GET和POST有什么区别

> 原文链接：https://www.zhihu.com/question/28586791

这个问题虽然看上去很初级，但实际上却涉及到方方面面，这也就是为啥面试里老爱问这个的原因之一。

HTTP最早被用来做浏览器与服务器之间交互HTML和表单的通讯协议；后来又被被广泛的扩充到接口格式的定义上。所以在讨论GET和POST区别的时候，需要现确定下到底是浏览器使用的GET/POST还是用HTTP作为接口传输协议的场景。

浏览器的GET和POST

这里特指浏览器中**非**Ajax的HTTP请求，即从HTML和浏览器诞生就一直使用的HTTP协议中的GET/POST。浏览器用GET请求来获取一个html页面/图片/css/js等资源；用POST来提交一个

表单，并得到一个结果的网页。

浏览器将GET和POST定义为：

GET

“读取”一个资源。比如Get到一个html文件。反复读取不应该对访问的数据有副作用。比如”GET一下，用户就下单了，返回订单已受理“，这是不可接受的。没有副作用被称为“幂等“（Idempotent)。

因为GET因为是读取，就可以对GET请求的数据做缓存。这个缓存可以做到浏览器本身上（彻底避免浏览器发请求），也可以做到代理上（如nginx），或者做到server端（用Etag，至少可以减少带宽消耗）

POST

在页面里 标签会定义一个表单。点击其中的submit元素会发出一个POST请求让服务器做一件事。这件事往往是有副作用的，不幂等的。

不幂等也就意味着不能随意多次执行。因此也就不能缓存。比如通过POST下一个单，服务器创建了新的订单，然后返回订单成功的界面。这个页面不能被缓存。试想一下，如果POST请求被浏览器缓存了，那么下单请求就可以不向服务器发请求，而直接返回本地缓存的“下单成功界面”，却又没有真的在服务器下单。那是一件多么滑稽的事情。

因为POST可能有副作用，所以浏览器实现为不能把POST请求保存为书签。想想，如果点一下书签就下一个单，是不是很恐怖？。

此外如果尝试重新执行POST请求，浏览器也会弹一个框提示下这个刷新可能会有副作用，询问要不要继续。

在chrome中尝试重新提交表单会弹框。

当然，服务器的开发者完全可以把GET实现为有副作用；把POST实现为没有副作用。只不过这和浏览器的预期不符。**把GET实现为有副作用是个很可怕的事情。** 我依稀记得很久之前百度贴吧有一个因为使用GET请求可以修改管理员的权限而造成的安全漏洞。反过来，把没有副作用的请求用POST实现，浏览器该弹框还是会弹框，对用户体验好处改善不大。

但是后边可以看到，将HTTP POST作为接口的形式使用时，就没有这种弹框了。于是把一个POST请求实现为幂等就有实际的意义。POST幂等能让很多业务的前后端交互更顺畅，以及避免一些因为前端bug，触控失误等带来的重复提交。将一个有副作用的操作实现为幂等必须得从业务上能定义出怎么就算是“重复”。如提交数据中增加一个dedupKey在一个交易会话中有效，或者利用提交的数据里可以天然当dedupKey的字段。这样万一用户强行重复提交，服务器端可以做一次防护。

GET和POST携带数据的格式也有区别。当浏览器发出一个GET请求时，就意味着要么是用户自己在浏览器的地址栏输入，要不就是点击了html里a标签的href中的url。所以其实并不是GET只能用url，而是浏览器直接发出的GET只能由一个url触发。所以没办法，GET上要在url之外带一些参数就只能依靠url上附带querystring。但是HTTP协议本身并没有这个限制。

浏览器的POST请求都来自表单提交。每次提交，表单的数据被浏览器用编码到HTTP请求的body里。浏览器发出的POST请求的body主要有有两种格式，一种是application/x-www-form-urlencoded用来传输简单的数据，大概就是"key1=value1&key2=value2"这样的格式。另外一种是传文件，会采用multipart/form-data格式。采用后者是因为application/x-www-form-urlencoded的编码方式对于文件这种二进制的数据非常低效。

浏览器在POST一个表单时，url上也可以带参数，只要里的url带querystring就行。只不过表单里面的那些用 等标签经过用户操作产生的数据都在会在body里。

