**数据结构：**

线性表、栈、队列、树、图

常见算法 （查找、排序、找最优解）

Leetcode

剑指offer

**Java :**

并发

IO(nio)

Collection

数据结构

反射

JVM(内存、调优)

jdbc

**SQL:**

Mysql

索引、事务、存储过程

mongoDB

oracle

**Web:**

Tomcat

Servlet/jsp

Session cookie

**存储：**

XML json

**设计模式：**

***单例：***

（饿汉式线程安全、双重校验锁、登记式静态内部类、枚举）

**框架：**

IOC/AOP: spring

MVC : springMVC

ORM: mybatis

**网络协议：**

http(s)

TCP/IP

**Linux:**

Shell

Network

Makefile

Awk/sed

**缓存：**

nosql

Redis

**工具：**

Git/maven

**Nginx:**

**RPC：**

**分布式架构：**

**1、get post 区别**

GET 方法

请注意，查询字符串（名称/值对）是在 GET 请求的 URL 中发送的：

/test/demo\_form.asp?name1=value1&name2=value2

有关 GET 请求的其他一些注释：

GET 请求可被缓存

GET 请求保留在浏览器历史记录中

GET 请求可被收藏为书签

GET 请求不应在处理敏感数据时使用

GET 请求有长度限制

GET 请求只应当用于取回数据

比较 GET 与 POST

下面的表格比较了两种 HTTP 方法：GET 和 POST。

GET POST

后退按钮/刷新 无害 数据会被重新提交（浏览器应该告知用户数据会被重新提交）。

书签 可收藏为书签 不可收藏为书签

缓存 能被缓存 不能缓存

编码类型 application/x-www-form-urlencoded application/x-www-form-urlencoded 或 multipart/form-data。为二进制数据使用多重编码。

历史 参数保留在浏览器历史中。 参数不会保存在浏览器历史中。

对数据长度的限制 是的。当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）。 无限制。

对数据类型的限制 只允许 ASCII 字符。 没有限制。也允许二进制数据。

安全性

与 POST 相比，GET 的安全性较差，因为所发送的数据是 URL 的一部分。

在发送密码或其他敏感信息时绝不要使用 GET ！

POST 比 GET 更安全，因为参数不会被保存在浏览器历史或 web 服务器日志中。

可见性 数据在 URL 中对所有人都是可见的。 数据不会显示在 URL 中。

其他 HTTP 请求方法

下面的表格列出了其他一些 HTTP 请求方法：

方法 描述

HEAD 与 GET 相同，但只返回 HTTP 报头，不返回文档主体。

PUT 上传指定的 URI 表示。

DELETE 删除指定资源。

OPTIONS 返回服务器支持的 HTTP 方法。

CONNECT 把请求连接转换到透明的 TCP/IP 通道。

GET在浏览器回退时是无害的，而POST会再次提交请求。 GET产生的URL地址可以被Bookmark，而POST不可以。 GET请求会被浏览器主动cache，而POST不会，除非手动设置。 GET请求只能进行url编码，而POST支持多种编码方式。 GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。 GET请求在URL中传送的参数是有长度限制的，而POST么有。 对参数的数据类型，GET只接受ASCII字符，而POST没有限制。 GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。 GET参数通过URL传递，POST放在Request body中。

在我大万维网世界中，TCP就像汽车，我们用TCP来运输数据，它很可靠，从来不会发生丢件少件的现象。但是如果路上跑的全是看起来一模一样的汽车，那这个世界看起来是一团混乱，送急件的汽车可能被前面满载货物的汽车拦堵在路上，整个交通系统一定会瘫痪。为了避免这种情况发生，交通规则HTTP诞生了。HTTP给汽车运输设定了好几个服务类别，有GET, POST, PUT, DELETE等等，HTTP规定，当执行GET请求的时候，要给汽车贴上GET的标签(设置method为GET)，而且要求把传送的数据放在车顶上(url中)以方便记录。如果是POST请求，就要在车上贴上POST的标签，并把货物放在车厢里。当然，你也可以在GET的时候往车厢内偷偷藏点货物，但是这是很不光彩;也可以在POST的时候在车顶上也放一些数据，让人觉得傻乎乎的。HTTP只是个行为准则，而TCP才是GET和POST怎么实现的基本。

但是，我们只看到HTTP对GET和POST参数的传送渠道(url还是requrest body)提出了要求。“标准答案”里关于参数大小的限制又是从哪来的呢?

