
Table of Contents

简介	1.1
前言	1.2
说明	1.2.1
第一部分: 半协程调度器	1.3
统一生成器接口	1.3.1
生成器迭代	1.3.2
生成器返回值	1.3.3
生成器委托	1.3.4
改写return	1.3.5
抽象异步模型	1.3.6
引入异常处理	1.3.7
异常: 嵌套任务透传	1.3.8
异常: 传递流程	1.3.9
异常: 重新进行CPS变换	1.3.10
异常: 重新加入Async	1.3.11
Syscall与Context	1.3.12
调度器: 里程碑	1.3.13
spawn	1.3.14
callcc	1.3.15
race与timeout	1.3.16
all与parallel	1.3.17
channel与协程间通信	1.3.18
无缓存channel	1.3.19
缓存channel	1.3.20

channel演示	1.3.21
FutureTask与fork	1.3.22
第二部分: Koa	1.4
穿越地心之旅	1.4.1
洋葱圈模型	1.4.2
rightReduce与中间件compose	1.4.3
Koa::Application	1.4.4
Koa::Context	1.4.5
Koa::Request	1.4.6
Koa::Response	1.4.7
Koa - HelloWorld	1.4.8
Middleware Interface	1.4.9
Middleware: 全局异常处理	1.4.10
Middleware: Router	1.4.11
Middleware: 请求超时	1.4.12
一个综合示例	1.4.13
附录	1.5
参考	1.6

PHP异步编程: 手把手教你实现co与Koa

- 前言
 - 说明
- 第一部分: 半协程调度器
 - 统一生成器接口
 - 生成器迭代
 - 生成器返回值
 - 生成器委托
 - 改写return
 - 抽象异步模型
 - 引入异常处理
 - 异常: 嵌套任务透传
 - 异常: 传递流程
 - 异常: 重新进行CPS变换
 - 异常: 重新加入Async
 - Syscall与Context
 - 调度器: 里程碑
 - spawn
 - callcc
 - race与timeout
 - all与parallel
 - channel与协程间通信
 - 无缓存channel
 - 缓存channel
 - channel演示
 - FutureTask与fork
- 第二部分: Koa
 - 穿越地心之旅
 - 洋葱圈模型
 - rightReduce与中间件compose
 - Koa::Application

简介

- [Koa::Context](#)
- [Koa::Request](#)
- [Koa::Response](#)
- [Koa - HelloWorld](#)
- [Middleware Interface](#)
- [Middleware: 全局异常处理](#)
- [Middleware: Router](#)
- [Middleware: 请求超时](#)
- [一个综合示例](#)
- [附录](#)
- [参考](#)

前言

近年来，在面向高并发编程的道路上，Node.js与Golang风生水起，让人们渐渐把目光从多线程模型转移到callback与CSP/Actor上，用惯了FPM多进程同步阻塞模型的PHPer中总难免有心动。多种EventLoop一直不温不火，而国内以swoole为代表，直接以扩展形式，提供了整套callback模型的PHP异步编程解决方案，正在逐渐的流行起来。

Node.js在JS上开花结果，也许是浏览器的DOM事件模型培养起来的callback书写习惯，与语言自身的函数式特性适合callback代码编写。但回调固有的逻辑割裂、调试维护难的问题随着node社区的繁荣逐渐显现，从老赵脑洞大开的windjs到co与Promise，方案层出不穷，最终Promise被采纳为官方「异步编程标准规范」，从C#借鉴过来的async/await被纳入语言标准。

因swoole与Node.js的I/O模型相同，PHPer有幸在高并发命题上遭遇与node一样的问题。Closure([RFC](#))一定程度从语言本身改善了异步编程的体验，受限于Zend引擎作用域实现机制，PHP因缺失词法作用域从而缺失词法闭包，Closure对象采用了use语法来显式捕获upValue到静态属性的方式(closure->func.op_array.static_variables)，我个人认为这有点像无法自动实现闭包的匿名函数。之后Nikita Popov在PHP中实现了Generator([RFC](#))，并且让PHPer意识到生成器原来可以实现实现非抢占任务调度([译文:在PHP中使用协程实现多任务调度](#))。我们最终可以借助于生成器实现半协程来解决该问题。

这篇文章秉承着造轮子的精神，我们从头实现一个全功能的基于生成器(Generator)的半协程调度器与相关基础组件，并基于该调度器实(chao)现(xi)JS社区当红的koa框架，最终加深我们对异步编程的理解。

说明

1. 下文中协程均指代使用生成器实现的半协程，具体概念参见[Wiki](#)。
2. 下文中耗时任务指代I/O或定时器，非CPU计算。
3. 广告 继TSF之后，我司去年了开源[Zan Framework](#)，内部的半协程调度器已经解决了swoole中回调接口的代码书写问题。
4. 下文实例代码，限于篇幅，每部分仅呈现改动部分，其余省略。

第一部分: 半协程调度器

谈及koa(1.x)首先得说co, co与Promise是JSer在解决回调地狱(callback hell)问题前仆后继的众多产物之一。

co其实是Generator的自动执行器(半协程调度器): 通过yield显式操纵控制流让我们可以做到以近乎同步的方式书写非阻塞代码。

Promise是一套比较完善的方案, 但关于如何实现Promise本身超出本文范畴, 且PHP没有大量异步接口的历史包袱需要thunks方案做转换。

综上所述, 我们的调度器仅基于一个简单的接口, 来抽象异步任务:

```
<?php
interface Async
{
    /**
     * 开启异步任务, 完成是执行回调, 任务结果或异常通过回调参数传递
     * @param callable $callback
     *          continuation :: (mixed $result = null, \Exception
     |null $ex = null)
     * @return void
     */
    public function begin(callable $callback);
}
```

1. 对co库不了解的同学可以先参考[阮一峰 - co 函数库的含义和用法](#)
2. co新版与旧版的区别在于对thunks的支持, 4.x只支持Promises.

我们首先构建Koa的基础设施, 渐进的实现一个约50+行代码的精练的半协程调度器。

统一生成器接口

由于内部隐式rewind，需要先调用 Generator::current 获取当前value，而直接调用 Generator::send 会跳到第二次yield。

1. send方法参考
2. 生成器参考

```
<?php

class Gen
{
    public $isfirst = true;
    public $generator;

    public function __construct(\Generator $generator)
    {
        $this->generator = $generator;
    }

    public function valid()
    {
        return $this->generator->valid();
    }

    public function send($value = null)
    {
        if ($this->isfirst) {
            $this->isfirst = false;
            return $this->generator->current();
        } else {
            return $this->generator->send($value);
        }
    }
}
```


生成器迭代

手动迭代生成器，递归执行 `AsyncTask::next`，调用 `Generator::send` 方法将将 `yield` 值作为 `yield` 表达式结果。

1. `yield` 表达式可能是一个异步调用，我们这里为之后把异步调用的结果作为 `yield` 表达式结果铺垫。
2. `yield` 外侧括号在 PHP5 必须，PHP7 不需要。

如，
 `$ip = (yield async_dns_lookup(...))`;
 ^ |
 | yield值
 | |
 yield表达式结果 yield 表达式

```
<?php

final class AsyncTask
{
    public $gen;

    public function __construct(\Generator $gen)
    {
        $this->gen = new Gen($gen);
    }

    public function begin()
    {
        $this->next();
    }

    public function next($result = null)
    {
        $value = $this->gen->send($result);
        if ($this->gen->valid()) {
```

生成器迭代

```
    $this->next($value);
}
}
```

```
<?php
function newGen()
{
    $r1 = (yield 1);
    $r2 = (yield 2);
    echo $r1, $r2;
}
$task = new AsyncTask(newGen());
$task->begin() // output: 12
```

生成器返回值

PHP7支持通过 `Generator::getReturn` 获取生成器方法return的返回值。

PHP5中我们约定使用Generator最后一次yield值作为返回值。

```
<?php

final class AsyncTask
{
    public function begin()
    {
        return $this->next();
    }

    // 添加return传递每一次迭代的结果，直到向上传递到begin
    public function next($result = null)
    {
        $value = $this->gen->send($result);

        if ($this->gen->valid()) {
            return $this->next($value);
        } else {
            return $result;
        }
    }
}
```

```
<?php
function newGen()
{
    $r1 = (yield 1);
    $r2 = (yield 2);
    echo $r1, $r2;
    yield 3;
```

生成器返回值

```
}

$task = new AsyncTask(newGen());
$r = $task->begin(); // output: 12
echo $r; // output: 3
```

生成器委托

PHP7中支持 `delegating generator`，可以自动展开 `subgenerator`；

A “subgenerator” is a Generator used in the portion of the `yield` from syntax.

我们需要在PHP5支持子生成器，将子生成器最后`yield`值作为父生成器`yield`表达式结果，仅只需要加两行代码，递归的产生一个 `AsyncTask` 对象来执行子生成器即可。

```
<?php

final class AsyncTask
{
    public function next($result = null)
    {
        $value = $this->gen->send($result);

        if ($this->gen->valid()) {
            if ($value instanceof \Generator) {
                $value = (new self($value))->begin();
            }
            return $this->next($value);
        } else {
            return $result;
        }
    }
}
```

```
<?php
function newSubGen()
{
    yield 0;
```

生成器委托

```
yield 1;  
}  
  
function newGen()  
{  
    $r1 = (yield newSubGen());  
    $r2 = (yield 2);  
    echo $r1, $r2;  
    yield 3;  
}  
$task = new AsyncTask(newGen());  
$r = $task->begin(); // output: 12  
echo $r; // output: 3
```

改写return

return其实可以被替换为单参数且永远不返回的函数，将return解糖进行CPS变换，改写为函数参数continuation，将迭代result作为回调参数返回，为引入异步迭代做准备。

```
<?php

final class AsyncTask
{
    public $continuation;

    public function begin(callable $continuation)
    {
        $this->continuation = $continuation;
        $this->next();
    }

    public function next($result = null)
    {
        $value = $this->gen->send($result);

        if ($this->gen->valid()) {
            if ($value instanceof \Generator) {
                // 父任务next方法是子任务的延续,
                // 子任务迭代完成后继续完成父任务迭代
                $continuation = [$this, "next"];
                (new self($value))->begin($continuation);
            } else {
                $this->next($value);
            }
        } else {
            $cc = $this->continuation;
            $cc($result);
        }
    }
}
```

改写return

```
    }  
}
```

```
<?php  
function newGen()  
{  
    $r1 = (yield newSubGen());  
    $r2 = (yield 2);  
    echo $r1, $r2;  
    yield 3;  
}  
$task = new AsyncTask(newGen());  
  
$trace = function($r) { echo $r; };  
$task->begin($trace); // output: 123
```

