

```

    // 调用 splice 函数触发派发更新
    // 该函数已被重写
    target.length = Math.max(target.length, key)
    target.splice(key, 1, val)
    return val
  }
  // 判断 key 是否已经存在
  if (key in target && !(key in Object.prototype)) {
    target[key] = val
    return val
  }
  const ob = (target: any).__ob__
  // 如果对象不是响应式对象，就赋值返回
  if (!ob) {
    target[key] = val
    return val
  }
  // 进行双向绑定
  defineReactive(ob.value, key, val)
  // 手动派发更新
  ob.dep.notify()
  return val
}

```

对于数组而言，**Vue** 内部重写了以下函数实现派发更新

```

// 获得数组原型
const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)
// 重写以下函数
const methodsToPatch = [
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]
methodsToPatch.forEach(function (method) {
  // 缓存原生函数
  const original = arrayProto[method]
  // 重写函数

```

js

```
def(arrayMethods, method, function mutator (...args) {  
  // 先调用原生函数获得结果  
  const result = original.apply(this, args)  
  const ob = this.__ob__  
  let inserted  
  // 调用以下几个函数时，监听新数据  
  switch (method) {  
    case 'push':  
    case 'unshift':  
      inserted = args  
      break  
    case 'splice':  
      inserted = args.slice(2)  
      break  
  }  
  if (inserted) ob.observeArray(inserted)  
  // 手动派发更新  
  ob.dep.notify()  
  return result  
})  
})
```

28.9.2 编译过程

想必大家在使用 Vue 开发的过程中，基本都是使用模板的方式。那么你有过「模板是怎么在浏览器中运行的」这种疑虑嘛？

- 首先直接把模板丢到浏览器中肯定是不能运行的，模板只是为了方便开发者进行开发。
Vue 会通过编译器将模板通过几个阶段最终编译为 `render` 函数，然后通过执行 `render` 函数生成 `Virtual DOM` 最终映射为真实 `DOM` 。
- 接下来我们就来学习这个编译的过程，了解这个过程中大概发生了什么事情。这个过程其中又分为三个阶段，分别为：
 - 将模板解析为 `AST`
 - 优化 `AST`
 - 将 `AST` 转换为 `render` 函数

在第一个阶段中，最主要的事情还是通过各种各样的正则表达式去匹配模板中的内容，然后将内容提取出来做各种逻辑操作，接下来会生成一个最基本的 `AST` 对象

```
{
  // 类型
  type: 1,
  // 标签
  tag,
  // 属性列表
  attrsList: attrs,
  // 属性映射
  attrsMap: makeAttrsMap(attrs),
  // 父节点
  parent,
  // 子节点
  children: []
}
```

- 然后会根据这个最基本的 AST 对象中的属性， 进一步扩展 AST 。
- 当然在这一阶段中， 还会进行其他的一些判断逻辑。比如说对比前后开闭标签是否一致， 判断根组件是否只存在一个， 判断是否符合 HTML5 Content Model 规范等等问题。
- 接下来就是优化 AST 的阶段。在当前版本下， Vue 进行的优化内容其实还是不多的。只是对节点进行了静态内容提取，也就是将永远不会变动的节点提取了出来， 实现复用 Virtual DOM， 跳过对比算法的功能。在下一个大版本中， Vue 会在优化 AST 的阶段继续发力， 实现更多的优化功能，尽可能的在编译阶段压榨更多的性能， 比如说提取静态的属性等等优化行为。
- 最后一个阶段就是通过 AST 生成 render 函数了。其实这一阶段虽然分支有很多，但是最主要的目的就是遍历整个 AST， 根据不同的条件生成不同的代码罢了。

28.9.3 NextTick 原理分析

nextTick 可以让我们在下次 DOM 更新循环结束之后执行延迟回调，用于获得更新后的 DOM 。

- 在 Vue 2.4 之前都是使用的 microtasks， 但是 microtasks 的优先级过高，在某些情况下可能会出现比事件冒泡更快的情况，但如果都使用 macrotasks 又可能会出现渲染的性能问题。所以在新版本中，会默认使用 microtasks，但在特殊情况下会使用 macrotasks， 比如 v-on。
- 对于实现 macrotasks， 会先判断是否能使用 setImmediate， 不能的话降级为 MessageChannel， 以上都不行的话就使用 setTimeout

```
if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
  macroTimerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else if (
  typeof MessageChannel !== 'undefined' &&
  (isNative(MessageChannel) ||
    // PhantomJS
    MessageChannel.toString() === '[object MessageChannelConstructor]')
) {
  const channel = new MessageChannel()
  const port = channel.port2
  channel.port1.onmessage = flushCallbacks
  macroTimerFunc = () => {
    port.postMessage(1)
  }
} else {
  macroTimerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}
```

