## 进程?

操作系统当中一个**进程**，就是一个正在运行的程序

Eg php aa.php 相当于创建了一个进程，这个进程会在系统中贮存，申请他自己的内存空间，系统资源，并且运行相应的程序

### 进程核心

进程核心1 内存：

进程核心2 上下文环境：

### 多进程

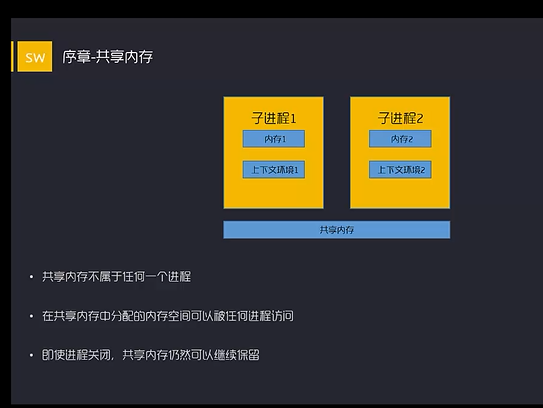
\*\* 复制 **主进程的内存和上下文环境**，子进程中变量修改不会影响父进程中的变量



### 进程间通讯?

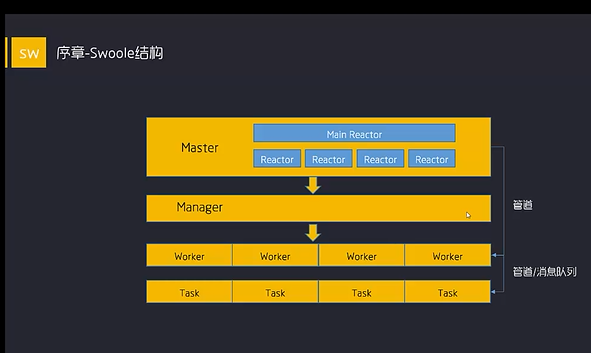
–共享内存（其中一种通讯方式）

不属于任何一个进程，任何进程都可以读取、修改



通过几个命令看一下 系统当中共享内存的分片

Ipcs –m



Web 依赖nginx php-fpm

Fpm 多进程php解析器

当新请求过来以后，fpm会创建一个进程去处理请求，系统开销是创建和销毁进程，导致响应效率不是特别高

## Swoole架构（可以分为三层）

Swlooe采用了和fpm完全不同的架构

### （第一层）Master进程

用于处理swoole核心的事件驱动，拥有若干reactor线程

每个reactor线程用于，每个线程拥有运行着一个epol函数的实例，对事件监听，在reactor中实现

A来自客户端的链接

B本地通讯用的管道

C异步操作用的文件描述符

链接、管道、文件

### （第二层）Manager 管理进程—仅仅只做进程管理和分配

创建管理下级进程（work、taskwork）

### （第三层）工作进程a&b

a Work 进程：主逻辑进程，用于处理来自客户端的请求

b Taskwork进程：异步工作进程，主要处理 耗时较长的同步任务

swoole流程：

a 在master进程当中，当reactror接收到客户端的数据的时候，这些数据会通过管道发送给work进程，由work进程进行处理

b 当work进程需要投递任务到task进程时，也是通过管道来实现数据的投递

**新客户端连接来到时，首先会main reactor线程接收到，然后将这个连接的读写操作的监听注册到reactor线程当中，并通知对应的work进程处理对应的onconnect回调（接收到连接的回调）**

**当客户端发送数据之后，reactor线程会收到这些数据，并通过管道发送给work进程进行处理，work进程如果需要投递任务，work进程会把数据通过管道发送给taskwork进程**

**Taskwork进程处理完之后，返回给work进程，work再通知reactor线程发送数据交回给客户端**

## 进程间通信 – 基于管道

可以配置swoole配置参数来设置，使通信走系统的消息队列

OnTask回调

Finsh函数

Task-常见问题

Task传递数据的大小

数据小于8k 直接通过管道传输

数据大于8k 超过了swoole的buff空间 写入临时文件传递（传文件名）

Task传递对象

可以通过序列化传递一个对象的拷贝；不是对象的引用

Task中对对象的改变，不会反映到Worker进程中

数据库的链接、网络链接对象不可传递

Task的onFinish回调

Task的onFinish回调会发回调用task方法的**worker进程**（耗时较长的同步任务执行完毕后）投递者 – 数据从worker到taskworker再回来，路径是可循的

# Work、Task进程

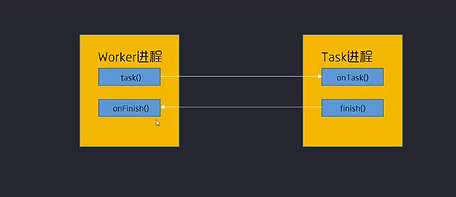
Task进程是swoole当中独立于work进程的工作进程，用于处理一些耗时较长的逻辑。这些逻辑在task进程中处理时，并不影响work进程去处理来自客户端的请求，由此，大大提高了swoole扩展处理并发的能力

