

19 | 如何用协程来优化多线程业务？

2019-07-02 刘超



你好，我是刘超。

近一两年，国内很多互联网公司开始使用或转型Go语言，其中一个很重要的原因就是Go语言优越的性能表现，而这个优势与Go实现的轻量级线程Goroutines（协程Coroutine）不无关系。那么Go协程的实现与Java线程的实现有什么区别呢？

线程实现模型

了解协程和线程的区别之前，我们不妨先来了解下底层实现线程几种方式，为后面的学习打个基础。

实现线程主要有三种方式：轻量级进程和内核线程一对一相互映射实现的1:1线程模型、用户线程和内核线程实现的N:1线程模型以及用户线程和轻量级进程混合实现的N:M线程模型。

1:1线程模型

以上我提到的内核线程（Kernel-Level Thread, KLT）是由操作系统内核支持的线程，内核通过调度器对线程进行调度，并负责完成线程的切换。

我们知道在Linux操作系统编程中，往往都是通过fork()函数创建一个子进程来代表一个内核中的线程。一个进程调用fork()函数后，系统会先给新的进程分配资源，例如，存储数据和代码的空间。然后把原来进程的所有值都复制到新的进程中，只有少数值与原来进程的值（比如PID）不同，这相当于复制了一个主进程。

采用**fork()**创建子进程的方式来实现并行运行，会产生大量冗余数据，即占用大量内存空间，又消耗大量**CPU**时间用来初始化内存空间以及复制数据。

如果是一份一样的数据，为什么不共享主进程的这一份数据呢？这时候轻量级进程（**Light Weight Process**，即**LWP**）出现了。

相对于**fork()**系统调用创建的线程来说，**LWP**使用**clone()**系统调用创建线程，该函数是将部分父进程的资源的数据结构进行复制，复制内容可选，且没有被复制的资源可以通过指针共享给予进程。因此，轻量级进程的运行单元更小，运行速度更快。**LWP**是跟内核线程一对一映射的，每个**LWP**都是由一个内核线程支持。

N:1线程模型

1:1线程模型由于跟内核是一对一映射，所以在线程创建、切换上都存在用户态和内核态的切换，性能开销比较大。除此之外，它还存在局限性，主要就是指系统的资源有限，不能支持创建大量的**LWP**。

N:1线程模型就可以很好地解决**1:1**线程模型的这两个问题。

该线程模型是在用户空间完成了线程的创建、同步、销毁和调度，已经不需要内核的帮助了，也就是说在线程创建、同步、销毁的过程中不会产生用户态和内核态的空间切换，因此线程的操作非常快速且低消耗。

N:M线程模型

N:1线程模型的缺点在于操作系统不能感知用户态的线程，因此容易造成某一个线程进行系统调用内核线程时被阻塞，从而导致整个进程被阻塞。

N:M线程模型是基于上述两种线程模型实现的一种混合线程管理模型，即支持用户态线程通过**LWP**与内核线程连接，用户态的线程数量和内核态的**LWP**数量是**N:M**的映射关系。

了解完这三个线程模型，你就可以清楚地了解到**Go**协程的实现与**Java**线程的实现有什么区别了。

JDK 1.8 Thread.java 中 **Thread#start** 方法的实现，实际上是通过**Native**调用**start0**方法实现的；在**Linux**下，**JVM Thread**的实现是基于**pthread_create**实现的，而**pthread_create**实际上是调用了**clone()**完成系统调用创建线程的。

所以，目前**Java**在**Linux**操作系统下采用的是用户线程加轻量级线程，一个用户线程映射到一个内核线程，即**1:1**线程模型。由于线程是通过内核调度，从一个线程切换到另一个线程就涉及到了上下文切换。

而**Go**语言是使用了**N:M**线程模型实现了自己的调度器，它在**N**个内核线程上多路复用（或调度）**M**个协程，协程的上下文切换是在用户态由协程调度器完成的，因此不需要陷入内核，相比