以上代码很简单，就是判断能不能使用相应的 API

29 React常考知识点

29.1 生命周期

在 V16 版本中引入了 **Fiber** 机制。这个机制一定程度上的影响了部分生命周期的调用，并且也引入了新的 2 个 API 来解决问题

在之前的版本中，如果你拥有一个很复杂的复合组件，然后改动了最上层组件的 **state**，那么调用栈可能会很长

- 调用栈过长，再加上中间进行了复杂的操作，就可能导致长时间阻塞主线程，带来不好的用户体验。**Fiber** 就是为了解决该问题而生

- **Fiber** 本质上是一个虚拟的堆栈帧，新的调度器会按照优先级自由调度这些帧，从而将之前的同步渲染改成了异步渲染，在不影响体验的情况下去分段计算更新
- 对于如何区别优先级，**React** 有自己的一套逻辑。对于动画这种实时性很高的东西，也就是 **16 ms** 必须渲染一次保证不卡顿的情况下，**React** 会每 **16 ms**（以内）暂停一下更新，返回来继续渲染动画
- 对于异步渲染，现在渲染有两个阶段：**reconciliation** 和 **commit**。前者过程是可以打断的，后者不能暂停，会一直更新界面直到完成。

1. Reconciliation 阶段

- **componentWillMount**
- **componentWillReceiveProps**
- **shouldComponentUpdate**
- **componentWillUpdate**

2. Commit 阶段

- **componentDidMount**
- **componentDidUpdate**
- **componentWillUnmount**

因为 **Reconciliation** 阶段是可以被打断的，所以 **Reconciliation** 阶段会执行的生命周期函数就可能会出现调用多次的情况，从而引起 **Bug**。由此对于 **Reconciliation** 阶段调用的几个函数，除了 **shouldComponentUpdate** 以外，其他都应该避免去使用，并且 **V16** 中也引入了新的 **API** 来解决这个问题。

getDerivedStateFromProps 用于替换 **componentWillReceiveProps**，该函数会在初始化和 **update** 时被调用

```
class ExampleComponent extends React.Component {  
  // Initialize state in constructor,  
  // Or with a property initializer.  
  state = {};  
  
  static getDerivedStateFromProps(nextProps, prevState) {  
    if (prevState.someMirroredValue !== nextProps.someValue) {  
      js
```

```

    return {
      derivedData: computeDerivedState(nextProps),
      someMirroredValue: nextProps.someValue
    };
  }

  // Return null to indicate no change to state.
  return null;
}
}

```

`getSnapshotBeforeUpdate` 用于替换 `componentWillUpdate`，该函数会在 `update` 后 `DOM` 更新前被调用，用于读取最新的 `DOM` 数据

更多详情 <http://blog.poetries.top/2018/11/18/react-lifecycle>

29.2 setState

- `setState` 在 `React` 中是经常使用的一个 `API`，但是它存在的一些问题经常会导致初学者出错，核心原因就是因为这个 `API` 是异步的。
- 首先 `setState` 的调用并不会马上引起 `state` 的改变，并且如果你一次调用了多个 `setState`，那么结果可能并不如你期待的一样。

```


handle() {
  // 初始化 `count` 为 0
  console.log(this.state.count) // -> 0
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
  console.log(this.state.count) // -> 0
}

```

js

- 第一，两次的打印都为 0，因为 `setState` 是个异步 `API`，只有同步代码运行完毕才会执行。`setState` 异步的原因我认为在于，`setState` 可能会导致 `DOM` 的重绘，如果调用一次就马上去进行重绘，那么调用多次就会造成不必要的性能损失。设计成异步的话，就可以将多次调用放入一个队列中，在恰当的时候统一进行更新过程。

```
Object.assign(
  {},
  { count: this.state.count + 1 },
  { count: this.state.count + 1 },
  { count: this.state.count + 1 },
)
```

