是各种协议，协议包含了商品信息、收货地址、收件人、联系方式等，然后还需要配送车、配送站、快递员，商品才能最终到达用户手中。

一般情况下，快递是不能直达的，需要先转发到对应的配送站，然后由配送站再进行派件。

配送车就是物理介质，配送站就是网关， 快递员就是路由器，收货地址就是IP地址，联系方式就是MAC地址。

快递员负责把包裹转发到各个配送站，配送站根据收获地址里的省市区，确认是否需要继续转发到其他配送站，当包裹到达了目标配送站以后，配送站再根据联系方式找到收件人进行派件。

1. 链路层

网络通信就是把有特定意义的数据通过物理介质传送给对方，单纯的发送 0 和 1 是没有意义的，要传输有意义的数据，就需要以字节为单位对 0 和 1 进行分组，并且要标识好每一组电信号的信息特征，然后按照分组的顺序依次发送。以太网规定一组电信号就是一个数据包，一个数据包被称为**一帧，** 制定这个规则的协议就是**以太网协议**。一个完整的以太网数据包如下图所示：

整个数据帧由**首部**、**数据**和**尾部**三部分组成，首部固定为14个字节，包含了目标MAC地址、源MAC地址和类型；数据最短为46个字节，最长为1500个字节，如果需要传输的数据很长，就必须分割成多个帧进行发送；尾部固定为4个字节，表示数据帧校验序列，用于确定数据包在传输过程中是否损坏。因此，以太网协议通过对电信号进行分组并形成数据帧，然后通过物理介质把数据帧发送给接收方。那么以太网如何来识接收方的身份呢?

以太网规协议定，接入网络的设备都必须安装网络适配器，即**网卡，** 数据包必须是从一块网卡传送到另一块网卡。而**网卡地址**就是数据包的发送地址和接收地址，也就是帧首部所包含的**MAC地址，**MAC地址是每块网卡的身份标识，就如同我们身份证上的身份证号码，具有全球唯一性。MAC地址采用十六进制标识，共6个字节， 前三个字节是厂商编号，后三个字节是网卡流水号，例如 **4C-0F-6E-12-D2-19**

有了MAC地址以后，以太网采用**广播**形式，把数据包发给该**子网内**所有主机，子网内每台主机在接收到这个包以后，都会读取首部里的**目标MAC地址**，然后和自己的MAC地址进行对比，如果相同就做下一步处理，如果不同，就丢弃这个包。

所以链路层的主要工作就是**对电信号进行分组并形成具有特定意义的数据帧，然后以广播的形式通过物理介质发送给接收方。**

2. 网络层

对于上面的过程，有几个细节问题值得我们思考：

> 发送者如何知道接收者的MAC地址?
>
> 发送者如何知道接收者和自己同属一个子网?
>
> 如果接收者和自己不在同一个子网，数据包如何发给对方?

为了解决这些问题，网络层引入了三个协议，分别是**IP协议**、**ARP协议**、**路由协议。**

IP协议

通过前面的介绍我们知道，MAC地址只与厂商有关，与所处的网络无关，所以无法通过MAC地址来判断两台主机是否属于同一个子网。

因此，网络层引入了IP协议，制定了一套新地址，使得我们能够区分两台主机是否同属一个网络，这套地址就是网络地址，也就是所谓的**IP地址。**

IP地址目前有两个版本，分别是**IPv4**和**IPv6**，IPv4是一个32位的地址，常采用4个十进制数字表示。IP协议将这个32位的地址分为两部分，前面部分代表网络地址，后面部分表示该主机在局域网中的地址。由于各类地址的分法不尽相同，以C类地址**192.168.24.1**为例，其中前24位就是网络地址，后8位就是主机地址。因此， **如果两个IP地址在同一个子网内，则网络地址一定相同。**为了判断IP地址中的网络地址，IP协议还引入了**子网掩码，** IP地址和子网掩码通过**按位与**运算后就可以得到网络地址。

