
技术创造未来
科技TEG
C++服务器开发实践
从异步噩梦走向未来(Future)
Tencent

Who am I?

吴锐 Roy

- 腾讯CDN服务器开发技术负责人
- 专注于CDN相关服务器，内核和网络等技术研发

01 NWS简介

02 传统Web框架

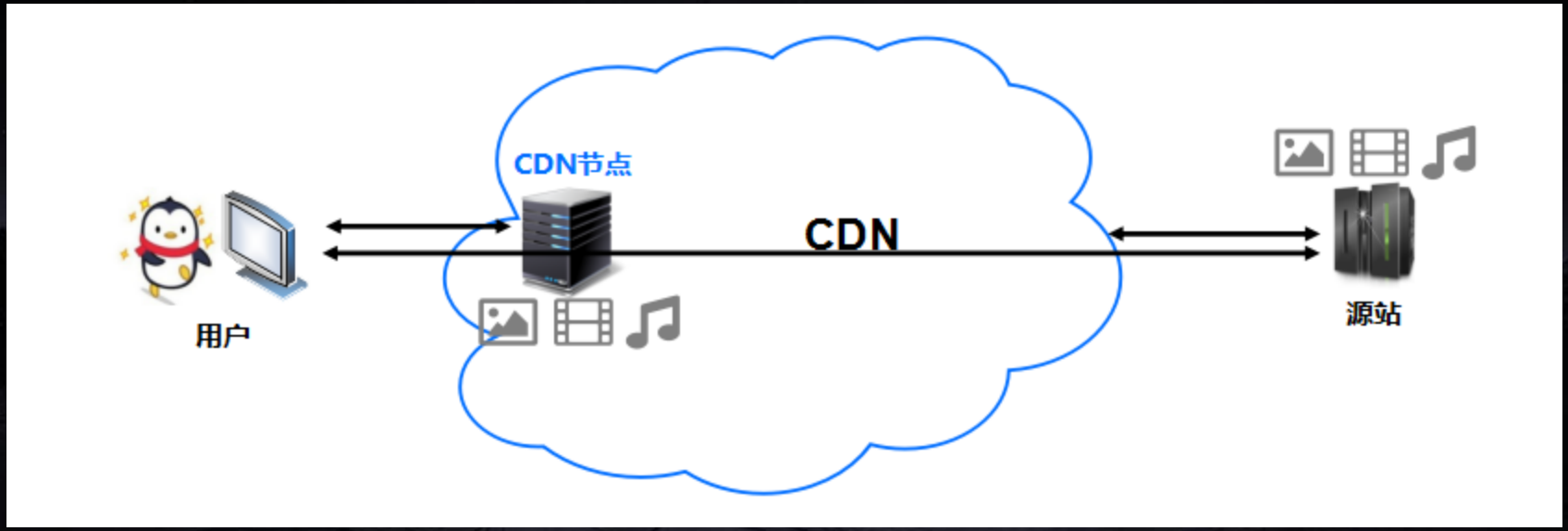
03 NWS架构实现

04 未来展望

01. NWS简介 -- CDN

CDN，内容分发网络(Content Delivery Network)

- 网络世界快递公司，有全国，全球快递网络，网点
- 用最快速度从“卖家”（源站）发送到“买家”（客户端）手上
- 会重复发送全国不同地方，可从“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. NWS简介 -- 腾讯CDN支持业务

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.COM 京东 | 58同城 | 腾讯网 QQ.com | 大众点评 dianping.com | 艺龙 eLong | e饿了么 | 拼多多 | 唯品会 vip.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 | 人人影视网址大全 ALLYINGSHI.COM

