李留广

< 2019年9月 >

日	一	二	三	四	五	六
1	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	1	2	3	4	5
6	7	8	9	10	11	12

昵称：李留广
园龄：2年6个月
粉丝：9
关注：119
+加关注

搜索

找找看
谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签

PHP(41)
Redis(34)
MySQL(32)
面试(7)
生活杂谈(6)
中间件(6)
微服务(5)
nginx(5)
闭包(5)
ddos(4)
更多

什么是协程

先搞清楚，什么是协程。

你可能已经听过『进程』和『线程』这两个概念。

进程就是二进制可执行文件在计算机内存里的一个运行实例，就好比你的.exe文件是个类，进程就是new出来的那个实例。

进程是计算机系统进行资源分配和调度的基本单位（调度单位这里别纠结线程进程的），每个CPU下同一时刻只能处理一个进程。

所谓的并行，只不过是看起来并行，CPU事实上在用很快的速度切换不同的进程。

进程的切换需要进行系统调用，CPU要保存当前进程的各个信息，同时还会使CPUCache被废掉。

所以进程切换不到非不得已就不做。

那么怎么实现『进程切换不到非不得已就不做』呢?

首先进程被切换的条件是：进程执行完毕、分配给进程的CPU时间片结束，系统发生中断需要处理，或者进程等待必要的资源（进程阻塞）等。你想下，前面几种情况自然没有什么话可说，但是如果是在阻塞等待，是不是就浪费了。

其实阻塞的话我们的程序还有其他可执行的地方可以执行，不一定要傻傻的等!

所以就有了线程。

线程简单理解就是一个『微进程』，专门跑一个函数（逻辑流）。

所以我们就可以在编写程序的过程中将可以同时运行的函数用线程来体现了。

线程有两种类型，一种是由内核来管理和调度。

我们说，只要涉及需要内核参与管理调度的，代价都是很大的。这种线程其实也就解决了当一个进程中，某个正在执行的线程遇到阻塞，我们可以调度另外一个可运行的线程来跑，但是还是在同一个进程里，所以没有了进程切换。

还有另外一种线程，他的调度是由程序员自己写程序来管理的，对内核来说不可见。这种线程叫做『用户空间线程』。

协程可以理解就是一种用户空间线程。

协程，有几个特点：

随笔档案

2019年9月(1)
2019年8月(5)
2019年7月(9)
2019年6月(11)
2019年5月(29)
2019年4月(21)
2019年3月(15)
2019年2月(31)
2019年1月(6)
2018年12月(15)
2018年11月(22)
2018年10月(12)
2018年9月(16)
2018年8月(18)
2018年7月(22)
2018年6月(9)
2018年4月(15)
2018年1月(1)
2017年11月(5)
2017年10月(7)
2014年7月(1)

最新评论

1. Re:Redis这篇就够了
ZSET和SET的图是不是放重复了啊
--王章章章
2. Re:程序员找工作那些事（二）海投面霸
为什么我的源站一直报异常?
--自由飞

阅读排行榜

1. 理解 PHP 依赖注入(6751)
2. redis该怎么用(4042)
3. PHP常用的三种设计模式(3541)
4. 什么是协程(3437)
5. MySQL百万级、千万级数据多表关联SQL语句调优(2831)

评论排行榜

1. 程序员找工作那些事（二）海投面霸(1)
2. Redis这篇就够了(1)

推荐排行榜

- 协同，因为是由程序员自己写的调度策略，其通过协作而不是抢占来进行切换
- 在用户态完成创建，切换和销毁
- ***⚠ 从编程角度上看，协程的思想本质上就是控制流的主动让出（yield）和恢复（resume）机制***
- generator经常用来实现协程

说到这里，你应该明白协程的基本概念了吧?

PHP实现协程

一步一步来，从解释概念说起!

