# Geocomputation with R

Robin Lovelace, Jakub Nowosad, Jannes Muenchow

2024-03-22

# 目录

**第一章 简介** **1**

1.1 什么是地理计算? . . . . . 1

1.2 为什么使用 R 语言进行地理计算? . . . . . 1

1.3 地理计算软件 . . . . . 1

1.4 R 语言中地理计算的软件生态 . . . . . 1

1.5 R 语言地理计算的发展史 . . . . . 1

1.6 练习 . . . . . 1

**第二章 R 语言中的地理数据** **5**

2.1 导读 . . . . . 6

2.2 矢量数据 . . . . . 7

2.2.1 简单要素介绍 . . . . . 9

2.2.2 为什么使用简单要素? . . . . . 13

2.2.3 基本地图制作 . . . . . 14

2.2.4 基本绘图参数 . . . . . 15

2.2.5 几何类型 . . . . . 17

2.2.6 简单要素几何（sfg） . . . . . 20

2.2.7 简单要素列（sfc） . . . . . 22
2.2.8 sf 类 . . . . . 25
2.3 栅格数据 . . . . . 27
2.3.1 栅格数据简介 . . . . . 28
2.3.2 基本地图制作 . . . . . 29
2.3.3 栅格类 . . . . . 30
2.4 坐标参考系 . . . . . 33
2.4.1 地理坐标系 . . . . . 33
2.4.2 投影坐标参考系统 . . . . . 34
2.4.3 R 中的 CRS . . . . . 34
2.5 测量单位 . . . . . 37
2.6 练习 . . . . . 39

*For Katy*

*Dla Jagody*

*Für meine Katharina und alle unsere Kinder*

# 前言

# 序

## 如何阅读这本

## 为什么选择 R 语言?

## 致谢

# 第一章　简介

## 1.1　什么是地理计算?

## 1.2　为什么使用 R 语言进行地理计算?

## 1.3　地理计算软件

## 1.4　R 语言中地理计算的软件生态

## 1.5　R 语言地理计算的发展史

## 1.6　练习

# (第一部分) 基础

# 第二章　R 语言中的地理数据

## 前提要求

这是本书的第一个实操章节，因此需要安装一些软件。我们假设你已经安装了最新版本的 R，并且熟悉使用带有命令行界面的软件，例如集成开发环境（IDE）RStudio。

如果你是 R 的初学者，我们建议你阅读 Gillespie and Lovelace (2016) 的在线书籍 *Efficient R Programming* 第 2 章，并参考 Grolemund and Wickham (2016) 或 DataCamp[1] 等资源学习 R 语言的基础知识。请管理好你的学习成果（比如创建多个 RStudio 项目），并给脚本起有意义的名字如 `chapter-02.R`，以记录你学习时编写的代码。

本章中使用的软件包可以使用以下命令进行安装：[2]

```
install.packages("sf")
install.packages("raster")
install.packages("spData")
devtools::install_github("Nowosad/spDataLarge")
```

如果你使用的是 Mac 或 Linux 系统，安装 **sf** 可能会遇到些麻烦。这些操作系统需要先安装“系统依赖”，这些依赖在包的 README[3] 中有描述。可以在网上找

[1] https://www.datacamp.com/courses/free-introduction-to-r

[2] **spDataLarge** 不在 CRAN 上，因此必须使用 **devtools** 或以下命令进行安装 `install.packages("spDataLarge", repos = "https://nowosad.github.io/drat/", type = "source")`.

到各种特定于操作系统的说明，例如博客 rtask.thinkr.fr 上的文章 Installation of R 3.5 on Ubuntu 18.04[4]。

要复现本书内容所需的所有软件包，可以用以下命令进行安装：

`devtools::install_github("geocompr/geocompkg")`。这些软件包可以用以下方式“加载”（技术上说，是把它们附加到环境中）：

```
library(sf)          # 矢量数据相关的类和函数
library(raster)      # 栅格数据相关的类和函数
```

以下安装的软件包中，包含了本书将使用的数据：

```
library(spData)        # 加载地理学数据集
library(spDataLarge)   # 加载大型地理学数据集
```

## 2.1 导读

本章将简要介绍基本的地理数据模型：矢量（vector）和栅格（raster）。在演示它们在 R 中的实现之前，我们将介绍每个数据模型背后的理论和它们主导的学科。

*矢量数据模型*使用点、线和多边形来表示世界。它们具有离散、明确定义的边界，因此矢量数据集通常具有高精度（但不一定准确，如第 2.5 节所述）[5]。*栅格数据模型*将表面划分成固定大小的单元格。栅格数据集是网络地图中使用的背景图的基础，自航空摄影和卫星遥感技术问世以来，一直是地理数据的重要来源。栅格数据将具有空间特定特征的数据聚合到给定的分辨率中，这意味着它们在空间上是一致的并且可扩展的（许多全球栅格数据集可用）。

哪种数据模型更好？答案可能取决于你的应用领域：

[5]译者注：高精度（precision）是指在数据的边界上有清晰的细节，而高度准确（accuracy）是指数据的边界要在正确的位置上。

- 矢量数据在社会科学中占主导地位，因为人类定居点往往具有离散的边界。
- 栅格数据通常在环境科学中占主导地位，因为它们依赖于遥感数据。

在某些存在很大重叠的领域，栅格和矢量数据集可以一起使用：例如，生态学家和人口统计学家通常同时使用矢量和栅格数据。此外，矢量和栅格可以相互转换（见第 **??** 节）。无论你的工作涉及更多矢量数据集还是更多栅格数据集，都值得在使用它们之前了解底层数据模型，后续章节将讨论这些内容。本书分别使用 **sf** 和 **raster** 软件包来处理矢量数据和栅格数据集。

## 2.2 矢量数据

在使用“vector”一词时要小心，因为它在本书中有两个含义：地理矢量数据和 R 中的 `vector`。前者是数据模型，后者是 R 中的数据结构，就像 `data.frame` 和 `matrix` 一样。尽管如此，两者之间仍然存在联系：地理矢量数据模型的核心是可以使用 `vector` 对象在 R 中表示空间坐标。[6]

地理矢量模型基于坐标参考系统（Coordinate Reference Systems，CRS）中的点。点可以表示独立的特征（例如，公交车站的位置），也可以将它们连接在一起形成更复杂的几何图形，例如线和多边形。大多数点数据仅包含两个维度（3 维坐标系会包含额外的 $z$ 值，通常表示海拔高度）。

举例来说，坐标系统中伦敦可以由坐标 `c(-0.1, 51.5)` 表示。这意味着它位于原点的东经 0.1 度和北纬 51.5 度。在这种情况下，原点位于地理（‘lon/lat’）CRS 中的 0 度经度（本初子午线）和 0 度纬度（赤道）（图2.1，左面板）。同样的点也可以在投影后的英国网格参考系统（British National Grid）[7]中用’Easting/Northing’ 值 `c(530000, 180000)` 来近似，这意味着伦敦位于 CRS 原点的东经 530 公里和北纬 180 公里处。这可以通过视觉验证：表示伦敦的点与原点之间有略多于 5 个“网格”，每个网格是以灰色网格线为界的、宽度为 100 公里的正方形区域（图2.1，右面板）。

[7] https://en.wikipedia.org/wiki/Ordnance_Survey_National_Grid

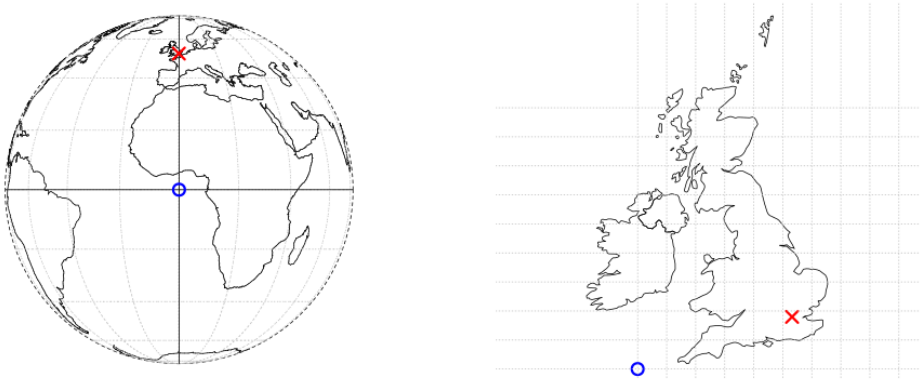

图 2.1: 矢量（点）数据的示例，其中伦敦的位置（红色 X）是相对于原点（蓝色圆圈）表示的。左图表示一个地理坐标参考系，其原点位于 0° 经度和纬度。右图表示一个投影坐标参考系，其原点位于西南半岛以外的海域。

英国网格参考系统的原点位于西南半岛以外的海域，这确保了英国大多数地点的东向和北向坐标值为正。[8] 关于 CRS 还有更多内容，请参见第 2.4 和第 **??** 节，但本节的目标是，只需知道坐标由两个数字组成，通常是 $x$ 维度在前 $y$ 维度在后，它们表示了与原点之间的距离。

**sf** 是一个为矢量数据提供类型支持的 R 包。它不仅取代了 **sp** 包，还提供了一个与 GEOS 和 GDAL 一致的命令行接口，取代了 **rgeos** 和 **rgdal**（在第 1.5 节中介绍）。本节介绍 **sf** 类，为后续章节（第 **??** 和第 **??** 章分别介绍 GEOS 和 GDAL 接口）做准备。

[8] 我们所指的原点如图 2.1 中所示，实际上是 “虚假” 的原点。“真正” 的原点，即畸变最小的位置，位于 2°W 和 49°N。这是由英国测量局选择的，纵向大致位于英国陆地的中心。

