计算机网络常见面试总结

应用	应用层协议	端口号	传输层协议	备注
域名解析	DNS	53	UDP/TCP	域名长度超过512字节用TCP协议
动态主机配置协议	DHCP	67/68	UDP	
简单网络管理协议	SNMP	161/162	UDP	
文件传输协议	FTP	20/21	TCP	控制连接21，数据连接20
远程终端协议	TELNET	23	TCP	
超文本传输协议	HTTP	80	TCP	
简单邮件传送协议	SMTP	25	TCP	
邮件读取协议	POP3	110	TCP	
网际报文存取协议	IMAP	143	TCP	

TCP粘包、拆包及解决办法?

TCP、UDP是传输层的协议，TCP会发生粘包，UDP会发生粘包吗?

UDP不会发生粘包现象。UDP是基于报文发送的，从UDP的帧结构可以看出，在UDP首部采用了16bit来只是UDP数据报文的长度，因此在应用层能够很好地将不同的数据报文区分开，从而避免粘包和拆包的问题。

TCP是基于字节流的瑞然应用层和TCP传输层之间的数据交互是大小不等的数据块，但是TCP把这些数据块仅仅看成一连串无结构的字节流，没有边界；另外从TCP的帧结构也可以看出，在TCP的首部没有表示数据长度的字段，基于上面两点，在使用TCP传输数据时，才有粘包或则拆包现象发生的可能。

TCP报文头格式

TCP Header

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port														
4	32	Sequence number																														
8	64	Acknowledgment number (if ACK set)																														
12	96	Data offset				Reserved 0 0 0				N S	C W R	E C E	U R G	A C K	P S H	R S T	S Y N	F I N	Window Size														
16	128	Checksum																Urgent pointer (if URG set)														
20	160	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.)																														
...	...	...																														

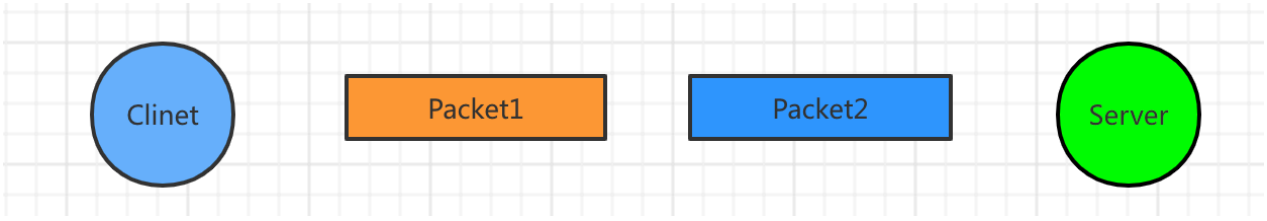
UDP报文头格式

UDP Header

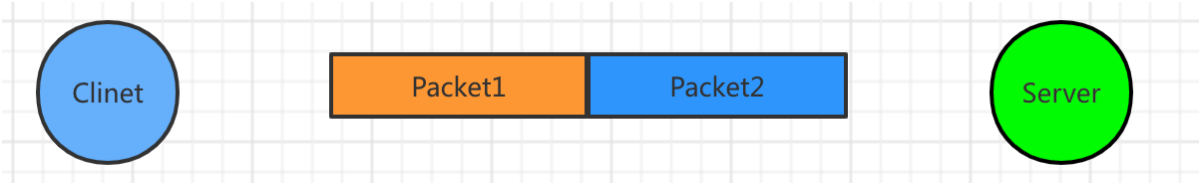
Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port														
4	32	Length																Checksum														

