第一章 Node.js组成和原理

1.1 Node.js简介

Node.js是基于事件驱动的单进程单线程应用，单线程具体体现在Node.js在单个线程中维护了一系列任务，然后在事件循环中不断消费任务队列中的节点，又不断产生新的任务，在任务的产生和消费中不断驱动着Node.js的执行。从另外一个角度来说，Node.js又可以说是多线程的，因为Node.js底层也维护了一个线程池，该线程池主要用于处理一些文件IO、DNS、CPU计算等任务。

Node.js主要由V8、Libuv，还有一些其它的第三方模块组成（cares异步DNS解析库、HTTP解析器、HTTP2解析器，压缩库、加解密库等）。Node.js源码分为三层，分别是JS、C++、C，Libuv是使用C语言编写，C++层主要是通过V8为JS层提供和底层交互的能力，C++层也实现了部分功能，JS层是面向用户的，为用户提供调用底层的接口。

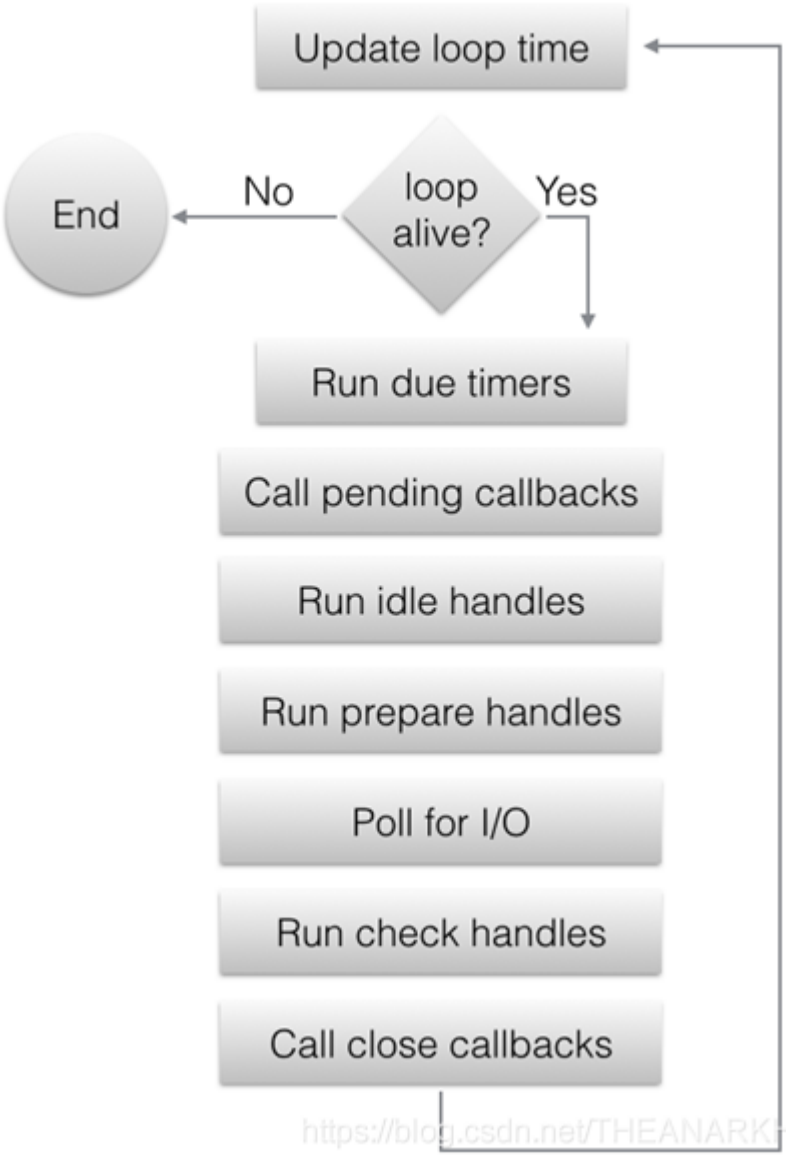
1.1.1 JS引擎V8

Node.js是基于V8的JS运行时，它利用V8提供的能力，极大地拓展了JS的能力。这种拓展不是为JS增加了新的语言特性，而是拓展了功能模块，比如在前端，我们可以使用Date这个函数，但是我们不能使用TCP这个函数，因为JS中并没有内置这个函数。而在Node.js中，我们可以使用TCP，这就是Node.js做的事情，让用户可以使用JS中本来不存在的功能，比如文件、网络。Node.js中最核心的部分是Libuv和V8，V8不仅负责执行JS，还支持自定义的拓展，实现了JS调用C++和C++调用JS的能力。比如我们可以写一个C++模块，然后在JS调用，Node.js正是利用了这个能力，完成了功能的拓展。JS层调用的所有C、C++模块都是通过V8来完成的。

