浅谈 web 前端开发中的国际化

I. 国际化、本地化、全球化

1.1 术语

提起国际化时，也常常牵扯出几个相似的术语：

- [国际化]： `internationalization` ，因首尾字母间有 18 个字母，简称为 `i18n` ；指的是将软件与特定语言及地区脱钩的过程。当软件被移植到不同的语言及地区时，软件本身不用做内部工程上的改变或修正
- [本地化]： `localization` ，由于同样的原因被简称为 `l10n` ；是指为特定区域翻译文件，并为了使软件能在该特定语言环境或地区使用，而应用特殊布局、加入本地特殊化部件等的过程
- [全球化]： `globalization` ，有时会用来表示以上两者的合称；也会简称为 `g11n`

简单来说：国际化搭台、本地化唱戏。国际化意味着产品有适用于任何地方的 “潜力”；本地化则是为了更适合于 “特定” 地方的使用，而另外增添的特色。用一项产品来说，国际化只需做一次，但本地化则要针对不同的区域各做一次。两者之间是互补的，并且合起来才能让一个系统适用于各地。

1.2 国际化的特征

一个经过国际化的软件具备如下特征：

- 可以快速的本地化
- 借助本地化的数据，相同的程序可以运行在世界各地
- 诸如状态提示、界面中的标题等文本元素，并非硬编码到软件中，而是存储到源代码之外，且能动态改变

- 不需要重新编译软件就能添加新的语言
- 货币、日期等信息的表现形式，适应于用户所在的地区和语言

1.3 需要本地化的元素

除了以上提到的货币和日期，还有很多元素与文化、地域、语言相关，比如：

- 书写方向、声音、颜色、图形、图标、时间、数字、度量、电话号码、敬语、头衔、邮政地址、分页、排序方法、输入处理、方言、法规、道德和习惯

II. 区域设置

除了 `i18n` 、 `l10n` 、 `g11n` 等这些奇怪的叫法，我们日常也经常见到 `locale` 这个词，其取值一般为 `zh-CN` 、 `en-US` 等。

> 一个 `locale` 对象就是个结合了语言和地区的特殊标识符

`locale` 由 "互联网工程任务组"（IETF）的 “BCP 47” 文档系列（ tools.ietf.org/html/bcp47 ） 定义。

其 常见的典型形式 由分别表示语言和地区的两部分组成，用 `-` 连接；也可以省略地区。

举例来说：

locale code	通常的含义
en	英语
en-US	美国讲的英语
en-GB	英国讲的英语
zh-CN	简体中文
zh-HK	香港地区繁体中文

locale code	通常的含义
zh-TW	台湾地区繁体中文
zh-SG	新加坡简体中文
zh-MO	澳门地区繁体中文
es-AR	阿根廷讲的西班牙语
ar-001	通用阿拉伯语
ar-AE	阿联酋讲的阿拉伯语

2.1 区域敏感

一般来说，根据 locale 的设置，把一个 `hello` 分别翻译成 “こんにちは” 或 “你好” 就可以了；但涉及数字、日期、货币等，需要特殊的格式，或计算年号等，可以用一些专用的类来处理。

> 如果代码的行为取决于 locale，则说它是 “区域敏感的（locale-sensitive）”

比如，Java 中的 NumberFormat 类就是区域敏感的；其数字返回值的格式取决于 `locale` ：可能为 `902 300` (法国)，或 `902.300` (德国)，又或者 `902,300` (美国)。

需要注意的是， `locale` 就只是个标识符，而识别单词边界、格式化等具体的工作，还是都需要由区域敏感的代码模块来完成的。

2.2 语言编码

locale 的前半部分表示语言，通常由 2 或 3 位小写字母组成，符合 `ISO 639` （ www.loc.gov/standards/i... 标准。

例如：

Language Code	Description
de	German

Language Code	Description
en	English
fr	French
ru	Russian
ja	Japanese
jv	Javanese
ko	Korean
zh	Chinese

2.3 地区编码

locale 的后半部分表示地区，由符合 `ISO 3166` （ www.chemie.fu-berlin.de/diverse/doc... 标准的 2 或 3 位大写字母，或符合 `UN M.49` 标准的 3 位数字组成。这部分是可选的。