在我大万维网世界中，还有另一个重要的角色：运输公司。不同的浏览器(发起http请求)和服务器(接受http请求)就是不同的运输公司。 虽然理论上，你可以在车顶上无限的堆货物(url中无限加参数)。但是运输公司可不傻，装货和卸货也是有很大成本的，他们会限制单次运输量来控制风险，数据量太大对浏览器和服务器都是很大负担。业界不成文的规定是，(大多数)浏览器通常都会限制url长度在2K个字节，而(大多数)服务器最多处理64K大小的url。超过的部分，恕不处理。如果你用GET服务，在request body偷偷藏了数据，不同服务器的处理方式也是不同的，有些服务器会帮你卸货，读出数据，有些服务器直接忽略，所以，虽然GET可以带request body，也不能保证一定能被接收到哦。

好了，现在你知道，GET和POST本质上就是TCP链接，并无差别。但是由于HTTP的规定和浏览器/服务器的限制，导致他们在应用过程中体现出一些不同。

GET和POST还有一个重大区别，简单的说：

GET产生一个TCP数据包;POST产生两个TCP数据包。

长的说：

对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200(返回数据);

而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok(返回数据)。

也就是说，GET只需要汽车跑一趟就把货送到了，而POST得跑两趟，第一趟，先去和服务器打个招呼“嗨，我等下要送一批货来，你们打开门迎接我”，然后再回头把货送过去。

因为POST需要两步，时间上消耗的要多一点，看起来GET比POST更有效。因此Yahoo团队有推荐用GET替换POST来优化网站性能。但这是一个坑!跳入需谨慎。为什么?