抽象异步模型

对回调模型抽象出异步接口 `Async`。

只有一个方法的接口通常都可以使用闭包代替，区别在于`interface`引入新类型，闭包则不会。如果说`thunkify`依赖了参数顺序的弱约定，`Async`相对严肃的依赖了类型。

```
<?php

interface Async
{
    public function begin(callable $callback);
}

// AsyncTask符合Async定义， 实现Async
final class AsyncTask implements Async
{
    public function next($result = null)
    {
        $value = $this->gen->send($result);

        if ($this->gen->valid()) {
            // \Generator -> Async
            if ($value instanceof \Generator) {
                $value = new self($value);
            }

            if ($value instanceof Async) {
                $async = $value;
                $continuation = [$this, "next"];
                $async->begin($continuation);
            } else {
                $this->next($value);
            }
        }
    }
}
```

```
    } else {
        $cc = $this->continuation;
        $cc($result);
    }
}
```

两个简单的对回调接口转换例子：

```
<?php

// 定时器修改为标准异步接口
class AsyncSleep implements Async
{
    public function begin(callable $cc)
    {
        swoole_timer_after(1000, $cc);
    }
}

// 异步dns查询修改为标准异步接口
class AsyncDns implements Async
{
    public function begin(callable $cc)
    {
        swoole_async_dns_lookup("www.baidu.com", function($host, $ip) use($cc) {
            // 这里我们会发现， 通过call $cc， 将返回值作为参数进行传递， 与callcc相像
            // $ip 通过$cc 从子生成器传入父生成器， 最终通过send方法成为yield表达式结果
            $cc($ip);
        });
    }
}

function newGen()
```

```
{  
    $r1 = (yield newSubGen());  
    $r2 = (yield 2);  
    $start = time();  
    yield new AsyncSleep();  
    echo time() - $start, "\n";  
    $ip = (yield new AsyncDns());  
    yield "IP: $ip";  
}  
$task = new AsyncTask(newGen());  
  
$trace = function($r) { echo $r; };  
$task->begin($trace);  
// output:  
// 1  
// IP: 115.239.210.27
```

引入异常处理

尽管只有寥寥几行代码，我们却已经实现了可工作的半协程调度器(缺失异常处理)。

没关系，下面先rollback回return的实现，开始引入异常处理，目标是在嵌套生成器之间正确向上抛出异常，跨生成器捕获异常。

```
<?php

// 为Gen引入throw方法
class Gen
{
    // PHP7 之前 关键词不能用作名字
    public function throw_(\Exception $ex)
    {
        return $this->generator->throw($ex);
    }
}

final class AsyncTask
{
    public function begin()
    {
        return $this->next();
    }

    // 这里添加第二个参数， 用来在迭代过程传递异常
    public function next($result = null, \Exception $ex = null)
    {
        if ($ex) {
            $this->gen->throw_($ex);
        }

        $ex = null;
    }
}
```

引入异常处理

```
try {
    // send方法内部是一个resume的过程:
    // 恢复execute_data上下文， 调用zend_execute_ex()继续
    // 执行,
    // 后续中op_array内可能会抛出异常
    $value = $this->gen->send($result);
} catch (\Exception $ex) {}

if ($ex) {
    if ($this->gen->valid()) {
        // 传递异常
        return $this->next(null, $ex);
    } else {
        throw $ex;
    }
} else {
    if ($this->gen->valid()) {
        // 正常yield值
        return $this->next($value);
    } else {
        return $result;
    }
}
}
```

```
<?php
function newGen()
{
    $r1 = (yield 1);
    throw new \Exception("e");
    $r2 = (yield 2);
    yield 3;
}
$task = new AsyncTask(newGen());
try {
```

引入异常处理

```
$r = $task->begin();
echo $r;
} catch (\Exception $ex) {
    echo $ex->getMessage(); // output: e
}
```

异常: 嵌套任务透传

重新处理生成器嵌套，需要将子生成器异常抛向父生成器。

当生成器迭代过程发生未捕获异常，生成器将会被关闭，`Generator::valid` 返回 `false`，未捕获异常会从生成器内部被抛向父作用域，嵌套子生成器内部的未捕获异常必须最终被抛向根生成器的 calling frame，PHP7 中 `yield-from` 对嵌套子生成器 `resume` 时产生的异常，采取 `goto try_again` 传递 + `while` 方式层层向上抛出，我们的代码因为递归迭代的原因，未捕获异常需要逆递归栈帧方向层层上抛，性能方面有改进余地。

```
<?php

final class AsyncTask
{
    public function next($result = null, \Exception $ex = null)
    {
        try {
            if ($ex) {
                // c. 直接抛出异常
                // $ex来自子生成器，调用父生成器throw抛出
                // 这里实现了 try { yield \Generator; } catch(\Exception $ex) { }
                // echo "c -> ";
                $value = $this->gen->throw_($ex);
            } else {
                // a2. 当前生成器可能抛出异常
                // echo "a2 -> ";
                $value = $this->gen->send($result);
            }

            if ($this->gen->valid()) {
                if ($value instanceof \Generator) {

```

异常: 嵌套任务透传

```
// a3. 子生成器可能抛出异常
// echo "a3 -> ";
$value = (new self($value))->begin();
}
// echo "a4 -> ";
return $this->next($value);
} else {
    return $result;
}
} catch (\Exception $ex) {
    // !! 当生成器迭代过程发生未捕获异常， 生成器将会被关闭,
valid()返回false,
    if ($this->gen->valid()) {
        // b1. 所以， 当前分支的异常一定不是当前生成器所抛出
        // 而是来自嵌套的子生成器
        // 此处将子生成器异常通过(c)向当前生成器抛出异常
        // echo "b1 -> ";
        return $this->next(null, $ex);
    } else {
        // b2. 逆向(递归栈帧)方向向上抛 或者 向父生成器(如果
存在)抛出异常
        // echo "b2 -> ";
        throw $ex;
    }
}
}
}
```

```
<?php
function newSubGen()
{
    yield 0;
    throw new \Exception("e");
    yield 1;
}
```

异常: 嵌套任务透传

```
function newGen()
{
    try {
        $r1 = (yield newSubGen());
    } catch (\Exception $ex) {
        echo $ex->getMessage();
    }
    $r2 = (yield 2);
    yield 3;
}
$task = new AsyncTask(newGen());
$r = $task->begin(); // output: e
echo $r; // output: 3
```

异常: 传递流程

基于上述注释观察并理解异常传递流程:

```
<?php

function g1()
{
    throw new \Exception();
    yield;
}

// a2 -> b2 ->
(new AsyncTask(g1()))->begin();


function g2()
{
    yield;
    throw new \Exception();
}

// a2 (-> a4 -> a2) -> b2 -> b2 ->
(new AsyncTask(g2()))->begin();


function g3()
{
    yield;
    throw new \Exception();
}

// a2 (-> a4 -> a2) -> b2 -> b2 ->
(new AsyncTask(g3()))->begin();


function g4()
{
    yield;
```

异常: 传递流程

```
yield;
throw new \Exception();
}

// a2 (-> a4 -> a2) (-> a4 -> a2) -> b2 -> b2 -> b2 ->
(new AsyncTask(g4()))->begin();

function g5()
{
    throw new \Exception();
    /** @noinspection PhpUnreachableStatementInspection */
    yield;
}

function g7()
{
    yield g5();
}

// (a2 -> a3) -> a2 (-> b2 -> b1 -> c) -> b2 ->
(new AsyncTask(g7()))->begin();

function g6()
{
    yield;
    throw new \Exception();
}

function g8()
{
    yield g6();
}

// (a2 -> a3) -> a2 -> a4 -> a2 -> b2 -> b2 -> b1 -> c -> b2
->
(new AsyncTask(g8()))->begin();

function g9()
{
    try {

```

异常: 传递流程

```
    yield g5();
} catch (\Exception $ex) {

}
// a2 -> a3 -> a2 -> b2 -> b1 -> c ->
(new AsyncTask(g9()))->begin();
```

异常: 重新进行CPS变换

我们把加入异常处理的代码重新修改为CPS方式:

```
<?php
final class AsyncTask
{
    public $continuation;

    public function begin(callable $continuation)
    {
        $this->continuation = $continuation;
        $this->next();
    }

    public function next($result = null, \Exception $ex = null)
    {
        try {
            if ($ex) {
                $value = $this->gen->throw_($ex);
            } else {
                $value = $this->gen->send($result);
            }

            if ($this->gen->valid()) {
                if ($value instanceof \Generator) {
                    // 注意这里
                    $continuation = [$this, "next"];
                    (new self($value))->begin($continuation);
                } else {
                    $this->next($value);
                }
            } else {
                // 迭代结束 返回结果
                $cc = $this->continuation; // cc指向 父生成器ne
            }
        }
    }
}
```

异常: 重新进行CPS变换

```
xt方法 或 用户传入continuation
    $cc($result, null);
}
} catch (\Exception $ex) {
    if ($this->gen->valid()) {
        // 抛出异常
        $this->next(null, $ex);
    } else {
        // 未捕获异常
        $cc = $this->continuation; // cc指向 父生成器ne
    }
}

xt方法 或 用户传入continuation
    $cc(null, $ex);
}
}
}
}
```

```
<?php
function tt()
{
    yield;
    throw new \Exception("e");
}

function t()
{
    yield tt();
    yield 1;
}

$task = new AsyncTask(t());
$trace = function($r, $ex) {
    if ($ex) {
        echo $ex->getMessage(); // output: e
    } else {
        echo $r;
    }
};
```

异常: 重新进行CPS变换

```
$task->begin($trace);
```

```
<?php
function newSubGen()
{
    yield 0;
    throw new \Exception("e");
    yield 1;
}

function newGen()
{
    try {
        $r1 = (yield newSubGen());
    } catch (\Exception $ex) {
        echo $ex->getMessage(); // output: e
    }
    $r2 = (yield 2);
    yield 3;
}
$task = new AsyncTask(newGen());
$trace = function($r, $ex) {
    if ($ex) {
        echo $ex->getMessage();
    } else {
        echo $r; // output: 3
    }
};
$task->begin($trace); // output: e
```

异常: 重新加入Async

重新加入Async, 修改continuation的签名, 加入异常参数:

```
<?php
interface Async
{
    // continuation :: (mixed $r, \Exception $ex) -> void
    public function begin(callable $continuation);
}

final class AsyncTask implements Async
{
    public function next($result = null, $ex = null)
    {
        try {
            // ...
            if ($this->gen->valid()) {
                if ($value instanceof \Generator) {
                    $value = new self($value);
                }

                if ($value instanceof Async) {
                    $cc = [$this, "next"];
                    $value->begin($cc);
                } else {
                    $this->next($value, null);
                }
            } else {
                $cc = $this->continuation;
                $cc($result, null);
            }
        } catch (\Exception $ex) {
            // ...
        }
    }
}
```

