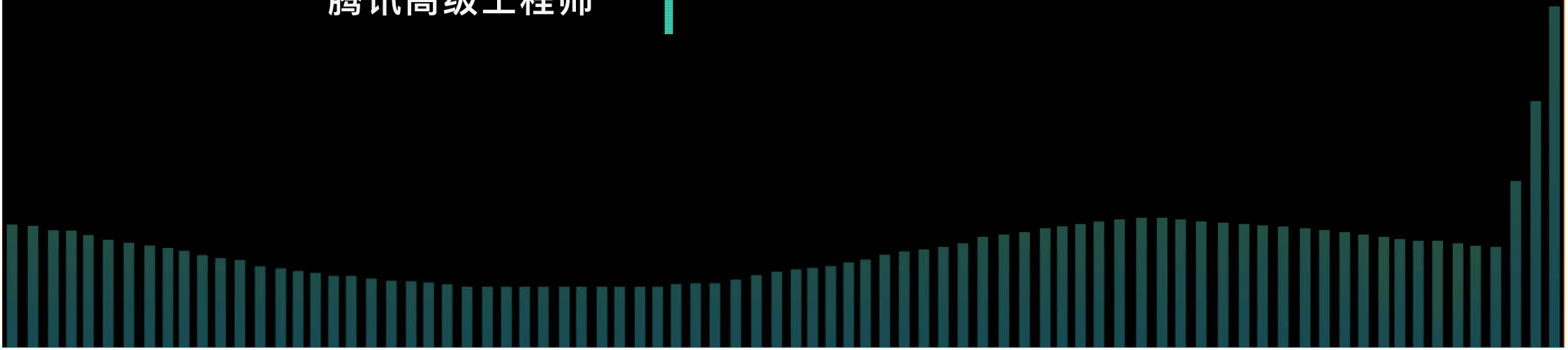
C++ Summit 2020

C++高性能大规模服务器开发实践

吴锐

腾讯高级工程师



01 Lego简介

02 传统Web框架

03 Lego架构实现

04 未来展望

00. Who am I

姓名：吴锐

英文名： royrwu

经历：加入腾讯后一直从事与CDN架构和运营相关的工作。参与过高性能服务器，CDN内核和海量运营相关的开发工作。

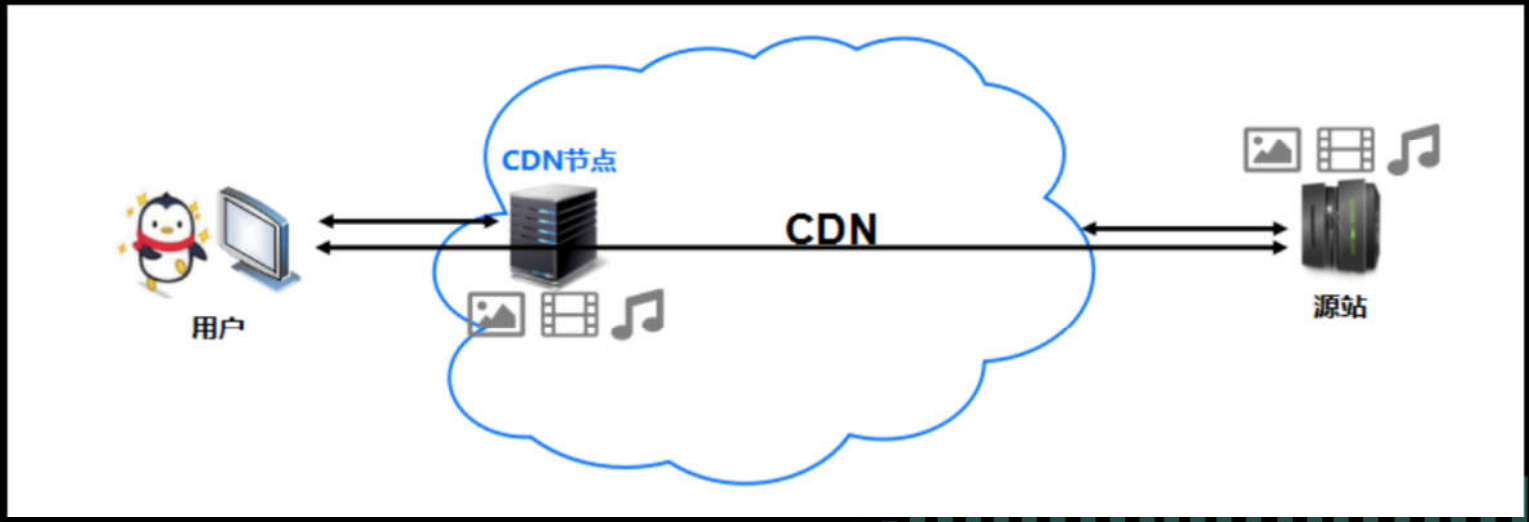
目前是腾讯CDN服务器开发技术负责人。

长期专注于CDN相关服务器，内核和网络等相关技术的研发和实践。

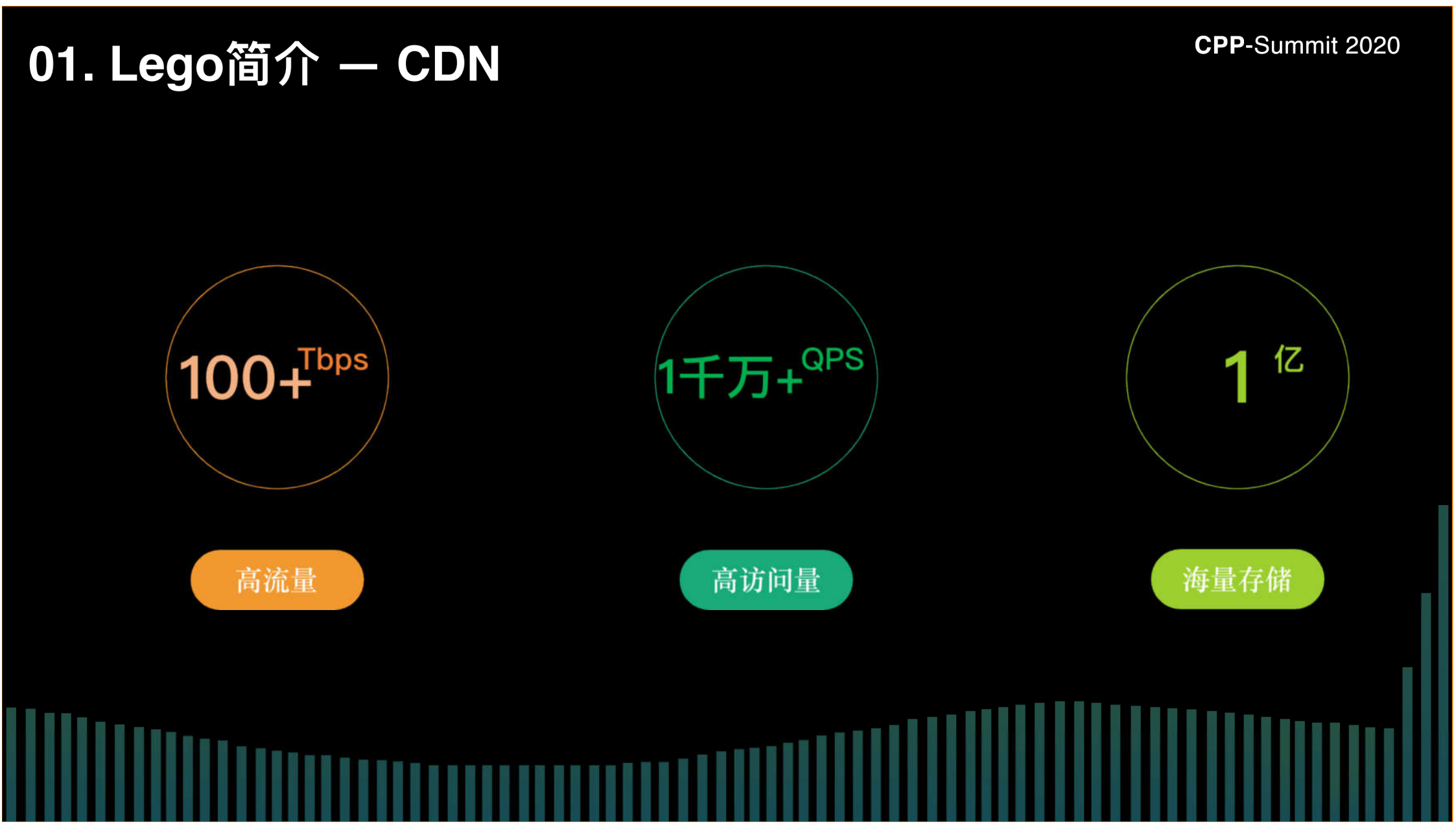
01. Lego简介 — CDN

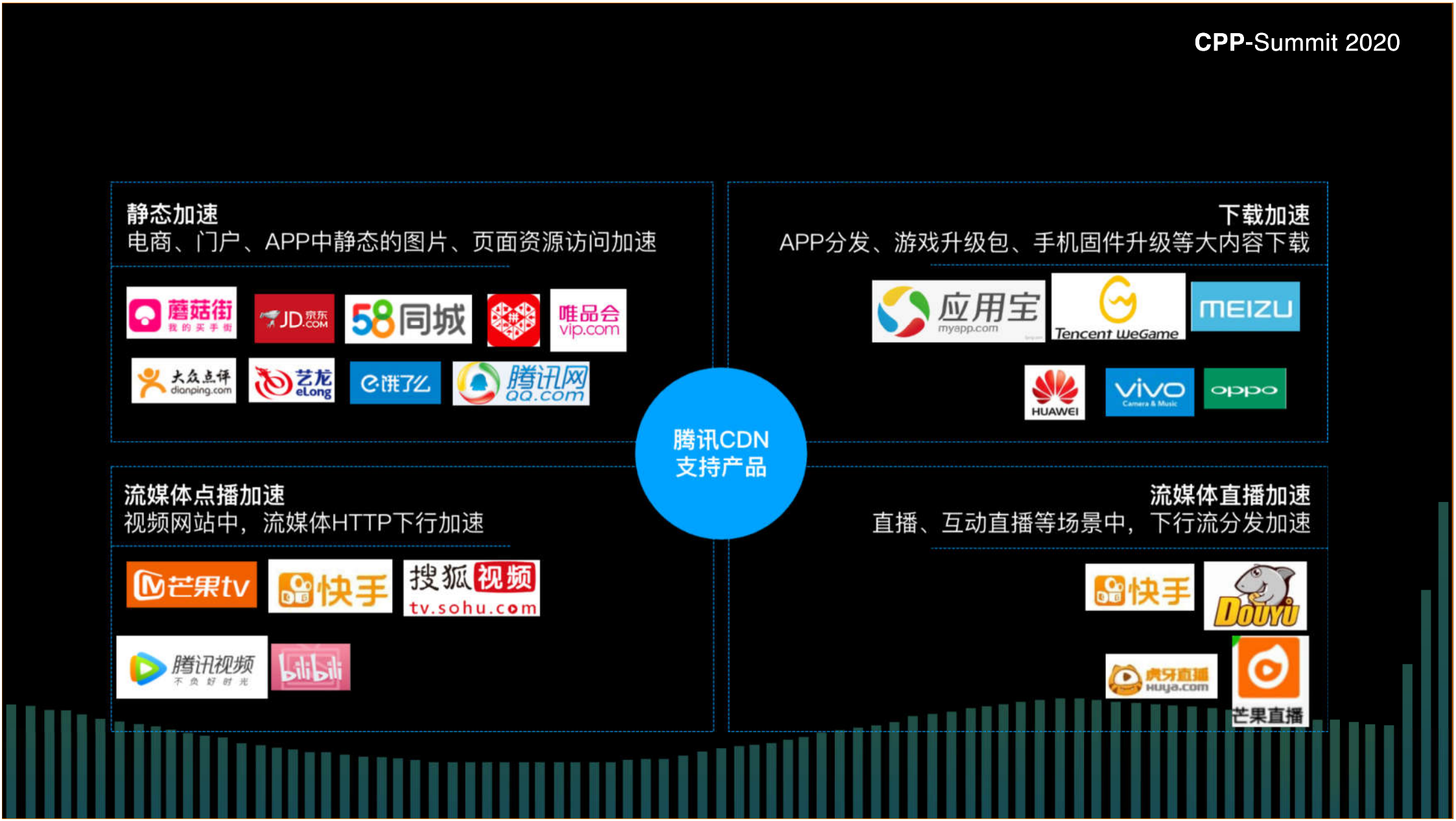
CDN，Content Delivery Network，全称内容分发网络。就是网络世界的快递公司，有着全国，全球的快递网络，网点。将快递用最快的速度从“卖家”（源站）发送到“买家”（客户端）手上。区别在于一份快递会重复发送的全国不同的地方，并且可以从“网点“（边缘节点）直接发货。

根据运营商，区域，负载，链路情况等因素提供最优节点给客户端就近访问。



01. Lego简介 — CDN




静态加速
电商、门户、APP中静态的图片、页面资源访问加速
蘑菇街
我的买手街
JD.京东.COM
58同城
唯品会
vip.com
大众点评
dianping.com
艺龙
eLong
e饿了么
腾讯网
qq.com
下载加速
APP分发、游戏升级包、手机固件升级等大内容下载
应用宝
myapp.com
Tencent WeGame
MEIZU
HUAWEI
vivo
Camera & Music
oppo
腾讯CDN
支持产品
流媒体点播加速
视频网站中，流媒体HTTP下行加速
芒果tv
快手
搜狐视频
tv.sohu.com
腾讯视频
不负好时光
bilibili
流媒体直播加速
直播、互动直播等场景中，下行流分发加速
快手
DOUYU
虎牙直播
huya.com
芒果直播