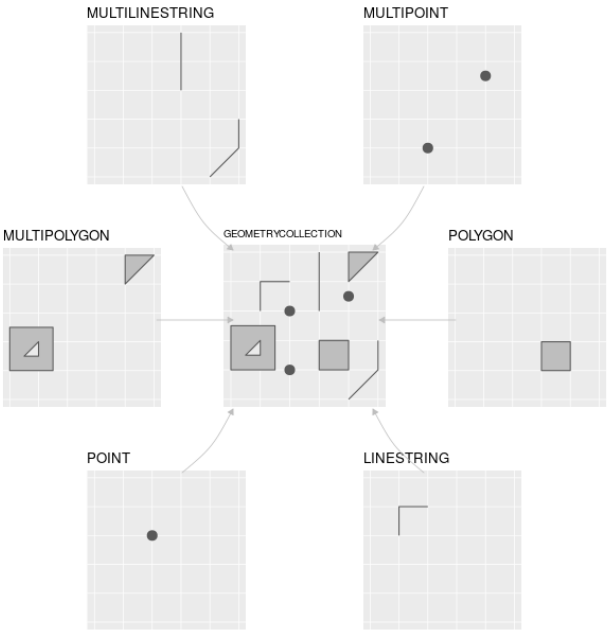


图 2.2: sf 包完整支持的简单要素类型。

### 2.2.1 简单要素介绍

简单要素（Simple features）是由开放地理空间联盟（OGC）开发和支持的一种开放标准[9]，OGC 是一个非营利组织，我们将在后面的章节（第 **??** 节）中再次介绍它。简单要素是一种分层数据模型，可以表示各种几何类型。在规范中定义的 17 种几何类型中，只有 7 种在绝大多数地理研究中使用（参见图 2.2）；这些核心几何类型在 R 包 **sf** 有完整的支持 (Pebesma, 2018)。[10]

**sf** 软件包可以表示所有常见的矢量几何类型（不支持栅格数据）：点、线、多边形及其各自的 “multi” 版本（将相同类型的要素组合成单个要素）。**sf** 还支持几何体集合（Geometry Collections），可以在单个对象中包含多个几何类型。**sf** 在很大程度上取代了 **sp** 生态系统，其中包括 **sp** (Pebesma and Bivand, 2018)、用于数据读写的 **rgdal** (Bivand et al., 2018) 和用于空间操作的 **rgeos** (Bivand and Rundel, 2018)。该软件包

[9] http://portal.opengeospatial.org/files/?artifact_id=25355

[10] 完整的 OGC 标准包括一些相当奇特的几何类型，包括 “表面” 和 “曲线” 几何类型，目前在实际场景中的应用有限。所有 17 种类型都可以用 **sf** 包表示，但（截至 2018 年夏季）只有核心的 7 种类型可以用于绘图。

有很好的文档，可以在其网站和 6 个文档中看到，使用以下方式查看：

```
vignette(package = "sf") # 查看可用示例
vignette("sf1")          # an introduction to the package
```

正如第一篇文档中所解释的那样，R 中的简单要素对象存储在数据框中，其中地理数据占据一个特殊的列，通常命名为 “geom” 或 “geometry”。我们将使用 **spData** 提供的 `world` 数据集，该数据集在本章开头已经加载了（有关软件包加载的数据集列表，请参见 nowosad.github.io/spData）。`world` 是一个空间对象，包含空间和属性列，函数 `names()` 返回列名（最后一列包含地理信息）：

```
names(world)
#>  [1] "iso_a2"    "name_long" "continent" "region_un" "subregion" "type"
#>  [7] "area_km2"  "pop"       "lifeExp"   "gdpPercap" "geom"
```

`geom` 列的值为 `sf` 对象赋予了空间能力：`world$geom` 是一个 “列表列[11]”，其中包含所有国家边界的多边形坐标。**sf** 包提供了一个 `plot()` 方法，用于可视化地理数据：下面的命令创建了图 2.3。

```
plot(world)
```

请注意，`plot()` 命令创建了多个地图，而不是像大多数 GIS 程序那样创建单个地图，每个地图对应 `world` 数据集中的一个变量。这种模式对于探索不同变量的空间分布非常有用，下面的第 2.2.3 节将进一步讨论这一点。

将空间对象视为具有空间能力的常规数据框具有许多优点，尤其是在你已经习惯于使用数据框的情况下。例如，常用的 `summary()` 函数提供了 `world` 对象中变量的概述。

[11] https://jennybc.github.io/purrr-tutorial/ls13_list-columns.html

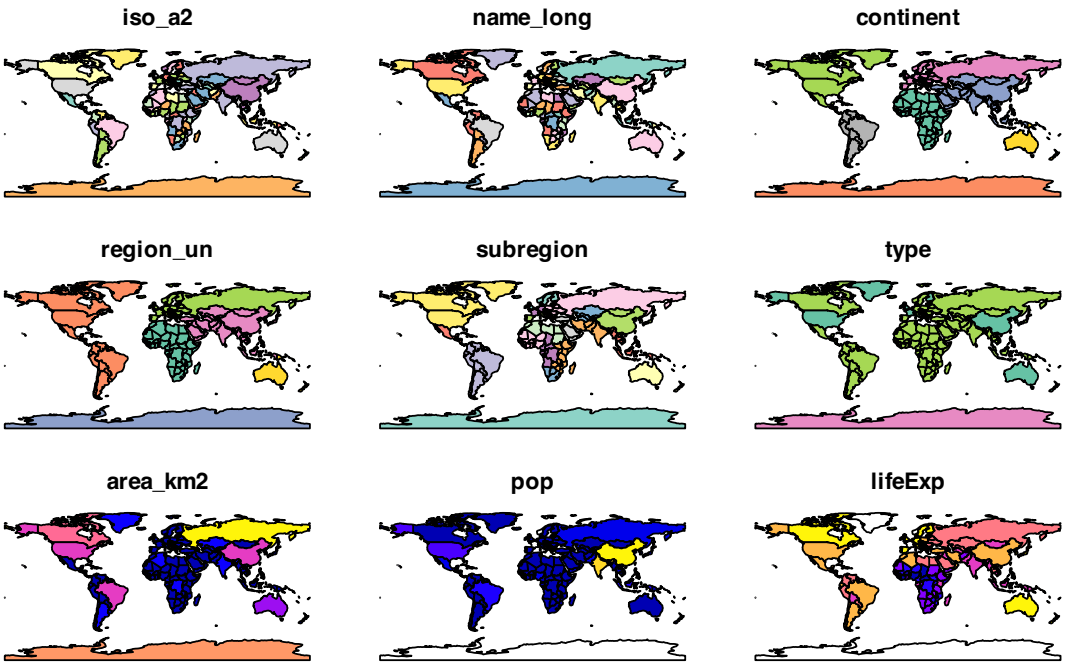


图 2.3: 使用 sf 包绘制的世界地图，每个属性都有一个子图。

```
summary(world["lifeExp"])
#>     lifeExp                geom
#>  Min.   :50.6   MULTIPOLYGON :177
#>  1st Qu.:65.0   epsg:4326    :  0
#>  Median :72.9   +proj=long...:  0
#>  Mean   :70.9
#>  3rd Qu.:76.8
#>  Max.   :83.6
#>  NA's   :10
```

虽然我们运行 `summary` 命令时只选择了一个变量，但它也输出了有关几何信息的报告。这展示了 **sf** 对象的几何列的“粘性”行为，这意味着除非用户有意删除它们，否则几何信息将被保留，正如我们将在第 **??** 节中看到的那样。返回结果中提供了有关 `world`

中包含的非空间数据和空间数据的摘要信息：所有国家的平均预期寿命为 71 岁（最低的不到 51 岁，最高的超过 83 岁，中位数为 73 岁）。

在 world 对象中，要素的几何类型为 MULTIPOLYGON，这种表示方法对于具有岛屿的国家（如印度尼西亚和希腊）是必要的。其他几何类型会在第 2.2.5 节中描述。

sf 对象的基本操作和内在元素值得深入研究，它可以被视为一个空间数据框（“spatial data frame”）。

sf 对象很容易进行子集操作。下面的代码显示了其前两行和前三列。与常规的 data.frame 相比，输出显示了两个主要的区别：包含了额外的地理数据（geometry type、dimension、bbox 和 CRS 信息 - epsg (SRID)、proj4string），以及存在一个名为 geom 的 geometry 列：

```
world_mini = world[1:2, 1:3]
world_mini
#> Simple feature collection with 2 features and 3 fields
#> Geometry type: MULTIPOLYGON
#> Dimension:     XY
#> Bounding box:  xmin: -180 ymin: -18.3 xmax: 180 ymax: -0.95
#> Geodetic CRS:  WGS 84
#> # A tibble: 2 x 4
#>   iso_a2 name_long continent                                        geom
#>   <chr>  <chr>     <chr>                                  <MULTIPOLYGON [°]>
#> 1 FJ     Fiji      Oceania   (((-180 -16.6, -180 -16.5, -180 -16, -180 -16.1, -~
#> 2 TZ     Tanzania  Africa    (((33.9 -0.95, 31.9 -1.03, 30.8 -1.01, 30.4 -1.13,~
```

所有这些可能看起来相当复杂，特别是对于一个想象中很简单的类系统。不过，用 **sf** 包的这种设计方式是事出有因的。

在描述 **sf** 包支持的每种几何类型之前，有必要退一步了解 sf 对象的组成部分。第 2.2.8 节展示了简单要素对象是数据框，具有特殊的几何列。这些空间列通常称为 geom

