

UnROOT.jl - 高能物理 (HEP) 数据分析

Jerry Ling (Harvard University/ATLAS Experiment)

Dec. 07, 2022

.root 文件概览

- `.root` 是 ROOT C++ 框架用来存放数据和 C++ Object 的一个文件格式。
- 由于可以存放的 Object 类型繁多，并且支持文件夹，所以比起文件更像是一个文件系统。
- 其中最为重要的是 `TTree` 一对撞机探测器的绝大多数物理数据都以这个类型存在与 `.root` 文件中。`TTree` 可以抽象地理解为一个 table, (row 又被称为"event"):

```
julia> my_ttree
```

Trigger	MET	lep_Pids
true	(E = 530.3, φ = 2.3)	[11, 13, -13]
true	(E = 752.1, φ = -0.7)	[11, -11]

处理 TTree 数据的挑战

眼尖的你可能已经注意到一些挑战，比如，我们的 table 无法被 numpy/pandas 处理—`my_ttree.lepPids` 这一 column (ROOT 称之为 branch) 在 numpy 下不合法。对于这种 vector of vectors (内 vector 不定长度)，我们称之为“Jagged array”。

在很多情况下就算解决了这个问题 (见 `awkward array`)，通常内存容量也不足以容纳整个 table。更重要的是很多数据分析的逻辑需要 event-level 的计算，无法用“columnar style”完成。Python HEP 生态里充满了为了解决这些问题的 `workaround`，让最终的代码难懂且难以维护。

在 ROOT 框架在 1995 年从 Fortran 切换到 C++ 后，大家分别在 Java, Go, 和 Python 里单独实现过 TTree 数据的处理，在我看来 Julia 比所有前者都更合适；UnROOT.jl 的目标是更快，但又对不精通 low-level 编程的物理学生更友好直接。

Julia 加入

我们继续用 Jagged array 做例子，因为它易懂却又不简单；它在高能物理数据里也颇具代表性——比如每个对撞 event 里产生的电子数量是不定的，有时候可能为 0，又有时候超过 10 个。（无法 pad 到定长，过于浪费空间）

对 Julia 而言其实 Jagged array 根本不是问题：

```
julia> a = [rand(5:7, i) for i=1:3]
3-element Vector{Vector{Int64}}:
 [6]
 [5, 7]
 [7, 6, 6]
```

这个数据结构在 Julia 里甚至也没有特别慢（这个在 Python 里约等于 10 万行代码，见 `awkward`）。

但我们可以更进一步，并且需要结合 table 具体是如何储存的：

硬盘上和内存里的 Jagged Array

`.root` 实际采取的策略更接近于:

```
julia> a = [[6], [5,7], [7,6,6]]
julia> data = [6,5,7,7,6,6]; offset = [1,2,4,7];
julia> using ArrayOfArrays # by Oliver Schulz
julia> VectorOfVectors(data, offset)
3-element VectorOfVectors{Int64, Vector{Int64}, Vector{Int64}, Vector{Tuple{}}}
 [6]
 [5, 7]
 [7, 6, 6]
```

Julia 的 composability 在于, DataFrames.jl / TypedTables.jl 并不在意每个 column 具体是什么 type 只要符合 `<: AbstractVector` 的 interface 即可, 甚至是 lazy 的 (not in RAM) 也无所谓!

Lazy Iteration 1

介于数据经常大于内存，并且有很多计算一定要在 event(row)-level 进行 (比如找一对轻子，质量最近接一个数值)，TTree 的设计不难理解。

如果你熟悉 Arrow/Parquet 的话，他们和 TTree 有一定相似性：

TTree	Parquet	Arrow/Feather
branch	column	field
--	--	array
cluster(not used)	row group	row group
--	column chunk	record batch
basket	page	buffer

Lazy Iteration 2

每一个 branch 概念上和一个 vector 一样，这也是为什么 TTree 几乎是一个“column-table”。但实际上每个 branch 又被分成了很多部分 (basket)，每个 basket 都分别被压缩，储存在文件中的不定位置。