01. Lego简介 — CDN服务器

LEGO

02. 传统Web框架

Nginx

异步回调

框架内部提供请求不同阶段的Hook，通过不同的Hook来实现功能。

需要对框架十分了解，调试比较复杂

Libco

Coroutine

基于协程来编程，应用程序通过协程库函数来驱动服务的运行，函数本身也要针对不同的事件调用对应的处理函数。

存在协程切换开销，上下文存储内存开销

ATS

Continuation

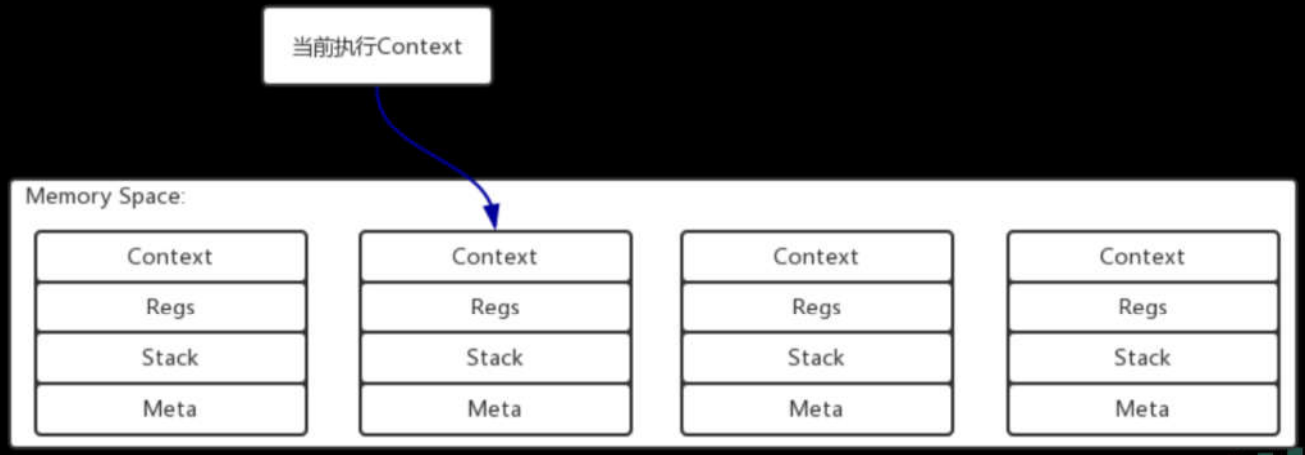
基于Continuation概念进行编程，具体事件被触发时调用Continuation，CPS编程模式的前身。

Continue本身含有锁，并且Continuation与框架紧耦合。

02. 传统Web框架

- **异步回调可维护性和可扩展性不佳：**
  - Nginx的将一个请求区分为11个阶段，哪个阶段实现逻辑最为合适?
  - Nginx框架通过设置不同的event handler将事件串联，代码逻辑分散在各个event handler中，如何管理代码?

- **Coroutinue栈空间分配管理复杂：**
  - 协程栈空间大小设置为多少合适?
  - 协程栈空间的拷贝带来的性能开销有多少?
  - 协程还是基于对不同事件的处理来实现业务逻辑，与异步回源的区别?



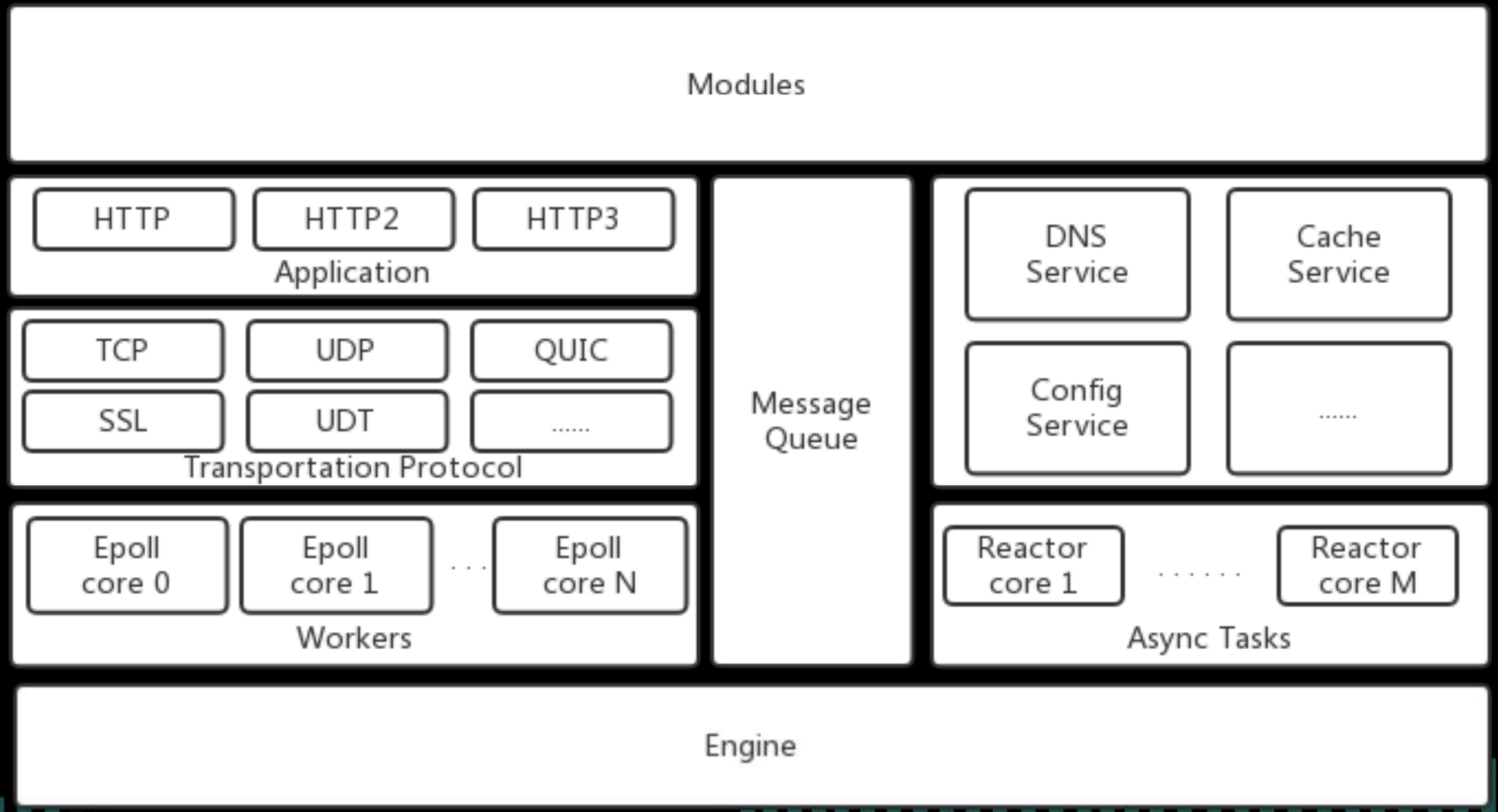
Nginx阶段枚举
NGX_HTTP_POST_READ_PHASE
NGX_HTTP_SERVER_REWRITE_PHASE
NGX_HTTP_POST_REWRITE_PHASE
NGX_HTTP_PREACCESS_PHASE
NGX_HTTP_ACCESS_PHASE
NGX_HTTP_POST_ACCESS_PHASE
NGX_HTTP_PRECONTENT_PHASE
NGX_HTTP_CONTENT_PHASE
NGX_HTTP_LOG_PHASE