或 `geometry`：`world$geom` 是上面描述的 `world` 对象的空间元素。这些几何列是 `sfc` 类（请参见第 2.2.7 节）的“列表列”。`sfc` 对象又由一个或多个 `sfg` 类对象组成，`sfg` 是指简单要素几何 (Simple Feature Geometries)，我们将在第 2.2.6 节中介绍。

为了理解简单要素的空间组件是如何工作的，了解简单要素几何是至关重要的。因此，在第 2.2.5 节中，我们介绍了目前支持的每种简单要素几何类型，然后再描述如何使用 `sfg` 对象在 R 中表示这些几何类型，这些对象构成了 `sfc` 和最终的完整 `sf` 对象的基础。

上面的代码块中使用 `=` 运算符创建一个名为 `world_mini` 的新对象，这被称为赋值。为了达到相同的结果，可以使用等价的命令 `world_mini <- world[1:2, 1:3]`。虽然“箭头赋值”更常用，但我们使用“等于赋值”，因为它打起来更快，而且它与常用的语言（如 Python 和 JavaScript）的赋值兼容，所以更易于教授。使用哪种方式主要是个人喜好，只要保持一致即可（可以使用 **styler** 等软件包来更改样式）。

### 2.2.2 为什么使用简单要素?

简单要素是一种广泛支持的数据模型，它是许多 GIS 应用程序（包括 QGIS 和 PostGIS）中数据结构的基础。这样设计的一个主要优点是，使用简单要素可以让你的数据方便地在各个 GIS 应用程序之间进行交叉传输，例如从空间数据库导入和导出。

从 R 的角度来看，一个更具体的问题是“**sp** 包已经经过了测试和验证，为什么还要使用 **sf** 包”？原因有很多（与简单要素模型的优势相关），包括：

- 快速读写数据。
- 提高绘图性能。
- **sf** 对象在大多数操作中可以被当做数据框。
- **sf** 函数可以使用`%>%` 运算符组合，并且可以与 R 的 `tidyverse` 系列的包配合使用。
- **sf** 函数名称相对一致且直观（所有函数都以 `st_` 开头）。

由于这些优势，一些空间包（包括 **tmap**、**mapview** 和 **tidycensus**）已经添加了对 **sf** 的支持。然而，大多数包要经过多年才能过渡到 **sf**，有些甚至永远不会支持。幸

运的是，通过将它们转换为 **sp** 中使用的 `Spatial` 类，这些包仍然可以在基于 `sf` 对象的工作流中使用：

```
library(sp)
world_sp = as(world, Class = "Spatial")
# sp functions ...
```

`Spatial` 对象可以通过相同的方式或使用 `st_as_sf()` 转换成 `sf` 对象：

```
world_sf = st_as_sf(world_sp, "sf")
```

### 2.2.3 基本地图制作

在 **sf** 中，可以使用 `plot()` 创建基本地图。默认情况下，这会创建一个多面板图（类似于 **sp** 的 `spplot()`），每个子图都对应对象的一个变量，如图 2.4 左侧面板所示。如果要绘制的对象只有一个变量，则会生成带有连续颜色的图例，如右侧面板所示。可以使用 `col =` 设置颜色，但这不会创建连续的调色板或图例。

```
plot(world[3:6])
plot(world["pop"])
```

通过设置 `add = TRUE`，可以将绘图结果添加为现有图像的图层。[12]接下来的代码块把亚洲国家组合在一起，演示了图层叠加，而且预告了第 **??** 和 **??** 章中属性和空间数据操作的内容：

```
world_asia = world[world$continent == "Asia", ]
asia = st_union(world_asia)
```

[12]对 **sf** 对象调用 `plot()` 函数，在幕后会使用 `sf:::plot.sf()` 函数进行操作。`plot()` 是一个通用方法，其行为取决于正在绘制的对象的类。

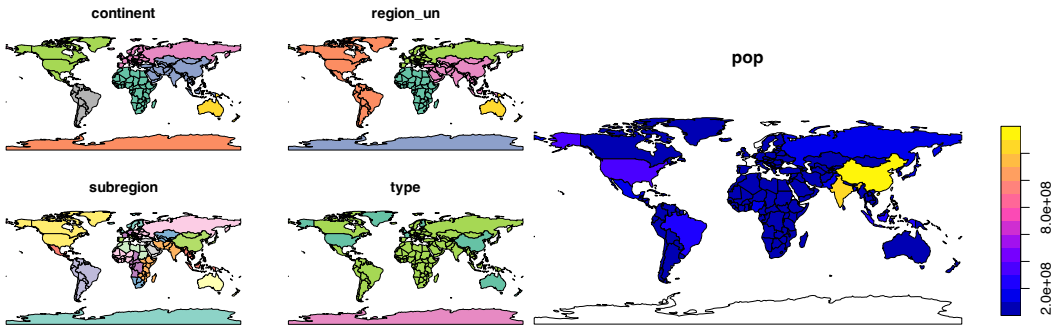


图 2.4: 使用 sf 绘图，多变量（左）和单变量（右）。

我们现在可以在世界地图上绘制亚洲大陆。请注意，第一个图必须只有一个面板，才能使用 `add = TRUE`。如果第一个图有一个图例，则必须使用 `reset = FALSE`（以下代码的绘图结果未展示，读者可自行尝试）：

```
plot(world["pop"], reset = FALSE)
plot(asia, add = TRUE, col = "red")
```

通过这种方式添加图层可以用于验证图层之间的地理对应关系：`plot()` 函数执行速度快，代码行数少，但不能创建具有广泛互动选项的交互式地图。对于更高级的地图制作，我们建议使用专用的可视化包，如 **tmap**（请参见第 **??** 章）。

### 2.2.4 基本绘图参数

使用 **sf** 的 `plot()` 方法可以以多种方式修改地图。由于 **sf** 扩展了 R 的基础绘图方法，因此 `plot()` 的参数（例如 `main =`，用于指定地图的标题）也适用于 `sf` 对象（请参见`?graphics::plot` 和`?par`）。[13]

图 2.5 展示 **sf** 绘图的灵活性，它在世界地图上叠加了一个圆形图层，圆的直径表示国家人口（使用 `cex =` 设置直径）。可以使用以下命令创建基本的地图（请参见本章末尾的练习和脚本 `02-contplot.R`[14]创建图 2.5）：

[13] 注意：当绘制多个 `sf` 列时，许多绘图参数在面板地图中会被忽略。

[14] https://github.com/Robinlovelace/geocompr/blob/master/code/02-contpop.R

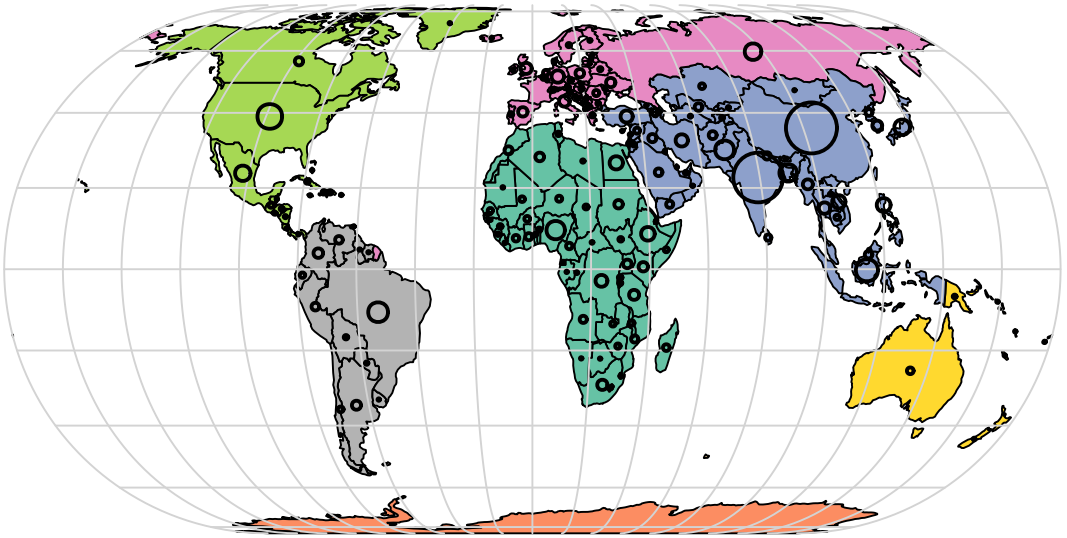

图 2.5: 各国所属的洲（用填充色表示）和 2015 年的人口数量（用圆圈表示，圆圈的面积与人口数量成正比）。

```
plot(world["continent"], reset = FALSE)
cex = sqrt(world$pop) / 10000
world_cents = st_centroid(world, of_largest = TRUE)
plot(st_geometry(world_cents), add = TRUE, cex = cex)
```

上面的代码使用函数 `st_centroid()` 将一个几何类型（多边形）转换为另一个几何类型（点）（请参见第 **??** 章），并使用 `cex` 参数来设置其大小。

**sf** 的 `plot()` 方法也有只适用于地理数据的特定参数。例如，`expandBB` 可以用于调整画布中 `sf` 对象的边界：它接受一个长度为四的百分比数值向量，按以下顺序扩展绘图的边界框：底部、左侧、顶部、右侧。下面的代码块使用 `expandBB` 将印度绘制在以整个亚洲为边界的画布中，东部的中国也被重点标记出来，生成图 2.6（请参见下面的练习，了解如何在图中添加文本）：


图 2.6: 亚洲中的印度，演示 expandBB 参数的用法。

```
india = world[world$name_long == "India", ]
plot(st_geometry(india), expandBB = c(0, 0.2, 0.1, 1), col = "gray", lwd = 3)
plot(world_asia[0], add = TRUE)
```