异常: 重新加入Async

```
}
```

```
<?php

$trace = function($r, $ex) {
    if ($ex instanceof \Exception) {
        echo "cc_ex:" . $ex->getMessage(), "\n";
    }
};

class AsyncException implements Async
{
    public function begin(callable $cc)
    {
        swoole_timer_after(1000, function() use($cc) {
            $cc(null, new \Exception("timeout"));
        });
    }
}

function newSubGen()
{
    yield 0;
    $async = new AsyncException();
    yield $async;
}

function newGen($try)
{
    $start = time();
    try {
        $r1 = (yield newSubGen());
    } catch (\Exception $ex) {
        // 捕获subgenerator抛出的异常
        if ($try) {
            echo "catch:" . $ex->getMessage(), "\n";
        } else {
            throw $ex;
        }
    }
}
```

异常: 重新加入Async

```
        }
    }

echo time() - $start, "\n";
}

// 内部try-catch异常
$task = new AsyncTask(newGen(true));
$task->begin($trace);
// output:
// catch:timeout
// 1

// 异常传递至AsyncTask的最终回调
$task = new AsyncTask(newGen(false));
$task->begin($trace);
// output:
// cc_ex:timeout
```

Syscall 与 Context

按照nikic的思路引入与调度器内部交互的 `Syscall`，将需要执行的函数打包成 `Syscall`，通过yield返回迭代器，可以从 `Syscall` 参数获取到当前迭代器对象，这里提供了一个外界与`AsyncTask`交互的扩展点。

我们借此演示如何添加跨生成器上下文，在嵌套生成器共享数据，解耦生成器之间依赖。

```
<?php

final class AsyncTask implements Async
{
    public $gen;
    public $continuation;
    public $parent;

    // 我们在构造器添加$parent参数， 把父子生成器链接起来，使其可以进行回溯
    public function __construct(\Generator $gen, AsyncTask $parent = null)
    {
        $this->gen = new Gen($gen);
        $this->parent = $parent;
    }

    public function begin(callable $continuation)
    {
        $this->continuation = $continuation;
        $this->next();
    }

    public function next($result = null, $ex = null)
    {
        try {
            if ($ex) {

```

```

        $value = $this->gen->throw_($ex);
    } else {
        $value = $this->gen->send($result);
    }

    if ($this->gen->valid()) {
        // 这里注意优先级， Syscall 可能返回\Generator 或
者 Async
        if ($value instanceof Syscall) { // Syscall 签
名见下方
            $value = $value($this);
        }

        if ($value instanceof \Generator) {
            $value = new self($value, $this);
        }

        if ($value instanceof Async) {
            $cc = [$this, "next"];
            $value->begin($cc);
        } else {
            $this->next($value, null);
        }
    } else {
        $cc = $this->continuation;
        $cc($result, null);
    }
} catch (\Exception $ex) {
    if ($this->gen->valid()) {
        $this->next(null, $ex);
    } else {
        $cc = $this->continuation;
        $cc($result, $ex);
    }
}
}
}

```

Syscall将 (callable :: AsyncTask \$task -> mixed) 包装成单独类型：

```
<?php
class Syscall
{
    private $fun;

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

    public function __invoke(AsyncTask $task)
    {
        $cb = $this->fun;
        return $cb($task);
    }
}
```

因为PHP对象属性为Hashtable实现，而生成器对象本身无任何属性，我们这里把 Context 的KV数据附加到根生成器对象上，然后得到的Context的Get与Set函数：

```
<?php
function getCtx($key, $default = null)
{
    return new Syscall(function(AsyncTask $task) use($key, $default) {
        while($task->parent && $task = $task->parent);
        if (isset($task->gen->generator->$key)) {
            return $task->gen->generator->$key;
        } else {
            return $default;
        }
    });
}
```

```
function setCtx($key, $val)
{
    return new Syscall(function(AsyncTask $task) use($key, $v
al) {
        while($task->parent && $task = $task->parent);
        $task->gen->generator->$key = $val;
    });
}
```

```
<?php
function setTask()
{
    yield setCtx("foo", "bar");
}

function ctxTest()
{
    yield setTask();
    $foo = (yield getCtx("foo"));
    echo $foo;
}

$task = new AsyncTask(ctxTest());
$task->begin($trace); // output: bar
```

调度器: 里程碑

除去接口声明不到60行代码，我们实现了支持 任务嵌套 与 异常处理 ， 并可以通过 Syscall 扩充功能的半协程调度器。

接下来我们主要演示如何转换异步回调接口，以及实现一些依赖调度器的实用组件。

spawn

为了易用性，我们为 `AsyncTask` 的创建一个可灵活传递参数函数入口。

```
<?php

/**
 * spawn one semicoroutine
 *
 * @internal param callable|\Generator|mixed $task
 * @internal param callable $continuation function($r = null,
 * $ex = null) {}
 * @internal param AsyncTask $parent
 * @internal param array $ctx Context也可以附加在 \Generator 对
象的属性上
 *
 * 第一个参数为task
 * 剩余参数(优先检查callable)
 *      如果参数类型 callable 则参数被设置为 Continuation
 *      如果参数类型 AsyncTask 则参数被设置为 ParentTask
 *      如果参数类型 array 则参数被设置为 Context
 */
function spawn()
{
    $n = func_num_args();
    if ($n === 0) {
        return;
    }

    $task = func_get_arg(0);
    $continuation = function() {};
    $parent = null;
    $ctx = [];

    for ($i = 1; $i < $n; $i++) {
        $arg = func_get_arg($i);
    }
}
```

spawn

```
if (is_callable($arg)) {
    $continuation = $arg;
} else if ($arg instanceof AsyncTask) {
    $parent = $arg;
} else if (is_array($arg)) {
    $ctx = $arg;
}

if (is_callable($task)) {
    try {
        $task = $task();
    } catch (\Exception $ex) {
        $continuation(null, $ex);
        return;
    }
}

if ($task instanceof \Generator) {
    foreach ($ctx as $k => $v) {
        $task->$k = $v;
    }
    (new AsyncTask($task, $parent))->begin($continuation)
;
} else {
    $continuation($task, null);
}
}
```

call/cc

wiki - call-with-current-continuation

the function call-with-current-continuation, abbreviated call/cc, is a control operator

异步回调API是无法直接使用yield语法的，需要使用thunk或者promise进行转换， thunkify是将多参数函数替换成单参数函数且参数只接受回调(参见：[Thunk 函数的含义和用法](#))。上文中我们将回调api显式实现Async接口，显得有些麻烦，这里可以把“通过call参数\$k传递异步结果”的模式抽象出来，实现一个穷人的call/cc。

```
<?php

// CallCC instanceof Async
class CallCC implements Async
{
    public $fun;

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

    public function begin(callable $continuation)
    {
        $fun = $this->fun;
        $fun($continuation);
    }
}

function callcc(callable $fn)
{
    return new CallCC($fn);
}
```

wiki - call-with-current-continuation

Taking a function f as its only argument, `call/cc` takes the current continuation as an object and applies f to it. The continuation object is a first-class value and is represented as a function, with function application as its only operation. When a continuation object is applied to an argument, the existing continuation is eliminated and the applied continuation is restored in its place, so that the program flow will continue at the point at which the continuation was captured and the argument of the continuation then becomes the "return value" of the `call/cc` invocation. Continuations created with `call/cc` may be called more than once, and even from outside the dynamic extent of the `call/cc` application.

我们的`call/cc`只可以调用一次(Generator是单向的), 虽然我们的`$k`也是 `first-class value`, 但即使 `$k($k)` 进行传递, 也无法达到wiki介绍的效果, PHP不支持Continuation, 我们创造的半协程中的`call/cc`的功能有限, 仅仅借用了`call/cc`的形式。

事实上, `yield`只能将控制权从Generator转移到起caller中:

Wiki - Coroutine

Generators, also known as semicoroutines, are also a generalisation of subroutines, but are more limited than coroutines. Specifically, while both of these can yield multiple times, suspending their execution and allowing re-entry at multiple entry points, they differ in that coroutines can control where execution continues after they yield, while generators cannot, instead transferring control back to the generator's caller. That is, since generators are primarily used to simplify the writing of iterators, the `yield` statement in a generator does not specify a coroutine to jump to, but rather passes a value back to a parent routine.

However, it is still possible to implement coroutines on top of a generator facility, with the aid of a top-level dispatcher routine (a trampoline, essentially) that passes control explicitly to child generators identified by tokens passed back from the generators

来看例子：

```
<?php

function async_sleep($ms)
{
    return callcc(function($k) use($ms) {
        swoole_timer_after($ms, function() use($k) {
            $k(null);
        });
    });
}

function async_dns_lookup($host)
{
    return callcc(function($k) use($host) {
        swoole_async_dns_lookup($host, function($host, $ip) use($k) {
            $k($ip);
        });
    });
}

class HttpClient extends \swoole_http_client
{
    public function async_get($uri)
    {
        return callcc(function($k) use($uri) {
            $this->get($uri, $k);
        });
    }
}
```

```
public function async_post($uri, $post)
{
    return callcc(function($k) use($uri, $post) {
        $this->post($uri, $post, $k);
    });
}

public function async_execute($uri)
{
    return callcc(function($k) use($uri) {
        $this->execute($uri, $k);
    });
}

// 这里!
spawn(function() {
    $ip = (yield async_dns_lookup("www.baidu.com"));
    $cli = new HttpClient($ip, 80);
    $cli->setHeaders(["foo" => "bar"]);
    $cli = (yield $cli->async_get("/"));
    echo $cli->body, "\n";
});
```

