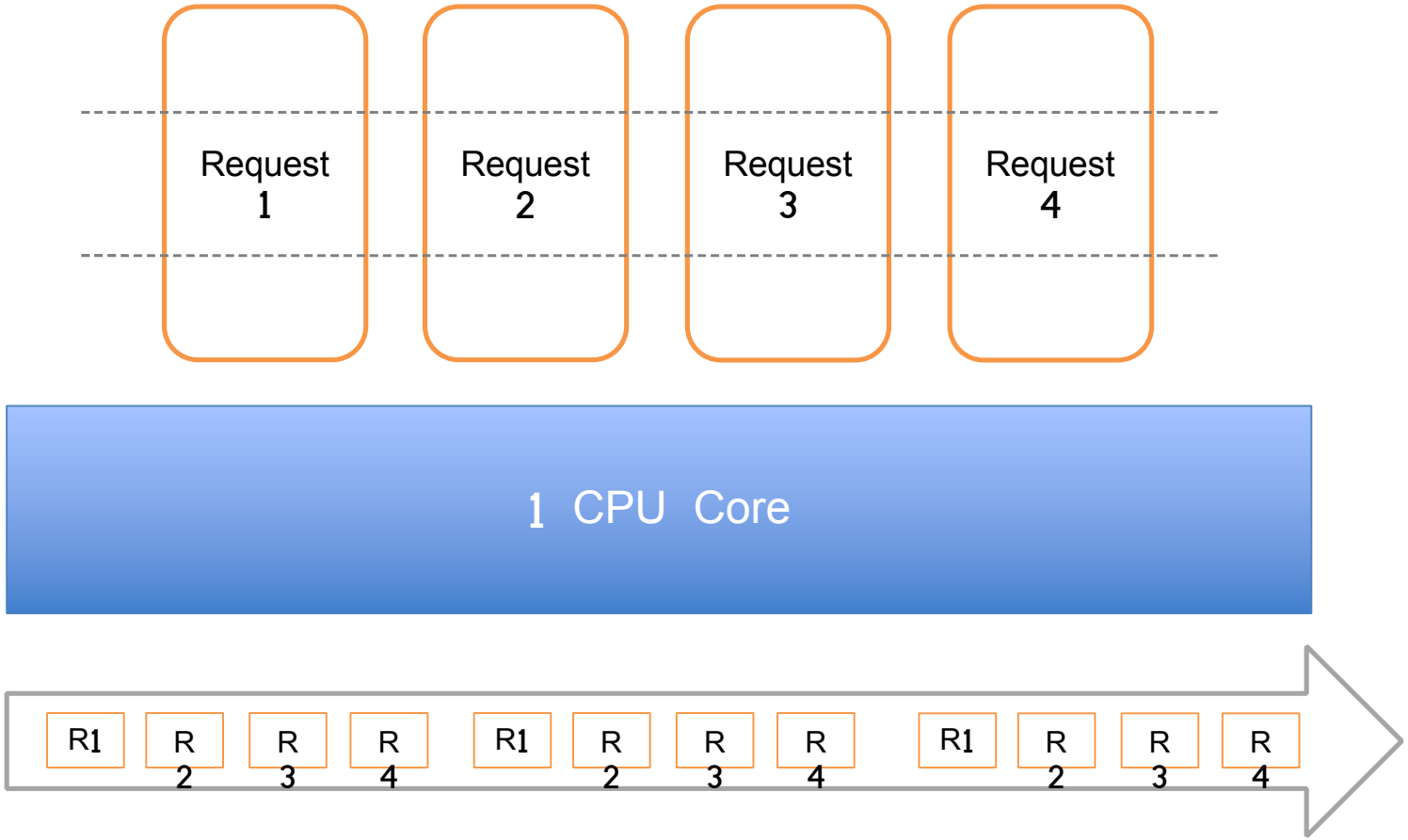
Web并发模型粗浅探讨V3

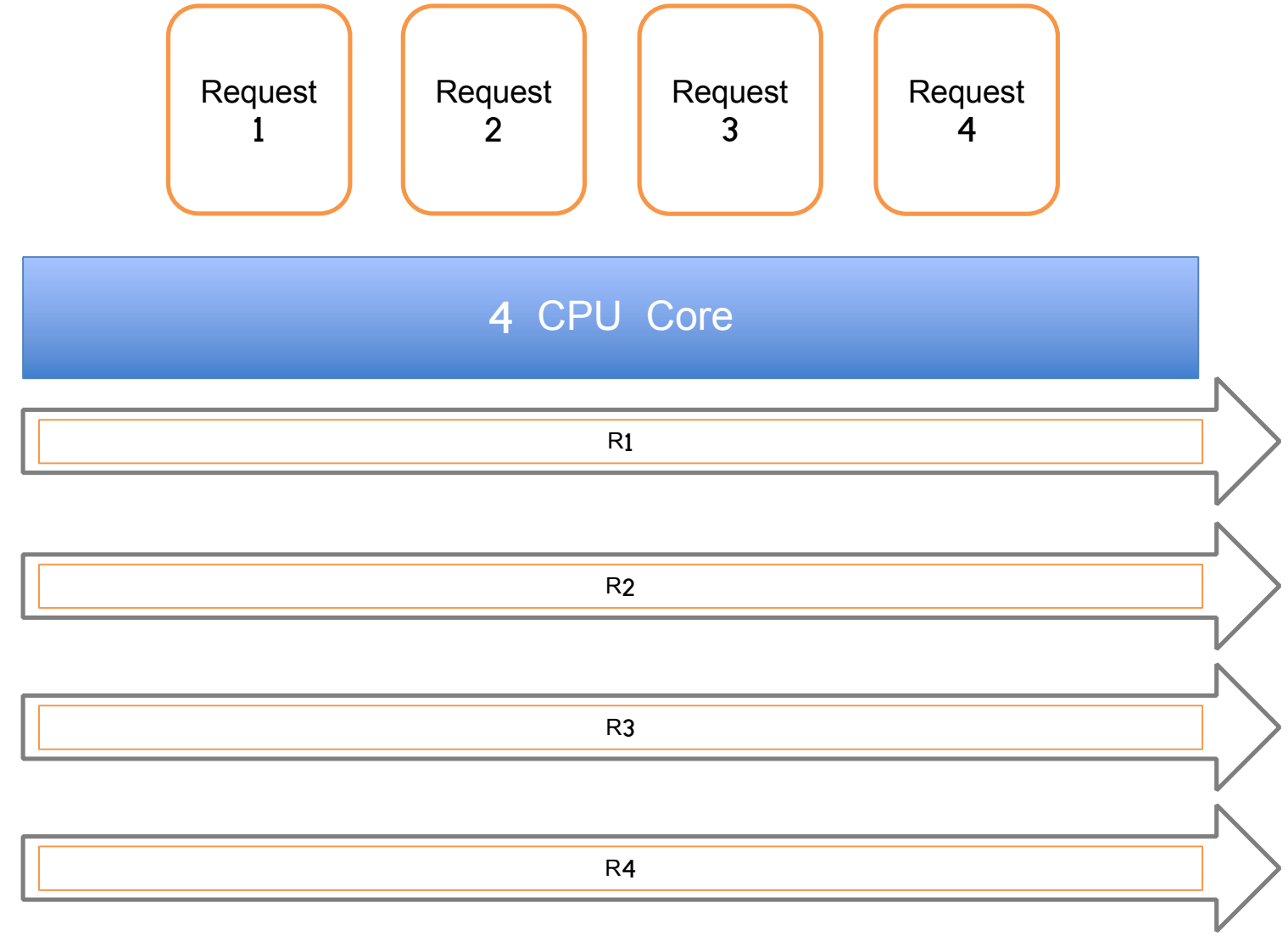
Robbin Fan

基本概念

- 并发concurrency
- 并行parallelism
- 吞吐量throughput



并发：CPU划分时间片，轮流执行每个请求任务，时间片到期后，换到下一个



并行：在多核服务器上，每个CPU内核执行一个任务，是真正的并行

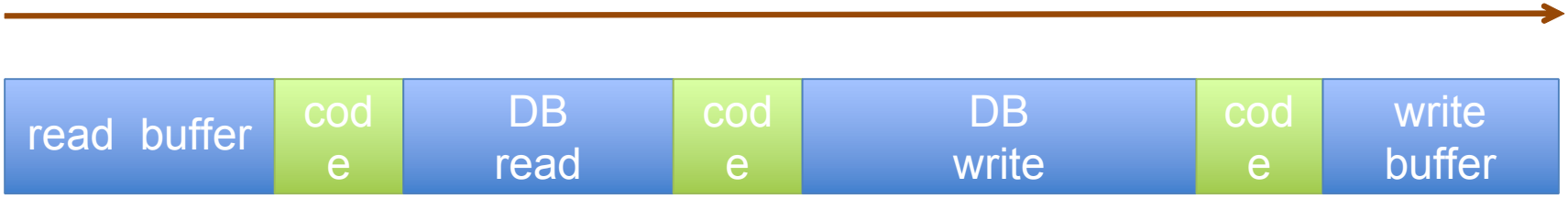
吞吐量

- 单位时间内服务器总的请求处理量
  - 以 request/second 来衡量，如1200rps
  - 每个请求的处理时间latency
  - 服务器处理请求的并发workers
  - 其他因素如GC也会影响吞吐量
- 举例：CSDN new bbs
  - 平均每个请求的latency – 200ms
  - 总共40个workers
  - 理论吞吐量上限 1000/200*40 = 200rps
  - 理论每日处理动态请求上限1700万，目前实际每日处理动态请求270-330万，预估实际处理上限600万

IO类型

- 磁盘文件操作，例如读硬盘文件
- 操作系统调用，例如shell命令
- 网络操作
  - 访问数据库 MySQL，MongoDB，...
  - 访问其他Web服务，发起网络连接
  - 访问缓存服务器 Memcached，Redis