由于发送者和接收者的IP地址是已知的(应用层的协议会传入)， 因此我们只要通过子网掩码对两个IP地址进行AND运算后就能够判断双方是否在同一个子网了。

ARP协议

即地址解析协议，是根据**IP地址**获取**MAC地址**的一个网络层协议。其工作原理如下：

ARP首先会发起一个请求数据包，数据包的首部包含了目标主机的IP地址，然后这个数据包会在链路层进行再次包装，生成**以太网数据包，**最终由以太网广播给子网内的所有主机，每一台主机都会接收到这个数据包，并取出标头里的IP地址，然后和自己的IP地址进行比较，如果相同就返回自己的MAC地址，如果不同就丢弃该数据包。ARP接收返回消息，以此确定目标机的MAC地址；与此同时，ARP还会将返回的MAC地址与对应的IP地址存入本机ARP缓存中并保留一定时间，下次请求时直接查询ARP缓存以节约资源。cmd输入 arp -a 就可以查询本机缓存的ARP数据。

路由协议

通过ARP协议的工作原理可以发现，**ARP的MAC寻址还是局限在同一个子网中**，因此网络层引入了路由协议，首先通过IP协议来判断两台主机是否在同一个子网中，如果在同一个子网，就通过ARP协议查询对应的MAC地址，然后以广播的形式向该子网内的主机发送数据包；如果不在同一个子网，以太网会将该数据包转发给本子网的**网关**进行路由。网关是互联网上子网与子网之间的桥梁，所以网关会进行多次转发，最终将该数据包转发到目标IP所在的子网中，然后再通过ARP获取目标机MAC，最终也是通过广播形式将数据包发送给接收方。

而完成这个路由协议的物理设备就是**路由器**，在错综复杂的网络世界里，路由器扮演者**交通枢纽**的角色，它会根据信道情况，选择并设定路由，以最佳路径来转发数据包。

IP数据包

在网络层被包装的数据包就叫**IP数据包**，IPv4数据包的结构如下图所示：

IP数据包由首部和数据两部分组成，首部长度为20个字节，主要包含了目标IP地址和源IP地址，目标IP地址是网关路由的线索和依据；数据部分的最大长度为65515字节，理论上一个IP数据包的总长度可以达到65535个字节，而以太网数据包的最大长度是1500个字符，如果超过这个大小，就需要对IP数据包进行分割，分成多帧发送。

所以，网络层的主要工作是**定义网络地址，区分网段，子网内MAC寻址，对于不同子网的数据包进行路由。**

3. 传输层

链路层定义了主机的身份，即MAC地址， 而网络层定义了IP地址，明确了主机所在的网段，有了这两个地址，数据包就从可以从一个主机发送到另一台主机。但实际上数据包是从一个主机的某个应用程序发出，然后由对方主机的应用程序接收。而每台电脑都有可能同时运行着很多个应用程序，所以当数据包被发送到主机上以后，是无法确定哪个应用程序要接收这个包。

因此传输层引入了**UDP协议**来解决这个问题，为了给每个应用程序标识身份，UDP协议定义了**端口**，同一个主机上的每个应用程序都需要指定唯一的端口号，并且规定网络中传输的数据包必须加上端口信息。 这样，当数据包到达主机以后，就可以根据端口号找到对应的应用程序了。UDP定义的数据包就叫做UDP数据包，结构如下所示：

UDP数据包由首部和数据两部分组成，首部长度为8个字节，主要包括源端口和目标端口；数据最大为65527个字节，整个数据包的长度最大可达到65535个字节。

UDP协议比较简单，实现容易，但它没有确认机制， 数据包一旦发出，无法知道对方是否收到，因此可靠性较差，为了解决这个问题，提高网络可靠性，**TCP协议**就诞生了，TCP即传输控制协议，是一种面向连接的、可靠的、基于字节流的通信协议。简单来说**TCP就是有确认机制的UDP协议，**每发出一个数据包都要求确认，如果有一个数据包丢失，就收不到确认，发送方就必须重发这个数据包。

为了保证传输的可靠性，TCP 协议在 UDP 基础之上建立了**三次对话**的确认机制，也就是说，在正式收发数据前，必须和对方建立可靠的连接。由于建立过程较为复杂，我们在这里做一个形象的描述：

