

利用高德 API 自动生成行政区划地图

远举云中

介绍

在利用地图开展各种分析或制图时，准确的行政区划底图往往是不可或缺的。但由于我国目前行政区划（特别是区县级）调整比较频繁，互联网上的免费资源通常比较陈旧，只用这样的底图往往就不能准确反映现实情况了。好在高德地图提供了行政区域查询的 WEB 服务 API 接口，可以通过这个接口查询我国各级行政区的行政区划地理空间信息，并利用返回的点坐标生成各级行政区划图。这样就可以利用高德地图的数据生产比较新的行政区划图了，而且高德说是“唯一能让用户查询到乡镇/街道级别信息且小时级更新数据的公开 API”。惊喜不惊喜，意外不意外？废话不多说，下面我们就用 R 语言来即时生成行政区划底图吧。

抓取信息

```
library('httr')
library('jsonlite')
library('tidyverse')
library('rlist')
library('Rgctc2', lib.loc = '~/GitHub/R_coordination_transformation')
library('sf')
library('showtext')
```

首先自然是写抓取信息的函数，利用高德的官方指南很容易搞定。

```
options(digits=11)
get_location<- function(address){
  key = '7c6b6c0d1b641f4aa9cdb7d2229ae728' # 需要预先申请一个高德 API 的 key
  url = 'http://restapi.amap.com/v3/config/district?' %>%
    paste('keywords=' , address ,
          '&key=' ,key ,
          '&subdistrict=1' , # 可以指定返回行政区的层级
          '&extensions=all',
          sep = '')
  geoinfo<-GET(url)%>% content(as="text",encoding="UTF-8") %>%
    fromJSON(flatten = TRUE) # 返回信息的类型是可以选择的
  return(geoinfo)
}
```

这个爬虫就只需要一个参数 *address*，支持中文地名，也支持数字形式的区域编码（adcode）。用中文地名查询时到了市、县这个层级就有可能碰到多义性的问题了，也就是一个关键字对应多个区域的情况，高德建议大家尽量使用 adcode，可以在这里下载。我们可以通过上级行政区域获得所有下级行政区域的 adcode，非常简单，可以用这个机制直接抓取指定行政区里所有下级行政区的 adcode。值得一提的是预定

设定小数点位数 `options(digits=11)`。由于地理坐标小数点后的位数比较多，如果不预先指定小数点位数的话，按照 R 默认的精度就会出现显示不完整的情况。下面，我们赶紧来拿浙江做个例子吧。

```
zj<-get_location(' 浙江省') %>% '['('districts')
```

返回的信息是一个包含了六个元素的列表，这个列表的 *districts*元素包含了我所需要的所有信息，所以这里就直接一步提取出来了，有兴趣的小伙伴可以自己查看返回列表的信息。下面我们就来看看这个 *districts*元素长什么样子吧。

```
## 'data.frame': 1 obs. of 7 variables:
## $ citycode :List of 1
## ..$ : list()
## $ adcode : chr "330000"
## $ name : chr "浙江省"
## $ polyline : chr "121.134149,27.786010;121.134130,27.785898;121.134110,27.785840;121.134079,27.785817"
## $ center : chr "120.152585,30.266597"
## $ level : chr "province"
## $ districts:List of 1
## ..$ : 'data.frame': 11 obs. of 6 variables:
## .. ..$ citycode : chr "0572" "0571" "0573" "0579" ...
## .. ..$ adcode : chr "330500" "330100" "330400" "330700" ...
## .. ..$ name : chr "湖州市" "杭州市" "嘉兴市" "金华市" ...
## .. ..$ center : chr "120.086809,30.89441" "120.209789,30.24692" "120.75547,30.746191" "119.647229,29.71297" ...
## .. ..$ level : chr "city" "city" "city" "city" ...
## .. ..$ districts:List of 11
## .. .. ..$ : list()
## .. .. ..$ : list()
## .. .. ..$ : list()
## .. .. ..$ : list()
## .. .. ..$ : list()
## .. .. ..$ : list()
## .. .. ..$ : list()
## .. .. ..$ : list()
## .. .. ..$ : list()
## .. .. ..$ : list()
## .. .. ..$ : list()
## .. .. ..$ : list()
```