1.1.2 Libuv

Libuv是Node.js底层的异步IO库，但它提供的功能不仅仅是IO，还包括进程、线程、信号、定时器、进程间通信等，而且Libuv抹平了各个操作系统之间的差异。Libuv提供的功能大概如下 • Full-featured event loop backed by epoll, kqueue, IOCP, event ports. • Asynchronous TCP and UDP sockets • Asynchronous DNS resolution • Asynchronous file and file system operations • File system events • ANSI escape code controlled TTY • IPC with socket sharing, using Unix domain sockets or named pipes (Windows) • Child processes • Thread pool • Signal handling • High resolution clock • Threading and synchronization primitives

Libuv的实现是一个经典的生产者-消费者模型。Libuv在整个生命周期中，每一轮循环都会处理每个阶段（phase）维护的任务队列，然后逐个执行任务队列中节点的回调，在回调中，不断生产新的任务，从而不断驱动Libuv。下是Libuv的整体执行流程



从上图中我们大致了解到，Libuv分为几个阶段，然后在一个循环里不断执行每个阶段里的任务。下面我们具体看一下每个阶段

1. 更新当前时间，在每次事件循环开始的时候，Libuv会更新当前时间到变量中，这一轮循环的剩下操作可以使用这个变量获取当前时间，避免过多的系统调用影响性能，额外的影响就是时间不是那么精确。但是在一轮事件循环中，Libuv在必要的时候，会主动更新这个时间，比如在epoll中阻塞了timeout时间后返回时，会再次更新当前时间变量。

2. 如果事件循环是处于alive状态，则开始处理事件循环的每个阶段，否则退出这个事件循环。alive状态是什么意思呢？如果有active和ref状态的handle，active状态的request或者closing状态的handle则认为事件循环是alive（具体的后续会讲到）。
3. timer阶段：判断最小堆中的节点哪个节点超时了，执行它的回调。
4. pending阶段：执行pending回调。一般来说，所有的IO回调（网络，文件，DNS）都会在Poll IO阶段执行，但是有的情况下，Poll IO阶段的回调会延迟到下一次循环执行，那么这种回调就是在pending阶段执行的，比如IO回调里出现了错误或写数据成功等等都会在下一个事件循环的pending阶段执行回调。
5. idle阶段：每次事件循环都会被执行（idle不是说事件循环空闲的时候才执行）。
6. prepare阶段：和idle阶段类似。
7. Poll IO阶段：调用各平台提供的IO多路复用接口（比如Linux下就是epoll模式），最多等待timeout时间，返回的时候，执行对应的回调。timeout的计算规则：
   1 如果时间循环是以UV_RUN_NOWAIT模式运行的，则timeout是0。
   2 如果时间循环即将退出（调用了uv_stop），则timeout是0。
   3 如果没有active状态的handle或者request，timeout是0。
   4 如果有idle阶段的队列里有节点，则timeout是0。
   5 如果有handle等待被关闭的（即调了uv_close），timeout是0。
   6 如果上面的都不满足，则取timer阶段中最快超时的节点作为timeout。 7 如果上面的都不满足则timeout等于-1，即一直阻塞，直到满足条件。
8. check阶段：和idle、prepare一样。
9. closing阶段：执行调用uv_close函数时传入的回调。
10. 如果Libuv是以UV_RUN_ONCE模式运行的，那事件循环即将退出。但是有一种情况是，Poll IO阶段的timeout的值是timer阶段的节点的值，并且Poll IO阶段是因为超时返回的，即没有任何事件发生，也没有执行任何IO回调，这时候需要在执行一次timer阶段。因为有节点超时了。

11. 一轮事件循环结束，如果Libuv以UV_RUN_NOWAIT 或 UV_RUN_ONCE模式运行的，则退出事件循环，如果是以UV_RUN_DEFAULT模式运行的并且状态是alive，则开始下一轮循环。否则退出事件循环。

