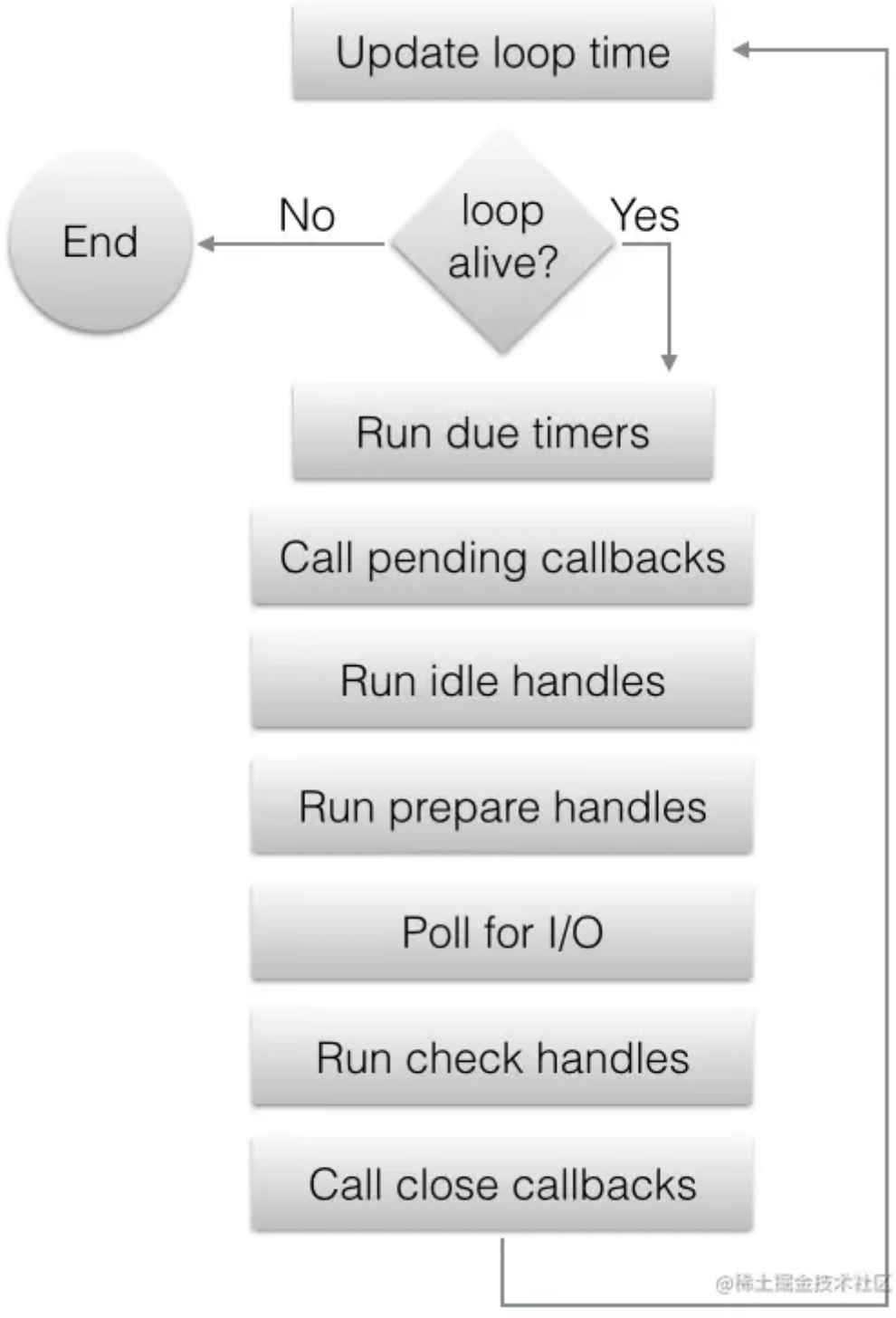
在 Node.js 中，异步体现在方方面面。不一定所有异步都与 I/O 有关。 `Timer` 相关的 API 就是与 I/O 无关的异步 API。

还记得之前在事件循环相关章节中提到的 `libuv` 设计概览吗?



