来完整演示一个中文句子在 **GPT 的 BPE（Byte Pair Encoding）分词**过程，
让你真正看到它是如何一步步把「汉字 → 字节 → token id」的。

示例句子

我们用一句最简单的中文：

```
我爱学习
```

Step 1：UTF-8 编码为字节（bytes）

中文字符在 UTF-8 下每个汉字通常占 3 个字节。

字	UTF-8 十六进制	十进制表示
我	E6 88 91	[230, 136, 145]
爱	E7 88 B1	[231, 136, 177]
学	E5 AD A6	[229, 173, 166]
习	E4 B9 A0	[228, 185, 160]

所以整个句子的字节流是：

```
[230,136,145,231,136,177,229,173,166,228,185,160]
```

也就是 11 个「初始 token」。

Step 2：BPE 的初始状态

BPE 一开始把每个字节都当作独立的「token」，
也就是从：

```
[230] [136] [145] [231] [136] [177] [229] [173] [166] [228] [185] [160]
```

Step 3：合并最常出现的字节对（pair）

BPE 在训练阶段统计了大量中文文本的字节对共现频率。
比如：

- `[230,136]` 这个组合几乎总是出现（因为"我"的前三个字节总是这两个）；
- `[136,145]` 也是；
- `[230,136,145]` → 出现频繁 → 学会把这三个合并成一个 token 表示"我"。

于是它在 merges.txt 里学到了：

```
230 136 → 230136
230136 145 → 230136145
```

分词器逐步执行合并：

```
[230,136,145] → token("我")
[231,136,177] → token("爱")
[229,173,166] → token("学")
[228,185,160] → token("习")
```

最终得到 token 序列：

```
["我", "爱", "学", "习"]
```

🗃️ Step 4：查 vocab.json 获取 token id

在 `vocab.json` 中，每个 token 都对应一个整数 ID，比如：

```
{
  "我": 32001,
  "爱": 37288,
  "学": 40123,
  "习": 40124
}
```

于是编码结果：

```
"我爱学习" → [32001, 37288, 40123, 40124]
```

🧠 Step 5：为什么 BPE 能学会「中文合并」

因为 BPE 训练时是**基于字节共现统计**的。
每个汉字对应固定的 3 个字节，它们总是成对出现。

> 所以模型很容易学到：
> “E6 88 91” 这三个字节频繁一起出现 → 合并成一个 token 表示「我」。

同样原理适用于任何语言（英文、中文、日文、emoji），
因为本质上 GPT 的分词器是在 **UTF-8 字节空间上做统计学习**。

📦 Step 6：完整流程总结图

```
原文本:  "我爱学习"
 ↓
UTF-8 字节流:
 [230,136,145,231,136,177,229,173,166,228,185,160]
 ↓
BPE 频率合并:
 [230,136,145] [231,136,177] [229,173,166] [228,185,160]
 ↓
合并后 token:
 ["我", "爱", "学", "习"]
 ↓
查 vocab.json 得到 ID:
 [32001, 37288, 40123, 40124]
```

`merges.txt` 不是直接列出“合并后的 token”，
而是列出 **“合并规则（pair merge rules）”**，
也就是：告诉 tokenizer 在编码时要**按什么顺序把小 token 合并成更大的 token**。

🧩 一句话定义

> `merges.txt` **是 BPE 的“合并配方表”，不是成品 token 列表。**

🧠 举个非常清晰的例子

假设训练时你的文本语料是：

```
low lower newest widest
```

初始时，每个字符都是一个 token：

```
l o w e r n e w e s t i d e s t
```

BPE 在统计后，发现这些字符对出现频率最高：

```
('l', 'o') → 50次
('lo', 'w') → 40次
('n', 'e') → 30次
('e', 's') → 20次
...
```

于是它在 `merges.txt` 中记录了"合并顺序"：

```
# merges.txt
l o
lo w
n e
ne w
e s
es t
```

👉 tokenizer 在使用时：

当它看到单词 `"lowest"` 时，会按顺序执行 merges.txt：

合并步骤	token 状态
初始	[l, o, w, e, s, t]
第1步 (l o)	[lo, w, e, s, t]
第2步 (lo w)	[low, e, s, t]
第3步 (e s)	[low, es, t]
第4步 (es t)	[low, est]

得到最终 token：

```
["low", "est"]
```

📓 然后 vocab.json 告诉你每个 token 的编号

```
{
  "l": 0,
  "o": 1,
  "w": 2,
  "lo": 3,
  "low": 4,
```

```
  "est": 5
}
```

于是：

```
"lowest" → [4, 5]
```

小结对比

文件	内容类型	示例	作用
`merges.txt`	BPE 合并顺序（规则）	`l o` `lo w`	指导 tokenizer 如何从字符逐步合并成子词
`vocab.json`	最终 token 与 ID 对照	`"low": 4`	用于把合并结果转成整数 ID
`tokenizer.json`	综合配置文件	包含 vocab + merges + 特殊符号	方便快速加载

再看中文情况

对于中文来说：

`merges.txt` 里可能有类似规则：

```
230 136
230136 145
231 136
231136 177
```

这些数字是字节（byte）的编码。
表示：

> 先把 230 和 136 合并
> 再把 230136 和 145 合并
> → 得到 “我”

所以，中文的「我」并不是直接出现在 merges.txt 里，
而是通过这些字节合并规则生成出来的。
最后再由 `vocab.json` 给出 `"我": 32001`。

☑ **总结一句话：**

> `merges.txt` 记录了“如何一步步合成 token”；
> `vocab.json` 记录了“这些 token 是什么、编号多少”；
> `tokenizer.json` 把两者打包起来方便加载。