之下，这个代价就很小了。

协程的实现原理

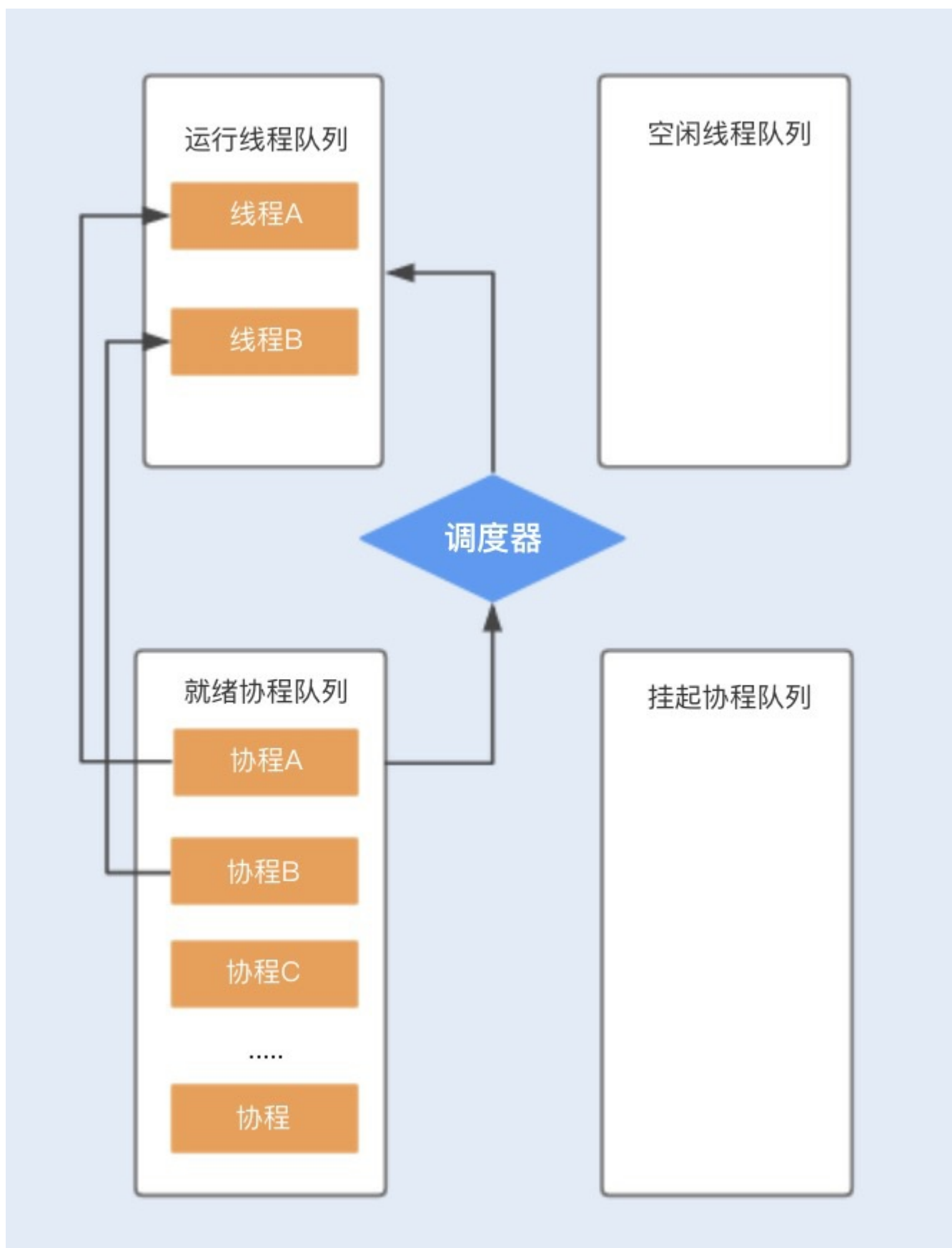
协程不只在Go语言中实现了，其实目前大部分语言都实现了自己的一套协程，包括C#、erlang、python、lua、javascript、ruby等。