在其他几层。

`setTimeout` 到底是哪的函数？怎么实现的？

很多人可能都有一种误解，一个 JavaScript 运行时天生就应该有 `setTimeout()` 这些函数。因为它实在是太常用了，在 Web 应用中真的是无处不在，在各种跨端场景中，也是最基础的 API 之一。Node.js、Deno 这些运行时都有。**它就是引擎中提供的 API！**

你要这么想就错了。 `setTimeout` 并非 ECMAScript 规范，而是 Web API。之所以 Node.js、Deno 这些都有，是因为这个 API 太深入人心了，不实现不舒服斯基。我们可以粗暴地理解为，ECMAScript 不定义穿插于事件循环不同 Tick 间的 API。

而问 `setTimeout()` 怎么实现的，这个问题也很泛。这个问题在知乎上有：JS 中 setTimeout 的实现机理是什么？

其实，抛开运行时问实现的都是耍流氓。下面的回答也都提了类似的这点，不同运行时实现机制不一样。各种答案里也给出了各种不同运行时的实现方式。

比如貘大给出了 Chromium 里面怎么实现的；Justjavac 介绍了 Deno 是基于红黑树搞的定时器；还有大佬披露了 React Native 的定时器是基于优先队列的。而我在当时开发的运行时里，也是用 C 写了个优先队列在事件循环里搞定计时器的。

一千个读者有一千个哈姆雷特，一千个厨师炒一千碗粉的味道也各自不同。

Node.js 的 `setTimeout()` 并不完全按规范来实现。比如 `setTimeout()` 的返回值 `id` 在规范中，是一个整数（`integer`），而 Node.js 实际上返回的是一个对象。还有一个就是网上常有的八股问题，如果嵌套层级大于 5，超时时间小于 4，则定义超时时间最小为 4——这个八股问题对 Node.js 同样不生效。

在 Node.js 中，`setTimeout()` 实际上是生成一个 `Timeout` 类的实例，在其内部控制定时器并触发回调，并且这个函数返回的也是该实例，并不是整数。

```js
function setTimeout(callback, after, arg1, arg2, arg3) {
  // ...

  const timeout = new Timeout(callback, after, args, false, true);
  insert(timeout, timeout._idleTimeout);

  return timeout;
}
```

这里有几个点好提。第一个是 `args` 的生成。我们看到 Node.js 声明 `setTimeout` 的时候，其参数除了前两个外，后面还额外附加了 `arg1`、`arg2` 和 `arg3`。这个是根据经验定的 3 个调用参数，基本上的情况下，调用 `setTimeout()` 回调函数的参数不会超过 3 个，于是这里显示声明 3 个，为生成 `args` 用。

```js
  let i, args;
  switch (arguments.length) {
    // fast cases
    case 1:
    case 2:
      break;
    case 3:
      args = [arg1];
      break;
    case 4:
      args = [arg1, arg2];
      break;
    default:
      args = [arg1, arg2, arg3];
      for (i = 5; i < arguments.length; i++) {
        // Extend array dynamically, makes .apply run much faster in v6.0.0
        args[i - 2] = arguments[i];
      }
```

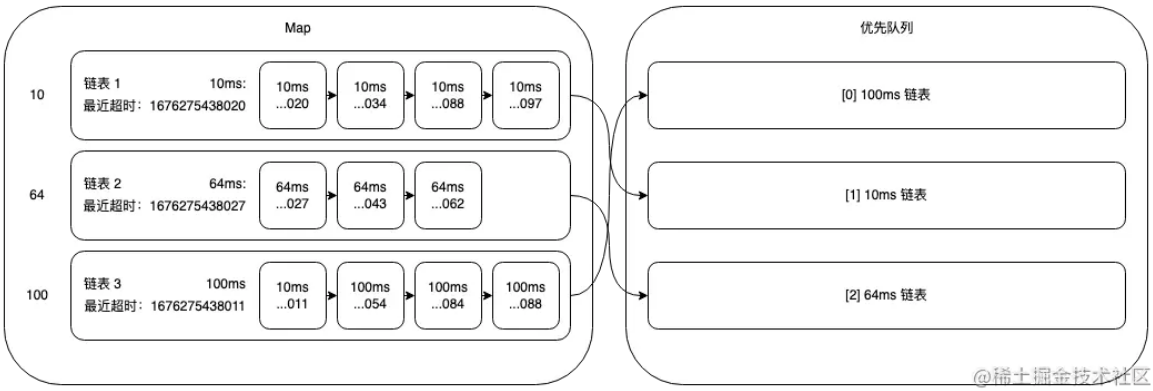
如果刚好是 3 个参数以内的，在生成 `args` 的时候会直接声明一个定长数组，这比后续不断去扩展数组来得快。如果再超长了，那再去动态扩展 `args` 数组长度。

两个关键对象：`Map` 与优先队列

后面那个 `insert()` 函数，是把新生成的 `Timeout` 实例插入两个对象中：

1. 一个 `Map`，其键名为 `timeout` 的值，键值为一条存储同 `timeout` 值的所有 `Timeout` 实例的链表，链表中额外存了一个最近的最终超时时间，用于做一些判断；
2. 一个优先队列，以链表的最终超时时间点为优先队列的权重。

