> 在 HTTP/2 中， 因为可以复用同一个 TCP 连接，你会发现发送请求是长这样的

33.2 二进制传输

> HTTP/2 中所有加强性能的核心点在于此 。在之前的 HTTP 版本中， 我们是通过文本的方式传输数据 。在 HTTP/2 中引入了新的编码机制，所有传输的数据都会被分割， 并采用二进制格式编码。

33.3 多路复用

- 在 HTTP/2 中，有两个非常重要的概念，分别是帧（frame）和流（ stream）。
- 帧代表着最小的数据单位，每个帧会标识出该帧属于哪个流， 流也就是多个帧组成的数据流。
- 多路复用，就是在一个 TCP 连接中可以存在多条流 。换句话说，也就是可以发送多个请求，对端可以通过帧中的标识知道属于哪个请求 。通过这个技术， 可以避免 HTTP 旧版本中的队头阻塞问题，极大的提高传输性能。

33.4 Header 压缩

- 在 HTTP/1 中， 我们使用文本的形式传输 header ，在 header 携带 cookie 的情况下， 可能每次都需要重复传输几百到几千的字节。
- 在 HTTP / 2 中，使用了 HPACK 压缩格式对传输的 header 进行编码，减少了 header 的大小 。并在两端维护了索引表，用于记录出现过的 header ，后面在传输过程中就可以传输已经记录过的 header 的键名，对端收到数据后就可以通过键名找到对应的值。

33.5 服务端 Push

- 在 HTTP/2 中， 服务端可以在客户端某个请求后， 主动推送其他资源。
- 可以想象以下情况，某些资源客户端是一定会请求的， 这时就可以采取服务端 push 的技术，提前给客户端推送必要的资源， 这样就可以相对减少一点延迟时间 。当然在浏览器兼容的情况下你也可以使用 prefetch

33.6 HTTP/3

- 虽然 HTTP/2 解决了很多之前旧版本的问题，但是它还是存在一个巨大的问题， 虽然这个问题并不是它本身造成的， 而是底层支撑的 TCP 协议的问题。
- 因为 HTTP/2 使用了多路复用，一 般来说同一域名下只需要使用一个 TCP 连接 。当这个连接中出现了丢包的情况，那就会导致 HTTP/2 的表现情况反倒不如 HTTP/1 了。
- 因为在出现丢包的情况下， 整个 TCP 都要开始等待重传，也就导致了后面的所有数据都被阻塞了 。但是对于 HTTP/1 来说， 可以开启多个 TCP 连接， 出现这种情况反到只会影响其中一个连接， 剩余的 TCP 连接还可以正常传输数据。
- 那么可能就会有人考虑到去修改 TCb 协议， 其实这已经是一件不可能完成的任务了 。因为 TCb 存在的时间实在太长， 已经充斥在各种设备中， 并且这个协议是由操作系统实现的，更新起来不大现实。
- 基于这个原因， Google 就更起炉灶搞了一个基于 UDP 协议的 QUIC 协议， 并且使用在了 HTTP/3 上， 当然 HTTP/3 之前名为 HTTP-over-QUIC ，从这个名字中我们也可以发现， HTTb/3 最大的改造就是使用了 QUIC ，接下来我们就来学习关于这个协议的内容。

QUIC

> 之前我们学习过 UDb 协议的内容，知道这个协议虽然效率很高，但是并不是那么的可靠 。QUIC 虽然基于 UDb，但是在原本的基础上新增了很多功能， 比如多路复用 、0-RTT、使用 TLS1.3 加密 、流量控制 、有序交付 、重传等等功能 。这里我们就挑选几个重要的功能学习下这个协议的内容。

多路复用

> 虽然 HTTb/2 支持了多路复用，但是 TCb 协议终究是没有这个功能的 。QUIC 原生就实现了这个功能， 并且传输的单个数据流可以保证有序交付且不会影响其他的数据流， 这样的技术就解决了之前 TCb 存在的问题。

- 并且 QUIC 在移动端的表现也会比 TCP 好 。因为 TCP 是基于 IP 和端口去识别连接的， 这种方式在多变的移动端网络环境下是很脆弱的 。但是 QUIC 是通过 ID ** 的方式去识别一个连接，不管你网络环境如何变化， 只要 ID 不变，就能迅速重连上。