因此我们一般会**泛泛的说**“GET请求没有body，只有url，请求数据放在url的querystring中；POST请求的数据在body中“。但这种情况仅限于浏览器发请求的场景。

接口中的GET和POST

这里是指通过浏览器的Ajax api，或者iOS/Android的App的http client，java的commons-httpclient/okhttp或者是curl，postman之类的工具发出来的GET和POST请求。此时GET/POST不光能用在前端和后端的交互中，还能用在后端各个子服务的调用中（即当一种RPC协议使用）。尽管RPC有很多协议，比如thrift，grpc，但是http本身已经有大量的现成的支持工具可以使用，并且对人类很友好，容易debug。HTTP协议在微服务中的使用是相当普遍的。

当用HTTP实现接口发送请求时，就没有浏览器中那么多限制了，只要是符合HTTP格式的就可以发。HTTP请求的格式，大概是这样的一个字符串（为了美观，我在\r\n后都换行一下）：

```
<METHOD> <URL> HTTP/1.1\r\n
<Header1>: <HeaderValue1>\r\n
<Header2>: <HeaderValue2>\r\n
...
<HeaderN>: <HeaderValueN>\r\n
\r\n
<Body Data....>
```

其中的“”可以是GET也可以是POST，或者其他的HTTP Method，如PUT、DELETE、OPTION……。从协议本身看，并没有什么限制说GET一定不能没有body，POST就一定不能把参放到的querystring上。因此其实可以更加自由的去利用格式。比如Elastic Search的_search api就用了带body的GET；也可以自己开发接口让POST一半的参数放在url的querystring里，另外一半放body里；你甚至还可以让所有的参数都放Header里——可以做各种各样的定制，只要请求的客户端和服务器端能够约定好。

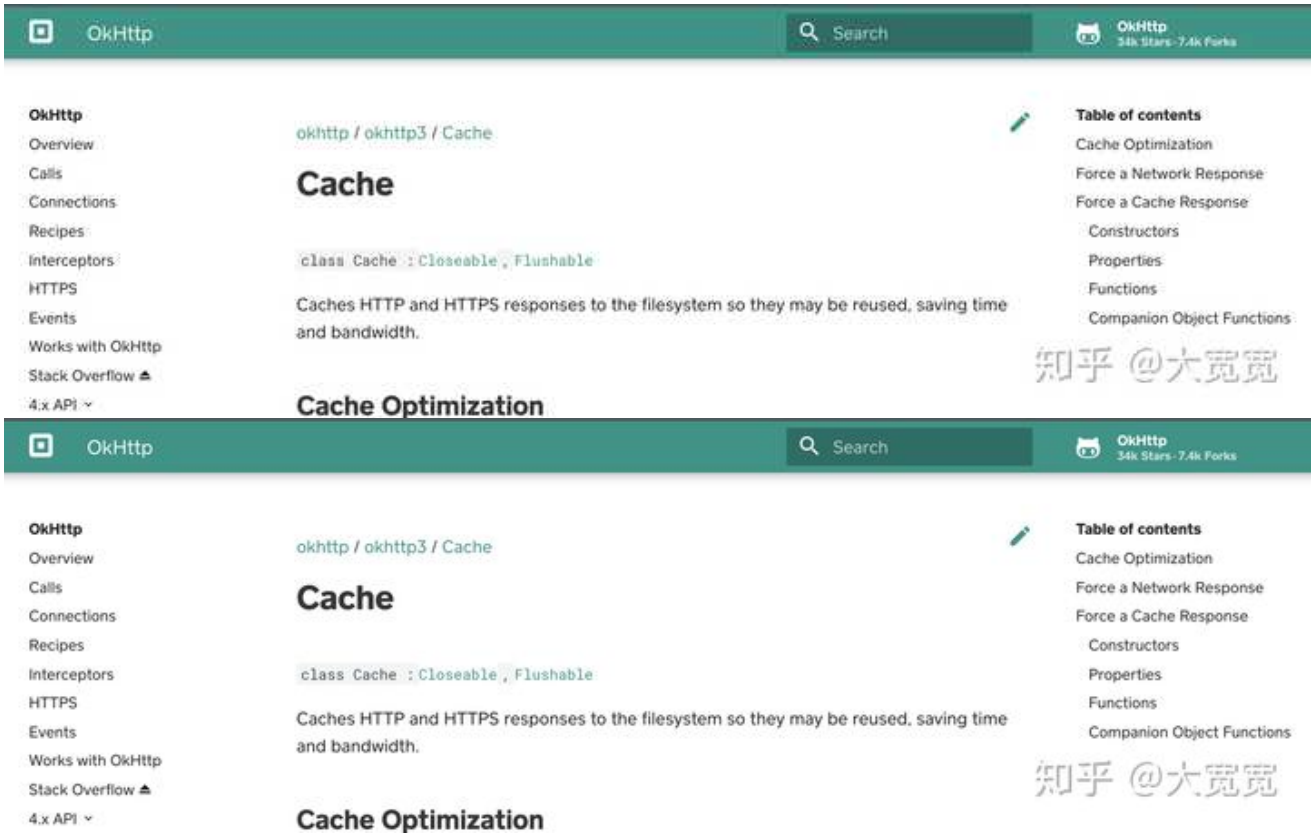
当然，太自由也带来了另一种麻烦，开发人员不得不每次讨论确定参数是放url的path里，querystring里，body里，header里这种问题，太低效了。于是就有了一些列接口规范/风格。其中名气最大的当属REST。REST充分运用GET、POST、PUT和DELETE，约定了这4个接口分别获取、创建、替换和删除“资源”，REST最佳实践还推荐在请求体使用json格式。这样仅仅通过看HTTP的method就可以明白接口是什么意思，并且解析格式也得到了统一。