流媒体直播加速
直播、互动直播等场景中，下行流分发加速

龙珠直播 Longzhu.com | 快手 | DOUYU | 虎牙直播 huya.com | 芒果直播

01. NWS简介 -- 腾讯CDN支持业务

高流量

高访问量

海量存储

01. NWS简介 -- 服务器特点

高性能

- QPS：CDN平台的QPS达到1000万+每秒
- 流量：承载着CDN大约50+Tbps的流量
- 存储：每台机器存储这上亿文件

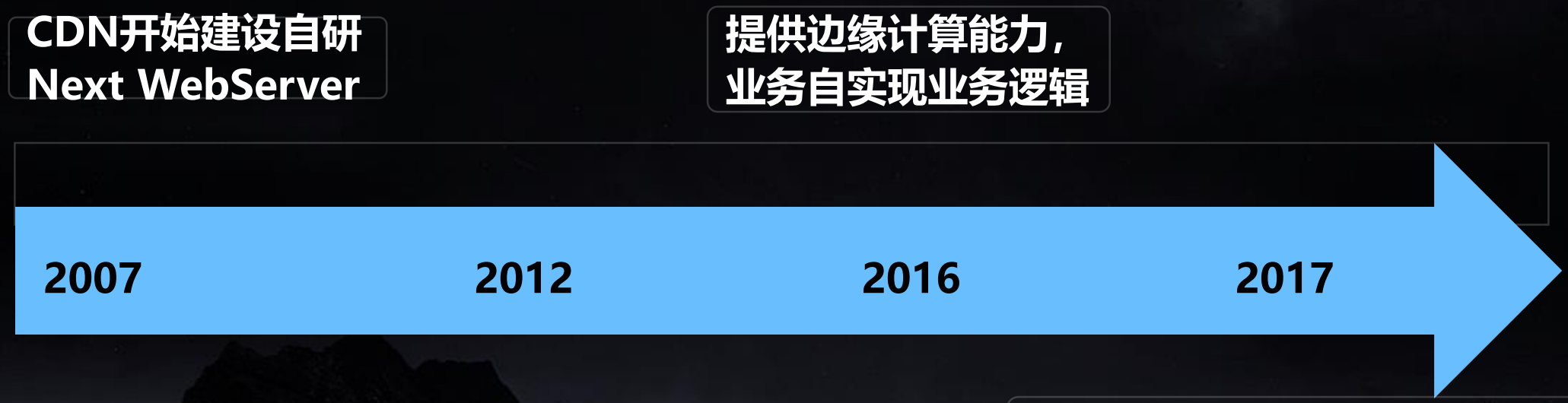
可扩展

- 业务需求：腾讯内部业务和腾讯云海量业务需求各不相同
- 可维护性：对不同功能的支持过程中，模块可以灵活插拔，模块间隔性强
- 学习门槛：新人学习成本低，提高开发效率

CDN特性

- 新协议：新的协议不断的涌现：HTTP2，HTTP3，RTMP
- 数据的搬运工：不仅与客户端进行交互，也需要与源站进行交互
- 缓存：存储热点文件，淘汰冷文件，提升用户体验