> 主机A：我想发数据给你，可以么？
>
> 主机B：可以，你什么时候发？
>
> 主机A：我马上发，你接着！

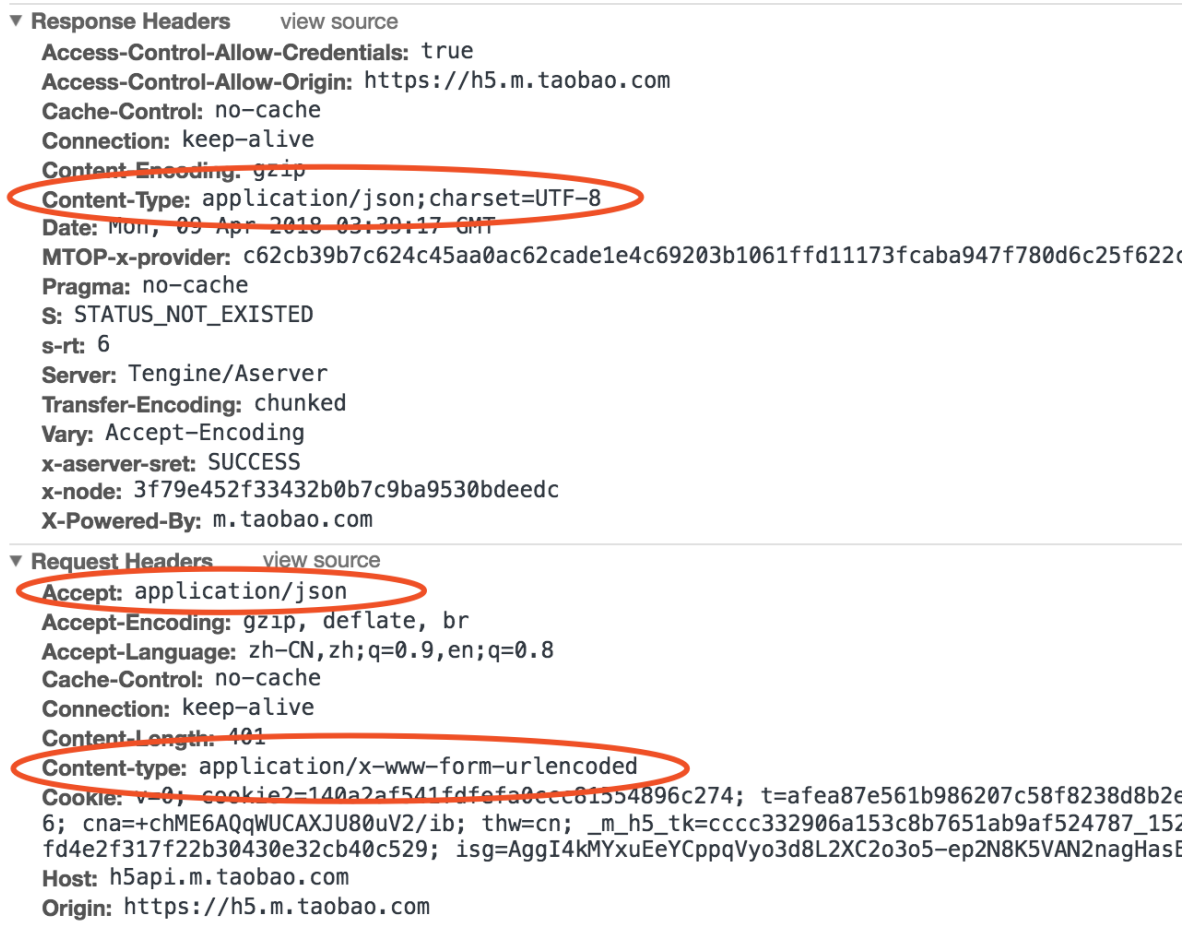
经过三次对话之后，主机A才会向主机B发送正式数据，而UDP是面向非连接的协议，它不与对方建立连接，而是直接就把数据包发过去了。所以 TCP 能够保证数据包在传输过程中不被丢失，但美好的事物必然是要付出代价的，相比 UDP，TCP 实现过程复杂，消耗连接资源多，传输速度慢。

TCP 数据包和 UDP 一样，都是由首部和数据两部分组成，唯一不同的是，TCP 数据包没有长度限制，理论上可以无限长，但是为了保证网络的效率，通常 TCP 数据包的长度不会超过IP数据包的长度，以确保单个 TCP 数据包不必再分割。

总结一下，传输层的主要工作是**定义端口，标识应用程序身份，实现端口到端口的通信，TCP协议可以保证数据传输的可靠性。**

4. 应用层

理论上讲，有了以上三层协议的支持，数据已经可以从一个主机上的应用程序传输到另一台主机的应用程序了，但此时传过来的数据是字节流，不能很好的被程序识别，操作性差。因此，应用层定义了各种各样的协议来规范数据格式，常见的有 HTTP、FTP、SMTP 等，HTTP 是一种比较常用的应用层协议，主要用于B/S架构之间的数据通信，其报文格式如下：