什么是粘包、拆包？为什么会发生TCP粘包、拆包？

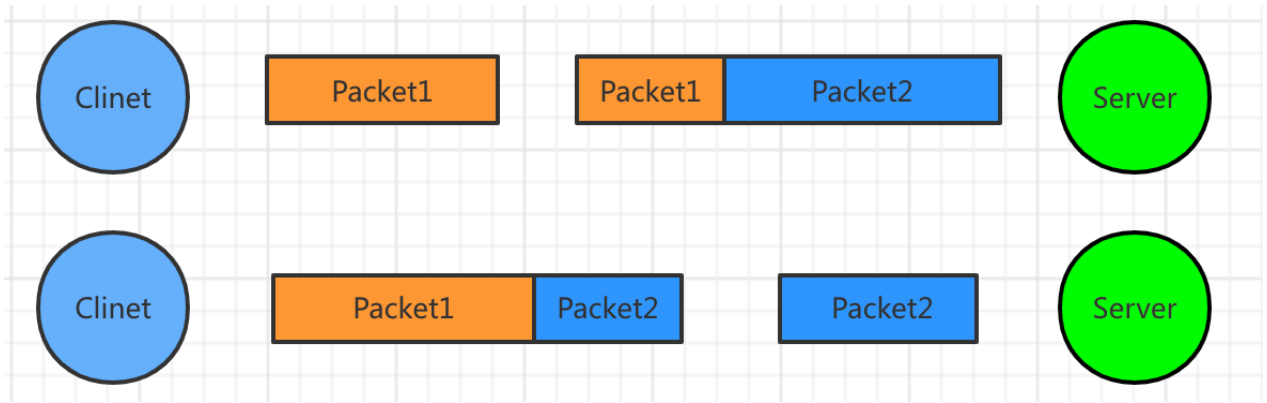
1. 正常情况



2. 只接收到一个数据包。这一个数据包中包含了发送端发送的连个数据包的信息，这种现象即为粘包。这种情况由于接收端不知道这两个数据包的界限，所以对于接收端来说很难处理。



3. 接收到两个数据包，但是连个数据包要么不完整，要么多了一块。这种情况就是发生了粘包和拆包。这种情况若不处理，对于接收端来说同样是不嗨哦处理的。



拆包

1. 应用程序写入的数据大于TCP发送缓冲区剩余缓空间大小，这将会发生拆包；
2. 待发送的数据大于MSS（最大报文长度），当TCP报文长度-TCP头部长度>MSS的时候将发生拆包

粘包

1. 要发送的数据小于TCP发送缓冲区的大小，TCP将多次写入缓冲区的数据一次发送出去，将会发生粘包；
2. 接受数据段的应用层没有及时读取接受缓存区中的数据，将发生粘包。

粘包、拆包解决办法

通过以上分析清楚了粘包和拆包产生的原因，如何解决这些问题？解决问题的关键在于如何给没饿过数据包添加边界信息，常用的方法有如下几个?

1. 发送端给每个数据包添加包首部，首部中应该至少包含数据包的长度，这样接收端在接收到数据后，通过读取包首部的长度字段，便知道每一个数据包的实际长度了；
2. 发送端将每个数据包缝状为固定长度（不够的可以通过补0填充），这样接收端每次从接受缓冲区中读取固定长度的数据就自然而然的把每个数据包拆分开来；
3. 可以在数据包之间设置边界，如添加特殊符号，这样，接收端通过这个边界就可以将不同的数据包拆分开。

等等

从输入URL到页面加载发生了什么?

总体来说分为以下几个过程:

1. DNS解析；
2. TCP连接；
3. 发送HTTP请求；
4. 服务器处理请求并返回HTTP报文；
5. 浏览器解析渲染页面
6. 连接结束

浏览器简历HTTP连接的过程

1. HTTP协议建立在TCP协议之上，HTTP请求前，需先进行TCP连接，形成客户端到服务器的稳定通道。俗称TCP的三次握手；
2. TCP连接完成后，HTTP请求开始，请求有多种方式，常见的有GET，POST等；
3. HTTP请求包含请求头，也可能包含请求体两部分，请求头重宝函我们希望对请求文件的操作的信息，请求体中宝函传递给后台的参数；
4. 服务器收到HTTP请求后，后台开始工作，如负载平衡，跨域等，这里就是后端的工作了；
5. 文件处理完毕，生成相应数据包，响应也宝函两部分，响应头和响应体，响应体就是我们所请求的文件；
6. 经过网络传输，文件被下载到本地客户端。