例如：

A-2 Code	A-3 Code	Numeric Code	Description
AU	AUS	036	Australia
BR	BRA	076	Brazil
CA	CAN	124	Canada
CN	CHN	156	China
DE	DEU	276	Germany
FR	FRA	250	France
IN	IND	356	India
RU	RUS	643	Russian Federation

A-2 Code	A-3 Code	Numeric Code	Description
US	USA	840	United States

III. 字符编码

计算机在设计之初，并没有考虑多个国家、多种不同语言的应用场景。当时定义一种 `ASCII` 码，将字母、数字和其他符号编号用 7 比特的二进制数来表示。

后来，计算机在世界范围内开始普及，为了适应多种文字，出现了多种编码格式，例如中文汉字一般使用的编码格式为 `GB2312` 、 `GBK` 。

由此，又产生了一个问题，不同字符编码之间互相无法识别。于是出现了 `Unicode` 编码。它为每种语言的每个字符设定了统一并且唯一的二进制编码。

但 `Unicode` 也有一个缺点：为了支持所有语言的字符，所以它需要用更多位数去表示，比如 `ASCII` 表示一个英文字符只需要一个字节，而 `Unicode` 则需要两个字节。很明显，字符数多，效率会很低。

为了解决这个问题，由出现了一些中间格式的字符编码：如常见的 `UTF-8` ，以及 `UTF-16` 、 `UTF-32` 等。

V. jQuery 中的国际化

作为早期推动 web 前端开发的最主要工具，jQuery 是一个时代当仁不让的开发标配。

在处理国际化需求时，jQuery 本身并不包含相关模块，应用比较广泛的一个第三方轻量化插件是 `jQuery.i18n.properties` 。

5.1 `jQuery.i18n.properties` 的特点

- 与 Java 里的做法如出一辙的是，`jQuery.i18n.properties` 采用 `.properties` 文件对 JavaScript 代码进行国际化。要在 Java 程序和前端 JavaScript 程序中共享资源文件时，这种方式也提供了便利。
- 资源文件命名为 `<资源名>_<语言代码>-<国家/地区编码>.properties` ，其中语言和区域之间采用 `-` 连接，目的是为了兼容浏览器由 `navigator.language` 或 `jQuery.i18n.browserLang()` 获得的 locale 值
- 可以在资源字符串中使用占位符（例如：hello= 你好 {0}! 今天是 {1}。 ）
- 资源文件中的 Key 支持命名空间（例如：com.company.msgs.hello = Hello! ）
- 支持资源文件中跨行的值
- 支持以 JavaScript 变量（或函数）或 Map 的方式使用资源文件中的 Key

5.2 基本用法

```
jQuery.i18n.properties({
   name:'strings',// 资源文件名称
   path:'bundle/',// 资源文件所在目录路径
   mode:'both',// 模式：变量或 Map
   language:'pt_PT',// 对应的语言
   cache:false, //浏览器是否对资源文件进行缓存
   encoding: 'UTF-8', //编码。默认为 UTF-8
   callback: function() {// 回调方法
   }
});
复制代码
```

在 callback 回调中，可以调用 `• jQuery.i18n.prop(key)` 方法。该方法以 map 的方式使用资源文件中的值，其中 key 指的是资源文件中的 key。当 key 指定的值含有占位符时，可以使用 `jQuery.i18n.prop(key,var1,var2 … )` 的形式，其中 var1,var2 ... 对各占位符依次进行替换。

VI. Vue.js 中的国际化

随着 Angular、React、Vue.js 具有指标性的开发工具相继问世，这三驾马车把前后端分离也带入了新的时代，更多的开发逻辑从后端让渡到前端来，相应的国际化需求也更为常见，对其灵活性易用性的要求也更高。

此处以 Vue.js 为例分析其常见的 i18n 解决方案。

`vue-i18n`（kazupon.github.io/vue-i18n/）是较常用的一款 Vue.js 国际化插件，其主要特性包括：

- 各种格式的本地化
- 支持复数等语言规则 (Pluralization)
- DateTime 本地化
- Number 本地化
- 基于组件的本地化
- 组件插值 (Component interpolation)
- Fallback 本地化

6.1 i18next

另一个优秀的 Vue.js 国际化插件是 `i18next`（github.com/i18next/i18...），本文不再展开介绍

VII. 新的 JS 国际化 API

在 2012 年末，ECMA International 推出了 Standard ECMA-402 标准的首个版本，也就是广为人知的 ECMAScript Internationalization API。这个标准弥补了 ECMAScript 中早该支持的本地化方法。基本上所有现代浏览器都已经支持该 API。

