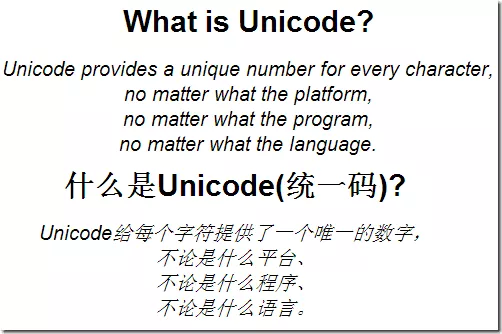
<https://mp.weixin.qq.com/s/QjU9lSekpbaF7fugZbyzkg>

# **什么是Unicode？**



Unicode编码定义了这个世界上几乎所有字符（就是你眼睛看的字符比如ABC，汉字等）的数字表示，而且Unicode还兼容了很多老版本的编码规范，例如你熟悉的 ASCII码。

# **什么是码点？**

我们国家的每一个人都对应唯一的一个身份证号，而Unicode也为了每个字符发了一张身份证，这张“身份证”上有一串唯一的数字ID确定了这个字符。

这串数字在整个计算机的世界具有唯一性，Unicode给这串数字ID起了个名字叫［码点］。

# **码点是如何表示的呢？**

先来说一声码点是如何表示的：

U+XXXXXX 是码点的表示形式，X 代表一个十六制数字，可以有 4-6 位，不足 4 位前补 0 补足 4 位，超过则按是几位就是几位。

字符A的ASCII码是众所周知是65吧，将65转换成16进制就是41（16×4+（16^0)×1 = 65）,按照规则前面补0，那么字符A的码点表示就是U+0041，依次类推B的码点表示就是U+0042...等等，汉字"你"的字符表示是“U+4F60”...

# **码点的取值范围**

码点的取值范围目前是 U+0000 ~ U+10FFFF，理论大小为 10FFFF+1=110000（为啥+1，因为从0开始嘛~）。

16进制嘛~后一个 1代表是 65536（16的4次方），因为是 16 进制，所以前一个 1 是后一个 1 的 16 倍，所以总共有1×16+1=17 个的 65536 的大小，粗略估算为 17×6万=102 万，所以这是一个百万级别的数。

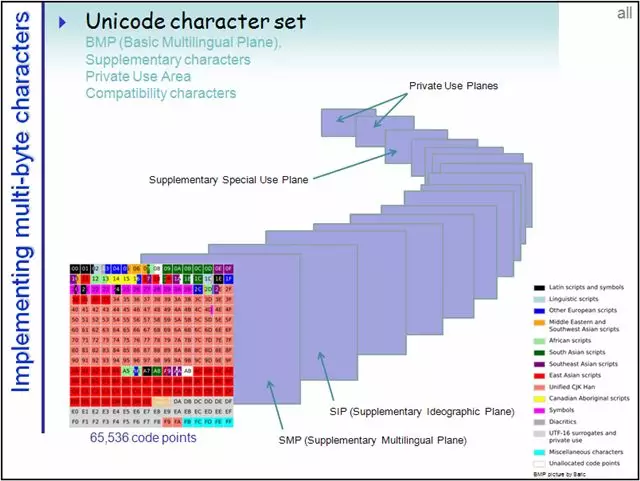
为了更好分类管理如此庞大的码点数，把每 65536 个码点作为一个**平面**，总共 17 个平面。

而我们说的代理区就在平面里面，而平面又有很多讲究。为了帮你搞懂代理区，先来聊一聊这平面的事。

## **平面，BMP，SP**

### **什么是平面？**

由前面可知，码点的全部范围可以均分成 17 个 65536 大小的部分，这里面的每一个部分就是一个**平面（Plane）**。编号从 0 开始，第一个平面称为 Plane 0。