HTML页面渲染与解析

- 客户端浏览器加载了HTML文件后，由上到下解析HTML为DOM树（DOM Tree）；
- 遇到css文件，css中的URL发起HTTP请求；
- 这是第二次HTTP请求，由于HTTP1.1协议增加了Connection:keep-alive声明，故TCP连接不会关闭。可以复用；
- HTTP连接时无状态连接，客户端与服务器端需要重新发起请求--响应。在请求css的过程中，解析器继续解析HTML，然后到了script标签；
- 由于script可能会改变DOM结构，故解析器停滞生成DOM树，解析器被js阻塞，等待js文件发起HTTP请求，然后加载。这就是第三次HTTP请求。js执行完成后解析器继续解析；
- 由于css文件可能会影响js文件的执行结果，因此需要等css文件加载完成后再执行；
- 浏览器收到css文件后，开始解析css文件为CSSOM树（css Rule Tree）；
- CSSDOM树生成后，DOM Tree与css Rule Tree结合生成渲染树（Render Tree）；
- Render Tree会被CSS文件阻塞，渲染树生成后，先布局，绘制渲染树中结点的属性（位置、高度、大小等），然后渲染，页面就会呈现信息；
- 继续边解析边渲染，遇到了林一个js文件，js文件执行后改变了DOM树，渲染树从被高变的dom开始再次渲染；
- 继续向下渲染，碰到一个img标签，浏览器发起HTTP请求，不会等待img加载完成，继续向下渲染，之后再重新渲染此部分；
- DOM树遇到HTML结束标签，停滞解析，进而渲染结束。

GET/POST，幂等?

GET用于获取资源，而POST用于传输数据

GET方法都是幂等的，但POST方法不是。幂等就是说，同样的请求被执行一次与连续执行多次的效果是一样的，服务器的状态也是一样。所以，幂等方法不应该具有副作用。

TCP协议如何保证可靠传输

1. 应用数据被分割成TCP认为最合适发送的数据块；
2. TCP给发送的每一个包进行编号，接受方对数据包进行排序，把有序数据传送给应用层；
3. **校验和**：TCP将保持它首部和数据的校验和。这是一个端到端的校验和，目的是检测数据再传输过程中的任何变化。如果收到段的校验和有差错，TCP将丢弃这个报文段和不确认收到此报文段；
4. TCP的接收端会丢弃重复的数据；
5. **流量控制**：TCP连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接受方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP使用的流量控制协议是可变大小的**滑动窗口协议**。（TCP利用滑动窗口实现流量控制）；
6. **拥塞控制**：当网络拥塞时，减少数据的发送；
7. **ARQ协议**：也是为了实现可靠传输的，它的基本原理就是每发送完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组；
8. **超时重传**：当TCP发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段；

ARQ协议时什么?

自动重传请求（Automatic Repeat-reQuest, ARQ）是OSI模型中数据链路层和传输层的错误纠正协议之一。它通过使用确认和超市这两个机制，在不可靠服务的基础上实现可靠的信息传输。如果发送方在发送后一段时间之内没有收到确认帧，它通常会重新发送。ARQ包括停止等待ARQ协议和连续ARQ协议。

停止等待ARQ协议

- 停止等待协议是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认（回复ACK）。如果过了一段时间（超时时间后），还是没有收到ACK确认，说明没有发送成功，需要重新发送，直到收到确认后再发下一个分组；
- 在停止等待协议中，若接受方收到重复分组，就丢弃该分组，但同时还要发送确认；

优点：简单

缺点：信道利用率低，等待时间长

连续ARQ协议

连续ARQ协议可提高信道利用率。发送方维持一个发送窗口，凡位于发送窗口内的分组可以连续发送出去，而不需要等待对方确认。接收方一般采用累计确认，对按序到达的最后一个分组发送确认，表明到这个分组为止的所有分组都已经正确收到了；

优点：信道利用率高，容易实现，即使确认丢失，也不必重传。

缺点：不能向发送方放映出接收方已经正确收到的所有分组的信息。比如：发送方发送了5条消息，中建第三条丢失（3号），这时接受方对前两个发送确认。发送方无法知道后3个分组的下落，而只好把后3个分组全部重传一遍。这也叫Go-Back-N（回退N），表示需要退回来重传已经发送过的N个消息。

滑动窗口和流量控制

TCP利用滑动窗口实现流量控制。流量控制是为了控制发送方发送速率，保证接受方来得及接收。接收方发送的确认报文中的窗口字段可以用来孔子发送窗口大小，从而影响发送方的发送速率。将窗口字段设置为0，则发送方不能发送数据。