> json相对于x-www-form-urlencoded的优势在于1）可以有嵌套结构；以及 2）可以支持更丰富的数据类型。通过一些框架，json可以直接被服务器代码映射为业务实体。用起来十分方便。但是如果是写一个接口支持上传文件，那么还是multipart/form-data格式更合适。

REST中GET和POST不是随便用的。在REST中，【GET】 + 【资源定位符】被专用于获取资源或者资源列表，比如：

```
GET http://foo.com/books           获取书籍列表
GET http://foo.com/books/:bookId   根据bookId获取一本具体的书
```

与浏览器的场景类似，REST GET也不应该有副作用，于是可以被反复无脑调用。浏览器（包括浏览器的Ajax请求）对于这种GET也可以实现缓存（如果服务器端提示了明确需要Caching）；但是如果用非浏览器，有没有缓存完全看客户端的实现了。当然，也可以从整个App角度，也可以完全绕开浏览器的缓存机制，实现一套业务定制的缓存框架。



okhttp中控制Cache的类

REST 【POST】+ 【资源定位符】则用于“创建一个资源”，比如：

```
POST http://foo.com/books
{
  "title": "大宽宽的碎碎念",
  "author": "大宽宽",
  ...
}
```

这里你就能留意到**浏览器中用来实现表单提交的POST，和REST里实现创建资源的POST**语义上的不同。

> 顺便讲下REST POST和REST PUT的区别。有些api是使用PUT作为创建资源的Method。PUT与POST的区别在于，PUT的实际语义是“replace”replace。REST规范里提到PUT的请求体应该是完整的资源，包括id在内。比如上面的创建一本书的api也可以定义为：

```
PUT http://foo.com/books
{
  "id": "BOOK:affe001bbe0556a",
  "title": "大宽宽的碎碎念",
  "author": "大宽宽",
  ...
}
```

> 服务器应该先根据请求提供的id进行查找，如果存在一个对应id的元素，就用请求中的数据**整体替换**已经存在的资源；如果没有，就用“把这个id对应的资源从【空】替换为【请求数据】“。直观看起来就是“创建”了。

> 与PUT相比，POST更像是一个“factory”，通过一组必要的数据创建出完整的资源。 至于到底用PUT还是POST创建资源，完全要看是不是提前可以知道资源所有的数据（尤其是id），以及是不是完整替换。比如对于AWS S3这样的对象存储服务，当想上传一个新资源时，其id就是“ObjectName”可以提前知道；同时这个api也总是完整的replace整个资源。这时的api用PUT的语义更合适；而对于那些id是服务器端自动生成的场景，POST更合适一些。
>
> 有点跑题，就此打住。

AWS S3 创建一个Object的API描述

回到接口这个主题，上面仅仅粗略介绍了REST的情况。但是现实中总是有REST的变体，也可能用非REST的协议（比如JSON-RPC、SOAP等），每种情况中的GET和POST又会有所不同。