相对于协程，你可能对进程和线程更为熟悉。进程一般代表一个应用服务，在一个应用服务中可以创建多个线程，而协程与进程、线程的概念不一样，**我们可以将协程看作是一个类函数或者一块函数中的代码**，我们可以在一个主线程里面轻松创建多个协程。

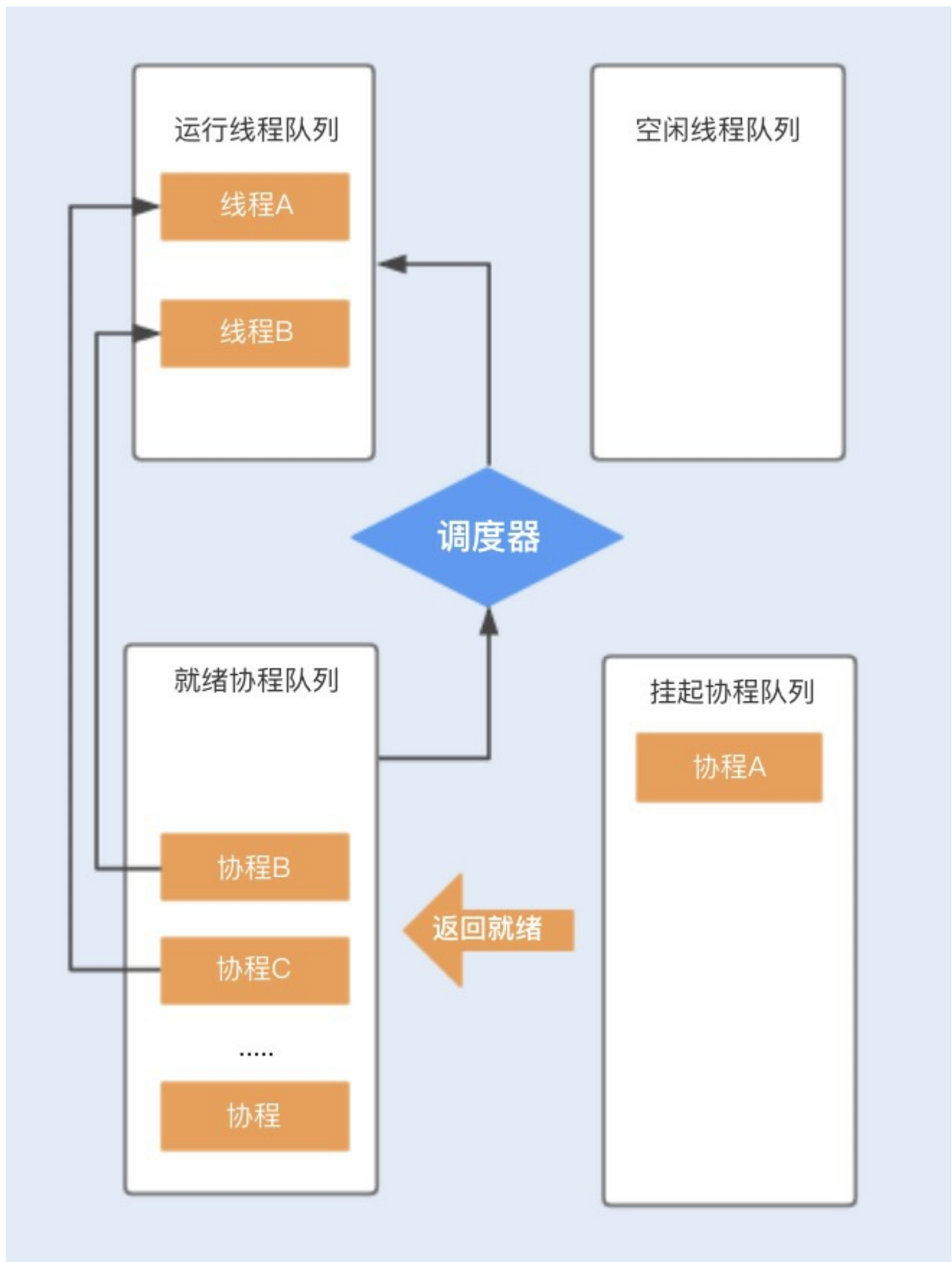
程序调用协程与调用函数不一样的是，协程可以通过暂停或者阻塞的方式将协程的执行挂起，而其它协程可以继续执行。这里的挂起只是在程序中（用户态）的挂起，同时将代码执行权转让给其它协程使用，待获取执行权的协程执行完成之后，将从挂起点唤醒挂起的协程。协程的挂起和唤醒是通过一个调度器来完成的。