拥塞控制

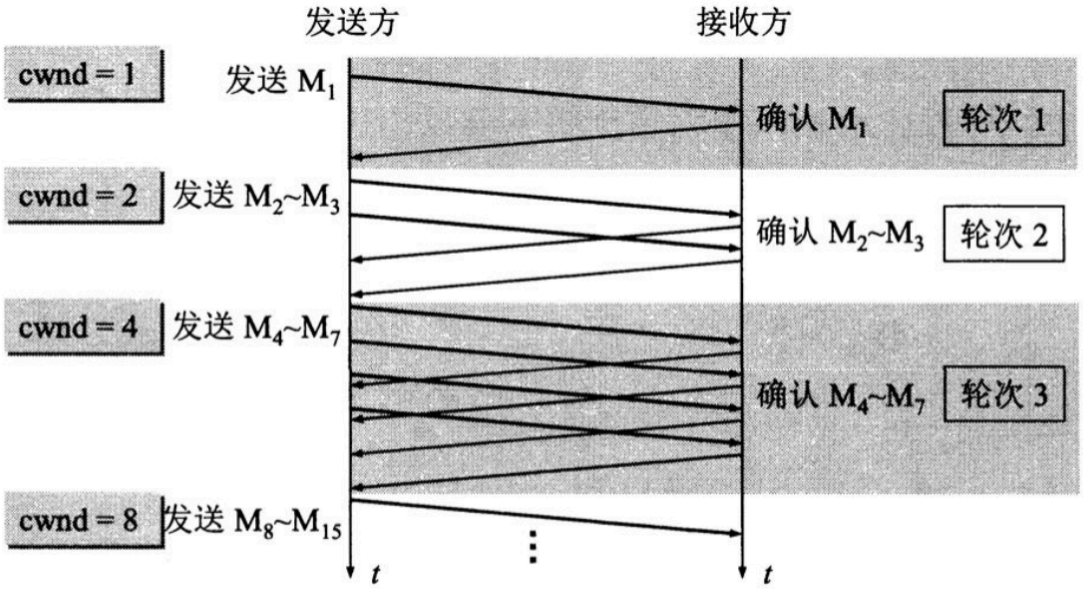
在某段时间，若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏。这种情况就叫拥塞。**拥塞控制就是为了防止过多的数据注入到网络中**，这样就可以使网络中的路由器或链路不致过载。拥塞控制所要做的都有一个前提，就是网络能够成熟现有的网络负荷。拥塞控制是一个全局性的过程，涉及到所有的主机，所有的路由器，以及与降低网络传输性能有关的所有因素。相反，流量控制往往是点对点通信量的控制，是个端到端的问题。流量控制所要做的就是抑制发送端发送数据的速率，以便使接收端来得及接收。

为了进行拥塞控制，TCP发送方要维持一个**拥塞窗口(cwnd, congestion window)**的状态变量。拥塞控制窗口的大小取决于网络的拥塞程度，并且动态变化。发送方让自己的发送窗口取为拥塞窗口和接收方的接收窗口中较小的一个。

TCP的拥塞控制采用四种算法：即**慢开始(slow start)**、**拥塞避免(Congestion Avoidance)**、**快重传(fast retransmit)**和**快恢复(fast recovery)**。在网络层也可以使路由器采用适当的分组丢弃策略（如主动队列管理AQM），以减少网络拥塞的发生。

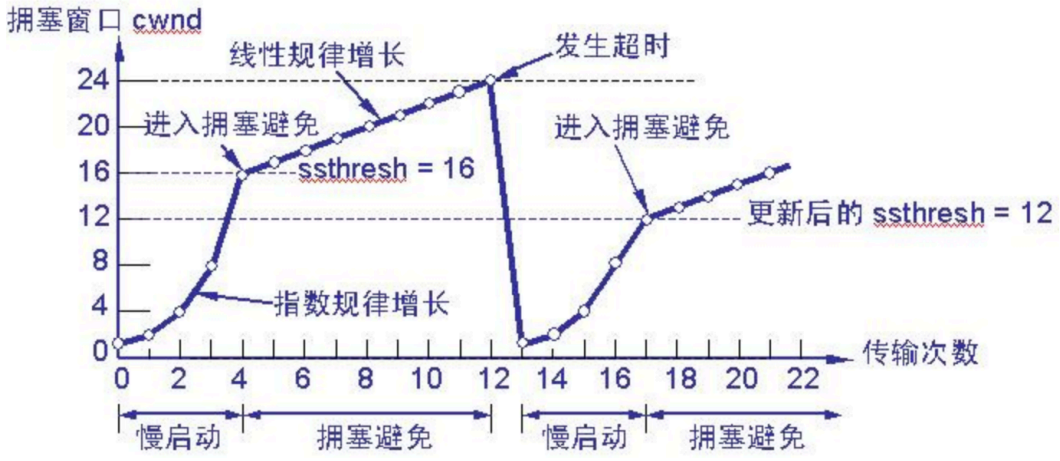
> 为了防止cwnd增长过大引起网络拥塞，还需要设置一个慢开始门限ssthresh状态变量。ssthresh的用法如下：
>
> 当cwnd < ssthresh时，使用慢开始算法；
>
> 当cwnd > ssthresh时，改用拥塞避免算法；
>
> 当cwnd = ssthresh时，慢开始与拥塞避免算法任意；