为了比较清晰的展示数据结构，我选择只返回了下一级行政区的信息（`subdistrict=1`），也就是浙江的地级市信息。在实际中我设定 `subdistrict=3`，可以一步返回浙江所有镇的信息。*polyline*这个元素里就是我们要找的行政区边界上的坐标点的信息，目前是一个很长的字符串，后面主要的工作其实就是分割——转换类型——转换坐标系——生成地理空间对象，然后就可以作图啦。下面，我们就来处理浙江省的边界。

```
## chr "121.134149,27.786010;121.134130,27.785898;121.134110,27.785840;121.134079,27.785817;121.134009,27.785788;121.133980,27.785760;121.133951,27.785732;121.133922,27.785704;121.133893,27.785676;121.133864,27.785648;121.133835,27.785620;121.133806,27.785592;121.133777,27.785564;121.133748,27.785536;121.133719,27.785508;121.133690,27.785480;121.133661,27.785452;121.133632,27.785424;121.133603,27.785396;121.133574,27.785368;121.133545,27.785340;121.133516,27.785312;121.133487,27.785284;121.133458,27.785256;121.133429,27.785228;121.133400,27.785200;121.133371,27.785172;121.133342,27.785144;121.133313,27.785116;121.133284,27.785088;121.133255,27.785060;121.133226,27.785032;121.133197,27.785004;121.133168,27.784976;121.133139,27.784948;121.133110,27.784920;121.133081,27.784892;121.133052,27.784864;121.133023,27.784836;121.132994,27.784808;121.132965,27.784780;121.132936,27.784752;121.132907,27.784724;121.132878,27.784696;121.132849,27.784668;121.132820,27.784640;121.132791,27.784612;121.132762,27.784584;121.132733,27.784556;121.132704,27.784528;121.132675,27.784500;121.132646,27.784472;121.132617,27.784444;121.132588,27.784416;121.132559,27.784388;121.132530,27.784360;121.132501,27.784332;121.132472,27.784304;121.132443,27.784276;121.132414,27.784248;121.132385,27.784220;121.132356,27.784192;121.132327,27.784164;121.132298,27.784136;121.132269,27.784108;121.132240,27.784080;121.132211,27.784052;121.132182,27.784024;121.132153,27.783996;121.132124,27.783968;121.132095,27.783940;121.132066,27.783912;121.132037,27.783884;121.132008,27.783856;121.131979,27.783828;121.131950,27.783800;121.131921,27.783772;121.131892,27.783744;121.131863,27.783716;121.131834,27.783688;121.131805,27.783660;121.131776,27.783632;121.131747,27.783604;121.131718,27.783576;121.131689,27.783548;121.131660,27.783520;121.131631,27.783492;121.131602,27.783464;121.131573,27.783436;121.131544,27.783408;121.131515,27.783380;121.131486,27.783352;121.131457,27.783324;121.131428,27.783296;121.131399,27.783268;121.131370,27.783240;121.131341,27.783212;121.131312,27.783184;121.131283,27.783156;121.131254,27.783128;121.131225,27.783100;121.131196,27.783072;121.131167,27.783044;121.131138,27.783016;121.131109,27.782988;121.131080,27.782960;121.131051,27.782932;121.131022,27.782904;121.130993,27.782876;121.130964,27.782848;121.130935,27.782820;121.130906,27.782792;121.130877,27.782764;121.130848,27.782736;121.130819,27.782708;121.130790,27.782680;121.130761,27.782652;121.130732,27.782624;121.130703,27.782596;121.130674,27.782568;121.130645,27.782540;121.130616,27.782512;121.130587,27.782484;121.130558,27.782456;121.130529,27.782428;121.130500,27.782400;121.130471,27.782372;121.130442,27.782344;121.130413,27.782316;121.130384,27.782288;121.130355,27.782260;121.130326,27.782232;121.130297,27.782204;121.130268,27.782176;121.130239,27.782148;121.130210,27.782120;121.130181,27.782092;121.130152,27.782064;121.130123,27.782036;121.130094,27.782008;121.130065,27.781980;121.130036,27.781952;121.130007,27.781924;121.129978,27.781896;121.129949,27.781868;121.129920,27.781840;121.129891,27.781812;121.129862,27.781784;121.129833,27.781756;121.129804,27.781728;121.129775,27.781700;121.129746,27.781672;121.129717,27.781644;121.129688,27.781616;121.129659,27.781588;121.129630,27.781560;121.129601,27.781532;121.129572,27.781504;121.129543,27.781476;121.129514,27.781448;121.129485,27.781420;121.129456,27.781392;121.129427,27.781364;121.129398,27.781336;121.129369,27.781308;121.129340,27.781280;121.129311,27.781252;121.129282,27.781224;121.129253,27.781196;121.129224,27.781168;121.129195,27.781140;121.129166,27.781112;121.129137,27.781084;121.129108,27.781056;121.129079,27.781028;121.129050,27.781000;121.129021,27.780972;121.128992,27.780944;121.128963,27.780916;121.128934,27.780888;121.128905,27.780860;121.128876,27.780832;121.128847,27.780804;121.128818,27.780776;121.128789,27.780748;121.128760,27.780720;121.128731,27.780692;121.128702,27.780664;121.128673,27.780636;121.128644,27.780608;121.128615,27.780580;121.128586,27.780552;121.128557,27.780524;121.128528,27.780496;121.128499,27.780
```

```

zj$polyline<-zj$polyline %>%
  str_split('\\|') %>%
  lapply(str_split(';') %>%
    '[ '(1)%>% lapply(str_split(',') %>%
      lapply(lapply,as.numeric) %>%
      lapply(list.rbind)%>%
      lapply(gcj02_wgs84_matrix_matrix) %>%
      lapply(list) %>%
      st_multipolygon %>% st_sfc(crs=4326)
zj_sf<-st_sf(zj)