1. GET与POST都有自己的语义，不能随便混用。

2. 据研究，在网络环境好的情况下，发一次包的时间和发两次包的时间差别基本可以无视。而在网络环境差的情况下，两次包的TCP在验证数据包完整性上，有非常大的优点。

3. 并不是所有浏览器都会在POST中发送两次包，Firefox就只发送一次。

**2、https**

https中的s表示SSL或者TLS，就是在原http的基础上加上一层用于数据加密、解密、身份认证的安全层。

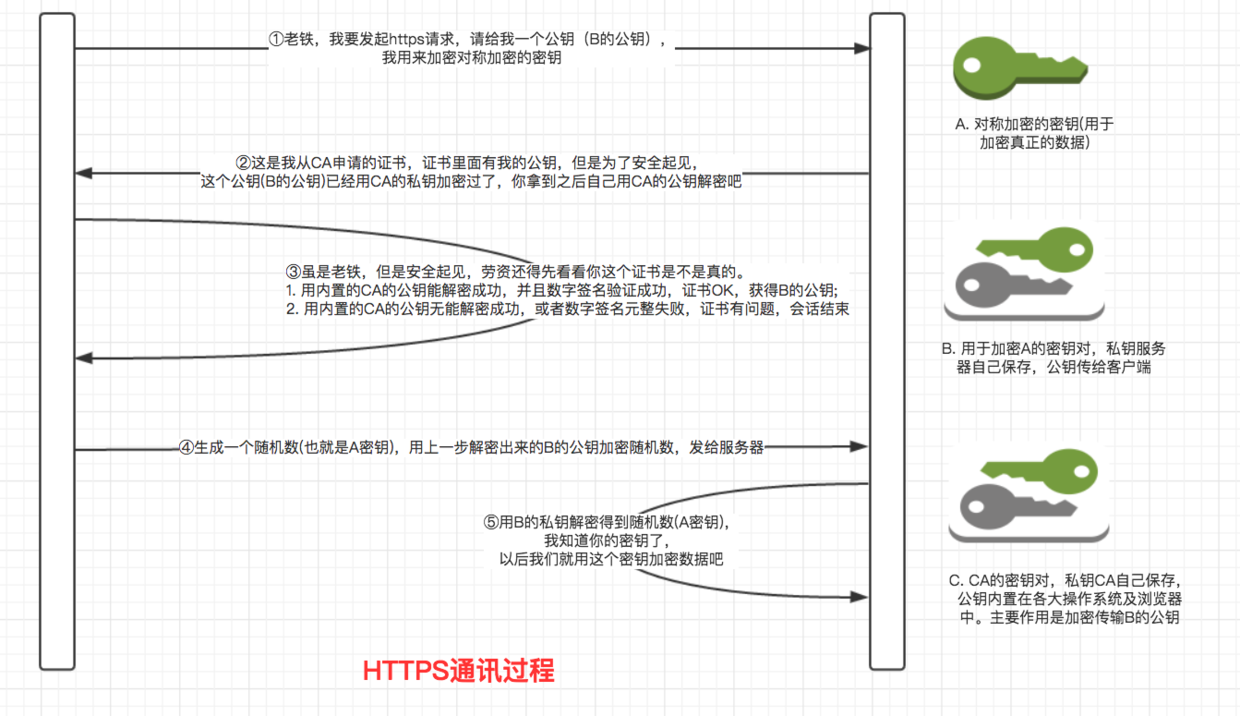
**\*\*\*\*\*\*对称加密传输数据 安全传输数据**

**\*\*\*\*\*\*\*客户端（用服务器公钥）非对称加密传输对称加密的密钥 安全传输密钥**

**\*\*\*\*\*\*\*\*CA（用私钥）非对称加密 服务器非对称加密的公钥 安全传输公钥**

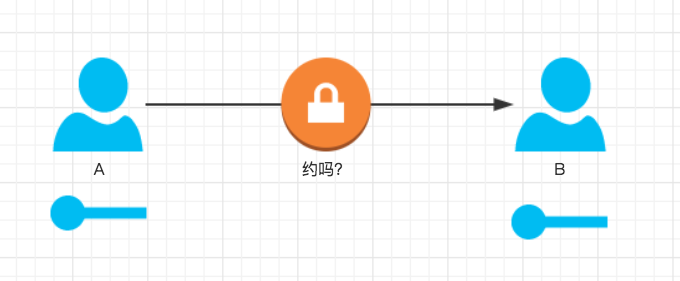
**\*\*\*\*\*\*\*\*\*\* CA加证书签名 安全传输证书**

**一层层揭开HTTPS神秘面纱**

虽然要层层渐进，但是我们不妨先奉上刚画好的还热乎着的https通信完整流程图：   


#### 第一层（安全传输数据）

假如我们要实现一个功能：**一个用户A给一个用户B发消息，但是要保证这个消息的内容只能被A和B知道，**



对于第一个问题，加密算法分为两类：对称加密和非对称加密，这里我们选择对称机密，原因有如下几个：

1. 对称加密速度快，加密时CPU资源消耗少；
2. 非对称加密对待加密的数据的长度有比较严格的要求，不能太长，但是实际中消息可能会很长

对于第二个问题，这是导致整个https通信过程很复杂的根本原因。   
如果A或B直接把他们之间用于解密的密钥通过互联网传输给对方，那一旦密钥被第三者劫持，第三者就能正确解密A,B之间的通信数据。