- **慢开始**：慢开始算法的思路是当主机开始发送数据时，如果立即把大量数据字节注入到网络，那么可能会引起网络拥塞，因为现在还不知道网络的实际情况。经验表明，较好的方法是先试探一下，即由小到大逐渐增大发送窗口，也就是由小到大逐渐增大拥塞窗口数值。cwnd初始值为1，每经过一个传播轮次，cwnd加倍。



- **拥塞避免**：拥塞避免算法的思路是让拥塞窗口cwnd缓慢增大，即每经过一个往返时间RTT就把发送方的cwnd加1.

  无论是在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞，就把**慢开始门限**设置为出现拥塞时的发送窗口大小的一半。然后把拥塞窗口设置为1，执行慢开始算法。



乘法减小和加法减小增大

乘法减小：是指不论在慢开始阶段还是拥塞避免阶段，只要出现超时，就把慢开始门限减半，即设置为当前拥塞窗口的一半（与此同时，执行慢开始算法）。当网络出现频繁拥塞时，ssthresh值就下降得很快，以大大减小注入到网络中的分组数；

加法增大：是指执行拥塞避免算法后使拥塞窗口缓慢增大，以防止网络过早出现拥塞。

- **快重传**：
  在TCP/IP中，快速重传和恢复（fast retransmit and recovery, FRR）是一种拥塞控制算法，它能快速恢复丢失的数据包。
  快速重传(Fast retransmit)要求接收方在收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方），而不要等到自己发送数据时捎带确认。
  快重传算法规定，发送方之遥以连收到3个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计数器时间到期；
- **快恢复（Fast Recovery）**

1. 当发送方连续发送三个重复确认，就执行“乘法减小“算法，把慢开始门限ssthresh减半。这是为了预防网络发生拥塞。请注意：接下去不执行慢开始算法；
2. 由于发送方现在认为网络很可能没有发生拥塞，因此与**慢开始不同**之处是现在不执行慢开始算法（即拥塞窗口cwnd现在不设置为1），而是把cwnd值设置为慢开始门限ssthresh减半后的数值，然后开始执行拥塞避免算法（“加法增大”），使拥塞窗口缓慢地先行增大。

发送方窗口的上线值 = Min(rwnd, cwnd)

当rwnd < cwnd 时，是接收方的接收能力限制发送方窗口的最大值；

当cwnd < rwnd 时，则是网络的拥塞限制发送方窗口的最大值

301和302状态吗有什么区别?

- 301是永久性重定向（Permanently Moved），表示一个旧的网址所代表的资源已经被永久地移除了，不能再访问了，并且搜索引擎在获取新的资源的同时也将旧的网址转换为重定向之后的地址；
- 302是临时重定向（Temporarily Moved），这个重定向只是临时从一个旧的地址跳转到一个新的地址，旧的地址的资源还在，还可以继续访问，搜索引擎获取资源并保存旧的地址；

HTTP长连接，短连接是什么?

在HTTP/1.0中默认使用短连接。也就是说，客户端和服务器没进行一次HTTP操作，就简历一次连接，任务结束就中断连接。当客户端浏览器访问的某个HTML或其他类型的Web页中包含有其他的Web资源（如JavaScript文件、图像文件、css文件等），每遇到这样一个Web资源，浏览器就会重新建立一个HTTP会话。

从HTTP/1.1开始，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头加入这行代码：

> Connection: keep-alive

在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。实现长连接需要客户端和服务端支持长连接。

HTTP协议的长连接和短连接，实质上是TCP连接的长连接和短连接。

HTTP是不包吃状态的协议，如何保存用户状态?

HTTP是一种不保存状态的协议，即无状态（stateless）协议，也就是说HTTP协议本身不对请求和响应之间的通信状体进行保存。那么我们如何保存用户状态呢?

Session机制的存在就是为了解决这个问题，Session的主要作用就是通过服务端记录用户的状态。

典型的场景是购物车，当你要添加商品到购物车的时候，系统不知道是哪个用户操作的，因为HTTP协议是无状态的。**服务端给特定的用户创建特定的Session之后就可以标识这个用户并且跟踪这个用户了**（一般情况下，服务器会在一定时间内保存这个Session，过了时间限制，就会笑会这个Session）

在服务端保存Session的方法很多，最常见的就是内存和数据库（比如使用内存数据库redis保存）。既然Session存放在服务端，那么我们如何实现Session的跟踪呢？大部分情况下，我们都是通过在Cookie中附加一个Session ID的方式来跟踪。

Cookie被禁用怎么办?