注意，代码中使用 `[0]` 仅保留几何列，设置 `lwd` 来强调印度。请参见第 **??** 节，了解表示各种几何类型的其他可视化技术。各种几何类型的介绍是下一节的主题。

### 2.2.5 几何类型

几何类型是简单要素的基本组成部分。在 R 中，简单要素可以采用 **sf** 包支持的 17 种几何类型之一。在本章中，我们将重点介绍七种最常用的类型：`POINT`、`LINESTRING`、`POLYGON`、`MULTIPOINT`、`MULTILINESTRING`、`MULTIPOLYGON` 和 `GEOMETRYCOLLECTION`。可以在 PostGIS 手册[15] 中找到所有可能的要素类型列表。

通常，简单要素几何体的标准编码是 well-known binary (WKB) 或 well-known text (WKT)。WKB 表示通常是十六进制字符串，容易被计算机读取。这就是为什么 GIS 和空间数据库使用 WKB 来传输和存储几何对象的原因。另一方面，WKT 是简单要素的人类可读的文本标记描述。两种格式是可以互相转换的，如果要展示其中一种，我们自然会选择人类可读的 WKT 表示。

[15] http://postgis.net/docs/using_postgis_dbmanagement.html

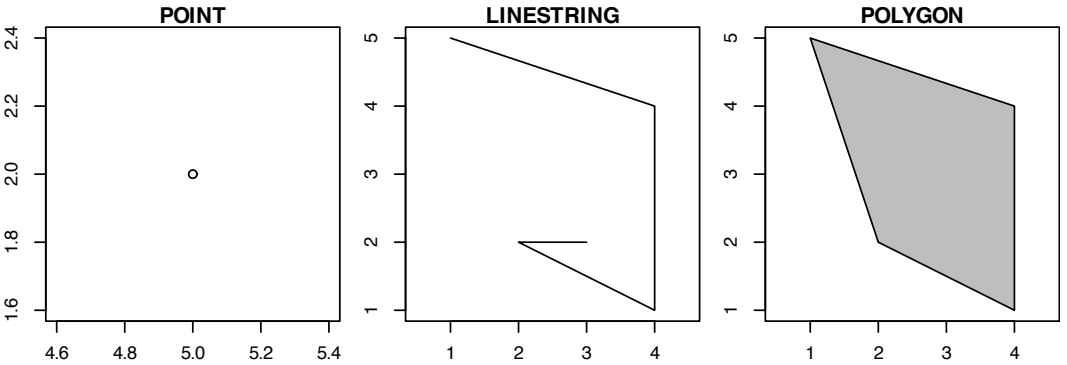


图 2.7: 点，线，多边形的示例。

每种几何类型的基础是点（POINT）。点是二维、三维或四维空间中的坐标（有关更多信息，请参见 `vignette("sf1")`），例如（请参见图 2.7 中的左侧面板）：

- `POINT (5 2)`

线（LINESTRING）是一系列通过直线相连的点，例如（请参见图 2.7 中的中间面板）：

- `LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2)`

多边形（POLYGON）是一系列点形成的一个封闭的、不相交的环。封闭意味着多边形的第一个和最后一个点有相同的坐标（参见图 2.7 中的右侧面板）[16]。

- 没有内环的多边形: `POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5))`

到目前为止，我们已经创建了每个要素中只有一个几何实体的几何体。然而，**sf** 也允许单个要素中存在多个几何体（因此称为“几何体集合”），比如使用每种几何类型的“多（multi）”版本：

- 多点（MULTIPOINT）：`MULTIPOINT (5 2, 1 3, 3 4, 3 2)`

[16]根据定义，多边形有一个外边界（外环），可以有零个或多个内边界（内环），也称为孔。带有孔的多边形可以是 `POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5), (2 4, 3 4, 3 3, 2 3, 2 4))` 这样的形式。

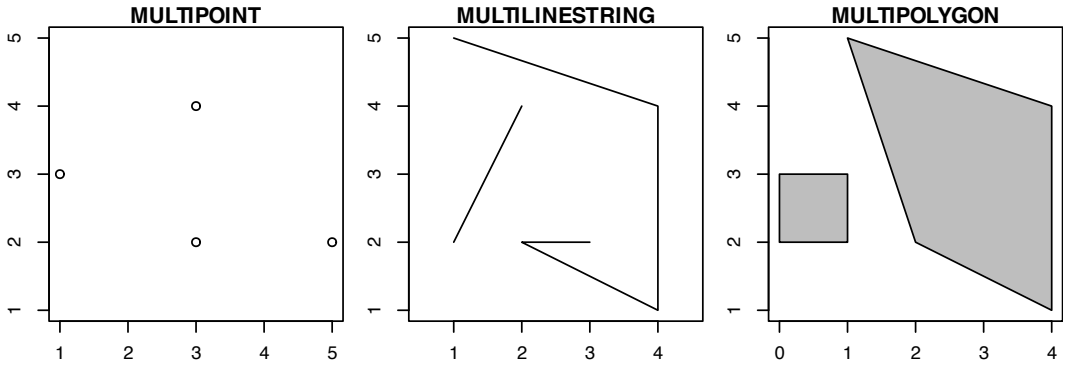


图 2.8: 多点，多线，多面的示例。

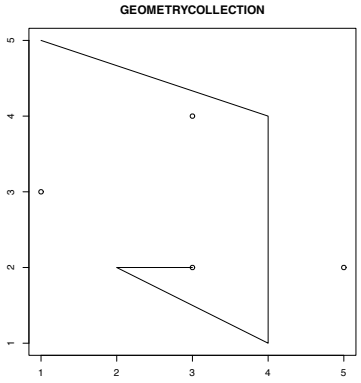


图 2.9: 几何体集合的示例。

- 多线（MULTILINESTRING）：`MULTILINESTRING ((1 5, 4 4, 4 1, 2 2, 3 2), (1 2, 2 4))`
- 多面（MULTIPOLYGON）：`MULTIPOLYGON (((1 5, 2 2, 4 1, 4 4, 1 5), (0 2, 1 2, 1 3, 0 3, 0 2)))`

最后，几何体集合可以包含任何几何体的组合，包括（多）点和线（请参见图 2.9）：

- 几何体集合（GEOMETRYCOLLECTION）：`GEOMETRYCOLLECTION (MULTIPOINT (5 2, 1 3, 3 4, 3 2), LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2))`

### 2.2.6 简单要素几何（sfg）

`sfg` 类在 R 中表示不同的简单要素几何类型：点、线、多边形（以及多点、多线等）或几何体集合。

通常情况下，你不需要自己创建几何体，因为可以直接导入已有的空间文件。但如果需要的话，你可以使用一组函数从头创建简单要素几何对象（`sfg`）。这些函数的名称简单且一致，它们都以 `st_` 前缀开头，以小写字母的几何类型名称结尾：

- 一个点（POINT）：`st_point()`
- 一个线（LINESTRING）：`st_linestring()`
- 一个多边形（POLYGON）：`st_polygon()`
- 多点（MULTIPOINT）：`st_multipoint()`
- 多线（MULTILINESTRING）：`st_multilinestring()`
- 多面（MULTIPOLYGON）：`st_multipolygon()`
- 几何体集合（GEOMETRYCOLLECTION）：`st_geometrycollection()`

`sfg` 对象可以从三种基本 R 数据类型创建：

1. 数值向量：单个点
2. 矩阵：一组点，其中每行表示一个点、多点或线
3. 列表：对象的集合，例如矩阵、多线或几何体集合

函数 `st_point()` 接收数值向量返回单个点：

```
st_point(c(5, 2))                 # XY point
#> POINT (5 2)
st_point(c(5, 2, 3))              # XYZ point
#> POINT Z (5 2 3)
st_point(c(5, 2, 1), dim = "XYM") # XYM point
#> POINT M (5 2 1)
st_point(c(5, 2, 3, 1))           # XYZM point
#> POINT ZM (5 2 3 1)
```

结果显示，XY（2D 坐标）、XYZ（3D 坐标）和 XYZM（3D 坐标加一个额外的变量，通常是测量精度）点类型分别从长度为 2、3 和 4 的向量创建。XYM 类型必须使用 `dim` 参数（即维度）指定。

相比之下，创建多点（`st_multipoint()`）和线（`st_linestring()`）对象时应该使用矩阵：

```
# 函数 rbind 简化了矩阵的创建
## 多点
multipoint_matrix = rbind(c(5, 2), c(1, 3), c(3, 4), c(3, 2))
st_multipoint(multipoint_matrix)
#> MULTIPOINT ((5 2), (1 3), (3 4), (3 2))
## 线
linestring_matrix = rbind(c(1, 5), c(4, 4), c(4, 1), c(2, 2), c(3, 2))
st_linestring(linestring_matrix)
#> LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2)
```

最后，使用列表来创建多线、多面和几何体集合：

```
## 多边形
polygon_list = list(rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5)))
st_polygon(polygon_list)
#> POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5))
```

```
## 有内环的多边形
polygon_border = rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5))
polygon_hole = rbind(c(2, 4), c(3, 4), c(3, 3), c(2, 3), c(2, 4))
polygon_with_hole_list = list(polygon_border, polygon_hole)
st_polygon(polygon_with_hole_list)
#> POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5), (2 4, 3 4, 3 3, 2 3, 2 4))
```

```
## 多线
multilinestring_list = list(rbind(c(1, 5), c(4, 4), c(4, 1), c(2, 2), c(3, 2)),
                            rbind(c(1, 2), c(2, 4)))
st_multilinestring((multilinestring_list))
#> MULTILINESTRING ((1 5, 4 4, 4 1, 2 2, 3 2), (1 2, 2 4))
```

```
## 多面
multipolygon_list = list(list(rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5))),
                         list(rbind(c(0, 2), c(1, 2), c(1, 3), c(0, 3), c(0, 2))))
st_multipolygon(multipolygon_list)
#> MULTIPOLYGON (((1 5, 2 2, 4 1, 4 4, 1 5)), ((0 2, 1 2, 1 3, 0 3, 0 2)))
```

```
## 几何体集合
gemetrycollection_list = list(st_multipoint(multipoint_matrix),
                              st_linestring(linestring_matrix))
st_geometrycollection(gemetrycollection_list)
#> GEOMETRYCOLLECTION (MULTIPOINT (5 2, 1 3, 3 4, 3 2),
#>   LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2))
```