当然你也可以通过以下方式来实现调用三次 `setState` 使得 `count` 为 

```
handle() {
  this.setState((prevState) => ({ count: prevState.count + 1 }))
  this.setState((prevState) => ({ count: prevState.count + 1 }))
  this.setState((prevState) => ({ count: prevState.count + 1 }))
}
```

如果你想在每次调用 `setState` 后获得正确的 `state`，可以通过如下代码实现

```
handle() {
  this.setState((prevState) => ({ count: prevState.count + 1 }), () => {
    console.log(this.state)
  })
}
```

更多详情 <http://blog.poetries.top/2018/12/20/react-setState>

29.3 性能优化

- 在 `shouldComponentUpdate` 函数中我们可以通过返回布尔值来决定当前组件是否需要更新。这层代码逻辑可以是简单地浅比较一下当前 `state` 和之前的 `state` 是否相同，也可以是判断某个值更新了才触发组件更新。一般来说不推荐完整地对比当前 `state` 和之前的 `state` 是否相同，因为组件更新触发可能会很频繁，这样的完整对比性能开销会有点大，可能会造成得不偿失的情况。

当然如果真的想完整对比当前 `state` 和之前的 `state` 是否相同，并且不影响性能也是行得通的，可以通过 `immutable` 或者 `immer` 这些库来生成不可变对象。这类库对于操作大规模的数据来说会提升不错的性能，并且一旦改变数据就会生成一个新的对象，对比前后 `state` 是否一致也就方便多了，同时也很推荐阅读下 `immer` 的源码实现

另外如果只是单纯的浅比较一下，可以直接使用 `PureComponent`，底层就是实现了浅比较 `state`

```
class Test extends React.PureComponent {  
  render() {  
    return (  
      <div>  
        PureComponent  
      </div>  
    )  
  }  
}
```

js

这时候你可能会考虑到函数组件就不能使用这种方式了，如果你使用 `16.6.0` 之后的版本的话，可以使用 `React.memo` 来实现相同的功能

```
const Test = React.memo(() => (  
  <div>  
    PureComponent  
  </div>  
))
```

js

通过这种方式我们就可以既实现了 `shouldComponentUpdate` 的浅比较，又能够使用函数组件

29.4 通信

1. 父子通信

- 父组件通过 `props` 传递数据给子组件，子组件通过调用父组件传来的函数传递数据给父组件，这两种方式是最常用的父子通信实现办法。
- 这种父子通信方式也就是典型的单向数据流，父组件通过 `props` 传递数据，子组件不能直接修改 `props`，而是必须通过调用父组件函数的方式告知父组件修改数据。

2. 兄弟组件通信

对于这种情况可以通过共同的父组件来管理状态和事件函数。比如说其中一个兄弟组件调用父组件传递过来的事件函数修改父组件中的状态，然后父组件将状态传递给另一个兄弟组件

3. 跨多层次组件通信

如果你使用 16.3 以上版本的话，对于这种情况可以使用 Context API

```
// 创建 Context， 可以在开始就传入值
const StateContext = React.createContext()
class Parent extends React.Component {
  render () {
    return (
      // value 就是传入 Context 中的值
      <StateContext.Provider value= 'yck'>
        <Child />
      </StateContext.Provider>
    )
  }
}
class Child extends React.Component {
  render () {
    return (
      <ThemeContext.Consumer>
        // 取出值
        {context => (
          name is { context }
        )}
      </ThemeContext.Consumer>
    );
  }
}
```

js

4. 任意组件

这种方式可以通过 Redux 或者 Event Bus 解决， 另外如果你不怕麻烦的话， 可以使用这种方式解决上述所有的通信情况

29.5 HOC 是什么？相比 mixins 有什么优点？

很多人看到高阶组件（HOC）这个概念就被吓到了，认为这东西很难，其实这东西概念真的很简单，我们先来看一个例子。

```
function add(a, b) {  
  return a + b  
}
```

js

现在如果我想给这个 `add` 函数添加一个输出结果的功能，那么你可能会考虑我直接使用 `console.log` 不就实现了么。说的没错，但是如果我们想做的更加优雅并且容易复用和扩展，我们可以这样做

```
function withLog (fn) {  
  function wrapper(a, b) {  
    const result = fn(a, b)  
    console.log(result)  
    return result  
  }  
  return wrapper  
}  
  
const withLogAdd = withLog(add)  
withLogAdd(1, 2)
```