02. 传统Web框架

- Continuation-Passing Style整个开发流程基本串行执行
- Continuation基本没有性能开销
- ATS中实现的Continuation并非完美的Continuation
- C++11实现了Continuation关键技术Lambda，并引入了future的概念
- C++新的future extension丰富了future的语义

```
TS_INLINE int
handleEvent(int event = CONTINUATION_EVENT_NONE, void *data = nullptr)
{
  // If there is a lock, we must be holding it on entry
  ink_release_assert(!mutex || mutex->thread_holding == this_ethread());
  return (this->*handler)(event, data);
}
```

03. Lego架构 — 架构图



03. Lego架构 — 异步回调之痛


```
void HandleRequest(Request req)
{
    //do something...
    req.SetReadHandler(ReadRequestHandler);
    req.SetWriteHandler(ErrorWriteHandler);
}

void HandleUpstreamResponse(Request req)
{
    //do something...
    req.SetReadHandler(ReadUpstreamResponseHandler);
    req.SetWriteHandler(ErrorWriteHandler);
}

int ErrorWriteHandler(Request req)
{
    // Something went wrong. How did I get here?
}
```

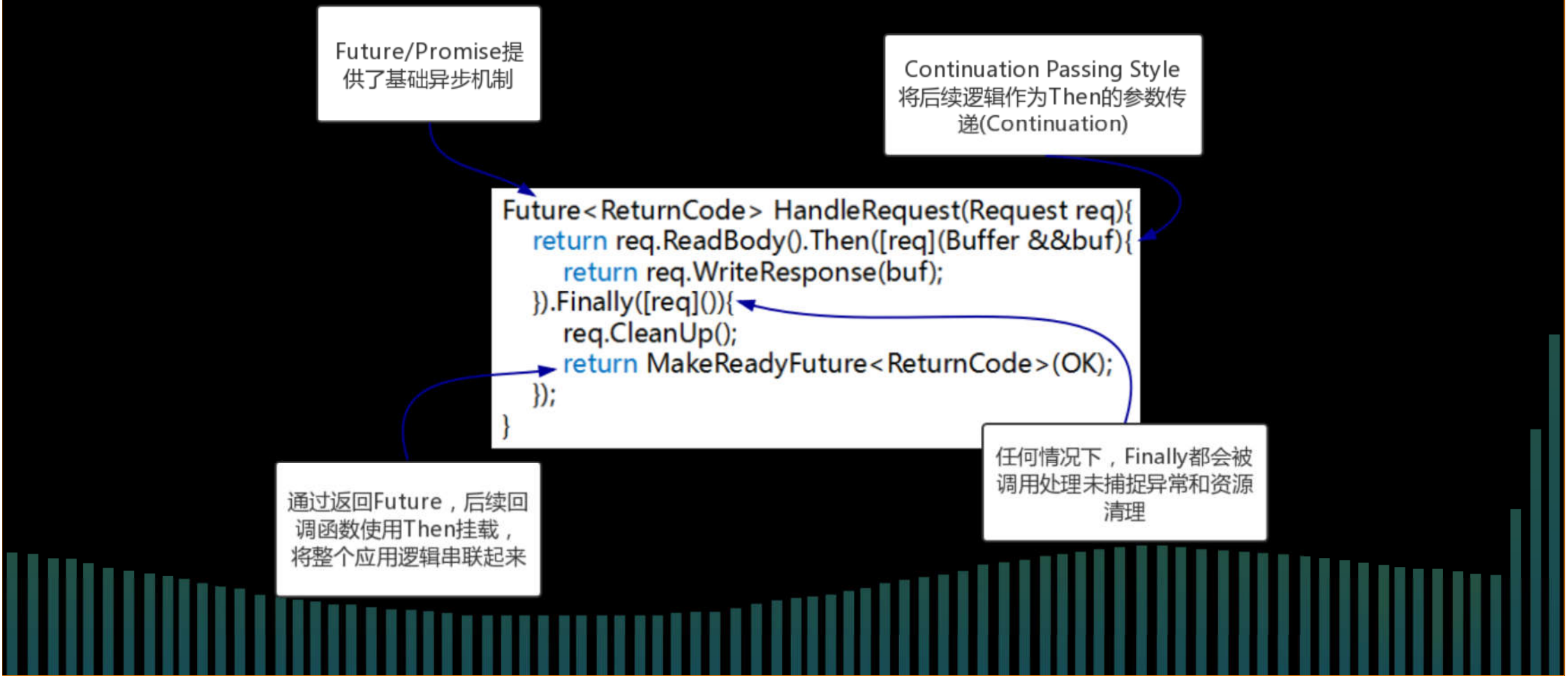
当发生异常时，导致异常的元凶已经逃离现场。Debug过程变得十分困难，主要依赖于程序员的额外记录的信息与经验。

之前执行的是`HandleRequest`？还是`HandleUpstreamResponse`？

Handler的设置导致代码分散，维护性差。

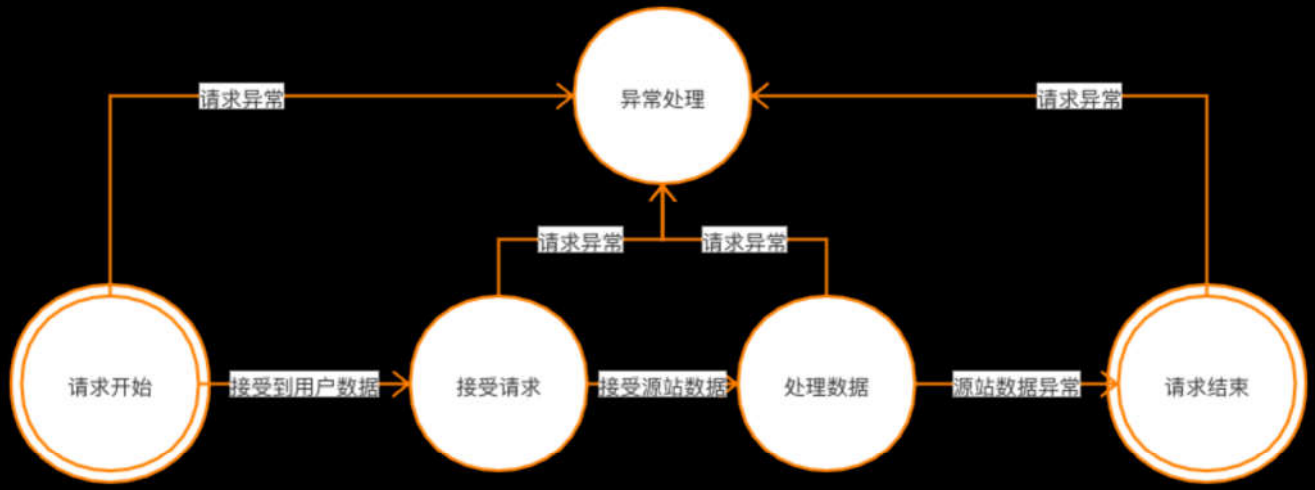
`ReadRequestHandler`和`ErrorWriteHandler`在`CodeBase`中如何组织？