我们可以用相同的方式来封装swoole其他的异步api(TcpClient, MysqlClient, RedisClient...), 大家可以举一反三(建议继承swoole原生类, 而不是直接实现Async)。

race与timeout

看到这里，你可能已经发现到我们封装的异步接口的问题了：没有任何超时处理。

通常情况我们会为每个异步调用添加定时器，回调成功取消定时器，否则在定时器回调透传异常，例如：

```
<?php

// helper function
function once(callable $fun)
{
    $has = false;
    return function(...$args) use($fun, &$has) {
        if ($has === false) {
            $fun(...$args);
            $has = true;
        }
    };
}

// helper function
function timeoutWrapper(callable $fun, $timeout)
{
    return function($k) use($fun, $timeout) {
        $k = once($k);
        $fun($k);
        swoole_timer_after($timeout, function() use($k) {
            // 这里异常可以从外部传入
            $k(null, new \Exception("timeout")));
        });
    };
}
```

```
<?php
// 为callcc添加超时处理
function callcc(callable $fun, $timeout = 0)
{
    if ($timeout > 0) {
        $fun = timeoutWrapper($fun, $timeout);
    }
    return new CallCC($fun);
}

// 我们的dns查询有了超时透传异常的能力了
function async_dns_lookup($host, $timeout = 100)
{
    return callcc(function($k) use($host) {
        swoole_async_dns_lookup($host, function($host, $ip) use($k) {
            $k($ip);
        });
    }, $timeout);
}

spawn(function() {
    try {
        yield async_dns_lookup("www.xxx.com", 1);
    } catch (\Exception $ex) {
        echo $ex; // ex!
    }
});
```

但是，我们可以有更优雅通用的方式来超时处理：

```
<?php

class Any implements Async
{
```

```
public $parent;
public $tasks;
public $continuation;
public $done;

public function __construct(array $tasks, AsyncTask $parent = null)
{
    assert(!empty($tasks));
    $this->tasks = $tasks;
    $this->parent = $parent;
    $this->done = false;
}

public function begin(callable $continuation)
{
    $this->continuation = $continuation;
    foreach ($this->tasks as $id => $task) {
        (new AsyncTask($task, $this->parent))->begin($this->continuation($id));
    };
}

private function continuation($id)
{
    return function($r, $ex = null) use($id) {
        if ($this->done) {
            return;
        }
        $this->done = true;

        if ($this->continuation) {
            $k = $this->continuation;
            $k($r, $ex);
        }
    };
}
```

```
<?php
// helper function
function await($task, ...$args)
{
    if ($task instanceof \Generator) {
        return $task;
    }

    if (is_callable($task)) {
        $gen = function() use($task, $args) { yield $task(...$args); };
    } else {
        $gen = function() use($task) { yield $task; };
    }
    return $gen();
}

function race(array $tasks)
{
    $tasks = array_map(__NAMESPACE__ . "\\\\await", $tasks);

    return new Syscall(function(AsyncTask $parent) use($tasks)
{
    if (empty($tasks)) {
        return null;
    } else {
        return new Any($tasks, $parent);
    }
});
}
```

我们构造了一个与Promise.race相同语义的接口，而我们之前构造Async接口则可以看成简陋版的Promise.then + Promise.catch。

```
<?php

// 我们重新来看这个简单dns查询函数
function async_dns_lookup($host)
{
    return callcc(function($k) use($host) {
        swoole_async_dns_lookup($host, function($host, $ip) use($k) {
            $k($ip);
        });
    });
}

// 我们有了一个纯粹的超时透传异常的函数
function timeout($ms)
{
    return callcc(function($k) use($ms) {
        swoole_timer_after($ms, function() use($k) {
            $k(null, new \Exception("timeout"));
        });
    });
}

// 当我们采取race语义并发执行dns查询与超时异常函数
// 其实我们构造了一个更为灵活的超时处理方案
spawn(function() {
    try {
        $ip = (yield race([
            async_dns_lookup("www.baidu.com"),
            timeout(100),
        ]));

        $res = (yield race([
            (new HttpClient($ip, 80))->awaitGet("/"),
            timeout(200),
        ]));
        var_dump($res->statusCode);
    } catch (\Exception $ex) {

```

race与timeout

```
        echo $ex;
        swoole_event_exit();
    }
});
```

我们非常容易构造出更多支持超时的接口, 但我们代码看起来比之前更加清晰;

```
<?php

class HttpClient extends \swoole_http_client
{
    public function awaitGet($uri, $timeout = 1000)
    {
        return race([
            callcc(function($k) use($uri) {
                $this->get($uri, $k);
            }),
            timeout($timeout),
        ]);
    }
    // ...
}

function async_dns_lookup($host, $timeout = 100)
{
    return race([
        callcc(function($k) use($host) {
            swoole_async_dns_lookup($host, function($host, $ip) use($k) {
                $k($ip);
            });
        }),
        timeout($timeout),
    ]);
}
```

```
// test
spawn(function() {
    try {
        $ip = (yield race([
            async_dns_lookup("www.baidu.com"),
            timeout(100),
        ]));
        $res = (yield (new HttpClient($ip, 80))>awaitGet("/"));
    });
    var_dump($res->statusCode);
} catch (\Exception $ex) {
    echo $ex;
    swoole_event_exit();
}
});
```

all与parallel

Any表示多个异步回调，任一回调完成则任务完成，All表示等待所有回调均执行完成才算任务完成，二者相同点是IO部分并发执行；

```
<?php

class All implements Async
{
    public $parent;
    public $tasks;
    public $continuation;

    public $n;
    public $results;
    public $done;

    public function __construct(array $tasks, AsyncTask $parent = null)
    {
        $this->tasks = $tasks;
        $this->parent = $parent;
        $this->n = count($tasks);
        assert($this->n > 0);
        $this->results = [];
    }

    public function begin(callable $continuation = null)
    {
        $this->continuation = $continuation;
        foreach ($this->tasks as $id => $task) {
            (new AsyncTask($task, $this->parent))->begin($this->continuation($id));
        };
    }
}
```

```

private function continuation($id)
{
    return function($r, $ex = null) use($id) {
        if ($this->done) {
            return;
        }

        // 任一回调发生异常，终止任务
        if ($ex) {
            $this->done = true;
            $k = $this->continuation;
            $k(null, $ex);
            return;
        }

        $this->results[$id] = $r;
        if (--$this->n === 0) {
            // 所有回调完成，终止任务
            $this->done = true;
            if ($this->continuation) {
                $k = $this->continuation;
                $k($this->results);
            }
        }
    };
}

function all(array $tasks)
{
    $tasks = array_map(__NAMESPACE__ . "\\\\await", $tasks);

    return new Syscall(function(AsyncTask $parent) use($tasks)
    {
        if (empty($tasks)) {
            return null;
        } else {

```

```

        return new All($tasks, $parent);
    }
});

}


```

```

<?php
spawn(function() {
    $ex = null;
    try {
        $r = (yield all([
            async_dns_lookup("www.bing.com", 100),
            async_dns_lookup("www.so.com", 100),
            async_dns_lookup("www.baidu.com", 100),
        ]));
        var_dump($r);

        /*
        array(3) {
            [0]=>
            string(14) "202.89.233.103"
            [1]=>
            string(14) "125.88.193.243"
            [2]=>
            string(15) "115.239.211.112"
        }
        */
    } catch (\Exception $ex) {
        echo $ex;
    }
});

```

我们这里实现了与Promise.all相同语义的接口，或者更复杂一些，我们也可以实现批量任务以chunk方式进行作业的接口，留待读者自己完成；

至此，我们已经拥有了 `spawn` `callcc` `race` `all` `timeout`。

channel与协程间通信

虽然已经构建了基于yield的半协程，之前所有讨论都集中在单个协程，我们可以再深入一步，构造带有阻塞语义的协程间通信原语--channel，这里按照Golang的channel来实现；

playground

By default, sends and receives block until the other side is ready.

This allows goroutines to synchronize without explicit locks or condition variables.

相比Golang，我们因为只有一个线程，对于chan发送与接收的阻塞的处理，最终会被转换为对使用channel的协程的控制权的转移。

无缓存channel

我们首先实现无缓存的Channel:

```
<?php

class Channel
{
    // 因为同一个channel可能有多个接收者，使用队列实现，保证调度均衡
    // 队列内保存的是被阻塞的接收者协程的控制流，即call/cc的参数，我们模拟的continuation
    public $recvQ;

    // 发送者队列逻辑相同
    public $sendQ;

    public function __construct()
    {
        $this->recvQ = new \SplQueue();
        $this->sendQ = new \SplQueue();
    }

    public function send($val)
    {
        return callcc(function($cc) use($val) {
            if ($this->recvQ->isEmpty()) {

                // 当chan没有接收者，发送者协程挂起(将$cc入列，不调用
                // $cc回送数据)
                $this->sendQ->enqueue([$cc, $val]);

            } else {

                // 当chan对端有接收者，将挂起接收者协程出列，
                // 调用接收者$recvCc发送数据，运行接收者协程后继代码
                // 执行完毕或者遇到Async挂起，$recvCc()调用返回，

            }
        });
    }
}
```

无缓存channel

```
// 调用$cc(), 控制流回到发送者协程
$recvCc = $this->recvQ->dequeue();
$recvCc($val, null);
$cc(null, null);

}

});

}

public function recv()
{
    return callcc(function($cc) {
        if ($this->sendQ->isEmpty()) {

            // 当chan没有发送者，接收者协程挂起（将$cc入列）
            $this->recvQ->enqueue($cc);

        } else {

            // 当chan对端有发送者，将挂起发送者协程与待发送数据出列

            // 调用发送者$sendCc发送数据，运行发送者协程后继代码
            // 执行完毕或者遇到Async挂起，$sendCc()调用返回，
            // 调用$cc(), 控制流回到接收者协程
            list($sendCc, $val) = $this->sendQ->dequeue()

;

            $sendCc(null, null);
            $cc($val, null);

        }
    });
}
```

缓存channel

接下来我们来实现带缓存的Channel:

Sends to a buffered channel block only when the buffer is full.