下面我能通过一个例子来了解libuv的基本原理。

```c
#include <stdio.h>
#include <uv.h>

int64_t counter = 0;

void wait_for_a_while(uv_idle_t* handle) {
    counter++;
    if (counter >= 10e6)
        uv_idle_stop(handle);
}

int main() {
    uv_idle_t idler;
     // 获取事件循环的核心结构体。并初始化一个idle
    uv_idle_init(uv_default_loop(), &idler);
    // 往事件循环的idle阶段插入一个任务
    uv_idle_start(&idler, wait_for_a_while);
    // 启动事件循环
    uv_run(uv_default_loop(), UV_RUN_DEFAULT);
    // 销毁libuv的相关数据
    uv_loop_close(uv_default_loop());
    return 0;
}
```

使用Libuv，我们首先需要获取Libuv的核心结构体uv_loop_t，uv_loop_t是一个非常大的结构体，里面记录了Libuv整个生命周期的数据。uv_default_loop为我们提供了一个默认已经初始化了的uv_loop_t结构体，当然我们也可以自己去分配一个，自己初始化。

```c
uv_loop_t* uv_default_loop(void) {
  // 缓存
  if (default_loop_ptr != NULL)
```

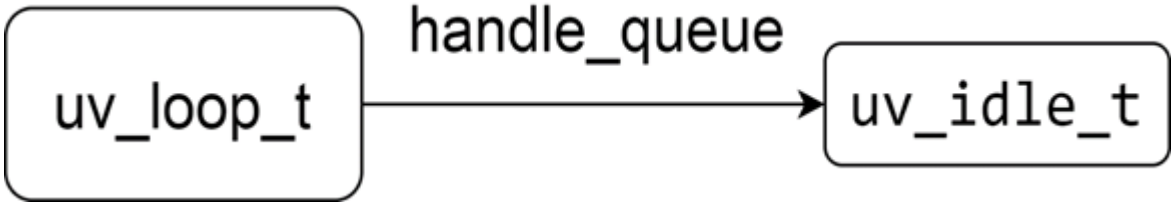
```
4           return default_loop_ptr;
5
6         if (uv_loop_init(&default_loop_struct))
7           return NULL;
8
9         default_loop_ptr = &default_loop_struct;
10        return default_loop_ptr;
11      }
```

Libuv维护了一个全局的uv_loop_t结构体，使用uv_loop_init进行初始化，不打算展开讲解uv_loop_init函数，w因为它大概就是对uv_loop_t结构体各个字段进行初始化。接着我们看一下uv_idle_*系列的函数。

1 uv_idle_init

```
1       int uv_idle_init(uv_loop_t* loop, uv_idle_t* handle)
2   {
3           /*
4             初始化handle的类型，所属loop，打上UV_HANDLE_REF，
5             并且把handle插入loop->handle_queue队列的队尾
6           */
7           uv__handle_init(loop, (uv_handle_t*)handle,
8   UV_IDLE);
9           handle->idle_cb = NULL;
            return 0;
        }
```