结合下图，你可以更清楚地了解到基于N:M线程模型实现的协程是如何工作的。

假设程序中默认创建两个线程为协程使用，在主线程中创建协程ABCD...，分别存储在就绪队列中，调度器首先会分配一个工作线程A执行协程A，另外一个工作线程B执行协程B，其它创建的协程将会放在队列中进行排队等待。



当协程A调用暂停方法或被阻塞时，协程A会进入到挂起队列，调度器会调用等待队列中的其它协程抢占线程A执行。当协程A被唤醒时，它需要重新进入到就绪队列中，通过调度器抢占线程，如果抢占成功，就继续执行协程A，失败则继续等待抢占线程。



相比线程，协程少了由于同步资源竞争带来的CPU上下文切换，I/O密集型的应用比较适合使用，特别是在网络请求中，有较多的时间在等待后端响应，协程可以保证线程不会阻塞在等待网络响应中，充分利用了多核多线程的能力。而对于CPU密集型的应用，由于在多数情况下CPU都比较繁忙，协程的优势就不是特别明显了。

Kilim协程框架

虽然这么多的语言都实现了协程，但目前Java原生语言暂时还不支持协程。不过你也不用泄气，我们可以通过协程框架在Java中使用协程。

目前Kilim协程框架在Java中应用得比较多，通过这个框架，开发人员就可以低成本地在Java中使用协程了。

在Java中引入 [Kilim](#)，和我们平时引入第三方组件不太一样，除了引入jar包之外，还需要通过Kilim提供的织入（Weaver）工具对Java代码编译生成的字节码进行增强处理，比如，识别哪些方式是可暂停的，对相关的方法添加上下文处理。通常有以下四种方式可以实现这种织入操作：

- 在编译时使用maven插件；
- 在运行时调用kilim.tools.Weaver工具；
- 在运行时使用kilim.tools.Kilim invoking调用Kilim的类文件；
- 在main函数添加 `if (kilim.tools.Kilim.trampoline(false,args)) return`。

Kilim框架包含了四个核心组件，分别为：任务载体（Task）、任务上下文（Fiber）、任务调度器（Scheduler）以及通信载体（Mailbox）。



Task对象主要用来执行业务逻辑，我们可以把这个比作多线程的Thread，与Thread类似，Task中也有一个run方法，不过在Task中方法名为execute，我们可以将协程里面要做的业务逻辑操作

写在`execute`方法中。

与`Thread`实现的线程一样，`Task`实现的协程也有状态，包括：`Ready`、`Running`、`Pausing`、`Paused`以及`Done`总共五种。`Task`对象被创建后，处于`Ready`状态，在调用`execute()`方法后，协程处于`Running`状态，在运行期间，协程可以被暂停，暂停中的状态为`Pausing`，暂停后的状态为`Paused`，暂停后的协程可以被再次唤醒。协程正常结束后的状态为`Done`。

`Fiber`对象与`Java`的线程栈类似，主要用来维护`Task`的执行堆栈，`Fiber`是实现N:M线程映射的关键。

`Scheduler`是`Kilim`实现协程的核心调度器，`Scheduler`负责分派`Task`给指定的工作者线程`WorkerThread`执行，工作者线程`WorkerThread`默认初始化个数为机器的CPU个数。

`Mailbox`对象类似一个邮箱，协程之间可以依靠邮箱来进行通信和数据共享。协程与线程最大的不同就是，线程是通过共享内存来实现数据共享，而协程是使用了通信的方式来实现了数据共享，主要就是为了避免内存共享数据而带来的线程安全问题。