典型IO密集请求



IO操作的延时远远高于CPU时钟周期和内存访问，所以一旦Web请求涉及IO操作，CPU处于wait状态，被浪费了

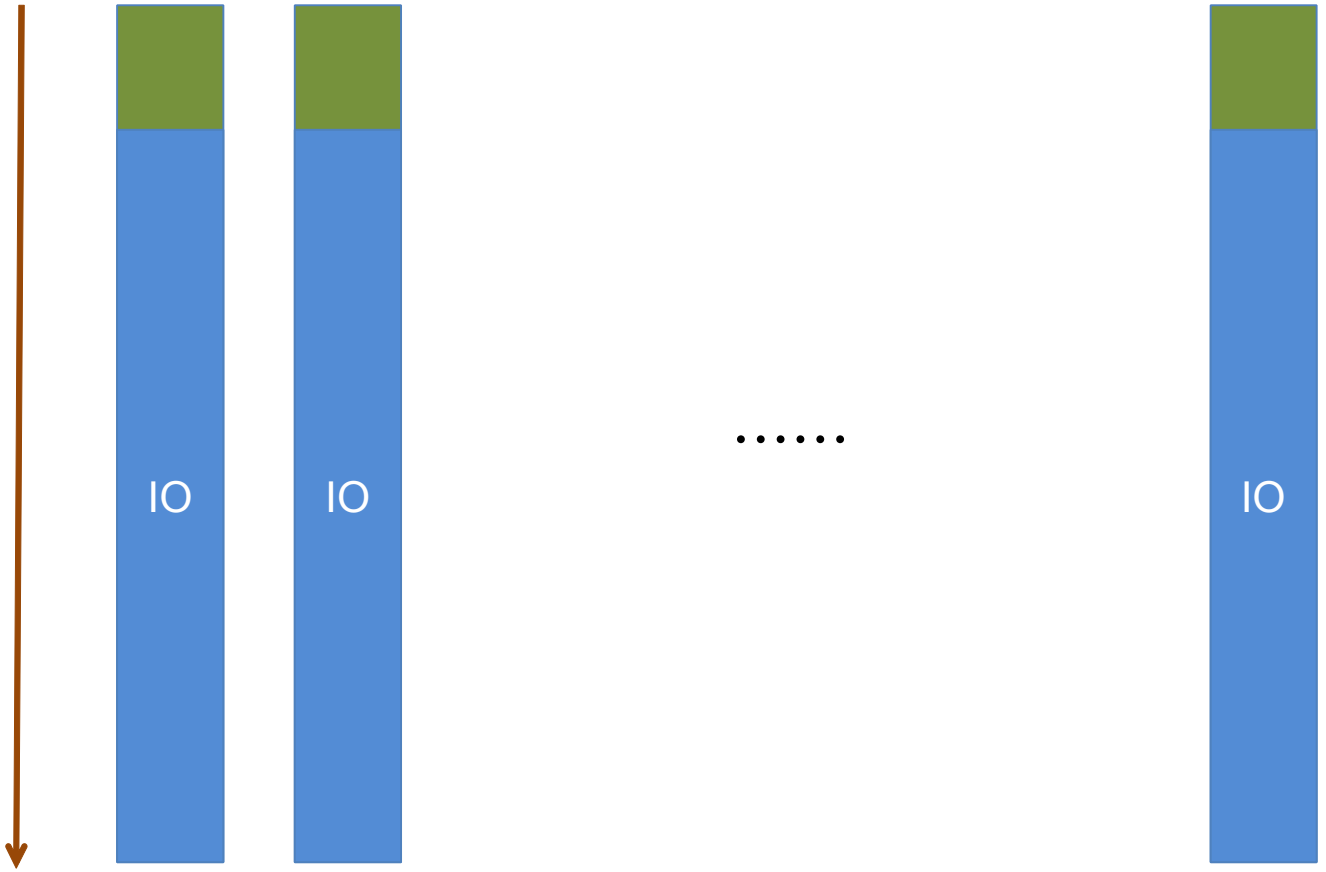
IO密集型并发

- 并发真能提高吞吐量吗？
  - 假设每个请求执行100ms，顺序执行10个请求共需要1s
  - 单核服务器并发处理10个请求，假设平均分配时间片10ms，请求1到请求10将在900ms到1000ms间执行完毕。吞吐量没有任何提高。并发越多，所有请求都变得非常缓慢。(*考虑到任务的场景切换开销，吞吐量还会下降，需要超过1s才能执行完毕*)。
- 大多数Web型应用都是IO密集型
  - 执行请求100ms当中，可能有80ms花在IO上，只有20ms消耗CPU时钟周期，最好情况下，请求1到请求10将在190ms到280ms间执行完毕，吞吐量极大提高。
- IO密集型应用，大部分CPU花在等待IO上了，所以并发可以有效提高系统的吞吐量

顺序执行10个请求，每个请求100ms，总共1s执行完毕

并发执行10个请求，每个请求分配10ms的时间片，仍然1s执行完毕
吞吐量没有提高，每个请求处理时间变长

顺序执行10个请求，每个请求100ms，总共1s执行完毕



并发执行10个请求，每个请求分配10ms的时间片
200ms之后CPU处于空闲状态，190ms到280ms请求全部执行完毕
吞吐量得到极大提高

并发和并行

- 纯CPU密集型的应用
  - 在单核上并发执行多个请求，不能提高吞吐量
    - 由于任务来回场景切换的开销，吞吐量反而会下降
  - 只有多核并行运算，才能有效提高吞吐量
- IO密集型的应用
  - 由于请求过程中，很多时间都是外部IO操作，CPU在wait状态，所以并发执行可以有效提高系统吞吐量

Web并发模型

- multi-process
- multi-thread
- multi-process + multi-thread(GIL)
- event I/O
- coroutine

multi-process

- 常见多进程Web服务端编程模型
  - PHP
  - Python
  - Ruby

多进程的管理

- 多进程并发
  - 每个进程可以并发处理1个请求，并发能力等于进程数量
  - 由操作系统负责进程调度，程序无法控制
  - 可以通过操作系统命令影响进程调度优先nice
- 多进程调度
  - 例如在一台4核服务器上，运行10个PHP进程。由操作系统负责给某个PHP进程分配某个CPU内核的时间片，实现10个并发处理

多进程优点

- 并发模型非常简单，由操作系统调度运行稳定强壮
- 非常容易管理
  - 很容易通过操作系统方便的监控，例如每个进程CPU，内存变化状况，甚至可以观测到进程处理什么Web请求
  - 很容易通过操作系统管理进程，例如可以通过给进程发送signal，实现各种管理：unicorn
- 隔离性非常好
  - 一个进程崩溃不会影响其他进程
  - 某进程出现问题的时候，只要杀掉它重启即可，不影响整体服务的可用性
  - 很容易实现在线热部署和无缝升级
- 代码兼容性极好，不必考虑线程安全问题
- 多进程可以有效利用多核CPU，实现并行处理