在 Resquest Headers 中，Accept 表示客户端期望接收的数据格式，而 ContentType 则表示客户端发送的数据格式；在 Response Headers 中，ContentType 表示服务端响应的数据格式，这里定义的格式，一般是和 Resquest Headers 中 Accept 定义的格式是一致的。

有了这个规范以后，服务端收到请求以后，就能正确的解析客户端发来的数据，当请求处理完以后，再按照客户端要求的格式返回，客户端收到结果后，按照服务端返回的格式进行解析。

所以应用层的主要工作就是**定义数据格式并按照对应的格式解读数据。**

（4）全流程

- **链路层**：对0和1进行分组，定义数据帧，确认主机的物理地址，传输数据；
- **网络层**：定义IP地址，确认主机所在的网络位置，并通过IP进行MAC寻址，对外网数据包进行路由转发；
- **传输层**：定义端口，确认主机上应用程序的身份，并将数据包交给对应的应用程序；
- **应用层**：定义数据格式，并按照对应的格式解读数据。

然后再把每层模型的职责串联起来，用一句通俗易懂的话讲就是：

> 当你输入一个网址并按下回车键的时候，首先，应用层协议对该请求包做了格式定义；紧接着传输层协议加上了双方的端口号，确认了双方通信的应用程序；然后网络协议加上了双方的IP地址，确认了双方的网络位置；最后链路层协议加上了双方的MAC地址，确认了双方的物理位置，同时将数据进行分组，形成数据帧，采用广播方式，通过传输介质发送给对方主机。而对于不同

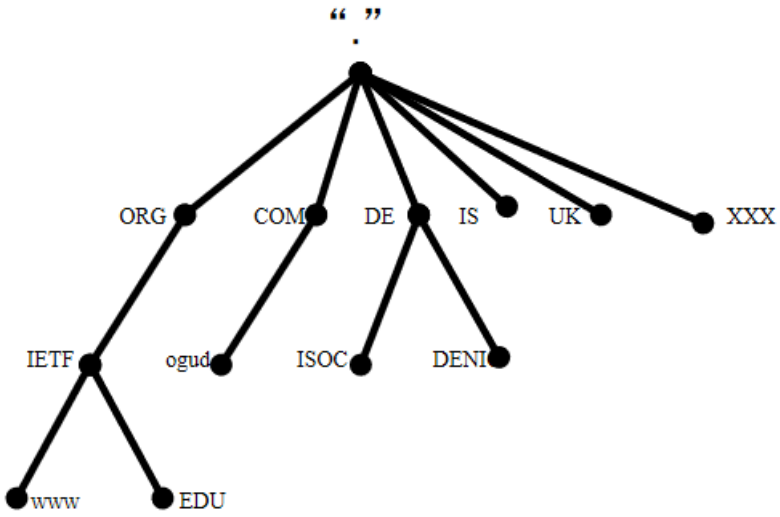
> 网段，该数据包首先会转发给网关路由器，经过多次转发后，最终被发送到目标主机。目标机接收到数据包后，采用对应的协议，对帧数据进行组装，然后再通过一层一层的协议进行解析，最终被应用层的协议解析并交给服务器处理。

6、DNS原理及其解析过程

（1）什么是DNS？DNS（Domain Name System）是“域名系统”的英文缩写，是一种组织成域层次结构的计算机和网络服务命名系统，它用于TCP/IP网络，它所提供的服务是用来将主机名和域名转换为IP地址的工作。

