无栈协程

摘要

协程作为 21 世纪的任务调度模式，已经在大量的编程语言和项目中得到了应用。在我们公司内部，大量服务端同学使用 go 语言每天启动大量的有栈协程，而中台同学和客户端同学使用 Rust 和 Kotlin 语言，启动了大量的无栈协程。作为一个端上程序员，了解协程不同类型的区别以及无栈协程的基础知识可以辅助我们写出可读性高、健壮性强的代码，能够辅助我们的业务高质量的不断迭代。

本文分为五个部分：

1. 在第一章中，我们会对有栈协程和无栈协程进行辨析；
2. 在第二章中我们详细介绍无栈协程；
    a. 首先分析无栈协程优缺点；
    b. 介绍无栈协程设计模式；
    c. 介绍无栈协程调度模式及优化细节；
3. 在第三章中，我们会结合 RustSDK 的使用简单的介绍一些 tokio 的使用；
4. 最后我们会提供 Rust 以及 Kotlin 中无栈协程的最佳实现。

栈——有栈协程同无栈协程实现的关键区别

> 这部分参考：https://mthli.xyz/stackful-stackless/

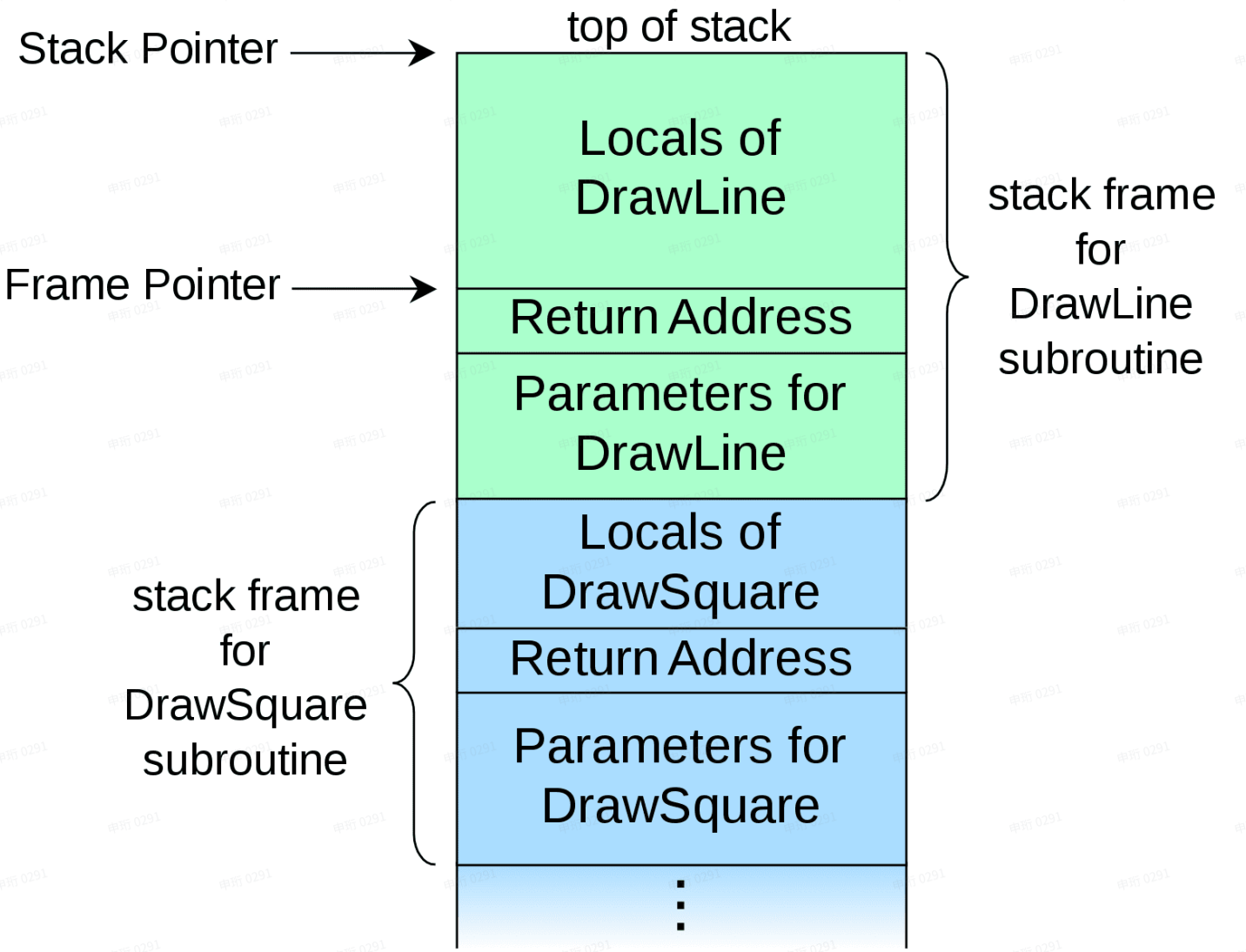
协程主要分为两类

1. 一类是有栈（stackful）协程，例如 **goroutine**；
2. 一类是无栈（stackless）协程，例如 **async/await**。

为了说明栈在协程中的作用，我们需要用一个具体的基于 X86 平台的例子，在 x86 平台中，调用栈的地址增长方向是从高位向低位增长的。简单起见，本文选取 32 位系统作为讨论对象。

函数调用栈

大家都比较熟悉栈了，不论是纯 C 还是 JVM 抽象，函数的执行都离不开一段在内存上连续的空间——栈，栈主要用于保存函**数执行时的局部变量**，并**先入先出**的进行更新。维基百科给了张图：



1. stack Pointer 即栈顶指针，总是指向调用栈的顶部地址，该地址由 esp 寄存器存储；
2. Frame Pointer 即基址指针，总是指向当前栈帧（当前正在运行的子函数）的底部地址，该地址由 ebp 寄存器存储。
3. Return Address 则在是 callee 返回后，caller 将继续执行的指令所在的地址，而指令地址是由 eip 寄存器负责读取的，且 eip 寄存器总是预先读取了**当前栈帧中**下一条将要执行的指令的地址。

为了更好的说明栈的调用形式，我们写一段 C：

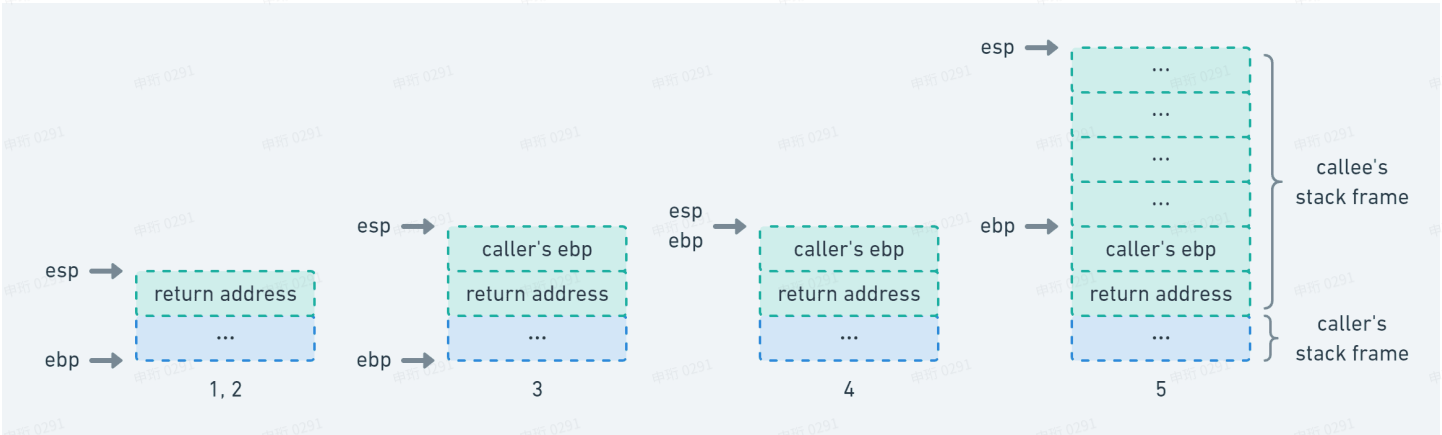
```
1  int callee() { // callee:
2                 //   pushl %ebp
3                 //   movl  %esp, %ebp
4                 //   subl  $16, %esp
5      int x = 0; //   movl  $0, -4(%ebp)
6      return x;  //   movl -4(%ebp), %eax
7                 //   leave
8                 //   ret
9  }
10
```

```
int caller() { // caller:
               //   pushl %ebp
               //   movl  %esp, %ebp
    callee();  //   call  callee
    return 0;  //   movl  $0, %eax
               //   popl  %ebp
               //   ret
}
```

当 caller 调用 callee 时，执行了以下步骤（注意注释中的执行顺序：

```
callee:// 3. 将 caller 的栈帧底部地址入栈保存
    pushl %ebp
    // 4. 将此时的调用栈顶部地址作为 callee 的栈帧底部地址
    movl  %esp, %ebp
    // 5. 将调用栈顶部扩展 16 bytes 作为 callee 的栈帧空间;
    //    在 x86 平台中，调用栈的地址增长方向是从高位向低位增长的，‘
    //    所以这里用的是 subl 指令而不是 addl 指令
    subl  $16, %esp
    ...
caller:...// "call callee" 等价于如下两条指令:
          // 1. 将 eip 存储的指令地址入栈保存;
          //    此时的指令地址即为 caller 的 return address,
          //    即 caller 的 "movl $0, %eax" 这条指令所在的地址
          // 2. 然后跳转到 callee
    pushl %eip
    jmp callee
    ...
```

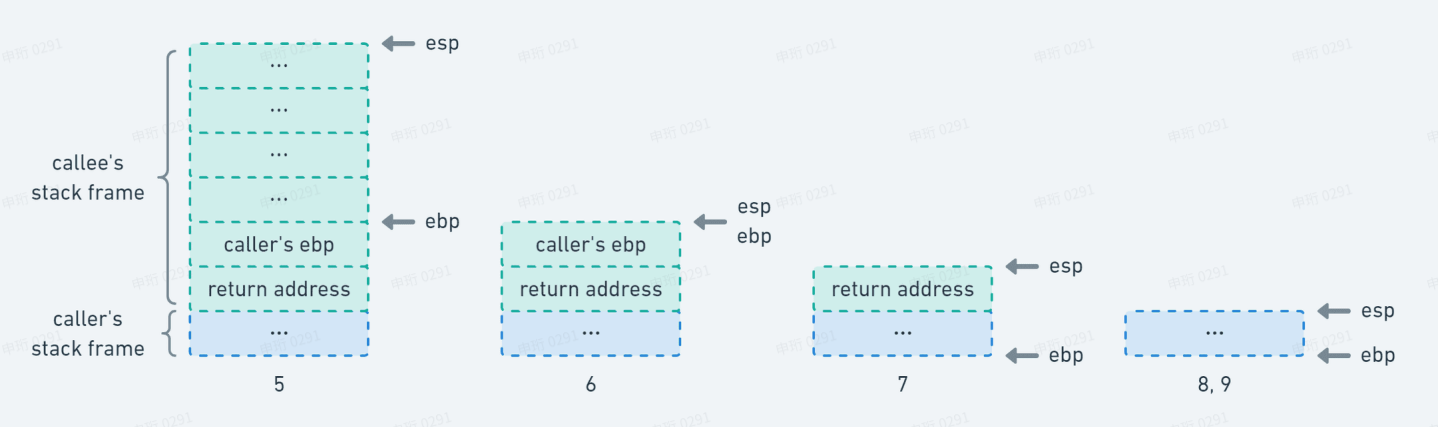
用图来看：



当 callee 返回 caller 时，则执行了以下步骤（注意注释中的执行顺序：

```
callee:...// "leave" 等价于如下两条指令:
          // 6. 将调用栈顶部与 callee 栈帧底部对齐，释放 callee 栈帧空间
          // 7. 将之前保存的 caller 的栈帧底部地址出栈并赋值给 ebp
    movl %ebp, %esp
    popl %ebp
    // "ret" 等价如下指令:
    // 8. 将之前保存的 caller 的 return address 出栈并赋值给 eip,
    //    即 caller 的 "movl $0, %eax" 这条指令所在的地址
    popl eip
caller:...// 9. 从 callee 返回了，继续执行后续指令
    movl $0, %eax
    ...
```

用图来看：



> 上面的内容省略了一些细节，大家感兴趣可以看这篇：
> https://redirect.cs.umbc.edu/~chang/cs313.s02/stack.shtml

有栈协程中的栈实现

从上面函数的调用可以知道，我们实现一个可以挂起的函数的关键点在于如何**保存、恢复和切换上下文**。

我们知道函数运行在调用栈上，可以自然地联想到

1. 保存上下文即是**保存从这个函数及其嵌套函数的（连续的）栈帧存储的值**，以及**此时寄存器存储的值**；
2. 恢复上下文即是**将这些值分别重新写入对应的栈帧和寄存器**；
3. 而切换上下文无非是**保存当前正在运行的函数的上下文，恢复下一个将要运行的函数的上下文**。

有栈协程便是这种朴素思想下的产物。我们可以简单的实现一个有栈协程：

1. 首先我们需要申请一段能**存储上下文的内存空间**。

> 在保存上下文时，我们可以选择把上下文都拷贝到这段内存；亦或者直接将这段内存作为协程运行时的栈帧空间，这样就能避免拷贝带来的性能损失了。注意，如果申请的内存空间小了，**协程在运行时**

> **会爆栈**；如果大了，则**浪费内存**；不过具体的分配策略我们就不做过多讨论了。

2. 接下来我们还需要保存寄存器的值。

> 这里便涉及到了函数调用栈中的一个知识点，根据约定，有的寄存器是由 caller 负责保存的，如 eax、ecx 和 edx；而有的寄存器是 callee 负责保存的，如 ebx、edi 和 esi。对于被调用的协程而言，只需要保存 callee 相关的寄存器的值，调用栈相关的 ebp 和 esp 的值，以及 eip 存储的 return address。

```
// *(ctx + CTX_SIZE - 1) 存储 return address
// *(ctx + CTX_SIZE - 2) 存储 ebx
// *(ctx + CTX_SIZE - 3) 存储 edi
// *(ctx + CTX_SIZE - 4) 存储 esi
// *(ctx + CTX_SIZE - 5) 存储 ebp
// *(ctx + CTX_SIZE - 6) 存储 esp
// 注意 x86 的栈增长方向是从高位向低位增长的，所以寻址是向下偏移的
char **init_ctx(char *func) {
    // 动态申请 CTX_SIZE 内存用于存储协程上下文
    size_t size = sizeof(char *) * CTX_SIZE;
    char **ctx = malloc(size);
    memset(ctx, 0, size);

    // 将 func 的地址作为其栈帧 return address 的初始值，
    // 当 func 第一次被调度时，将从其入口处开始执行
    *(ctx + CTX_SIZE - 1) = (char *) func;

    // https://github.com/mthli/blog/pull/12
    // 需要预留 6 个寄存器内容的存储空间，
    // 余下的内存空间均可以作为 func 的栈帧空间
    *(ctx + CTX_SIZE - 6) = (char *) (ctx + CTX_SIZE - 7);
    return ctx + CTX_SIZE;
}
```

接下来，为了保存和恢复寄存器的值，我们还需要撰写几段汇编代码。假设此时我们已经将存储上下文的内存地址赋值给了 eax，则保存的逻辑如下：

> Rust 有 asm! 宏直接联编汇编代码。

```
// 依次将各个寄存器的值存储；
// 注意 x86 的栈增长方向是从高位向低位增长的，所以寻址是向下偏移的
movl %ebx,  -8(%eax)
movl %edi, -12(%eax)
movl %esi, -16(%eax)
movl %ebp, -20(%eax)
movl %esp, -24(%eax)
```

```
8
9  //  %esp  存储的是当前调用栈的顶部所在的地址，
10 // (%esp) 是顶部地址所指向的内存区域存储的值，
11 // 将这个值存储为 return addressmovl (%esp), %ecx
12 movl %ecx, -4(%eax)
```

而与之相对应的恢复逻辑如下：

```
1  // 依次将存储的值写入各个寄存器;
2  // 注意 x86 的栈增长方向是从高位向低位增长的，所以寻址是向下偏移的
3  movl  -8(%eax), %ebx
4  movl -12(%eax), %edi
5  movl -16(%eax), %esi
6  movl -20(%eax), %ebp
7  movl -24(%eax), %esp
8
9  //  %esp  存储的是当前调用栈的顶部所在的地址，
10 // (%esp) 是顶部地址所指向的内存区域存储的值，
11 // 将存储的 return address 写入到该内存区域
12 movl -4(%eax), %ecx
13 movl %ecx, (%esp)
```

（上面代码其实就是顺着在内存上保存几个寄存器）

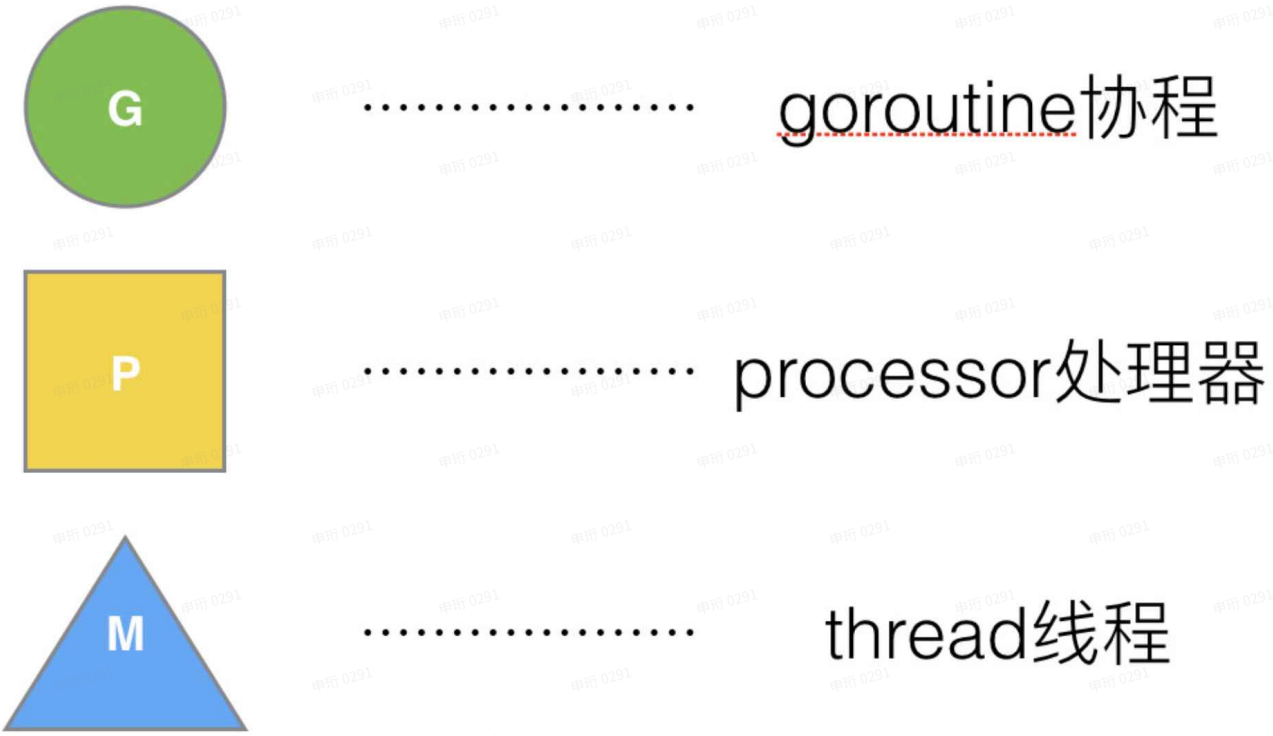
如果有同学写过操作系统的话，就会发现，上面的逻辑和线程一毛一样哈哈哈哈哈。然后整个流程串起来也和线程切换一毛一样：保存上下文 --> 切换到另一个线程 --> 读取上一次保存的上下文 --> 沿着程序计数器所指位置继续执行：

```
1  char **MAIN_CTX;
2  char **NEST_CTX;
3  char **FUNC_CTX_1;
4  char **FUNC_CTX_2;
5  void nest_yield() {
6      yield();
7  }
8
9  void nest() {
10     // 随机生成一个整数作为 tag
11     int tag = rand() % 100;
12     for (int i = 0; i < 3; i++) {
13     printf("nest, tag: %d, index: %d\n", tag, i);
14     nest_yield();
15     }
```