这个设计主要有两个考量：

- 新增数据时不需要重新压缩并复写整个 branch (column)。
- 读取数据是不需要解压缩整个 branch (n.b. 内存通常不足以容纳)。

最终的数据分析流程大致是这样：

1. loop over event
2. 对于每个 event 涉及到的 branch，找到对应的 basket
3. 找到 basket 在文件中的位置并解压，cache 其结果（因为下一个 event 还要用到的可能性很高）

性能问题 1: row-iteration type stability

内容部分来自 [presentation](#) @ PyHEP 2021's Julia workshop.

Julia 有两种 type stability:

1. return type 能否纯粹由 input **types** 决定? (type stable)
2. 函数内的所有变量的类型是不是固定的? (type grounding)

2 通常可以影响到 1。


type un-groundness 导致 type instability

```
julia> function f(x)
    z = 0
    z += rand() > 0.5 ? x : x/2
    return z
end

julia> @code_warntype f(1)
Locals
  z::Union{Float64, Int64}
  @_4::Union{Float64, Int64}
Body::Union{Float64, Int64} # these are red
```

性能问题 1: row-iteration type stability

```
for evt in mytree
    compute(evt.Muon_pt)
    # more stuff
end
```

如果编译器不能推导出 `evt.Muon_pt` 的类型, 那  只能通过 dynamic dispatch (call) 来执行 `compute()`, 造成性能丢失。

有时类型不稳定还会导致“boxing/unboxing” (由于无法推导未知变量类别), 这会导致额外的 allocation, 同样会造成性能下降。

1 的解决方案

基本逻辑是这样的：

- 在读取 branch metadata 的时候 `eltype` 是已知的
- `LazyTree` 由 branch 组成
- 为了 `getproperty(evt, :Muon_pt)` type stable, 编译器必须要有 `Symbol → Type` 信息, a `NamedTuple`!
- 这也是 `TypedTables.jl` 的实现, wrapper of `NamedTuple`

让我们检查 type stability:

```
julia> function f(t)
    z = 0
    for evt in t z+=evt.nMuon end
    return z
end
```