可迭代对象

PHP5提供了一种定义对象的方法使其可以通过单元列表来遍历，例如用foreach语句。

你如果要实现一个可迭代对象，你就要实现Iterator接口:

```
<?php
class MyIterator implements Iterator
{
    private $var = array();

    public function __construct($array)
    {
        if (is_array($array)) {
            $this->var = $array;
        }
    }

    public function rewind() {
        echo "rewinding\n";
        reset($this->var);
    }

    public function current() {
        $var = current($this->var);
        echo "current: $var\n";
        return $var;
    }

    public function key() {
        $var = key($this->var);
        echo "key: $var\n";
        return $var;
    }

    public function next() {
        $var = next($this->var);
        echo "next: $var\n";
        return $var;
    }
```

```
    public function valid() {
        $var = $this->current() !== false;
        echo "valid: {$var}\n";
        return $var;
    }
}

$values = array(1,2,3);
$it = new MyIterator($values);

foreach ($it as $a => $b) {
    print "$a: $b\n";
}
```

生成器

可以说之前为了拥有一个能够被foreach遍历的对象，你不得不去实现一堆的方法，yield关键字就是为了简化这个过程。

生成器提供了一种更容易的方法来实现简单的对象迭代，相比较定义类实现Iterator接口的方式，性能开销和复杂性大大降低。

```
<?php
function xrange($start, $end, $step = 1) {
    for ($i = $start; $i <= $end; $i += $step) {
        yield $i;
    }
}

foreach (xrange(1, 1000000) as $num) {
    echo $num, "\n";
}
```

记住，一个函数中如果用了yield，他就是一个生成器，直接调用他是没有用的，不能等同于一个函数那样去执行！

所以，yield就是yield，下次谁再说yield是协程，我肯定把你xxxx。

PHP协程

前面介绍协程的时候说了，协程需要程序员自己去编写调度机制，下面我们来看这个机制怎么写。

0）生成器正确使用

既然生成器不能像函数一样直接调用，那么怎么才能调用呢?

方法如下:

1. foreach他
2. send($value)
3. current / next...

1）Task实现

Task就是一个任务的抽象，刚刚我们说了协程就是用户空间线程，线程可以理解就是跑一个函数。

所以Task的构造函数中就是接收一个闭包函数，我们命名为coroutine。

```
/**
 * Task任务类
 */
class Task
{
    protected $taskId;
    protected $coroutine;
    protected $beforeFirstYield = true;
    protected $sendValue;

    /**
     * Task constructor.
     * @param $taskId
     * @param Generator $coroutine
     */
    public function __construct($taskId, Generator $coroutine)
    {
        $this->taskId = $taskId;
        $this->coroutine = $coroutine;
    }

    /**
     * 获取当前的Task的ID
     *
     * @return mixed
     */
    public function getTaskId()
    {
        return $this->taskId;
    }

    /**
     * 判断Task执行完毕了没有
     *
     * @return bool
     */
    public function isFinished()
    {
        return !$this->coroutine->valid();
    }

    /**
     * 设置下次要传给协程的值，比如 $id = (yield $xxxx)，这个值就给了$id了
     *
     * @param $value
     */
    public function setSendValue($value)
```

```
    {
        $this->sendValue = $value;
    }

    /**
     * 运行任务
     *
     * @return mixed
     */
    public function run()
    {
        // 这里要注意，生成器的开始会reset，所以第一个值要用
current获取
        if ($this->beforeFirstYield) {
            $this->beforeFirstYield = false;
            return $this->coroutine->current();
        } else {
            // 我们说过了，用send去调用一个生成器
            $retval = $this->coroutine->send($this->sendValue);
            $this->sendValue = null;
            return $retval;
        }
    }
}
```