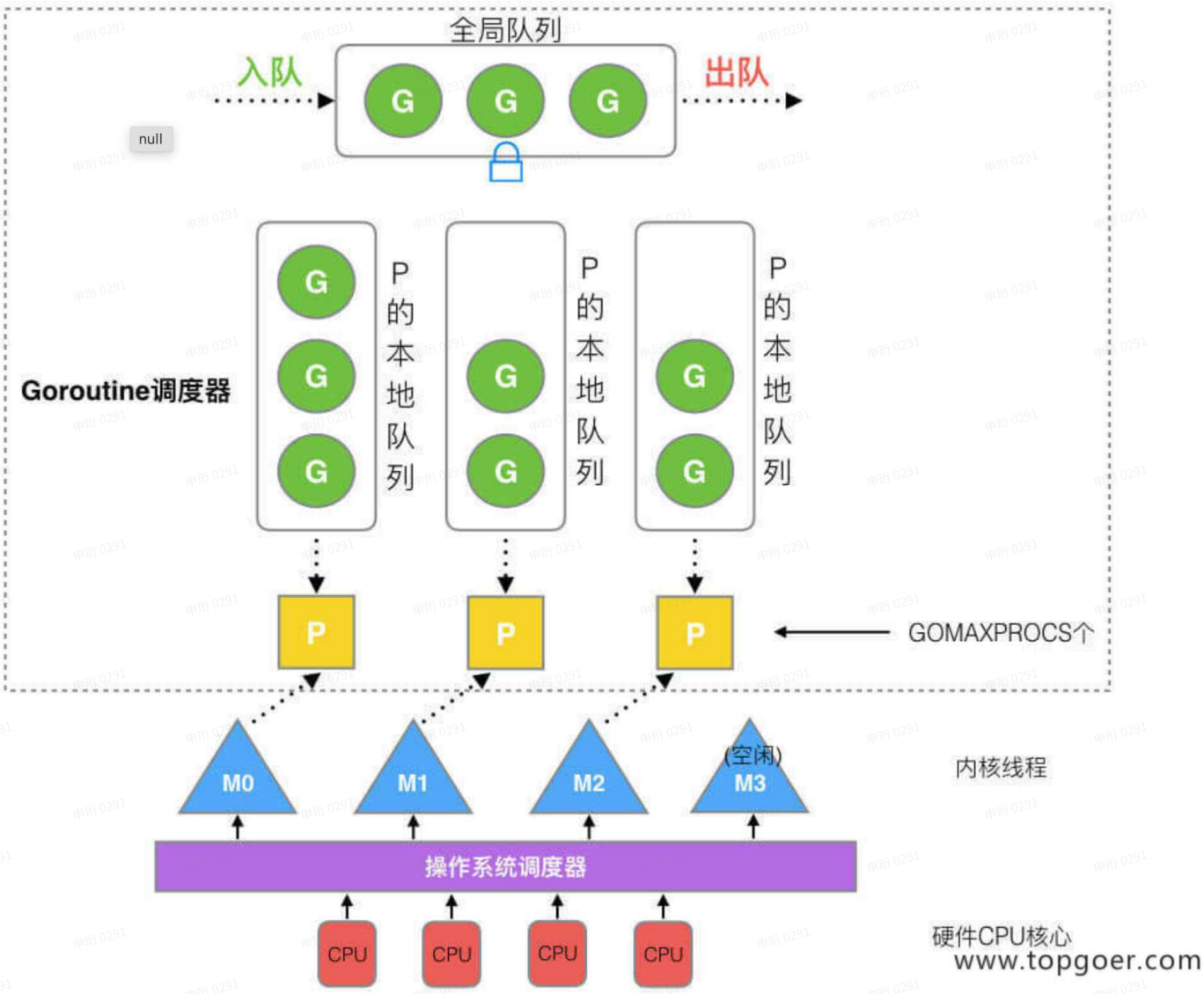
```
}

void func() {
    // 随机生成一个整数作为 tag
    int tag = rand() % 100;
    for (int i = 0; i < 3; i++) {
        printf("func, tag: %d, index: %d\n", tag, i);
        yield();
    }
}

int main() {
    MAIN_CTX = init_ctx((char *) main);
    // 证明 nest() 可以在其任意嵌套函数中被挂起
    NEST_CTX = init_ctx((char *) nest);
    // 证明同一个函数在不同的栈帧空间上运行
    FUNC_CTX_1 = init_ctx((char *) func);
    FUNC_CTX_2 = init_ctx((char *) func);
    int tag = rand() % 100;
    for (int i = 0; i < 3; i++) {
        printf("main, tag: %d, index: %d\n", tag, i);
        yield();
    }
    free(MAIN_CTX - CTX_SIZE);
    free(NEST_CTX - CTX_SIZE);
    free(FUNC_CTX_1 - CTX_SIZE);
    free(FUNC_CTX_2 - CTX_SIZE);
    return 0;
}
```

GMP 调度模型

GMP 实际上是三个词的组合：goroutine、processer、thread

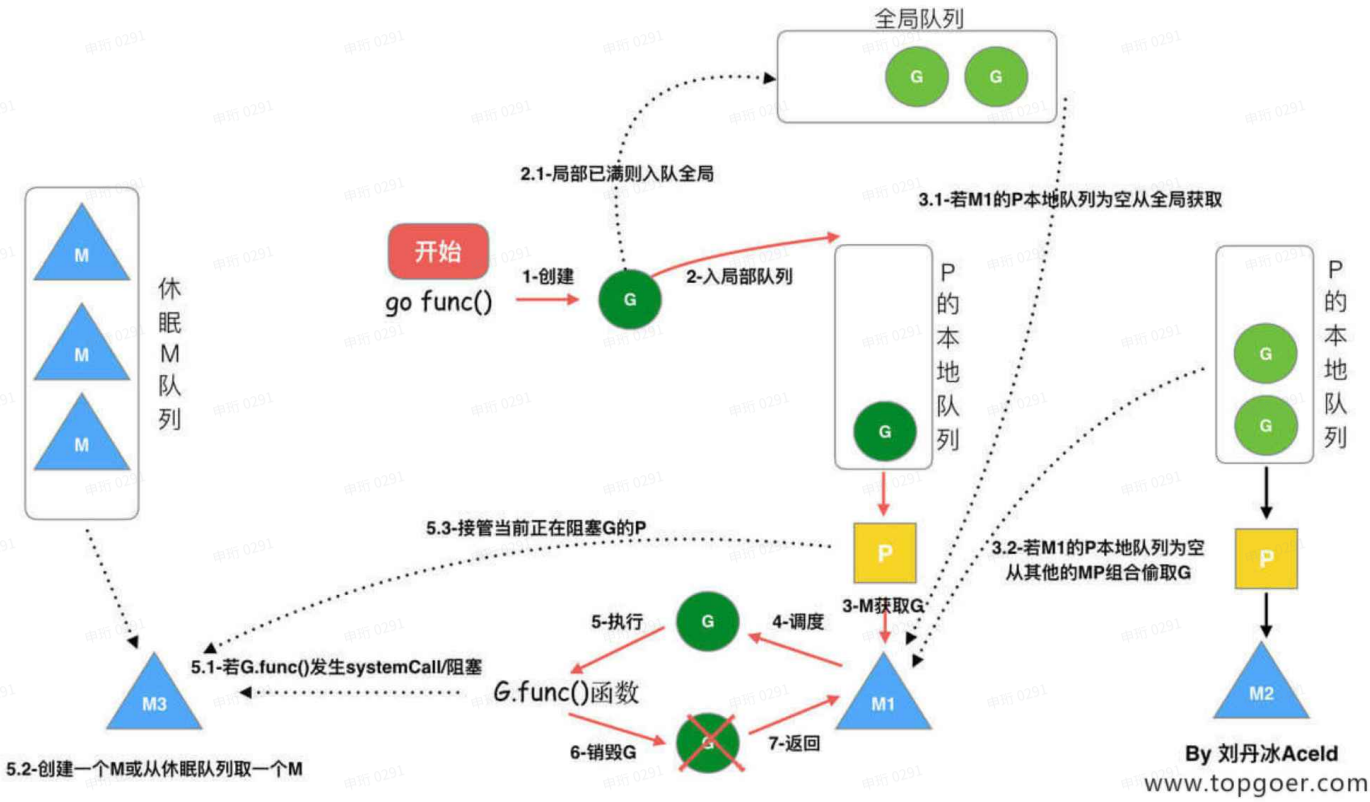


其中，gorotuine 是异步任务抽象、processer 是异步任务调度器、thread 是执行异步任务的主体。他们的调度模型大概符合下图：



1. 首先，存在一个**全局队列**，每一个启动的 goroutine 都会保存在其中
2. 然后**每一个调度器都会有着自己的本地队列**（woker-steal queue 这个数据结构），他们能够线程安全的“偷”其他队列的任务来执行。
3. 一个调度器负责调度一列 goroutine 到**对应的 OS 线程上**。

对于调度算法来说，**各个调度器之间的负载均衡策略**就是实现的关键。调度流程：



有栈协程优缺点分析

基于上面的实现以及 go 语法，我们可以大致总结出有栈协程所具有的几大优点：

1. 和线程实现模型一样，可以仿照线程的实现直接实现，**运行时极为简单易懂**。
2. 有栈协程栈的保存**屏蔽了复杂的内存细节**，程序员使用时心智负担较低。

但是有栈协程有着两个巨大的缺陷：

1. **内存分配问题**。如果协程栈用的多容易爆栈；而如果用得少则容易浪费内存。
2. **协程切换时需要读写上下文**，那线程的开销巨大的问题又引入进来了。

综上，实际上追求性能的语言，最终都还是抛弃了有栈协程，例如 C++、Rust、C#。

无栈协程

无栈协程优缺点分析

从上面有栈协程的实现可以看到无栈协程解决了两大痛点：

1. 无需栈内存分配，不用对 OS 线程层面做侵入式改造。
2. 没有协程切换上下文开销。

除了上面两个痛点外，无栈协程还有两个优点：

1. **切换极快，纳秒级别**，可以轻松起 500w 个协程（8 核 CPU）
2. **显著减少 cache miss**，这是因为基于时间/空间局部性，无栈协程内部的数据天然满足时间空间相近原则，实际实现是给包进了一个 struct 里面。

> cache miss 实际上是现代 APP 中真正的性能杀手，尤其存在多线程的程序中。像 Rust 的经典多线程库 crossbeam 中为了避免 cachemiss，是直接补齐 64 位数据的进行读取，也即读取空数据的性能开销也远远小于 cachemiss 时需要去内存刷新数据的开销。
>
> 因此评估软件性能时，cache miss 会是关键指标之一。

```rust
/// .... 还有其他处理器
#[cfg_attr(
    not(any(
        target_arch = "x86_64",
        target_arch = "aarch64",
        target_arch = "powerpc64",
        target_arch = "arm",
        target_arch = "mips",
        target_arch = "mips64",
        target_arch = "riscv32",
        target_arch = "riscv64",
        target_arch = "sparc",
        target_arch = "hexagon",
        target_arch = "m68k",
        target_arch = "s390x",
    )),
    repr(align(64))    // 对齐到 64 位
)]
pub struct CachePadded<T> {
    value: T,
}
```

但是无栈协程也有着问题：

1. 最大的问题在于，**会对程序员的编码方式带来较大的侵入性**。异步同步代码天生难以共存。
2. **无栈协程兼容性极差，基本所有异步库都得重写**，而有栈协程可以换个运行时直接使用。

> 现在 Kotlin 基本上都支持完毕了异步了，所以大家平时不用担心这个问题。

无栈协程心智模型

Reactor

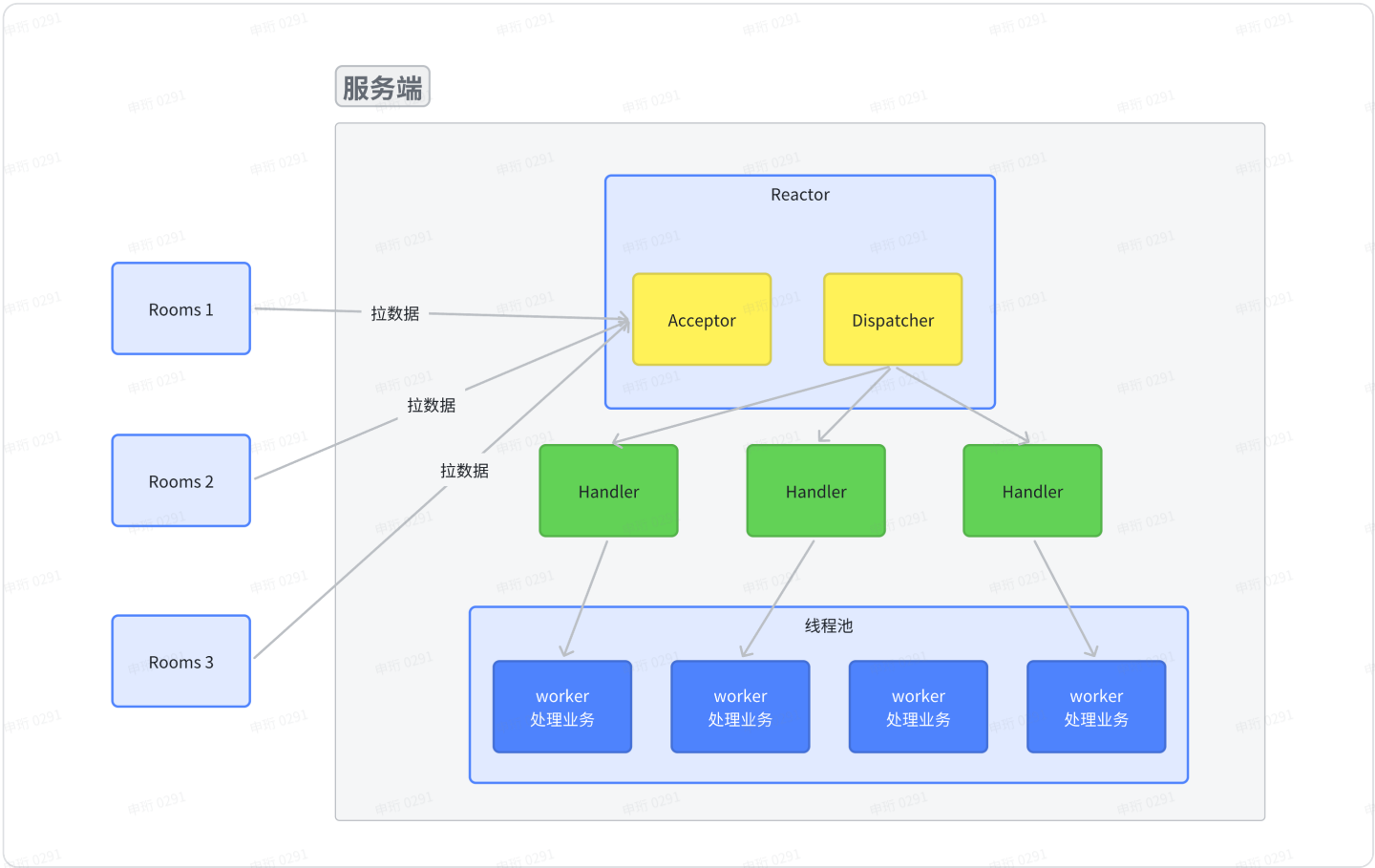
在讲无栈协程的心智模型之前，我们首先需要简单了解一些 Reactor 的实现。`Reactor` 模式一般翻译成**反应器模式**，也有人称为**分发者模式**。是**基于事件驱动的设计模式**，拥有**一个或多个并发输入源**，有一个服务处理器和多个请求处理器，服务处理器会同步的将输入的请求事件以**多路复用的**方式分发给相应的请求处理器。

Reactor 模式的典型实现大家都很熟悉——Java 的 NIO 库。在 Reactor 的概念中，有着三个关键角色：

1. `Reactor`：负责响应事件，将事件分发到绑定了对应事件的 `Handler`，如果是连接事件，则分发到 `Acceptor`；
2. `Handler`：事件处理器。负责执行对应事件对应的业务逻辑；
3. `Acceptor`：绑定了 `connect` 事件，当客户端发起 `connect` 请求时，`Reactor` 会将 `accept` 事件分发给 `Acceptor` 处理；

> 1、单reactor单线程，前台接待员、服务员是同一个人，全程为顾客服务。
>
> 2、单reactor多线程，1个前台接待，多个服务员，接待员只负责接待。
>
> 3、主从reactor多线程，多个前台接待，多个服务员。

上面的内容有点抽象，我们简单画个图



实际上就像我们 Rooms 在向服务端同时发起请求时，也会有一个 Reactor 做这种分发的操作。但是服务端的具体实现不是我们关心的重点，我们的重点在于，如果上面的图，他不是一个分布式的，而是

跑在一个计算机中，此时我们就得到了**协程概念中的 Reactor —— 专注于 IO 复用的组件**。

> 这是需要辨析的第一个概念，协程中的 Reactor 其实是指的 IO 复用。

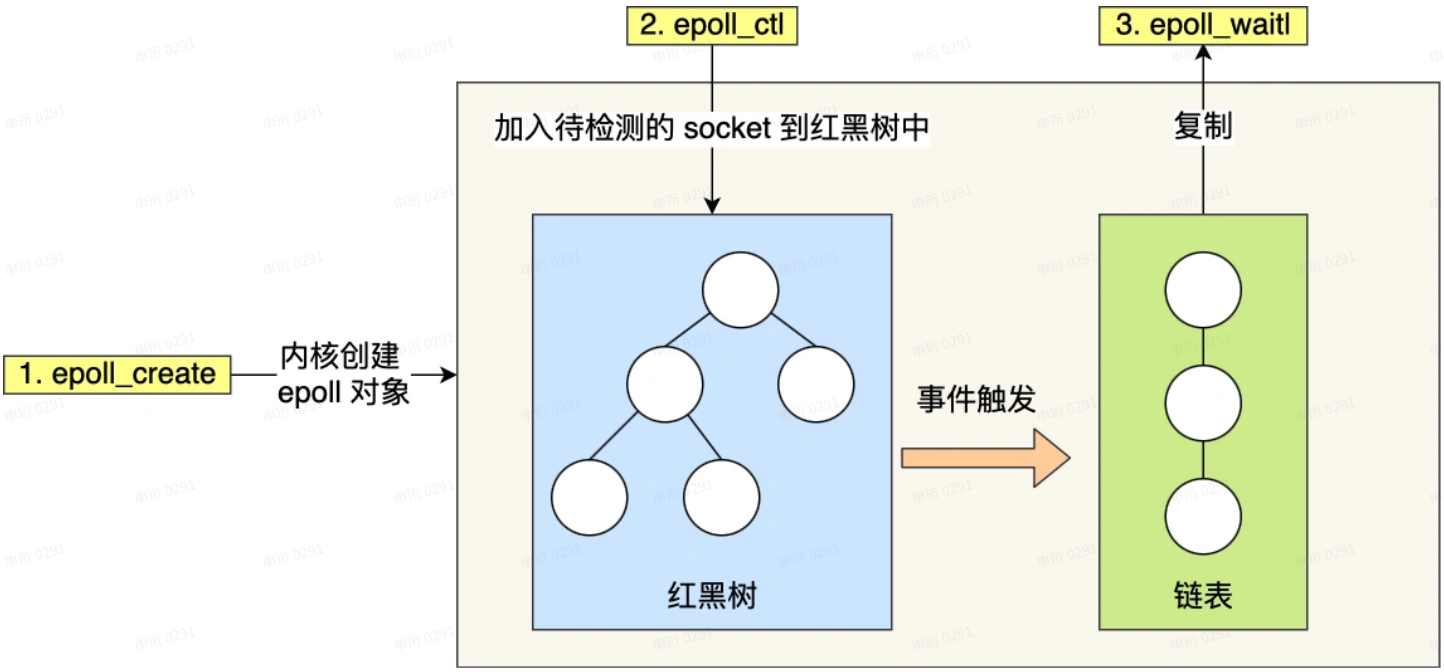
在上面的例子中，我们可以知道，Reactor 本质上是负责网络连接（文件连接）与任务分发的组件，在他的具体实现时会遇到两个问题：

1. **如果同时有多个 socket 连接，但是没有信息传输怎么办**？如果每个线程都跑一个死循环监听的话，会浪费大量资源。甚至可能需要起上万个线程。
2. **如果某个连接完毕的 socket 通知了**，应该怎样去把他**以尽可能低的时间复杂度拿出来**？

Linux 在 OS 层面实现了一个叫做 Epoll 的 Reactor 实现，这个实现实际上也是最常见的异步实现后端。Epoll 分为两部分：

1. 存储需要关心的 socket（或者文件连接）的数据结构——红黑树。
2. 用于通知到外部的事件链表。

他们之间的交互如下图：



> epoll 支持两种事件触发模式，分别是边缘触发（*edge-triggered，ET*）和水平触发（*level-triggered，LT*）。
>
> - 使用边缘触发模式时，当被监控的 Socket 描述符上有可读事件发生时，服务器端只会从 epoll_wait 中苏醒一次，即使进程没有调用 read 函数从内核读取数据，也依然只苏醒一次，因此我们程序要保证一次性将内核缓冲区的数据读取完；
> - 使用水平触发模式时，当被监控的 Socket 上有可读事件发生时，服务器端不断地从 epoll_wait 中苏醒，直到内核缓冲区数据被 read 函数读完才结束，目的是告诉我们有数据需要读取；
>
> 举个例子，你的快递被放到了一个快递箱里，如果**快递箱只会通过短信通知你一次**，即使你一直没有去取，它也不会再发送第二条短信提醒你，这个方式就是边缘触发；