### 2.2.7 简单要素列（sfc）

一个 `sfg` 对象只包含单个简单要素几何体。一个简单要素几何列（`sfc`）是一个 `sfg` 对象的列表，它还能够包含坐标参考系的信息。例如，要将两个简单要素组合成一个具有两个要素的对象，可以使用 `st_sfc()` 函数。注意，`sfc` 对象其实就是 **sf** 数据框中的几何列：

```
# sfc POINT
point1 = st_point(c(5, 2))
point2 = st_point(c(1, 3))
points_sfc = st_sfc(point1, point2)
points_sfc
#> Geometry set for 2 features
#> Geometry type: POINT
#> Dimension:     XY
#> Bounding box:  xmin: 1 ymin: 2 xmax: 5 ymax: 3
#> CRS:           NA
#> POINT (5 2)
#> POINT (1 3)
```

在大多数情况下，`sfc` 对象包含相同几何类型的对象。因此，当我们将类型为多边形的 `sfg` 对象转换为简单要素几何列时，我们也会得到一个类型为多边形的 `sfc` 对象，可以使用 `st_geometry_type()` 进行验证。同样，多线的几何列将生成类型为多线的 `sfc` 对象：

```
# sfc POLYGON
polygon_list1 = list(rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5)))
polygon1 = st_polygon(polygon_list1)
polygon_list2 = list(rbind(c(0, 2), c(1, 2), c(1, 3), c(0, 3), c(0, 2)))
polygon2 = st_polygon(polygon_list2)
polygon_sfc = st_sfc(polygon1, polygon2)
st_geometry_type(polygon_sfc)
#> [1] POLYGON POLYGON
#> 18 Levels: GEOMETRY POINT LINESTRING POLYGON MULTIPOINT ... TRIANGLE
```

```
# sfc MULTILINESTRING
multilinestring_list1 = list(rbind(c(1, 5), c(4, 4), c(4, 1), c(2, 2), c(3, 2)),
                             rbind(c(1, 2), c(2, 4)))
multilinestring1 = st_multilinestring((multilinestring_list1))
multilinestring_list2 = list(rbind(c(2, 9), c(7, 9), c(5, 6), c(4, 7), c(2, 7)),
                             rbind(c(1, 7), c(3, 8)))
multilinestring2 = st_multilinestring((multilinestring_list2))
multilinestring_sfc = st_sfc(multilinestring1, multilinestring2)
st_geometry_type(multilinestring_sfc)
#> [1] MULTILINESTRING MULTILINESTRING
#> 18 Levels: GEOMETRY POINT LINESTRING POLYGON MULTIPOINT ... TRIANGLE
```

还可以使用不同几何类型的 `sfg` 对象创建 `sfc` 对象：

```
# sfc GEOMETRY
point_multilinestring_sfc = st_sfc(point1, multilinestring1)
st_geometry_type(point_multilinestring_sfc)
#> [1] POINT           MULTILINESTRING
#> 18 Levels: GEOMETRY POINT LINESTRING POLYGON MULTIPOINT ... TRIANGLE
```

如前所述，`sfc` 对象可以额外存储有关坐标参考系统（CRS）的信息。要指定特定的 CRS，可以使用 `sfc` 对象的 `epsg (SRID)` 或 `proj4string` 属性。`epsg (SRID)` 和 `proj4string` 的默认值为 `NA`（不可用），可以使用 `st_crs()` 进行验证：

```
st_crs(points_sfc)
#> Coordinate Reference System: NA
```

`sfc` 对象中的所有几何体必须具有相同的 CRS。我们可以将坐标参考系统作为 `st_sfc()` 函数的 `crs` 参数添加。该参数接受一个整数，例如 `epsg` 代码 `4326`，它会自动添加'proj4string'（请参见第 2.4 节）：

```
# EPSG 定义
points_sfc_wgs = st_sfc(point1, point2, crs = 4326)
st_crs(points_sfc_wgs)
#> Coordinate Reference System:
#> EPSG: 4326
#> proj4string: " +proj=longlat +datum=WGS84 +no_defs"
```

st_sfc() 函数还可以接受一个 proj4string 作为参数（执行结果未展示，读者可自行尝试）：

```
# PROJ4STRING 定义
st_sfc(point1, point2, crs = "+proj=longlat +datum=WGS84 +no_defs")
```

有时 st_crs() 会返回一个 proj4string，但不会返回 epsg 代码。这是因为没有通用的方法可以从 proj4string 转换为 epsg（请参见第 ?? 章）。

### 2.2.8 sf 类

第 2.2.5 到第 2.2.7 节介绍了纯几何对象、**sf** 几何对象和 **sf** 列对象，它们都是简单要素所表示的矢量数据的组件。最后一个组件是非地理属性，表示要素的名称或其他属性，例如测量值、分组等其他值。

为了说明属性，我们以 “2017 年 6 月 21 日伦敦 25°C 温度” 为示例。此示例包含几何（坐标）和三个具有不同类别的属性（地名、温度和日期）。[17] sf 类的对象通过将属性（data.frame）与简单要素几何列（sfc）组合来表示此类数据。它们使用 st_sf() 创建，如下所示，它创建了上述伦敦的示例：

[17] 其他属性可能包括城市或村庄等类别，或者一条备注来说明是否是自动气象站测量的。

```
lnd_point = st_point(c(0.1, 51.5))                 # sfg 对象
lnd_geom = st_sfc(lnd_point, crs = 4326)           # sfc 对象
lnd_attrib = data.frame(                           # data.frame 对象
  name = "London",
  temperature = 25,
  date = as.Date("2017-06-21")
  )
lnd_sf = st_sf(lnd_attrib, geometry = lnd_geom)    # sf 对象
```

上述代码做了什么？首先，使用坐标创建了简单要素几何 (sfg)。其次，将几何体转换为带有 CRS 的简单要素几何列 (sfc)。第三，属性被存储在 data.frame 中，然后使用 st_sf() 将其与 sfc 对象组合。最终生成一个 sf 对象，结果如下所示（省略了一些输出）：

```
lnd_sf
#> Simple feature collection with 1 features and 3 fields
#> ...
#>     name temperature       date         geometry
#> 1 London          25 2017-06-21 POINT (0.1 51.5)
```

```
class(lnd_sf)
#> [1] "sf"         "data.frame"
```

结果显示，sf 对象实际上有两个类，sf 和 data.frame。简单要素实际上只是一个具有一列存储在列表中的空间属性的数据框，这一列通常称为 geometry，如第 2.2.1 节所述。这种二元性是简单要素概念的核心：大多数情况下，sf 对象可以像 data.frame 一样处理。简单要素本质上是具有空间扩展的数据框。

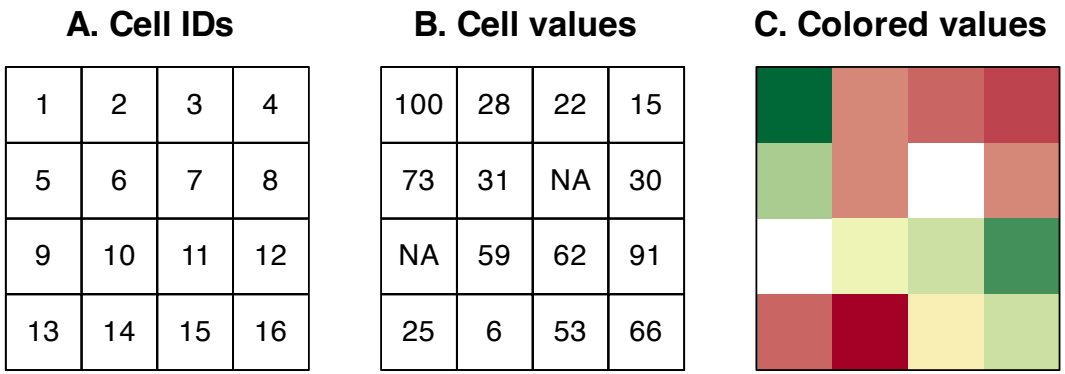


图 2.10: 栅格数据类型：（A）单元格 ID，（B）单元格值，（C）彩色栅格地图。

## 2.3 栅格数据

地理栅格数据模型通常由栅格头信息和一个矩阵组成，其中矩阵表示等间距的单元格（通常也称为像素；图 2.10:A）。[18]栅格头信息定义了坐标参考系统、范围和起点。起点通常是矩阵左下角的坐标（然而，**raster** 包默认使用左上角，如图 2.10:B 所示）。头信息通过列数、行数和单元格大小分辨率定义了范围。因此，从起点开始，我们可以通过单元格的 ID（图 2.10:B）或显式指定行和列来轻松访问和修改每个单元格。矩形矢量多边形会存储四个边界点的坐标，而这种矩阵表示则避免了这种冗余（它实际上只存储一个坐标，即起点）。这种表示方法和地图代数使得栅格处理比矢量数据处理更快更高效。然而，与矢量数据相比，一个栅格层的单元格只能容纳一个值。该值可以是数值或分类值（图 2.10:C）。