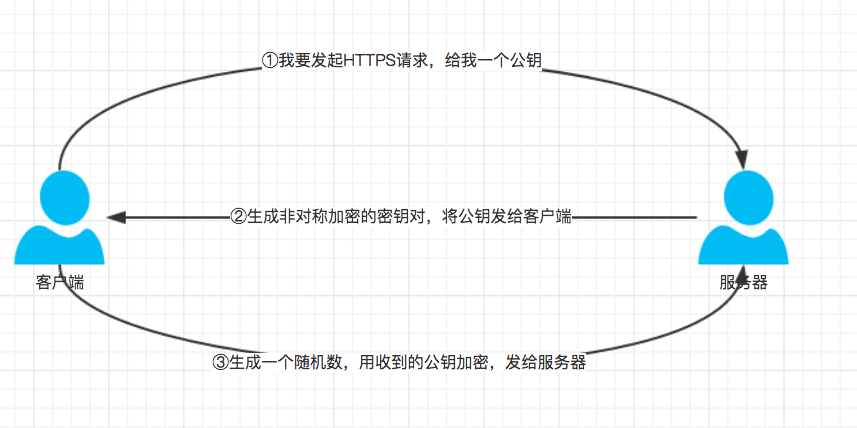
#### 第二层（安全传输密钥）

通过第一层的描述，第二层需要解决的问题是：**安全地传输A,B之间用于解密数据的密钥**

因为如果传输过程中这把密钥被第三者拿到了，就能解密传通信数据，所以，这把密钥必须得加密，就算第三者劫持到这把加密过的密钥，他也不能解密，得到真正的密钥。

这里有一个问题，那要用什么方式加密这把密钥呢？如果使用对称加密，那这个对称加密的密钥又怎么安全地告诉对方呢？完了，陷入死循环了…. 所以，一定不能用对称加密

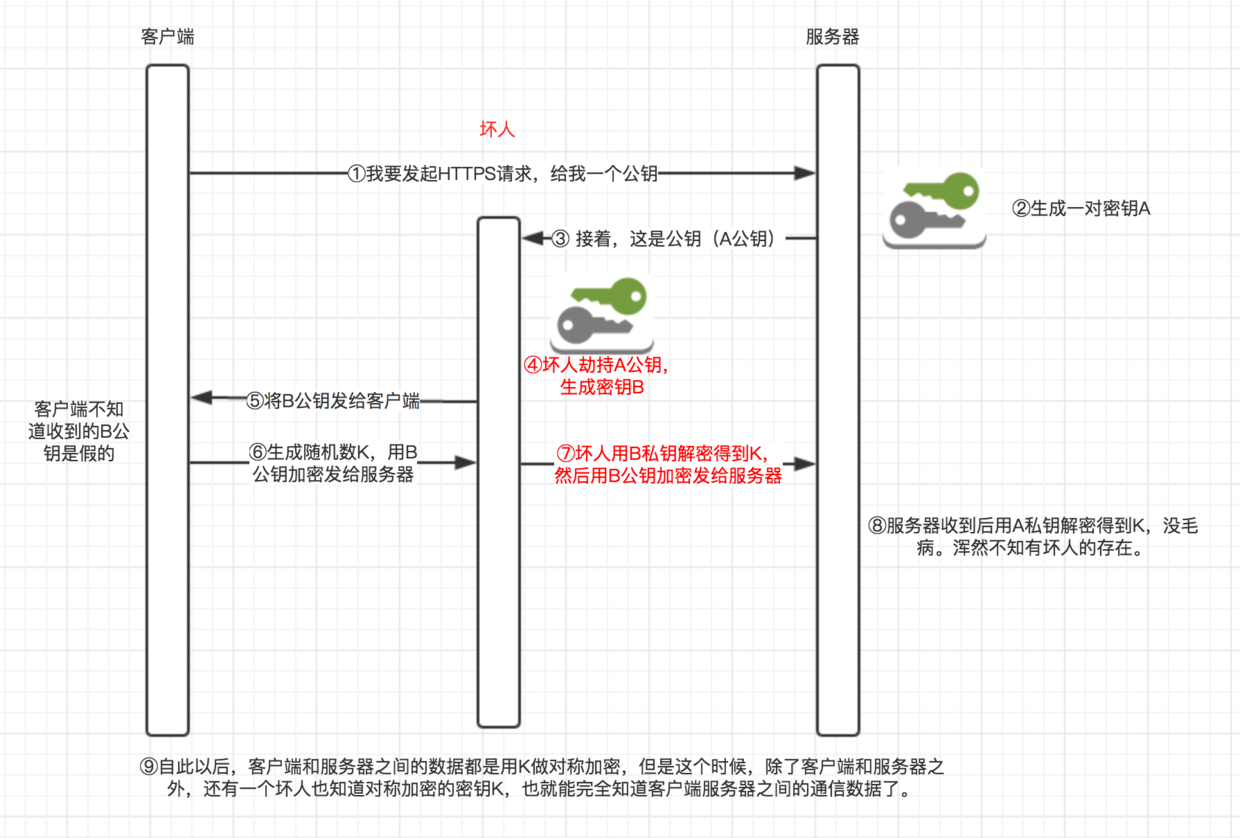
那就是用非对称加密咯，那如何应用非对称加密来加密那把密钥呢



1. 客户端： 我要发起HTTPS请求，麻烦给我一个非对称加密的公钥；
2. 服务器： (生成一对非对称加密的密钥对，然后把公钥发给客户端)，接着，这是公钥；
3. 客户端：（收到公钥，生成一个随机数，作为上图中那一把密钥，用刚才收到的公钥加密这个密钥，然后发给服务器）这是我刚生成的加密过的密钥；
4. 服务器：（收到加密后的密钥，用本地的第一步自己生成的非对称加密的私钥解密，得到真正的密钥）;
5. 现在，客户端和服务器都知道了这把密钥，就能愉快地用这个密钥对称加密数据…

**上述步骤中最终用于加密数据的密钥是客户端生成并且用公钥加密之后传给服务器的，因为私钥只有服务器才有，所以也就只有服务器才能解开客户端上报的密钥；**

**要保证传输的密钥只能被服务器解密，就得保证用于加密密钥的公钥一定是服务器下发的，绝对不可能被第三方篡改过；**



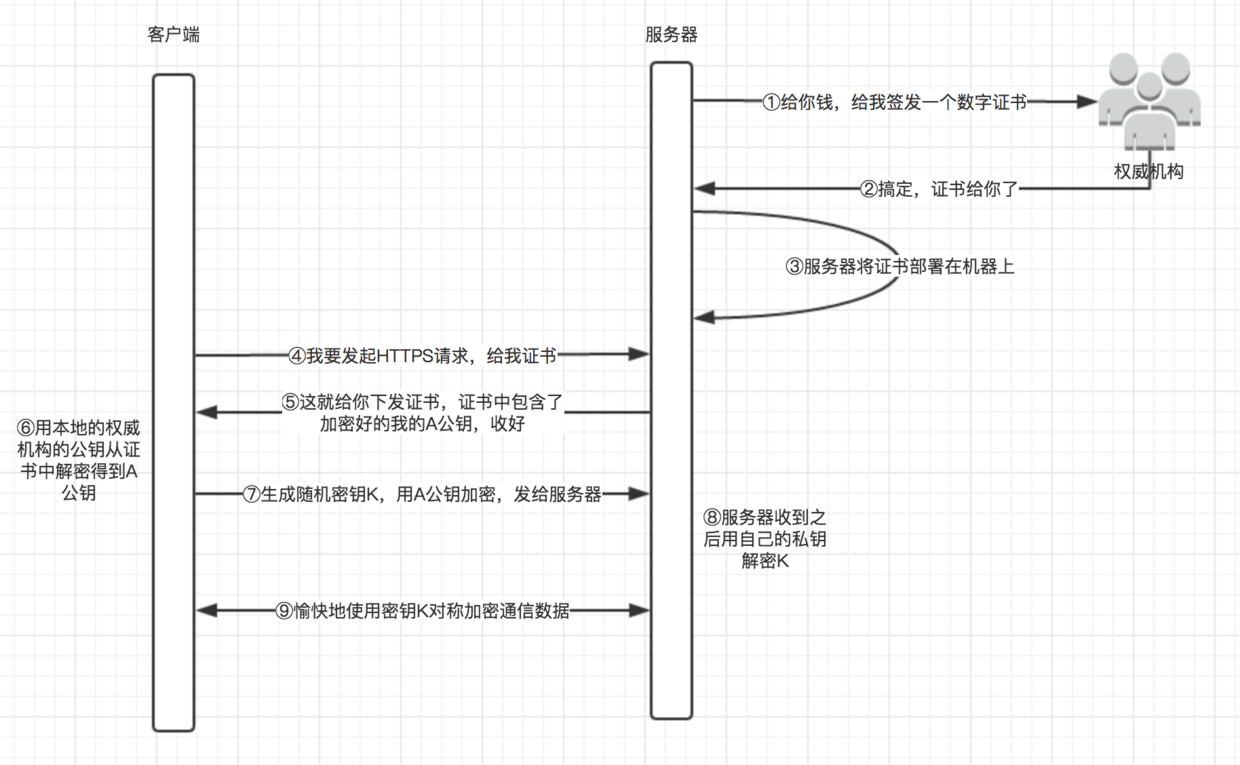
**上图第7步，应该是『坏人用B私钥解密得到K，然后使用A公钥加密发给服务器』**