js

- 其实这个做法在函数式编程里称之为高阶函数，大家都知道 `React` 的思想中是存在函数式编程的，高阶组件和高阶函数就是同一个东西。我们实现一个函数，传入一个组件，然后在函数内部再实现一个函数去扩展传入的组件，最后返回一个新的组件，这就是高阶组件的概念，作用就是为了更好的复用代码。
- 其实 HOC 和 `Vue` 中的 `mixins` 作用是一致的，并且在早期 `React` 也是使用 `mixins` 的方式。但是在使用 `class` 的方式创建组件以后，`mixins` 的方式就不能使用了，并且其实 `mixins` 也是存在一些问题的，比如
 1. 隐含了一些依赖，比如我在组件中写了某个 `state` 并且在 `mixin` 中使用了，就这存在了一个依赖关系。万一下次别人要移除它，就得去 `mixin` 中查找依赖

2. 多个 `mixin` 中可能存在相同命名的函数，同时代码组件中也不能出现相同命名的函数，否则就是重写了，其实我一直觉得命名真的是一件麻烦事。。
3. 雪球效应，虽然我一个组件还是使用着同一个 `mixin`，但是一个 `mixin` 会被多个组件使用，可能会存在需求使得 `mixin` 修改原本的函数或者新增更多的函数，这样可能就会产生一个维护成本

`HOC` 解决了这些问题，并且它们达成的效果也是一致的，同时也更加的政治正确（毕竟更加函数式了）

29.6 事件机制

`React` 其实自己实现了一套事件机制，首先我们考虑一下以下代码：

```
const Test = ({ list, handleClick }) => ({  
  list.map((item, index) => (  
    <span onClick={handleClick} key={index}>{index}</span>  
  ))  
})
```

js

- 以上类似代码想必大家经常会写到，但是你是否考虑过点击事件是否绑定在了每一个标签上？事实当然不是，`JSX` 上写的事件并没有绑定在对应的真实 `DOM` 上，而是通过事件代理的方式，将所有的事件都统一绑定在了 `document` 上。这样的方式不仅减少了内存消耗，还能在组件挂载销毁时统一订阅和移除事件。
- 另外冒泡到 `document` 上的事件也不是原生浏览器事件，而是 `React` 自己实现的合成事件（`SyntheticEvent`）。因此我们如果不想要事件冒泡的话，调用 `event.stopPropagation` 是无效的，而应该调用 `event.preventDefault`

那么实现合成事件的目的是什么呢？总的来说在我看来好处有两点，分别是：

1. 合成事件首先抹平了浏览器之间的兼容问题，另外这是一个跨浏览器原生事件包装器，赋予了跨浏览器开发的能力
2. 对于原生浏览器事件来说，浏览器会给监听器创建一个事件对象。如果你有很多的事件监听，那么就需要分配很多的事件对象，造成高额的内存分配问题。但是对于合成事件来说，有一个事件池专门来管理它们的创建和销毁，当事件需要被使用时，就会从池子中复用对象，事件回调结束后，就会销毁事件对象上的属性，从而便于下次复用事件对象。

30 监控

前端监控一般分为三种，分别为页面埋点、性能监控以及异常监控。

这一章节我们将来学习这些监控相关的内容，但是基本不会涉及到代码，只是让大家了解下前端监控该用什么方式实现。毕竟大部分公司都只是使用到了第三方的监控工具，而不是选择自己造轮子

30.1 页面埋点

页面埋点应该是大家最常写的监控了，一般起码会监控以下几个数据：

- PV / UV
- 停留时长
- 流量来源
- 用户交互

对于这几类统计，一般的实现思路大致可以分为两种，分别为手写埋点和无埋点的方式。

相信第一种方式也是大家最常用的方式，可以自主选择需要监控的数据然后在相应的地方写入代码。这种方式的灵活性很大，但是唯一的缺点就是工作量较大，每个需要监控的地方都得插入代码。

另一种无埋点的方式基本不需要开发者手写埋点了，而是统计所有的事件并且定时上报。这种方式虽然没有前一种方式繁琐了，但是因为统计的是所有事件，所以还需要后期过滤出需要的数据。