这两个对象看起来像这样：



> **图中的时间为为方便表示而随机拟的时间，所以有瑕疵。事实上通常来说，一个 `10ms` 的链表中是不可能存在头尾节点差距超过 `10ms` 的情况。除非是一个同步的大循环（比如该同步大循环会耗时 100ms）中不断插入 `Timeout` 。如：**

```js
const timer1 = setTimeout(() => {
  console.log(1);
  const timer3 = setTimeout(() => {
    console.log(3);
  }, 10);
  console.log(timer3);
}, 10);
console.log(timer1)
const now = Date.now();
```

```
const timer2 = setTimeout(() => {
  console.log(2);
}, 10);
console.log(timer2);
```

从图中看，这是一个拥有 10 个 `Timeout` 的场景的两个对象表示图。10ms 定时器的最近过期时间为 1676275438020，64ms 的为 1676275438027，而 100ms 的则为 1676275438011。这个时候，优先队列中，以最近超时为权重，自然就是 100ms 的链表、10ms 链表、64ms 链表这个顺序了。

> 上方优先队列的图只是为了简易展示数据便于理解，真实优先队列内部的数据结构并不是这样的。

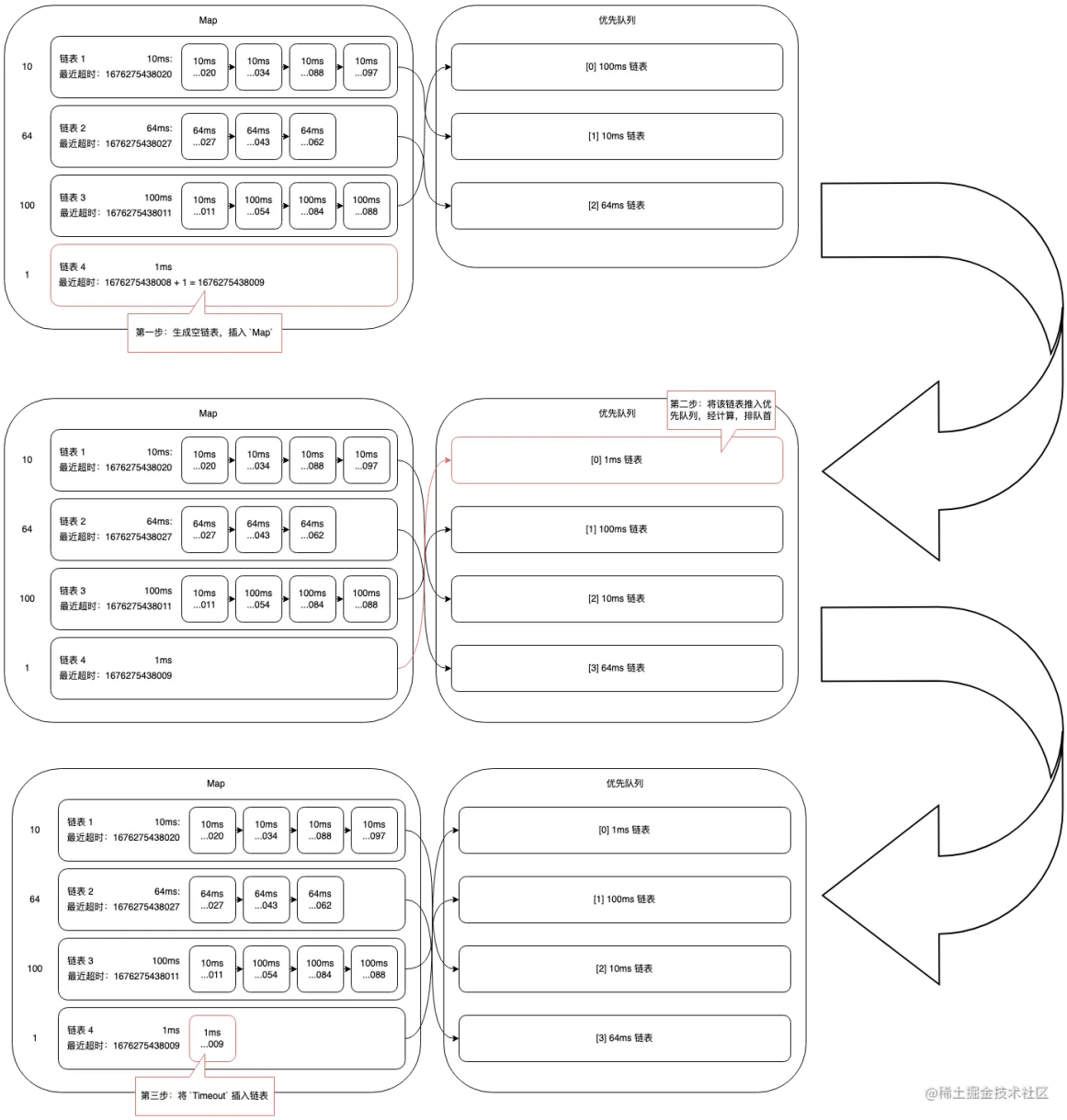
Node.js 这么管理 `Timeout` 对象是为了在使用的时候方便些。

```js
function insert(item, msecs, start = getLibuvNow()) {
  // Truncate so that accuracy of sub-millisecond timers is not assumed.
  msecs = MathTrunc(msecs);
  item._idleStart = start;

  // Use an existing list if there is one, otherwise we need to make a new one.
  let list = timerListMap[msecs];
  if (list === undefined) {
    debug('no %d list was found in insert, creating a new one', msecs);
    const expiry = start + msecs;
    timerListMap[msecs] = list = new TimersList(expiry, msecs);
    timerListQueue.insert(list);

    if (nextExpiry > expiry) {
      scheduleTimer(msecs);
      nextExpiry = expiry;
    }
  }

  L.append(list, item);
}
```