关于安全性

我们常听到GET不如POST安全，因为POST用body传输数据，而GET用url传输，更加容易看到。但是从攻击的角度，无论是GET还是POST都不够安全，因为HTTP本身是**明文协议。每个HTTP请求和返回的每个byte都会在网络上明文传播，不管是url，header还是body**。这完全不是一个“是否容易在浏览器地址栏上看到”的问题。

为了避免传输中数据被窃取，**必须做从客户端到服务器的端端加密。业界的通行做法就是https**——即用SSL协议协商出的密钥加密明文的http数据。这个加密的协议和HTTP协议本身相互独立。如果是利用HTTP开发公网的站点/App，要保证安全，https是最最基本的要求。

> 当然，端端加密并不一定非得用https。比如国内金融领域都会用私有网络，也有GB的加密协议SM系列。但除了军队，金融等特殊机构之外，似乎并没有必要自己发明一套类似于ssl的协议。

回到HTTP本身，的确GET请求的参数更倾向于放在url上，因此有更多机会被泄漏。比如携带私密信息的url会展示在地址栏上，还可以分享给第三方，就非常不安全了。此外，从客户端到服务器端，有大量的中间节点，包括网关，代理等。他们的access log通常会输出完整的url，比如nginx的默认access log就是如此。如果url上携带敏感数据，就会被记录下来。但请注意，**就算私密数据在body里，也是可以被记录下来的**，因此如果请求要经过不信任的公网，避免泄密的**唯一手段就是https**。这里说的“避免access log泄漏”仅仅是指避免可信区域中的http代理的默认行为带来的安全隐患。比如你是不太希望让自己公司的运维同学从公司主网关的log里看到用户的密码吧。

另外，上面讲过，如果是用作接口，GET实际上也可以带body，POST也可以在url上携带数据。所以实际上到底怎么传输私密数据，要看具体场景具体分析。当然，绝大多数场景，用POST + body里写私密数据是合理的选择。一个典型的例子就是“登录”：

```
POST http://foo.com/user/login
{
  "username": "dakuankuan",
  "passowrd": "12345678"
}
```

安全是一个巨大的主题，有由很多细节组成的一个完备体系，比如返回私密数据的mask，XSS，CSRF，跨域安全，前端加密，钓鱼，salt，..... POST和GET在安全这件事上仅仅是个小角色。因此单独讨论POST和GET本身哪个更安全意义并不是太大。只要记得一般情况下，私密数据传输用POST + body就好。

关于编码

常见的说法有，比如GET的参数只能支持ASCII，而POST能支持任意binary，包括中文。但其实从上面可以看到，GET和POST实际上都能用url和body。因此所谓编码确切地说应该是http中url用什么编码，body用什么编码。

先说下url。url只能支持ASCII的说法源自于RFC1738

> Thus, only alphanumerics, the special characters "$-_.+!*'(),", and reserved characters used for their reserved purposes may be used unencoded within a URL.

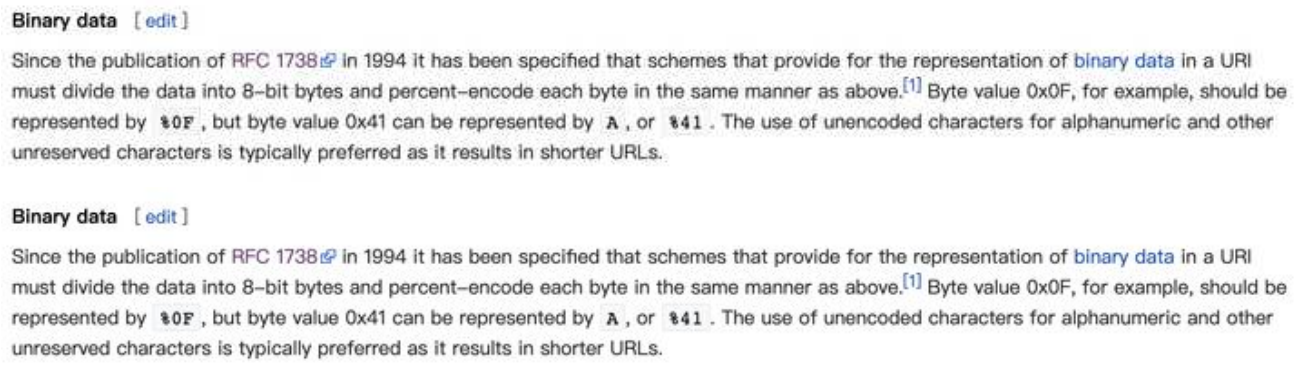
实际上这里规定的仅仅是一个ASCII的子集[a-zA-Z0-9$-_.+!*'(),]。它们是可以“不经编码”在url中使用。比如尽管空格也是ASCII字符，但是不能直接用在url里。