### **什么是 BMP？**

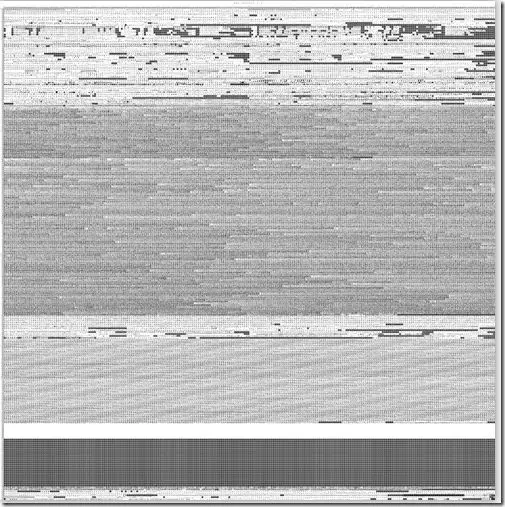
第一个平面即是 **BMP（Basic Multilingual Plane 基本多语言平面）**，也叫 Plane 0，它的码点范围是 U+0000 ~ U+FFFF。这也是我们最常用的平面，日常用到的字符绝大多数都落在这个平面内。

上图中第一个花花绿绿的平面就是 BMP。

UTF-16 只需要用两字节编码此平面内的字符。

最常用的 BMP，它的码点空间也有 6 万多，如果把这些字符都放到一张图片上，会是什么情况呢？GNU Unifont 就制作了一张这样的图片。见http://unifoundry.com/pub/unifont-7.0.03/unifont-7.0.03.bmp

下图是它的一个缩略版本：

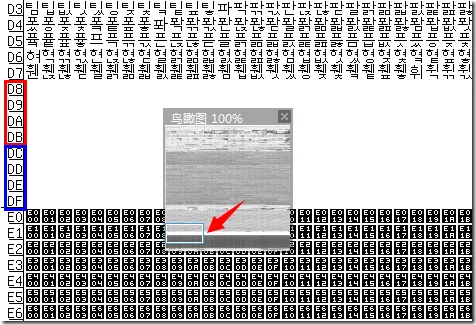


### **什么是SP（增补平面）？**

后续的 16 个平面称为 **SP（Supplementary Planes）**。显然，这些码点已经是超过 U+FFFF 的了，所以已经超过了 16 位空间的理论上限，对于这些平面内的字符，UTF-16 采用了四字节编码。

### **代理区**

你可能还注意到前面的 BMP 缩略图中有一片空白，这白花花一片亮瞎了我们的猿眼的是啥呢？这就是所谓的**代理区（Surrogate Area）**了。

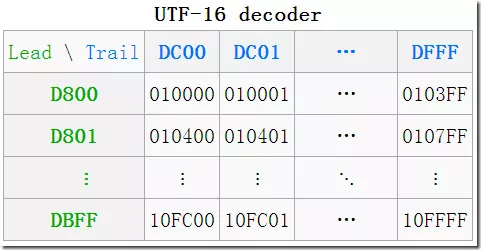


可以看到这段空白从 D8~DF。其中前面的红色部分 D800–DBFF 属于高代理区（High Surrogate Area），后面的蓝色部分 DC00–DFFF 属于低代理区（Low Surrogate Area），各自的大小均为 4×256=1024。

### **UTF-16如何用代理区编码？**

UTF-16 是一种变长的 2 或 4 字节编码模式。对于 BMP 内的字符使用 2 字节编码，其它的则使用 4 字节组成所谓的代理对来编码。

在前面的鸟瞰图中，我们看到了一片空白的区域，这就是所谓的**代理区（Surrogate Area）**了，代理区是 UTF-16 为了编码增补平面中的字符而保留的，总共有 2048 个位置，均分为**高代理区**（D800–DBFF）和**低代理区**（DC00–DFFF）两部分，各1024，这两个区组成一个二维的表格，共有1024×1024=210×210=24×216=16×65536，所以它恰好可以表示增补的 16 个平面中的所有字符。



#### **什么是代理对？**

一个高代理区（即上图中的Lead（头），行）的加一个低代理区（即上图中的Trail（尾），列）的编码组成一对即是一个**代理对（Surrogate Pair）**，必须是这种先高后低的顺序，如果出现两个高，两个低，或者先低后高，都是非法的。