协程与线程的性能比较

接下来，我们通过一个简单的生产者和消费者的案例，来对比下协程和线程的性能。可通过[Github](#) 下载本地运行代码。

Java多线程实现源码：

```
public class MyThread {
    private static Integer count = 0; //
    private static final Integer FULL = 10; //最大生产数量
    private static String LOCK = "lock"; //资源锁

    public static void main(String[] args) {
        MyThread test1 = new MyThread();

        long start = System.currentTimeMillis();

        List<Thread> list = new ArrayList<Thread>();
        for (int i = 0; i < 1000; i++) { //创建五个生产者线程
            Thread thread = new Thread(test1.new Producer());
            thread.start();
            list.add(thread);
        }
    }
}
```

```

for (int i = 0; i < 1000; i++) { //创建五个消费者线程
    Thread thread = new Thread(test1.new Consumer());
    thread.start();
    list.add(thread);
}

try {
    for (Thread thread : list) {
        thread.join(); //等待所有线程执行完
    }
} catch (InterruptedException e) {
    e.printStackTrace();
}

long end = System.currentTimeMillis();
System.out.println("子线程执行时长: " + (end - start));
}

//生产者
class Producer implements Runnable {
    public void run() {
        for (int i = 0; i < 10; i++) {
            synchronized (LOCK) {
                while (count == FULL) { //当数量满了时
                    try {
                        LOCK.wait();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
                count++;
                System.out.println(Thread.currentThread().getName() + "生产者生产，目前总共有" + count);
                LOCK.notifyAll();
            }
        }
    }
}

//消费者

```



```
class Consumer implements Runnable {  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            synchronized (LOCK) {  
                while (count == 0) { //当数量为零时  
                    try {  
                        LOCK.wait();  
                    } catch (Exception e) {  
                    }  
                }  
                count--;  
                System.out.println(Thread.currentThread().getName() + "消费者消费，目前总共有" + count);  
                LOCK.notifyAll();  
            }  
        }  
    }  
}
```

Kilim协程框架实现源码：

```
public class Coroutine {

    static Map<Integer, Mailbox<Integer>> mailMap = new HashMap<Integer, Mailbox<Integer>>(); //为每个协程创建

    public static void main(String[] args) {

        if (kilim.tools.Kilim.trampoline(false, args)) return;
        Properties propes = new Properties();
        propes.setProperty("kilim.Scheduler.numThreads", "1"); //设置一个线程
        System.setProperties(propes);
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 1000; i++) { //创建一千生产者
            Mailbox<Integer> mb = new Mailbox<Integer>(1, 10);
            new Producer(i, mb).start();
            mailMap.put(i, mb);
        }

        for (int i = 0; i < 1000; i++) { //创建一千个消费者
            new Consumer(mailMap.get(i)).start();
        }

        Task.idledown(); //开始运行

        long endTime = System.currentTimeMillis();

        System.out.println( Thread.currentThread().getName() + "总计花费时长: " + (endTime- startTime));
    }

}
```

//生产者

```
public class Producer extends Task<Object> {
```

```
    Integer count = null;
```

```
    Mailbox<Integer> mb = null;
```

```
    public Producer(Integer count, Mailbox<Integer> mb) {
```

```
        this.count = count;
```

```
        this.mb = mb;
```

```
    }
```

```
    public void execute() throws Pausable {
```

```
        count = count*10;
```

```
        for (int i = 0; i < 10; i++) {
```

```
            mb.put(count);//当空间不足时，阻塞协程线程
```

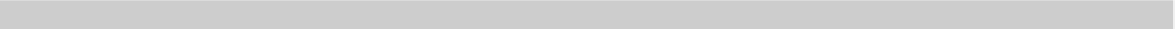
```
            System.out.println(Thread.currentThread().getName() + "生产者生产，目前总共有" + mb.size() + "生产了：" + c
```

```
            count++;
```

```
        }
```

```
    }
```

```
}
```

