python-holiday仓库分析

**软件包的结构**

holidays包，包含constants, helpers, holiday\_base, observed\_holiday\_base, registry, utils模块，以及calendars, countries, financial, groups子软件包。

holiday\_base是所有国家假期类的基类，是此软件包的核心，主要负责节假日填充逻辑代码。其继承字典类型，支持下标访问，支持多种格式的日期直接查询，内置了日期类型转换函数\_\_keytransform\_\_。\_populate函数将填充指定一年的假期。

observed\_holiday\_base继承holiday\_base，增加了调休逻辑。文件内还定义了各个星期的调休规则。

registry是本模块的加载器，用于实现模块的动态加载。

utils包含了一些静态函数，产生特定参数的假期字典，也可以获取本模块对各国家语言的支持情况。

helper包含一些辅助函数，进行参数类型的转换。

constants定义了有关日期、星期的常量，以及各种节日类型名称。

calendars软件包包括了世界各种日历，日历内包含各个节日对应年份的日期，以及get方法。还包括gregorian模块定义了公历日历的常用函数，custom模块包含StaticHoliday类方便加载自定义的日历，一般是特例节假日。

groups中的模块将一个或多个日历结合成群组，使群组中的日历可以被加载进字典。还包括了international，可以加载国际假期。

countries是本软件包的主要组成部分，包含各个国家的假期字典。每个国家假期对象都继承自HolidayBase，此外还可选继承自相应的日历群组（来自groups包），可选继承自StaticHoliday。例如，China由ObservedHolidayBase, ChineseCalendarHolidays, InternationalHolidays, StaticHolidays继承而来，其中StaticHolidays加载了自定义的ChinaStaticHolidays类，填补了特殊节假日（抗日70周年假日等）。

financial是金融假期包，其结构与countries相同。

**运行时流程**

本软件包宣称*“as fast and flexible as possible”*，多处运用到了即时构建（built on the fly）的思想。

为了直观地看到代码运行的流程，使用sys库中的settrace, gettrace进行语句追踪。将import和函数调用的语句分开读取，分析执行过程。

|  |
| --- |
| **from** sys **import** settrace, gettrace **import** linecache  f = open(**'trace\_record\_import.txt'**, **'w+'**, encoding=**'utf-8'**) proj\_prefix = **'D:\\Docs\\Course\_sync\\OpenSource\\Final\\demo\\python-holidays\\holidays\\'** last\_filename = **None   def** tracer(frame, event, arg):  old = gettrace()  settrace(**None**)  **global** last\_filename  **if** event == **'call'**:  filename = frame.f\_code.co\_filename  **if** filename.startswith(proj\_prefix):  lineno = frame.f\_lineno  func = frame.f\_code.co\_name  f.write(**f'call {**filename**}:{**lineno**}: {**func**}\n'**)   **if** event == **'line'**:  filename = frame.f\_code.co\_filename  *# if filename != last\_filename and filename.startswith(proj\_prefix):* **if** filename.startswith(proj\_prefix):  *# if True:* line\_no = frame.f\_lineno  line = linecache.getline(filename, line\_no).strip()  filename = frame.f\_code.co\_filename  f.write(**f'{**filename**}:{**line\_no**}: {**line**}\n'**)  last\_filename = filename  settrace(old)  **return** tracer   settrace(tracer) **import** holidays  f = open(**'trace\_record\_call.txt'**, **'w+'**, encoding=**'utf-8'**) holidays.CN()[**'2023-04-05'**] *# 清明节* |

得到两个追踪文件，从中可以看到执行过程。从加载软件包开始，import holidays并不将所有国家的假期全部加载进来，而是只加载必要的常量和工具类模块。使用registery中的EntityLoader进行动态加载模块。EntityLoader在初始化时只将模块名加载进变量空间，而不产生实体类，被调用时才会加载。以下代码来自EntityLoader的get\_entity和\_\_call\_\_。

|  |
| --- |
| **def** get\_entity(self) -> Optional[HolidayBase]:  *"""Return lazy-loaded entity."""* **if** self.entity **is None**:  self.entity = getattr(importlib.import\_module(self.module\_name), self.entity\_name)   **return** self.entity  **def** \_\_call\_\_(self, \*args, \*\*kwargs) -> HolidayBase:  *"""Create a new instance of a lazy-loaded entity."""* cls = self.get\_entity()  **return** cls(\*args, \*\*kwargs) *# type: ignore[misc, operator]* |