03. Lego架构 — Future/Promise



03. Lego架构 — Future/Promise

状态机



CPS

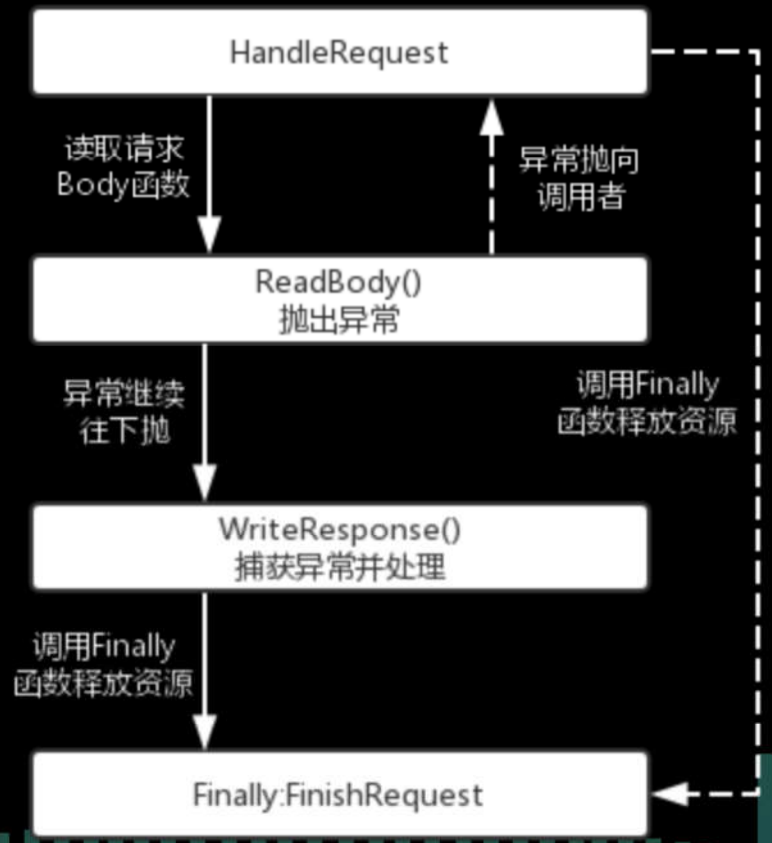
03. Lego架构 —异常处理

自定义Exception结构：降低Exception处理开销。

1. C++目前的Exception处理涉及部分锁，以及复杂的Unwind，查表等过程。性能开销比较大；
2. 自定义轻量化Exception架构，仅包含处理异常必要信息。

异常处理流程：NWS的exception支持finally语义和轻量级catch语义。

1. 顺着Then链路向下抛，而非向上抛；
2. 大部分业务流程并不关心后续发生的异常，反而后续流程更关心之前发生异常；
3. Finally负责处理未捕捉异常，清理资源。



03. Lego架构 — 蜕变

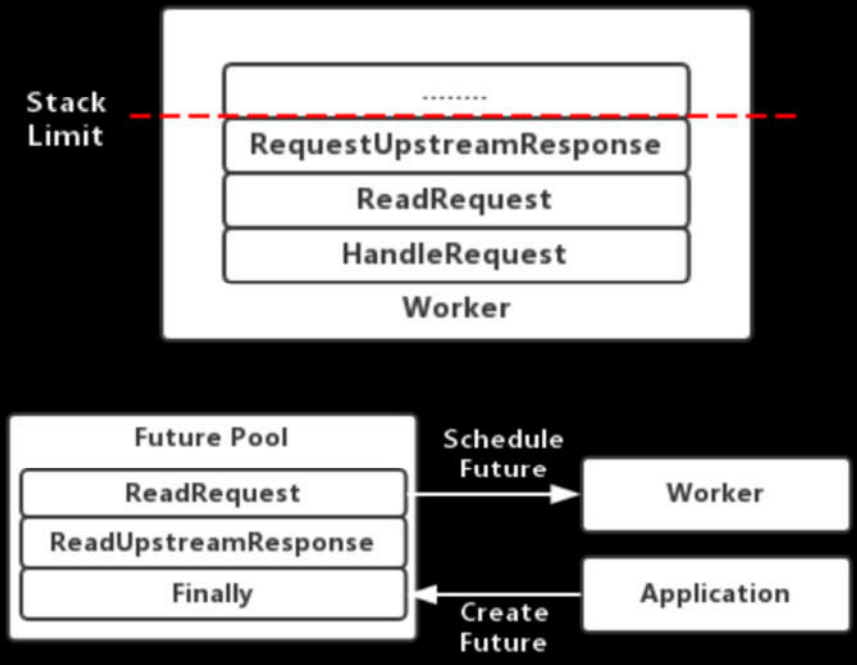

```
void HandleRequest(Request req)
{
    //do something...

    req.SetReadHandler(ReadRequestHandler);
    req.SetWriteHandler(ErrorWriteHandler);
}

void HandleUpstreamResponse(Request req)
{
    //do something...
    req.SetReadHandler(ReadUpstreamResponseHandler);
    req.SetWriteHandler(ErrorWriteHandler);
}

int ErrorWriteHandler(Request req)
{
    // Something went wrong. How did I get here?
}
```

```
void HandleRequest(Request req)
{
    //do something...
    return req.ReadRequest.Then([req](){
        // do something...
        return req.ReadUpstreamResponse();
    }).Finally([]{
        // clean up...
    }).GetValue();
}
```

- 接近Single Thread的编程模式，代码有更强的可读性和可维护性。
- 不需要维护额外的栈信息，没有任何额外的性能开销。
- 相对于异步调用，模块的扩展性更加灵活

03. Lego架构 — Revisited Continuation Again

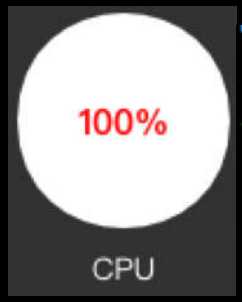

```
void HandleRequest(Request req)
{
    //do something...
    return req.ReadRequest().Then([req](){
        // do something...
        return req.ReadUpstreamResponse();
    }).Finally([]{
        // clean up...
    }).GetValue();
}
```



- Scheduler负责对Future进行优先级调度
- Future Folding，否则由于Future的串联导致栈空间不足

03. Lego架构 — 内存管理

- 大量的Future/Promise对象创建导致内存开销增加




```
-   4.37%  libjemalloc.so       [.] free
   + free
+   2.90%  lego_server          [.] _aesni_ctr32_ghash_6x
-   2.78%  libc-2.17.so         [.] __memcpy_ssse3
   + __memcpy_ssse3
-   2.19%  libjemalloc.so       [.] malloc
   + malloc
```