> 如果快递箱**发现你的快递没有被取出，它就会不停地发短信通知你**，直到你取出了快递，它才消停，这个就是水平触发的方式。
>
> 这就是两者的区别，水平触发的意思是只要满足事件的条件，比如内核中有数据需要读，就一直不断地把这个事件传递给用户；而边缘触发的意思是只有第一次满足条件的时候才触发，之后就不会再传递同样的事件了。

无栈协程三大组件

在了解了 Reactor 之后，就可以继续聊无栈协程的模型了。在无栈协程中，整个运行时由三部分组成：

1. Executor（业务的执行者）
2. Reactor（这个就是上面说的 Reactor）
3. Future（任务抽象）

这三大组件之间的通信方式由 Waker 完成。

Future

首先看一下 Future 的类型：

```rust
pub trait Future {
    /// The type of value produced on completion.
    #[stable(feature = "futures_api", since = "1.36.0")]
    #[rustc_diagnostic_item = "FutureOutput"]
    type Output;

    #[lang = "poll"]
    #[stable(feature = "futures_api", since = "1.36.0")]
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

对于一个 Future 抽象来说，最为重要的就是理解到他**是一个状态机**，他本质上存在三个状态：

1. running，这个状态是在**使用 .await 激活时**，其执行态。
2. pending，这个状态是 `Output` 中的状态，其作为一个返回值可以被 Runtime 感知，此时 Future 释放了 CPU 资源，在等待 IO。
3. `Ready(val)`，这个状态也是 `Output` 中的状态，其作为一个返回值可以被 Runtime 感知，此时 Future 完成。

Future 本身极为好懂，但是 `Pin` 的历史包袱就比较大了。`Pin` 这个类型几乎是单纯为了异步而设计的，根源在于解决自引用导致的内存安全问题。详情见附录。

Executor

Executor 说起来复杂，实际上就是一个把各个 task 调度到对应线程上的执行死循环的函数，我们拿一个 tokio 的 `block_on` 方法康康：

```rust
pub(crate) fn block_on<F: Future>(&mut self, f: F) -> Result<F::Output, AccessError> {
    use std::task::Context;
    use std::task::Poll::Ready;

    let waker = self.waker()?;
    let mut cx = Context::from_waker(&waker);

    pin!(f);

    loop {
        if let Ready(v) = crate::runtime::coop::budget(|| f.as_mut().poll(&mut cx)) {
            return Ok(v);
        }

        self.park();
    }
}
```

但是 Kotlin 中的实现就比较复杂了，我们先聊完 Rust 的再说回 Kotlin。

Waker

Waker 是 Rust 异步心智模型中的重要一环，其是被定义在标准库中的，而不是由运行时实现。他的抽象也是比较简单的：

```rust
#[derive(PartialEq, Debug)]
#[stable(feature = "futures_api", since = "1.36.0")]
pub struct RawWaker {
    /// 一个 void 指针，它可以指向 Execoutor 需要的任何一个数据。
    /// 这个数据可以被虚表中的任何一个函数指针访问到
    data: *const (),
    /// Waker 行为定义，使用虚表来完成
    vtable: &'static RawWakerVTable,
}

pub struct RawWakerVTable {
    /// 用于 clone 一份 Waker
    clone: unsafe fn(*const ()) -> RawWaker,

```

```
    /// 用于唤醒任务
    wake: unsafe fn(*const ()),

    /// 也是唤醒任务，不同的是其用引用唤醒
    wake_by_ref: unsafe fn(*const ()),

    /// 回收数据
    drop: unsafe fn(*const ()),
}
```

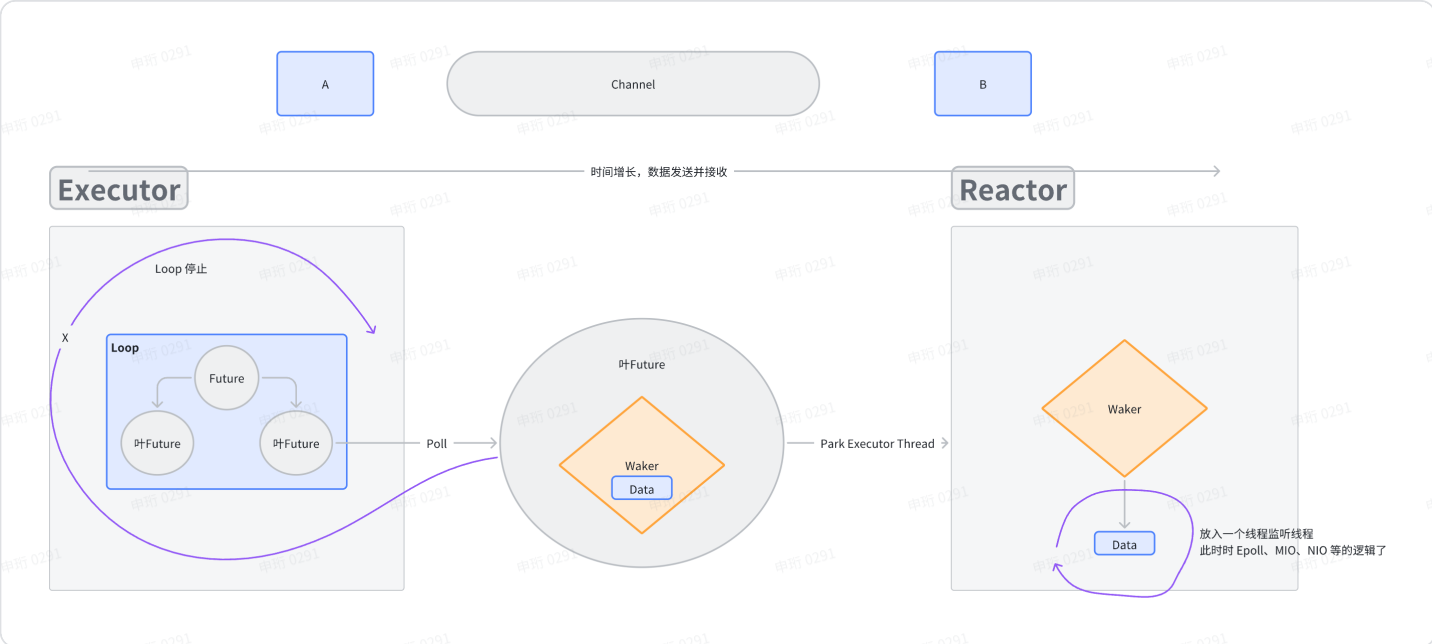
Waker 抽象大家可能看得有点懵，实际上这个抽象是为了解耦 Reactor 和 Executor 而被提出的，Kotlin 中没有这玩意儿，所以写得会更复杂一些。Waker 是 Rust 协程心智模型的核心部分。我们可以这么理解：

Future 本质上是一个传递 Waker 的 channel，该 channel 起始于第一次 Poll 该 Future 的 Executor 处，在获取到 Waker 之后终止于某个叶子 Future，Waker 作为其中进行传输的句柄，**将会最终由叶子 Future 交付给 Reactor**。

组件间的交互

一图胜千言：

首先我们看 Executor 通过 Future 将 Waker 发送到 Reactor 的过程：

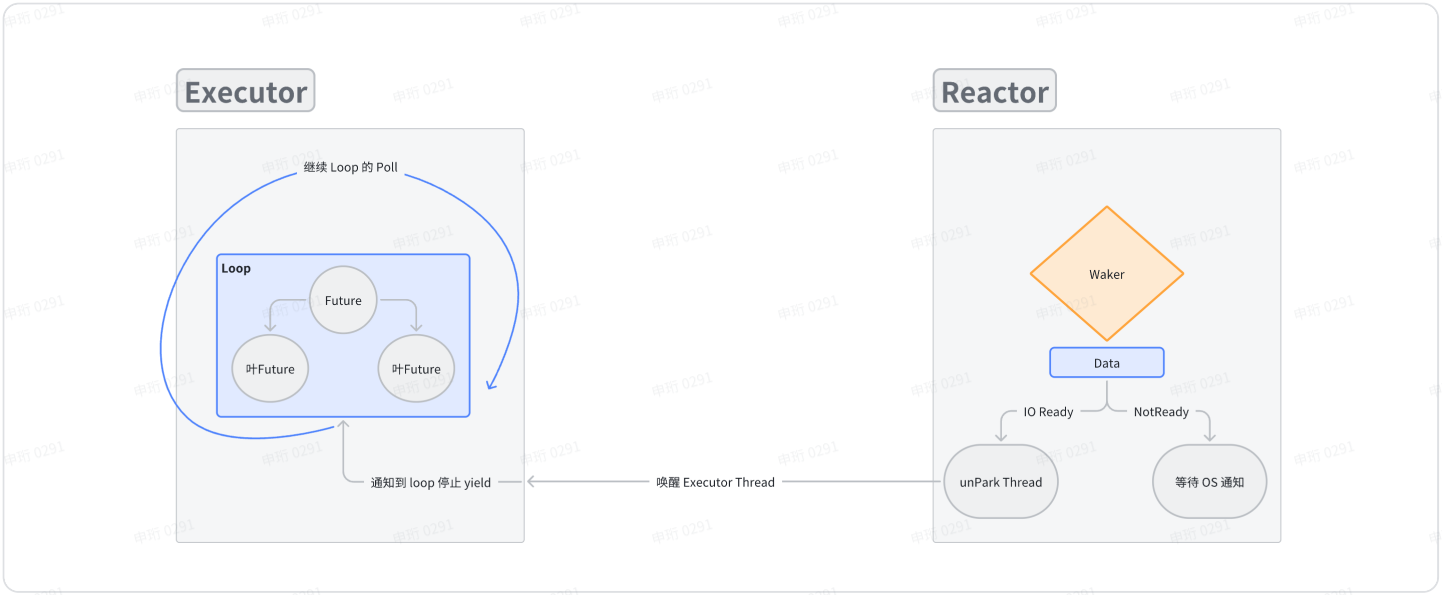


其中最为关键的就是：

1. Waker 抽象中，会把其中的数据带着给到 Reactor，**这个数据可以想象为一个线程句柄**。
2. Executor 只是去执行了 Future 的 `poll` 方法，而 Future 自己实现了将 Waker 放入 Reactor 的方法，**此时，Future 会持有 Reactor 的引用，而 Executor 不会持有（Executor 只管塞进来的 Future，但是 Future 绑定的 Reactor 不管，也即这样才可以做多 Reactor 多 IO 复用）**，完成解耦。

3. Future 把 Waker 成功传输到 Reactor 之后，**他会 yield Executor thread**，释放资源。

当 Reactor 接收到 Waker 之后，它会有一个同步机制，在 IO 资源 ready 之后通知到 Executor：



在返回时我们可以看到，实际上不是一个对等的传送过程，**Future 除了执行我们的计算逻辑外，就是在等待 IO 资源时传输 Waker 到 Reactor**。而 Reactor 只是通过 Waker 这个抽象来通知到 Executor 可以继续执行了，**因为 Future 中持有 Reactor 的引用**，此时可以直接通过 Reactor 拿到 socket 或者读取完毕的文件数据，执行下一步操作。

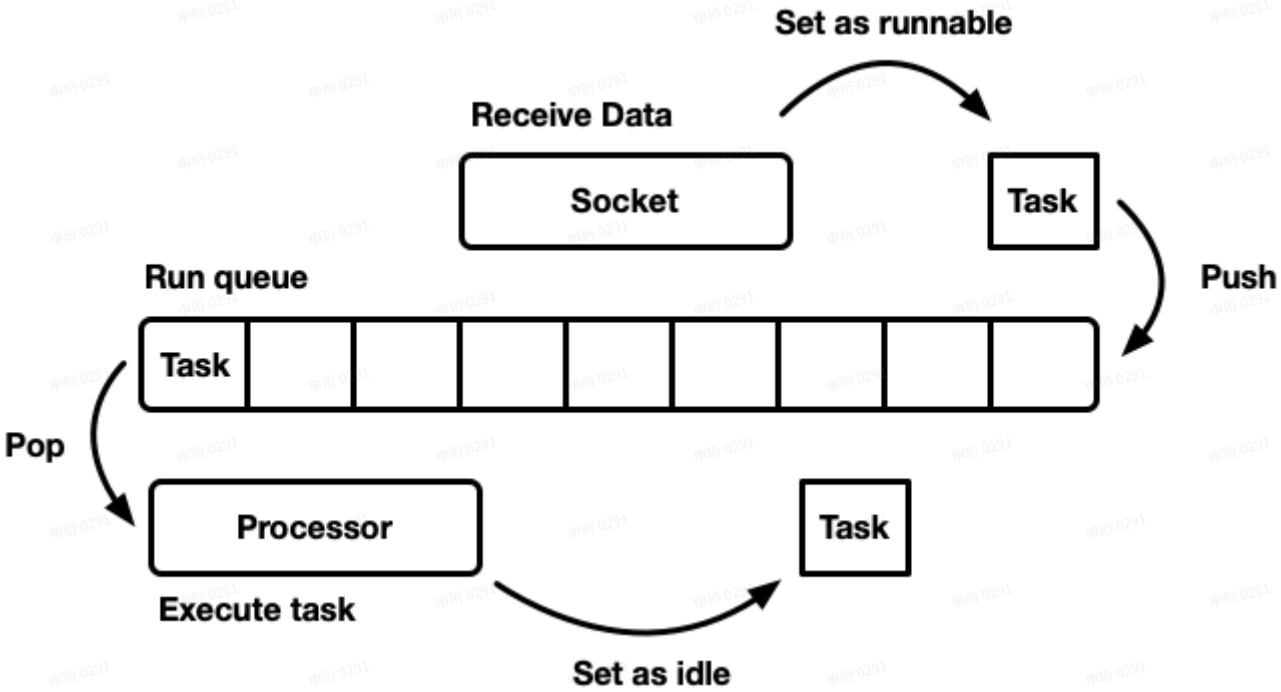
无栈协程调度模式

计算机世界最初的调度

Executor 实际上包含了两部分：调度器和执行器。执行器比较简单，就是普通的跑代码加上一些日志监测。而调度器较为负载，其核心是调度算法——用户空间的调度程序，即在操作系统线程之上运行的调度程序（这些线程又由内核级调度程序驱动）。Tokio调度程序是M：N线程模式，许多用户空间任务在少数操作系统线程上进行多路复用。

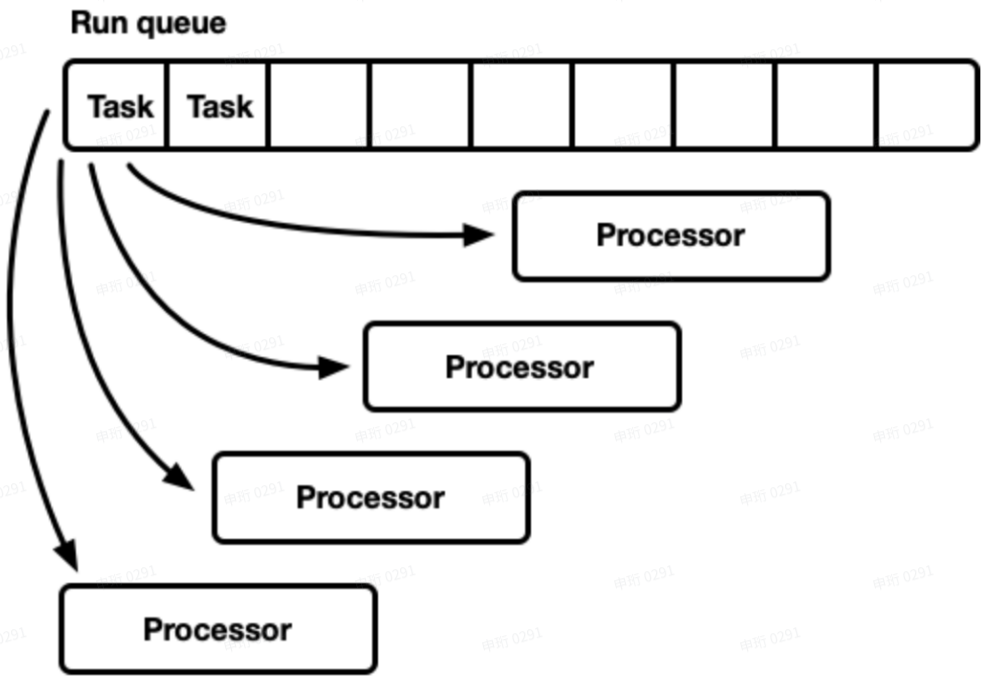
有许多不同的调度程序建模方式，每种都有优缺点。在最基本的层面上，调度程序可以被建模为运行队列和处理器来处理该队列，最为简单的一个调度器就是一个 FIFO queue：