那这个“编码”是什么呢？如果有了特殊符号和中文怎么办呢？一种叫做percent encoding的编码方法就是干这个用的：

https://en.wikipedia.org/wiki/Percent-encodingen.wikipedia.org/wiki/Percent-encoding

这也就是为啥我们偶尔看到url里有一坨%和16位数字组成的序列。

使用Percent Encoding，即使是**binary data，也是可以通过编码后放在URL上**的。



Binary data [edit]

Since the publication of RFC 1738 in 1994 it has been specified that schemes that provide for the representation of binary data in a URI must divide the data into 8-bit bytes and percent-encode each byte in the same manner as above.[1] Byte value 0x0F, for example, should be represented by `%0F`, but byte value 0x41 can be represented by `A`, or `%41`. The use of unencoded characters for alphanumeric and other unreserved characters is typically preferred as it results in shorter URLs.

Binary data [edit]

Since the publication of RFC 1738 in 1994 it has been specified that schemes that provide for the representation of binary data in a URI must divide the data into 8-bit bytes and percent-encode each byte in the same manner as above.[1] Byte value 0x0F, for example, should be represented by `%0F`, but byte value 0x41 can be represented by `A`, or `%41`. The use of unencoded characters for alphanumeric and other unreserved characters is typically preferred as it results in shorter URLs.

但要特别注意，这个**编码方式只管把字符转换成URL可用字符，但是却不管字符集编码**（比如中文到底是用UTF8还是GBK）这块早期一直都相当乱，也没有什么统一规范。比如有时跟网页编码一样，有的是操作系统的编码一样。最要命的是浏览器的地址栏是不受开发者控制的。这样，对于同样一个带中文的url，如果有的浏览器一定要用GBK（比如老的IE8），有的一定要用UTF8（比如chrome）。后端就可能认不出来。对此常用的办法是避免让用户输入这种带中文的url。如果有这种形式的请求，都改成用户界面上输入，然后通过Ajax发出的办法。Ajax发出的编码形式开发者是可以100%控制的。

> 不过目前基本上utf8已经大一统了。现在的开发者除非是被国家规定要求一定要用GB系列编码的场景，基本上不会再遇到这类问题了。

关于url的编码，阮一峰的一篇文章有比较详细的解释：

关于URL编码 - 阮一峰的网络日志www.ruanyifeng.com/blog/2010/02/url_encoding.html![img](https://pic1.zhimg.com/v2-ceb5198ed552632b53c3dcd34b85a230_180x120.jpg)

> 顺便说一句，尽管在浏览器地址栏可以看到中文。但这种url在发送请求过程中，浏览器会把中文用字符编码+Percent Encode翻译为真正的url，再发给服务器。浏览器地址栏里的中文只是想让用户体验好些而已。

再讨论下Body。HTTP Body相对好些，因为有个Content-Type来比较明确的定义。比如：

```
POST xxxxxx HTTP/1.1
...
Content-Type: application/x-www-form-urlencoded ; charset=UTF-8
```

这里Content-Type会同时定义请求body的格式（application/x-www-form-urlencoded）和字符编码（UTF-8）。

所以body和url都可以提交中文数据给后端，但是POST的规范好一些，相对不容易出错，容易让开发者安心。对于GET+url的情况，只要不涉及到在老旧浏览器的地址栏输入url，也不会有什么太大的问题。

回到POST，浏览器直接发出的POST请求就是表单提交，而表单提交只有application/x-www-form-urlencoded针对简单的key-value场景；和multipart/form-data，针对只有文件提交，或者同时有文件和key-value的混合提交表单的场景。

如果是Ajax或者其他HTTP Client发出去的POST请求，其body格式就非常自由了，常用的有json，xml，文本，csv……甚至是你自己发明的格式。只要前后端能约定好即可。