Receives block when the buffer is empty.

```
<?php

class BufferChannel
{
    // 缓存容量
    public $cap;

    // 缓存
    public $queue;

    // 同无缓存Channel
    public $recvCc;

    // 同无缓存Channel
    public $sendCc;

    public function __construct($cap)
    {
        assert($cap > 0);
        $this->cap = $cap;
        $this->queue = new \SplQueue();
        $this->sendCc = new \SplQueue();
        $this->recvCc = new \SplQueue();
    }

    public function recv()
    {
        return callcc(function($cc) {

```

缓存channel

```
    if ($this->queue->isEmpty()) {

        // 当无数据可接收时, $cc入列, 让出控制流, 挂起接收者协程

        $this->recvCc->enqueue($cc);

    } else {

        // 当有数据可接收时, 先接收数据, 然后恢复控制流
        $val = $this->queue->dequeue();
        $this->cap++;
        $cc($val, null);

    }

    // 递归唤醒其他被阻塞的发送者与接收者收发数据, 注意顺序
    $this->recvPingPong();
};

}

public function send($val)
{
    return callcc(function($cc) use($val) {
        if ($this->cap > 0) {

            // 当缓存未满, 发送数据直接加入缓存, 然后恢复控制流
            $this->queue->enqueue($val);
            $this->cap--;
            $cc(null, null);

        } else {

            // 当缓存满, 发送者控制流与发送数据入列, 让出控制流, 挂
            // 起发送者协程
            $this->sendCc->enqueue([$cc, $val]);

        }

    })
}
```

缓存channel

```
// 递归唤醒其他被阻塞的接收者与发送者收发数据，注意顺序
$this->sendPingPong();

// 如果全部代码都为同步，防止多个发送者时，数据全部来自某个
发送者
    // 应该把sendPingPong 修改为异步执行 defer([$this, "s
endPingPong"]);
    // 但是swoole本身的defer实现有bug，除非把defer 实现为s
woole_timer_after(1, ...)
    // recvPingPong 同理
}
}

public function recvPingPong()
{
    // 当有阻塞的发送者，唤醒其发送数据
    if (!$this->sendCc->isEmpty() && $this->cap > 0) {
        list($sendCc, $val) = $this->sendCc->dequeue();
        $this->queue->enqueue($val);
        $this->cap--;
        $sendCc(null, null);

        // 当有阻塞的接收者，唤醒其接收数据
        if (!$this->recvCc->isEmpty() && !$this->queue->i
sEmpty()) {
            $recvCc = $this->recvCc->dequeue();
            $val = $this->queue->dequeue();
            $this->cap++;
            $recvCc($val);

            $this->recvPingPong();
        }
    }
}

public function sendPingPong()
{
    // 当有阻塞的接收者，唤醒其接收数据
```

缓存channel

```
if (!$this->recvCc->isEmpty() && !$this->queue->isEmpty()) {
    $recvCc = $this->recvCc->dequeue();
    $val = $this->queue->dequeue();
    $this->cap++;
    $recvCc($val);

    // 当有阻塞的发送者，唤醒其发送数据
    if (!$this->sendCc->isEmpty() && $this->cap > 0)
    {
        list($sendCc, $val) = $this->sendCc->dequeue();
        $this->queue->enqueue($val);
        $this->cap--;
        $sendCc(null, null);

        $this->sendPingPong();
    }
}
}

}
```

channel演示

这是我们最终得到的接口：

```
<?php

function go(...$args)
{
    spawn(...$args);
}

function chan($n = 0)
{
    if ($n === 0) {
        return new Channel();
    } else {
        return new BufferChannel($n);
    }
}
```

第一个典型例子，PINGPONG。

与golang的channel类似， 我们可以在两个semicoroutine之间做同步：

```
<?php

// 构建两个单向channel， 我们只单向收发数据

$pingCh = chan();
$pongCh = chan();

go(function() use($pingCh, $pongCh) {
    while (true) {
        echo (yield $pingCh->recv());
        yield $pongCh->send("PONG\n");
    }
});
```

```
// 递归调度器实现，需要引入异步的方法退栈，否则Stack Overflow
...
// 或者考虑将send或者recv以defer方式实现
yield async_sleep(1);
}

});

go(function() use($pingCh, $pongCh) {
    while (true) {
        echo (yield $pongCh->recv());
        yield $pingCh->send("PING\n");

        yield async_sleep(1);
    }
});

// start up
go(function() use($pingCh) {
    echo "start up\n";
    yield $pingCh->send("PING");
});

// output:
/*
start up
PING
PONG
PING
PONG
PING
PONG
PING
...
*/
```

当然，我们可以很轻易构建一个生产者-消费者模型：

```
<?php

$ch = chan();

// 生产者1
go(function() use($ch) {
    while (true) {
        yield $ch->send("producer 1");
        yield async_sleep(1000);
    }
});

// 生产者2
go(function() use($ch) {
    while (true) {
        yield $ch->send("producer 2");
        yield async_sleep(1000);
    }
});

// 消费者1
go(function() use($ch) {
    while (true) {
        $recv = (yield $ch->recv());
        echo "consumer1: $recv\n";
    }
});

// 消费者2
go(function() use($ch) {
    while (true) {
        $recv = (yield $ch->recv());
        echo "consumer2: $recv\n";
    }
});
```

channel演示

```
// output:  
/*  
consumer1 recv from producer 1  
consumer1 recv from producer 2  
consumer1 recv from producer 1  
consumer2 recv from producer 2  
consumer1 recv from producer 2  
consumer2 recv from producer 1  
consumer1 recv from producer 1  
consumer2 recv from producer 2  
consumer1 recv from producer 2  
consumer2 recv from producer 1  
consumer1 recv from producer 1  
consumer2 recv from producer 2  
.....  
*/
```

channel 自身是 first-class value , 所以可传递:

```
<?php  
  
// 我们通过一个chan来发送另一个chan  
// 然后等待接收到这个chan的semicoroutine回送数据  
  
$ch = chan();  
  
go(function() use ($ch) {  
    $anotherCh = chan();  
    yield $ch->send($anotherCh);  
    echo "send another channel\n";  
    yield $anotherCh->send("HELLO");  
    echo "send hello through another channel\n";  
});  
  
go(function() use($ch) {  
    /** @var Channel $anotherCh */  
    $anotherCh = (yield $ch->recv());
```

channel演示

```
echo "recv another channel\n";
$val = (yield $anotherCh->recv());
echo $val, "\n";
});

// output:
/*
send another channel
recv another channel
send hello through another channel
HELLO
*/
```

我们通过控制channel缓存大小 观察输出结果：

```
<?php
$ch = chan($n);

go(function() use($ch) {
    $recv = (yield $ch->recv());
    echo "recv $recv\n";
    $recv = (yield $ch->recv());
    echo "recv $recv\n";
});

go(function() use($ch) {
    yield $ch->send(1);
    echo "send 1\n";
    yield $ch->send(2);
    echo "send 2\n";
    yield $ch->send(3);
    echo "send 3\n";
    yield $ch->send(4);
```

channel演示

```
echo "send 4\n";
});

// $n = 1;
// output:
/*
send 1
recv 1
send 2
recv 2
send 3
recv 3
send 4
recv 4
*/\n\n

// $n = 2;
// output:
/*
send 1
send 2
recv 1
recv 2
send 3
send 4
recv 3
recv 4
*/\n\n

// $n = 3;
// output:
/*
send 1
send 2
send 3
recv 1
recv 2
recv 3
```

```
send 4
recv 4
*/
```

一个更具体的生产者消费者的例子：

```
<?php

// 缓存两个结果
$ch = chan(2);

// 从channel接口请求写过写文件
go(function() use($ch) {
    $file = new AsyncFile("path/to/save");
    while (true) {
        list($host, $status) = (yield $ch->recv());
        yield $file->write("$host: $status\n");
    }
});

// 请求并写入chan
go(function() use($ch) {
    while (true) {
        $host = "www.baidu.com";
        $resp = (yield async_curl_get($host));
        yield $ch->send([$host, $resp->statusCode]);
    }
});

// 请求并写入chan
go(function() use($ch) {
    while (true) {
        $host = "www.bing.com";
        $resp = (yield async_curl_get($host));
        yield $ch->send([$host, $resp->statusCode]);
    }
});
```

```
// output:
```

channel的发送与接受没有超时机制，Golang可以select多个chan实现超时处理，我们可以做一个select设施，或者在send于recv接受直接添加超时参数，扩展接口功能，留待读者自行实现。

FutureTask与fork

在多线程代码中我们经常会遇到这种模型，将一个耗时任务，new一个新的Thread或者通常放到线程池后台执行，当前线程执行另外任务，之后通过某个api接口阻塞获取后台任务结果。

Java童鞋应该对这个概念非常熟悉——JDK给予直接支持的Future。

同样的模型我们可以利用channel对多个协程做进行同步来实现，代码很简单：

```
<?php

go(function() {
    $start = microtime(true);

    $ch = chan();

    // 开启一个新的协程，异步执行耗时任务
    spawn(function() use($ch) {
        yield delay(1000);
        yield $ch->send(42); // 通知+传送结果
    });

    yield delay(500);
    $r = (yield $ch->recv()); // 阻塞等待结果
    echo $r; // 42

    // 我们这里两个耗时任务并发执行，总耗时约1000ms
    echo "cost ", microtime(true) - $start, "\n";
});
```

事实上我们也很容易把Future模型移植过来：

```
<?php
```

FutureTask与fork

```
final class FutureTask
{
    const PENDING = 1;
    const DONE = 2;
    private $cc;

    public $state;
    public $result;
    public $ex;

    public function __construct(\Generator $gen)
    {
        $this->state = self::PENDING;

        $asyncTask = new AsyncTask($gen);
        $asyncTask->begin(function($r, $ex = null) {
            $this->state = self::DONE;
            if ($cc = $this->cc) {
                // 有cc, 说明有call get方法挂起协程, 在此处唤醒
                $cc($r, $ex);
            } else {
                // 无挂起, 暂存执行结果
                $this->result = $r;
                $this->ex = $ex;
            }
        });
    }

    public function get()
    {
        return callcc(function($cc) use($timeout) {
            if ($this->state === self::DONE) {
                // 获取结果时, 任务已经完成, 同步返回结果
                // 这里也可以考虑用defer实现, 异步返回结果, 首先先释放php栈, 降低内存使用
                $cc($this->result, $this->ex);
            } else {
                // 获取结果时未完成, 保存$cc, 挂起等待
            }
        });
    }
}
```

FutureTask与fork

```
        $this->cc = $cc;
    }
});
```

}

```
// helper
function fork($task, ...$args)
{
    $task = await($task); // 将task转换为生成器
    yield new FutureTask($task);
}
```

还是刚才那个例子，我们改写为FutureTask版本：

```
<?php
go(function() {
    $start = microtime(true);

    // fork 子协程执行耗时任务
    /** @var $future FutureTask */
    $future = (yield fork(function() {
        yield delay(1000);
        yield 42;
    }));
    yield delay(500);

    // 阻塞等待结果
    $r = (yield $future->get());
    echo $r; // 42

    // 总耗时仍旧只有1000ms
    echo "cost ", microtime(true) - $start, "\n";
});
```