上面这段代码就是 `insert()` 逻辑了。我们忽略前面几行代码，无关紧要。 `let list = ...` 这句就是从上面图中那个 `Map` 对象中拿到对应 `timeout` 的链表。若链

插，并同时往优先队列中一插，就完成了。我们先略过 `scheduleTimer()` ，直接到最后，如果事情都做完了，我再把传进来的 `Timeout` 对象往刚才拿到或者生成出来的链表中插到最后一位即可。

比如，若现在时间是 1676275438008，我现在往其中插入一个 1ms 的 `Timeout` ，其对象的变化如下：



> **思考：** 为什么在 `Map` 中，用链表就可以了，不用优先队列或其他一些有序列表？
>
> **提示：** 时间是一往无前不能回头的，同个 `timeout` 值的 `Timeout` 在不同时刻创建时，其超时顺序是怎么样的？

就“一个”定时器

我们前面提到了，一次 `setTimeout()` 就生成一个 `Timeout` 实例，实际上 `setInterval()` 同理，只是 `Timeout` 里面的“是否循环”字段不一样而已。这样看起来，一次 `setTimeout()` 我们就多了一个定时器，至少表面上看起来是这样的。

你要说 JavaScript 层，的确看起来如此，而且被 `Map` 和优先队列管理起来。实际上，这只是 JavaScript 侧的类，并不实际参与调度，只是在真正 libuv 定时器调度的时候，被引用一下而已。真正在 libuv 层，定时器就一个。

`uv_timer_t`

libuv 中，定时器是以定时器句柄（ `uv_timer_t` ）的形式进行操作的。而在事件循环中激活一个定时器，它是靠这个 API 的：

```c
int uv_timer_start(uv_timer_t *handle, uv_timer_cb cb, uint64_t timeout, uint64_t repeat
```

我们通过 `uv_timer_start` 对一个已初始化的定时器进行激活。事件循环运行过程中，它会在 `timeout` 时间到之后触发传入的 `cb` 回调函数。最后一个参数代表“是否循环”，即触发一次之后，是否等 `timeout` 时间到会再次出发。

你很聪明，这个 `repeat` 参数看起来的确可以用于区分 `setTimeout()` 和 `setInterval()` 。不过 Node.js 中的 `Timer` 机制并非如此，通过 `uv_timer_start()` 中的 `repeat` 参数指定并不适用。