```
while let Some(task) = self.queue.pop() {
    task.run();
}
```



多核时代的朴素调度策略

而一般单线程的调度总是会存在资源利用上的问题，这个问题更多是 CPU 层面的——“小核干活，大核围观”。多核多线程的调度模式，一般都是将 processer 放到对应的 OS 线程中去：



从这个层面上理解，可以看到无栈协程实际上的调度模式和有栈协程是类似的。

> 其实这也揭露了一个重要的点，分布式调度算法在人类历史上有着五六十年的历史了，现代技术不论怎么变化都很难跳出其原理范畴，因为这个理论的基础在于基础时空观。

在实现多处理器时，此时一般会用到侵入式的链表作为数据结构：

> 一个侵入式的数据结构是指**需要来自它打算存储的元素的帮助才能存储它们自己**的数据结构。

> 当将某个数据放入该数据结构时，**“某个数据”以某种方式意识到它在那个数据结构中**。将元素添加到数据结构会改变元素。
>
> 例如，一个非侵入式的二叉树，典型特点是其中**每个节点都有对左右子树、该节点存储的数据两部分引用**。而侵入式的二叉树中，**子树的引用被嵌入到被存储的数据内部**。
>
> 用在多消费者多生产者场景中，是因为**如果元素发生更改，列表就需要重新排序**，因此列表对象必须侵入元素的数据以获得这种合作——即元素必须感知到它所在的列表，并通知该列表发生了更改。

此时，Task 抽象中会包括指向 queue 中下一个 Task 的指针，而不是用常见的链表 Node 包装 Task 抽象。优点在于：

1. 可以避免在 push 和 pop 操作中进行堆分配。
2. 可以使用无锁 push 操作，但弹出需要互斥锁来协调消费者。

其缺点在于：

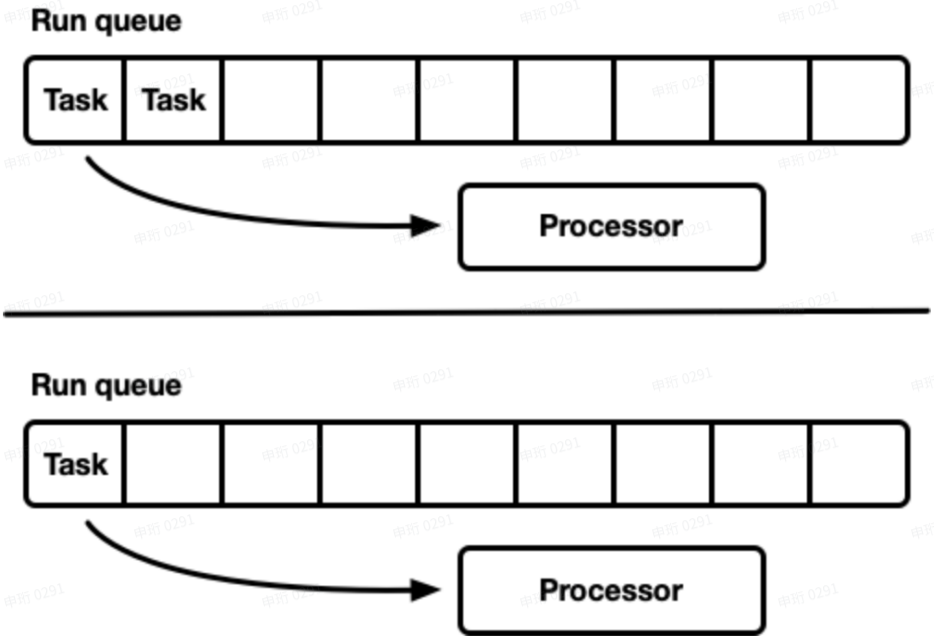
- 所有处理器在队列的 tail 竞争。**对于通用线程池来说，这通常不是一个问题**。因为**处理器执行任务的时间远远超过了从运行队列中弹出任务所花费的时间**。当任务执行时间较长时，队列竞争会减少。然而，Rust的异步任务在从运行队列中弹出时预计执行时间很短。在这种情况下，**在队列上竞争导致的开销会变得很大**。

运行队列的引入

上面的模型中遇到的显著矛盾是：queue 上任务调度时延远超过处理器执行时延。

解决这个问题时，天然的引入了另一个朴素思想——避免调度。具体的实现方式就是使用多个单线程的调度程序。**每个 core 都有自己的运行队列，任务被固定到特定的处理器**。这避免了同步问题。由于 Rust 的任务模型需要能够从任何线程排队一个任务，因此仍然需要一种**线程安全的方式将任务注入调度程序**。此时有两种方案：

1. 要么每个处理器的运行队列支持线程安全的推送操作（MPSC）
2. 要么每个处理器有两个运行队列：一个非同步队列和一个线程安全队列。



这是Seastar使用的策略。由于这种方案几乎完全避免了同步，这种策略可以非常快。然而，**这种策略并不是银弹**。除非**工作负载完全均匀，否则一些处理器会变得空闲，而其他处理器则处于负载状态，导致资源利用不足**。

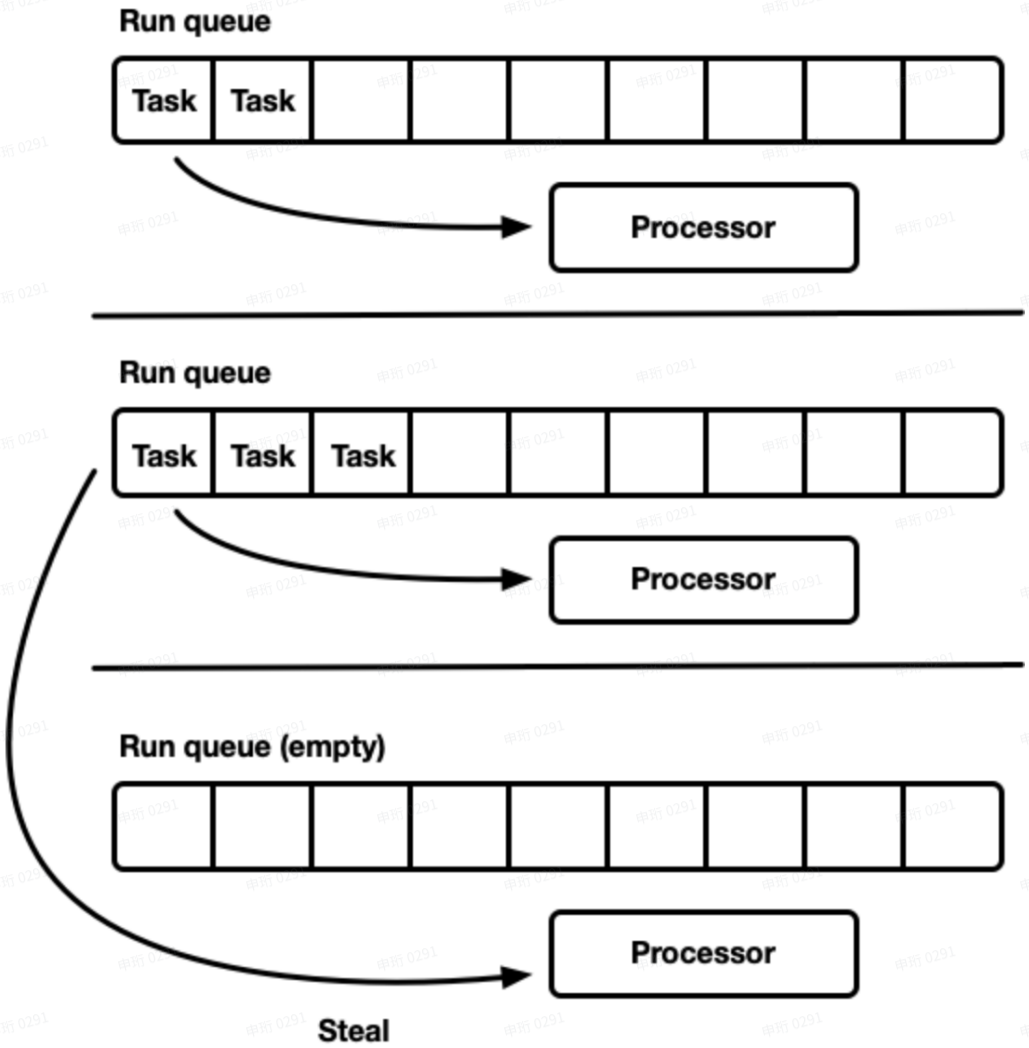
> 这是由于任务被固定到特定处理器上。当一个单处理器上一堆任务被批量调度时，即使其他处理器处于空闲状态，该单处理器也要负责处理高峰期的工作。

大多数“真实世界”的工作负载并不均匀。因此，通用目的的调度器倾向于避免使用这种模型。

最终的方案——work steal queue

Work-steal 调度器基于**分片调度器模型（OS 历史上那个经典的分片调度）**，并解决了上面提到的问题。他的特点在于：

1. 每个处理器维护自己的运行队列。变得可运行的任务会被推到当前处理器的运行队列，处理器会消耗它们的本地运行队列。
2. 当处理器变得空闲时，它会检查兄弟处理器的运行队列并尝试从中窃取任务。
3. 处理器只有在从兄弟运行队列中找不到任务时才会进入睡眠状态。



> 可以看到很像 GMP 算法了，其实本质上就是一个玩意哈哈哈哈。

在模型级别上，这是一种“两全其美”的方法：

- 处理器独立运行，避免了同步开销。
- 在负载在处理器之间分布不均匀的情况下，调度程序能够重新分配负载。

因为这个特性，工作窃取调度程序是Go、Erlang、Java等语言的选择。

> 更进一步的，NGINX 的负载均衡调度算法无非就是这个算法推到了多计算机层面，这揭示了计算机世界的另一个真理——计算机间的通信可以用计算机内部的通信进行抽象，其本质上很简单——数数需要一个一个数，时间始终向后流。

这种方法的缺点是算法本身的复杂度提升；**运行队列算法必须支持窃取操作，并且需要一些跨处理器的同步来确保运行顺利**。如果操作不正确，实现工作窃取模型的开销可能会超过收益。

> 一个典型的场景是：
>
> - 处理器A当前正在运行一个任务，而且运行队列是空的。
> - 处理器B处于空闲状态；它尝试窃取任务但失败了，所以进入睡眠状态。
> - 然后，处理器A执行的任务产生了20个新任务。

> 我们的目标是让处理器B醒来并窃取处理器A中一些新产生的任务。简单想一下，要实现的话，work-steal 调度程序需要在处理器观察到自己队列里有新的任务时，通知处于睡眠状态的兄弟处理器。这就引入了额外的同步，因此这些操作必须最小化时延。

基于 work-steal 模型下 tokio 的优化路径

最开始踩过的坑

1. Tokio 0.1 的调度程序**假定处理器线程在空闲一定时间后应该关闭**。调度程序最初是为了成为Rust futures的“通用”线程池执行程序而设计的。

> 最开始的模型是基于I/O的任务将在与I/O Reactor（epoll、kqueue、iocp等）共同定位的单个线程上执行。而其余的 CPU 线程绑定推迟到了线程池中完成。在这种情况下，活动线程的数量应该是灵活的，并且关闭空闲线程更有意义。
>
> 然而，**模型转变为在工作窃取调度程序上运行异步任务**，此时保持一小部分线程始终活动更有意义。

2. 它使用了crossbeam双端队列实现。**这个实现基于Chase-Lev队列(05 年的一篇论文)，不适合于调度独立的异步任务的使用场景**。原因在下一节说明。
3. 实现过于复杂。这在一定程度上是因为这是作者第一次实现调度程序。
4. 作者过于急切地在代码中使用原子操作，而大多数情况下，使用互斥锁就足够了。

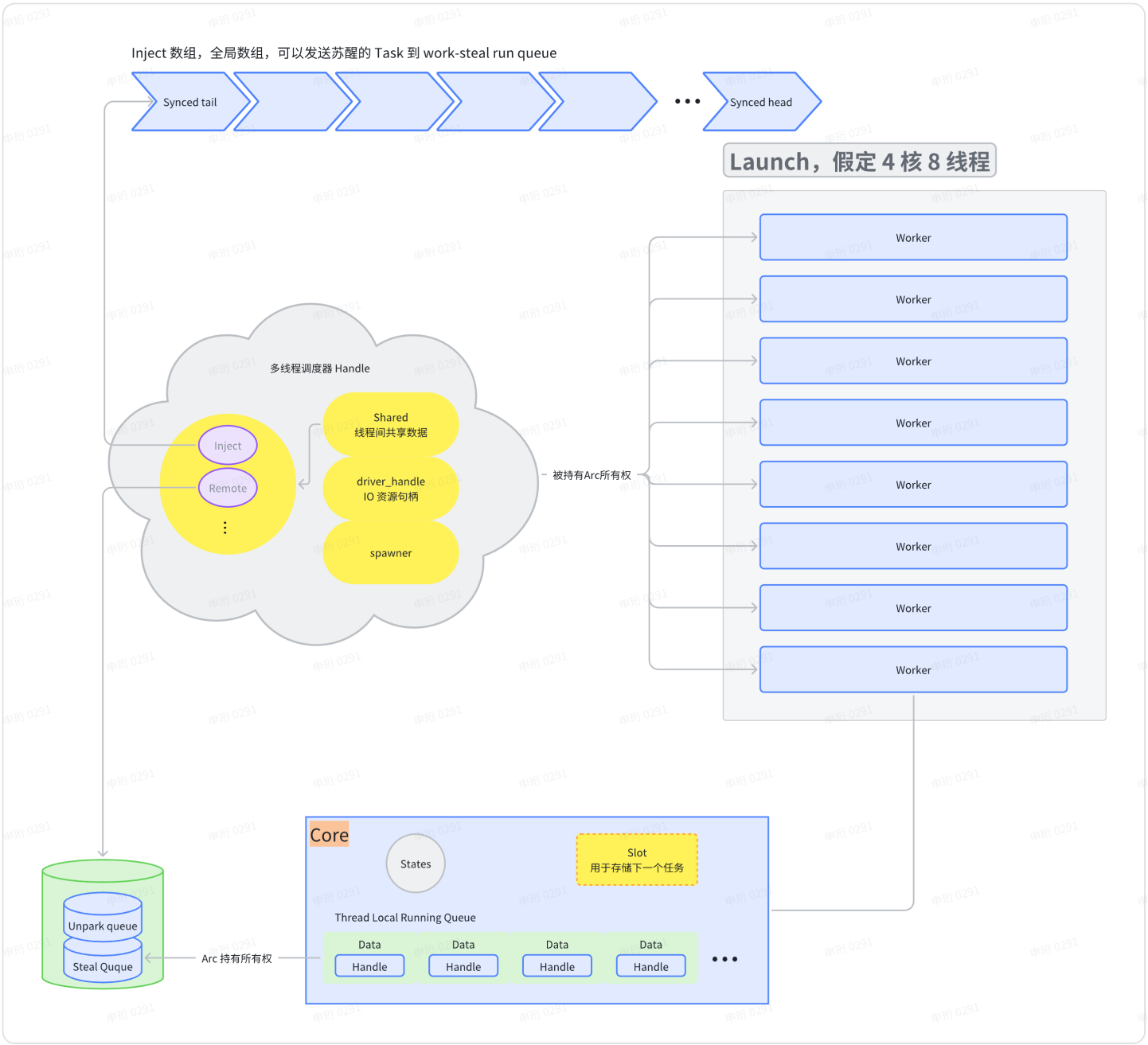
> 一个重要的教训是，在很多情况下，互斥锁是最佳选择，选择正确的锁可以避免很多麻烦的内存操作。

明晰结构

在讲解优化之前，需要澄清几个 Tokio 中的以及结构：

1. 处理器（代码中 core 翻译过来）对应一个 OS 线程，在 Builder 中指定数量。
2. Driver 抽象了 IO 接口完善时的通知。
3. Parker 取自多线程中的 `park` ，这里含义是 park runtime 而不是 park thread。

整个 Tokio 目前的结构体大致如图：



Queue 本身的优化

Tokio 最开始用的是 crossbeam 的 deque 实现，它是**单生产者、多消费者的双端队列**。任务被推送到尾，值从头弹出。在大部分时间，**push 值的线程会 pop 它，但其他线程偶尔会通过自己 pop 值来“窃取”**。双端队列由数组和一组跟踪头部和尾部的索引支持。当双端队列满时，向其推送会导致存储的增长。此时会分配一个新的更大的数组，并将值移入新的存储。它有着以下问题：

1. Deque 的自增长能力带来了复杂性和开销。
2. 推/弹出操作必须考虑 grow 带来的内存分配影响。
3. 此外，在 grow 时，释放原始数组会带来额外的同步问题。

> 例如，在一个具有 GC 能力的语言中，旧数组将超出作用域，并最终被垃圾回收。但是，Rust不带有垃圾回收功能。
>
> 这意味着需要手动释放数组，**但线程此时可能正在同时访问内存**。Crossbeam 的解决方案是使用代际 GC 策略（对，就是 Java 那个）。虽然不是一个非常昂贵的操作，但在 hot 路径中确实增加了额

> 外开销。使得每个操作必须在**进入和退出临界区时发出原子RMW（读取-修改-写入）操作**，以向其他线程发出信号，表示内存正在使用中，并避免释放。

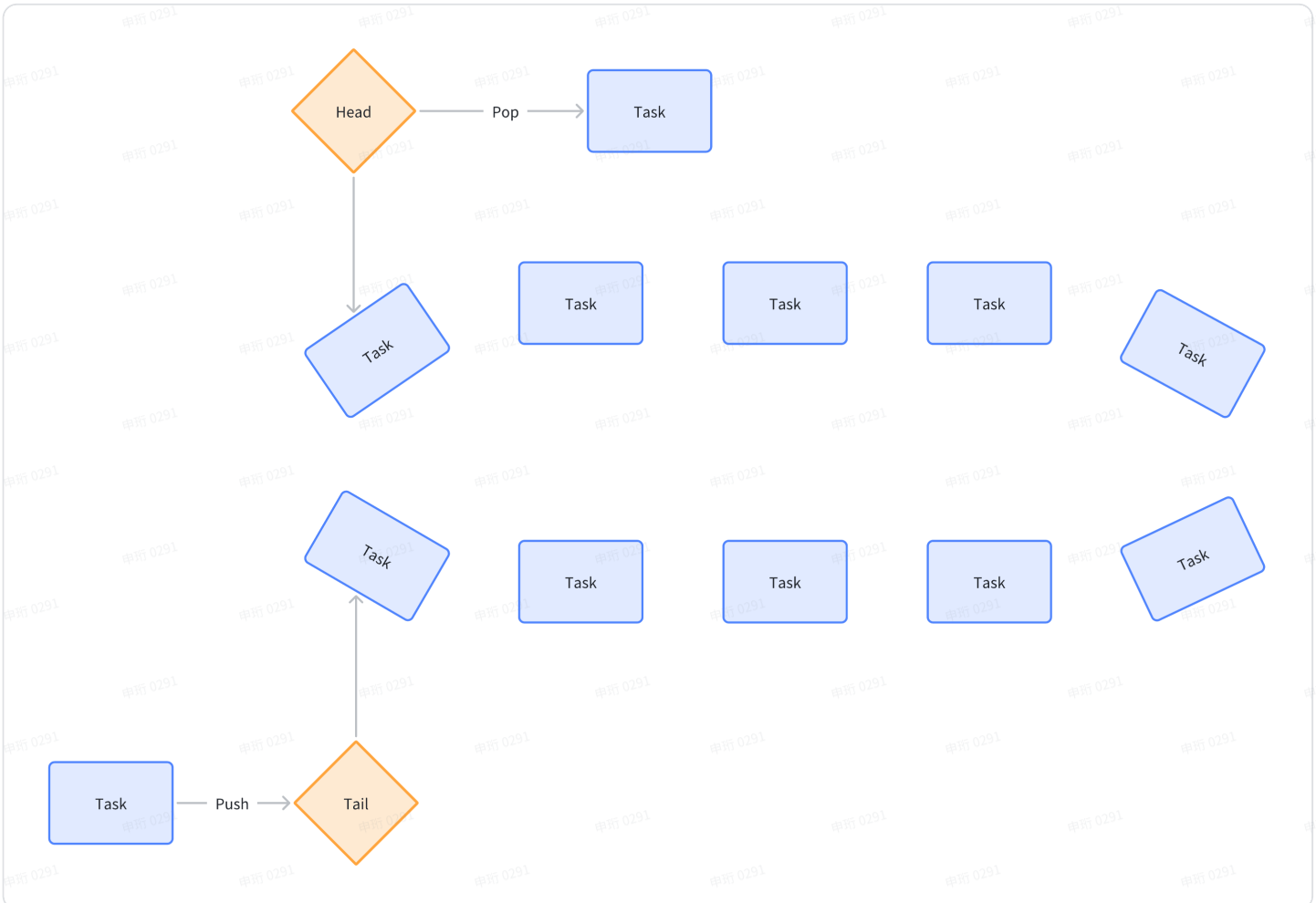
Tokio 新调度程序的策略是对**每个进程使用固定大小的队列**。当队列已满时，将任务推送到一个全局、多消费者、多生产者的队列中，而不是增加本地队列。处理器需要偶尔检查这个全局队列，但频率要比本地队列低得多。

> 大家此时能知道为啥 go 会用个全局队列了吧。
>
> 早期 tokio 用有界的 mpmc 队列替换了 crossbeam 队列。但是并没有带来太大改善，因为 push 和 pop 都执行了大量同步操作。而 work-steal 与之相悖的一个关键问题是，**在负载下，因为每个处理器只访问自己的队列，所以几乎没有争用队列**。

Tokio 作者仔细地阅读了 Go 源码，并发现他们使用了一个**固定大小的单生产者、多消费者队列**。这个队列令人印象深刻的地方在于它**运行所需的同步非常少**。最终用这个队列的算法来完成了 Tokio 的调度程序，但是也做了一些改动。

> 值得注意的是，Go实现使用**顺序一致性来进行原子操作**（也即最强的内存顺序一把嗦，C++ 中的 **Sequentially-consistent ordering**）。因此 Tokio 调度程序还减少了频繁的代码路径上的一些顺序一致性的复制操作。

队列的实现是一个**环形队列**，使用数组来存储值。原子整数用于跟踪头部和尾部位置：




```
1 struct Queue {
```

```
    /// 多线程写 head
    head: AtomicU32,

    /// 单线程写，多线程读
    tail: AtomicU32,

    /// Masks 头尾 index 来获取 buffer 中的 index，一个编码计算方式
    mask: usize,

    /// Task 的存储队列，一组堆数组
    buffer: Box<[MaybeUninit<Task>]>,
}
```

创建时，Local 队列被创建了一份，但是有两份所有权：

1. 一份是线程队列自身。
2. 另一份是 Steal 抽象持有。

```
/// Create a new local run-queue
pub(crate) fn local<T: 'static>() -> (Steal<T>, Local<T>) {
    let mut buffer = Vec::with_capacity(LOCAL_QUEUE_CAPACITY);

    for _ in 0..LOCAL_QUEUE_CAPACITY {
        buffer.push(UnsafeCell::new(MaybeUninit::uninit()));
    }

    let inner = Arc::new(Inner {
        head: AtomicUnsignedLong::new(0),
        tail: AtomicUnsignedShort::new(0),
        buffer: make_fixed_size(buffer.into_boxed_slice()),
    });

    let local = Local {
        inner: inner.clone(),
    };

    let remote = Steal(inner);

    (remote, local)
}
```

单线程执行 push 到线程独有运行队列的操作：

```rust
loop {
    let head = self.head.load(Acquire);

    // safety: 有且仅有这个线程会更新 tail，因此不做任何内存顺序保证。
    let tail = self.tail.unsync_load();

    if tail.wrapping_sub(head) < self.buffer.len() as u32 {
        // Map the position to a slot index.
        let idx = tail as usize & self.mask;

        // Don't drop the previous value in `buffer[idx]` because
        // it is uninitialized memory.
        self.buffer[idx].as_mut_ptr().write(task);

        // Make the task available
        self.tail.store(tail.wrapping_add(1), Release);

        return;
    }

    // The local buffer is full. Push a batch of work to the global
    // queue.
    match self.push_overflow(task, head, tail, global) {
        Ok(_) => return,
        // Lost the race, try again
        Err(v) => task = v,
    }
}
```

为了优化到极致的性能，在 push 中，**唯一的原子操作是使用 `Acquire` 顺序进行 head 的 load 和使用 `Release` 顺序进行 store。**

> 没有RMW (读-修改-写操作，`compare_and_swap`，`fetch_and`。。。），也没有顺序一致性（sequential consistency ording）。这一点很重要，**因为在x86芯片上，所有加载/存储操作已经是“原子”的**。因此，在CPU级别上，这个函数没有同步。使用原子操作会影响编译器，因为它会阻止某些优化，但是也仅此而已了。
>
> Acquire load 甚至可以使用 Relax 顺序安全地执行，但是实际上没有可测量到的收益。

当队列已满时，将调用 push_overflow 函数。**此函数将本地队列中的一半任务移动到全局队列中。**

> 全局队列是一个由**互斥锁保护的链表**。将要移动到全局队列的 Task 首先链接在一起，然后获取互斥锁，并通过更新全局队列的尾指针插入所有任务。**这保持了临界区维持在两个指针大小的水平，减小了同步风险。**

如果大家熟悉原子内存顺序，可能会注意到上面所示的 push 函数存在潜在的“问题”：**Acquire Load 是一个非常弱的原子操作。**

> **弱原子操作意味着，可能会返回过时的值（非 volatile）**，即并发的窃取操作可能已经增加了 self.head 的值，但执行 push 的线程在缓存中具有旧值，因此并没有注意到窃取操作。
>
> 这对算法的正确性来说并不是问题。在 push 代码中，我们**只关心本地运行队列是否已满**。鉴于当前线程是唯一可以 push 到运行队列的线程，读到旧值时，观测到的运行队列 size 会大于实际的 size（head 小了，数组变大）。进而提前进入 `push_overflow` 函数，而该函数包括更 strong 的原子操作，实际的检查由他来完成。
>
> 如果 `push_overflow` 确定队列实际上并没有满，它会返回错误并再次尝试推送。这也是为什么 `push_overflow` 仅将一半的运行队列移动到全局队列的另一个原因。通过移动一半队列，能够较少地发生“运行队列为空”的问题。

线程运行队列的 Pop 函数也是很简单：