执行uv_idle_init函数后，Libuv的内存视图如下图所示

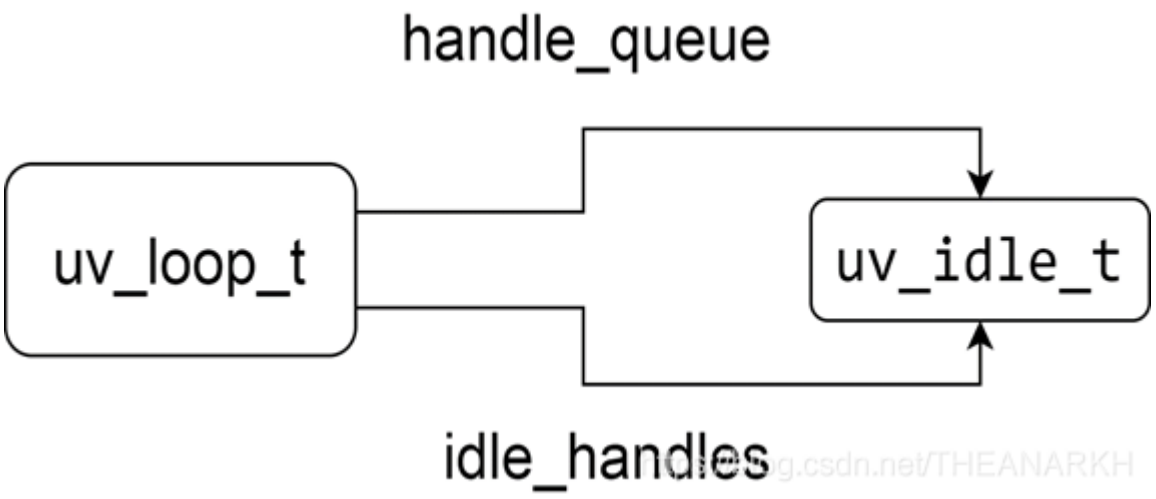


2 uv_idle_start

```
1
```

```
    int uv_idle_start(uv_idle_t* handle, uv_idle_cb cb)
{
      // 如果已经执行过start函数则直接返回
       if (uv__is_active(handle)) return 0;
       // 把handle插入loop中idle的队列
       QUEUE_INSERT_HEAD(&handle->loop->idle_handles,
&handle->queue);
       // 挂载回调，下一轮循环的时候被执行
       handle->idle_cb = cb;
       /*
         设置UV_HANDLE_ACTIVE标记位，并且loop中的handle数加
一，
        init的时候只是把handle挂载到loop，start的时候handle
才
        处于激活态
      */
       uv__handle_start(handle);
       return 0;
     }
```

执行完uv_idle_start的内存视图如下图所示。



然后执行uv_run进入Libuv的事件循环。

```
    int uv_run(uv_loop_t* loop, uv_run_mode mode) {
      int timeout;
```

```
        int r;
        int ran_pending;
        // 在uv_run之前要先提交任务到loop
        r = uv__loop_alive(loop);
        // 没有任务需要处理或者调用了uv_stop
        while (r != 0 && loop->stop_flag == 0) {
          // 处理idle队列
          uv__run_idle(loop);
        }

        // 是因为调用了uv_stop退出的，重置flag
        if (loop->stop_flag != 0)
          loop->stop_flag = 0;
        /*
          返回是否还有活跃的任务（handle或request），
          业务代表可以再次执行uv_run
        */
        return r;
      }
```

我们看到有一个函数是uv__run_idle，这就是处理idle阶段的函数。我们看一下它的实现。

```
      // 在每一轮循环中执行该函数，执行时机见uv_run
        void uv__run_idle(uv_loop_t* loop) {
          uv_idle_t* h;
          QUEUE queue;
          QUEUE* q;
          /*
              把该类型对应的队列中所有节点摘下来挂载到queue变
量，
              变量回调里不断插入新节点，导致死循环
           */
          QUEUE_MOVE(&loop->idle_handles, &queue);
          // 遍历队列，执行每个节点里面的函数
          while (!QUEUE_EMPTY(&queue)) {
            // 取下当前待处理的节点
            q = QUEUE_HEAD(&queue);
            // 取得该节点对应的整个结构体的基地址
            h = QUEUE_DATA(q, uv_idle_t, queue);
```

```
            // 把该节点移出当前队列，否则循环不会结束
            QUEUE_REMOVE(q);
           // 重新插入原来的队列
            QUEUE_INSERT_TAIL(&loop->idle_handles, q);
           // 执行回调函数
            h->idle_cb(h);
          }
        }
```

我们看到uv__run_idle的逻辑并不复杂，就是遍历idle_handles队列的节点，然后执行回调，在回调里我们可以插入新的节点（产生新任务），从而不断驱动Libuv的运行。我们看到uv_run退出循环的条件下面的代码为false。

```
    r != 0 && loop->stop_flag == 0
```

stop_flag由用户主动关闭Libuv事件循环。

```
    void uv_stop(uv_loop_t* loop) {
      loop->stop_flag = 1;
    }
```

r是代表事件循环是否还存活，这个判断的标准是由uv__loop_alive提供

```
    static int uv__loop_alive(const uv_loop_t* loop) {
      return loop->active_handles > 0 ||
             loop->active_reqs.count > 0 ||
             loop->closing_handles != NULL;
    }
```

这时候我们有一个actived handles，所以Libuv不会退出。当我们调用uv_idle_stop函数把idle节点移出handle队列的时候，Libuv就会退出。后面我们会具体分析Libuv事件循环的原理。