01. NWS简介 -- 架构演进



02. 现有Web框架 -- 总览

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 -- 不足

- **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？

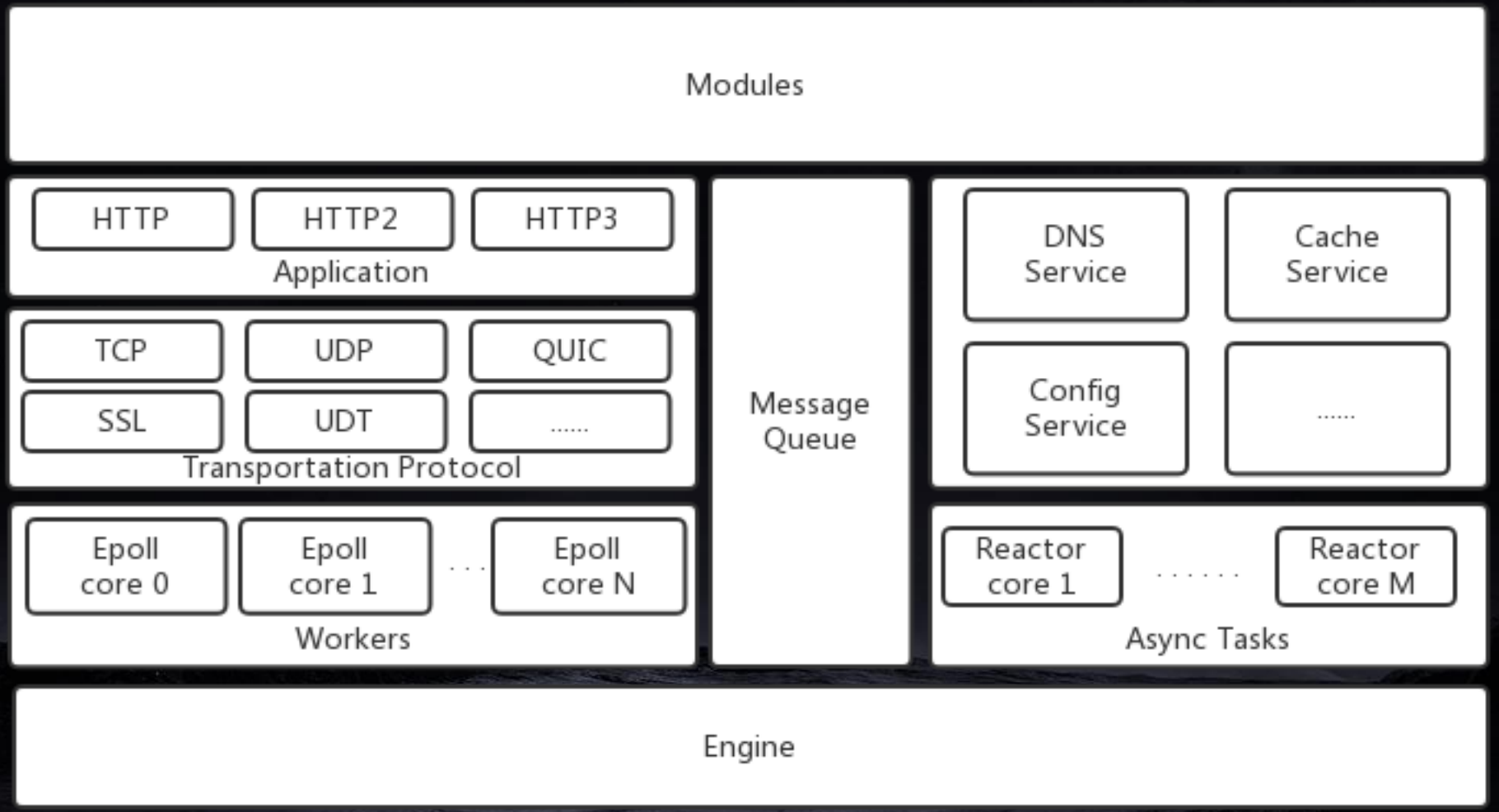
Nginx阶段枚举
NGX_HTTP_POST_READ_PHASE
NGX_HTTP_SERVER_REWRITE_PHASE
NGX_HTTP_POST_REWRITE_PHASE
NGX_HTTP_PREACCESS_PHASE
NGX_HTTP_ACCESS_PHASE
NGX_HTTP_POST_ACCESS_PHASE
NGX_HTTP_PRECONTENT_PHASE
NGX_HTTP_CONTENT_PHASE
NGX_HTTP_LOG_PHASE