栅格地图通常表示连续数值的指标，例如高程、温度、人口密度或光谱数据（图 2.11）。当然，我们也可以使用栅格数据模型来表示离散特征，例如土壤或土地覆盖类（图 2.11）。但是这些特征的离散边界会变得模糊，使用矢量表示可能更合适。

[18]头信息可以是数据文件的一部分，例如 GeoTIFF，也可以是存储在额外的文件中，例如 ASCII 格式的网格。还有一种无头（平面）二进制栅格格式，它可以方便地导入各种软件程序。

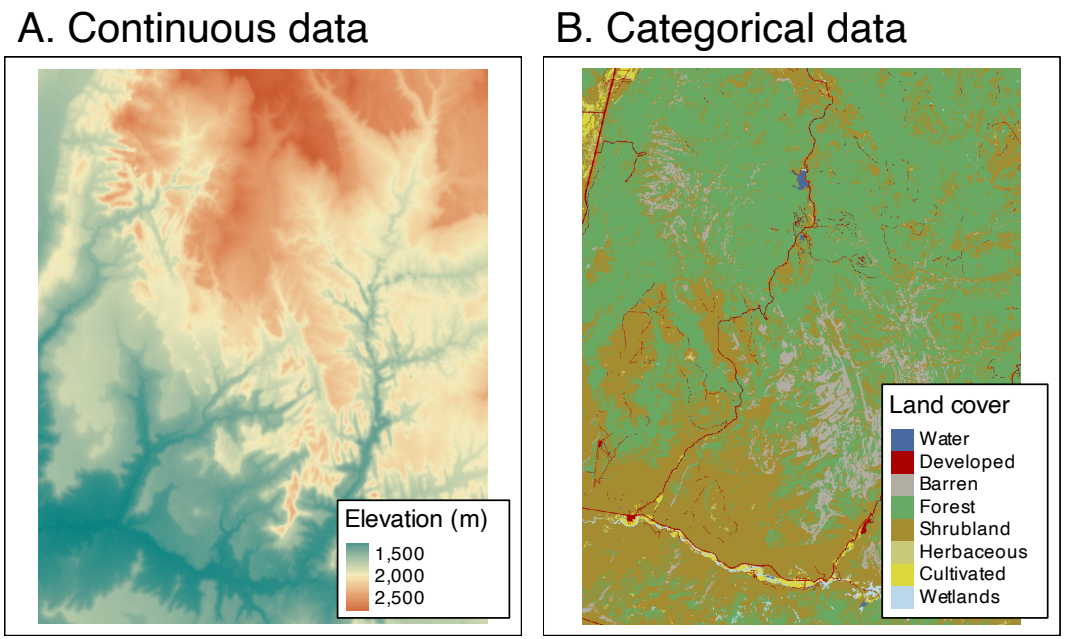


图 2.11: 连续和分类栅格的示例。

### 2.3.1 栅格数据简介

**raster** 包支持 R 中的栅格对象。它提供了丰富的函数用于创建、读取、导出、处理栅格数据集。除了常用的栅格数据操作外，**raster** 还提供了许多低级函数，可以组合开发出更高级的栅格处理功能。**raster** 还可以处理比内存更大的栅格数据集。在这种情况下，**raster** 提供了流式处理的功能，它将栅格分成较小的块（行或块），并迭代地处理这些块而不是将整个栅格文件全部加载到内存（有关更多信息，请参见 `vignette("functions", package = "raster")`）。

为了演示 **raster**，我们使用来自 **spDataLarge** 的数据集（注意，这些包在本章的开头已加载）。它由几个栅格对象和一个覆盖锡安国家公园（Zion National Park，位于美国犹他州）区域的矢量对象组成。其中，`srtm.tif` 文件记录了该区域的数字高程模型（有关更多详细信息，请参见其文档 `?srtm`）。首先，让我们创建一个名为 `new_raster` 的 `RasterLayer` 对象：

```
raster_filepath = system.file("raster/srtm.tif", package = "spDataLarge")
new_raster = raster(raster_filepath)
```

在控制台中输入栅格的名称，将打印出栅格头信息（范围、维度、分辨率、CRS）和一些附加信息（类、数据源名称、栅格值的摘要）：

```
new_raster
#> class       : RasterLayer
#> dimensions  : 457, 465, 212505  (nrow, ncol, ncell)
#> resolution  : 0.000833, 0.000833  (x, y)
#> extent      : -113, -113, 37.1, 37.5  (xmin, xmax, ymin, ymax)
#> coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
#> data source : /home/robin/R/x86_64-pc-linux../3.5/spDataLarge/raster/srtm.tif
#> names       : srtm
#> values      : 1024, 2892  (min, max)
```

还有多个专用函数可以输出不同的信息：`dim(new_raster)` 返回行数、列数和层数；`ncell()` 函数返回单元格（像素）的数量；`res()` 返回栅格的空间分辨率；`extent()` 返回其空间范围；`crs()` 返回其坐标参考系统（栅格重投影在第 **??** 节中介绍）。`inMemory()` 报告了栅格数据是存储在内存中（默认）还是存储在磁盘上。

`help("raster-package")` 命令返回 **raster** 包中所有可用函数的完整列表。

### 2.3.2 基本地图制作

与 **sf** 包类似，**raster** 包也为它自己的类提供了 `plot()` 方法。

```
plot(new_raster)
```

在 R 中，还有其他几种绘制栅格数据的方法，不过这些超出了本节的范围。包括：

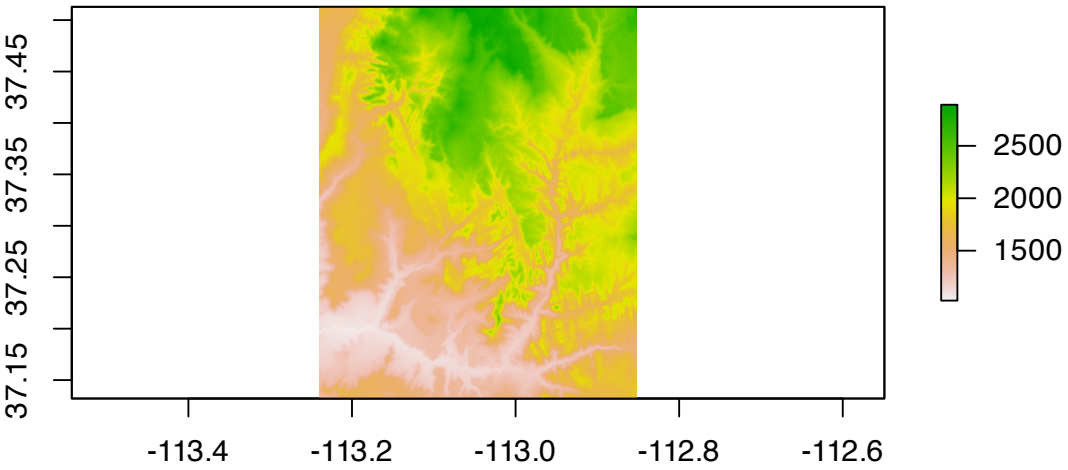


图 2.12: 基本的栅格绘图。

- 使用 `spplot()` 和 `levelplot()` 函数（分别来自 **sp** 和 **rasterVis** 包）创建多面板图，这是一种常见的可视化随时间变化的数据的技术。
- 使用 **tmap**、**mapview** 和 **leaflet** 等包创建栅格和矢量对象的交互式地图（参见第 **??** 章）。

### 2.3.3 栅格类

`RasterLayer` 类是栅格对象最简单的表示，仅包含一个图层。在 R 中创建栅格对象的最简单方法是从磁盘或服务器中读取栅格文件。

```
raster_filepath = system.file("raster/srtm.tif", package = "spDataLarge")
new_raster = raster(raster_filepath)
```

在 **rgdal** 的帮助下，**raster** 包支持读写多种文件格式。要查看文件格式列表和系统上可用的驱动程序，请运行 `raster::writeFormats()` 和 `rgdal::gdalDrivers()`。

栅格对象还可以使用 `raster()` 函数从头创建。下面的代码块演示了如何创建一个新的 `RasterLayer` 对象。生成的栅格由 36 个单元格组成（由 `nrows` 和 `ncols` 指定 6 行 6 列），以本初子午线和赤道的交点为中心（请参见 `xmn`、`xmx`、`ymn` 和 `ymx` 参数）。CRS 参

数使用栅格对象的默认值：WGS84。这意味着分辨率的单位是度，我们将其设置为 0.5（res 参数）。每个单元格都被赋予一个值（vals 参数）：1 赋值给单元格 1，2 赋值给单元格 2，以此类推。请注意：raster() 按行填充单元格（与 matrix() 不同），从左上角开始，这意味着第一行包含值 1 到 6，第二行包含值 7 到 12，以此类推。

```
new_raster2 = raster(nrows = 6, ncols = 6, res = 0.5,
                     xmn = -1.5, xmx = 1.5, ymn = -1.5, ymx = 1.5,
                     vals = 1:36)
```

其他创建栅格对象的方法，请参见?raster。

除了 RasterLayer 之外，还有两个额外的类：RasterBrick 和 RasterStack。两者都可以处理多个图层，但在支持的文件格式数量、内部表示类型和处理速度方面有所差异。