这种情况下，客户端和服务器之间通信的数据就完全被坏人破解了。

#### 第三层（安全传输公钥）

从上一层可以知道，要保证数据的安全，就必须得保证服务器给客户端下发的公钥是真正的公钥，而不是中间人伪造的公钥。那怎么保证呢？

那就得引入数字证书了，数字证书是服务器主动去权威机构申请的，证书中包含了上一个图中的加密过的A公钥和权威机构的信息，所以服务器只需要给客户端下发数字证书即可。现在流程图如下：



那数字证书中的A公钥是如何加密的呢？

答案是非对称加密，只不过这里是使用只有权威机构自己才有的私钥加密。

等一下，既然A公钥被权威机构的私钥加密了，那客户端收到证书之后怎么解密证书中的A公钥呢?需要有权威机构的公钥才能解密啊！**那这个权威机构的公钥又是怎么安全地传输给客户端的呢？**感觉进入了鸡生蛋，蛋生鸡的悖论了~~

别慌，**答案是权威机构的公钥不需要传输，因为权威机构会和主流的浏览器或操作系统合作，将他们的公钥内置在浏览器或操作系统环境中。**客户端收到证书之后，只需要从证书中找到权威机构的信息，并从本地环境中找到权威机构的公钥，就能正确解密A公钥。

这样就绝对安全了吗？既然权威技能能给服务器签发数字证书，那为什么就不可能给中间人签发数字证书呢?毕竟赚钱的生意权威机构也不会拒绝的呀。

试想一下：

服务器给客户端下发数字证书时证书被中间人劫持了，中间人将服务器的证书替换成自己的证书下发给客户端，客户端收到之后能够通过权威机构的公钥解密证书内容（因为中间人的证书也是权威机构私钥加密的），从而获取公钥，但是，这里的公钥并不是服务器原本的A公钥，而是中间人自己证书中的B公钥。从第二层可知，如果不能保证客户端收到的公钥是服务器下发的，那整个通信数据的安全就没法保证。简单总结就是证书被调包~

所以，还得保证客户端收到的证书就是服务器下发的证书，没有被中间人篡改过。

#### 第四层（安全传输证书）

这一层，我们的任务是：**保证客户端收到的证书是服务器下发的证书，没有被中间人篡改过。**

所以，这里就有两个需求:

* 证明证书内容没有被第三方篡改过；
* 证明证书是服务器下发的；

其实这些问题，数字证书本身已经提供方案了，数字证书中除了包含加密之后的服务器公钥，权威机构的信息之外，还包含了证书内容的签名(先通过Hash函数计算得到证书数字摘要，然后用权威机构私钥加密数字摘要得到数字签名)，签名计算方法以及证书对应的域名。这样一来，客户端收到证书之后：

* 使用权威机构的公钥解密数字证书，得到证书内容（服务器的公钥）以及证书的数字签名，然后根据证书上描述的计算证书签名的方法计算一下当前证书的签名，与收到的签名作对比，如果一样，表示证书一定是服务器下发的，没有被中间人篡改过。因为中间人虽然有权威机构的公钥，能够解析证书内容并篡改，但是篡改完成之后中间人需要将证书重新加密，但是中间人没有权威机构的私钥，无法加密，强行加密只会导致客户端无法解密，如果中间人强行乱修改证书，就会导致证书内容和证书签名不匹配。**所以证书签名就能判断证书是否被篡改**
* 再考虑证书被掉包的情况：中间人同样可以向权威机构申请一份证书，然后在服务器给客户端下发证书的时候劫持原证书，将自己的假证书下发给客户端，客户端收到之后依然能够使用权威机构的公钥解密证书，并且证书签名也没问题。但是这个时候客户端还需要检查证书中的域名和当前访问的域名是否一致。如果不一致，会发出警告！