30.2 性能监控

- 性能监控可以很好的帮助开发者了解在各种真实环境下，页面的性能情况是如何的。
- 对于性能监控来说，我们可以直接使用浏览器自带的 **Performance API** 来实现这个功能。

- 对于性能监控来说， 其实我们只需要调用

`performance.getEntriesByType('navigation')` 这行代码就行了。对，你没看错，一行代码我们就可以获得页面中各种详细的性能相关信息

我们可以发现这行代码返回了一个数组， 内部包含了相当多的信息，从数据开始在网络中传输到页面加载完成都提供了相应的数据

30.3 异常监控

- 对于异常监控来说， 以下两种监控是必不可少的，分别是代码报错以及接口异常上报。
- 对于代码运行错误， 通常的办法是使用 `window.onerror` 拦截报错。该方法能拦截到大部分的详细报错信息，但是也有例外
 1. 对于跨域的代码运行错误会显示 `Script error`。对于这种情况我们需要给 `script` 标签添加 `crossorigin` 属性
 2. 对于某些浏览器可能不会显示调用栈信息， 这种情况可以通过 `arguments.callee.caller` 来做栈递归
- 对于异步代码来说， 可以使用 `catch` 的方式捕获错误。比如 `Promise` 可以直接使用 `catch` 函数， `async await` 可以使用 `try catch`
- 但是要注意线上运行的代码都是压缩过的， 需要在打包时生成 `sourceMap` 文件便于 `debug`
- 对于捕获的错误需要上传给服务器， 通常可以通过 `img` 标签的 `src` 发起一个请求。
- 另外接口异常就相对来说简单了， 可以列举出出错的状态码。一旦出现此类的状态码就可以立即上报出错。接口异常上报可以让开发人员迅速知道有哪些接口出现了大面积的报错， 以便迅速修复问题。

31 TCP/UDP

31.1 UDP

网络协议是每个前端工程师都必须要掌握的知识， 我们将先来学习传输层中的两个协议： `UDP` 以及 `TCP`。对于大部分工程师来说最常用的协议也就是这两个了， 并且面试中经常会提问的也是关于这两个协议的区别

常考面试题： UDP 与 TCP 的区别是什么？

首先 UDP 协议是面向无连接的，也就是说不需要在正式传递数据之前先连接起双方。然后 UDP 协议只是数据报文的搬运工，不保证有序且不丢失的传递到对端，并且 UDP 协议也没有任何控制流量的算法，总的来说 UDP 相较于 TCP 更加的轻便

1. 面向无连接

- 首先 UDP 是不需要和 TCP 一样在发送数据前进行三次握手建立连接的，想发数据就可以开始发送了。
- 并且也只是数据报文的搬运工，不会对数据报文进行任何拆分和拼接操作。

具体来说就是：

- 在发送端，应用层将数据传递给传输层的 UDP 协议，UDP 只会给数据增加一个 UDP 头标识下是 UDP 协议，然后就传递给网络层了 在接收端，网络层将数据传递给传输层，UDP 只去除 IP 报文头就传递给应用层，不会任何拼接操作

2. 不可靠性

- 首先不可靠性体现在无连接上，通信都不需要建立连接，想发就发，这样的情况肯定不可靠。
- 并且收到什么数据就传递什么数据，并且也不会备份数据，发送数据也不会关心对方是否已经正确接收到数据了。
- 再者网络环境时好时坏，但是 UDP 因为没有拥塞控制，一直会以恒定的速度发送数据。即使网络条件不好，也不会对发送速率进行调整。这样实现的弊端就是在网络条件不好的情况下可能会导致丢包，但是优点也很明显，在某些实时性要求高的场景（比如电话会议）就需要使用 UDP 而不是 TCP

3. 高效

- 虽然 UDP 协议不是那么的可靠，但是正是因为它不是那么的可靠，所以也就没有 TCP 那么复杂了，需要保证数据不丢失且有序到达。
- 因此 UDP 的头部开销小，只有八字节，相比 TCP 的至少二十字节要少得多，在传输数据报文时是很高效的。

UDP 头部包含了以下几个数据

- 两个十六位的端口号，分别为源端口（可选字段）和目标端口 整个数据报文的长度
- 整个数据报文的检验和（IPv4 可选 字段），该字段用于发现头部信息和数据中的错误