多进程的监控手段

- 监控进程CPU top -p pid
  - 简单处理甚至可以查看进程处理的URL请求
- 监控进程的IO iotop -p pid
- 监控进程的物理内存使用 ps，/proc

```
229m 193m 3140 S 11.3  2.5   0:27.37 /wiki/topic/86387*
228m 191m 3176 S  7.0  2.4   0:09.74 /wiki/topic/157969*
226m 189m 3212 S  2.3  2.4   0:21.41 /topic/92370*
224m 188m 3152 S  7.3  2.4   0:18.49 /*
224m 188m 3140 S 14.6  2.4   0:11.36 /rss/news*
224m 188m 3140 S  5.0  2.4   0:11.93 /topic/199362*
221m 186m 3152 S  2.7  2.4   0:28.56 /forums/45/topics/284097/posts*
221m 185m 3144 S  3.0  2.4   0:09.90 /admin/blogs/284179*
220m 184m 3140 S  4.3  2.4   0:14.29 /post/769384*
219m 184m 3148 S  3.7  2.3   0:10.08 /admin/mygroups*
219m 184m 3148 S  4.3  2.3   0:23.27 /topic/259458*
218m 183m 3140 S  4.0  2.3   0:11.90 /topic/1925*
217m 181m 3140 S  3.7  2.3   0:12.15 /rss/board/Job*
216m 180m 3148 S  3.0  2.3   0:14.72 /post/683400*
216m 180m 3148 S  6.0  2.3   0:09.71 /*
215m 180m 3140 S  4.0  2.3   0:10.56 /topic/250780*
215m 180m 3140 S  5.7  2.3   0:10.49 /blog/profile*
213m 177m 3140 S  1.7  2.3   0:09.36 /topic/284150*
211m 175m 3140 S 15.0  2.2   0:10.89 /blog/234786*
```

多进程缺点

- 内存消耗大户
  - 每个独立进程都需要加载完整的应用环境，内存消耗超大。(*COW模式可以缓解这个问题*)
  - 例如每个Rails进程物理内存占用为150MB，20个workers，则需要3GB物理内存。
- CPU消耗偏高
  - 多进程并发，需要CPU内核在多个进程间频繁切换，而进程的场景切换(context switch)是非常昂贵的，需要大量的内存换页操作。
- 很低的I/O并发处理能力

多进程的低IO并发问题

- 多进程的并发能力非常有限
  - 每个进程只能并发处理1个请求
  - 单台服务器启动的进程数有限，并发处理能力无法有效提高
    - Rails进程消耗内存很大，单台服务器一般启动30个Rails实例
    - PHP FastCGI进程非常轻量级，但单台服务器一般启动64个
- 只适合处理短请求，不适合处理长请求
  - 每个请求都能在很短时间内执行完毕，因而不会造成进程被长期阻塞
  - 一旦某个操作特别是IO操作阻塞，就会造成进程阻塞，当大面积IO操作阻塞发生，服务器就无法响应了
  - 对于无法预知的外部IO操作，应用代码必须设置timeout参数，以防进程阻塞

缓解多进程低IO并发问题

- 用nginx做前端Web Server
  - 适当增大proxy buffer size，避免多进程request/response buffer IO开销
  - 使用X-sendfile，避免多进程读取大文件IO开销
- 凡IO操作都要设置timeout
  - 避免无法预知的IO挂起造成进程阻塞
- 编写智能防火墙代码
  - 防止劣质爬虫和其他恶意的DOS攻击行为
- 长请求和短请求分离开，不要放在一起

多进程spawn

- 静态spawn
  - 适合大型应用，稳定处理请求
  - FastCGI(PHP/Ruby) / WSGI(Python)
  - Unicorn(Ruby)
- 动态spawn
  - 适合较小型应用，请求伸缩变化大，节省资源
  - Apache module(PHP)
  - Passenger(Ruby/Python)

PHP的轻量级进程

- 轻量级进程
  - 无虚拟机，无GC，简单解析器，内存管理简单
  - 每请求的内存管理模型：每个请求都需要初始化整个框架，请求执行完毕释放所有内存
  - 每个进程初始化一般10MB，远远小于Python/Ruby

PHP的优点

- 每个进程占用内存少，可以多启动一些进程数量，并发处理能力高于Python/Ruby
- 基本没有内存泄露的烦恼
  - 即使应用代码有内存泄露问题，每个请求执行完毕，所有内存对象全部释放，基本不会造成严重后果

PHP的缺点

- 每请求的内存管理模型造成PHP性能低下
  - 每个请求都需要初始化整个应用代码，造成bootstrap速度很缓慢
  - 重型PHP框架性能不可避免的低下：例如Drupal性能尤其差，必须依赖多种缓存机制缓解性能问题
  - PHP社区多采用轻量级框架缓解性能问题
- 由于每请求都彻底释放内存，无法实现进程内跨请求共享资源
  - 通用数据库连接池
  - 内存字典表
  - 其他昂贵的需要耗时建立的共享资源

PHP的大型应用

- PHP在大型应用的场景
  - 由于PHP的内存模型限制，很多大型应用在较早期就会不可避免的遇到PHP的瓶颈
  - PHP的瓶颈无法解决，必须调整架构，因此在早期就会引入中间应用层(C++，Java)，让PHP退化为View层模板语言
  - 案例：taobao，facebook，yahoo

multi-thread

- 常见多线程模型(1:1)
  - Java VM / .net CLR
  - 1 native thread : 1 process thread
  - 在一个重量级进程当中启动多个线程并发处理请求
- 多线程并发
  - 每个线程可以并发处理1个请求，并发能力取决于线程数量
  - 线程的调度由VM负责，可以通过编程控制

多线程优点

- 多线程并发内存消耗比较少
  - 每个线程需要一个thread stack保存线程场景，thread stack一般只需要十几到几十KB内存，不像多进程，每个进程需要加载完整的应用环境，需要分配十几到上百MB内存。
  - 线程可以共享资源，特别是可以共享整个应用环境，不必像多进程每个进程要加载应用环境。
- 多线程并发CPU消耗比较小
  - 线程的场景切换开销小于进程的场景切换
- 很容易创建和高效利用共享资源
  - 数据库线程池
  - 字典表，进程内缓存.....
- IO并发能力很高
  - Java VM可以轻松维护几百个并发线程的线程切换开销，远高于多进程单服务器上几十个并发的处理能力
- 可有效利用多核CPU，实现并行运算