```rust
loop {
    let head = self.head.load(Acquire);

    // safety: this is the **only** thread that updates this cell.
    let tail = self.tail.unsync_load();

    if head == tail {
        // queue is empty
        return None;
    }

    // Map the head position to a slot index.
    let idx = head as usize & self.mask;

    let task = self.buffer[idx].as_ptr().read();

    // Attempt to claim the task read above.
    let actual = self
        .head
        .compare_and_swap(head, head.wrapping_add(1), Release);

    if actual == head {
        return Some(task.assume_init());
    }
}
```

在这个函数中，仅有一个单一原子 load 和一个带 Release 的 `compare_and_swap`。主要开销来自 `compare_and_swap`。

steal 函数类似于pop函数，但从 self.tail 加载必须是原子的。此外，类似于push_overflow，steal 操作将**尝试获取队列的一半，而不是单个任务**。

最后是消耗全局队列。这个队列用于处理处理器本地队列的溢出，以及**从非处理器线程向调度程序提交任务**。如果处理器负载较重，即本地运行队列有任务。处理器会尝试在从本地队列执行约 60 个任务后从全局队列弹出。它还在“搜索”状态下检查全局队列。

消息传输机制（channel）优化

通常使用小 Task 建模来恒量 Tokio 编写的应用程序的性能。 这些 Task 往往使用消息传递方式（基本就是 channel 了）相互通信。 这种模式类似于其他语言，如Go和Erlang。

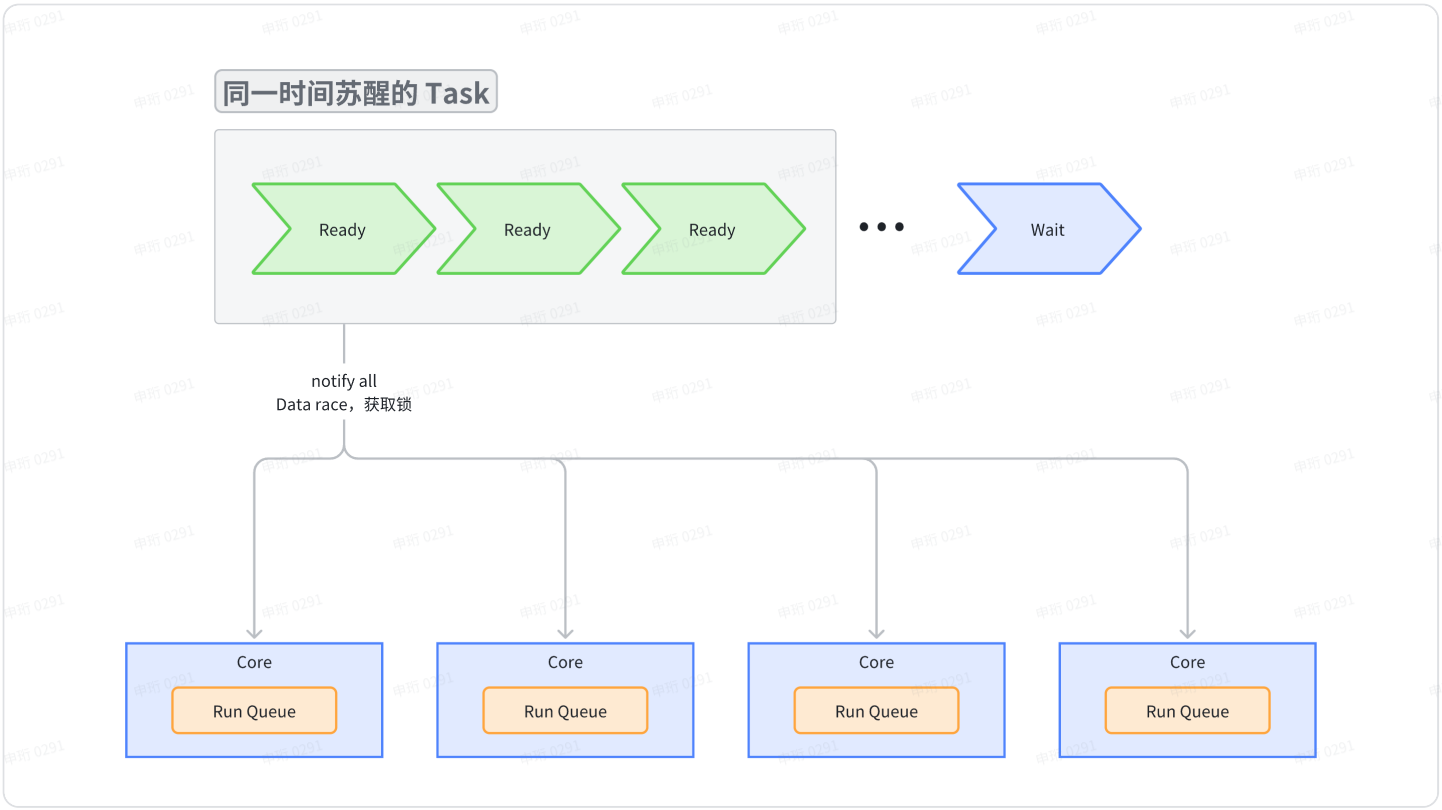
问题描述：给定任务A和任务B。 任务A当前正在执行并通过 channel 向任务B发送消息。 **channel 是任务B当前被阻塞的资源**，发送消息的操作将导致**任务B转换为可运行状态**，并被**推送到当前处理器的运行队列中**。 然后处理器将从运行队列中弹出下一个任务，执行它，并重复执行，直到达到任务B。 问题在于**消息发送和任务B执行之间可能存在显着的延迟**。 此外，“热”数据，如 channel 中发送的 message ，在**发送时存储在CPU的高速缓存中**，但到任务B被调度的时候，***相关的缓存很可能已经被清除***。

为了解决这个问题，新的Tokio调度程序实现了一个优化（也存在于Go和Kotlin的调度程序中）。 当任务转换为可运行状态时，不是将其推送到运行队列的末尾，而是将其存储在一个特殊的“下一个任务” slot 中。 处理器在检查运行队列之前始终会检查此槽。 在将任务插入此 slot 时，如果 slot 中已经存储了任务，则旧任务将从 slot 中移除并 push 到运行队列的末尾。 在基于 channel 通信的情况下，这使得消息的 receiver 会被安排为下一个要运行的任务。

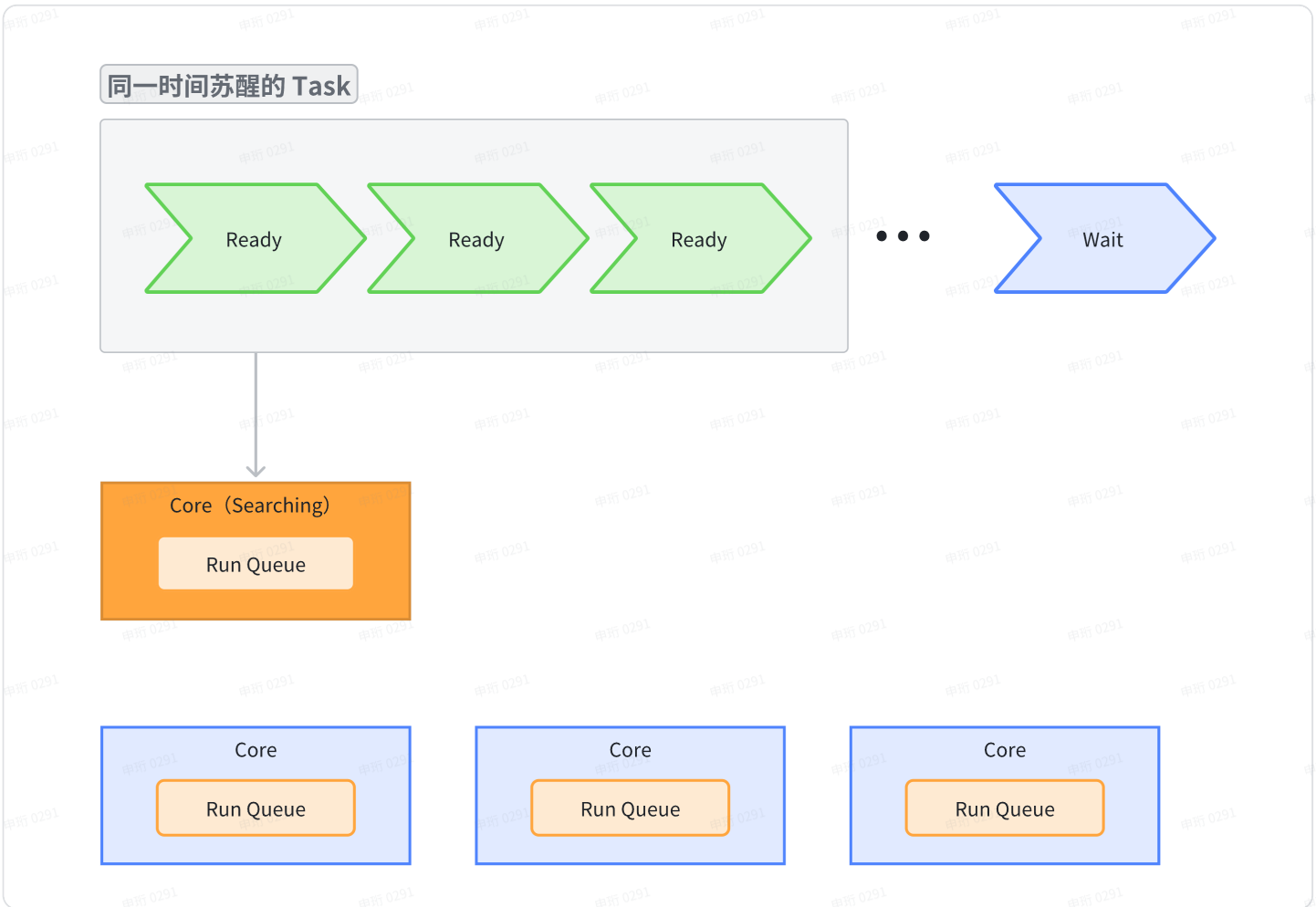
惊群问题（NGINX 八股经典问题）

在 work-steal 调度程序中，当处理器的运行队列为空时，处理器将尝试从兄弟处理器中窃取任务。为了做到这一点，会随机选择一个处理器作为起点，然后处理器对该兄弟执行窃取操作。如果没有找到任务，将尝试下一个兄弟，直到找到任务为止。

实际上，许多处理器在**大致相同的时间内完成处理它们的运行队列**是很普遍的行为（例如在**一批工作**到达时发生，典型的当轮询 epoll 以获取准备好的 socket 时）。此时，处理器会被唤醒，然后获取任务并运行直到完成。这会导致**所有处理器在同一时间尝试执行窃取**，这意味着许多线程尝试访问相同的队列。此时就会有 data race 了。随机选择起点有助于减少争用，但它仍然可能相当糟糕。



为了解决这个问题，新的调度程序**限制了并发执行窃取操作的处理器数量**。将处理器尝试执行窃取的状态称为“searching”状态（稍后会谈到这一点）。这一优化是通过拥有一个原子整数来实现的，在处理器开始搜索前进行递增，退出搜索状态时进行递减。最大搜索者数量是处理器总数的一半。



> 这个限制是粗略数字，但是可以接受。因为我们不需要对搜索者数量有硬性限制，而是只需要一个控制极值的节流器，为算法效率牺牲精度。一旦进入 searching 状态，处理器就会尝试从兄弟处理器窃取，同时检查全局队列。

减少线程间通信

Steal-work 调度程序的另一个关键部分是兄弟通知（sibling notification）。

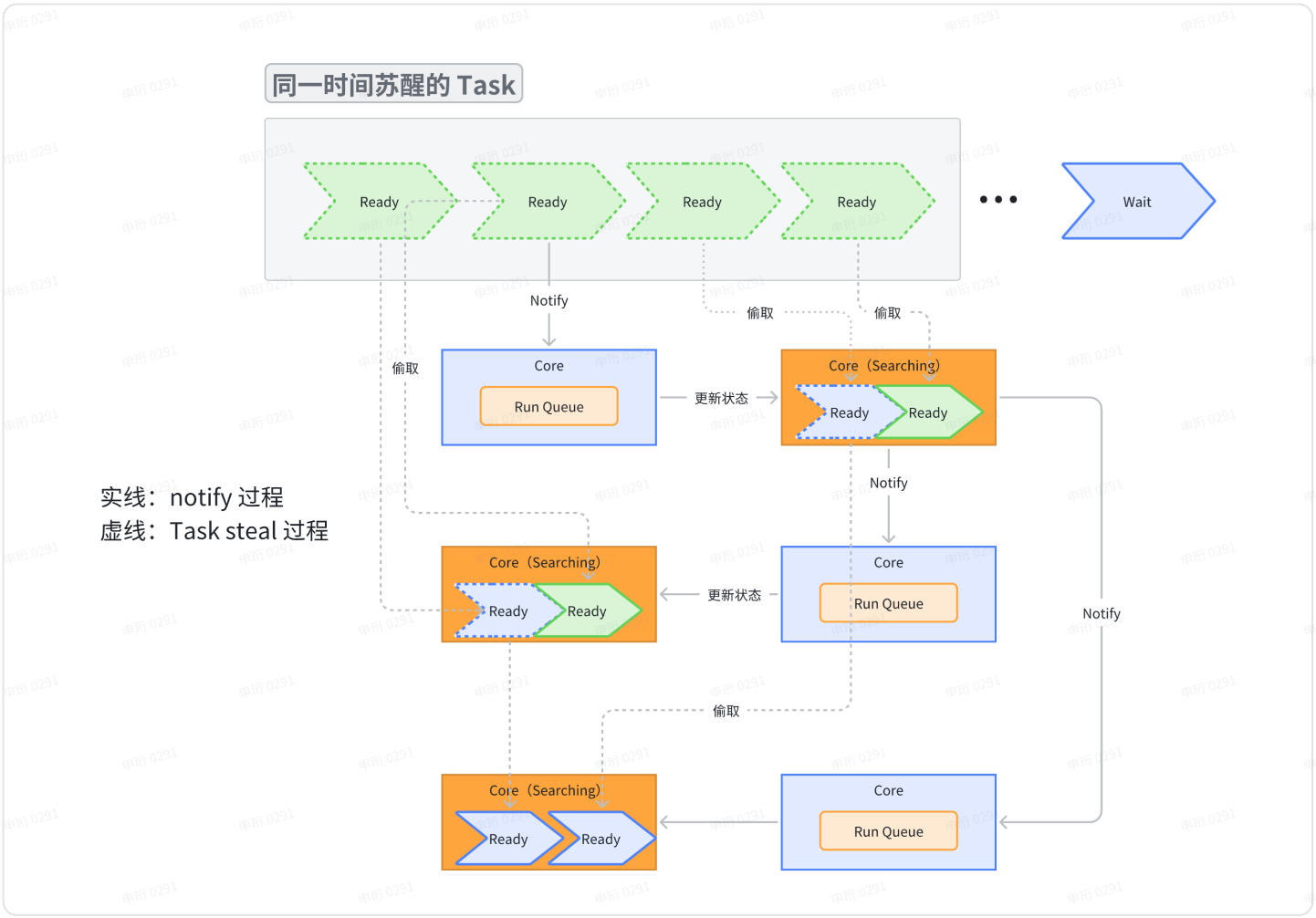
> 当处理器在观察到新任务时他会通知到兄弟处理器的地方。如果兄弟处理器正在 sleep，那么它会被唤醒然后执行任务窃取。
>
> 除此之外，通知操作还有另一个关键的任务，回忆一下，上面的队列算法使用了弱原子排序（获取/释放），此时由于原子内存排序的工作方式，实际上我们没有进行额外的同步机制，因此此时不存在一个机制来保证兄弟处理器会正确的读取到队列中的任务数量，完成窃取。这种情况下，**通知操作还负责为兄弟处理器建立必要的同步机制**，以便正确看到任务以便窃取它们。

这些要求使通知操作变得十分昂贵。**目标是尽可能少地执行通知操作，而不会导致 CPU 利用率不足，即处理器有任务而兄弟无法窃取**。但是过于激进的通知机制又会导致惊群问题。

原始的 Tokio 调度程序在通知方面采取了一种天真的方法。

> 每当将新任务推送到运行队列时，都会通知处理器。每当通知处理器并在唤醒时发现任务时，它会然后通知另一个处理器。这种逻辑很快导致所有处理器都被唤醒并搜索工作（引起争用）。如上节第一张图。
>
> 很多时候，大多数这些处理器因为无法找到 Task 而快速再次进入睡眠状态。

新的调度程序借鉴了 Go 调度程序。通知的时间点不变，还是新任务被推送到运行队列时。不同点在于，只有在当前没有处于“searching”状态的 worker 时才会进行通知。而当 worker 被通知时，其立即转换为“searching”状态。当处于搜索状态的处理器发现新任务时，它首先会退出“searching”状态，然后通知另一个处理器。



> 好处是限制了处理器醒来的速度。
>
> 如果一批任务一次调度（例如，对套接字准备情况进行 epoll 轮询时），第一个将导致通知一个处理器。该处理器现在处于搜索状态。批中其余调度的任务将不会通知处理器，因为至少有一个处于搜索状态。被通知的处理器**将窃取批中一半的任务，然后通知另一个处理器**。此时，第三个处理器将醒来，**从前两个处理器中找到任务并窃取其中一半。这导致处理器的平稳增加以及任务的快速负载平衡**。

减少堆分配

之前 Task 的数据结构是：

```rust
struct Task {
    /// All state needed to manage the task
    state: TaskState,

    /// The logic to run is represented as a future trait object.
    future: Box<dyn Future<Output = ()>>,
}
```

上面代码中，Task 结构体会被分配到一个 Box 中（等同于 Java 中的 new 分配）。因为，std::alloc 稳定。以及 Future 任务系统修改为了一种显式的 vtable 策略。因此终于消除了**每个任务效率低下的双**

重分配的问题。

现在的结构体如下：

```
struct Task<T> {
    header: Header,
    future: T,
    trailer: Trailer,
}
```

header 和 trailer 都是执行任务所需的状态，然而它们分为“热”数据（header）和“冷”数据（trailer），即经常访问的数据和很少使用的数据。**热数据放在结构的头部，并尽量保持较小。当CPU解引用任务指针时，它会一次加载一定量的缓存行大小的数据（在64到128字节之间），**这些数据尽可能相关使得 CPU 可以充分利用空间局部性原理。

Tokio 中的桥接方式

在我们谈论 Tokio 的方式前，先看一下 RustSDK 中的使用：

```
/// byteview-sdk/rust-sdk/lib-ffi/src/host_callback/worker.rs
/// 这是一个同步方法，前面没有 async 修饰
fn thread_func(&self) -> Result<()> {
    SDK_THREAD_REGISTER.call_once(lib_util::sdk_threads::register);

    let (tx, rx) = mpsc::unbounded();
    *self.tx.write_lock() = Some(tx);

    let runtime = runtime::Builder::new_current_thread()
        .enable_all()
        .build()?;

    #[cfg(target_os = "android")]
    let processor = {
        let env = JVM
            .get()
            .expect("jvm is not ready")
            .attach_current_thread_permanently()
            .expect("attach thread to jvm failed.");
        Processor::new(env)
    };

    #[cfg(not(target_os = "android"))]
    let processor = Processor::new();

```

```
    let work = rx.for_each(|task| {
        consume(task, &processor);
        futures::future::ready(())
    });

    info!("host_callback worker is up");
    self.handle.notify_thread_up();

    // 开启异步执行
    runtime.block_on(work);
    debug!("recever");

    #[cfg(not(target_arch = "wasm32"))]
    lib_util::sdk_threads::unregister();
    Ok(())
}
```

可以看到在我们推送来之后，定制化的处理推送就是抛了一个异步任务去做的，基于上一节的知识，这个异步任务是一个 top-level 任务，因此使用了一个执行器去 block 的执行。

在 Tokio 中，`block_on` 函数是 Executor 的主要入口，也是最重要和最灵活的异步同步桥接方案：

```
/// 典型宏异步 main
#[tokio::main]
async fn main() {
    println!("Hello world");
}