4. 传输方式

UDP 不止支持一对一的传输方式，同样支持一对多，多对多，多对一的方式，也就是说 UDP 提供了单播，多播，广播的功能。

5. 适合使用的场景

UDP 虽然对比 TCP 有很多缺点，但是正是因为这些缺点造就了它高效的特性，在很多实时性要求高的地方都可以看到 UDP 的身影。

5.1 直播

- 想必大家都看过直播吧，大家可以考虑下如果直播使用了基于 TCb 的协议会发生什么事情？
 - TCP 会严格控制传输的正确性，一旦有某一个数据对端没有收到，就会停止下来直到对端收到这个数据。这种问题在网络条件不错的情况下可能并不会发生什么事情，但是在网络情况差的时候就会变成画面卡住，然后再继续播放下一帧的情况。
 - 但是对于直播来说，用户肯定关注的是最新的画面，而不是因为网络条件差而丢失的老旧画面，所以 TCP 在这种情况下无用武之地，只会降低用户体验。

5.2 王者荣耀

- 首先对于王者荣耀来说，用户体量是相当大的，如果使用 TCP 连接的话，就可能会出现服务器不够用的情况，因为每台服务器可供支撑的 TCP 连接数量是有限制的。
- 再者，因为 TCP 会严格控制传输的正确性，如果因为用户网络条件不好就造成页面卡顿然后再传输旧的游戏画面是肯定不能接受的，毕竟对于这类实时性要求很高的游戏来说，最新的游戏画面才是最需要的，而不是老旧的画面，否则角色都不知道死多少次了。

31.2 TCP

常考面试题： UDP 与 TCP 的区别是什么？

TCP 基本是和 UDP 反着来，建立连接断开连接都需要先需要进行握手。在传输数据的过程中，通过各种算法保证数据的可靠性，当然带来的问题就是相比 UDP 来说不那么的高效

1. 头部

从这个图上我们就可以发现 TCP 头部比 UDP 头部复杂的多

对于 TCP 头部来说，以下几个字段是很重要的

- Sequence number，这个序号保证了 TCb 传输的报文都是有序的，对端可以通过序号顺序的拼接报文
- Acknowledgement Number，这个序号表示数据接收端期望接收的下一个字节的编号是多少，同时也表示上一个序号的数据已经收到
- Window Size，窗口大小，表示还能接收多少字节的数据，用于流量控制
- 标识符

URG=1：该字段为一表示本数据报的数据部分包含紧急信息，是一个高优先级数据报文，此时紧急指针有效。紧急数据一定位于当前数据包数据部分的最前面，紧急指针标明了紧急数据的尾部。

ACK=1：该字段为一表示确认号字段有效。此外，TCP 还规定在连接建立后传送的所有报文段都必须把 ACK 置为一。

PSH=1：该字段为一表示接收端应该立即将数据 push 给应用层，而不是等到缓冲区满后再提交。

RST=1：该字段为一表示当前 TCP 连接出现严重问题，可能需要重新建立 TCb 连接，也可以用于拒绝非法的报文段和拒绝连接请求。

SYN=1：当 SYN=1，ACK=0 时，表示当前报文段是一个连接请求报文。当

SYN=1，ACK=1 时，表示当前报文段是一个同意建立连接的应答报文。

FIN=1：该字段为一表示此报文段是一个释放连接的请求报文。

2. 状态机

TCP 的状态机是很复杂的，并且与建立断开连接时的握手息息相关，接下来就来详细描述下两种握手

在这之前需要了解一个重要的性能指标 RTT 。该指标表示发送端发送数据到接收到对端数据所需的往返时间

2.1. 建立连接三次握手

- 首先假设主动发起请求的一端称为客户端，被动连接的一端称为服务端。不管是客户端还是服务端，TCP 连接建立完后都能发送和接收数据，所以 TCP 是一个全双工的协议。
- 起初，两端都为 CLOSED 状态。在通信开始前，双方都会创建 TCB 。服务器创建完 TCB 后便进入 LISTEN 状态，此时开始等待客户端发送数据