浏览器的POST需要发两个请求吗？

上文中的"HTTP 格式"清楚的显示了HTTP请求可以被大致分为"请求头"和"请求体"两个部分。使用HTTP时大家会有一个约定，即所有的"控制类"信息应该放在请求头中，具体的数据放在请求体里"。于是服务器端在解析时，总是会先完全解析全部的请求头部。这样，服务器端总是希望能够了解请求的控制信息后，就能决定这个请求怎么进一步处理，是拒绝，还是根据content-type去调用相应的解析器处理数据，或者直接用zero copy转发。

比如在用Java写服务时，请求处理代码总是能从HttpSerlvetRequest里getParameter/Header/url。这些信息都是请求头里的，框架直接就解析了。而对于请求体，只提供了一个inputstream，如果开发人员觉得应该进一步处理，就自己去读取和解析请求体。这就能体现出服务器端对请求头和请求体的不同处理方式。

举个实际的例子，比如写一个上传文件的服务，请求url中包含了文件名称，请求体中是个尺寸为几百兆的压缩二进制流。服务器端接收到请求后，就可以先拿到请求头部，查看用户是不是有权限上传，文件名是不是符合规范等。如果不符合，就不再处理请求体的数据了，直接丢弃。而不用等到整个请求都处理完了再拒绝。

为了进一步优化，客户端可以利用HTTP的Continued协议来这样做：客户端总是先发送所有请求头给服务器，让服务器校验。如果通过了，服务器回复"100 - Continue"，客户端再把剩下的数据发给服务器。如果请求被拒了，服务器就回复个400之类的错误，这个交互就终止了。这样，就可以避免浪费带宽传请求体。但是代价就是会多一次Round Trip。如果刚好请求体的数据也不多，那么一次性全部发给服务器可能反而更好。

基于此，客户端就能做一些优化，比如内部设定一次POST的数据超过1KB就先只发"请求头"，否则就一次性全发。客户端甚至还可以做一些Adaptive的策略，统计发送成功率，如果成功率很高，就总是全部发等等。不同浏览器，不同的客户端（curl，postman）可以有各自的不同的方案。不管怎样做，优化目的总是在提高数据吞吐和降低带宽浪费上做一个折衷。

因此到底是发一次还是发N次，客户端可以很灵活的决定。因为不管怎么发都是符合HTTP协议的，因此我们应该视为这种优化是一种实现细节，而不用扯到GET和POST本身的区别上。更不要当个什么世纪大发现。

到底什么算请求体

看完了上面的内容后，读者也许会对"什么是请求体"感到困惑不已，比如x-www-form-endocded编码的body算不算"请求体"呢?

从HTTP协议的角度，"请求头"就是Method + URL（含querystring）+ Headers；再后边的都是请求体。

但是从业务角度，如果你把一次请求立即为一个调用的话。比如上面的

```
POST http://foo.com/books
{
  "title": "大宽宽的碎碎念",
  "author": "大宽宽",
  ...
}
```

用Java写大概等价于

```
createBook("大宽宽的碎碎念", "大宽宽");
```

那么这一行函数名和两个参数都可以看作是一个**请求，不区分头和体**。即便用HTTP协议实现，title和author编码到了HTTP请求体中。Java的HttpServletRequest支持用getParameter方法获取x-www-url-form-encoded中的数据，表达的意思就是"请求"的"参数"。

对于HTTP，**需要区分【头】和【体】**，Http Request和Http Response都这么区分。Http这么干主要用作：

- 对于HTTP代理
  - 支持转发规则，比如nginx先要解析请求头，拿到URL和Header才能决定怎么做（转发proxy_pass，重定向redirect，rewrite后重新判断……）
  - 需要用请求头的信息记录log。尽管请求体里的数据也可以记录，但一般只记录请求头的部分数据。
  - 如果代理规则不涉及到请求体，那么请求体就可以不用从内核态的page cache复制一份到用户态了，可以直接zero copy转发。这对于上传文件的场景极为有效。
  - ……
- 对于HTTP服务器
  - 可以通过请求头进行ACL控制，比如看看Athorization头里的数据是否能让认证通过
  - 可以做一些拦截，比如看到Content-Length里的数太大，或者Content-Type自己不支持，或者Accept要求的格式自己无法处理，就直接返回失败了。
  - 如果body的数据很大，利用Stream API，可以方便支持一块一块的处理数据，而不是一次性全部读取出来再操作，以至于占用大量内存。
  - ……