```

- 列表操作 **rlist**包。这是 R 语言里对 list 对象进行操作的神器。由于 list 对象的非结构性，并且可以多层嵌套，所以一个字符串经过多次分割后就形成了有三四个层次深度嵌套的 list 对象，后面的工作无论是数据类型转换还是坐标系的转换都涉及到对 list 对象的深层操作。我基本上用的是 **base**包里的 **lapply**函数嵌套以应对，但是也少不了要借助 **rlist**包很多函数。这段最后的代码里真正属于 **rlist**包的函数实际上只有 **list.rbind**一个了，但是实际上在调试的过程中我借助了很多这个包里的函数进行 debug。如果对抓取网络地理信息数据有兴趣的话，应该都是要对抓下来的数据首先进行一番这样的操作的。
- 地理坐标转换 **Rgctc2**包。出于保密的需要，高德、百度提供的坐标信息都是经过转换加密的，一般转换成 wgs84 投影坐标系通用性会更好。这种转换网上有各种语言的源代码也有各种小工具，但是目前 R 语言里好像还没有专门用于这个转换的包。于是我就自己简单写了一个包，可以在这里下载。这个包功能很简单，目前就只能从高德转换到 WGS84（完整的功能应该能实现高德、百度、WGS84 中任意两种坐标系互转，而且在 R 中要用的方便话还要考虑各种数据类型的输入输出，所以还是个蛮大的工程，如果后面有兴趣就慢慢完善吧。因为不会，所以还没有搞帮助文档。）**gcj02**就是高德采用的坐标系，**wgs84**是我们转换成的通用坐标系，**matrix_matrix**分别是输入和输出的数据类型。这个包里所有的函数基本上都是这个命名规则。
- 生成地理空间对象 **sf**包。它是 **sp**包的升级换代，相比于传统的 **sp**包属性层和地理空间信息层分隔的复杂存储方法，**sf**包就是基于数据框的，非常便于在 R 语言中进行操作，也是更加主流的 GIS 数据存储方式。由于目前得到了 **ggplot2** 的支持，专门加了一个对象 **geom_sf**用于 **sf**对象的作图，所以目前看来前途非常光明啊。这里可以再看一下 **zj** 这个数据框，实际上只要把 **polyline**这一列转换成 **sf**对象必需的 **sfc**列（也就是专门用于存储地理空间信息的列），整个 **zj** 数据框就可以定位为 **sf**对象了，非常方便。**st_sfc(crs=4326)**这个命令就是用来转换 **polyline**列的。转换完成后我们再看 **polyline**列的