1.1.3 其它第三方库

Node.js中第三方库包括异步DNS解析（cares）、HTTP解析器（旧版使用http_parser，新版使用llhttp）、HTTP2解析器（nghttp2）、解压压缩库(zlib)、加密解密库(openssl)等等，不一一介绍。

1.2 Node.js工作原理

1.2.1 Node.js是如何拓展JS功能的?

V8提供了一套机制，使得我们可以在JS层调用C++、C语言模块提供的功能。Node.js正是通过这套机制，实现了对JS能力的拓展。Node.js在底层做了大量的事情，实现了很多功能，然后在JS层暴露接口给用户使用，降低了用户成本，也提高了开发效率。

1.2.2 如何在V8新增一个自定义的功能?

```
    // C++里定义
    Handle<FunctionTemplate> Test =
FunctionTemplate::New(cb);
    global->Set(String::New("Test"), Test);
    // JS里使用
    const test = new Test();
```

我们先有一个感性的认识，在后面的章节中，会具体讲解如何使用V8拓展JS的功能。

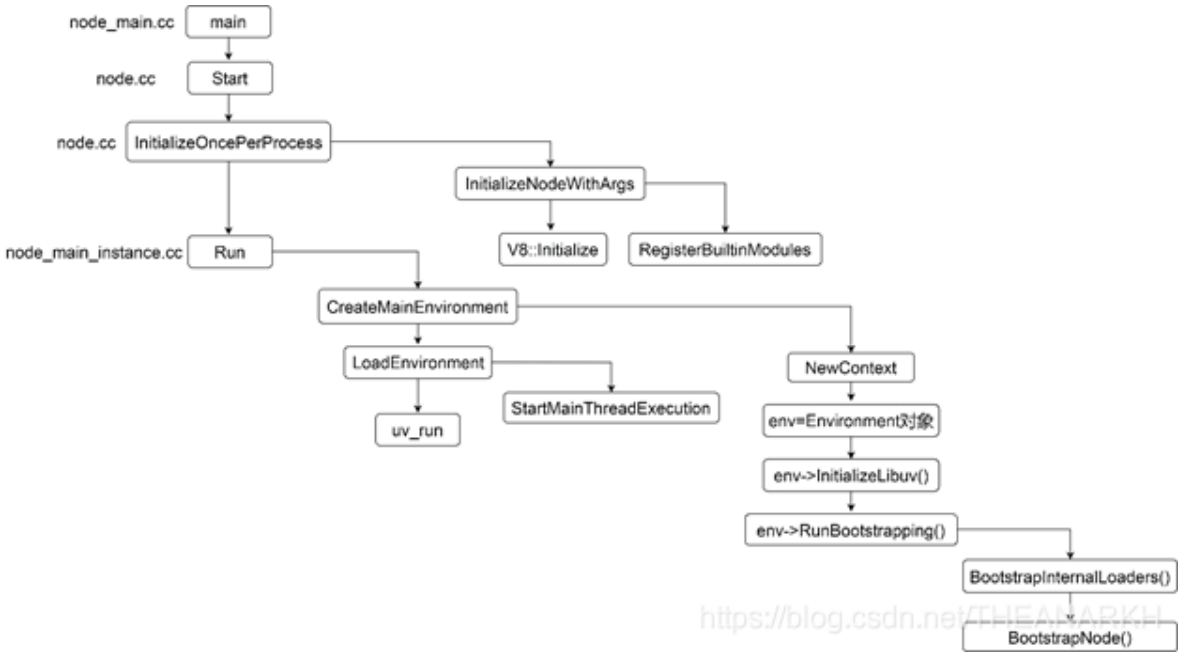
1.2.3 Node.js是如何实现拓展的?