多线程的缺点

- VM的内存管理要求超高
  - 对内存管理要求非常高，应用代码稍不注意，就会产生OOM(out of memory)，需要应用代码长期和内存泄露做斗争
  - GC的策略会影响多线程并发能力和系统吞吐量，需要对GC策略和调优有很好的经验
  - 在大内存服务器上的物理内存利用率问题
    - VM内存堆不宜过大，一般2GB为宜。过大的内存堆会造成GC效率下降。在物理内存很多的服务器上为了有效利用更多内存，不得不跑多个Java VM，增加了复杂度。
- 对共享资源的操作
  - 对共享资源的操作要非常小心，特别是修改共享资源需要加锁操作，很容易引发死锁问题
- 应用代码和第三方库都必须是线程安全的
  - 使用了非线程安全的库会造成各种潜在难以排查的问题
- 单进程多线程模型不方便通过操作系统管理
  - 一旦出现线程死锁或者线程阻塞很容易导致整个VM进程挂起失去响应，隔离性很差

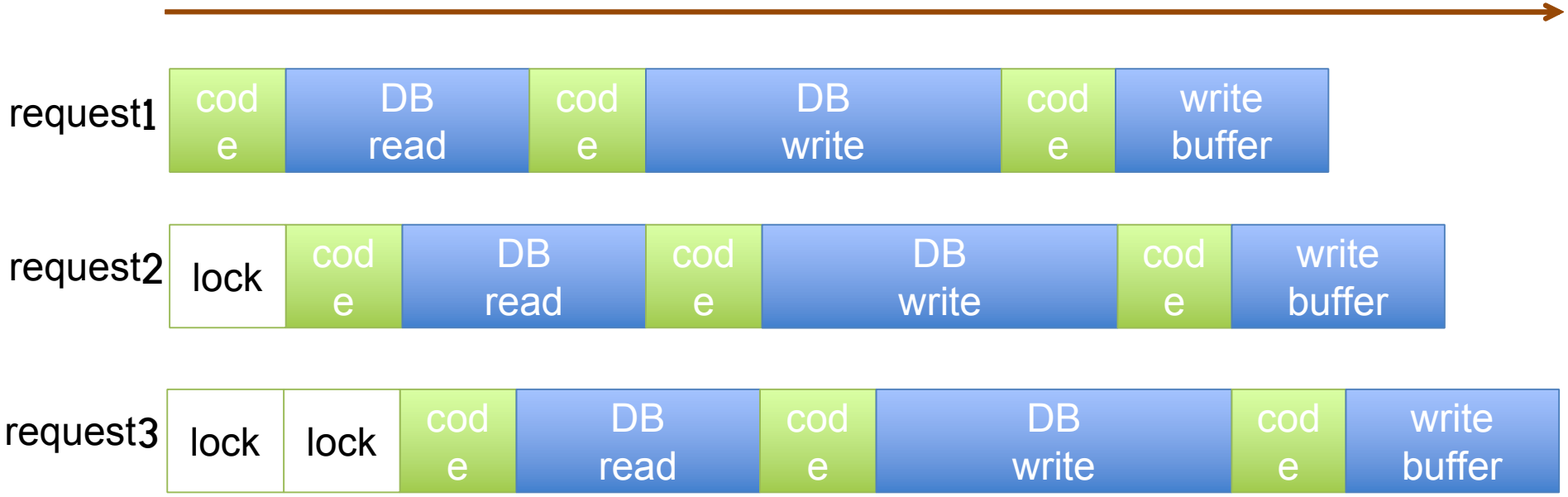
multi-thread with GIL

- Ruby MRI / Python
  - Global Interpreter Lock：有限制的并发
  - 执行纯Ruby code的时候@mutex.lock，只有一个线程并发，其他线程等待，无法实现并发
  - IO操作或者操作系统调用，释放锁，多线程IO并发
  - 由于加锁，无法利用多核，只能使用1个CPU内核，因而无法实现多核并行运算

Why multi-thread(GIL)

- 提供简化的并发策略
  - 对CPU密集型运算，并发不能提高吞吐量：加锁，禁止并发
  - 对IO密集型运算，并发可以有效提高吞吐量：解锁，允许多线程并发
- 性能
  - 对CPU密集型运算，多线程并发由于线程场景切换带来的开销，吞吐量要差于单进程顺序执行
- 兼容性
  - 加锁可以保证代码和库的兼容性
- GC和内存管理
  - Ruby和Python的GC内存管理无法和Java VM相提并论，严重依赖多线程会带来非常严重的内存管理问题

MRI multi-thread IO Concurrency



rainbows!

multi-process + multi-thread(GIL)

- why multi-thread(GIL) + multi-process
  - 由于GIL，多线程只能跑在1个CPU内核上，无法有效利用多核CPU，跑多个进程可以有效利用多核
    - 一般进程数略多于服务器CPU内核数
  - 一个进程不宜跑过多线程，否则会引发严重的GC内存管理问题
- pros and cons
  - 内存消耗低于单纯的多进程并发
  - 非常有效的提高了IO并发处理能力
  - IO库和操作系统调用库必须保证线程安全