但从高一级的业务角度，我们在意的其实是【请求】和【返回】。当我们在说"请求头"这三个字时，也许实际的意思是【请求】。而用HTTP实现【请求】时，可能仅仅用到【HTTP的请求头】（比如大部分GET请求），也可能是【HTTP请求头】+【HTTP请求体】（比如用POST实现一次下单）。

总之，这里有两层，不要混哦。

关于URL的长度

因为上面提到了不论是GET和POST都可以使用URL传递数据，所以我们常说的"GET数据有长度限制"其实是指"URL的长度限制"。

HTTP协议本身对URL长度并没有做任何规定。实际的限制是由客户端/浏览器以及服务器端决定的。

先说浏览器。不同浏览器不太一样。比如我们常说的2048个字符的限制，其实是IE8的限制。并且原始文档的说的其实是"URL的最大长度是2083个字符，path的部分最长是2048个字符"。见https://support.microsoft.com/en-us/help/208427/maximum-url-length-is-2-083-characters-in-internet-explorer。IE8之后的IE URL限制我没有查到明确的文档，但有些资料称IE 11的地址栏只能输入法2047个字符，但是允许用户点击html里的超长URL。我没实验，哪位有兴趣可以试试。

Summary
Microsoft Internet Explorer has a maximum uniform resource locator (URL) length of 2,083 characters. Internet Explorer also has a maximum path length of 2,048 characters. This limit applies to both POST request and GET request URLs.

Chrome的URL限制是2MB，见
https://chromium.googlesource.com/chromium/src/+/master/docs/security/url_display_guidelines/url_display_guidelines.md

URL Length

In general, the *web platform* does not have limits on the length of URLs (although 2^31 is a common limit). *Chrome* limits URLs to a maximum length of **2MB** for practical reasons and to avoid causing denial-of-service problems in inter-process communication.

On most platforms, Chrome's omnibox limits URL display to **32kB** (`kMaxURLDisplayChars`) although a **1kB** limit is used on VR platforms.

URL Length

In general, the *web platform* does not have limits on the length of URLs (although 2^31 is a common limit). *Chrome* limits URLs to a maximum length of **2MB** for practical reasons and to avoid causing denial-of-service problems in inter-process communication.

On most platforms, Chrome's omnibox limits URL display to **32kB** (`kMaxURLDisplayChars`) although a **1kB** limit is used on VR platforms.

Safari，Firefox等浏览器也有自己的限制，但都比IE大的多，这里就不挨个列出了。

然而新的IE已经开始使用Chrome的内核了，也就意味着"浏览器端URL的长度限制为2048字符"这种说法会慢慢成为历史。

其他的客户端，比如Java的，js的http client大多数也并没有限制URL最大有多长。

除了浏览器，服务器这边也有限制，比如apache的LimieRequestLine指令。

LimitRequestLine Directive

Description:	Limit the size of the HTTP request line that will be accepted from the client
Syntax:	`LimitRequestLine` *bytes*
Default:	`LimitRequestLine 8190`
Context:	server config, virtual host
Status:	Core
Module:	core

This directive sets the number of *bytes* that will be allowed on the HTTP request-line.

The `LimitRequestLine` directive allows the server administrator to set the limit on the allowed size of a client's HTTP request-line. Since the request-line consists of the HTTP method, URI, and protocol version, the `LimitRequestLine` directive places a restriction on the length of a request-URI allowed for a request on the server. A server needs this value to be large enough to hold any of its resource names, including any information that might be passed in the query part of a `GET` request.

This directive gives the server administrator greater control over abnormal client request behavior, which may be useful for avoiding some forms of denial-of-service attacks.

LimitRequestLine Directive

Description:	Limit the size of the HTTP request line that will be accepted from the client
Syntax:	`LimitRequestLine` *bytes*
Default:	`LimitRequestLine 8190`
Context:	server config, virtual host
Status:	Core
Module:	core

This directive sets the number of *bytes* that will be allowed on the HTTP request-line.

The `LimitRequestLine` directive allows the server administrator to set the limit on the allowed size of a client's HTTP request-line. Since the request-line consists of the HTTP method, URI, and protocol version, the `LimitRequestLine` directive places a restriction on the length of a request-URI allowed for a request on the server. A server needs this value to be large enough to hold any of its resource names, including any information that might be passed in the query part of a `GET` request.