程序执行过程中会创建大量的Future/Promise结构，导致内存占用持续上涨
解决方案：

1. 替换malloc库，如jemalloc，tmalloc等
2. 每个进程独立TLS内存池，自行管理内存分配

- 局部变量holder问题，编程易出错

03. Lego架构 — 内存管理

- 大量的Future/Promise对象创建导致内存开销增加
- 局部变量holder问题，编程易出错

```
void DoWithoutHolder(Request req){
    User test = ParseUser(req);

    req.ReadRequest().Then([req, test](){
        return req.ReadUpstreamResponse(test);   // test可能已经被释放了
    });
}

void DoWithHolder(){
    std::shared_ptr<User> test = std::make_shared(std::move(ParseUser(req)));

    req.ReadRequest().Then([test, req](){
        return req.ReadUpstreamResponse(test);
    });
}
```

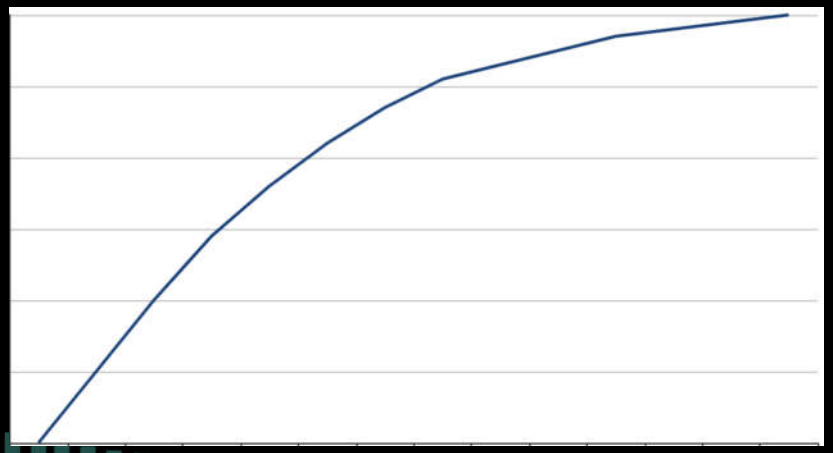
03. Lego架构 — 内存管理

- 大量的Future/Promise对象创建导致内存开销增加
- 局部变量holder问题，编程易出错

```
void DoWithoutHolder(Request req){
    User test = ParseUser(req);

    req.ReadRequest().Then([req, test](){
        return req.ReadUpstreamResponse(test);   // test可能已经被释放了
    });
}

void DoWithHolder(){
    std::shared_ptr<User> test = std::make_shared(std::move(ParseUser(req)));

    req.ReadRequest().Then([test, req](){
        return req.ReadUpstreamResponse(test);
    });
}
```

03. Lego架构 — Shared-Nothing

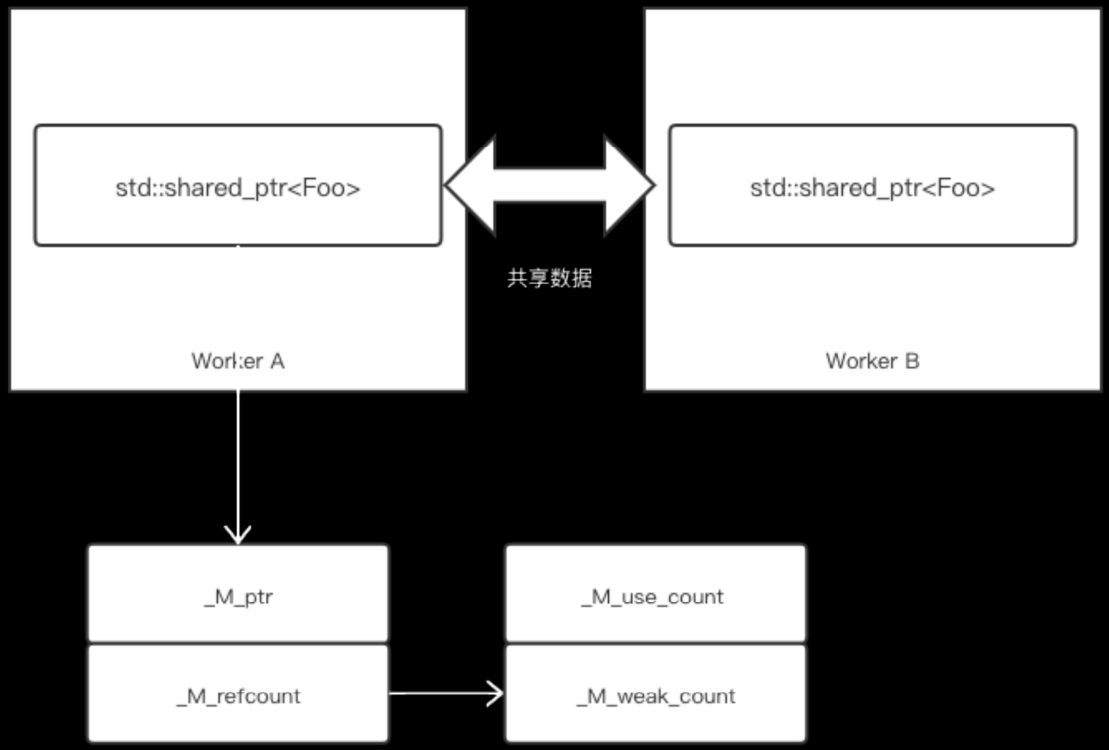
- CPU核数变为原有4倍，服务器性能却只增加2倍
- 单机服务器性能无法线性扩展，cache和锁等问题导致核数增加并不会带来期望的性能提升
- 每个核心独立运行完整程序才能达到性能平行扩展



Numbers Everyone Should Know (Jeff Dean, Google)

L1 cache reference:	0.5 ns
Branch mis-predict:	5 ns
L2 cache reference:	7 ns
Mutex lock/unlock:	25 ns
Main memory reference	100 ns
Compress 1K Bytes with Zippy	3000 ns
Send 2K Bytes over 1 GBPS network	20000 ns
Read 1 MB sequentially from memory	250000 ns
Round trip within data center	500000 ns
Disk seek	1000000 ns
Read 1MB sequentially from disk	2000000 ns
Send one packet from CA to Europe	15000000 ns