再进一步，我们扩充FutureTask的状态，阻塞获取结果加入超时选项：

```
<?php

final class FutureTask
{
    const PENDING = 1;
    const DONE = 2;
    const TIMEOUT = 3;

    private $timerId;
    private $cc;

    public $state;
    public $result;
    public $ex;

    // 我们这里加入新参数，用来链接futureTask与caller父任务
    // 这样的好处比如可以共享父子任务上下文
    public function __construct(\Generator $gen, AsyncTask $parent = null)
    {
        $this->state = self::PENDING;

        if ($parent) {
            $asyncTask = new AsyncTask($gen, $parent);
        } else {
            $asyncTask = new AsyncTask($gen);
        }

        $asyncTask->begin(function($r, $ex = null) {
            // PENDING or TIMEOUT
            if ($this->state === self::TIMEOUT) {
                return;
            }

            // PENDING -> DONE
            $this->state = self::DONE;
        });
    }
}
```

```

        if ($cc = $this->cc) {
            if ($this->timerId) {
                swoole_timer_clear($this->timerId);
            }
            $cc($r, $ex);
        } else {
            $this->result = $r;
            $this->ex = $ex;
        }
    });
}

// 这里超时时间0为永远阻塞,
// 否则超时未获取到结果, 将向父任务传递超时异常
public function get($timeout = 0)
{
    return callcc(function($cc) use($timeout) {
        // PENDING or DONE
        if ($this->state === self::DONE) {
            $cc($this->result, $this->ex);
        } else {
            // 获取结果时未完成, 保存$cc, 开启定时器(如果需要),
挂起等待
            $this->cc = $cc;
            $this->getResultTimeout($timeout);
        }
    });
}

private function getResultTimeout($timeout)
{
    if (!$timeout) {
        return;
    }

    $this->timerId = swoole_timer_after($timeout, function
() {

```

FutureTask与fork

```
        assert($this->state === self::PENDING);
        $this->state = self::TIMEOUT;
        $cc = $this->cc;
        $cc(null, new AsyncTimeoutException());
    });
}
}
```

因为引入parentTask参数，需要将父任务隐式传参，而我们执行通过Syscall与执行当前生成器的父任务交互，所以我们重写fork辅助函数，改用Syscall实现：

```
<?php

/**
 * @param $task
 * @return Syscall
 */
function fork($task)
{
    $task = await($task);
    return new Syscall(function(AsyncTask $parent) use($task)
{
    return new FutureTask($task, $parent);
});
}
```

下面看一些关于超时的示例：

```
<?php

go(function() {
    $start = microtime(true);
```

FutureTask与fork

```
/** @var $future FutureTask */
$future = (yield fork(function() {
    yield delay(500);
    yield 42;
}));

// 阻塞等待超时，捕获到超时异常
try {
    $r = (yield $future->get(100));
    var_dump($r);
} catch (\Exception $ex) {
    echo "get result timeout\n";
}

yield delay(1000);

// 因为我们只等待子任务100ms，我们的总耗时只有 1100ms
echo "cost ", microtime(true) - $start, "\n";
});

go(function() {
    $start = microtime(true);

    /** @var $future FutureTask */
    $future = (yield fork(function() {
        yield delay(500);
        yield 42;
        throw new \Exception();
    }));
}

yield delay(1000);

// 子任务500ms前发生异常，已经处于完成状态
// 我们调用get会当即引发异常
try {
    $r = (yield $future->get());
    var_dump($r);
} catch (\Exception $ex) {
```

FutureTask与fork

```
    echo "something wrong in child task\n";
}

// 因为耗时任务并发执行，这里总耗时仅1000ms
echo "cost ", microtime(true) - $start, "\n";
});
```

第二部分: Koa

Koa自述是下一代web框架:

由 Express 原班人马打造的 Koa，致力于成为一个更小、更健壮、更富有表现力的 Web 框架。使用 Koa 编写 web 应用，通过组合不同的 generator，可以免除重复繁琐的回调函数嵌套，并极大地提升常用错误处理效率。Koa 不在内核方法中绑定任何中间件，它仅仅提供了一个轻量优雅的函数库，使得编写 Web 应用变得得心应手。

像 Ruby Rack Python django Golang martini Node Express PHP Laravel Java Spring，Web框架大多会有一个面向AOP的中间件模块，内部操纵Req/Res对象可选执行next动作，Koa与martini类似，都属于设计清爽的中间件web框架，采用洋葱模型middleware stack，但合并了Request与Response对象，编写更直观方便。

Koa's middleware stack flows in a stack-like manner, allowing you to perform actions downstream then filter and manipulate the response upstream. Each middleware receives a Koa context object that encapsulates an incoming http message and the corresponding response to that message. ctx is often used as the parameter name for the context object.

[Laravel Middleware](#) Laravel after方法的书写不够自然，需要中间变量保存 Response; Spring的Interceptor功能强大，但编写略繁琐; Koa则简单的多，而且可以在某个中间件中灵活控制之后中间件执行时机。

其实对于web框架来讲，业务逻辑归根结底都在处理请求与相应用对象，web中间件实质就是在请求与响应中间开放出来的可编排的扩展点，比如修改请求做URLRewrite，比如请求日志，身份验证...

真正的业务逻辑都可以通过middleware实现，或者说按特定顺序对中间件灵活组合编排。

我们基于PHP5.6与yz-swoole(有赞内部自研稳定版本的swoole, 17年6月即将开源, 敬请期待), 用少量的代码即可实现(chao)现(xi)一个php-koa。

Koa2.x决定全面使用async/await, 受限于PHP语法, 我们这里仅实现Koa1.x, 将Generator作为middleware实现形式。

穿越地心之旅

(如果你了解洋葱圈模型，略过本小节)

首先让我们对洋葱圈模型有一个直观的认识：

地球构造

物理学上,地球可划分为岩石圈、软流层、地幔、外核和内核5层。

化学上,地球被划分为地壳、上地幔、下地幔、外核和内核。地质学上对地球各层的划分

演示Koa的中间件之前，我们用函数来描述一场穿越地心之旅：

```
<?php
function crust($next)
{
    echo "到达<地壳>\n";
    $next();
    echo "离开<地壳>\n";
}

function upperMantle($next)
{
    echo "到达<上地幔>\n";
    $next();
    echo "离开<上地幔>\n";
}

function mantle($next)
{
    echo "到达<下地幔>\n";
    $next();
    echo "离开<下地幔>\n";
```

```
}

function outerCore($next)
{
    echo "到达<外核>\n";
    $next();
    echo "离开<外核>\n";
}

function innerCore($next)
{
    echo "到达<内核>\n";
}

// 我们逆序组合组合，返回入口
function makeTravel(...$layers)
{
    $next = null;
    $i = count($layers);
    while ($i--) {
        $layer = $layers[$i];
        $next = function() use($layer, $next) {
            // 这里next指向穿越下一次的函数，作为参数传递给上一层调用
            $layer($next);
        };
    }
    return $next;
}

// 我们途径 crust -> upperMantle -> mantle -> outerCore -> innerCore 到达地心
// 然后穿越另一半球 -> outerCore -> mantle -> upperMantle -> crust

$travel = makeTravel("crust", "upperMantle", "mantle", "outerCore", "innerCore");
$travel(); // output:
```

穿越地心之旅

```
/*
到达<地壳>
到达<上地幔>
到达<下地幔>
到达<外核>
到达<内核>
离开<外核>
离开<下地幔>
离开<上地幔>
离开<地壳>
*/
```

```
<?php
```

```
function outerCore1($next)
{
    echo "到达<外核>\n";
    // 我们放弃内核，仅仅绕外壳一周，从另一侧返回地表
    // $next();
    echo "离开<外核>\n";
}
```

```
$travel = makeTravel("crust", "upperMantle", "mantle", "outer
Core1", "innerCore");
$travel(); // output:
/*
到达<地壳>
到达<上地幔>
到达<下地幔>
到达<外核>
离开<外核>
离开<下地幔>
离开<上地幔>
```

```
离开<地壳>
```

```
*/
```

```
<?php
function innerCore1($next)
{
    // 我们到达内核之前遭遇了岩浆，计划终止，等待救援
    throw new \Exception("岩浆");
    echo "到达<内核>\n";
}

$travel = makeTravel("crust", "upperMantle", "mantle", "outer
Core", "innerCore1");
$travel(); // output:
/*
到达<地壳>
到达<上地幔>
到达<下地幔>
到达<外核>
Fatal error: Uncaught exception 'Exception' with message '岩浆
'
*/
/*
```

```
<?php
```

```
function mantle1($next)
{
    echo "到达<下地幔>\n";
    // 我们在下地幔的救援团队及时赶到 (try catch)
    try {
        $next();
    } catch (\Exception $ex) {
```

```
    echo "遇到", $ex->getMessage(), "\n";
}

// 我们仍旧没有去往内核， 绕道对端下地幔， 返回地表
echo "离开<下地幔>\n";
}

$travel = makeTravel("crust", "upperMantle", "mantle1", "outerCore",
"innerCore1");
$travel(); // output:
/*
到达<地壳>
到达<上地幔>
到达<下地幔>
到达<外核>
遇到岩浆
离开<下地幔>
离开<上地幔>
离开<地壳>
*/

```

```
<?php

function upperMantle1($next)
{
    // 我们放弃对去程上地幔的暂留
    // echo "到达<上地幔>\n";
    $next();
    // 只在返程时暂留
    echo "离开<上地幔>\n";
}

function outerCore2($next)
{
    // 我们决定只在去程考察外核
}
```

```
echo "到达<外核>\n";
$next();
// 因为温度过高，去程匆匆离开外壳
// echo "离开<外核>\n";
}

$travel = makeTravel("crust", "upperMantle1", "mantle1", "outerCore2", "innerCore1");
$travel(); // output:
/*
到达<地壳>
到达<上地幔>
到达<下地幔>
到达<外核>
遇到岩浆
离开<下地幔>
离开<上地幔>
离开<地壳>
*/
```

洋葱圈模型

我们把函数从内向外组合，把内层函数的执行控制权包裹成next参数传递给外层函数，让外层函数自行控制内层函数执行时机，我们再一次把控制流暴露出来，第一次是引入continuation，把return决定的控制流暴露到参数中。

于是 我们可以在外层函数 执行next的前后加入自己的逻辑，得到 AOP 的 before与after语义，但不仅仅如此，我们甚至可以不执行内层函数，然后我们穿越地心的过程沿着某个半圆绕过了地核。

再或者，我们也可以放弃after，函数成为 前置过滤器，如果我们放弃 before，函数成为 后置过滤器。

关于错误处理，我们可以在某层的函数 try-catch next调用，从而阻止内层函数的异常向上传递，想想我们在地底深处包裹了一层可以抵御岩浆外太空物质，岩浆被安全的阻止在地心。

事实上这就是一个极简的洋葱圈结构。

rightReduce与中间件compose

然后，有心同学可能会发现，我们的makeTravel其实是一个对函数列表进行rightReduce的函数：

```
<?php

function compose(...$fns)
{
    return array_right_reduce($fns, function($carry, $fn) {
        return function() use($carry, $fn) {
            $fn($carry);
        };
    });
}

function array_right_reduce(array $input, callable $function,
    $initial = null)
{
    return array_reduce(array_reverse($input, true), $function,
        $initial);
}
```