7.1 Intl 全局对象

`Intl` 对象是 ECMAScript 国际化 API 的一个命名空间，它提供了精确的字符串对比，数字格式化，日期和时间格式化。 `Collator` ，`NumberFormat` 和 `DateTimeFormat` 对象的构造函数是 `Intl` 对象的属性。

- `Intl.Collator` collators 的构造函数，用于启用对语言敏感的字符串比较的对象。
- `Intl.DateTimeFormat` 用于启用语言敏感的日期和时间格式的对象的构造函数。
- `Intl.NumberFormat` 用于启用语言敏感数字格式的对象的构造函数。
- `Intl.PluralRules` 用于启用多种敏感格式和多种语言语言规则的对象的构造函数。

正如我们在 2.3 中已经见过的 `DateTimeFormat` 例子，这几种构造函数都使用同样的模式来识别语言区域和确定使用哪一种语言格式：它们都接收 locales 和 options 参数。

`options` 参数必须是一个对象，其属性值在不同的构造函数和方法中会有所变化。如果 `options` 参数未提供或者为 `undefined` ，所有的属性值则使用默认的。

再看几个简单的例子：

```
var number = 123456.789;

// 德语使用逗号作为小数点,使用.作为千位分隔符
console.log(new Intl.NumberFormat('de-DE').format(number));
// → 123.456,789

// 通过编号系统中的nu扩展键请求, 例如中文十进制数字
console.log(new Intl.NumberFormat('zh-Hans-CN-u-nu-hanidec').format(number));
// → 一二三,四五六.七八九


// 德语中, ä 使用 a 的排序
console.log(new Intl.Collator('de').compare('ä', 'z'));
// → -1

// 瑞典语中, ä 在 z 的后面
console.log(new Intl.Collator('sv').compare('ä', 'z'));
```

```
// → 1


var date = new Date(Date.UTC(2012, 11, 20, 3, 0, 0));

// 使用24小时制
options = {
  year: 'numeric', month: 'numeric', day: 'numeric',
  hour: 'numeric', minute: 'numeric', second: 'numeric',
  hour12: false
};
console.log(date.toLocaleString('en-US', options));
// → "12/19/2012, 19:00:00"
复制代码
```

7.2 intl.js polyfill

对于一些没有原生 Intl API 的老旧浏览器，可以使用 github.com/andyearnsha... 达到大部分功能的支持。

7.3 附：一个取得农历日期的方法

在新的 API 出现之前，计算农历是一件困难的事情，且有最大年份限制；采用新的 API 可以很好的解决这个问题。

以下方法改良自 jsfiddle.net/DerekL/mGXK...

```
function getLunarDate(date) {
  const TIAN_GAN = ["甲", "乙", "丙", "丁", "戊", "己", "庚", "辛", "壬", "癸"];
  const DI_ZHI = ["子", "丑", "寅", "卯", "辰", "巳", "午", "未", "申", "酉", "戌
  const SHI = ["初", "十", "廿", "三"];
  const YUE = ["", "十"];
  const GE = ["一", "二", "三", "四", "五", "六", "七", "八", "九", "十"];

  const locale = "zh-TW-u-ca-chinese";
  const fmt = (key, d=null)=>{
        return Intl.DateTimeFormat(locale,{[key]:"numeric"}).format(d||date).matc
  };
  const isLeapMonth = (d)=>{
        let _date = new Date(date);
    _date.setDate(-d);
    return fmt("month", _date) === m;
  };

  let y = fmt("year");
```

```
  let m = fmt("month");
  let d = fmt("day");

  isL = isLeapMonth(d);

  y = TIAN_GAN[(y - 1) % 10]
        + DI_ZHI[(y - 1) % 12];
  m = (YUE[(m - 1) / 10 | 0]
        + GE[(m - 1) % 10]).replace(/^一$/, "正");
  d = (SHI[(d) / 10 | 0]
        + GE[(d - 1) % 10]).replace(/^十十$/, "初十").replace(/^廿十$/, "二十");

  return y + "年" + (isL ? "閏" : "") + m + "月" + d;
}

var date = new Date(1945,7,15);
var lunar = getLunarDate(date);
console.log(lunar); //乙酉年七月初八
```