第一次握手

客户端向服务端发送连接请求报文段。该报文段中包含自身的数据通讯初始序号。请求发送后，客户端便进入 SYN-SENT 状态

第二次握手

服务端收到连接请求报文段后，如果同意连接，则会发送一个应答，该应答中也会包含自身的数据通讯初始序号，发送完成后便进入 SYN-RECEIVED 状态

第三次握手

- 当客户端收到连接同意的应答后，还要向服务端发送一个确认报文。客户端发完这个报文段后便进入 ESTABLISHED 状态，服务端收到这个应答后也进入 ESTABLISHED 状态，此时连接建立成功。
- PS：第三次握手中可以包含数据，通过快速打开（TFO）技术就可以实现这一功能。其实只要涉及到握手的协议，都可以使用类似 TFO 的方式，客户端和服务端存储相同的 cookie，下次握手时发出 cookie 达到减少 RTT 的目的。

常考面试题：为什么 **TCP** 建立连接需要三次握手，明明两次就可以建立起连接

- 因为这是为了防止出现失效的连接请求报文段被服务端接收的情况，从而产生错误。
- 可以想象如下场景。客户端发送了一个连接请求 **A**，但是因为网络原因造成了超时，这时 **TCP** 会启动超时重传的机制再次发送一个连接请求 **B**。此时请求顺利到达服务端，服务端应答完就建立了请求，然后接收数据后释放了连接。

假设这时候连接请求 **A** 在两端关闭后终于抵达了服务端，那么此时服务端会认为客户端又需要建立 **TCP** 连接，从而应答了该请求并进入 **ESTABLISHED** 状态。但是客户端其实是 **CLOSED** 的状态，那么就会导致服务端一直等待，造成资源的浪费。

PS：在建立连接中，任意一端掉线，**TCP** 都会重发 **SYN** 包，一般会重试五次，在建立连接中可能会遇到 **SYN Flood** 攻击。遇到这种情况你可以选择调低重试次数或者干脆在不能处理的情况下拒绝请求

2.2. 断开链接四次握手

TCP 是全双工的，在断开连接时两端都需要发送 **FIN** 和 **ACK**

第一次握手

若客户端 **A** 认为数据发送完成，则它需要向服务端 **B** 发送连接释放请求。

第二次握手

B 收到连接释放请求后，会告诉应用层要释放 **TCP** 链接。然后会发送 **ACK** 包，并进入 **CLOSE_WAIT** 状态，此时表明 **A** 到 **B** 的连接已经释放，不再接收 **A** 发的数据了。但是因为 **TCP** 连接是双向的，所以 **B** 仍旧可以发送数据给 **A**

3. ARQ 协议

ARQ 协议也就是超时重传机制。通过确认和超时机制保证了数据的正确送达， ARQ 协议包含停止等待 ARQ 和连续 ARQ 两种协议。

停止等待 ARQ

正常传输过程

只要 A 向 B 发送一段报文，都要停止发送并启动一个定时器，等待对端回应，在定时器时间内接收到对端应答就取消定时器并发送下一段报文。

报文丢失或出错

- 在报文传输的过程中可能会出现丢包。这时候超过定时器设定的时间就会再次发送丢失的数据直到对端响应，所以需要每次都备份发送的数据。
- 即使报文正常的传输到对端，也可能出现在传输过程中报文出错的问题。这时候对端会抛弃该报文并等待 A 端重传。
- PS：一般定时器设定的时间都会大于一个 RTT 的平均时间。

第三次握手

- B 如果此时还有没发完的数据会继续发送，完毕后会向 A 发送连接释放请求，然后 B 便进入 LAST-ACK 状态。
- PS：通过延迟确认的技术（通常有时间限制，否则对方会误认为需要重传），可以将第二次和第三次握手合并，延迟 ACK 包的发送。

第四次握手

A 收到释放请求后，向 B 发送确认应答，此时 A 进入 TIME-WAIT 状态。该状态会持续 2MSL（最大段生存期，指报文段在网络中生存的时间，超时会被抛弃）时间，若该时间段内没有 B 的重发请求的话，就进入 CLOSED 状态。当 B 收到确认应答后，也便进入 CLOSED 状态。

- 为什么 A 要进入 TIME-WAIT 状态，等待 2MSL 时间后才进入 CLOSED 状态？
- 为了保证 B 能收到 A 的确认应答。若 A 发完确认应答后直接进入 CLOSED 状态，如果确认应答因为网络问题一直没有到达，那么会造成 B 不能正常关闭。

ACK 超时或丢失