<  >

```
//消费者
public class Consumer extends Task<Object> {

    Mailbox<Integer> mb = null;

    public Consumer(Mailbox<Integer> mb) {
        this.mb = mb;
    }

    /**
     * 执行
     */
    public void execute() throws Pausable {
        Integer c = null;
        for (int i = 0; i < 10000; i++) {
            c = mb.get();//获取消息，阻塞协程线程

            if (c == null) {
                System.out.println("计数");
            }else {
                System.out.println(Thread.currentThread().getName() + "消费者消费，目前总共有" + mb.size() + "消费了：" + c);
                c = null;
            }
        }
    }
}
```

在这个案例中，我创建了1000个生产者和1000个消费者，每个生产者生产10个产品，1000个消费者同时消费产品。我们可以看到两个例子运行的结果如下：

多线程执行时长：2761

协程执行时长：1050

通过上述性能对比，我们可以发现：在有严重阻塞的场景下，协程的性能更胜一筹。其实，I/O

阻塞型场景也就是协程在Java中的主要应用。

总结

协程和线程密切相关，协程可以认为是运行在线程上的代码块，协程提供的挂起操作会使协程暂停执行，而不会导致线程阻塞。

协程又是一种轻量级资源，即使创建了上千个协程，对于系统来说也不是很大的负担，但如果在程序中创建上千个线程，那系统可真就压力山大了。可以说，协程的设计方式极大地提高了线程的使用率。

通过今天的学习，当其他人侃侃而谈Go语言在网络编程中的优势时，相信你不会一头雾水。学习Java的我们也不要觉得，协程离我们很遥远了。协程是一种设计思想，不仅仅局限于某一语言，况且Java已经可以借助协程框架实现协程了。

但话说回来，协程还是在Go语言中的应用较为成熟，在Java中的协程目前还不是很稳定，重点是缺乏大型项目的验证，可以说Java的协程设计还有很长的路要走。

思考题

在Java中，除了Kilim框架，你知道还有其它协程框架也可以帮助Java实现协程吗？你使用过吗？

期待在留言区看到你的见解。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起讨论。



Java 性能调优实战

覆盖 80% 以上 Java 应用调优场景

刘超

金山软件西山居技术经理



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有现金奖励。



周星星

1

老师有几个疑问

1. 协程必须手动调用等待或阻塞才能被安排到等待队列去吗？还是说协程也可以跟线程一样会被随机丢到等待队列去每个协程也有个运行时间片？如果可以随机一般是如何实现的？
2. 协程之间的争抢基于什么实现的？我想的话可以使用CAS来实现没有抢到的再次被丢到等待队列不知道对不对。
3. 我看例子上邮箱里面没有数据时消费者协程没有类似线程的等待机制，这个要如何写呢？

2019-07-02



QQ怪

1

老师讲协程讲的最深最易懂的一个，赞赞赞

2019-07-02



宝玉

0

有没有具体场景的实战优化呢？

2019-07-06

作者回复

在socket通信中，可以使用协程优化IO读写这块。例如，可以尝试用协程写个非阻塞Socket通信，或用协程简单改造下Netty。具体的实现自己可以尝试，如果遇到问题欢迎沟通。

2019-07-07



小橙橙

0

老师，以后的JAVA版本是不是也会自带协程功能？

2019-07-05

作者回复

是的，Java未来的三个主要项目之一Loom项目 引入了被称为 **fibers** 的新型轻量级用户线程，甲骨文公司 Loom 项目技术负责人 Ron Pressler 在 QCon 伦敦 2019 大会上指出：“利用 **fibers**，如果我们确保其轻量化程度高于内核提供的线程，那么问题就得到了解决。大家将能够尽可能多地使用这些用户模式下的轻量级线程，且基本不会出现任何阻塞问题”。