2）Scheduler实现

接下来就是Scheduler这个重点核心部分，他扮演着调度员的角色。

```
/**
 * Class Scheduler
 */
Class Scheduler
{
    /**
     * @var SplQueue
     */
    protected $taskQueue;
    /**
     * @var int
     */
    protected $tid = 0;

    /**
     * Scheduler constructor.
     */
    public function __construct()
    {
        /* 原理就是维护了一个队列，
         * 前面说过，从编程角度上看，协程的思想本质上就是控制流的
主动让出（yield）和恢复（resume）机制
         * */
        $this->taskQueue = new SplQueue();
```

```
    }

    /**
     * 增加一个任务
     *
     * @param Generator $task
     * @return int
     */
    public function addTask(Generator $task)
    {
        $tid = $this->tid;
        $task = new Task($tid, $task);
        $this->taskQueue->enqueue($task);
        $this->tid++;
        return $tid;
    }

    /**
     * 把任务进入队列
     *
     * @param Task $task
     */
    public function schedule(Task $task)
    {
        $this->taskQueue->enqueue($task);
    }

    /**
     * 运行调度器
     */
    public function run()
    {
        while (!$this->taskQueue->isEmpty()) {
            // 任务出队
            $task = $this->taskQueue->dequeue();
            $res = $task->run(); // 运行任务直到 yield

            if (!$task->isFinished()) {
                $this->schedule($task); // 任务如果还没完全执行完毕，入队等下次执行
            }
        }
    }
}
```

这样我们基本就实现了一个协程调度器。

你可以使用下面的代码来测试：

```
<?php
function task1() {
    for ($i = 1; $i <= 10; ++$i) {
        echo "This is task 1 iteration $i.\n";
```

```
        yield; // 主动让出CPU的执行权
    }
}

function task2() {
    for ($i = 1; $i <= 5; ++$i) {
        echo "This is task 2 iteration $i.\n";
        yield; // 主动让出CPU的执行权
    }
}

$scheduler = new Scheduler; // 实例化一个调度器
$scheduler->addTask(task1()); // 添加不同的闭包函数作为任务
$scheduler->addTask(task2());
$scheduler->run();
```

关键说下在哪里能用得到PHP协程。

```
function task1() {
        /* 这里有一个远程任务，需要耗时10s，可能是一个远程机器抓取分析远程网址的任务，我们只要提交最后去远程机器拿结果就行了 */
        remote_task_commit();
        // 这时候请求发出后，我们不要在这里等，主动让出CPU的执行权给task2运行，他不依赖这个结果
        yield;
        yield (remote_task_receive());
        ...
}

function task2() {
    for ($i = 1; $i <= 5; ++$i) {
        echo "This is task 2 iteration $i.\n";
        yield; // 主动让出CPU的执行权
    }
}
```

这样就提高了程序的执行效率。

关于『系统调用』的实现，鸟哥已经讲得很明白，我这里不再说明。

3）协程堆栈

鸟哥文中还有一个协程堆栈的例子。

我们上面说过了，如果在函数中使用了yield，就不能当做函数使用。

所以你在一个协程函数中嵌套另外一个协程函数：

```
<?php
function echoTimes($msg, $max) {
    for ($i = 1; $i <= $max; ++$i) {
        echo "$msg iteration $i\n";
        yield;
    }
}
```

```
function task() {
    echoTimes('foo', 10); // print foo ten times
    echo "---\n";
    echoTimes('bar', 5); // print bar five times
    yield; // force it to be a coroutine
}

$scheduler = new Scheduler;
$scheduler->addTask(task());
$scheduler->run();
```

这里的echoTimes是执行不了的！所以就需要协程堆栈。

不过没关系，我们改一改我们刚刚的代码。

把Task中的初始化方法改下，因为我们在运行一个Task的时候，我们要分析出他包含了哪些子协程，然后将子协程用一个堆栈保存。（C语言学的好的同学自然能理解这里，不理解的同学我建议去了解下进程的内存模型是怎么处理函数调用）

```
 /**
     * Task constructor.
     * @param $taskId
     * @param Generator $coroutine
     */
    public function __construct($taskId, Generator $coroutine)
    {
        $this->taskId = $taskId;
        // $this->coroutine = $coroutine;
        // 换成这个，实际Task->run的就是stackedCoroutine这个函数，不是$coroutine保存的闭包函数了
        $this->coroutine = stackedCoroutine($coroutine);
    }
```