RasterBrick 由多个图层组成，通常对应于单个多光谱卫星文件或单个多层对象。brick() 函数可以创建一个 RasterBrick 对象。通常，你需要提供一个多层栅格文件的文件名，但也可以使用其他栅格对象和其他空间对象（请参见 ?brick 以获取所有支持的格式）。

```
multi_raster_file = system.file("raster/landsat.tif", package = "spDataLarge")
r_brick = brick(multi_raster_file)
```

```
r_brick
#> class       : RasterBrick
#> resolution  : 30, 30  (x, y)
#> ...
#> names       : landsat.1, landsat.2, landsat.3, landsat.4
#> min values  :      7550,      6404,      5678,      5252
#> max values  :     19071,     22051,     25780,     31961
```

nlayers() 函数用于检索存储在 Raster* 对象中的图层数量。

```
nlayers(r_brick)
#> [1] 4
```

RasterStack 类似于 RasterBrick，由多个图层组成。但与 RasterBrick 不同的是，RasterStack 允许你连接存储在不同文件或内存中的多个栅格对象。更具体地说，RasterStack 是具有相同范围和分辨率的 RasterLayer 对象列表。因此，创建 RasterStack 的一种方法是使用已存在于 R 全局环境中的空间对象。同样，也可以简单地指定存储在磁盘上的文件路径。

```
raster_on_disk = raster(r_brick, layer = 1)
raster_in_memory = raster(xmn = 301905, xmx = 335745,
                          ymn = 4111245, ymx = 4154085,
                          res = 30)
values(raster_in_memory) = sample(seq_len(ncell(raster_in_memory)))
crs(raster_in_memory) = crs(raster_on_disk)
```

```
r_stack = stack(raster_in_memory, raster_on_disk)
r_stack
#> class : RasterStack
#> dimensions : 1428, 1128, 1610784, 2
#> resolution : 30, 30
#> ...
#> names       :   layer, landsat.1
#> min values  :       1,      7550
#> max values  : 1610784,     19071
```

另一个区别是，RasterBrick 对象的处理时间通常比 RasterStack 对象短。

选择使用哪个 Raster* 类主要取决于输入数据的特性。处理单个多层文件或对象时，使用 RasterBrick 最高效，而 RasterStack 允许基于多个文件、多个 Raster* 对象或两者的混合进行计算。

对 RasterBrick 和 RasterStack 对象的操作通常会返回一个 RasterBrick 对象。

## 2.4 坐标参考系

不论是矢量数据还是栅格数据，空间数据的固有概念是共通的。也许最基本的概念之一是坐标参考系统（CRS），它定义了数据的空间元素与地球表面（或其他物体）的关系。CRS 可以是地理坐标系或投影坐标系，这在本章开头已经介绍过（请参见图 2.1）。本节将解释这两类坐标系，为第 **??** 节的 CRS 转换奠定基础。

### 2.4.1 地理坐标系

地理坐标系使用经度和纬度两个值来标识地球表面上的任何位置。*经度*是指东西方向上与本初子午线平面的角度距离。*纬度*是指距赤道平面向北或向南的角度距离。因此，地理坐标系中的距离不是以米为单位进行测量的。这点很重要，详见第 **??** 节。

地理坐标系中的地球表面可以用球面或椭球面来表示。球面模型假设地球是一个给定半径的完美球体。球面模型具有简单性的优点，但很少使用，因为它们不准确：地球不是一个完美球体！椭球面模型由两个参数定义：赤道半径和极半径。这个模型是合适的，因为地球是扁的：赤道半径比极半径长约 11.5 公里 (Maling, 1992)。[19]

椭球体是 CRSs 的更广泛组成部分—*基准面*（*datum*）的一部分。其中包含有关要使用哪个椭球体（使用 PROJ CRS 库中的 ellps 参数）以及笛卡尔坐标与地球表面位置之间的精确关系的信息。这些额外的细节存储在 proj4string[20] 符号中的 towgs84 参数中（有关详细信息，请参见 proj4.org/parameters.html[21]）。基准面可以考虑地球表面的局部变化，例如大型山脉的存在，可以使用局部 CRS 来调整。有两种类型的基准：本地基准和地心基准。在本地基准中，例如 NAD83，椭球面被移动到与特定位置的表面对

[19] 压缩程度通常称为*扁率*，以赤道半径 ($a$) 和极半径 ($b$) 定义，如下所示：$f = (a - b)/a$。也可以使用*椭圆度*和*压缩度*这两个术语 (Maling, 1992)。由于 $f$ 是一个相当小的值，椭球模型使用 “反扁率”($rf = 1/f$) 来定义地球的压缩。各种椭球模型中的 $a$ 和 $rf$ 的值可以通过执行 st_proj_info(type = "ellps") 来查看。译者注：在最新的 sf 包中，st_proj_info 函数已更名为 sf_proj_info。

[20] https://proj4.org/operations/conversions/latlon.html?highlight=towgs#cmdoption-arg-towgs84

[21] https://proj4.org/usage/projections.html

齐。在地心基准中，例如 WGS84，中心是地球的重心，投影的精度没有针对特定位置进行优化。可以通过执行 `st_proj_info(type = "datum")` 来查看可用的基准及其定义。

### 2.4.2 投影坐标参考系统

投影坐标参考系统是基于隐含的平面上的笛卡尔坐标。它们有一个原点、x 和 y 轴，以及一个线性测量单位，例如米。所有投影坐标参考系都是基于某个地理坐标参考系，前面的章节已经介绍过，依靠地图投影将地球的三维表面转换为投影坐标参考系中的东向和北向（x 和 y）坐标。

这种转换过程中不可避免地会引入一些畸变。因此，地球表面的某些属性，如面积、方向、距离和形状，都会在这个过程中发生畸变。投影坐标系只能保留其中的一到两个属性。投影通常根据它们保留的属性进行命名：等面积投影保持面积不变，方位角投影保持方向，等距投影保持距离，共形投影保持局部形状。

投影类型主要分为三类：圆锥形、柱面形和平面形。在圆锥形投影中，地球表面沿着一个或两个正切线被投影到一个圆锥体上。在这种投影中，正切线上的畸变最小，畸变并随着与正切线的距离增加而增大。因此，它最适合用于中纬度地区的地图。柱面形投影将地球表面投影到一个圆柱体上。这种投影也可以通过沿着一个或两个正切线接触地球表面来创建。当绘制整个世界地图时，柱面形投影最常用。平面形投影将数据投影到一个在点或正切线处接触地球的平面上。它通常用于极地地区的制图。`st_proj_info(type = "proj")` 列出了 PROJ 库支持的可用投影列表。

### 2.4.3 R 中的 CRS

在 R 中描述 CRS 有两种主要方法：`epsg` 代码和 `proj4string` 定义。这两种方法各有优劣。`epsg` 代码通常较短，更容易记忆，它还暗含了一个唯一的，明确定义好的坐标参考系统。另一方面，`proj4string` 定义在指定不同参数（如投影类型、基准面和椭球体）方面具有更大的灵活性。[22] 使用 `proj4string` 可以指定许多不同的投影，并修改现有的投影。这也使 `proj4string` 方法更加复杂。而 `epsg` 则指向一个确切的特定的 CRS。

相关的 R 包支持丰富的坐标参考系，它们主要使用历史悠久的 PROJ[23] 库。除了

[22] 完整的 `proj4string` 参数列表可以在 https://proj4.org/ 找到。

[23] http://proj4.org/

在线搜索 EPSG 代码之外，另一种快速了解可用 CRS 的方法是通过 `rgdal::make_EPSG()` 函数，它输出一个包含可用投影的数据框。在进一步了解更多细节之前，值得学习如何在 R 中查看和过滤它们，这可以节省在互联网上搜索的时间。以下代码将交互式地显示可用的 CRS，可以过滤感兴趣的 CRS（读者可以尝试过滤 OSGB CRS）：

```
crs_data = rgdal::make_EPSG()
View(crs_data)
```

在 **sf** 中，可以使用 `st_crs()` 获取对象的 CRS。为了举例说明，我们先读取一个矢量数据集：

```
vector_filepath = system.file("vector/zion.gpkg", package = "spDataLarge")
new_vector = st_read(vector_filepath)
```

新对象 `new_vector` 是一个多边形，表示锡安国家公园的边界 (`?zion`)。

```
st_crs(new_vector) # get CRS
#> Coordinate Reference System:
#> No EPSG code
#> proj4string: "+proj=utm +zone=12 +ellps=GRS80 ... +units=m +no_defs"
```

在坐标参考系缺失或设置错误的情况下，可以使用 `st_set_crs()` 函数：

```
new_vector = st_set_crs(new_vector, 4326) # set CRS
#> Warning: st_crs<- : replacing crs does not reproject data; use st_transform for
#> that
```

警告消息告诉我们，`st_set_crs()` 函数不会将数据从一个 CRS 转换为另一个 CRS。`projection()` 函数可用于从 `Raster*` 对象中访问 CRS 信息：