将上述makeTravel替换成compose，我们将得到完全相同的结果。

我们修改函数列表中函数的签名，`middleware :: (Context $ctx, Generator $next) -> void`，我们把满足`List<Middleware>`的函数列表进行组合(rightReduce)，得到中间件入口函数，只需要稍加改造，我们的compose方法便可以处理生成器函数，用来支持我们上文的半协程：

```
<?php
function compose(...$middleware)
{
```

rightReduce与中间件compose

```
return function(Context $ctx = null) use($middleware) {
    $ctx = $ctx ?: new Context(); // Context 参见下文
    $next = null;
    $i = count($middleware);
    while ($i--) {
        $curr = $middleware[$i];
        $next = $curr($ctx, $next);
        assert($next instanceof \Generator);
    }
    return $next;
};
```

如果是中间件是\closure，打包过程可以将\$this变量绑定到\$ctx，中间件内可以使用\$this代替\$ctx，Koa1.x采用这种方式，Koa2.x已经废弃；

```
if ($middleware[$i] instanceof \Closure) {
    //
    $curr = $middleware[$i]->bindTo($ctx, Context::class);
}
```

rightReduce版本：

```
<?php
function compose(array $middleware)
{
    return function(Context $ctx = null) use($middleware) {
        $ctx = $ctx ?: new Context(); // Context 参见下文

        return array_right_reduce($middleware, function($rightNext, $leftFn) use($ctx) {
            return $leftFn($ctx, $rightNext);
        }, null);
    };
}
```


接下来我们来看Koa的仅有四个组件， Application Context Request Response 依次给予实现：

Koa-guide

Koa::Application

The application object is Koa's interface with node's http server and handles the registration

of middleware, dispatching to the middleware from http, default error handling, as well as

configuration of the context, request and response objects.

我们的 App 模块是对 swoole_http_server 的简单封装，在onRequest回调中对中间件compose并执行：

```
<?php

class Application
{
    /** @var \swoole_http_server */
    public $httpServer;
    /** @var Context Prototype Context */
    public $context;
    public $middleware = [];
    public $fn;

    public function __construct()
    {
        // 我们构造一个Context原型
        $this->context = new Context();
        $this->context->app = $this;
    }
}
```

```

// 我们用use方法添加符合接口的中间件
// middleware :: (Context $ctx, $next) -> void
public function use(callable $fn)
{
    $this->middleware[] = $fn;
    return $this;
}

// compose中间件 监听端口提供服务
public function listen($port = 8000, array $config = [])
{
    $this->fn = compose(...$this->middleware);
    $config = ['port' => $port] + $config + $this->defaultConfig();
    $this->httpServer = new \swoole_http_server($config['host'], $config['port'], SWOOLE_PROCESS, SWOOLE_SOCK_TCP);
    $this->httpServer->set($config);
    // 省略绑定 swoole HttpServer 事件, start shutdown connect close workerStart workerStop workerError request
    // ...
    $this->httpServer->start();
}

public function onRequest(\swoole_http_request $req, \swoole_http_response $res)
{
    $ctx = $this->createContext($req, $res);
    $reqHandler = $this->makeRequestHandler($ctx);
    $resHandler = $this->makeResponseHandler($ctx);
    spawn($reqHandler, $resHandler);
}

protected function makeRequestHandler(Context $ctx)
{
    return function() use($ctx) {
        yield setCtx("ctx", $ctx);
        $ctx->res->status(404);
        $fn = $this->fn;
    };
}

```

```
        yield $fn($ctx);
    };

protected function makeResponseHandler(Context $ctx)
{
    return function($r = null, \Exception $ex = null) use
($ctx) {
        if ($ex) {
            $this->handleError($ctx, $ex);
        } else {
            $this->respond($ctx);
        }
    };
}

protected function handleError(Context $ctx, \Exception $ex = null)
{
    if ($ex === null) {
        return;
    }

    if ($ex && $ex->getCode() !== 404) {
        sys_error($ctx);
        sys_error($ex);
    }

    // 非 Http异常， 统一500 status，对外显示异常code
    // Http 异常，自定义status，自定义是否暴露Msg
    $msg = $ex->getMessage();
    if ($ex instanceof HttpException) {
        $status = $ex->status ?: 500;
        $ctx->res->status($status);
        if ($ex->expose) {
            $msg = $ex->getMessage();
        }
    } else {

```

```

        $ctx->res->status(500);
    }

    // force text/plain
    $ctx->res->header("Content-Type", "text"); // TODO accept
    $ctx->res->write($msg);
    $ctx->res->end();
}

protected function respond(Context $ctx)
{
    if ($ctx->respond === false) return; // allow bypassing Koa

    $body = $ctx->body;
    $code = $ctx->status;

    if ($code !== null) {
        $ctx->res->status($code);
    }
    // status.empty() $ctx->body = null; res->end()

    if ($body !== null) {
        $ctx->res->write($body);
    }

    $ctx->res->end();
}

protected function createContext(\swoole_http_request $req, \swoole_http_response $res)
{
    // 可以在Context挂其他组件 $app->foo = bar; $app->listen()
    $context = clone $this->context;

    $request = $context->request = new Request($this, $co

```

Koa::Application

```
    ntext, $req, $res);
        $response = $context->response = new Response($this,
$context, $req, $res);

        $context->app = $this;
        $context->req = $req;
        $context->res = $res;

        $request->response = $response;
        $response->request = $request;

        $request->originalUrl = $req->server["request_uri"];
        $request->ip = $req->server["remote_addr"];

        return $context;
    }
}
```

Koa::Context

Context 组件代理了 Request 与 Response 中的方法和属性，简化了使用方式与中间件接口，这里用php的魔术方法来处理：

```
<?php

class Context
{
    public $app;
    /** @var Request */
    public $request;
    /** @var Response */
    public $response;
    /** @var \swoole_http_request */
    public $req;
    /** @var \swoole_http_response */
    public $res;
    public $state = [];
    public $respond = true;
    /** @var string */
    public $body;
    /** @var int */
    public $status;

    public function __call($name, $arguments)
    {
        $fn = [$this->response, $name];
        return $fn(...$arguments);
    }

    public function __get($name)
    {
        return $this->request->$name;
    }
}
```

```
public function __set($name, $value)
{
    $this->response->$name = $value;
}

// thr[o]w o 为希腊字母 Ομικρον
public function throw($status, $message)
{
    if ($message instanceof \Exception) {
        $ex = $message;
        throw new HttpException($status, $ex->getMessage(),
        $ex->getCode(), $ex->getPrevious());
    } else {
        throw new HttpException($status, $message);
    }
}
```

Koa::Request

Request 组件是对 swoole_http_request 的简单封装：

```
<?php

class Request
{
    /** @var Application */
    public $app;
    /** @var \swoole_http_request */
    public $req;
    /** @var \swoole_http_response */
    public $res;
    /** @var Context */
    public $ctx;
    /** @var Response */
    public $response;
    /** @var string */
    public $originalUrl;
    /** @var string */
    public $ip;

    public function __construct(Application $app, Context $ctx,
        \swoole_http_request $req, \swoole_http_response $res)
    {
        $this->app = $app;
        $this->ctx = $ctx;
        $this->req = $req;
        $this->res = $res;
    }

    public function __get($name)
    {
```

```

        switch ($name) {
            case "rawcontent":
                return $this->req->rawContent();
            case "post":
                return isset($this->req->post) ? $this->req->
post : [];
            case "get":
                return isset($this->req->get) ? $this->req->g
et : [];
            case "cookie":
            case "cookies":
                return isset($this->req->cookie) ? $this->req
->cookie : [];
            case "request":
                return isset($this->req->request) ? $this->re
q->request : [];
            case "header":
            case "headers":
                return isset($this->req->header) ? $this->req
->header : [];
            case "files":
                return isset($this->req->files) ? $this->req-
>files : [];
            case "method":
                return $this->req->server["request_method"];
            case "url":
            case "origin":
                return $this->req->server["request_uri"];
            case "path":
                return isset($this->req->server["path_info"])
? $this->req->server["path_info"] : "";
            case "query":
            case "querystring":
                return isset($this->req->server["query_string"]
) ? $this->req->server["query_string"] : "";
            case "host":
            case "hostname":
                return isset($this->req->header["host"]) ? $t

```

Koa::Request

```
his->req->header["host"] : "";
    case "protocol":
        return $this->req->server["server_protocol"];
    default:
        return $this->req->$name;
}
}
}
```

Koa::Response

Response 组件是对 swoole_http_response 的简单封装：

```
<?php

class Response
{
    /* @var Application */
    public $app;
    /** @var \swoole_http_request */
    public $req;
    /** @var \swoole_http_response */
    public $res;
    /** @var Context */
    public $ctx;
    /** @var Request */
    public $request;
    public $isEnd = false;

    public function __construct(Application $app, Context $ctx,
        \swoole_http_request $req, \swoole_http_response $res)
    {
        $this->app = $app;
        $this->ctx = $ctx;
        $this->req = $req;
        $this->res = $res;
    }

    public function __call($name, $arguments)
    {
        /** @var $fn callable */
        $fn = [$this->res, $name];
        return $fn(...$arguments);
    }
}
```

```
}

public function __get($name)
{
    return $this->res->$name;
}

public function __set($name, $value)
{
    switch ($name) {
        case "type":
            return $this->res->header("Content-Type", $value);
        case "lastModified":
            return $this->res->header("Last-Modified", $value);
        case "etag":
            return $this->res->header("ETag", $value);
        case "length":
            return $this->res->header("Content-Length", $value);
        default:
            return $this->res->header($name, $value);
    }
}

public function end($html = "")
{
    if ($this->isEnd) {
        return false;
    }
    $this->isEnd = true;
    return $this->res->end($html);
}

public function redirect($url, $status = 302)
{
    $this->res->header("Location", $url);
```

Koa::Response

```
    $this->res->header("Content-Type", "text/plain; charset=utf-8");
    $this->ctx->status = $status;
    $this->ctx->body = "Redirecting to $url.";
}

public function render($file)
{
    $this->ctx->body = ($yield Template::render($file, $this->ctx->state));
}
}
```

Koa - HelloWorld

以上便是全部了，我们重点来看示例，我们只注册一个中间件, Hello Worler Server:

```
<?php

$app = new Application();

// ...

$app->use(function(Context $ctx) {
    $ctx->status = 200;
    $ctx->body = "<h1>Hello World</h1>";
});

$app->listen(3000);
```

我们在Hello中间件前面注册一个Reponse-Time中间件，注意看，我们的逻辑是连贯的：

```
<?php

$app->use(function(Context $ctx, $next) {

    $start = microtime(true);

    yield $next; // 执行后续中间件

    $ms = number_format(microtime(true) - $start, 7);

    // response header 写入 X-Response-Time: xxxms
    $ctx->{"X-Response-Time"} = "{$ms}ms";
});
```


Middleware Interface

我们为Middleware声明一个接口，让稍微复杂一点的中间件以类的形式存在，实现该接口的类实例自身满足callable类型，我们的中间件接受任何callable，所以，这并不是必须的，仅仅是为了更好阻止代码，且对PSR4的autoload友好；

```
<?php

interface Middleware
{
    public function __invoke(Context $ctx, $next);
}
```

函数与类并没有孰优孰劣：

Objects are state data with attached behavior;

Closures are behaviors with attached state data and without the overhead of classes.