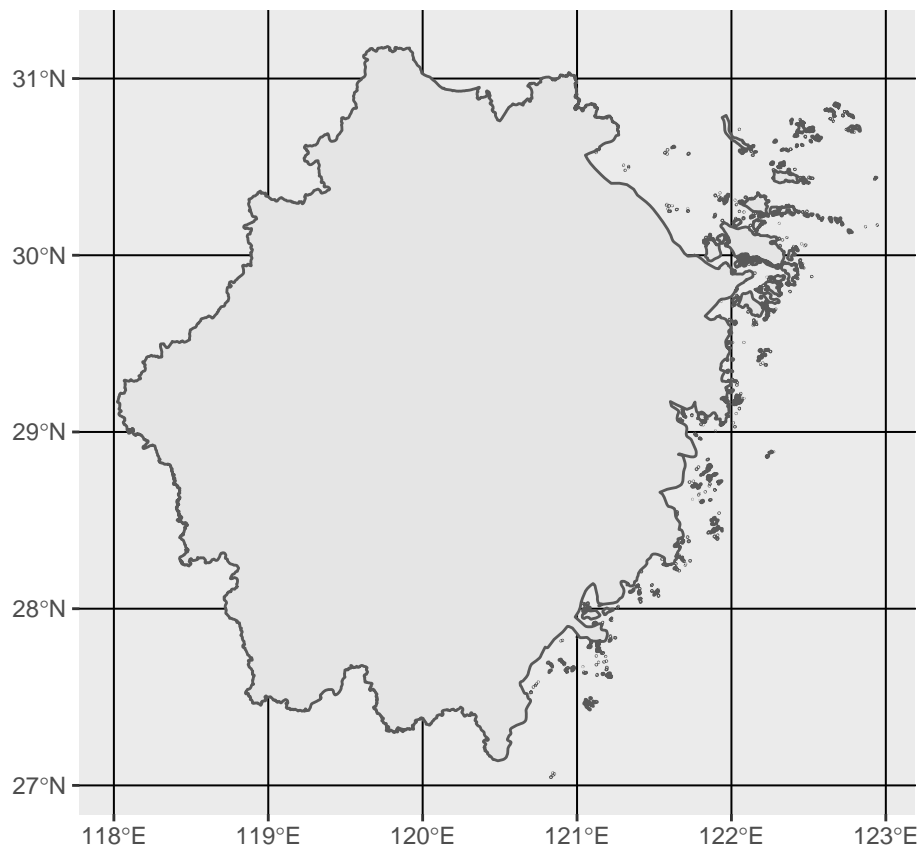


Figure 1: 浙江省行政区图

属性

```
## [1] "sfc_MULTIPOLYGON" "sfc"
```

可以看到已经成为 `sfc` 列了。

不方便的地方在于，不同的地理空间对象类型（就是点、线面）所对应的 `sfc` 列对数据格式的要求不同而且比较严格，比如说我这里所采用的 `multipolygon` 这个对象，他只接受 `list(list(matrix), list(matrix), ...)` 这种格式的数据，而这里的 `matrix` 是由坐标对构成的矩阵，所以也就是 `matrix(c(lng1, lat2), c(lng2, lat3), ...)`，是一个一维的矩阵，在 R 里面生成这样一个矩阵还是有点技巧的，需要一定时间来熟悉这个包的特殊用法。### 成图最艰难的阶段已经过去啦，下面就是愉快的画图啦。加载 `ggplot2`，一行命令就出图。

```
library(ggplot2)
ggplot()+geom_sf(data=zj_sf)
```

延伸

通过以上的办法，我们可以一个行政区一个行政区的得到他们的边框，然后通过 `rbind` 得到更大区域的图了。然而，这样的方式是太 low 了。我们要充分利用 R 语言里的 `apply` 函数家族，一下子得到浙江省所有地级市的行政区图。直接上代码。

```

zj_city<-zj %>% "["('districts') %>% '['(1) %>% '['(1) # 提取各城市 adcode
zj_city<-lapply(zj_city$adcode,get_location) %>% # 利用 lapply 函数提取所有城市的地理信息
  list.map(districts) %>%
  lapply(select,-districts) %>%
  list.rbind
zj_city$polyline <- zj_city$polyline %>%
  str_split('\\|') %>%
  lapply(str_split(';') %>%
    lapply(lapply,str_split(',') %>%
      lapply(lapply,lapply,as.numeric) %>%
      lapply(lapply,list.rbind) %>%
      lapply(lapply,gcj02_wgs84_matrix_matrix) %>%
      lapply(lapply,list) %>%
      lapply(st_multipolygon) %>%st_sf(crs=4326)
zj_city <- st_sf(zj_city)