在图中可以看到一些转换的例子，如

（D8 00 DC 00）—>U+10000，左上角，第一个增补字符

（DB FF DF FF）—>U+10FFFF，右下角，最后一个增补字符

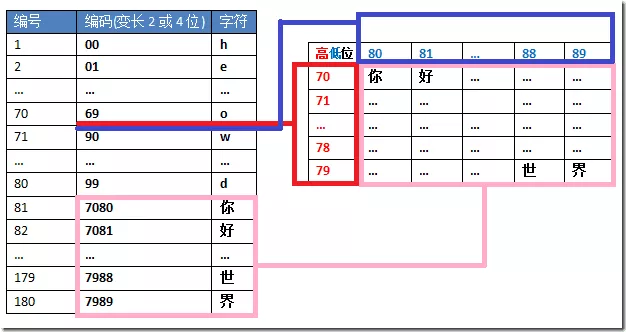
### **那UTF-16为何要采用代理对？**

最开始是采用定长二字节方案，但是无法满足容量增长，因为两个字节也就216 = 65536个而已，我们天朝的汉字就比这65536还多，那怎么办？扩呗~

于是转向定长四字节，但是转到4个字节虽然解决了容量的问题，又会引发了效率危机，比如一个字符A用一个字节就够存了，你非要用4个字节存，之前1G的·文件现在可能要4G去存，这不费钱吗~

那这咋办？于是各路大牛开天辟地，建立自己的编码方案，力图在效率和容量上取到一个平衡，其中一位大牛建立了UTF-16的编码方案！

看下面这个图，可以看到编码不是递增的，70-89的编码没有与之对应的字符。



这里挖出 70-89 间的码位，形成横竖 10×10 的编码空间，使得能**再扩展 100 个编码空间**。原来 2 位 100 个空间损失了 20，为啥这么说，因为70-89是20个，这部分不参与编码，那不就是少了20个吗

但是这20个编码通过形成 **代理对** 的方式又新增了100个代码空间，一来一回多了 80。这样一种变长方式也就是 UTF-16 所采用的。

小萌：哦，懂了~ 小萌：UTF-16相当于牺牲了**高代理区**（D800–DBFF）和**低代理区**（DC00–DFFF）两部分空间，但是确新增了1024\**1024=16\**65536的空间。依次来实现了扩容！

#### **码点到 UTF-16 如何转换？**

乔哥：继续上个例子。转换分成两部分：

1. BMP 中直接对应，无须做任何转换，也就是如果U<0x10000，U的UTF-16编码就是U对应的16位无符号整数；

2. 增补平面 SP 中，则需要做相应的计算。也就是如果U≥0x10000的情况

我们先计算U'=U-0x10000，然后将U'写成二进制形式：yyyy yyyy yyxx xxxx xxxx，U的UTF-16编码（二进制）就是：110110yyyyyyyyyy 110111xxxxxxxxxx。

Unicode编码0x20C30，减去0x10000后，得到0x10C30，写成二进制是：0001 0000 1100 0011 0000。用前10位依次替代模板中的y，用后10位依次替代模板中的x，就得到：1101100001000011 1101110000110000，转换为16进制即0xD843 0xDC30。

注意：以上计算方式仅用于说明转换原理，不代表实际采用的计算方式。

### **UTF-32**

我们说码点最大的 10FFFF 也就 21 位，而 UTF-32 采用的定长四字节则是 32 位，所以它表示所有的码点不但毫无压力，反而绰绰有余，所以只要把码点的表示形式以前补 0 的形式补够 32 位即可。这种表示的最大缺点是占用空间太大。