This directive gives the server administrator greater control over abnormal client request behavior, which may be useful for avoiding some forms of denial-of-service attacks.

apache实际上限制的是HTTP请求第一行“Request Line”的长度，即 那一行。

再比如nginx用`large_client_header_buffers`指令来分配请求头中的很长数据的buffer。这个buffer可以用来处理url，header value等。

Syntax: **large_client_header_buffers** *number size*;
Default: large_client_header_buffers 4 8k;
Context: http, server

Sets the maximum *number* and *size* of buffers used for reading large client request header. A request line cannot exceed the size of one buffer, or the 414 (Request-URI Too Large) error is returned to the client. A request header field cannot exceed the size of one buffer as well, or the 400 (Bad Request) error is returned to the client. Buffers are allocated only on demand. By default, the buffer size is equal to 8K bytes. If after the end of request processing a connection is transitioned into the keep-alive state, these buffers are released.

Syntax: **large_client_header_buffers** *number size*;
Default: large_client_header_buffers 4 8k;
Context: http, server

Sets the maximum *number* and *size* of buffers used for reading large client request header. A request line cannot exceed the size of one buffer, or the 414 (Request-URI Too Large) error is returned to the client. A request header field cannot exceed the size of one buffer as well, or the 400 (Bad Request) error is returned to the client. Buffers are allocated only on demand. By default, the buffer size is equal to 8K bytes. If after the end of request processing a connection is transitioned into the keep-alive state, these buffers are released.

Tomcat的限制是web.xml里maxHttpHeaderSize来设置的，控制的是整个“请求头”的总长度。

	be imposed. If not specified, the default value of 8192 will be used.
maxHttpHeaderSize	The maximum size of the request and response HTTP header, specified in bytes. If not specified, this attribute is set to 8192 (8 KB).

	be imposed. If not specified, the default value of 8192 will be used.
maxHttpHeaderSize	The maximum size of the request and response HTTP header, specified in bytes. If not specified, this attribute is set to 8192 (8 KB).

为啥要限制呢？如果写过解析一段字符串的代码就能明白，解析的时候要分配内存。对于一个字节流的解析，必须分配buffer来保存所有要存储的数据。而URL这种东西必须当作一个整体看待，无法一块一块处理，于是就处理一个请求时必须分配一整块足够大的内存。如果URL太长，而并发又很高，就容易挤爆服务器的内存；同时，超长URL的好处并不多，我也只有处理老系统的URL时因为不敢碰原来的逻辑，又得追加更多数据，才会使用超长URL。

对于开发者来说，使用超长的URL完全是给自己埋坑，需要同时要考虑前后端，以及中间代理每一个环节的配置。此外，超长URL会影响搜索引擎的爬虫，有些爬虫甚至无法处理超过2000个字节的URL。这也就意味着这些URL无法被搜到，坑爹啊。

其实并没有太大必要弄清楚精确的URL最大长度限制。我个人的经验是，只要某个要开发的资源/api的URL长度**有可能达到2000个bytes以上，就必须使用body来传输数据，除非有特殊情况。**至于到底是GET + body还是POST + body可以看情况决定。

> 留意，1个汉字字符经过UTF8编码 + percent encoding后会变成9个字节，别算错哦。

总结

上面讲了一大堆，是希望读者不要死记硬背GET和POST的区别，而是能从更广的层面去看待和思考这个问题。

最后，**协议都是人定的。**只要客户端和服务器能彼此认同，就能工作。在常规的情况下，用符合规范的方式去实现系统可以减少很多工作量——大家都约定好了，就不要折腾了。但是，总会有一些情况用常规规范不合适，不满足需求。这时思路也不能被规范限制死，更不要死抠RFC。这些规范也许不能处理你遇到的特殊问题。比如：

- Elastic Search的_search接口使用GET，却用body来表达查询，因为查询很复杂，用querystring很麻烦，必须用json格式才舒服，在请求体用json编码更加容易，不用折腾percent encoding。
- 用POST写一个接口下单时可能也要考虑幂等，因为前端可能实现“下单按键”有bug，造成用户一次点击发出N个请求。你不能说因为POST by design应该是不幂等就不管了。

协议是死的，人是活的。遇到实际的问题时灵活的运用手上的工具满足需求就好。