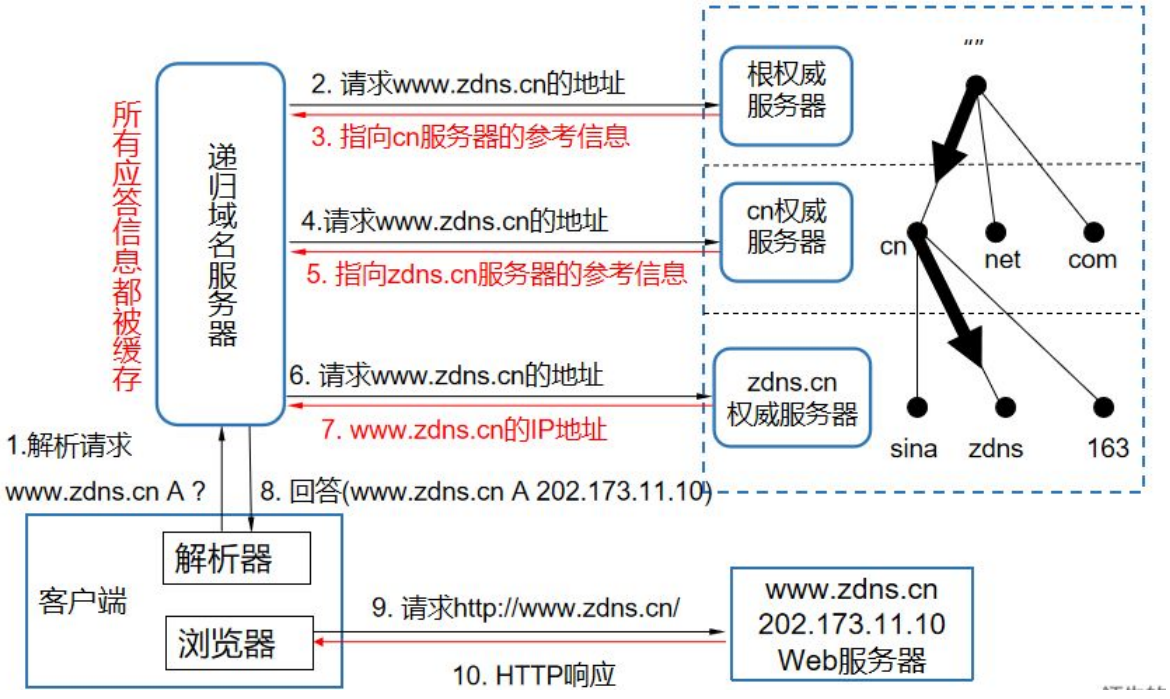
（2）域名系统作为一个层次结构和分布式数据库，包含各种类型的数据，包括主机名和域名。DNS数据库中的名称形成一个分层树状结构，称为域命名空间，域名包含单个标签分隔点。

（3）完全限定域名（FQDN）包含两部分：主机名和域名。FQDN总是以主机名开始并且以顶级域名结束。域名解析结构：



域名结构是树状结构，树的最顶端代表根服务器，下面的依次是：顶级域，二级域，子域，主机名。

（4）DNS解析流程



1. 客户端打开浏览器以后，比如输入zdns.cn域名，它首先是由浏览器发起一个DNS解析请求，如果本地缓存服务器中找不到结果，则首先会向根服务器查询，根服务器里面记录的都是各个顶级域所

在的服务器的位置，当向根请求zdns.cn时，根服务器会返回.cn服务器的位置信息；
2. 递归服务器拿到.cn的权威服务器地址以后，就会询问cn的权威服务器，知不知道zdns.cn的位置，这个时候cn权威服务器查找并返回zdns.cn服务器的地址；
3. 继续向zdns.cn的权威服务器去查询这个地址，由zdns.cn的服务器给出了地址202.173.11.10；
4. 最终才能进行http的链接，顺利访问网站；
5. 一旦递归服务器拿到解析记录，就会在本地进行缓存，如果下次客户端再请求本地的递归域名服务器相同域名的时候，就直接使用本地服务器已经由的缓存。

7、TCP和UDP的区别和各自优缺点

（1）是否基于连接：TCP是面向连接的协议；UDP是无连接的协议，即发送数据之前不需要建立连接。

（2）可靠性和有序性：TCP提供交付保证，无差错、不丢失、不重复、且按序到达，也保证了消息的有序性；UDP不提供任何有序或序列性的保证，UDP尽最大努力交付，数据包将以任何可能的顺序到达；TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道；

（3）实时性：UDP具有较好的实时性，工作效率比TCP高，适用于对高速传输和实时性要求较高的通信或广播通信；

（4）协议首部大小：TCP首部开销20字节；UDP首部开销8字节；

（5）运行速度：TCP速度比较慢，UDP速度比较快，因为TCP必须创建连接，以保证消息的可靠交付和有序性；

（6）拥塞机制：UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低；

（7）流模式（TCP）和数据报模式（UDP）：TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流；UDP是面向报文的；