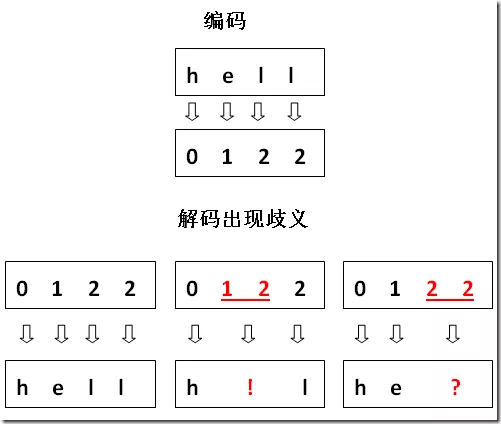
再来看稍复杂一点的 UTF-8。

### **UTF-8**

### **UTF-8的好处**



你的方案的想法很美好，它试图跟随编号来自然增长，它还是可以编码的，但在解码时则遇到了困难。



可见，由于低位的码位被“榨干”了，导致单个位与多位间无法区分，所以你的方案是行不通的。

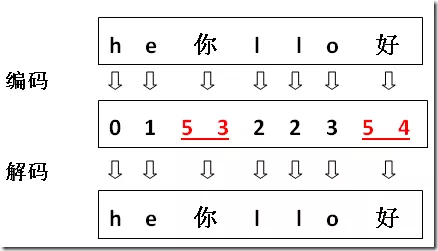
下图中的编码方案2是我的改进方案。



这是我的第二种编码方案，既然之前的无法区分，那我就把低位空间腾出来，5 及以上的就不使用了5,6,7...到49这些编码都不使用了，直接跳到50。然后引入一条变长解码规则：

从左向右扫描，读到 5 以下数字按单个位解码；读到 5 或以上数字时，把当前数字及下一个数字**两位一起**读上来解码。

看个实例

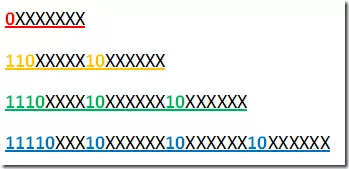


0和1是5以下的（5 以下数字按单个位解码），所以解码出来he,而当读取到了5（读到 5 或以上数字时，把当前数字及下一个数字**两位一起**读上来解码。），那么5和3连接起来就是53，查一下编码表53就是 “你”，这种方案避免了歧义。

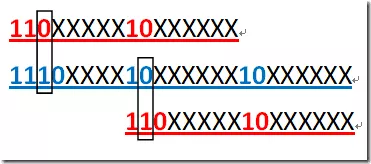
乔哥：这还是非常粗糙的设计，如果我们想在这串字符中搜索“o”这个字符，它的编码是 3， 首先会找到3和53，这样在匹配时也会匹配上 53 中的 3，这种设计会让我们在实现匹配算法时不好实现啊。

其实关键就在于用高位保留位来做区分，缺点就是有效编码空间少了

UTF-8 是变长的编码方案，可以有 1，2，3，4 四种字节组合。UTF-8 采用了高位保留方式来区别不同变长，如下：



可以看到，由于最高位不同，多字节中不会包含一字节的模式。对于 UTF-8 而言，二字节的模式也不会包含在三字节模式中，也不会在四字节中；三字节模式也不会在四字节模式中，这样就解决上面所说的搜索匹配难题。



可以看到，由于固定位上的 0 和 1 的差别，使得二字节既不会与三字节的前两字节相同，也不会它的后两字节相同。

这也每当进行搜索的时候，每个二字节和三字节的编码没有重叠，因为最高位不同呀~所以不会出现搜索同一个出现两个的结果。不过就是有效编码空间少了.

### **UTF-8如何与码点进行转换**

| Unicode编码(十六进制) | UTF-8 字节流(二进制) |
| --- | --- |
| 000000-00007F | 0xxxxxxx |
| 000080-0007FF | 110xxxxx 10xxxxxx |
| 000800-00FFFF | 1110xxxx 10xxxxxx 10xxxxxx |
| 010000-10FFFF | 11110xxx10xxxxxx10xxxxxx10xxxxxx |

对于Unicode的编码首先确定它的范围，找到它是对应的几字节[3]。

对于0x00-0x7F之间的字符，UTF-8编码与[ASCII编码]完全相同。

“汉”字的Unicode编码是0x6C49。0x6C49在0x0800-0xFFFF之间，使用3字节模板：1110xxxx 10xxxxxx 10xxxxxx。将0x6C49写成二进制是：0110 1100 0100 1001， 用这个比特流依次代替模板中的x，得到：**1110**0110 **10**110001 **10**001001，即E6 B1 89。