0-RTT

> 通过使用类似 TCb 快速打开的技术， 缓存当前会话的上下文，在下次恢复会话的时候， 只需要将之前的缓存传递给服务端验证通过就可以进行传输了。

纠错机制

- 假如说这次我要发送三个包，那么协议会算出这三个包的异或值并单独发出一个校验包，也就是总共发出了四个包。
- 当出现其中的非校验包丢包的情况时， 可以通过另外三个包计算出丢失的数据包的内容 。
- 当然这种技术只能使用在丢失一个包的情况下， 如果出现丢失多个包就不能使用纠错机制了， 只能使用重传的方式了

34 设计模式

> 设计模式总的来说是一个抽象的概念， 前人通过无数次的实践总结出的一套写代码的方式， 通过这种方式写的代码可以让别人更加容易阅读 、维护以及复用。

34.1 工厂模式

> 工厂模式分为好几种， 这里就不一一讲解了， 以下是一个简单工厂模式的例子

```js
class Man {
  constructor(name) {
    this.name = name
  }
  alertName() {
    alert(this.name)
  }
}

class Factory {
  static create(name) {
    return new Man(name)
  }
}
```

```
Factory.create( 'yck').alertName()
```

- 当然工厂模式并不仅仅是用来 new 出实例。
- 可以想象一个场景 。假设有一份很复杂的代码需要用户去调用，但是用户并不关心这些复杂的代码， 只需要你提供给我一个接口去调用，用户只负责传递需要的参数， 至于这些参数怎么使用， 内部有什么逻辑是不关心的， 只需要你最后返回我一个实例 。这个构造过程就是工厂。
- 工厂起到的作用就是隐藏了创建实例的复杂度， 只需要提供一个接口， 简单清晰。
- 在 Vue 源码中，你也可以看到工厂模式的使用， 比如创建异步组件

```js
export function createComponent (
  Ctor: Class<Component> | Function | Object | void,
  data: ?VNodeData,
  context: Component,
  children: ?Array<VNode>,
  tag?: string
): VNode | Array<VNode> | void {

    // 逻辑处理 ...

  const vnode = new VNode(
    `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,
    data, undefined, undefined, undefined, context,
    { Ctor, propsData, listeners, tag, children },
    asyncFactory
  )

  return vnode
}
```

> 在上述代码中， 我们可以看到我们只需要调用 createComponent 传入参数就能创建一个组件实例，但是创建这个实例是很复杂的一个过程， 工厂帮助我们隐藏了这个复杂的过程， 只需要一句代码调用就能实现功能

34.2 单例模式

- 单例模式很常用， 比如全局缓存 、全局状态管理等等这些只需要一个对象，就可以使用单例模式。

- 单例模式的核心就是保证全局只有一个对象可以访问 。因为 JS 是门无类的语言，所以别的语言实现单例的方式并不能套入 JS 中， 我们只需要用一个变量确保实例只创建一次就行， 以下是如何实现单例模式的例子

```js
class Singleton {
  constructor() {}
}

Singleton.getInstance = (function() {
  let instance
  return function() {
    if ( !instance) {
      instance = new Singleton()
    }
    return instance
  }
})()

let s1 = Singleton.getInstance()
let s2 = Singleton.getInstance()
console.log(s1 === s2) // true
```

> 在 Vuex 源码中，你也可以看到单例模式的使用， 虽然它的实现方式不大一样， 通过一个外部变量来控制只安装一次 Vuex

```js
let Vue //  bind on install

export function install (_Vue) {
  if (Vue && _Vue === Vue) {
    // 如果发现 Vue 有值，就不重新创建实例了
    return
  }
  Vue = _Vue
  applyMixin(Vue)
}
```

34.3 适配器模式

- 适配器用来解决两个接口不兼容的情况，不需要改变已有的接口， 通过包装一层的方式实现两个接口的正常协作。

- 以下是如何实现适配器模式的例子

```js
class Plug {
  getName() {
    return '港版插头'
  }
}

class Target {
  constructor() {
    this.plug = new Plug()
  }
  getName() {
    return this.plug.getName() + ' 适配器转二脚插头'
  }
}

let target = new Target()
target.getName() // 港版插头 适配器转二脚插头
```

> 在 Vue 中， 我们其实经常使用到适配器模式 。比如父组件传递给子组件一个时间戳属性， 组件内部需要将时间戳转为正常的日期显示，一 般会使用 computed 来做转换这件事情， 这个过程就使用到了适配器模式