1 的解决方案

```

1: 2:in164
2: 3: 4: 5: 6: 7: 8: 9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: 66: 67: 68: 69: 70: 71: 72: 73: 74: 75: 76: 77: 78: 79: 80: 81: 82: 83: 84: 85: 86: 87: 88: 89: 90: 91: 92: 93: 94: 95: 96: 97: 98: 99: 100: 101: 102: 103: 104: 105: 106: 107: 108: 109: 110: 111: 112: 113: 114: 115: 116: 117: 118: 119: 120: 121: 122: 123: 124: 125: 126: 127: 128: 129: 130: 131: 132: 133: 134: 135: 136: 137: 138: 139: 140: 141: 142: 143: 144: 145: 146: 147: 148: 149: 150: 151: 152: 153: 154: 155: 156: 157: 158: 159: 160: 161: 162: 163: 164: 165: 166: 167: 168: 169: 170: 171: 172: 173: 174: 175: 176: 177: 178: 179: 180: 181: 182: 183: 184: 185: 186: 187: 188: 189: 190: 191: 192: 193: 194: 195: 196: 197: 198: 199: 200: 201: 202: 203: 204: 205: 206: 207: 208: 209: 210: 211: 212: 213: 214: 215: 216: 217: 218: 219: 220: 221: 222: 223: 224: 225: 226: 227: 228: 229: 230: 231: 232: 233: 234: 235: 236: 237: 238: 239: 240: 241: 242: 243: 244: 245: 246: 247: 248: 249: 250: 251: 252: 253: 254: 255: 256: 257: 258: 259: 260: 261: 262: 263: 264: 265: 266: 267: 268: 269: 270: 271: 272: 273: 274: 275: 276: 277: 278: 279: 280: 281: 282: 283: 284: 285: 286: 287: 288: 289: 290: 291: 292: 293: 294: 295: 296: 297: 298: 299: 300: 301: 302: 303: 304: 305: 306: 307: 308: 309: 310: 311: 312: 313: 314: 315: 316: 317: 318: 319: 320: 321: 322: 323: 324: 325: 326: 327: 328: 329: 330: 331: 332: 333: 334: 335: 336: 337: 338: 339: 340: 341: 342: 343: 344: 345: 346: 347: 348: 349: 350: 351: 352: 353: 354: 355: 356: 357: 358: 359: 360: 361: 362: 363: 364: 365: 366: 367: 368: 369: 370: 371: 372: 373: 374: 375: 376: 377: 378: 379: 380: 381: 382: 383: 384: 385: 386: 387: 388: 389: 390: 391: 392: 393: 394: 395: 396: 397: 398: 399: 400: 401: 402: 403: 404: 405: 406: 407: 408: 409: 410: 411: 412: 413: 414: 415: 416: 417: 418: 419: 420: 421: 422: 423: 424: 425: 426: 427: 428: 429: 430: 431: 432: 433: 434: 435: 436: 437: 438: 439: 440: 441: 442: 443: 444: 445: 446: 447: 448: 449: 450: 451: 452: 453: 454: 455: 456: 457: 458: 459: 460: 461: 462: 463: 464: 465: 466: 467: 468: 469: 470: 471: 472: 473: 474: 475: 476: 477: 478: 479: 480: 481: 482: 483: 484: 485: 486: 487: 488: 489: 490: 491: 492: 493: 494: 495: 496: 497: 498: 499: 500: 501: 502: 503: 504: 505: 506: 507: 508: 509: 510: 511: 512: 513: 514: 515: 516: 517: 518: 519: 520: 521: 522: 523: 524: 525: 526: 527: 528: 529: 530: 531: 532: 533: 534: 535: 536: 537: 538: 539: 540: 541: 542: 543: 544: 545: 546: 547: 548: 549: 550: 551: 552: 553: 554: 555: 556: 557: 558: 559: 560: 561: 562: 563: 564: 565: 566: 567: 568: 569: 570: 571: 572: 573: 574: 575: 576: 577: 578: 579: 580: 581: 582: 583: 584: 585: 586: 587: 588: 589: 590: 591: 592: 593: 594: 595: 596: 597: 598: 599: 600: 601: 602: 603: 604: 605: 606: 607: 608: 609: 610: 611: 612: 613: 614: 615: 616: 617: 618: 619: 620: 621: 622: 623: 624: 625: 626: 627: 628: 629: 630: 631: 632: 633: 634: 635: 636: 637: 638: 639: 640: 641: 642: 643: 644: 645: 646: 647: 648: 649: 650: 651: 652: 653: 654: 655: 656: 657: 658: 659: 660: 661: 662: 663: 664: 665: 666: 667: 668: 669: 670: 671: 672: 673: 674: 675: 676: 677: 678: 679: 680: 681: 682: 683: 684: 685: 686: 687: 688: 689: 690: 691: 692: 693: 694: 695: 696: 697: 698: 699: 700: 701: 702: 703: 704: 705: 706: 707: 708: 709: 710: 711: 712: 713: 714: 715: 716: 717: 718: 719: 720: 721: 722: 723: 724: 725: 726: 727: 728: 729: 730: 731: 732: 733: 734: 735: 736: 737: 738: 739: 740: 741: 742: 743: 744: 745: 746: 747: 748: 749: 750: 751: 752: 753: 754: 755: 756: 757: 758: 759: 760: 761: 762: 763: 764: 765: 766: 767: 768: 769: 770: 771: 772: 773: 774: 775: 776: 777: 778: 779: 780: 781: 782: 783: 784: 785: 786: 787: 788: 789: 790: 791: 792: 793: 794: 795: 796: 797: 798: 799: 800: 801: 802: 803: 804: 805: 806: 807: 808: 809: 810: 811: 812: 813: 814: 815: 816: 817: 818: 819: 820: 821: 822: 823: 824: 825: 826: 827: 828: 829: 830: 831: 832: 833: 834: 835: 836: 837: 838: 839:
```