Node.js 中针对 `Timer` 的定时器

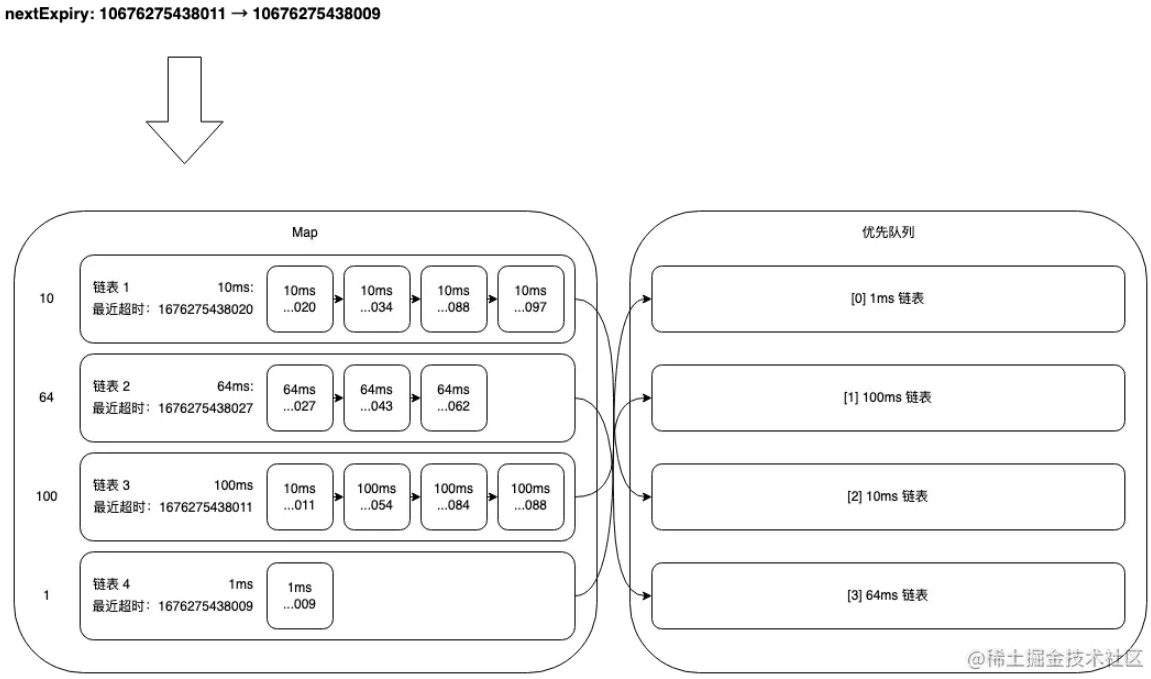
Node.js 中，针对 `Timer` 的单个定时器被存在 `Environment` 类中。 `Environment` 是针对 Node.js 中每个 V8 的 `Isolate` 都存在一份的环境相关类，里面有很多相关的工具、数据等等。因为 `Timer` 只需要一份，所以作为执行数据存在 `Environment` 中也是自然的。

```c++
class Environment : public ... {
  ...
```

这个 `timer_handle_` 怎么初始化就不提了，不重要。重要的是，还记得之前那段 `insert()` 代码吗？里面有这段我故意跳过没讲的条件判断：

```js
if (nextExpiry > expiry) {
  scheduleTimer(msecs);
  nextExpiry = expiry;
}
```

这里就是做这个定时器调度的代码了。首先有个 `nextExpiry` 变量，存的是所有 `Timeout` 中，最近要过期的时间。也就是说，如果回到上面那个图，中间应该还有一步，就是用 `expiry` 替换掉这个 `nextExpiry` 。



> **思考：** 为什么这个替换只出现在“ `Map` 中不存在该链表”的情况下做判断？
>
> **提示：** 还是跟之前一样，时序相关。

上面的代码中，我们除了替换 `nextExpiry` ，还要更新那个 `timer_handle_` 的触发事件，也将其更新为最新的 `nextExpiry` 。这样，定时器就会在更近的这个新时间点触发了。

```c++
void Environment::ScheduleTimer(int64_t duration_ms) {
  if (started_cleanup_) return;
  uv_timer_start(timer_handle(), RunTimers, duration_ms, 0);
}
```

不管第一行，是边界条件。所以里面实际上最终是执行我们刚才提到的 `uv_timer_start()` 。如果这个定时器处于未激活状态，则将该定时器触发事件改为传进来的最新时间并激活；如果定时器已经处于激活状态，则直接通过 `uv_timer_start()` 将定时器的触发事件更新为最新时间。逻辑很简单：**我只需要一个定时器触发最近一次 `Timeout` ，触发后我再取一个最近的时间开始定时。**

定时器回调

在 `timer_handle_` 这个定时器中，一旦定时器被触发，执行的始终是传进去的 `RunTimers()` 回调函数。这个函数也是写在 `Environment` 中的。它的作用是经过一系列逻辑后，调用到 JavaScript 侧的定时回调函数，在那个函数中才会去找对应的 `Timeout` 并触发对应的真实回调函数，即该 `RunTimers()` 最终达到的是一个 Dispatcher 的效果。

在 `RunTimers()` 中，参数规定是当前触发当前定时的 `uv_timer_t` 句柄。并且，该函数是可以以简单函数指针的形式传给 `uv_timer_start()` ，它得是一个 `static` 静态方法。所以在这个内部是无法通过 `this` 来代表对应的 `Environment` 实例的，Node.js 就用了一个 `Environment::from_timer_handle()` 静态方法来让我们可以通过对应 `uv_timer_t` 获取其所属的 `Environment` 实例。

```c++
void Environment::RunTimers(uv_timer_t* handle) {
  Environment* env = Environment::from_timer_handle(handle);
  ...

  Local<Object> process = env->process_object();
  ...

  Local<Function> cb = env->timers_callback_function();
  MaybeLocal<Value> ret;
  Local<Value> arg = env->GetNow();
  do {
    TryCatchScope try_catch(env);
    try_catch.SetVerbose(true);
```

```
  ...

  int64_t expiry_ms =
      ret.ToLocalChecked()->IntegerValue(env->context()).FromJust();

  uv_handle_t* h = reinterpret_cast<uv_handle_t*>(handle);
  if (expiry_ms != 0) {
    int64_t duration_ms =
        llabs(expiry_ms) - (uv_now(env->event_loop()) - env->timer_base());

    env->ScheduleTimer(duration_ms > 0 ? duration_ms : 1);

    if (expiry_ms > 0)
      uv_ref(h);
    else
      uv_unref(h);
  } else {
    uv_unref(h);
  }
}
```