当Task->run()的时候，一个循环来分析：

```
/**
 * @param Generator $gen
 */
function stackedCoroutine(Generator $gen)
{
    $stack = new SplStack;

    // 不断遍历这个传进来的生成器
    for (; ;) {
        // $gen可以理解为指向当前运行的协程闭包函数（生成器）
        $value = $gen->current(); // 获取中断点，也就是yield出来的值

        if ($value instanceof Generator) {
            // 如果是也是一个生成器，这就是子协程了，把当前运行的协程入栈保存
            $stack->push($gen);
            $gen = $value; // 把子协程函数给gen，继续执行，注意接下来就是执行子协程的流程了
```

```
            continue;
        }

        // 我们对子协程返回的结果做了封装，下面讲
        $isReturnValue = $value instanceof CoroutineReturnValue;
// 子协程返回`$value`需要主协程帮忙处理

        if (!$gen->valid() || $isReturnValue) {
            if ($stack->isEmpty()) {
                return;
            }
            // 如果是gen已经执行完毕，或者遇到子协程需要返回值给
主协程去处理
            $gen = $stack->pop(); //出栈，得到之前入栈保存的主协
程
            $gen->send($isReturnValue ? $value->getValue() :
NULL); // 调用主协程处理子协程的输出值
            continue;
        }

        $gen->send(yield $gen->key() => $value); // 继续执行子协
程
    }
}
```

然后我们增加echoTime的结束标示:

```
class CoroutineReturnValue {
    protected $value;

    public function __construct($value) {
        $this->value = $value;
    }

    // 获取能把子协程的输出值给主协程，作为主协程的send参数
    public function getValue() {
        return $this->value;
    }
}

function retval($value) {
    return new CoroutineReturnValue($value);
}
```

然后修改echoTimes:

```
function echoTimes($msg, $max) {
    for ($i = 1; $i <= $max; ++$i) {
        echo "$msg iteration $i\n";
        yield;
    }
    yield retval("");  // 增加这个作为结束标示
}
```

Task变为：

```
function task1()
{
    yield echoTimes('bar', 5);
}
```

这样就实现了一个协程堆栈，现在你可以举一反三了。

4）PHP7中yield from关键字

PHP7中增加了yield from，所以我们不需要自己实现携程堆栈，真是太好了。

把Task的构造函数改回去：

```
    public function __construct($taskId, Generator $coroutine)
    {
        $this->taskId = $taskId;
        $this->coroutine = $coroutine;
        // $this->coroutine = stackedCoroutine($coroutine); //不需要自己实现了，改回之前的
    }
```

echoTimes函数：

```
function echoTimes($msg, $max) {
    for ($i = 1; $i <= $max; ++$i) {
        echo "$msg iteration $i\n";
        yield;
    }
}
```

task1生成器：

```
function task1()
{
    yield from echoTimes('bar', 5);
}
```

这样，轻松调用子协程。

标签：PHP

« 上一篇： PHP 使用协同程序实现合作多任务
» 下一篇： 有点忙啊

posted on 2018-04-12 15:23 李留广 阅读(3437) 评论(0) 编辑 收藏

刷新评论 刷新页面 返回顶部

相关博文：

· Linux性能优化实战：CPU的上下文切换是什么意思（03）
· Python并发编程系列之常用概念剖析：并行串行并发同步异步阻塞非阻塞进...
· 深入理解yield(二)：yield与协程
· C#并行编程（1）：理解并行
· 关于go中并发的初步理解

最新 IT 新闻:

· 印度小哥被乌鸦追杀三年，起因只是一场误会?
· GNOME 3.36 稳定版定于明年三月发布
· End Software Patents，反对软件专利，你觉得怎么样?
· 首例基因编辑干细胞治疗艾滋病：北大邓宏魁参与，达到最佳治疗效果
· 天文学家使用望远镜捕获迄今为止最清晰的星际彗星的彩色图像
» 更多新闻...