从上面的分析可以看到，数字证书中的信息确实能让客户端辨别证书的真伪

**3、一个 Lock 对象和一个 synchronized 代码块之间的主要不同点是：**

Lock操作更精细

synchronized 代码块不能够保证进入访问等待的线程的先后顺序。

你不能够传递任何参数给一个 synchronized 代码块的入口。因此，对于 synchronized 代码块的访问等待设置超时时间是不可能的事情。

synchronized 块必须被完整地包含在单个方法里。而一个 Lock 对象可以把它的 lock() 和 unlock() 方法的调用放在不同的方法里。

**4、CyclicBarrier、CountDownLatch、Phaser**

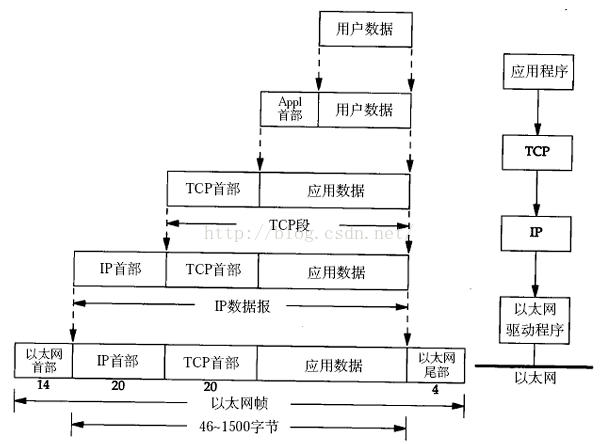
CyclicBarrier，允许一组线程互相等待，直到到达某个公共屏障点。它提供的await()可以实现让所有参与者在临界点到来之前一直处于等待状态。

CountDownLatch，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。它提供了await()、countDown()两个方法来进行操作。

在Phaser中，它把多个线程协作执行的任务划分为多个阶段，编程时需要明确各个阶段的任务，每个阶段都可以有任意个参与者，线程都可以随时注册并参与到某个阶段。

**5、TCP、UDP数据包大小的限制**

1、概述

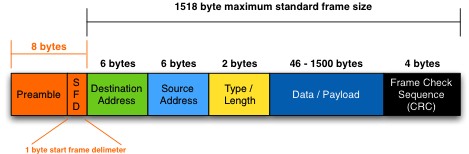
首先要看TCP/IP协议，涉及到四层：链路层，网络层，传输层，应用层。 　　  
其中以太网（Ethernet）的数据帧在链路层 　　  
IP包在网络层 　　  
TCP或UDP包在传输层 　　  
TCP或UDP中的数据（Data)在应用层 　　  
它们的关系是 数据帧｛IP包｛TCP或UDP包｛Data｝｝｝ 　　  
  
    不同的协议层对数据包有不同的称谓，在传输层叫做段(segment)，在网络层叫做数据报(datagram)，在链路层叫做帧(frame)。数据封装成帧后发到传输介质上，到达目的主机后每层协议再剥掉相应的首部，最后将应用层数据交给应用程序处理。

在应用程序中我们用到的Data的长度最大是多少，直接取决于底层的限制。 　　  
我们从下到上分析一下： 　　  
1.在链路层，由以太网的物理特性决定了数据帧的长度为(46＋18)－(1500＋18)，其中的18是数据帧的头和尾，也就是说数据帧的内容最大为1500(不包括帧头和帧尾)，即MTU(Maximum Transmission Unit)为1500； 　  
2.在网络层，因为IP包的首部要占用20字节，所以这的MTU为1500－20＝1480；　  
3.在传输层，对于UDP包的首部要占用8字节，所以这的MTU为1480－8＝1472； 　　  
所以，在应用层，你的Data最大长度为1472。当我们的UDP包中的数据多于MTU(1472)时，发送方的IP层需要分片fragmentation进行传输，而在接收方IP层则需要进行数据报重组，由于UDP是不可靠的传输协议，如果分片丢失导致重组失败，将导致UDP数据包被丢弃。 　　  
从上面的分析来看，在普通的局域网环境下，UDP的数据最大为1472字节最好(避免分片重组)。 　　  
但在网络编程中，Internet中的路由器可能有设置成不同的值(小于默认值)，Internet上的标准MTU值为576，所以Internet的UDP编程时数据长度最好在576－20－8＝548字节以内。