这里面，我删除了部分代码，保留骨干部分。可以看出，逻辑就是从拿到一个 `cb` 函数，并去执行该回调函数，传入的参数为当前时间，并且 `this` 对象为 `process` 。 `cb` 的返回值是由回调函数计算出来的下一次定时器触发时间。若触发时间为 `0` ，则说明之后没有 `Timeout` 定时器了，就不需要重新调度；若非 `0` ，则通过一定逻辑计算出真正下一次触发所需的时刻，并通过调用之前解释过的 `env->ScheduleTimer()` 函数重启定时器。

```c++
Local<Object> process = env->process_object();
Local<Function> cb = env->timers_callback_function();
```

这两行代码，主要就是获取 `process` 对象，以及获取一个事先注册到 `Environment` 中的 JavaScript 侧的定时器回调函数（也就是之前提到过产生 Dispatcher 效果的函数）。

这个 `timers_callback_function()` 是在 Node.js 启动的时候注册进去的。注册了两个个函数，分别是 `processImmediate()` 和 `processTimers()` ，分别对应处理哪个函数大家从名字应该能看出来。当下我们先只剖析 `processTimers()` 这个 `Dispatcher` 。

```
  let list;
  let ranAtLeastOneList = false;
  while ((list = timerListQueue.peek()) != null) {
    if (list.expiry > now) {
      nextExpiry = list.expiry;
      return timeoutInfo[0] > 0 ? nextExpiry : -nextExpiry;
    }
    if (ranAtLeastOneList)
      runNextTicks();
    else
      ranAtLeastOneList = true;
    listOnTimeout(list, now);
  }
  return 0;
}
```

逻辑很简单，不断从 `timerListQueue` 这个优先队列中获取队首元素，也就是过期时间最近的那条链表。由于逻辑特性，链表头肯定是时间最近的元素。

判断一下，链表的过期时间是否大于当前时间。如果大于当前时间，则说明这个 `Timeout` 还未轮到执行，于是我将 `nextExpiry` 用该链表的过期时间替换。我们之前提到，该回调函数返回值是下一次过期时间。注意这里的返回先判断了一个 `timeoutInfo[0]` 的值大小，并以此返回正负的值。

实际上在 C++ 侧的 `RunTimers()` 中始终用的是该值的绝对值，所以正负只是一个简易的判断标识，用于判断是否需要对 libuv 进行 Reference 或者 Unreference 操作，以让 Node.js 决定需不需要“及时退出”。这个事情不在主干上，不重要，暂且不提。所以先忽略正负吧。总之，如果 `Timer` 还不需要执行，则返回 `nextExpiry` 以让 `RunTimers()` 开启下一轮的 libuv 的定时器。

如果链表过期时间小于等于当前时间，则说明在当前状态下，该 `Timeout` 是需要被触发的。由于时间的不精确性，如时间循环卡了一下，导致一下子过了好几毫秒，而在这之前有好几条链表都会过期，那么我们就需要在一次 `processTimers` 里面持续执行 `Timeout` 直到获取的 `Timeout` 未过期。所以这里一整套逻辑都是被一个 `while` 所包围。