如下图，work进程当中，通过调用task()方法发送数据通知到taskwork进程

Taskwork进程会在ontask回调中接收到这些数据进行处理，处理完成之后，调用finsh()函数或者直接return返回消息给work进程

Work进程在onfinsh函数中收到这些消息 并进行处理

Unixsock管道 通讯



====================================================================

新增Test类，有index属性

onReceive方法中new var\_dump并task给taskwork进程中

onTask中修改index属性值为2 打印 data 打印 $this->index

onFinish中修打印$this->test

==============================================================

**Task进程有哪些实际的用处、通过一个模拟的mysql连接池进行示范**

创建一个空server，设置指定的回调函数

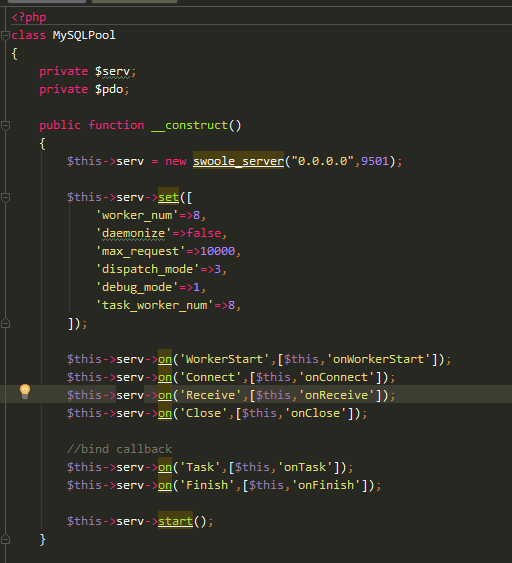
在onWorkerStart函数中创建了一个PDO连接

在onReceive的时候收到来自客户端的请求

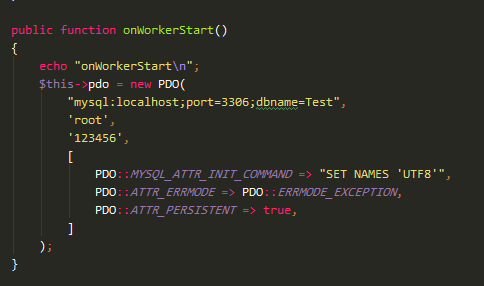
并将请求中对应的sql数据传递给Task，在TaskWorker中进行sql处理

处理完成后，返回数据给worker进程，由worker进程通知给客户端

这样就实现了一个异步的mysql操作



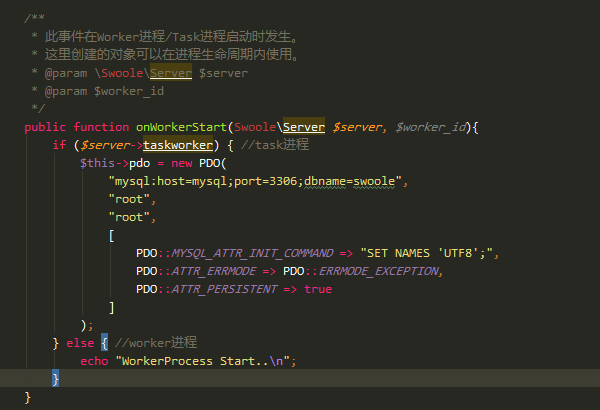
在onWorkStart函数中创建pdo连接



在onReceive的时候，收到来自客户端的请求

并将请求中对应的sql的数据传递给Task进程，在task进程中进行mysql的处理，返回数据给work进程，由work进程通知给客户端，实现异步mysql操作

\*onWorkerStart 并不区分是Worker进程还是TaskerWorker进程,可以用swoole对象的属性$server->taskworker去区分,eg



# Timer定时器

用于取代crontab等类似定时器，实现一个精度更高的毫秒级定时器

在版本1.7时代 swoole的timer进程还是一个线程，线程当中进行了一个循环，循环在一定的间隔进行唤醒，来检查每个定时器是否可以调用

后来发现 a这种方式创建的定时器，数量不能太多，太多了每次遍历的时间会变长，可能导致前面定时器还没执行完，下次定时器已经又响应了；b 线程存在隐患，可能由于某些原因挂掉，导致整个定时器失效

因此，在新版本的swoole中，timer已经使用了epoll的timeout时间进行派递

一个定时器注册一个epoll实例的timeout回调，当这个ep在指定的毫秒内没有处理到指定的事件的时候，就会中断，并且回调一个指定的函数，这个函数会去检查存储在内存当中的所有定时器，他们是否可以被运行

在新的swoole扩展当中，timer进程是基于Reacter线程的epoll实例来运行的。在Taskworker进程当中是没有Reachor线程的，不能调用swoole的异步IO的这些函数，因此在Taskworker进程中使用的是系统定时器也就是timefb相关的函数（自行google）

Timer的实现是基于epoll的timeout机制，

为了提高timer的检索效率，使用堆存放timer（最小堆）存放的索引是每个定时器距离下一次响应剩余的时间，时间越小，离堆顶越近。每次遍历的时候，就从堆顶往下检索。

## 使用：

1 swoole\_timer\_tick() 多次

2 swoole\_timer\_after() 一次性

3

Ep 的回调timeout机制 堆 剩余时间

堆算法

使用

永久定时器 tick

1. After