```

可以看出来，最后一段代码不过是又套了一层 lapply 而已。在这里我们先不着急出图，我们把各城市的名字标出来。位置就放在高德给出的城市中心位置，原汁原味吗。这里又要进行一下坐标的转换。

```

zj_city      <-zj_city$center %>%
  str_split(';') %>%
  lapply(str_split(',') %>%
    lapply(lapply,as.numeric) %>%
    lapply(list.rbind) %>% list.rbind %>%
    gcj02_wgs84_matrix_df %>%
    bind_cols(zj_city)
zj_city      <-st_sf(zj_city)

```

我们来检验一下最终成果吧。

```

showtext_begin() # 由于 ggplot2 对中文支持不友好，这里需要加载一个显示中文的包
font_add('fzbs',regular='C:/Windows/Fonts/方正小标宋 _GBK.TTF')
ggplot() + geom_sf(data=zj_city) +geom_text(data=zj_city,aes(x=wgs84_lng,y=wgs84_lat,label=name),family=
showtext_end()

```

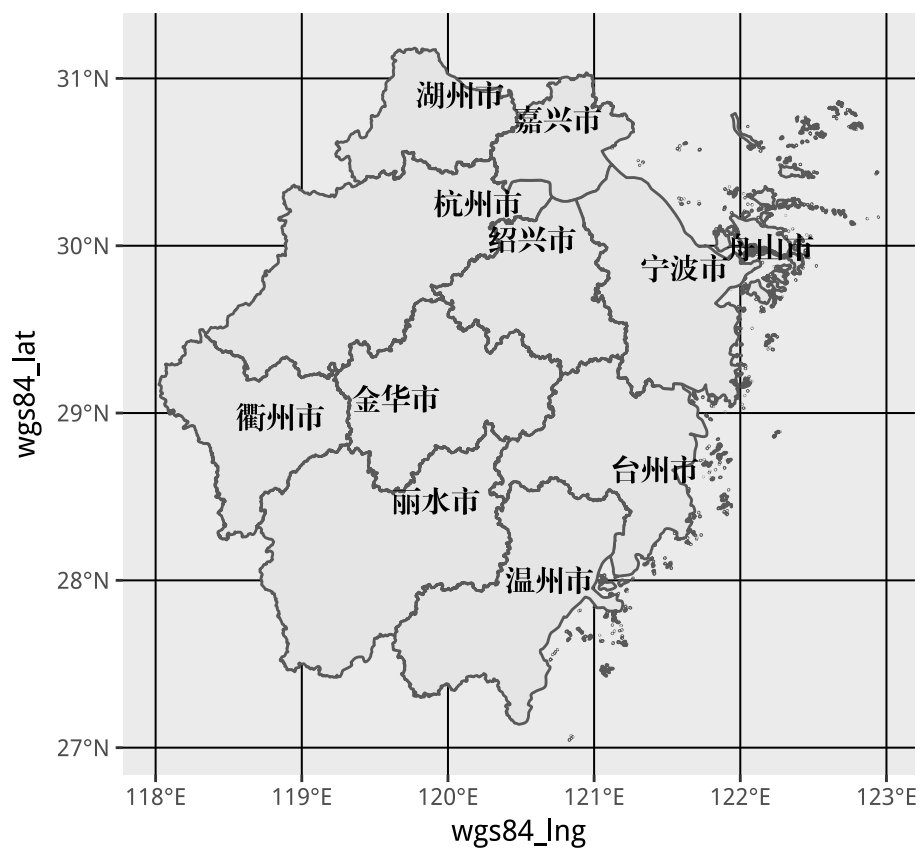


Figure 2: 浙江省分市行政区划图