这样做使得不必要的国家假期不会加载到内存中，大大节省加载软件包的时间。

每个国家假期的填充同样是动态的。初始时holiday\_base只是一个空字典。每当查询到来，传入的关键字被\_\_ keytransform\_\_转换成date类型，接着，如果这个日期所在的年份没有被填充，则执行\_populate填充。一次只填充一年的假期。以下代码显示假期的动态填充。

|  |
| --- |
| *# Automatically expand for `expand=True` cases.* **if** self.expand **and** dt.year **not in** self.years:  self.years.add(dt.year)  self.\_populate(dt.year) |

这样进一步减少了调用时的执行时间。

**该软件包的难点**

其一是维护全球一百多个国家的节假日需要大量当地程序员，并且要及时跟进政策更新。对于免费开源仓库来说，很难保证各个国家的开发者长期坚持贡献。

另外，本地化工作比较艰难。本仓库对世界多语言的支持仍然有很大缺陷，每个节日翻译成多国语言需要语言专业方面的工作，完美维护需要大量人力。节日翻译至今仍有很多新问题，例如observed holiday被直译成了观察日，实际上这里的observe是庆祝的意思，更本土化的说法叫调休。

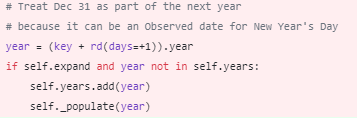
大多数节假日虽然日期固定、有确定的判断逻辑，但是每年仍有一些国家存在特殊庆祝的假期，以及政策调休的问题。这导致节假日判断问题不可能一劳永逸地解决。本开源仓库针对这个问题已经带来了很方便的方案，但仍需要永久维护。

**部分Bug分析**

#3 <https://github.com/vacanza/python-holidays/issues/3> RuntimeError when creating custom class with Dec 31st holiday

Bug现象：查询某一年的12月31日导致RuntimeError（无限递归调用爆栈）。

相关代码：来自修改前的\_\_keytransform\_\_函数



分析：此代码块在模块基类HolidayBase中的\_populate\_函数中。这段代码的作者认为12月31号可以作为第二年新年的调休日，所以直接放到下一年进行拓展。也就是说，对第year的拓展包含了第year-1年12月31日的假期。这里对year的转换看似巧妙，实则隐含漏洞。由于HolidayBase类的\_\_getitem\_\_和\_\_setitem\_\_的参数都需要调用\_\_keytransform\_\_处理，在\_populate函数中对节假日的设定隐含地调用了\_\_keytransform。那么这里的year就会被不断+1到下一年，执行新的填充，没有终止条件。

解决方案：将12月31仍作为当年节假日，对调休的判断做相应调整，不能在\_\_keytransform中做太多无关动作。

#4 <https://github.com/vacanza/python-holidays/issues/4> Error when combining holiday objects for different provinces

Bug现象：对同一个国家的不同地区的假期进行相加操作会出错（基类重复）。

相关代码：

|  |
| --- |
| **def** createHolidaySum(class1, class2):  **class** HolidaySum(class1, class2):  **def** \_populate(self, year):  class2.\_populate(self, year)  class1.\_populate(self, year)  **return** HolidaySum |

分析：需要进行两个国家或地区的假期融合时，原作者的代码通过继承原来的两个假期类，自然融合成新的假期类。当class1和class2不同时，确实没什么问题。但是假期类是由国家划分的，而同一国家的不同地区假期可能不同，但其均为同一个类，不能被重复继承。

解决方案：放弃继承融合，使用有状态的HolidaySum。HolidaySum存储一个holidays列表，像做加法一样加入两边的所有假期。原来的融合只关注类，而更改后加法作用与具体对象。

|  |
| --- |
| **def** createHolidaySum(h1, h2):  **class** HolidaySum(HolidayBase):  **def** \_\_init\_\_(self, \*\*kwargs):  self.holidays = []  **if** getattr(h1, **'holidays'**, **False**):  **for** h **in** h1.holidays:  self.holidays.append(h)  **else**:  self.holidays.append(h1)  **if** getattr(h2, **'holidays'**, **False**):  **for** h **in** h2.holidays:  self.holidays.append(h)  **else**:  self.holidays.append(h2)  HolidayBase.\_\_init\_\_(self, \*\*kwargs)   **def** \_populate(self, year):  **for** h **in** self.holidays[::-1]:  h.\_populate(year)  self.update(h)  **return** HolidaySum |

#1644 <https://github.com/vacanza/python-holidays/issues/1644> Chinese Spring Festival holiday in 2020 error

Bug现象：2020年中国春节假期因为疫情延长3天，代码中缺少实现。

分析：此类假期属于延期情况，软件包的StaticHolidays已经为此做了铺垫，在自定义类ChinaStaticHolidays中增加相应假期即可，不需要其他结构方面的改动。