Node.js并不是给每个功能都拓展一个对象，然后挂载到全局变量中，而是拓展一个process对象，再通过process.binding拓展js功能。Node.js定义了一个全局的JS对象process，映射到一个C++对象process，底层维护了一个C++模块的链表，JS通过调用JS层的process.binding，访问到C++的process对象，从

而访问C++模块(类似访问JS的Object、Date等)。不过Node.js 14版本已经改成internalBinding的方式，通过internalBinding就可以访问C++模块，原理类似。

1.3 Node.js启动过程

下面是Node.js启动的主流程图如图1-4所示。



我们从上往下，看一下每个过程都做了些什么事情。

1.3.1 注册C++模块

RegisterBuiltinModules函数（node_binding.cc）的作用是注册C++模块。

```
    void RegisterBuiltinModules() {
    #define V(modname) _register_##modname();
      NODE_BUILTIN_MODULES(V)
    #undef V
    }
```

NODE_BUILTIN_MODULES是一个C语言宏，宏展开后如下（省略类似逻辑）

```
    void RegisterBuiltinModules() {
    #define V(modname) _register_##modname();
      V(tcp_wrap)
      V(timers)
      ...其它模块
    #undef V
    }
```

再一步展开如下

```
    void RegisterBuiltinModules() {
      _register_tcp_wrap();
      _register_timers();
    }
```

执行了一系列_register开头的函数，但是我们在Node.js源码里找不到这些函数，因为这些函数是在每个C++模块定义的文件里（.cc文件的最后一行）通过宏定义的。以tcp_wrap模块为例，看看它是怎么做的。文件tcp_wrap.cc的最后一句代码 NODE_MODULE_CONTEXT_AWARE_INTERNAL(tcp_wrap, node::TCPWrap::Initialize) 宏展开是

```
    #define NODE_MODULE_CONTEXT_AWARE_INTERNAL(modname,
regfunc)  \
    NODE_MODULE_CONTEXT_AWARE_CPP(modname,
                                  regfunc,
                                  nullptr,
                                  NM_F_INTERNAL)
```

继续展开

```
     #define NODE_MODULE_CONTEXT_AWARE_CPP(modname,
 regfunc, priv, flags\
      static node::node_module _module = {
 \
```

```
          NODE_MODULE_VERSION,                          \
          flags,                          \
          nullptr,                          \
          __FILE__,                          \
          nullptr,                          \
          (node::addon_context_register_func)(regfunc),
\
          NODE_STRINGIFY(modname),
\
          priv,                          \
          nullptr};                          \
      void _register_tcp_wrap() {
node_module_register(&_module); }
```

我们看到每个C++模块底层都定义了一个_register开头的函数，在Node.js启动时，就会把这些函数逐个执行一遍。我们继续看一下这些函数都做了什么，在这之前，我们要先了解一下Node.js中表示C++模块的数据结构。

```
    struct node_module {
      int nm_version;
      unsigned int nm_flags;
      void* nm_dso_handle;
      const char* nm_filename;
      node::addon_register_func nm_register_func;
      node::addon_context_register_func
nm_context_register_func;
      const char* nm_modname;
      void* nm_priv;
      struct node_module* nm_link;
    };
```

我们看到_register开头的函数调了node_module_register，并传入一个node_module数据结构，所以我们看一下node_module_register的实现

```
    void node_module_register(void* m) {
      struct node_module* mp = reinterpret_cast<struct
node_module*>(m);
      if (mp->nm_flags & NM_F_INTERNAL) {
```

```
        mp->nm_link = modlist_internal;
        modlist_internal = mp;
      } else if (!node_is_initialized) {
        mp->nm_flags = NM_F_LINKED;
        mp->nm_link = modlist_linked;
        modlist_linked = mp;
      } else {
        thread_local_modpending = mp;
      }
    }
```

C++内置模块的flag是NM_F_INTERNAL，所以会执行第一个if的逻辑，modlist_internal类似一个头指针。if里的逻辑就是头插法建立一个单链表。C++内置模块在Node.js里是非常重要的，很多功能都会调用，后续我们会看到。

1.3.2 创建Environment对象

1 CreateMainEnvironment

Node.js中Environment类（env.h）是一个很重要的类，Node.js中，很多数据由Environment对象进行管理。

```
    context = NewContext(isolate_);
    std::unique_ptr<Environment> env =
std::make_unique<Environment>(
          isolate_data_.get(),
          context,
          args_,
          exec_args_,
          static_cast<Environment::Flags>
(Environment::kIsMainThread |
    Environment::kOwnsProcessState |
Environment::kOwnsInspector));
```