Rainbows! configuration

```
Rainbows! do
    name = 'website'
    use :ThreadPool
    worker_processes 6
    worker_connections 32
    listen "unix:/var/run/#{name}.sock"
    pid "/var/run/#{name}.pid"
    stderr_path "/var/log/#{name}.log"
    stdout_path "/var/log/#{name}.log"
end
```

event IO

- 常见event IO编程模型
  - Nginx / Lighttpd
  - Ruby EventMachine / Python Twisted
  - node.js
- event IO原理
  - 单进程单线程
  - 内部维护一个事件队列
  - 每个请求切成多个事件
    - 每个IO调用切成一个事件
    - 编程调用process.next_Tick()方法切分事件
  - 单进程顺序从事件队列当中取出每个事件执行下去

event IO并发示意

event IO的优点

- 惊人的IO并发处理能力
  - nginx单机可以处理50K以上的HTTP并发连接
  - node.js单机可以处理几千上万个HTTP并发连接
- 极少的内存消耗
  - 单一进程单一线程，无场景切换无需保存场景
- CPU消耗偏低
  - 无进程或者线程场景切换的开销

event IO的缺点

- 必须使用异步编程
  - 异步编程是一种原始的编程方式
  - 代码量和复杂度都会有很大的增加，提高了编程的难度，以及开发和维护成本
  - 复杂的业务逻辑(例如工作流业务)会造成代码迅速膨胀，极难维护
  - 异步事件流使得异常处理和调试有很大困难
- CPU密集型的运算会阻塞住整个进程
  - 需要通过编程，将密集型的任务拆分为多个事件
- 所有IO操作必须使用异步库
  - 一旦不小心使用同步IO操作，会造成整个进程阻塞，库的兼容性必须非常小心
- 只能跑在1个CPU内核上，无法有效利用多核并行运算
  - 运行多个进程来利用多核CPU

```
var copy = function(src, dst) {
    var fd_src = fs.open(src, "r");
    var fd_dst = fs.open(dst, "w");
    while (true) {
        var buffer_size = 4096;
        var buffer = new Buffer(buffer_size);
        var bytes_read = fs.read(fd_src, buffer, 0, buffer_size);
        if (bytes_read > 0) {
            var write_at = 0;
            while (write_at < bytes_read) {
                write_at += fs.write(fd_dst, buffer, write_at, bytes_read - write_at);
            }
        } else {
            break;
        }
    }
};
```

读写文件的代码片段
常见同步代码

```
var copy = function(src, dst) {
    fs.open(src, "r", undefined, function(err, fd_src) {
        fs.open(dst, "w", undefined, function(err, fd_dst) {
            var copy_rec = function(position) {
                var buffer_size = 4096;
                var buffer = new Buffer(4096);
                fs.read(fd_src, buffer, 0, buffer_size, position, function(err, bytes_read) {
                    if (bytes_read > 0) {
                        var write_rec = function(write_at) {
                            if (write_at < bytes_read) {
                                fs.write(fd_dst, buffer, 0, bytes_read, position, function(err, written) {
                                    if (written > 0) {
                                        write_rec(write_at + written);
                                    }
                                });
                            } else {
                                copy_rec(position + bytes_read);
                            }
                        };
                        write_rec(0);
                    }
                });
            };
            copy_rec(0);
        });
    });
};
```

node.js的异步代码(尚未处理异常)

node.js的异步编程

- IO操作都需要异步回调，稍微复杂的代码有十几层嵌套，可阅读性，代码维护性很差
- 缓解异步回调嵌套的努力
  - eventproxy
  - async.js
  - wind.js
  - 把嵌套层次平铺展开，并未从根本上消除问题
- 用coroutine解决异步编程问题

coroutine

- 原生支持coroutine的编程语言
  - Go goroutine
  - Erlang process
  - Lua coroutine
  - Scala(Java VM) actor
  - Ruby Fiber
- 库或者第三方实现
  - Stackless Python
  - node-fibers

coroutine原理

- 在单个线程上运行多个纤程，每个纤程维护1个context
- 纤程非常轻量级，单个线程可以轻易维护几万个纤程
- 纤程调度依赖于应用程序框架
- 纤程切换
  - 必须自己编程来实现
  - 一般应用层代码不需要编程，框架层实现纤程调度
- 纤程本质上是基于event IO之上的高级封装，但消除了event IO原始的异步编程复杂度