/// 拆宏后
fn main() {
    tokio::runtime::Builder::new_multi_thread()
        .enable_all()
        .build()
        .unwrap()
        /// 可以看到也是 block_on
        .block_on(async {
            println!("Hello world");
        })
}
```

在这个例子以及 RustSDK 的例子中，我们可以注意到三个重要的地方：

1. Runtime
2. Thread

3. 调度方式

接下来会沿着这两条线去讲述我们怎么在同步代码中跑起来一个异步 top-level task 的。

Runtime

Runtime 是 tokio 的核心，也是异步函数执行的必要条件。他包含了上面提到的心智模型中的三个主要部分，抽象为了以下运行时服务：

1. 一个I/O事件循环，称为 driver，驱动I/O资源并将I/O事件分派到依赖它们的任务。（Reactor 模型）
2. 一个 scheduler 来执行使用这些I/O资源的任务。（Executor）
3. 一个定时器，用于在一段时间后调度工作运行。（Executor 调度方案——定制化措施）

Tokio 将上面这些服务捆绑为单个类型 Runtime，允许它们一起启动、关闭和配置。Runtime 启动之后，会给到一个 Context 的概念，在这个 Context 中，spawn 出来的 task（注意对比线程的概念）才能够被 Executor 执行：

```rust
fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Create the runtime
    let rt  = Runtime::new()?;

    // Spawn the root task
    rt.block_on(async {
        let listener = TcpListener::bind("127.0.0.1:8080").await?;

        loop {
            let (mut socket, _) = listener.accept().await?;

            // leaf-task, 只要 socket 接收到数据就 spawn 一个出来进行处理
            tokio::spawn(async move {
                let mut buf = [0; 1024];

                // In a loop, read data from the socket and write the data back.
                loop {
                    let n = match socket.read(&mut buf).await {
                        // socket closed
                        Ok(n) if n == 0 => return,
                        Ok(n) => n,
                        Err(e) => {
                            println!("failed to read from socket; err = {:?}", e);
                            return;
                        }
                    };
```

```
27
28                      // Write the data back
29                      if let Err(e) = socket.write_all(&buf[0..n]).await {
30                          println!("failed to write to socket; err = {:?}", e);
31                          return;
32                      }
33                  }
34              });
35          }
36      })
37  }
```

每个 Context 是线程的 local 变量，我们可以直接看源码：

```
1  // 每个线程都有一个 Context
2  tokio_thread_local! {
3      static CONTEXT: Context = const {
4          Context {
5              #[cfg(feature = "rt")]
6              thread_id: Cell::new(None),
7
8              // Tracks the current runtime handle to use when spawning,
9              // accessing drivers, etc...
10             #[cfg(feature = "rt")]
11             current: current::HandleCell::new(),
12
13             // Tracks the current scheduler internal context
14             #[cfg(feature = "rt")]
15             scheduler: Scoped::new(),
16
17             #[cfg(feature = "rt")]
18             current_task_id: Cell::new(None),
19
20             // Tracks if the current thread is currently driving a runtime.
21             // Note, that if this is set to "entered", the current scheduler
22             // handle may not reference the runtime currently executing. This
23             // is because other runtime handles may be set to current from
24             // within a runtime.
25             #[cfg(feature = "rt")]
26             runtime: Cell::new(EnterRuntime::NotEntered),
27
28             #[cfg(any(feature = "rt", feature = "macros"))]
29             rng: Cell::new(None),
30
31             budget: Cell::new(coop::Budget::unconstrained()),
```

```
32
33              #[cfg(all(
34                  tokio_unstable,
35                  tokio_taskdump,
36                  feature = "rt",
37                  target_os = "linux",
38                  any(
39                      target_arch = "aarch64",
40                      target_arch = "x86",
41                      target_arch = "x86_64"
42                  )
43              ))]
44              trace: trace::Context::new(),
45          }
46      }
47  }
```

其实看到这个数据结构就很清晰了，关键在于是否能够直接拿到当前线程 Runtime 中的 Executor 去执行 task，而 tash 不管 Waker 怎么弄，入口都是 Executor。这里我们使用 enter 方法来简单的看下源码就可：

> `block_on()` 是进入runtime的主要方式。但还有另一种进入runtime的方式： `enter()` 。
>
> `block_on()` 进入runtime时，会阻塞当前线程， `enter()` 进入runtime时，不会阻塞当前线程，它会返回一个 `EnterGuard` 。EnterGuard没有其它作用，它仅仅只是声明从它开始的所有异步任务都将在runtime上下文中执行，直到删除该EnterGuard。
>
> 删除 EnterGuard 并不会删除 runtime，只是释放之前的 runtime 上下文声明。因此，删除 EnterGuard 之后，可以声明另一个 EnterGuard，这可以再次进入 runtime 的上下文环境。我们可以在 EnterGuard 的 drop 方法中找到对应的处理。

```
1  use tokio::{self, runtime::Runtime, time};
2  use chrono::Local;
3  use std::thread;
4
5  fn now() -> String {
6    Local::now().format("%F %T").to_string()
7  }
8
9  fn main() {
10     let rt = Runtime::new().unwrap();
11
12     // 进入runtime，但不阻塞当前线程
13     let guard1 = rt.enter();
14
```

```
    // 生成的异步任务将放入当前的runtime上下文中执行
    tokio::spawn(async {
      time::sleep(time::Duration::from_secs(5)).await;
      println!("task1 sleep over: {}", now());
    });

    // 释放runtime上下文，这并不会删除runtime
    drop(guard1);

    // 可以再次进入runtime
    let guard2 = rt.enter();
    tokio::spawn(async {
      time::sleep(time::Duration::from_secs(4)).await;
      println!("task2 sleep over: {}", now());
    });

    drop(guard2);

    // 阻塞当前线程，等待异步任务的完成
    thread::sleep(std::time::Duration::from_secs(10));
}
```

而 spawn 出来的一个 task 就是获取到了这个 Executor handler 并立即执行：

```
#[track_caller]
pub(super) fn spawn_inner<T>(future: T, name: Option<&str>) ->
JoinHandle<T::Output>
where
    T: Future + Send + 'static,
    T::Output: Send + 'static,
{
    use crate::runtime::{context, task};

    #[cfg(all(
        tokio_unstable,
        tokio_taskdump,
        feature = "rt",
        target_os = "linux",
        any(
            target_arch = "aarch64",
            target_arch = "x86",
            target_arch = "x86_64"
        )
    ))]
    let future = task::trace::Trace::root(future);
```

```
    let id = task::Id::next();
    let task = crate::util::trace::task(future, "task", name, id.as_u64());

    // Executor 的 handle 会去调度这个 task 一次
    match context::with_current(|handle| handle.spawn(task, id)) {
        Ok(join_handle) => join_handle,
        Err(e) => panic!("{}", e),
    }
}

pub(crate) fn with_current<F, R>(f: F) -> Result<R, TryCurrentError>
where
    F: FnOnce(&scheduler::Handle) -> R,
{
    // 从 thread_local 中拿到 handle 来进行执行
    match CONTEXT.try_with(|ctx| ctx.current.handle.borrow().as_ref().map(f)) {
        Ok(Some(ret)) => Ok(ret),
        Ok(None) => Err(TryCurrentError::new_no_context()),
        Err(_access_error) =>
Err(TryCurrentError::new_thread_local_destroyed()),
    }
}
```

线程

在 Tokio 中，Runtime 主要分为两个类型：

1. 单一线程的 runtime(single thread runtime，也称为current thread runtime)
2. 多线程(线程池)的 runtime(multi thread runtime)

> 这里的所说的线程是Rust线程，而每一个Rust线程都是一个OS线程。

IO并发类任务较多时，单一线程的runtime性能不如多线程的runtime，但因为多线程runtime使用了多线程，使得线程间的通信变得更为复杂，也加重了线程间切换的开销，使得有些情况下的性能不如使用单线程runtime。因此，在要求极限性能的时候，建议测试两种工作模式的性能差距来选择更优解。

默认情况下(比如以上两种方式)，创建出来的runtime都是多线程runtime，且没有指定工作线程数量时，默认的工作线程数量将和CPU核数(虚拟核，即CPU线程数)相同。

```
use std::thread;
use std::time::Duration;
use tokio::runtime::Runtime;

```

```rust
fn main() {
  // 在第一个线程内创建一个多线程的runtime
  let t1 = thread::spawn(||{
    let rt = Runtime::new().unwrap();
    thread::sleep(Duration::from_secs(10));
  });

  // 在第二个线程内创建一个多线程的runtime
  let t2 = thread::spawn(||{
    let rt = Runtime::new().unwrap();
    thread::sleep(Duration::from_secs(10));
  });

  t1.join().unwrap();
  t2.join().unwrap();
}
```

对于4核8线程的电脑，此时总共有19个OS线程：16个worker-thread，2个spawn-thread，一个main-thread。

> runtime实现了 `Send` 和 `Sync` 这两个Trait，因此可以将runtime包在 `Arc` 里，然后跨线程使用同一个runtime。

此外还需要注意，tokio提供了两种功能的线程：

- 用于异步任务的工作线程(worker thread)
- 用于同步任务的阻塞线程(blocking thread)

单个线程或多个线程的 runtime，指的都是工作线程，即只用于执行异步任务的线程，这些任务主要是IO密集型的任务。**tokio默认会将每一个工作线程均匀地绑定到每一个CPU核心上**。

但是，有些必要的任务可能会长时间计算而占用线程，甚至任务可能是同步的，它会直接阻塞整个线程(比如 `thread::time::sleep()`)，这类任务如果计算时间或阻塞时间较短，勉强可以考虑留在异步队列中，但如果任务计算时间或阻塞时间可能会较长，它们将不适合放在异步队列中，因为它们会破坏异步调度，使得同线程中的其它异步任务处于长时间等待状态，也就是说，这些异步任务可能会被饿很长一段时间。

例如，直接在runtime中执行阻塞线程的操作，由于这类阻塞操作不在tokio系统内，tokio无法识别这类线程阻塞的操作，tokio只能等待该线程阻塞操作的结束，才能重新获得那个线程的管理权。

> 换句话说，worker thread被线程阻塞的时候，它已经脱离了tokio的控制，在一定程度上破坏了tokio的调度系统。

因此，tokio提供了这两类不同的线程。worker thread只用于执行那些异步任务，异步任务指的是不会阻塞线程的任务。而一旦遇到本该阻塞但却不会阻塞的操作(如使用 `tokio::time::sleep()` 而不是 `std::thread::sleep()`)，会直接放弃CPU，将线程交还给调度器，使该线程能够再次被调度器分配到其它异步任务。blocking thread则用于那些长时间计算的或阻塞整个线程的任务。

blocking thread 默认是不存在的，只有在调用了 `spawn_blocking()` 时才会创建一个对应的 blocking thread。

blocking thread 不用于执行异步任务，**因此runtime不会去调度管理这类线程，它们在本质上相当于一个独立的 `thread::spawn()` 创建的线程，它也不会像 `block_on()` 一样会阻塞当前线程**。它和独立线程的唯一区别，是blocking thread是在runtime内的，可以在runtime内对它们使用一些异步操作，例如await。

```rust
use std::thread;
use chrono::Local;
use tokio::{self, runtime::Runtime, time};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt1 = Runtime::new().unwrap();
    // 创建一个blocking thread，可立即执行(由操作系统调度系统决定何时执行)
    // 注意，不阻塞当前线程
    let task = rt1.spawn_blocking(|| {
      println!("in task: {}", now());
      // 注意，是线程的睡眠，不是tokio的睡眠，因此会阻塞整个线程
      thread::sleep(std::time::Duration::from_secs(10))
    });

    // 小睡1毫秒，让上面的blocking thread先运行起来
    std::thread::sleep(std::time::Duration::from_millis(1));
    println!("not blocking: {}", now());

    // 可在runtime内等待blocking_thread的完成
    rt1.block_on(async {
      task.await.unwrap();
      println!("after blocking task: {}", now());
    });
}

// 输出
in task: 2024-02-19 16:18:36
not blocking: 2024-02-19 16:18:36
after blocking task: 2024-02-19 16:18:46
```

再次强调，blocking thread生成的任务虽然绑定了runtime，但是它不是异步任务，不受tokio调度系统控制。因此，如果在 `block_on()` 中生成了blocking thread或普通的线程， `block_on()` 不会

等待这些线程的完成。

```
rt.block_on(async{
  // 生成一个blocking thread和一个独立的thread
  // block on不会阻塞等待两个线程终止，因此block_on在这里会立即返回
  rt.spawn_blocking(|| std::thread::sleep(std::time::Duration::from_secs(10)));
  thread::spawn(||std::thread::sleep(std::time::Duration::from_secs(10)));
});
```

Tokio 允许的 blocking thread 队列很长(默认512个)，且可以在 runtime build 时通过 `max_blocking_threads()` 配置最大长度。如果超出了最大队列长度，新的任务将放在一个等待队列中进行等待(比如当前已经有512个正在运行的任务，下一个任务将等待，直到有某个 blocking thread 空闲)。

blocking thread 执行完对应任务后，并不会立即释放，而是继续保持活动状态一段时间，此时它们的状态是空闲状态。当空闲时长超出一定时间后(可在runtime build时通过 `thread_keep_alive()` 配置空闲的超时时长)，该空闲线程将被释放。

blocking thread 有时候是非常友好的，它像独立线程一样，但又和 runtime 绑定，它不受 tokio 的调度系统调度， tokio 不会把其它任务放进该线程，也不会把该线程内的任务转移到其它线程。换言之，它有机会完完整整地发挥单个线程的全部能力，而不像 worker 线程一样，可能会被调度器打断。

附录

Pin

自引用结构体（Self-Referential Structs）是一个这个样的结构体，它内部某个成员是对另外一个成员的引用：

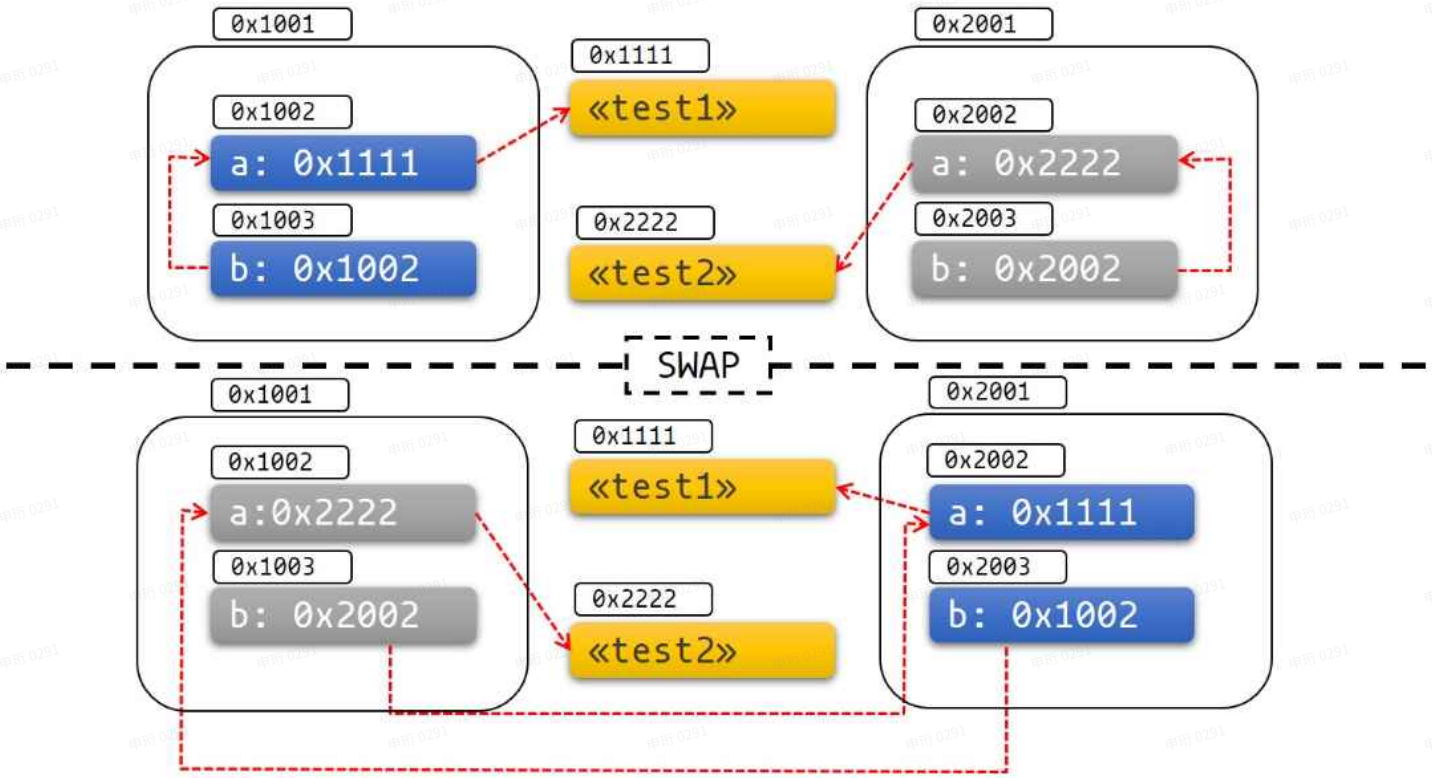
```
struct Test<'a> {
    a: String,
    b: &'a String,
}

fn main() {
    let a = String::from("Hello");
    let _test = Test { a, b: &a };
//  编译不过:
//    let _test = Test { a, b: &a };
//  |                        -     ^^ value borrowed here after move
//  |                        |
//  |                        value moved here
}
```

如果使用指针绕开这个问题：

```rust
#[derive(Debug)]
struct Test {
    a: String,
    b: *const String, // 改成指针
}

impl Test {
    fn new(txt: &str) -> Self {
        Test {
            a: String::from(txt),
            b: std::ptr::null(),
        }
    }

    fn init(&mut self) {
        let self_ref: *const String = &self.a;
        self.b = self_ref;
    }

    fn a(&self) -> &str {
        &self.a
    }

    fn b(&self) -> &String {
        unsafe {&*(self.b)}
    }
}
/// 如果出现 move 的话
fn main() {
    let mut test1 = Test::new("test1");
    test1.init();
    let mut test2 = Test::new("test2");
    test2.init();

    println!("a: {}, b: {}", test1.a(), test1.b());
    // 使用swap()函数交换两者，这里发生了move
    std::mem::swap(&mut test1, &mut test2);
    test1.a = "I've totally changed now!".to_string();
    println!("a: {}, b: {}", test2.a(), test2.b());
}

/// 会出现奇怪的结果
```

```
43 a: test1, b: test1
44 a: test1, b: I've totally changed now!
```

有没有发现，出问题了！问题出在哪？原因是Test结构体中的字段b是一个指向字段a的指针，它在栈上存的是字段a的地址。通过 `swap()` 函数交换两个Test结构体之后，字段a,b分别移动到对方的内存区域上，但是a和b本身的内容没有变。也就是指针b依然指向的是原来的地址，但是这个地址现在已经属于另外一个结构体了！这不仅不是自引用结构体了，更可怕的是这个指针可能导致更危险的问题，这是Rust决不允许出现的！👇下面这张图可以帮助理解：



上面出现的问题根因在于 `std::mem::swap(&mut test1, &mut test2)` 允许在这种情况下交换数据，此时出现了 UB。

```rust
#[inline]
#[stable(feature = "rust1", since = "1.0.0")]
pub fn swap<T>(x: &mut T, y: &mut T) {
    // SAFETY: the raw pointers have been created from safe mutable references satisfying all the
    // constraints on `ptr::swap_nonoverlapping_one`
    unsafe {
        ptr::swap_nonoverlapping_one(x, y);
    }
}
```

而实际上，异步代码块(函数)本身会含有一个对自己的引用，这也是为什么自引用问题在异步中如此重要的原因。但其实解决他的方法也十分简单，其实就是依靠类型系统(这也是为什么老外对于 Rust 的类

型系统极为自豪的原因，他们很喜欢说"with type system")，如果一个类型的子类是 Unpin 的，那么其父类就是 Unpin 的，而 Unpin 的数据只需要禁止其可变指针的直接使用即可。

> 我们找到了问题的根源在哪，**Pin**就是从根源上解决这个问题的。现在我们很清晰了，似乎是不是可以用一句话概括：**Pin***就是一个不会让你在Safe Rust暴露可变借用**&mut 的智能指针?*
>
> 答案是：不全正确。这就是**Pin**概念起初让大家一脸懵逼的地方。下面让**Pin**自己来解答大家的疑惑，**Pin**说：“你们不是想让我保证被我包裹的指针P<T>永远钉住不让move吗？我可以答应，但我有一个原则。那就是我永远不能钉住持有通行证的朋友，这张通行证就是**Unpin**。如果没有这张通行证，请放心，我会把你钉得死死的！”

实际上就是对于一个 Pin 的类型：