Isolate，Context是V8中的概念，Isolate用于隔离实例间的环境，Context用于提供JS执行时的上下文，kIsMainThread说明当前运行的是主线程，用于区分Node.js中的worker_threads子线程。Environment类非常庞大，我们看一下初始化的代码

```
    Environment::Environment(IsolateData* isolate_data,
                             Local<Context> context,
                             const
std::vector<std::string>& args,
                             const
std::vector<std::string>& exec_args,
                             Flags flags,
                             uint64_t thread_id)
        : isolate_(context->GetIsolate()),
          isolate_data_(isolate_data),
          immediate_info_(context->GetIsolate()),
          tick_info_(context->GetIsolate()),
          timer_base_(uv_now(isolate_data-
>event_loop())),
          exec_argv_(exec_args),
          argv_(args),
          exec_path_(GetExecPath(args)),
          should_abort_on_uncaught_toggle_(isolate_, 1),
          stream_base_state_(isolate_,
StreamBase::kNumStreamBaseStateFields),
          flags_(flags),
          thread_id_(thread_id == kNoThreadId ?
AllocateThreadId() : thread_id),
          fs_stats_field_array_(isolate_,
kFsStatsBufferLength),
          fs_stats_field_bigint_array_(isolate_,
kFsStatsBufferLength),
          context_(context->GetIsolate(), context) {
      // 进入当前的context
      HandleScope handle_scope(isolate());
      Context::Scope context_scope(context);
      // 保存环境变量
      set_env_vars(per_process::system_environment);
      // 关联context和env
```

```
        AssignToContext(context, ContextInfo(""));
        // 创建其它对象
        CreateProperties();
      }
```

我们只看一下AssignToContext和CreateProperties，set_env_vars会把进程章节讲解。

1.1 AssignToContext

```
    inline void
Environment::AssignToContext(v8::Local<v8::Context>
context,
                                                const
ContextInfo& info) {
      // 在context中保存env对象
      context-
>SetAlignedPointerInEmbedderData(ContextEmbedderIndex::kEnvi
 this);
      // Used by Environment::GetCurrent to know that we
are on a node context.
      context-
>SetAlignedPointerInEmbedderData(ContextEmbedderIndex::kCont
 Environment::kNodeContextTagPtr);

    }
```

AssignToContext用于保存context和env的关系。这个逻辑非常重要，因为后续执行代码时，我们会进入V8的领域，这时候，我们只知道Isolate和context。如果不保存context和env的关系，我们就不知道当前所属的env。我们看一下如何获取对应的env。

```
    inline Environment*
Environment::GetCurrent(v8::Isolate* isolate) {
      v8::HandleScope handle_scope(isolate);
      return GetCurrent(isolate->GetCurrentContext());
    }
```

```
6
7       inline Environment*
8   Environment::GetCurrent(v8::Local<v8::Context> context) {
9         return static_cast<Environment*>(
              context-
    >GetAlignedPointerFromEmbedderData(ContextEmbedderIndex::kEr

      }
```

1.2 CreateProperties

接着我们看一下CreateProperties中创建process对象的逻辑。

```
1         Isolate* isolate = env->isolate();
2       EscapableHandleScope scope(isolate);
3       Local<Context> context = env->context();
4       // 申请一个函数模板
5       Local<FunctionTemplate> process_template =
6    FunctionTemplate::New(isolate);
7       process_template->SetClassName(env-
8    >process_string());
9       // 保存函数模板生成的函数
10      Local<Function> process_ctor;
11      // 保存函数模块生成的函数所新建出来的对象
12      Local<Object> process;
13      if (!process_template-
    >GetFunction(context).ToLocal(&process_ctor)||
    !process_ctor->NewInstance(context).ToLocal(&process)) {
          return MaybeLocal<Object>();
        }
```

process所保存的对象就是我们在JS层用使用的process对象。Node.js初始化的时候，还挂载了一些属性。

```
1     READONLY_PROPERTY(process,
2                       "version",
3                       FIXED_ONE_BYTE_STRING(env-
    >isolate(),
```

```
4
5   NODE_VERSION));
        READONLY_STRING_PROPERTY(process, "arch",
    per_process::metadata.arch);......
```

创建完process对象后，Node.js把process保存到env中。

```
1       Local<Object> process_object =
2   node::CreateProcessObject(this).FromMaybe(Local<Object>
    ());
        set_process_object(process_object)
```