获取纤程控制权

异步调用

待回调完成后恢复现场

放弃纤程控制权

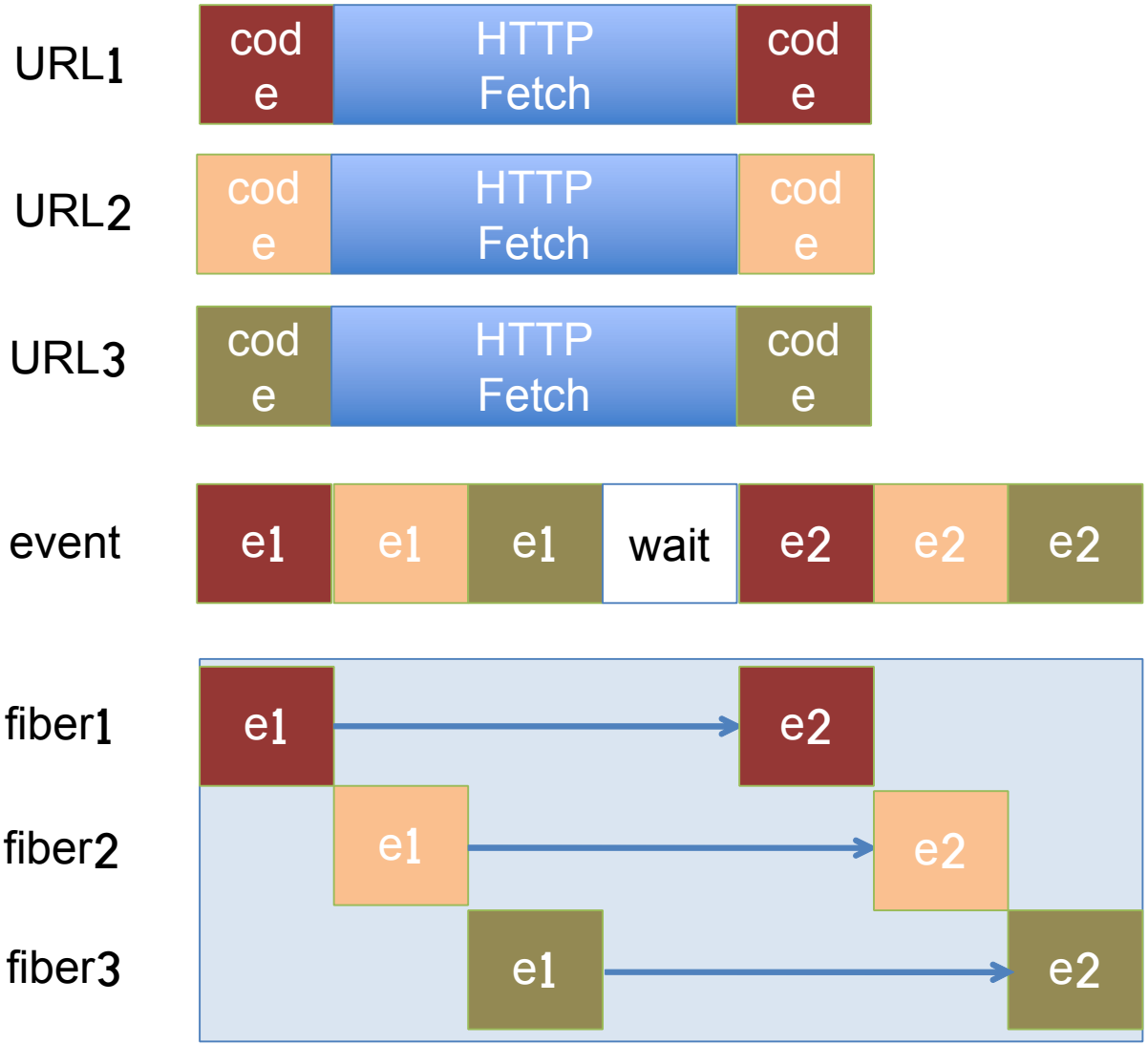
```ruby
def async_fetch(url)
  f = Fiber.current
  http = EventMachine::HttpRequest.new(url).get :timeout => 10
  http.callback { f.resume(http) }
  http.errback  { f.resume(http) }
  Fiber.yield
  http
end

EventMachine.run do
  Fiber.new{
    puts "Setting up HTTP request #1"
    data = async_fetch('http://0.0.0.0/')
    puts "Fetched page #1: #{data.response_header.status}"

    puts "Setting up HTTP request #2"
    data = async_fetch('http://www.yahoo.com/')
    puts "Fetched page #2: #{data.response_header.status}"

    puts "Setting up HTTP request #3"
    data = async_fetch('http://non-existing.domain/')
    puts "Fetched page #3: #{data.response_header.status}"
    EventMachine.stop
  }.resume
end
```

Ruby Fiber Example



单一线程通过程序调度了3个纤程并发，底层仍然是event IO驱动
但是有3个清晰的并发执行体，仍然是同步并发编程风格，但实现了异步驱动

coroutine的优点

- 支持极高的IO并发，和event IO基本相当
- 纤程的创建和切换的系统开销非常小，CPU和内存消耗都很小
- 编程方式和常见的同步编程基本一致，是event IO的高级封装形式

coroutine的缺点

- 纤程运行在单线程上，无法有效利用多核实现并行运算
  - 通过启动多个进程或者多个线程来利用多核CPU
- CPU密集型的运算会阻塞住整个进程
  - 通过编程，将密集型的任务拆分为多步
- 所有IO操作必须使用异步库
  - 一旦不小心使用同步IO操作，会造成整个进程阻塞，库的兼容性必须非常小心

```
var call_cc = function(cont) {
    var fiber = Fiber.current;
    cont(function (a) {
        fiber.run(a);
    });
    return Fiber["yield"]();
};

var efs = {
    open: function (name, mode) {
        return call_cc(function (continuation) {
            fs.open(name, mode, undefined, function(err, fd_src) {
                continuation(fd_src);
            });
        });
    },
    read: function (fd, buffer, offset, length) {
        return call_cc(function (continuation) {
            fs.read(fd, buffer, offset, length, null, function(err, bytes_read) {
                continuation(bytes_read);
            });
        });
    },
    write: function (fd, buffer, offset, length) {
        return call_cc(function (continuation) {
            fs.write(fd, buffer, offset, length, null, function(err, written) {
                continuation(written);
            });
        });
    }
};

var copy = function(src, dst) {
    var fd_src = efs.open(src, "r");
    var fd_dst = efs.open(dst, "w");
    while (true) {
        var buffer_size = 4096;
        var buffer = new Buffer(buffer_size);
        var bytes_read = efs.read(fd_src, buffer, 0, buffer_size);
        if (bytes_read > 0) {
            var write_at = 0;
            while (write_at < bytes_read) {
                write_at += efs.write(fd_dst, buffer, write_at, bytes_read - write_at);
            }
        } else {
            break;
        }
    }
};

Fiber(function () {
    copy("src", "dst");
}).run();
```

用node-fibers实现
node.js的coroutine编程

Web并发模型对比列表

	多进程	多线程	多进程＋多线程	event IO	coroutine
CPU消耗	高	低	比较高	很低	很低
内存消耗	很高	稍高	比较高	非常低	非常低
IO并发	很低	很高	高	极高	极高
并行运算	支持	支持	支持	不支持	不支持
编程难度	很低	高	低	很高	很低
库和框架兼容性	要求很低	要求高	要求高	要求很高	要求很高
内存管理	要求很低	要求极高	要求高	要求不高	要求不高
调度方式	操作系统	虚拟机	操作系统和虚拟机	事件队列和编程	编程
监控和管理难度	很低	很高	低	高	高