具体的可以阅读以下openjdk官网链接：

<https://cr.openjdk.java.net/~rpressler/loom/Loom-Proposal.html>

2019-07-07



Liam

0

消费者从mailbox拿数据为空时，或生产者往mailbox写数据没有可用空间时，不会阻塞吗？类似于队列

2019-07-03

作者回复

会的，在mailbox为空时，消费者get会被阻塞；在mailbox满了，生产者put会阻塞。

2019-07-04



cricket1981

👍 0

AKKA Framework算得上一个，另外还有Vert.X

2019-07-02



-W.LI-

👍 0

老师好!1:1, N:1, N:M的线程模型。总感觉上学的时候有讲可是又想不起来。谢谢老师的讲解。不过还是有些不明白的地方。

Java是采用1:1的映射方式。一个Java线程就对应一个内核线程。用户态和内核态的切换和这个映射模型有关系么(用户态和内核态，和用户线程内核线程是否有关系)?

从用户态切换为内核态的时候具体做了哪些操作?之前讲IO模型时老师讲了，用户空间和内核空间，多次数据拷贝。和用户线程内核线程有什么联系么?后面会讲么?

2019-07-02



周星星

👍 0

有个疑问，线程的切换是由系统来保证的，那协程之间的切换也能跟线程那样由调度器保证，还是说协程必须要手动调用或io

2019-07-02

作者回复

由调度器保证

2019-07-04



风翱

👍 0

一核的CPU是否有使用协程的必要? 按目前cpu的调度是采用时间片的方式，一核的CPU也存在上下文切换。一核的CPU也可以应用到协程带来的好处。不知道这个理解是否正确?

2019-07-02

作者回复

对的

2019-07-04



crazypokerk

👍 0

老师，不是建议不要使用String对象作为锁对象吗?为什么上面的代码private static String LOCK = "lock", 要这么写呢?

2019-07-02

作者回复

没有其他成员变量引用"lock", 这里使用没有问题，具体场景具体分析。

2019-07-04



Jxin

👍 0

loom项目也可以让javaer玩玩协程，不过仅限于玩玩。总归没有大项目验证，而且都非官方版，而是以框架的形式引用。不建议在真实项目中去使用。另外erlang的并发编程并不弱于go。就目前来看，感觉还是java比较适合写web项目。虽然go写并发编程很爽，但web开发组件不健全，很多东西得自己实现，而自己实现意味着成本和风险。能有现有，经过大项目和时间验

证的组件终是比较让人舒心的。

2019-07-02

作者回复

对的，一些技术框架还是需要经受大公司或者一些大型项目的验证，不过我们也可以先在一些小型项目中先行使用。

2019-07-02



QQ怪

0

听过quasar，好像这个早就有了，但不知道为啥没火起来

2019-07-02



明翼

0

除了内存模型还有线程模型，这种有虚拟机的语言是不是都有这种对应关系啊。协程用的不多，以前想用kilim但是有点麻烦，就没用了

2019-07-02



nightmare

0

终于看懂协程和线程的区别了，协程基于N: M的线程模型实现，M个协程被调度器调度，实际上也是被内核线程执行，不过由于需要的内核线程少，一个内核线程可以执行多个协程，占用资源少，而且上下文切换也更加少，而基于线程的1: 1模型只有有阻塞就会有上下文切换

2019-07-02



听雨

0

老师，读了今天的内容，我理解的意思是：

1.因为每个轻量级线程都有一个内核线程支持，而java中，每个用户线程对应一个轻量级线程，可以看作用户线程和支持轻量级线程的内核线程是一一对一的，所以就说java线程模型是用户线程和内核线程一对一。

2.那这里轻量级线程属于内核线程吗，我看文中说的是由内核线程clone而来的，那它算内核线程吗？

请老师解答一下！

2019-07-02

作者回复

1、对的

2、属于用户线程，与内核线程一对一映射

2019-07-02



Liam

0

请问mailbox是如何实现阻塞的呢？

2019-07-02

作者回复

mailbox不会阻塞，这是一个信箱，协程之间通过mailbox来分享共享变量

2019-07-02



沧颜

0



kotlin的协程设计应该和这个框架差不多吧

2019-07-02

作者回复

kotlin也很多人使用

2019-07-02