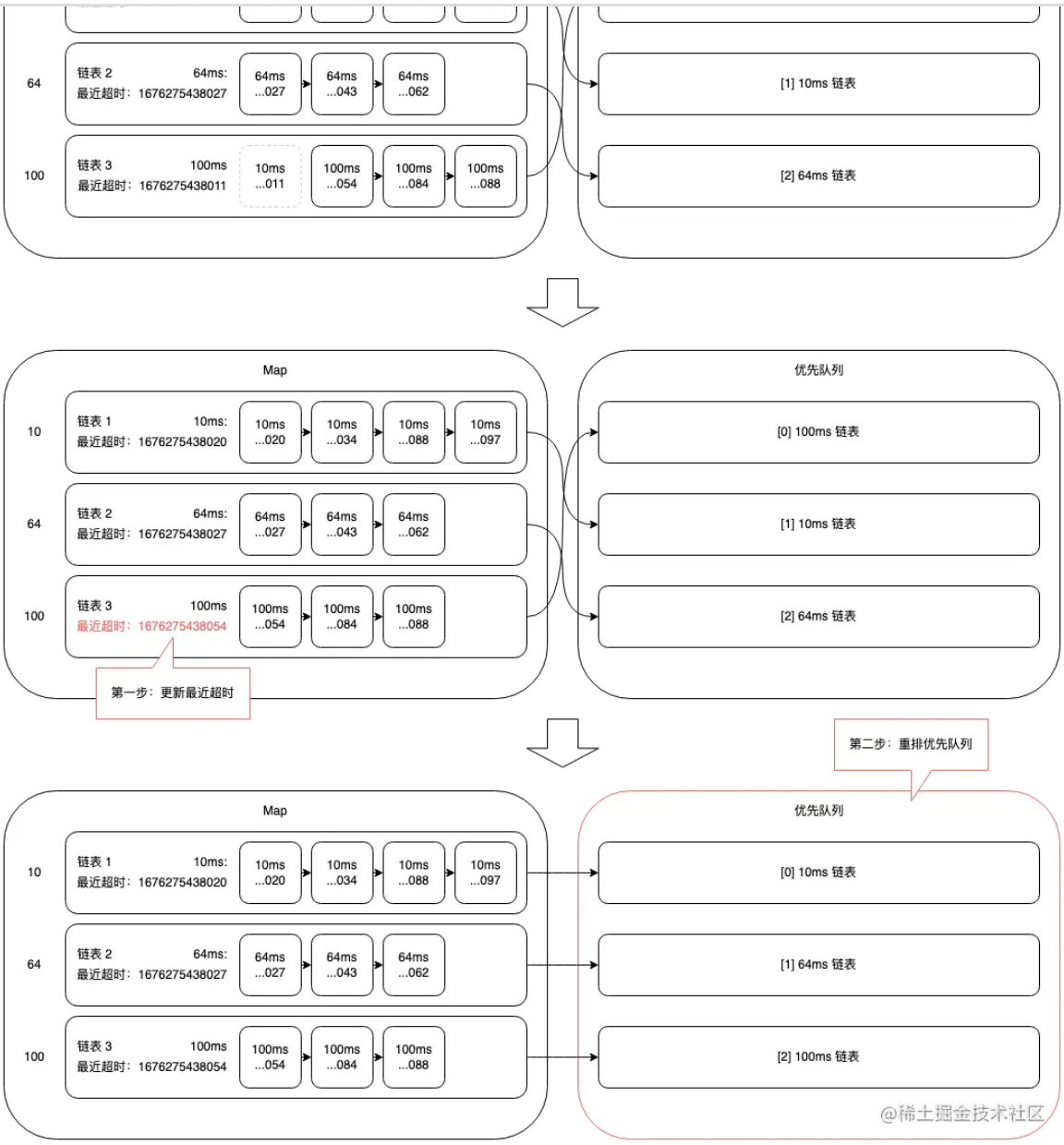
在执行 `Timeout` 之前，先判断一下当前的 `while` 里面是不是已经执行过至少一个 `Timeout` 了。若未执行过，则直接执行；若已经执行过，则在 Node.js 的语义中已

`runNextTicks()` 里面主要做的事情就是去处理微任务、 `Promise` 的 `rejection` 等。毕竟在 Node.js 的语义中，一个 Tick 结束要做的事情有好多。

到下一个 Tick 之后，就可以触发 JavaScript 侧的 `Timeout` 了。代码有点长，就不放出来了，有兴趣可以自己读代码，也可以听我瞎逼逼。

触发的逻辑是在 `listOnTimeout()` 中。它所做的事情是从刚才拿出来的链表中不断获取第一个 `Timeout` 。我们之前说过了，由于时间一往无前的特性，第一个 `Timeout` 肯定是该链表中最早触发的 `Timeout` ，然后依次往后排。每次获取首个 `Timeout` 都先判断确认一下，该 `Timeout` 是否应该在当前时间点触发。

若不该，则说明该链表能处理的 `Timeout` 都处理完了。接下去扫尾，更新这条链表的最早过期时间，也就是当前 `Timeout` 的过期时间，更新完之后，再重排一下优先队列。比如最早的那张图，若当前时间是 `1676275438013` ，那么执行 `Timeout` 是这样的：



若是当前 `Timeout` 的确已经可以被触发的话，仍旧先走一遍 `runNextTicks()` 的逻辑，然后从链表中将当前 `Timeout` 移除。做了一系列额外逻辑后（如操作 `Timeout` 的 Reference 值等），就是通过 `try-catch` 去执行 `Timeout` 实例的 `_onTimeout()` 方法了。这里的 `_onTimeout()` 暂时不研究，它就是某个 `Timeout` 触发后真正要执行的函数，并且它内部会调用 `setTimeout()` 时传进去的回调。