Web并发模型适用场景

- 应用场景CPU密集运算多，应使用多进程和多线程
  - CPU密集：大量字符串运算，正则表达式匹配，大量内存对象创建和操作等
- 应用场景IO密集并发，应使用event IO和coroutine
  - IO密集：大量IO操作(发起HTTP连接，数据库访问)，维持长连接等

Web并发应用场景

- website
- web api service
- real-time / long-poll

website

- 场景特点
  - 业务逻辑比较复杂
  - 功能点多，页面处理逻辑多，代码量比较大
  - IO并发要求不高，一般几十并发workers足够
- 常用编程语言
  - Java / .net
  - PHP
  - Python / Ruby

web service

- 场景特点
  - 业务逻辑比较简单，功能点较少
  - 无页面逻辑，输出json/xml数据
  - IO并发要求比较高，往往要求几百上千并发
- 常用编程语言
  - Java / .net
  - node.js
  - Go
  - Python / Ruby (multithread or Fiber)

real-time

- 场景特点
  - 业务逻辑很简单，功能单一
  - 无页面逻辑，服务器维持长连接，持续push数据json/xml到客户端
  - IO并发要求极高，单机往往维持几千上万长连接
- 常用编程语言
  - node.js
  - Go

Web应用场景对比列表

	Website	Web Service	Read-time
业务逻辑和功能	复杂，功能多	简单，功能少	单一，功能极少
输出数据	HTML	json/xml	json/xml
IO并发要求	很低	高	极高
常用编程语言	PHP，Python，Ruby	Java，.net	Go，node.js
个人偏好	Ruby	Ruby/Java	node.js

选择编程语言

- 编程语言特性，适用场景
- 编程语言的难易度，生产效率
- 库和框架的支持程度
- 编程语言社区的活跃度和发展前景
- 具体应用的使用场景
- 团队的技术背景和招募难度

Ruby

- 支持多种Web并发模式
  - Ruby社区当前主流并发模式是多进程模型
  - 随着Rails4.0即将推出，默认设置多线程安全模式，社区逐渐向多线程并发发展
  - 在Web service场景也有优秀的基于Fiber的框架，可支持极高的IO并发
    - Goliath (em-synchrony)
    - sinatra-synchrony
- MRI虚拟机性能很差
  - MRI的代码结构不清晰，改进难度大
  - core team团队比较小，进展比较缓慢
  - GC效率差，和主流商业虚拟机Java VM，V8有巨大差距，是Ruby应用迈向顶级规模的主要障碍，例如Twitter

Matz谈Twitter迁移

在我看来，在网站所提供的服务还没有完全成型的时候，最重要的是能够对需求的变化做出快速的反应，这个时候就需要Ruby这样灵活性比较高的语言；

而在网站获得成功之后，遇到了设计瓶颈，用一种新的语言，比如Scala，来编写一个新的架构，以节约一定的资源，我认为这也是很好的一个结果。

Twitter转向Scala还只是在其核心部分，而在Web前端和一些内部工具上还有很多地方在用Ruby。

其实，上个月我还去拜访了一下Twitter，跟他们的工程师进行了一些交流，Ruby还是用得很多的。

Ruby应用场景

- website
  - 大型website极适合用rails，提供了应有尽有的基础设施
  - rails框架比较重，性能瓶颈往往在View层页面模板渲染和Restful的routes上，对于小型website，可以使用轻量级框架sinatra
- web service
  - sinatra (multi-thread + multi-process)
  - sinatra-synchrony(fiber)
  - Goliath(fiber)
- real-time
  - Goliath(fiber)

node.js

- 特点
  - 纯event IO并发，极其适合超高IO并发场景
  - JavaScript V8引擎的性能非常好
  - 仅适合IO密集型并发，不适合CPU密集并发
  - 不适合开发业务逻辑过于复杂的应用
  - 异步编程复杂度和调试难度偏高
- 场景
  - 网页游戏服务器端
  - 网页长连接服务器端
- 问题
  - event IO异步编程是一种相当原始的编程方式，coroutine提供了更高抽象层次的编程方式，但是node社区似乎并不热衷于coroutine，仍然陶醉于异步事件回调编程(why?)

编程语言并发模型对比列表

	多进程	多线程	多进程＋多线程	event IO	coroutine
PHP	支持(主流)				
Python	支持(主流)		支持	支持	第三方支持
Ruby	支持(主流)		支持	支持	支持
Java		支持(主流)		支持	支持(scala)
.net		支持(主流)		支持	
node.js				支持(主流)	第三方支持
Go					支持(主流)
Lua					支持(主流)

参考资料

ruby multithread and GIL discussion

1. [multithreaded-rails-is-generally-better](http://www.unlimitednovelty.com/2010/08/multithreaded-rails-is-generally-better.html)
2. [threads-in-ruby-enough-already](http://yehudakatz.com/2010/08/14/threads-in-ruby-enough-already/)
3. [about-concurrency-and-the-gil](http://merbist.com/2011/10/03/about-concurrency-and-the-gil/)
4. [the-ruby-global-interpreter-lock](http://ablogaboutcode.com/2012/02/06/the-ruby-global-interpreter-lock/)

node.js Call/CC

1. [callcc-and-node-js](http://blog.kghost.info/index.php/2011/10/callcc-and-node-js/)

Non blocking ruby web framework: Goliath

1. [goliath-non-blocking-web-framework](http://www.igvita.com/2011/03/08/goliath-non-blocking-ruby-19-web-server/)

Golang

1. [Go语言并发编程之