1 的解决方案

滚屏五次后:

```
Body::Int64 #<----- stable
|   %2  = t::LazyTree with 1479 branches:
.... # pages and pages of craziness
|   %10 = z::Core.Const(0)
|   %11 = Base.getproperty(evt, :nMuon)::UInt32 #<-----stable
|       (z = %10 + %11)
└       goto #4
...
4 ...      return z
```

1 解决方案利弊

这个方法的主要弊端是大幅增加了编译器的负担，因为有时候我们的 `TTree` 宽度会有 1000+；这个问题在 1.8 里有所改善。这一弊端也是为什么 `DataFrames.jl` 不在类型里 `encode` 每个 `column` 的类型信息，但那样做的代价是 `row-iteration` 性能不佳：

- <https://discourse.julialang.org/t/fast-iteration-over-rows-of-a-dataframe/24612>
- <https://discourse.julialang.org/t/how-to-speed-up-the-for-loop-with-dataframe-access/79447>

`DataFrames.jl` 作者的回应：

<https://bkamins.github.io/julialang/2022/07/08/iteration.html>.

性能问题 2: Lazy and subsetting of branches

虽然一个 TTree 有时候可以有上千个 branch, 但是用户不一定需要所有 branch, 有两处可以实现这个功能:

1. 创建 LazyTree 的时候让用户决定什么 branch 需要
2. 在 event loop 内, 不对没有被用户用到的 branch 开销 I/O

UnROOT.jl 对这两种模式都有支持!

```
# 1
LazyTree(path_or_ROOTFile, ["MET", r".*Muon.*"])

# 2
for evt in tree
    evt.nMuon # only I/O for this `nMuon` branch
end
```

Thread-safety

之前提到由于 `basket` 需要解压缩，所以我们至少会保留最近一个 `basket` 在内存里；为了让 `event loop` 可以并行，我们只要保证 `basket` 的解压后的缓存是 `thread-local`(用 `threadid()`) 即可：

```
Threads.@thread for evt in mytree
    evt.nMuon < 4 && continue
    # more stuff
end
```

Julia 默认分割恰好符合我们的 I/O 模型—相邻的 `event` 几乎永远属于同一个 `basket`，我们希望分配给同一个 `thread`。例如我们有 100 个 `event` 并且 `Threads.nthreads() == 4`，那么第一个 `thread` 会处理 `1:25`，第二个 `thread` 处理 `26:50`，以此类推。

Benchmark 结果

测试例子基于 4-muon to Higgs rejecting Z candidates 的一个处理, 数据来自 CMS Open Data.

- Repo: https://github.com/Moelf/UnROOT_RDataFrame_MiniBenchmark
- 基于 C++ ROOT 反馈, 接近最优 C++ Loop ([ref](#))

Language	1st Run	2nd Run (JIT time excluded)
Julia	15.99 s	15.05 s
PyROOT RDF	43.74 s	--
C++ ROOT Loop	19.53 s	--
Compiled RDF	24.94 s	--
Julia 4-threads	4.72 s	4.61 s
Compiled RDF 4-threads	10.23 s	--

展望 Julia 1.9: 免费的午餐

Julia 1.9 (i.e. 等 PR [#47184](#)), time to first event 将大幅缩短:

```
using UnROOT
LazyTree(UnROOT.samplefile("tree_with_jagged_array.root"), "t1") ▷ show
```

1.8.2

Executed in	7.19 secs	fish	external
usr time	7.32 secs	0.00 micros	7.32 secs
sys time	1.46 secs	515.00 micros	1.46 secs

With the PR

Executed in	4.05 secs	fish	external
usr time	4.26 secs	11.66 millis	4.25 secs
sys time	1.43 secs	3.16 millis	1.42 secs

(这也是我喜欢 Julia 的一个理由, 有时候莫名其妙代码就变快了!)

End