```rust
/// 1. 直接 get_mut()
impl<'a, T: ?Sized> Pin<&'a mut T> {
    #[stable(feature = "pin", since = "1.33.0")]
    #[inline(always)]
    pub fn get_mut(self) -> &'a mut T where T: Unpin {
        self.pointer
    }
}
/// 2. 继承 DerefMut
#[stable(feature = "pin", since = "1.33.0")]
impl<P: DerefMut<Target: Unpin>> DerefMut for Pin<P> {
    fn deref_mut(&mut self) -> &mut P::Target {
        Pin::get_mut(Pin::as_mut(self))
    }
}
```

从上面代码可以看出，当且仅当 `Pin` 中的数据是 `Unpin` 的才可以拿到可变指针，如果可变指针都拿不到，何尝来谈起自引用问题。

大多数情况下我们无需考虑 Pin 的问题，因为所有的类型都是默认 Unpin 的：

```rust
#[lang = "unpin"]
pub auto trait Unpin {}

#[stable(feature = "pin", since = "1.33.0")]
impl<'a, T: ?Sized + 'a> Unpin for &'a T {}

#[stable(feature = "pin", since = "1.33.0")]
impl<'a, T: ?Sized + 'a> Unpin for &'a mut T {}

#[stable(feature = "pin_raw", since = "1.38.0")]
impl<T: ?Sized> Unpin for *const T {}
```

```rust

#[stable(feature = "pin_raw", since = "1.38.0")]
impl<T: ?Sized> Unpin for *mut T {}
```

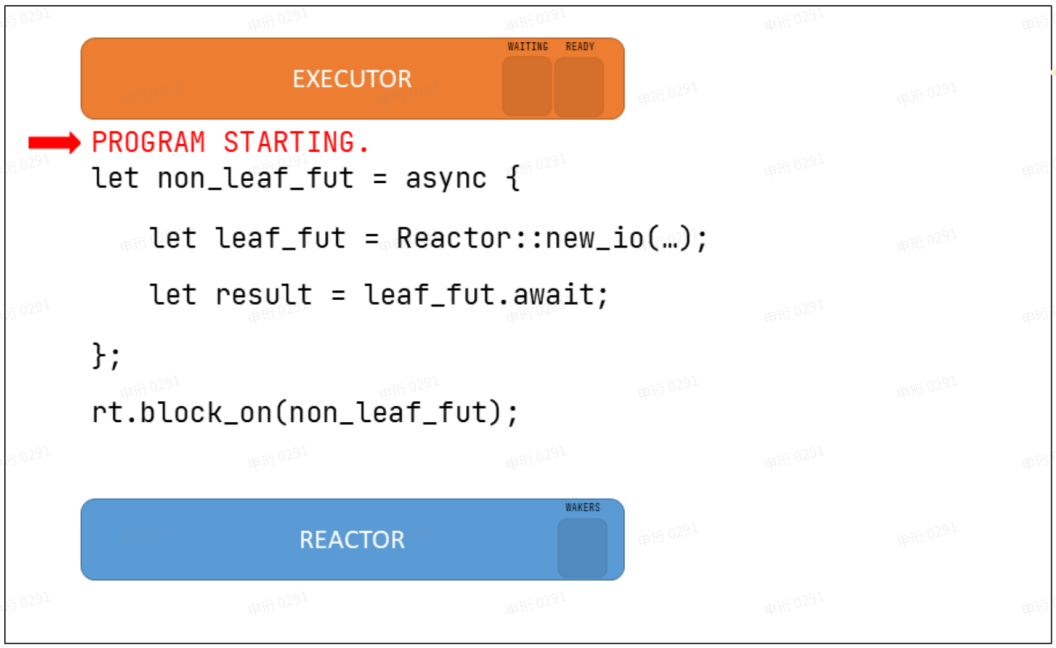
如果要标记一个类型是 `!Unpin` 的，则需要使用到了幽灵类型(不占任何内存，仅仅标记父 struct act like T in it)：

```rust
#[stable(feature = "pin", since = "1.33.0")]
#[derive(Debug, Copy, Clone, Eq, PartialEq, Ord, PartialOrd, Hash)]
pub struct PhantomPinned;