02. 传统Web框架 -- Continuation

- 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. NWS架构 -- 架构简介



03. NWS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. NWS架构 -- Revisit Cotinuation

Future/Promise提供了基础异步机制

Continuation Passing Style 将后续逻辑作为Then的参数传递(Continuation)

```
Future<ReturnCode> HandleRequest(Request req){
    return req.ReadBody().Then([req](Buffer &&buf){
        return req.WriteResponse(buf);
    }).Finally([req](){
        req.CleanUp();
        return MakeReadyFuture<ReturnCode>(OK);
    });
}
```

通过返回Future，后续回调函数使用Then挂载，将整个应用逻辑串联起来

任何情况下，Finally都会被调用处理未捕捉异常和资源清理

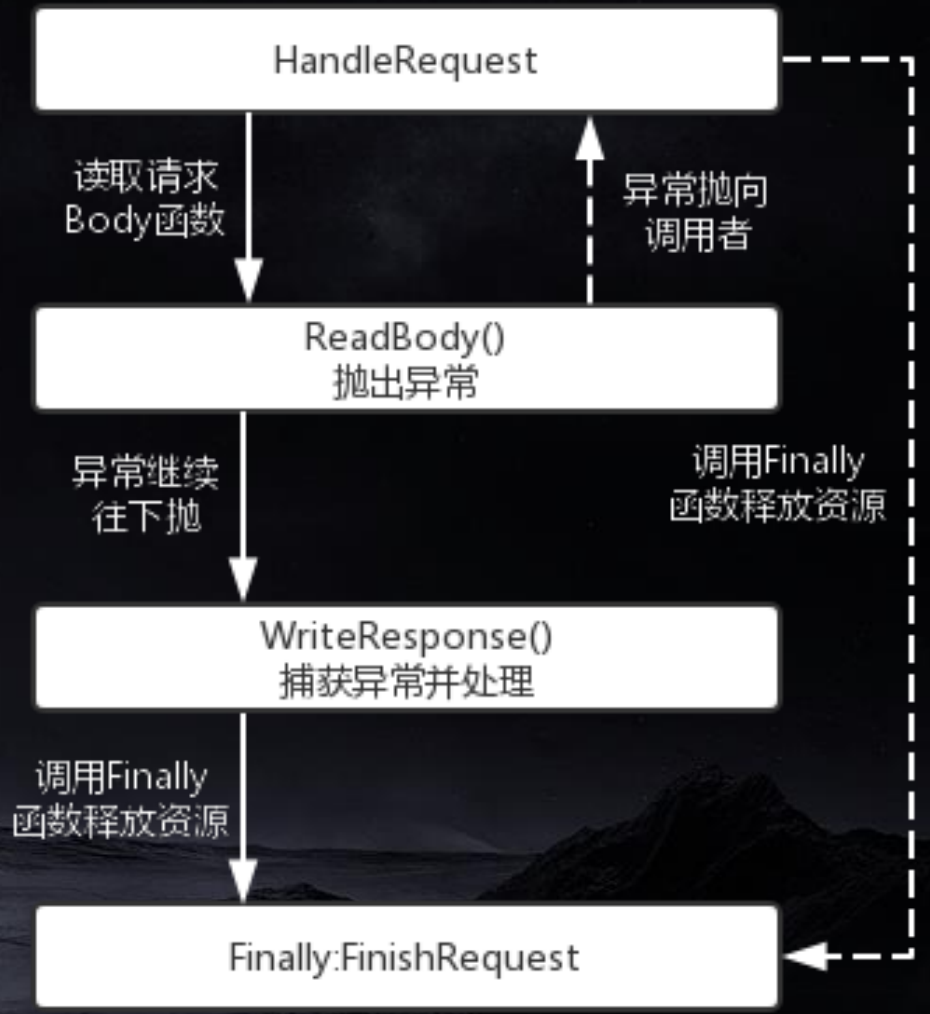
03. NWS架构 -- Exception Handle

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. NWS架构 -- 蜕变

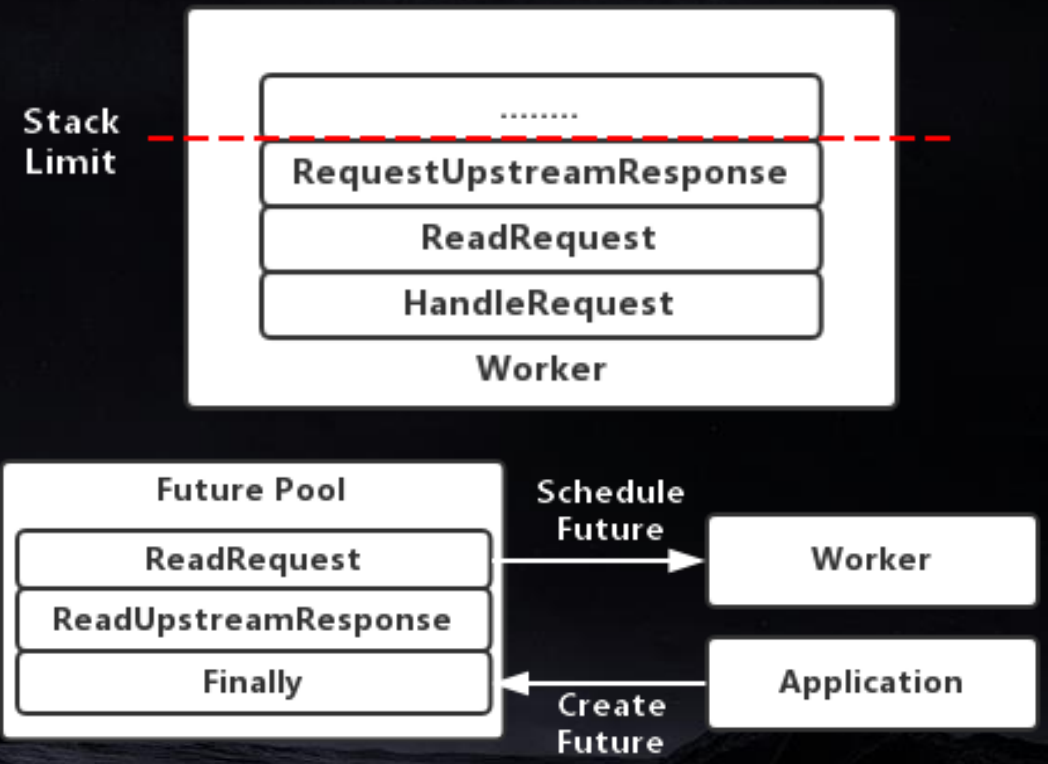
```
void HandleRequest(Request req)
{
    //do something...

    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. NWS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

04. 未来展望 -- C++新特性

atomic smart pointer: 提供了实现RCU和无锁算法的工具。

```
atomic<shared_ptr<T>>
atomic<weak_ptr<T>>
atomic<unique_ptr<T>>
```

Lambda Capture: C++11的Lambda Capture并不完美，C++14/17支持了[&]identifier initializer 等更完备的功能。

```
std::unique_ptr<char> old_value(new char(0xFF));
auto new_lambda = [move_test = std::move(old_value)] { return *old_value + 1; };
```

Resumable Functions: 通过Then进行串联，可能会导致Future Callback Hell。

```
future<int> f(stream str) async
{
    shared_ptr<vector<char>> buf = ...;
    int count = await str.read(512, buf);
    return count + 11;
}
```

04. 未来展望 -- 最完美的代码

```
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。

认识我们

技术创造未来 科技TEG

腾讯架构师

吴锐