我们可以在对象形式的中间件附加更多的状态，以应对复杂的场景。

Middleware: 全局异常处理

我们在岩浆的实例其实已经注意到了，compose 的连接方式，让我们有能力精确控制异常。

Koa中间件最终行为强依赖注册顺序，比如我们这里要引入的异常处理，必须在业务逻辑中间件前注册，才能捕获后续中间件中未捕获异常，回想一下我们的调度器实现的异常传递流程。

```
<?php

class ExceptionHandler implements Middleware
{
    public function __invoke(Context $ctx, $next)
    {
        try {
            yield $next;
        } catch (\Exception $ex) {
            $status = 500;
            $code = $ex->getCode() ?: 0;
            $msg = "Internal Error";

            // HttpException的异常通常是通过Context的throw方法抛出
            // 状态码与Msg直接提取可用
            if ($ex instanceof HttpException) {
                $status = $ex->status;
                if ($ex->expose) {
                    $msg = $ex->getMessage();
                }
            }
            // 这里可这对其他异常区分处理
            // else if ($ex instanceof otherException) { }

            $err = [ "code" => $code, "msg" => $msg ];
            if ($ctx->accept("json")) {
                $ctx->status = 200;
            }
        }
    }
}
```

```
        $ctx->body = $err;
    } else {
        $ctx->status = $status;
        if ($status === 404) {
            $ctx->body = (yield Template::render(__DI
R__ . "/404.html"));
        } else if ($status === 500) {
            $ctx->body = (yield Template::render(__DI
R__ . "/500.html", $err));
        } else {
            $ctx->body = (yield Template::render(__DI
R__ . "/error.html", $err));
        }
    }
}

$app->use(new ExceptionHandler());
```

可以将FastRoute与Exception中间件结合，很容易可以定制一个按路由匹配注册的异常处理器，留待读者自行实现。

顺序问题再比如session中间件，需要优先于业务处理中间件，

而像处理404状态码的中间件，则需要在upstream流程中(逆序)的收尾阶段，我们仅仅只关心next后逻辑：

```
<?php
function notfound(Context $ctx, $next)
{
    yield $next;

    if ($ctx->status !== 404 || $ctx->body) {
        return;
    }
}
```

```
$ctx->status = 404;

if ($ctx->accept("json")) {
    $ctx->body = [
        "message" => "Not Found",
    ];
    return;
}

$ctx->body = "<h1>404 Not Found</h1>";
}
```

Middleware: Router

路由是httpServer必不可少的组件，我们使用nikic的[FastRoute](#)来实现一个路由中间件：

```
<?php

use FastRoute\Dispatcher;
use FastRoute\RouteCollector;
use Minimalism\A\Server\Http\Context;

// 这里继承FastRoute的RouteCollector
class Router extends RouteCollector
{
    public $dispatcher;

    public function __construct()
    {
        $routeParser = new \FastRoute\RouteParser\Std();
        $dataGenerator = new \FastRoute\DataGenerator\GroupCo
untBased();
        parent::__construct($routeParser, $dataGenerator);
    }

    // 返回路由中间件
    public function routes()
    {
        $this->dispatcher = new \FastRoute\Dispatcher\GroupCo
untBased($this->getData());
        return [$this, "dispatch"];
    }

    // 路由中间件的主要逻辑
    public function dispatch(Context $ctx, $next)
    {
        if ($this->dispatcher === null) {
```

```

        $this->routes();
    }

    $uri = $ctx->url;
    if (false !== $pos = strpos($uri, '?')) {
        $uri = substr($uri, 0, $pos);
    }
    $uri = rawurlencode($uri);

    // 从Context提取method与url进行分发
    $routeInfo = $this->dispatcher->dispatch(strtoupper($
ctx->method), $uri);
    switch ($routeInfo[0]) {
        case Dispatcher::NOT_FOUND:
            // 状态码写入Context
            $ctx->status = 404;
            yield $next;
            break;
        case Dispatcher::METHOD_NOT_ALLOWED:
            $ctx->status = 405;
            break;
        case Dispatcher::FOUND:
            $handler = $routeInfo[1];
            $vars = $routeInfo[2];

            // 从路由表提取处理器
            yield $handler($ctx, $next, $vars);
            break;
    }
}

$router = new Router();
$router->get('/user/{id:\d+}', function(Context $ctx, $next,
$vars) {
    $ctx->body = "user={$vars['id']}";
});

```

```
// $route->post('/post-route', 'post_handler');
$router->addRoute(['GET', 'POST'], '/test', function(Context
$cxt, $next, $vars) {
    $cxt->body = "";
});

// 分组路由
$router->addGroup('/admin', function (RouteCollector $router)
{
    // handler :: (Context $cxt, $next, array $vars) -> void
    $router->addRoute('GET', '/do-something', 'handler');
    $router->addRoute('GET', '/do-another-thing', 'handler');
    $router->addRoute('GET', '/do-something-else', 'handler')
;
});

$app->use($router->routes());
```

我们已经拥有了一个支持 多方法 正则 参数匹配 分组 功能的路由中间件。

Middleware: 请求超时

请求超时控制也是不可或缺的中间件：

```
<?php

class RequestTimeout implements Middleware
{
    public $timeout;
    public $exception;

    private $timerId;

    public function __construct($timeout, \Exception $ex = null)
    {
        $this->timeout = $timeout;
        if ($ex === null) {
            $this->exception = new HttpException(408, "Request timeout");
        } else {
            $this->exception = $ex;
        }
    }

    public function __invoke(Context $ctx, $next)
    {
        yield race([
            callcc(function($k) {
                $this->timerId = swoole_timer_after($this->timeout, function() use($k) {
                    $k(null, $this->exception);
                });
            }),
            function() use ($next){
                yield $next;
            },
        ]);
    }
}
```

```
        if (swoole_timer_exists($this->timerId)) {
            swoole_timer_clear($this->timerId);
        }
    ],
]);
}

$app->use(new RequestTimeout(2000));
```

也可以结合FastRoute构造出一个按路由匹配请求超时的中间件，留给读者自行实现。

一个综合示例

```
<?php

$app = new Application();

$app->use(new Logger());
$app->use(new Favicon(__DIR__ . "/favicon.ico"));
$app->use(new BodyDetecter()); // 输出特定格式body
$app->use(new ExceptionHandler()); // 处理异常
$app->use(new NotFound()); // 处理404
$app->use(new RequestTimeout(200)); // 处理请求超时，会抛出HttpException

$router = new Router();

$router->get('/index', function(Context $ctx) {
    $ctx->status = 200;
    $ctx->state["title"] = "HELLO WORLD";
    $ctx->state["time"] = date("Y-m-d H:i:s", time());
    $ctx->state["table"] = $_SERVER;
    yield $ctx->render(__DIR__ . "/index.html");
});

$router->get('/404', function(Context $ctx) {
    $ctx->status = 404;
});

$router->get('/bt', function(Context $ctx) {
    $ctx->body = "<pre>" . print_r(debug_backtrace(DEBUG_BACKTRACE_IGNORE_ARGS), true);
});

$router->get('/timeout', function(Context $ctx) {
    // 超时
});
```

一个综合示例

```
    yield async_sleep(500);
});

$router->get('/error/http_exception', function(Context $ctx)
{
    // 抛出带status的错误
    $ctx->throw(500, "Internal Error");
    // 等价于 throw new HttpException(500, "Internal Error");
    yield;
});

$router->get('/error/exception', function(Context $ctx) {
    // 直接抛出错误, 500 错误
    throw new \Exception("some internal error", 10000);
    yield;
});

$router->get('/user/{id:\d+}', function(Context $ctx, $next,
$vars) {
    $ctx->body = "user={$vars['id']}";
    yield;
});

// http://127.0.0.1:3000/request/www.baidu.com
$router->get('/request/{url}', function(Context $ctx, $next,
$vars) {
    $r = (yield async_curl_get($vars['url']));
    $ctx->body = $r->body;
    $ctx->status = $r->statusCode;
});

$app->use($router->routes());

$app->use(function(Context $ctx) {
    $ctx->status = 200;
    $ctx->body = "<h1>Hello World</h1>";
    yield;
});
```

一个综合示例

```
});  
  
$app->listen(3000);
```

以上我们完成了一个基于swoole的版本的php-koa。

附录

个人对yield与coroutine的理解与总结，有问题欢迎指正。

在上文半协程中：

1. 从抽象角度可以将「yield」理解成为CPS变换的语法糖，yield帮我们优雅的链接了异步任务序列；
2. 从控制流角度可以将「yield」理解为将程序控制权从callee(Generator)转移到caller，只有借由底层eventloop驱动，将控制权重新转移回Generator；
3. 从实现角度来看「yield」，每个生成器对象都会有自己的zend_execute_data与zend_vm_stack，调用send\next\throw方法，都需要首先备份EG中相关上下文，然后将Generator的execute_data信息恢复到EG，然后调用zend_execute_ex()从当前上下文恢复执行执行，之后最后恢复执行前EG信息；
4. 因为ZendVM中stack与execute_data的保存与切换工作已经由Generator实现了，基于Generator构建半协程的核心问题是控制流转换；
5. 「yield」并没有消除回调，只是让代码从视觉上变成同步，实际仍异步执行，事实上任何异步模型，底层最后都是基于回调的。

参考

1. koa - Github
2. koa - 官网
3. Koa - 中文文档
4. wiki - Coroutine
5. wiki - 地球构造
6. nikic - 在PHP中使用协程实现多任务调度(译文)
7. nikic - FastRoute
8. 阮一峰 - Generator 函数的含义与用法
9. 阮一峰 - Thunk 函数的含义和用法
10. 阮一峰 - co 函数库的含义和用法
11. PHP RFC - generator
12. PHP RFC - delegating generator