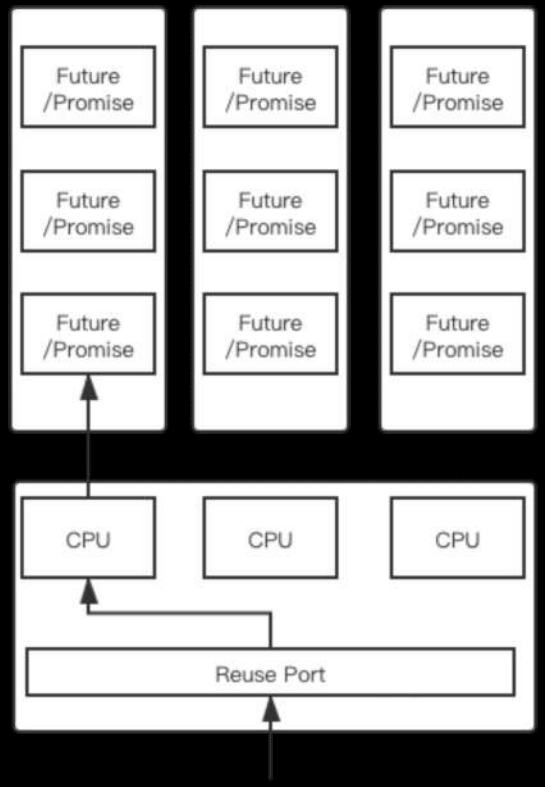
03. Lego架构 — Share Pointer



- std::share_ptr并不智能
- Atomic reference counter带来的性能开销不可控，部分场景下导致CPU高负载
- 跨线程/进程共享数据时，仅有引用计数是thread-safe，指针操作并不安全

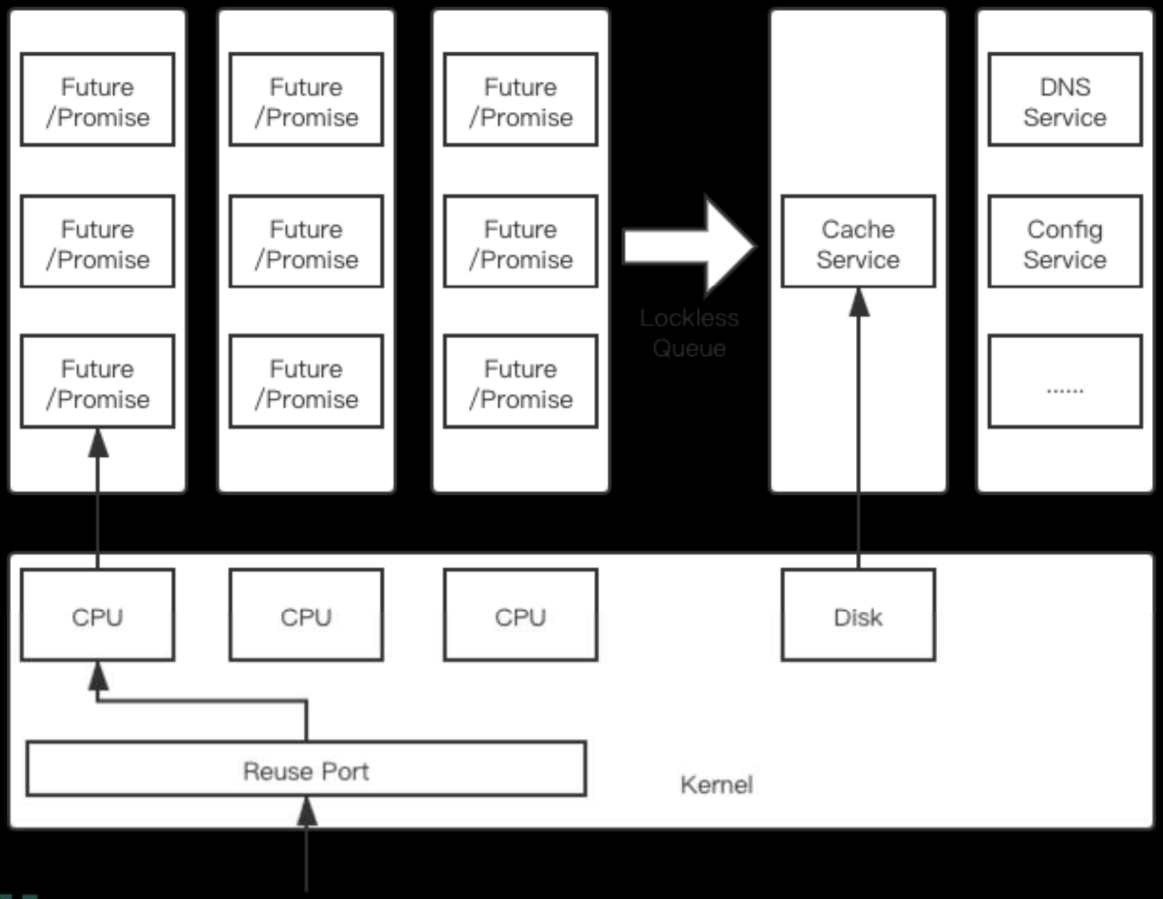
- 不同场景下需要不一样的智能指针
- 单进程下引用计数使用替代atomic变量
- 类似atomic_load解决方案，内部使用mutex数组来解决并发问题

03. Lego架构 — Shared-Nothing



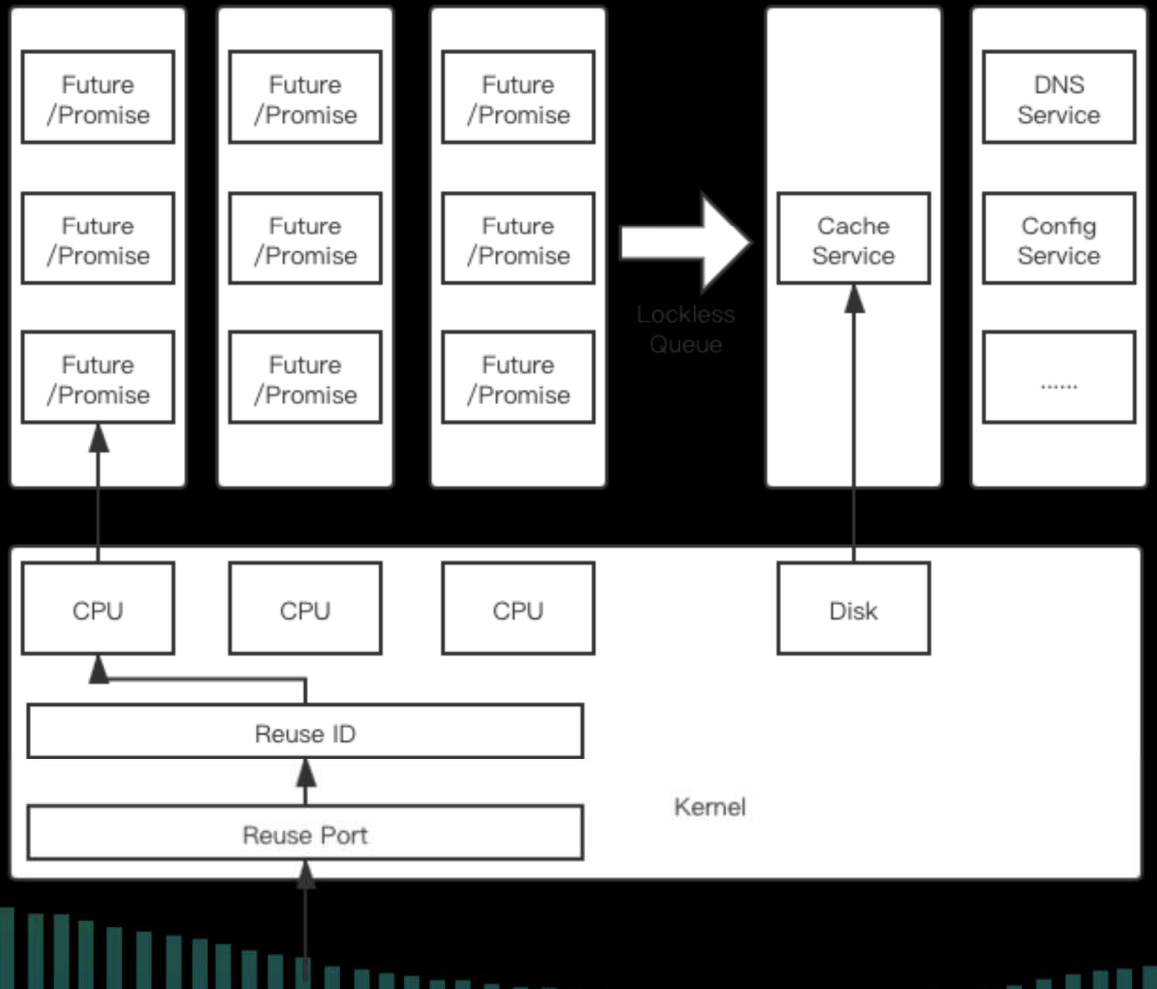
- 每个进程/线程互相之间尽可能避免共享全局数据规避锁和在不同CPU间读写数据时的Cacheline影响
- 通过Linux Kernel Reuse Port特性，自动将链接分发至不同的进程，之后对应链接的请求完全由对应进程/线程处理
- 每个进程绑定固定的CPU，规避进程在不同CPU间迁移
- 服务器性能可以达到近乎平行扩展的能力
- 总是有需要全局共享的数据/处理复杂度较高的任务，如何处理?