（8）资源占用：TCP对系统资源要求比较多；UDP对系统资源要求比较少；

（9）应用：TCP连接只能是点对点的；UDP支持一对一、一对多、多对一和多对多的交互通信。

UDP优点：简单、传输快；（1）网速的提升给UDP的稳定性提供可靠网络保证，丢包率很低，如果使用应用层重传，能够确保传输的可靠性；（2）TCP为了实现网络通信的可靠性，使用了复杂的拥塞控制算法，建立了复杂的握手过程，采用TCP，一旦发生丢包，TCP会将后续的包缓存起来，等前面的包重传并接收到后再继续发送，延时会越来越大；基于UDP对实时性要求较为严格的情况下，采用自定义重传机制，能够把丢包产生的延迟降到最低。

UDP缺点：不可靠、不稳定；

UDP应用场景：面向数据报方式；网络数据大多为短消息；拥有大量client；对数据安全性无特殊要求；网络负担非常高，但对响应速度要求高。如：IP电话、实时视频会议。

TCP优点：可靠、稳定；TCP的可靠体现在TCP在传输数据之前，会有三次握手来创建连接，而且在数据传递时，有确认、窗口、重传、拥塞控制机制，在数据传完之后，还会断开连接节省系统资源；

TCP缺点：慢、效率低、占用系统资源高、易被攻击；

TCP应用场景：对数据准确性要求较高，速度可以相对较慢的，如文件传输、邮件的发送与接收等。

8、java引用类型有哪几种

java主要有8种基本数据类型和引用数据类型；

基本数据类型：boolean、char、byte、short、int、long、float、double；

引用数据类型：类、接口、数组。

9、集合线程安全问题

（1）ArrayList、LinkedList、HashSet、LinkedHashSet是线程不安全的集合，Vector是线程安全的集合。

（2）导致集合线程不安全的原因：并发争抢导致，例如一个人正在写入，另一个人来争抢，导致数据不一致异常，并发修改异常。

（3）解决ArrayList线程安全问题方案

```
List<String> list1=new Vector<>();方案一
List<String> list2= Collections.synchronizedList(new ArrayList<>());方案二
List<String> list3=new CopyOnWriteArrayList<>();方案三
```

从CopyOnWriteArrayList的源码可以看出，其add方法中先对集合复制，再添加元素，是写时复制，读写分离的思想。添加完成元素之后，再将原容器的引用指向新的容器setArray(newElements),这样做的好处是可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。

10、对象序列化的方法

（1）序列化：将对象的状态信息转换为可以存储或传输的形式的过程。目的：以某种存储形式使自定义对象持久化；将对象从一个地方传递到另一个地方。

（2）对象的序列化：将java对象，以流的形式写入到文件中保存，即写出对象。

```
FileOutputStream fos=new FileOutputStream("c:\\person.txt");
ObjectOutputStream oos=new ObjectOutputStream(fos);
Person p=new Person();
p.setName("aaa");
oos.writeObject(p);
```

示例中的Person类必须实现Serializable接口，否则会抛异常。

（3）对象的反序列化：以流的形式，将对象从文件中读取出来，即读取对象。

```
FileInputStream fis=new FileInputStream("c:\\person.txt");
ObjectInputStream ois=new ObjectInputStream(fis);
Object obj=ois.readObject();
```

（4）注意：

1. 对象的类名,属性(包括基本类型,数组,对其他对象的引用)都会被序列化;
2. 方法, static属性(静态属性), transient属性(瞬态属性)都不会被序列化.
3. 保证序列化对象的属性的类型也是可序列化的, 否则, 该类是不可序列化的.
4. 反序列化对象时必须有对象的class文件.

（5）若要自定义序列化机制，也可使对象的类实现Externalizable接口，但是，要实现两个方法

```
@Override
public void writeExternal(ObjectOutput out) throws IOException {
}
@Override
public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
}
```

（6） 实现Serializable接口 和实现Externalizable接口序列化对比

实现Serializable接口	实现Externalizable接口
系统自动存储必要信息	程序员决定存储哪些信息(自定义序列化)
易于实现,只需实现该接口 即可,无需任何代码支持	提供两个空方法,实现该接口必须为两个空方法提供实现
性能略差	性能略高