执行完 `Timeout` 后，先判断该 `Timeout` 是否需要重复执行（即 `setInterval()`）。若需要重复执行，则将该 `Timeout` 实例以新的一些参数重新调用 `insert()` 回到链表中。因为该链表已存在，所以不需要生成新的，又因为当前 `Timeout` 处理完之后，该条链表最后的时间不会超过 `100ms`（即当前链表对应的 `timeout`），所以新插入的 `Timeout` 在链表尾不会影响现有的时序。

接下去就开始下一条循环，从链表中再获取下一条 `Timeout` 重复上面的操作。如果链表空了，则退出。退出之后在外层循环实际上就是 Node.js 继续从优先队列中获取再继续判断了。

简单总结这个流程就是：从优先队列拿最早的链表，里面元素一个个都做了，一直做到元素未超时，再把链表放回去重排优先队列；然后再从优先队列拿最早的链表，里面元素一个个都做了，一直做到元素未超时，再把链表放回去重排优先队列，一直做到优先队列里面第一条链表也未超时为止。

我称之为**薅羊毛算法（我瞎编的）**。这就像是薅羊毛。比如有红黑黄三种羊的羊圈，每个羊圈若干羊。每次都选一只所有羊中羊毛长得最好最多的羊，进入它的羊圈，把羊毛薅秃了，并按羊毛多少，把这个圈里的羊依次薅一遍，直到没羊毛可薅。然后出羊圈后，再用同样逻辑找羊圈，进去薅一圈。这个时候可能第一个羊圈的毛又长差不多了，这个时候再用同样逻辑选羊圈薅羊毛。直到所有羊毛都薅没了，就等下一次可以薅的时候。

~~薅羊毛（不是）~~

或者也可以称其为**劫匪算法（还是我瞎编的）**。劫匪从一堆珠宝店里选出有最贵珠宝的店，进去从贵到便宜把能抢的都抢了。然后换一家继续。第二家最贵的不一定比第一家最便宜的便宜，正如一条链表内可触发的最迟的定时器不一定比后一条链表的可触发的最早的定时器要早。

这种做法虽然**说不定**效率比把所有 `Timeout` 都归为一个大的优先队列好，但是在极端情况下会出一些岔子。根据上面的算法，我们很容易能倒推出下面这段代码：

```
setTimeout(() => {
  console.log(1);
}, 10);
setTimeout(() => {
  console.log(2);
}, 15);
let now = Date.now();
while (Date.now() - now < 100) {
  //
}
setTimeout(() => {
  console.log(3);
}, 10);
now = Date.now();
while (Date.now() - now < 100) {
  //
}
```

照理说，按实际 `Timeout` 触发时间的迟早进行排序触发，3 个 `Timeout` 触发时机分别为 `10` 、 `15` 、 `110` ，所以顺序应该一次是 `1` 、 `2` 、 `3` 。哪怕第一个 Tick 是经过 `200` 毫秒后才结束，开始触发第一个 `Timeout` 的时刻已经是 200 毫秒之后了，三个 `Timeout` 都应该马上被触发。但这种情况下，触发顺序不再是 `1` 、 `2` 、 `3` ，而是 `1` 、 `3` 、 `2` 。至于为什么，想想羊毛是怎么薅的你就晓得了。**第三只羊在第一只羊被薅之后，就因为与它同处一个羊圈而先于第二只羊被薅了。**

哦对了，最后忘了说，如果链表中的 `Timeout` 全过完了，且没有因为 `repeat` 被重新插回去，也就是说最后链表空了的情况下，在这个函数的最末尾会从 `Map` 以及优先队列中把该链表删除。这就好比，羊毛薅完了还会长，但是薅完羊毛里面的羊都死光了，那羊圈就没必要留着了。

小结

`setTimeout` 并不是 ECMAScript 标准的函数，而是 Web 标准的。这意味着引擎不管这东西，需要各运行时自行实现。不同运行时实现方式不一样，Node.js 使用**薅羊毛手法**来实现。抛开运行时问实现的方式是耍流氓。

具体的实现方式在上文已经给出解析了。这种方式在大部分情况下都是很欢乐的，性能也不赖。但是在极端情况下，还是会有无伤大雅的时序问题。虽然我没试过给

趣学 Node.js

留言

输入评论（Enter换行，Ctrl + Enter发送）

发表评论

全部评论（1）

欢乐的马儿有草吃 JY.3 2天前

很详尽，慢慢学习

点赞 回复

趣学 Node.js