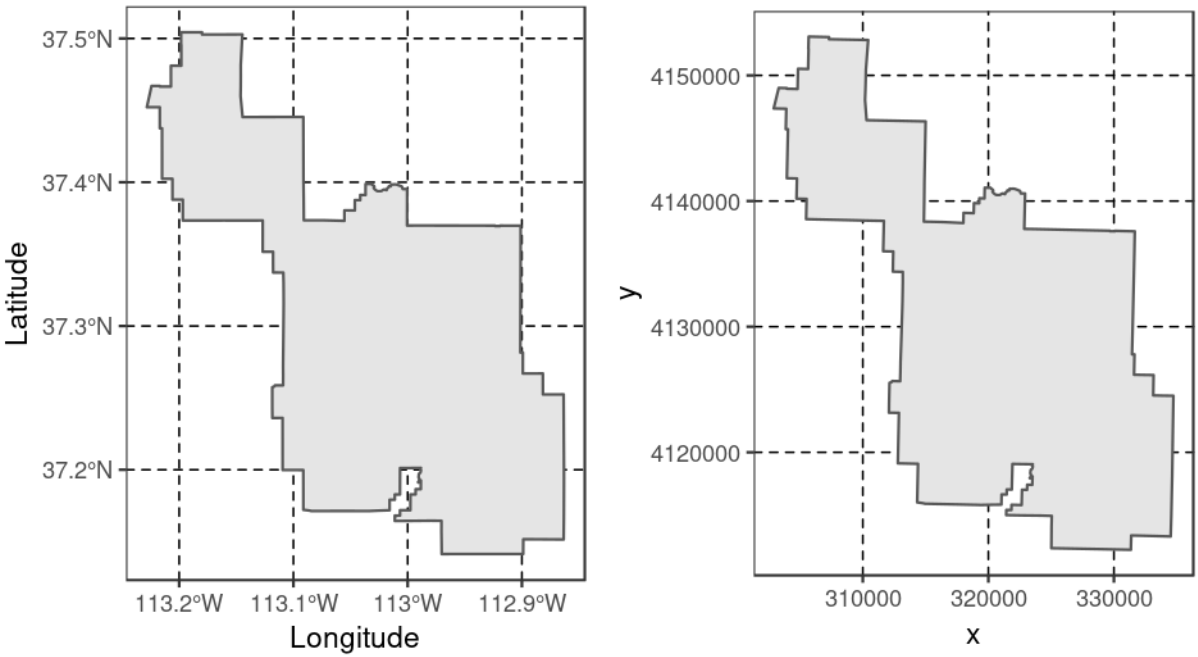


图 2.13: 矢量数据在地理坐标系（WGS 84；左）和投影坐标参考系（NAD83 / UTM zone 12N；右）下的示例。

```
projection(new_raster) # get CRS
#> [1] "+proj=longlat +datum=WGS84 +no_defs"
```

`projection()` 函数还可以用于为栅格对象设置 CRS。与矢量数据相比，主要区别在于，栅格对象仅接受 `proj4` 定义：

```
projection(new_raster) = "+proj=utm +zone=12 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0
                          +units=m +no_defs" # set CRS
```

我们将在第 **??** 章中更详细地介绍 CRS 以及如何从一个 CRS 投影到另一个 CRS。

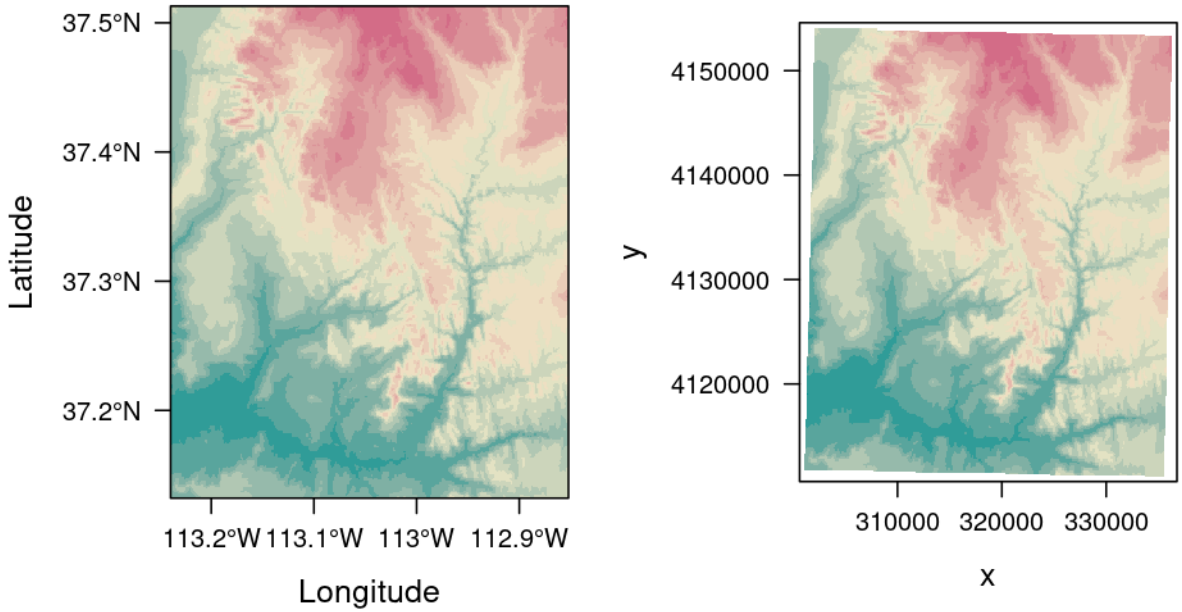


图 2.14: 矢量数据在地理坐标系（WGS 84；左）和投影坐标参考系（NAD83 / UTM zone 12N；右）下的示例。

## 2.5 测量单位

CRS 的一个重要特征是它们包含有关空间单位的信息。显然，知道一座房子的尺寸是以英尺还是米为单位非常重要，地图也是如此。在地图上添加比例尺是很好的实践经验，用来演示页面或屏幕上的距离与地面距离之间的关系。类似的，明确几何数据或像素的测量单位十分重要，后续的计算也要确保使用相同的测量单位。

`sf` 对象中几何数据的一个新特性是它们对单位的原生支持。这意味着在 **sf** 中进行的距离、面积和其他几何计算会返回带有 `units` 属性的值，该属性由 **units** 包定义 (Pebesma et al., 2016)。这可以避免由不同单位引起的混淆（大多数 CRS 使用米，有些使用英尺），并提供有关维度的信息。查看下面的代码演示，它计算了卢森堡的面积：

```
luxembourg = world[world$name_long == "Luxembourg", ]
```

```
st_area(luxembourg)
#> 2.41e+09 [m^2]
```

输出的单位为平方米($m^2$)。这些信息被存储为属性(读者可以执行 `attributes(st_area(luxembourg))` 查看)，可以用于后续涉及到单位的计算，例如人口密度（通常以每平方公里的人数为单位)。报告单位可以避免混淆。以卢森堡为例，如果单位未指定，则可能错误地假设单位为公顷。为了将巨大的数字转换为更易读的形式，可以把结果除以一百万（一平方千米等于一百万平方米)：

```
st_area(luxembourg) / 1000000
#> 2409 [m^2]
```

但是直接除以一百万，结果中的单位仍然是平方米，这明显是错误的。解决方案是使用 **units** 包设置正确的单位：

```
units::set_units(st_area(luxembourg), km^2)
#> 2409 [km^2]
```

在栅格数据的处理过程中，单位同样重要。不过到目前为止，**sf** 是唯一支持单位的空间包，这意味着处理栅格数据的人们应该谨慎地处理分析单位的更改（例如，将像素宽度从英制转换为十进制单位)。`new_raster` 对象（见上文）使用 WGS84 投影，单位是十进制的度。因此，它的分辨率也以十进制的度给出，这点你必须心里有数，因为 `res()` 函数仅返回数值。

```
res(new_raster)
#> [1] 0.000833 0.000833
```

如果我们使用 UTM 投影，单位将会改变。

```
repr = projectRaster(new_raster, crs = "+init=epsg:26912")
res(repr)
#> [1] 0.000833 0.000833
```

再次强调，`res()` 命令返回一个没有任何单位的数值向量，我们必须清楚 UTM 投影的单位是米。

## 2.6 练习

1. 对 `world` 数据对象的几何列使用 `summary()`。根据输出结果回答：
    - 它的几何类型?
    - 国家的数量?
    - 它的坐标参考系统（CRS)?
2. 运行代码，生成第 2.2.4 节末尾，图 2.5 中的世界地图。找出你生成的图与书中图像的两个相似之处和两个不同之处。
    - `cex` 参数的作用是什么（见 `?plot`)?
    - 为什么将 `cex` 设置为 `sqrt(world$pop) / 10000`?
    - 附加题: 尝试用不同的方式来可视化全球人口。
3. 使用 `plot()` 函数创建尼日利亚的地图（参见第 2.2.4 节)。
    - 调整 `plot()` 函数的 `lwd`、`col` 和 `expandBB` 参数。
    - 挑战：阅读 `text()` 函数的文档，并在地图里添加注释。
4. 创建一个名为 `my_raster` 的空 `RasterLayer` 对象，它有 10 列和 10 行。将新栅格赋予 0 到 10 之间的随机值并绘制它。
5. 从 **spDataLarge** 包中读取 `raster/nlcd2011.tif` 文件。关于此文件属性，你可以获取哪些信息?

提示：答案可以在 `https://geocompr.github.io` 上找到。

# 参考文献


Bivand, R., Keitt, T., and Rowlingson, B. (2018). *rgdal: Bindings for the 'Geospatial' Data Abstraction Library*. R package version 1.3-3.

Bivand, R. and Rundel, C. (2018). *rgeos: Interface to Geometry Engine - Open Source ('GEOS')*. R package version 0.3-28.

Gillespie, C. and Lovelace, R. (2016). *Efficient R Programming: A Practical Guide to Smarter Programming*. O'Reilly Media.

Grolemund, G. and Wickham, H. (2016). *R for Data Science*. O'Reilly Media.

Maling, D. H. (1992). *Coordinate Systems and Map Projections*. Pergamon Press, Oxford ; New York, second edition.

Pebesma, E. (2018). Simple features for R: Standardized support for spatial vector data. *The R Journal*.

Pebesma, E. and Bivand, R. (2018). *sp: Classes and Methods for Spatial Data*. R package version 1.3-1.

Pebesma, E., Mailund, T., and Hiebert, J. (2016). Measurement Units in R. *The R Journal*, 8(2):486–494.