最常用的就是利用URL重写把Session ID直接付家在URL路径的后面。

Cookie的作用是什么？和Session有什么区别?

Cookie和Session都是用来跟踪浏览器用户身份的会话方式，但是两者的应用场景不太一样。

Cookie一般用来保存用户信息

比如：

1. 在Cookie中保存已经登录过的用户信息，下次访问网站的时候页面可以自动帮你把一些登录的基本信息给填了；
2. 一般的网站都会有保持登录也就是你再次访问网站的时候就不需要重新登录了，**这是因为用户登录的时候我们可以存放一个Token在Cookie中**，下次登录的时候只需要根据Token值来查找用户即可（为了安全考虑，重新登录一般都要将Token重写）；
3. 登录一次网站后访问网站其他页面不需要重新登录。**Session的主要作用就是通过服务端记录用户的状态**。

典型的场景是购物车，当你要添加商品到购物车时，系统不知道是哪个用户操作的，因为HTTP协议是无状态的。服务端给特定的用户创建特定的Session之后就可以表示这个用户并且跟踪这个用户了。

Cookie数据保存在客户端（浏览器端），Session数据保存在服务端。

Cookie存储在客户端中，而Session存储在服务器中，相对勒说Session安全性更高。一些敏感信息不要写入Cookie中，最好能将Cookie信息加密然后使用到的时候再去服务器端解密。

HTTP1.0和HTTP1.1的主要区别是什么?

HTTP1.0最早在网页中使用是在1996年，那个时候只是使用一些较为简单的网页上的网络请求，而HTTP1.1则是在1999年才开始广泛应用于现在的各大浏览器网络请求中，同时HTTP1.1也是当前使用最为广泛的HTTP协议。主要区别体现在：

1. **长连接：在HTTP/1.0中，默认使用的是短连接**，也就是说每次请求都要建立一次连接。HTTP是基于TCP/IP协议的，每一次建立或则断开连接都需要三次握手和四次回首的开销，如果每次请求都要这样的话，开销会比较大。因此最好能维持一个长连接，可以用这个长连接来发多个请求。**HTTP1.1起，默认使用长连接**。流水线方式使客户在收到HTTP响应报文之前就能接着发送新的请求报文。与之相对应的非流水线方式是客户端在收到一个响应后才能发送下一个请求。
2. **错误状态响应码**：在HTTP1.1中新增了24个错误状态响应码，如409（Conflict）表示请求的资源与资源当前的状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。
3. **缓存处理**：在HTTP1.0中主要使用header里面的If-Modified-Since, Expires来作为缓存判断的标准，HTTP1.1则引入了更多的缓存控制策略例如Entry-tag, If-Unmodified-Since, If-Match, If-None-Match等等多可供选择的缓存头来控制缓存策略；
4. 贷款优化及网络连接的使用：HTTP1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1则在请求头引入了染革头域，它允许只请求资源的某个部分，即返回码是206（Partial Content），这样就方便了开发者自由的选择以便于充分利用贷款和连接

URI和URL的区别?

URI（Uniform Resource Identifier）统一资源定位符，可以为宜标识一个资源；

URL（Uniform Resource Location）统一资源定位符，可以提供该资源的路径。它是一种具体的URI，即URL可以用来标识一个资源，而且还指明了如何locate这个资源。（URI的范围更广泛一些，只要能定位到资源就可以，URL是有具体格式的，是一种具体的URI）

HTTP和HTTPS的区别?

- 端口：HTTP的URL由"HTTP://"起始且默认使用端口80，而HTTPS的URL由"HTTPS://"起始且默认使用端口443
- 安全性和资源消耗：

  HTTP协议运行在TCP之上，所有传输的内容都是明文，客户端和服务器端都无法校验对方的身份。

  HTTPS是运行在SSL/TLS之上的HTTP协议，SSL/TLS运行在TCP之上。所有传输的内容都要经过加密，机密采用对称加密，但对称加密的密钥用服务器方的正式进行了非对称加密。所以说，HTTP安全性没有HTTPS高，但是HTTPS比HTTP耗费更多的服务器资源。