03. Lego架构 — Shared-Nothing



- 每个进程/线程互相之间尽可能避免共享全局数据规避锁和在不同CPU间读写数据时的Cacheline影响
- 通过Linux Kernel Reuse Port特性，自动将链接分发至不同的进程，之后对应链接的请求完全由对应进程/线程处理
- 每个进程绑定固定的CPU，规避进程在不同CPU间迁移
- 服务器性能可以达到近乎平行扩展的能力
- 总是有需要全局共享的数据/处理复杂度较高的任务，如何处理？

03. Lego架构 — Shared-Nothing



- Quic类型的请求无法通过Reuse Port进行正确的路由
- 客户端IP和Port变更后，仍然能够转发到对应的线程，解决用户使用场景变更问题，比如4G切换到Wifi
- Quic中使用了一个Connection ID作为标识
- 根据对应ID进行请求路由

03. Lego架构 — 总结

不断攀登

高性能

- Shared-Nothing
- Shared_Ptr
- Exception Handle
- 内存管理

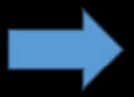
选好工具

可维护

- Future/Promise
- Continuation Passing Style
- 代码规范
- 模块隔离

04. 未来展望 — 最完美代码


```
void HandleRequest(Request req)
{
    //do something...
    return req.ReadRequest.Then([req](){
        // do something...
        return req.SelectUpstream().Then([req](){
            // do something...
            return req.CreateUpstream().Then([req](){
                // do something...
                return req.ReadUpstreamResponse();
            });
        });
    }).Finally([](){
        // clean up...
    }).GetValue();
}
```



```
auto HandleRequest(Request req)
{
    // do something...
    req.ReadRequest();
    // do something...
    req.SelectUpstream();
    // do something...
    req.CreateUpstream();
    // do something...
    auto result = req.ReadUpstreamResponse();
    // clean up...
    return result;
}
```

- 完全Single Thread的写法，避免了Callback Hell。
- 编译器有更多的优化空间，进一步的提升程序性能。

04. 未来展望 — C++20与服务器

- C++20 Coroutine

```
task<> tcp_echo_server() {
  char data[1024];
  for (;;) {
    size_t n = co_await socket.async_read_some(buffer(data));
    co_await async_write(socket, buffer(data, n));
  }
}
```

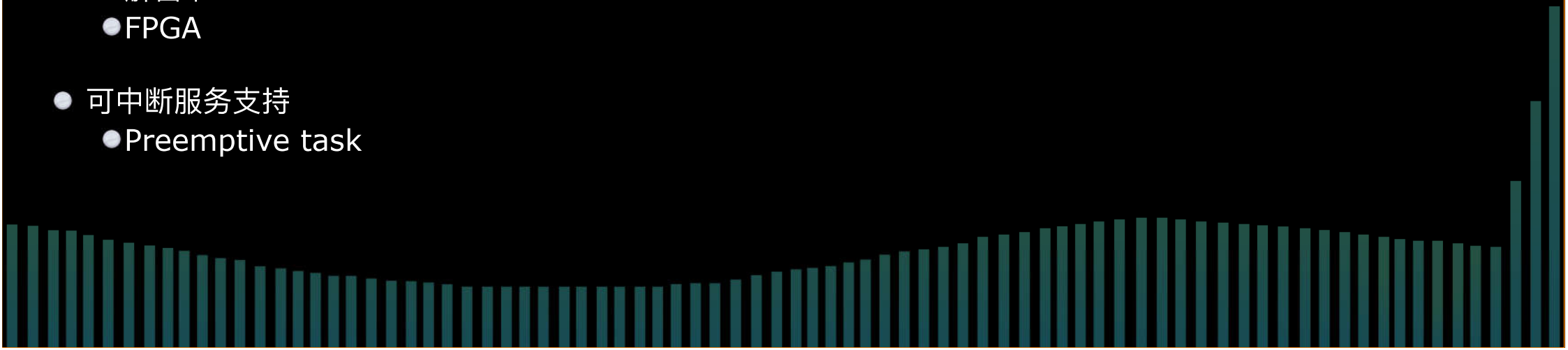
`operator co_await(static_cast<Awaitable&&>(awaitable))` for the non-member overload)

```
auto switch_to_new_thread(std::jthread& out) {
  struct awaitable {
    std::jthread* p_out;
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
      std::jthread& out = *p_out;
      if (out.joinable())
        throw std::runtime_error("Output jthread parameter not empty");
      out = std::jthread([h] { h.resume(); });
      // Potential undefined behavior: accessing potentially destroyed *this
      // std::cout << "New thread ID: " << p_out->get_id() << '\n';
      std::cout << "New thread ID: " << out.get_id() << '\n'; // this is OK
    }
    void await_resume() {}
  };
  return awaitable{&out};
}
```

- 可以最大化利用C++编译器优化能力
- 自动管理Coroutine的上下文
- 目前要求整链路都是Awaitable
  - func1->func2->func3
- 实际性能和内存开销待验证

04. 未来展望 — 待优化点

- 云原生
  - 服务器开包即用
- Kernel Bypassing
  - DPDK/XDP
  - SPDK/io_uring
- 软硬结合
  - 解密卡
  - FPGA
- 可中断服务支持
  - Preemptive task



The End