2、TCP、UDP数据包最大值的确定

        UDP和TCP协议利用端口号实现多项应用同时发送和接收数据。数据通过源端口发送出去，通过目标端口接收。有的网络应用只能使用预留或注册的静态端口；而另外一些网络应用则可以使用未被注册的动态端口。因为UDP和TCP报头使用两个字节存放端口号，所以端口号的有效范围是从0到65535。动态端口的范围是从1024到65535。

        MTU最大传输单元，这个最大传输单元实际上和链路层协议有着密切的关系，EthernetII帧的结构DMAC+SMAC+Type+Data+CRC由于以太网传输电气方面的限制，每个以太网帧都有最小的大小64Bytes最大不能超过1518Bytes，对于小于或者大于这个限制的以太网帧我们都可以视之为错误的数据帧，一般的以太网转发设备会丢弃这些数据帧。



        由于以太网EthernetII最大的数据帧是1518Bytes这样，刨去以太网帧的帧头（DMAC目的MAC地址48bits=6Bytes+SMAC源MAC地址48bits=6Bytes+Type域2Bytes）14Bytes和帧尾CRC校验部分4Bytes那么剩下承载上层协议的地方也就是Data域最大就只能有1500Bytes这个值我们就把它称之为MTU。

UDP 包的大小就应该是 1500 - IP头(20) - UDP头(8) = 1472(Bytes)  
TCP 包的大小就应该是 1500 - IP头(20) - TCP头(20) = 1460 (Bytes)

注\*PPPoE所谓PPPoE就是在以太网上面跑“PPP”。随着宽带接入（这种宽带接入一般为Cable Modem或者xDSL或者以太网的接入），因为以太网缺乏认证计费机制而传统运营商是通过PPP协议来对拨号等接入服务进行认证计费的，所以引入PPPoE。PPPoE导致MTU变小了以太网的MTU是1500，再减去PPP的包头包尾的开销（8Bytes），就变成1492。不过目前大多数的路由设备的MTU都为1500。

        如果我们定义的TCP和UDP包没有超过范围，那么我们的包在IP层就不用分包了，这样传输过程中就避免了在IP层组包发生的错误；如果超过范围，既IP数据报大于1500字节，发送方IP层就需要将数据包分成若干片，而接收方IP层就需要进行数据报的重组。更严重的是，如果使用UDP协议，当IP层组包发生错误，那么包就会被丢弃。接收方无法重组数据报，将导致丢弃整个IP数据报。UDP不保证可靠传输；但是TCP发生组包错误时，该包会被重传，保证可靠传输。

        UDP数据报的长度是指包括报头和数据部分在内的总字节数，其中报头长度固定，数据部分可变。数据报的最大长度根据操作环境的不同而各异。从理论上说，包含报头在内的数据报的最大长度为65535字节(64K)。

       我们在用Socket编程时，UDP协议要求包小于64K。TCP没有限定，TCP包头中就没有“包长度”字段，而完全依靠IP层去处理分帧。这就是为什么TCP常常被称作一种“流协议”的原因，开发者在使用TCP服务的时候，不必去关心数据包的大小，只需讲SOCKET看作一条数据流的入口，往里面放数据就是了，TCP协议本身会进行拥塞/流量控制。

       不过鉴于Internet(非局域网)上的标准MTU值为576字节，所以建议在进行Internet的UDP编程时，最好将UDP的数据长度控制在548字节 (576-8-20)以内。

3、TCP、UDP数据包最小值的确定

     在用UDP局域网通信时，经常发生“Hello World”来进行测试，但是“Hello World”并不满足最小有效数据(64-46)的要求，为什么小于18个字节，对方仍然可用收到呢？因为在链路层的MAC子层中会进行数据补齐，不足18个字节的用0补齐。但当服务器在公网，客户端在内网，发生小于18个字节的数据，就会出现接收端收不到数据的情况。

       以太网EthernetII规定，以太网帧数据域部分最小为46字节，也就是以太网帧最小是6＋6＋2＋46＋4＝64。除去4个字节的FCS，因此，抓包时就是60字节。当数据字段的长度小于46字节时，MAC子层就会在数据字段的后面填充以满足数据帧长不小于64字节。由于填充数据是由MAC子层负责，也就是设备驱动程序。不同的抓包程序和设备驱动程序所处的优先层次可能不同，抓包程序的优先级可能比设备驱动程序更高，也就是说，我们的抓包程序可能在设备驱动程序还没有填充不到64字节的帧的时候，抓包程序已经捕获了数据。因此不同的抓包工具抓到的数据帧的大小可能不同。下列是本人分别用wireshark和sniffer抓包的结果，对于TCP 的ACK确认帧的大小一个是54字节，一个是60字节，wireshark抓取时没有填充数据段，sniffer抓取时有填充数据段。

4、实际应用

        用UDP协议发送时，用sendto函数最大能发送数据的长度为：65535- IP头(20) - UDP头(8)＝65507字节。用sendto函数发送数据时，如果发送数据长度大于该值，则函数会返回错误。