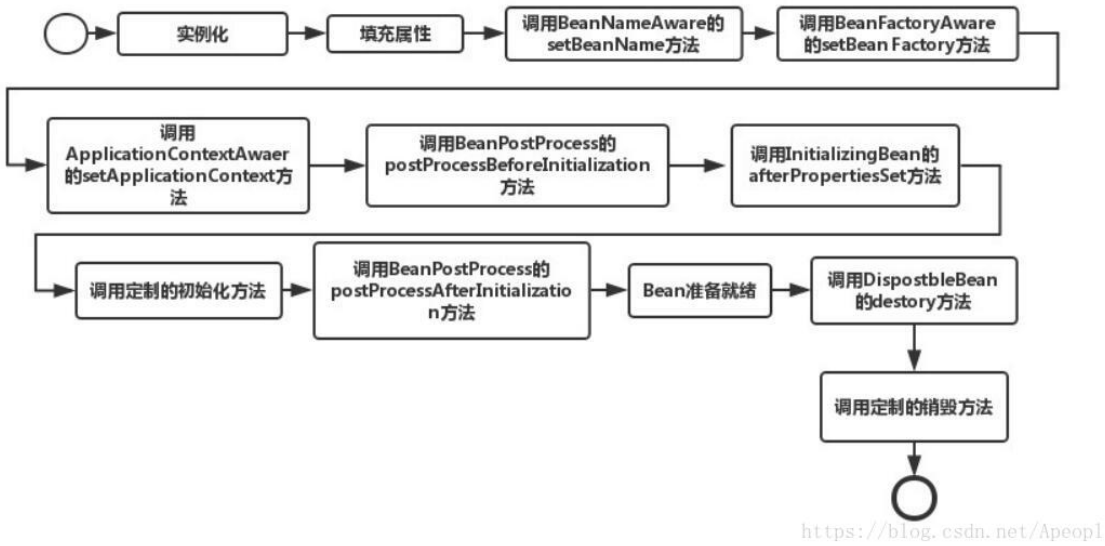
（7）序列化时的序列号冲突问题

对创建对象的类的源代码进行修改，会重新编译生成class文件，根据类的成员，重新生成序列号，此时反序列化，就会出现序列号冲突的问题。

解决方法：在类中自定义序列号，编译器不会生成序列号，如：

```
static final long serialVersionUID=42L;
```

11、Spring bean的生命周期



Spring Bean的生命周期分四个阶段（偏向过程），实例化（Instantiation）、属性赋值（Populate）、初始化（Initialization）、销毁（Destruction）。

（1）对象的生命周期（偏向生命长度）：单例对象的生命周期与容器相同；多例对象在使用对象时创建，当对象长时间不用，且没有别的对象引用时，由Java的垃圾回收器回收。

12、Spring框架的项目，在项目启动时就执行指定的方法。

（1）在方法上加注解@PostConstruct

```
@PostConstruct
public void init(){
    //程序启动就会执行这个方法
}
```

（2）xml配置init-method

```
<bean id="InitDemo" class="com.xxx.InitDEmo" scope="singleton" init-
method="init">
</bean>
```

（3）实现InitializingBean接口，重写afterPropertiesSet方法

```
public class InitDemo implements InitializingBean {
    @Override
    public void afterPropertiesSet() throws Exception {
        //项目启动就会执行这个方法
    }
}
```

注：spring bean的初始化执行顺序：构造方法--> @PostConstruct注解的方法--> afterPropertiesSet方法-->init-method指定的方法；afterPropertiesSet通过接口实现方式调用，效率上高一点；@PostConstruct和init-method都是通过反射机制调用。

13、spring bean的作用范围

bean标签的scope属性，用于指定bean的作用范围，取值有：

（1）singleton：单例（默认），使用该属性定义bean时，IOC容器仅创建一个Bean实例，IOC容器每次返回的是同一个Bean实例。

（2）prototype：多例，IOC容器可以创建多个Bean实例，每次返回的都是一个新的实例。

（3）request：仅对HTTP请求产生作用，每次HTTP请求都会创建一个新的Bean，适用于WebApplicationContext环境。

（4）session：仅用于HTTP Session，同一个Session共享一个Bean实例，不同的Session使用不同的Session。

（5）global-session：仅用于HTTP Session，同session作用域不同的是，所以Session共享一个Bean实例。

【注】你使用了单例作用域的bean,会出现线程不安全问题么？什么情况下会出现线程不安全？如何解决这种问题?

会出现线程不安全问题，单例情况下，不能使用共享成员变量，在并发情况下本来是true，但另一线程进来，变为false，期望结果是true，并发就导致了数据不准确。

解决办法：1. 使用jdk提供的ThreadLocal来包装该成员变量，即线程间的数据隔离；2. 使用prototype作用域，每次都会创建新的bean。