34.4 装饰模式

- 装饰模式不需要改变已有的接口，作用是给对象添加功能 。就像我们经常需要给手机戴个保护套防摔一样，不改变手机自身， 给手机添加了保护套提供防摔功能。
- 以下是如何实现装饰模式的例子，使用了 ES7 中的装饰器语法

```js
function readonly(target, key, descriptor) {
  descriptor.writable = false
  return descriptor
}

class Test {
  @readonly
  name = 'yck'
}

let t = new Test()
```

```
t.yck = '111' // 不可修改
```

> 在 React 中， 装饰模式其实随处可见

```js
import { connect } from 'react-redux'
class MyComponent extends React.Component {
    // ...
}
export default connect(mapStateToProps)(MyComponent)
```

34.5 代理模式

- 代理是为了控制对对象的访问，不让外部直接访问到对象 。在现实生活中，也有很多代理的场景 。比如你需要买一件国外的产品， 这时候你可以通过代购来购买产品。
- 在实际代码中其实代理的场景很多，也就不举框架中的例子了， 比如事件代理就用到了代理模式

```html
<ul id="ul">
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ul>
<script>
    let ul = document.querySelector( '#ul')
    ul.addEventListener( 'click', (event) => {
        console.log(event.target);
    })
</script>
```

> 因为存在太多的 li，不可能每个都去绑定事件 。这时候可以通过给父节点绑定一个事件，让父节点作为代理去拿到真实点击的节点。

34.6 发布-订阅模式

- 发布-订阅模式也叫做观察者模式 。通过一对一或者一对多的依赖关系， 当对象发生改变时，订阅方都会收到通知 。在现实生活中，也有很多类似场景， 比如我需要在购物网站上购买一个产品，但是发现该产品目前处于缺货状态， 这时候我可以点击有货通知的按钮，让网站在产品有货的时候通过短信通知我。
- 在实际代码中其实发布-订阅模式也很常见， 比如我们点击一个按钮触发了点击事件就是使用了该模式

```html
<ul id="ul"></ul>
<script>
    let ul = document.querySelector( '#ul')
    ul.addEventListener( 'click', (event) => {
        console.log(event.target);
    })
</script>
```

> 在 Vue 中， 如何实现响应式也是使用了该模式 。对于需要实现响应式的对象来说，在 get 的时候会进行依赖收集， 当改变了对象的属性时，就会触发派发更新。

34.7 外观模式

- 外观模式提供了一个接口， 隐藏了内部的逻辑，更加方便外部调用。
- 个例子来说， 我们现在需要实现一个兼容多种浏览器的添加事件方法

```js
function addEvent(elm, evType, fn, useCapture) {
  if (elm.addEventListener) {
    elm.addEventListener(evType, fn, useCapture)
    return true
  } else if (elm.attachEvent) {
    var r = elm.attachEvent("on" + evType, fn)
    return r
  } else {
    elm["on" + evType] = fn
  }
}
```

> 对于不同的浏览器，添加事件的方式可能会存在兼容问题 。如果每次都需要去这样写一遍的话肯定是不能接受的，所以我们将这些判断逻辑统一封装在一个

> 接口中，外部需要添加事件只需要调用 `addEvent` 即可。

35 常见数据结构

35.1 时间复杂度

在进入正题之前， 我们先来了解下什么是时间复杂度。

- 通常使用最差的时间复杂度来衡量一个算法的好坏。
- 常数时间 `O(1)` 代表这个操作和数据量没关系， 是一个固定时间的操作， 比如说四则运算。
- 对于一个算法来说， 可能会计算出操作次数为 `aN + 1` ， `N` 代表数据量 。那么该算法的时间复杂度就是 `O(N)` 。因为我们在计算时间复杂度的时候，数据量通常是非常大的， 这时候低阶项和常数项可以忽略不计。
- 当然可能会出现两个算法都是 `O(N)` 的时间复杂度，那么对比两个算法的好坏就要通过对比低阶项和常数项了

35.2 栈

概念

- 栈是一个线性结构，在计算机中是一个相当常见的数据结构。
- 栈的特点是只能在某一端添加或删除数据， 遵循先进后出的原则

实现

> 每种数据结构都可以用很多种方式来实现， 其实可以把栈看成是数组的一个子集，所以这里使用数组来实现

```js
class Stack {
  constructor() {
    this.stack = []
  }
  push(item) {
    this.stack.push(item)
  }
  pop() {
```