        用TCP协议发送时，由于TCP是数据流协议，因此不存在包大小的限制（暂不考虑缓冲区的大小），这是指在用send函数时，数据长度参数不受限制。而实际上，所指定的这段数据并不一定会一次性发送出去，如果这段数据比较长，会被分段发送，如果比较短，可能会等待和下一次数据一起发送。

1、数据链路层对数据帧的长度都有一个限制，也就是链路层所能承受的最大数据长度，这个值  
称为最大传输单元，即MTU。以以太网为例，这个值通常是1500字节。  
2、对于IP数据包来讲，也有一个长度，在IP包头中，以16位来描述IP包的长度，也就是说，  
一个IP包，最长可能是65535字节。  
  
3、结合以上两个概念，第一个重要的结论就出来了，如果IP包的大小，起过了MTU值，那么就需要  
分片，也就是把一个IP包分为多个，这个概念非常容易理解，一个载重5T的卡车，要拉10T的货，它  
当然就得分几次来拉了。  
4、IP分片是很多资料常讲的内容，但是我倒是觉得分不分片其实不重要，重要的是另一个东西。一个  
数据包穿过一个大的网络，它其间会穿过多个网络，每个网络的MTU值是不同的。我们可以设想，如果  
接受/发送端都是以太网，它们的MTU都是1500，我们假设发送的时候，数据包会以1500来封装，然而，  
不幸的是，传输中有一段X.25网，它的MTU是576，这会发生什么呢？我想，这个才是我们所关心的。  
当然，结论是显而易见的，这个数据包会被再次分片，咱开始用火车拉，到了半路，不通火车，只通汽车，  
那一车货会被分为很多车……仅此而已，更重要的是，这种情况下，如果IP包被设置了“不允许分片标志”，那  
会发生些什么呢？对，数据包将被丢弃，然事收到一份ICMP不可达差错，告诉你，需要分片！  
这个网络中最小的MTU值，被称为路径MTU，我们应该有一种有效的手段，来发现这个值，最笨的方法或许是先  
用traceroute查看所有节点，然后一个个ping……  
  
5、到了传输层，也会有一个最大值的限制，当然，对于只管发，其它都不管的UDP来说，不再我们讨论之列。这里  
说的是TCP协议。说到大小，或许会让人想到TCP著名的滑动窗口的窗口大小，它跟收发两端的缓存有关，这里讨论  
的是传输的最大数据包大小，所以，它也不在讨论之列。  
TCP的选项字段中，有一个最大报文段长度（MSS），表示了TCP传往另一端的最大数据的长度，当一个连接立时，连接  
的双方都要通告各自的MSS，也就是说，它说是与TCP的SYN标志在一起的。当然，对于传输来讲，总是希望MSS越大越好，  
现在超载这么严重，谁家不希望多拉点货……但是，MSS总是有个限制的，也就是MTU-IP头长度-TCP头长度，对于以太网来讲  
它通常是1500-20-20=1460，虽然总是希望它能很大(如1460)，但是大多数BSD实现，它都是512的倍数，如1024…  
6、回到分片上来，例如，在Win2000下执行如下命令：  
ping 192.168.0.1 -l 1473  
  
按刚才的说法，1473+20(ip头)+8(icmp头)=1501，则好大于1500，它会被分片，但是，我们关心的是：  
这个数据包会被怎么样分法？  
可以猜想，第一个包是  
以太头+IP头+ICMP头+1427的数据；  
那第二个分片包呢？  
它可以是：  
以太头+IP头+ICMP头+1个字节的数据  
或者是：  
以太头+IP头+1个字节的数据  
也就是省去ICMP头的封装，当然，IP头是不可以省的，否则怎么传输了……  
事实上，TCP/IP协议采用的是后一种封装方式，这样，一次可以节约8个字节的空间。IP包头中，用了三个标志来描述一个分片包：  
1、分片标志：如果一个包被分片了，被置于1，最后一个分片除外；——这样，对于接收端来讲，可以根据这个标志位做为重组的重  
要依据之一；  
2、分片偏移标志：光有一个标志位说明“自己是不是分片包”是不够的，偏移标志位说明了自己这个分片拉于原始数据报的什么位置，  
很明显，这两个标志一结合，就很容易重组分片包了。  
3、不允许分片标志：如果数据包强行设置了这个标志，那么在应该分片的时候，…… err，刚才已经说过了  
  
在以太网中，数据包的大小范围是在64—1518字节之间，如果除去头部开销，则实际的数据大小为46—1500字节之间。 一般情况下，数据包的大小都是在这个范围内，如果数据包小于64字节，称为碎片；而如果大于1518字节，称为特大数据包。这两种类型的数据包都是非正常的以太网数据包，它们将影响网络的正常运行。无论是碎片或特大数据包，都会增加网络的负载，导致网络故障的发生。所以，我们在对网络进行分析的时候，对数据包大小的判断也是不可缺少的一个环节。