#[stable(feature = "pin", since = "1.33.0")]
impl !Unpin for PhantomPinned {}
```

只需要在类型中加入 `PhantomPinned` ，此时这个类型就是 `!Unpin` 的。此外也可以自己手动 `impl !Unpin for XXX {}` 。Rust 还有一个 unsafe 的接口提供给高阶玩家：

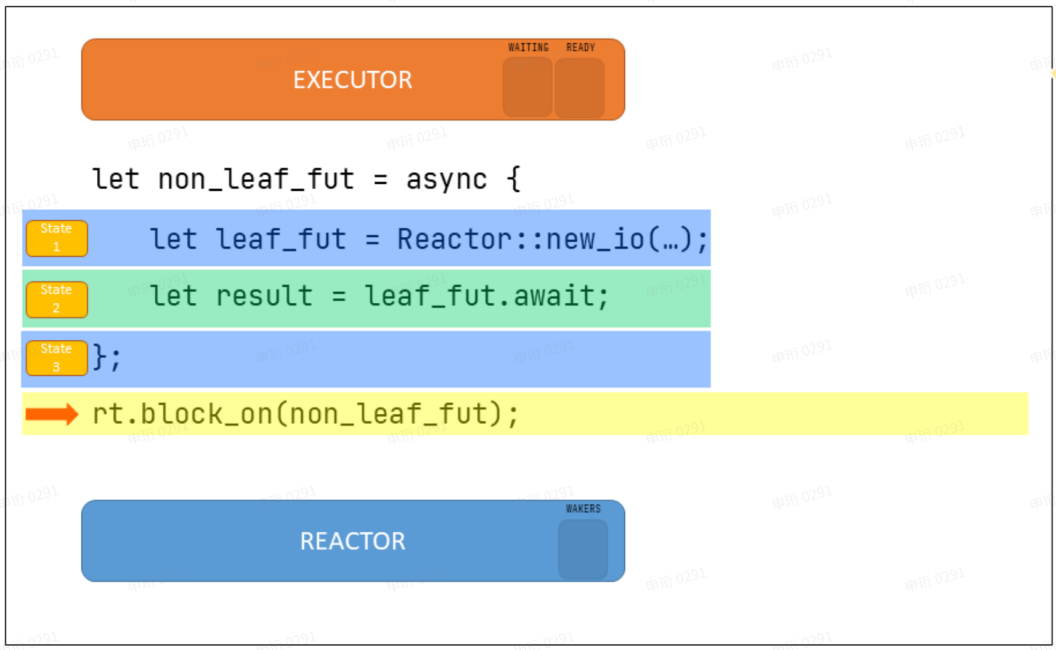
```rust
/// 不管你有没有实现Unpin，你都可以通过调用这个方法拿到&mut T
impl<'a, T: ?Sized> Pin<&'a mut T> {
    #[stable(feature = "pin", since = "1.33.0")]
    #[inline(always)]
    pub unsafe fn get_unchecked_mut(self) -> &'a mut T {
        self.pointer
    }
}
```

德国大佬的流程图



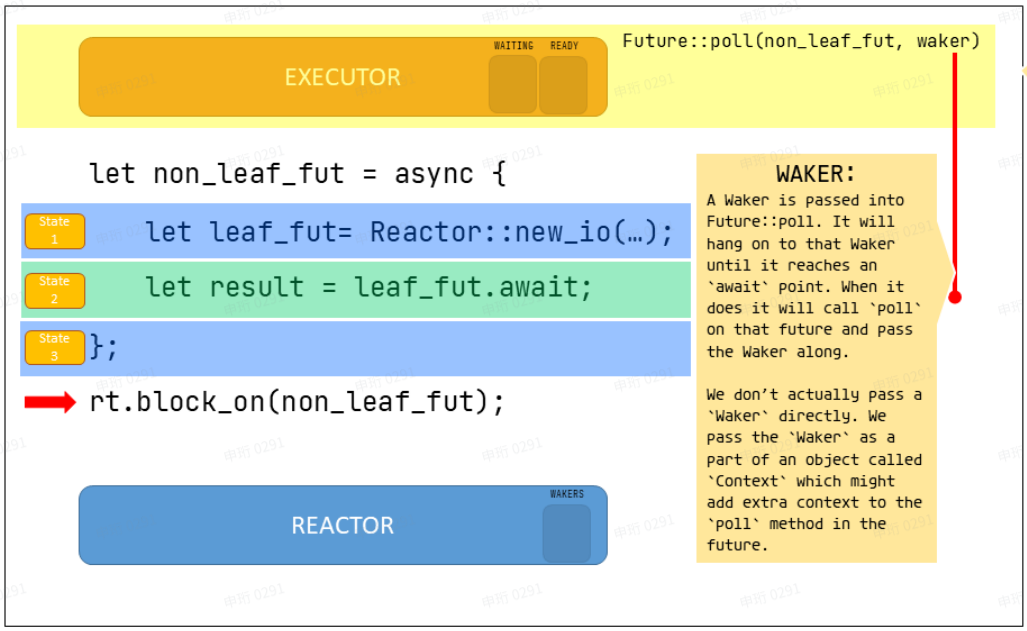
TIP:
I'll provide explanations and extra information in these boxes along the way.

0



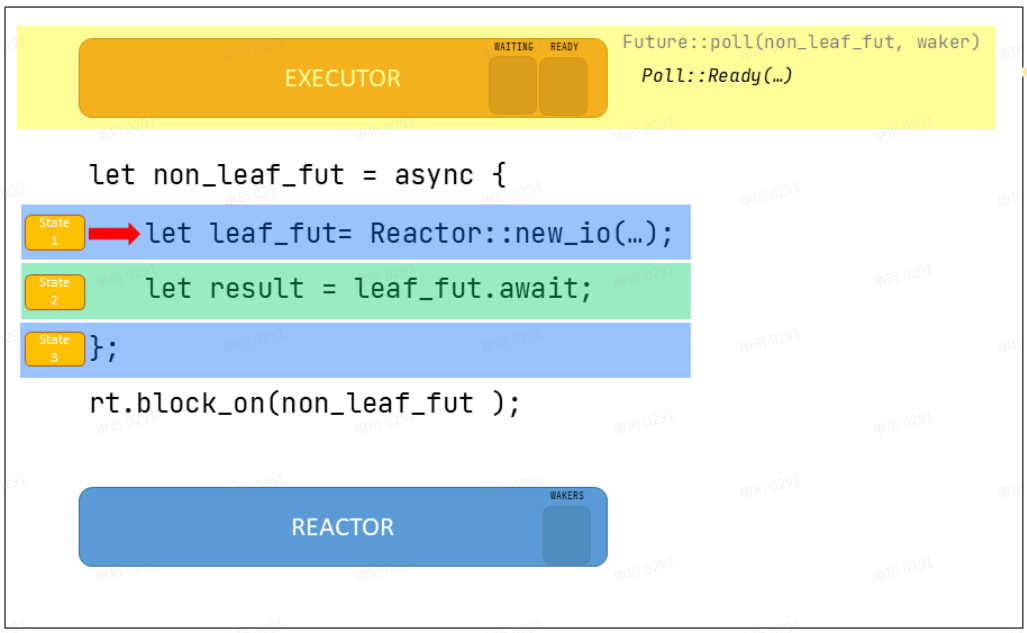
TIP:
The «async» keyword rewrites our code block to a state machine. Each «await» point represents a state change.

2



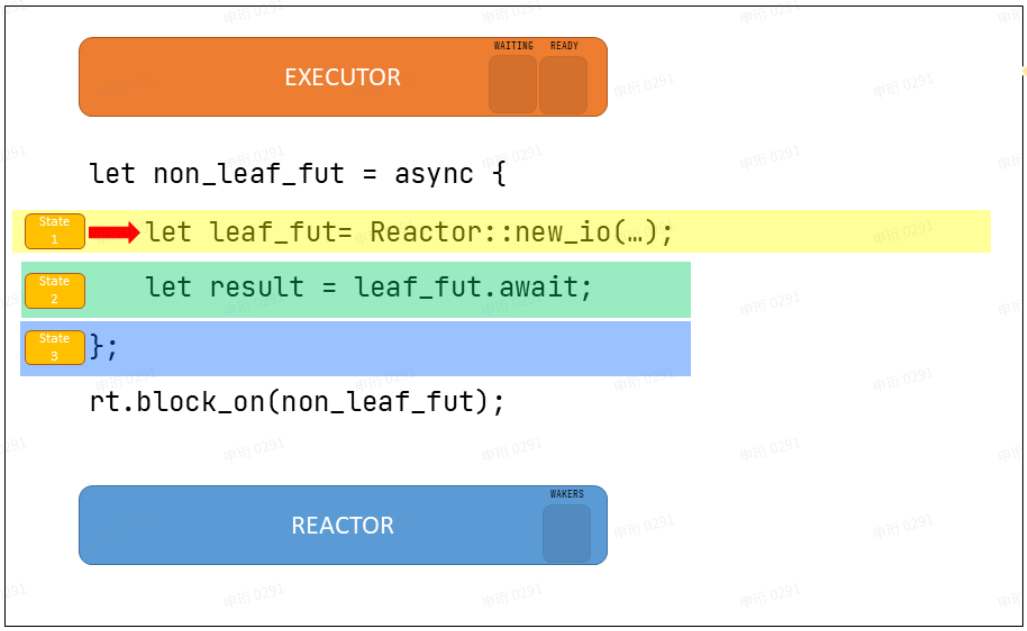
TIP:
«Future::Poll» could naively be thought of as calling «non_leaf_fut.poll(waker)» but due to its signature we can't call it as an instance method.

3



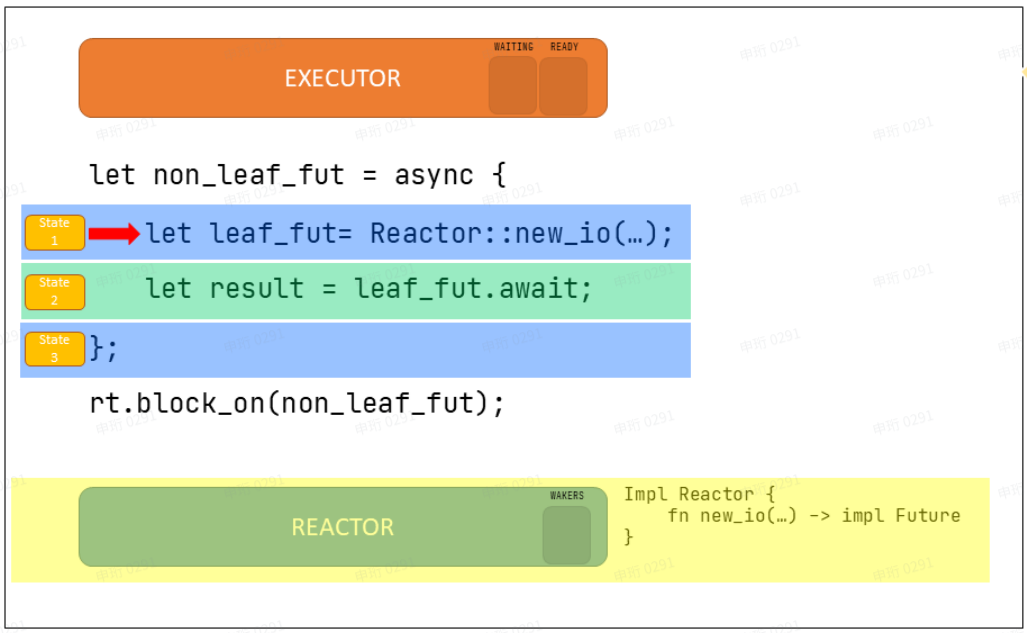
TIP:
Since this «non-leaf future» is rewritten to a state machine. The first call to «poll» will actually run the code block before the first «await».

4



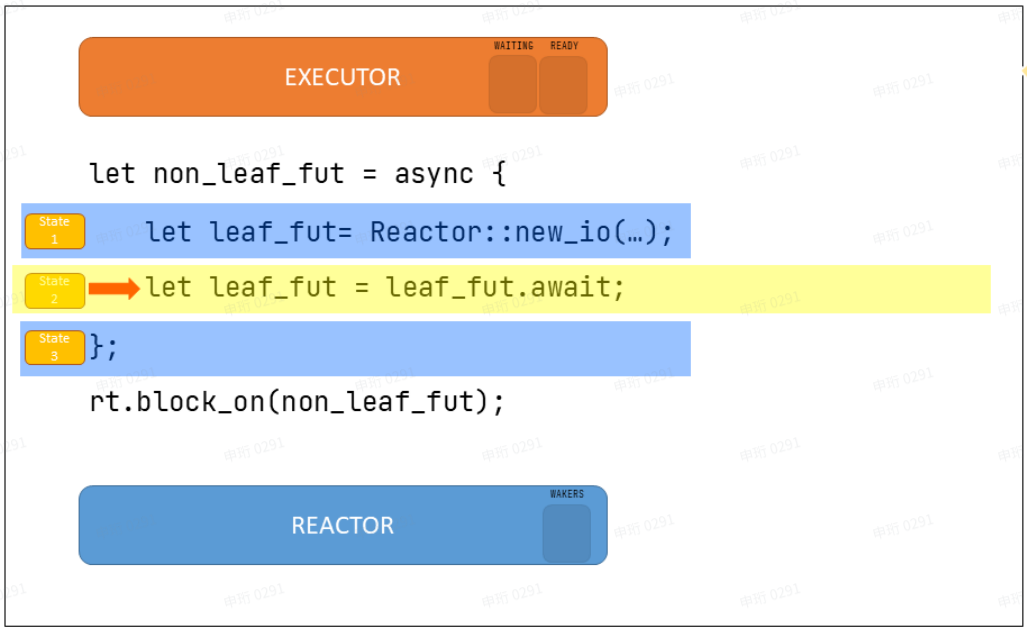
TIP:
Leaf futures are created by the Reactor. It represents an I/O resource, like a network call. Nothing special happens when we create the future…

5



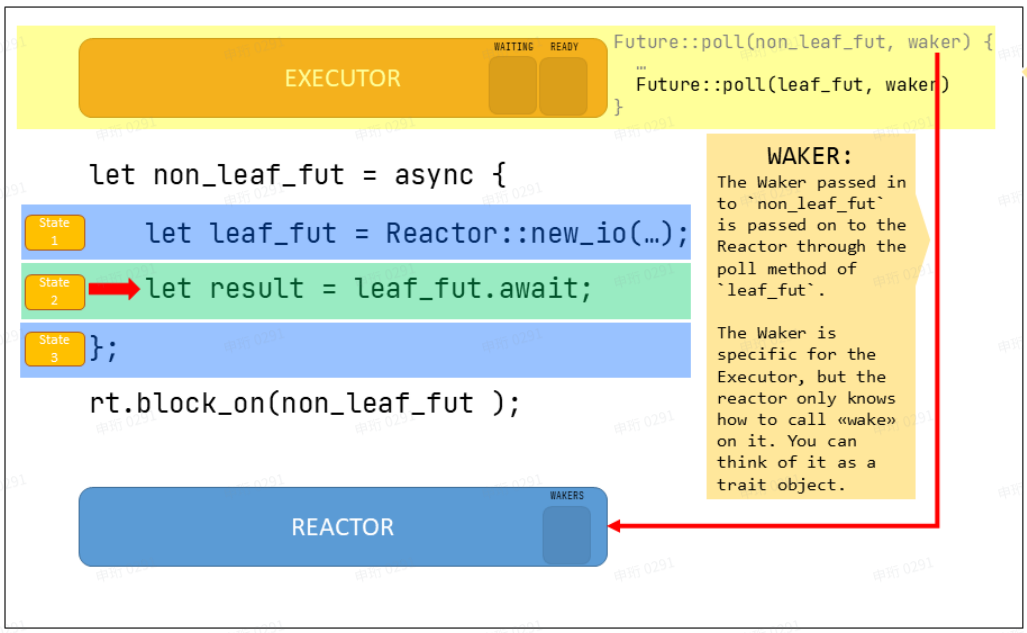
TIP:
…the Reactor just creates an object implementing the «Future» trait and returns it.

6



TIP:
As mentioned in 5, nothing happens when the leaf future is created, but once we «await» it several pieces starts moving…

7

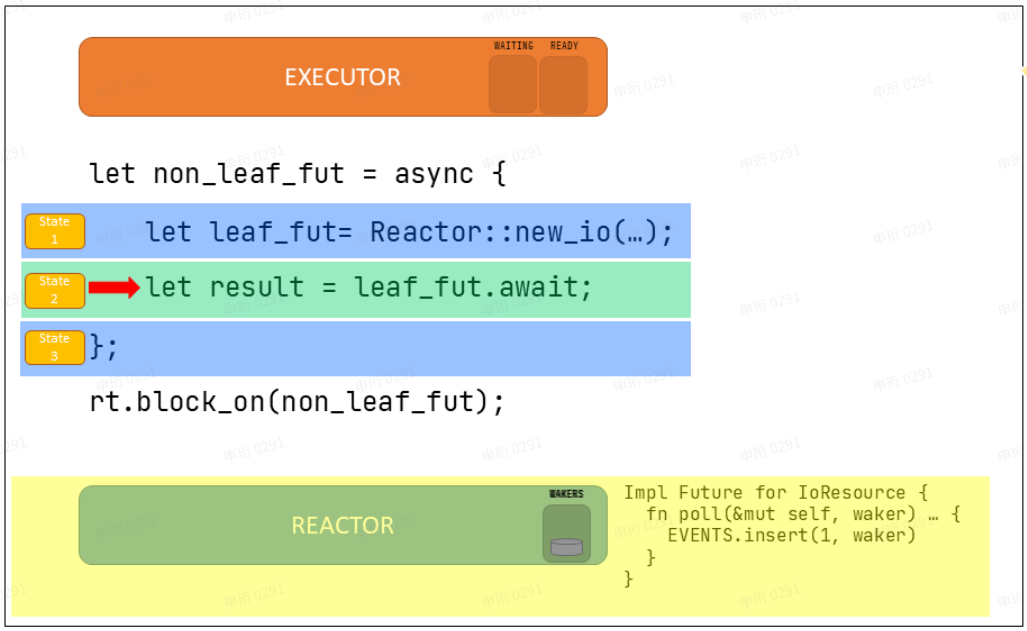


WAKER:
The Waker passed in to `non_leaf_fut` is passed on to the Reactor through the poll method of `leaf_fut`.

The Waker is specific for the Executor, but the reactor only knows how to call «wake» on it. You can think of it as a trait object.

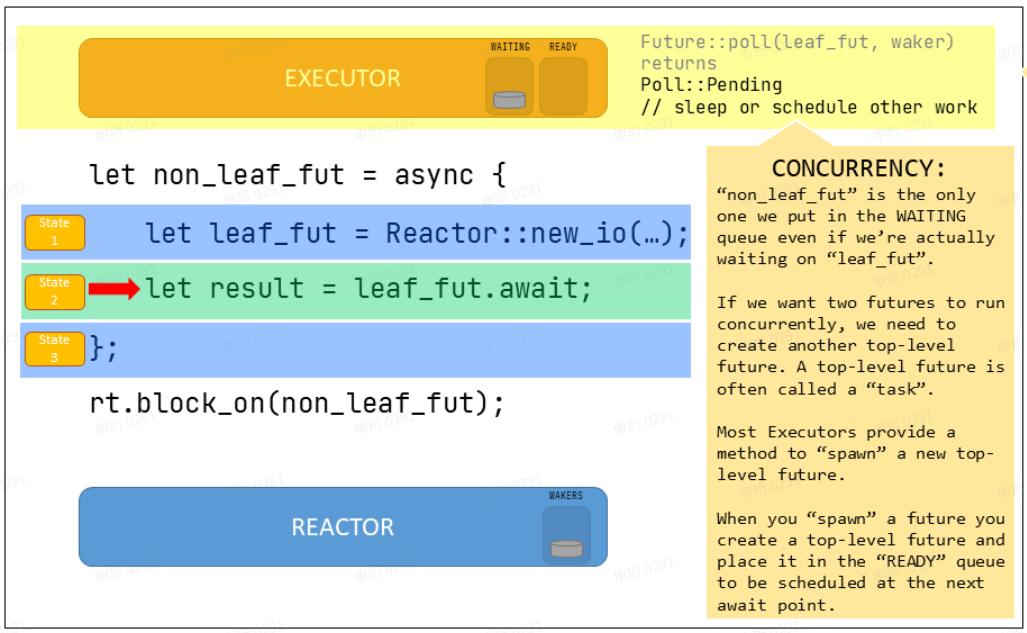
TIP:
Once again, we can think of this as «leaf_fut.poll(waker)». We pass inn a Waker to the leaf future created by the Reactor.

8



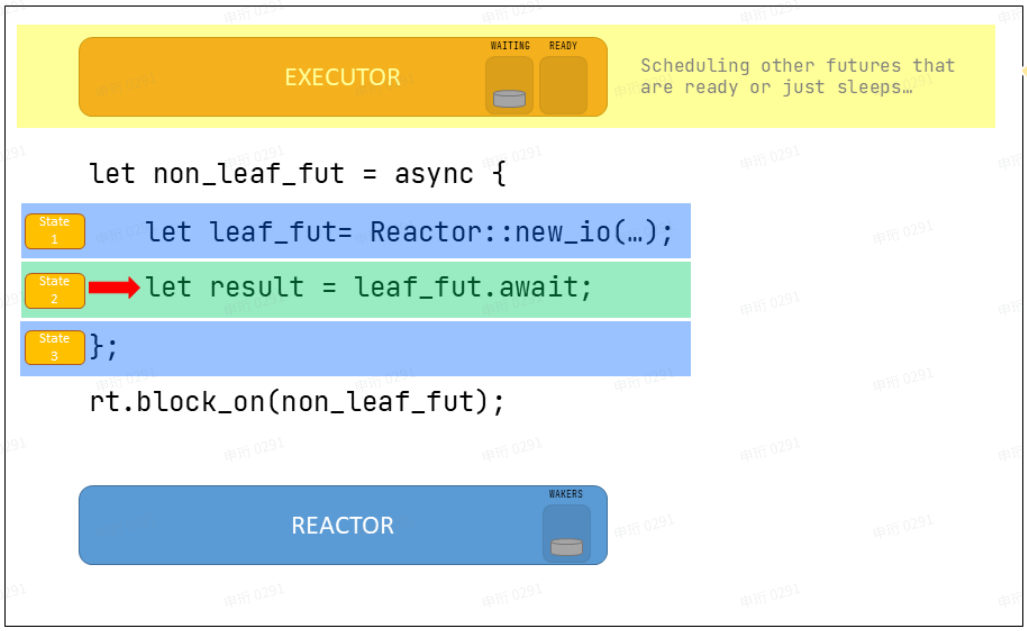
TIP:
Here we see the implementation of the Future trait for the object created by the Reactor. It stores the Waker together with the token «1» identifying the event.

9



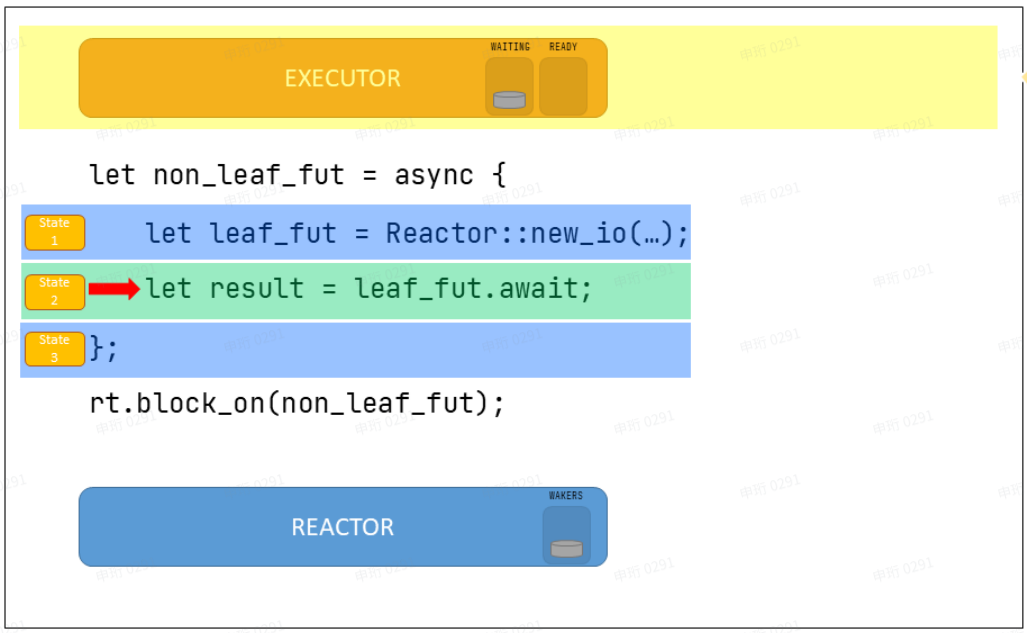
TIP:
Poll (see 9) returns «Poll::Pending» since we actually have to wait until the I/O resource is ready. It places the Future in the waiting queue

10



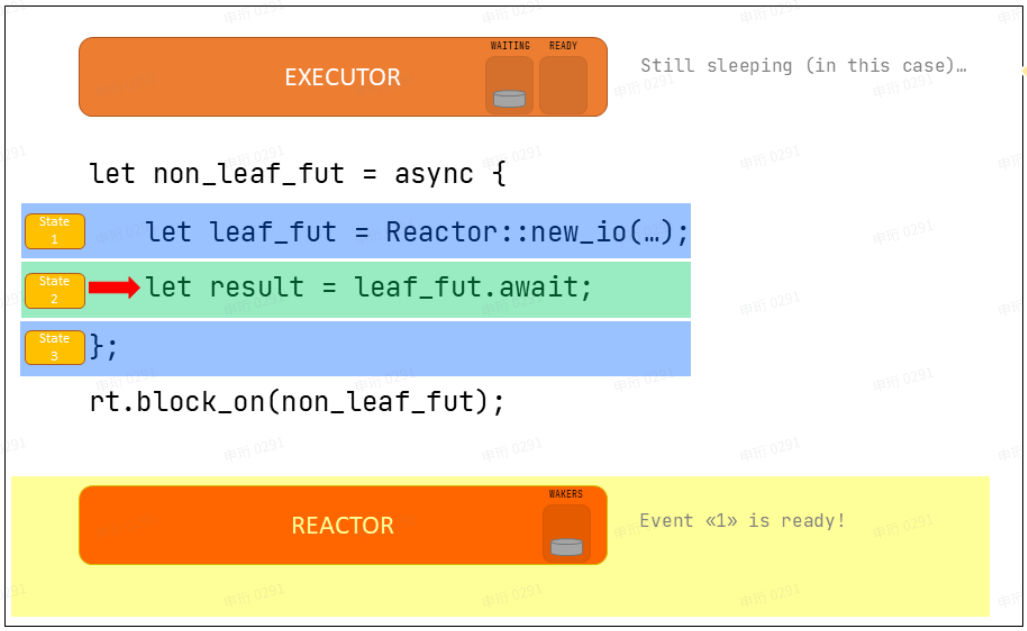
TIP:
At this point the executor will look through its Ready queue to see if there are any Futures ready to progress or it will sleep to preserve resources

11



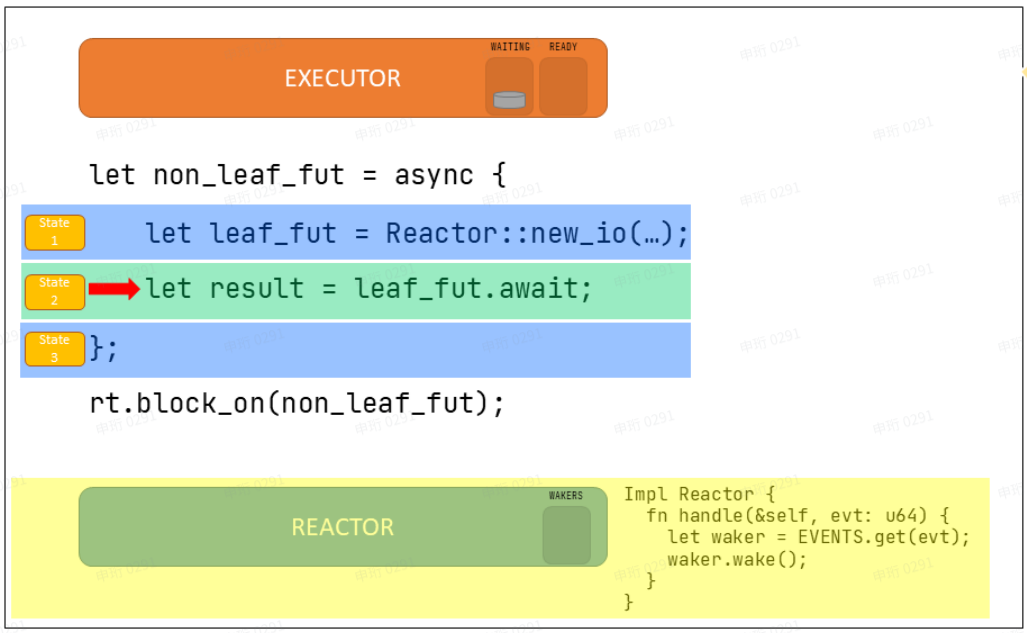
TIP:
In our example it just sleeps. If we were to poll our top-level future once more it would be «stuck» at State 2, returning Poll::Pending. It only advances once it returns Poll::Ready

12



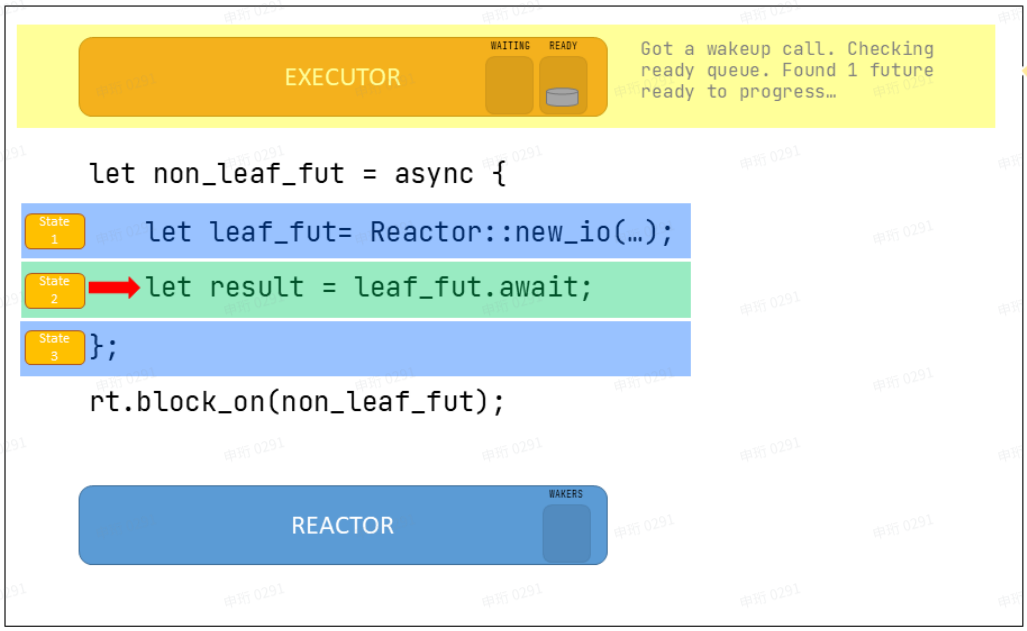
TIP:
The Reactor notices (or gets notified) that an event is ready

13



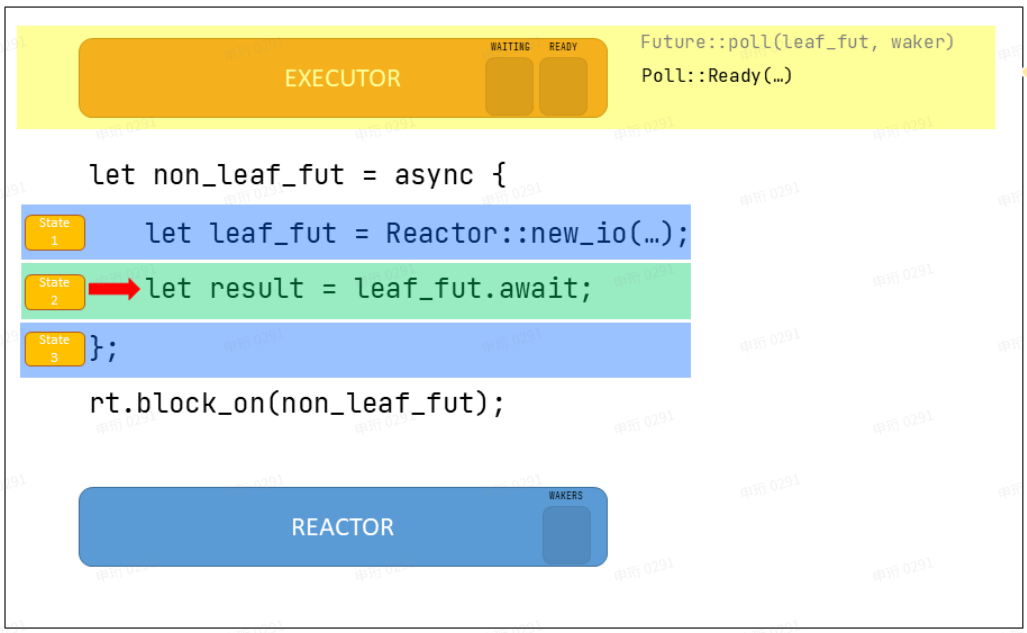
TIP:
The token is «1» so it fetches the Waker associated with that token and calls «Waker::wake».

14



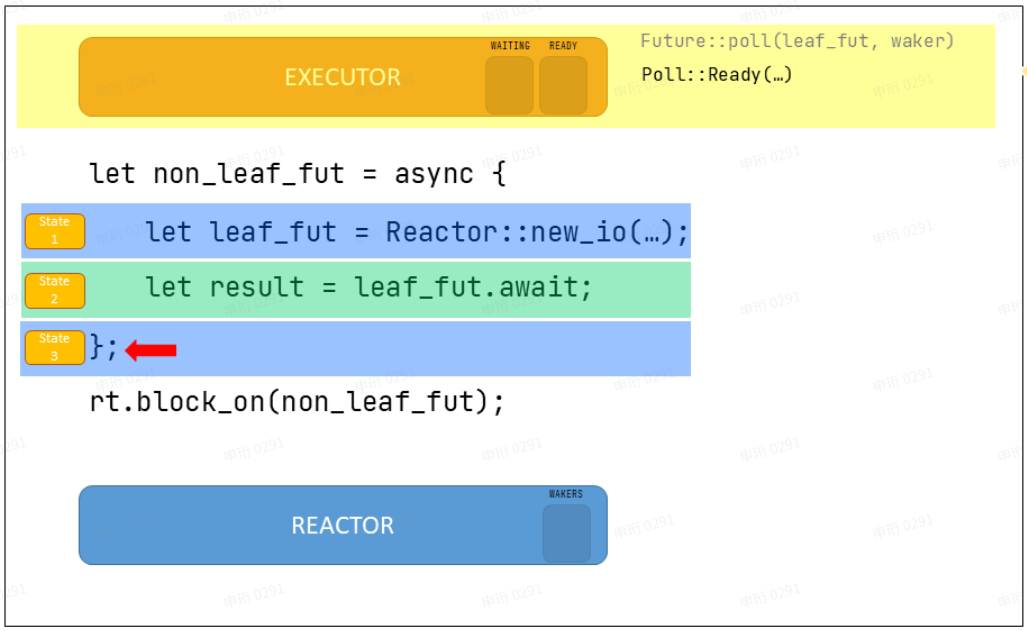
TIP:
Calling «wake» places our state-machine future (non_leaf_fut) in the Ready queue and wakes up our executor

15



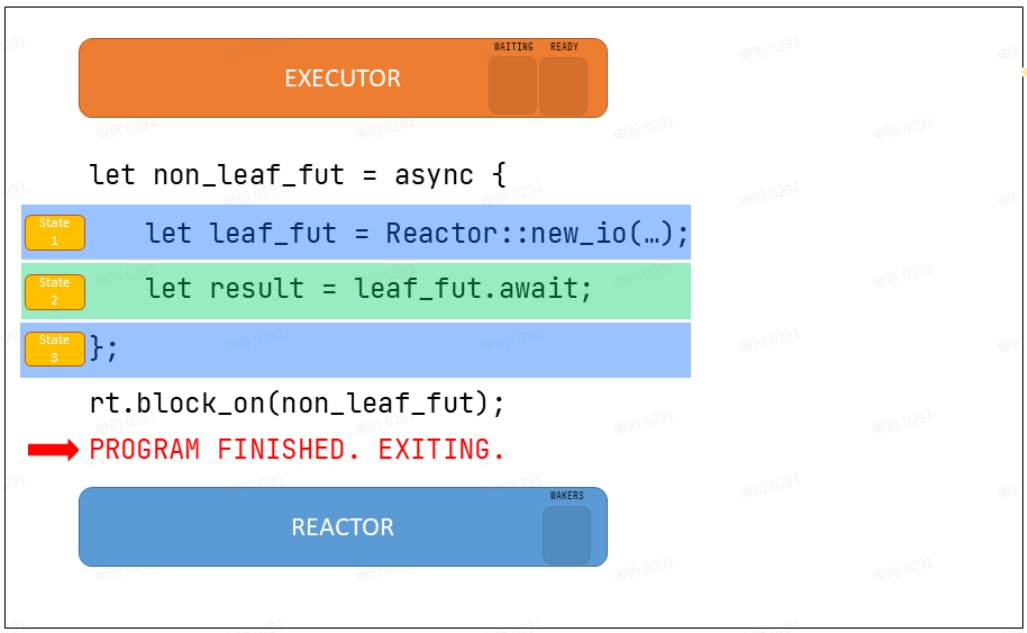
TIP:
The Executor takes the future(s) in Ready queue out and calls «Future::poll» once more. This time the future returns «Poll::Ready» with some data…

16



TIP:
Our state machine advances one more only to discover that it's finished.

17



TIP:
Since there are no more futures in the Waiting queue the executor returns from «block_on». Our program is finished.

18