1.3.3 初始化Libuv任务

```
1   InitializeLibuv函数中的逻辑是往Libuv中提交任务。
2       void Environment::InitializeLibuv(bool
3   start_profiler_idle_notifier) {
4         HandleScope handle_scope(isolate());
5         Context::Scope context_scope(context());
6         CHECK_EQ(0, uv_timer_init(event_loop(),
7   timer_handle()));
8         uv_unref(reinterpret_cast<uv_handle_t*>
9   (timer_handle()));
10        uv_check_init(event_loop(),
11  immediate_check_handle());
12        uv_unref(reinterpret_cast<uv_handle_t*>
13  (immediate_check_handle()));
14        uv_idle_init(event_loop(),
15  immediate_idle_handle());
16        uv_check_start(immediate_check_handle(),
17  CheckImmediate);
18        uv_prepare_init(event_loop(),
19  &idle_prepare_handle_);
20        uv_check_init(event_loop(), &idle_check_handle_);
21        uv_async_init(
22            event_loop(),
23            &task_queues_async_,
24            [](uv_async_t* async) {
```

```
            Environment* env = ContainerOf(
                &Environment::task_queues_async_,
async);
            env->CleanupFinalizationGroups();
            env->RunAndClearNativeImmediates();
          });
      uv_unref(reinterpret_cast<uv_handle_t*>
(&idle_prepare_handle_));
      uv_unref(reinterpret_cast<uv_handle_t*>
(&idle_check_handle_));
      uv_unref(reinterpret_cast<uv_handle_t*>
(&task_queues_async_));
      // …
    }
```

这些函数都是Libuv提供的，分别是往Libuv不同阶段插入任务节点，uv_unref是修改状态。

1 timer_handle是实现Node.js中定时器的数据结构，对应Libuv的time阶段

2 immediate_check_handle是实现Node.js中setImmediate的数据结构，对应Libuv的check阶段。

3 task_queues_async_用于子线程和主线程通信。

1.3.4 初始化Loader和执行上下文

RunBootstrapping里调用了BootstrapInternalLoaders和BootstrapNode函数，我们一个个分析。

1 初始化loader

BootstrapInternalLoaders用于执行internal/bootstrap/loaders.js。我们看一下具体逻辑。首先定义一个变量，该变量是一个字符串数组，用于定义函数的形参列表，一会我们会看到它的作用。

```
```

```
    std::vector<Local<String>> loaders_params = {
          process_string(),
          FIXED_ONE_BYTE_STRING(isolate_,
"getLinkedBinding"),
          FIXED_ONE_BYTE_STRING(isolate_,
"getInternalBinding"),
          primordials_string()};
```

然后再定义一个变量，是一个对象数组，用作执行函数时的实参。

```
    std::vector<Local<Value>> loaders_args = {
        process_object(),
        NewFunctionTemplate(binding::GetLinkedBinding)
            ->GetFunction(context())
            .ToLocalChecked(),
        NewFunctionTemplate(binding::GetInternalBinding)
            ->GetFunction(context())
            .ToLocalChecked(),
        primordials()};
```

接着Node.js编译执行internal/bootstrap/loaders.js，这个过程链路非常长，最后到V8层，就不贴出具体的代码，具体的逻辑转成JS如下。

```
    function demo(process,
                  getLinkedBinding,
                  getInternalBinding,
                  primordials) {
      // internal/bootstrap/loaders.js 的代码
    }
    const process = {};
    function getLinkedBinding(){}
    function getInternalBinding() {}
    const primordials = {};
    const export = demo(process,
                        getLinkedBinding,
                        getInternalBinding,
                        primordials);
```

V8把internal/bootstrap/loaders.js用一个函数包裹起来，形参就是loaders_params变量对应的四个字符串。然后执行这个函数，并且传入loaders_args里的那四个对象。internal/bootstrap/loaders.js会导出一个对象。在看internal/bootstrap/loaders.js代码之前，我们先看一下getLinkedBinding, getInternalBinding这两个函数，Node.js在C++层对外暴露了AddLinkedBinding方法注册模块，Node.js针对这种类型的模块，维护了一个单独的链表。getLinkedBinding就是根据模块名从这个链表中找到对应的模块，但是我们一般用不到这个，所以就不深入分析。前面我们看到对于C++内置模块，Node.js同样维护了一个链表，getInternalBinding就是根据模块名从这个链表中找到对应的模块。现在我们可以具体看一下internal/bootstrap/loaders.js的代码了。

```
    let internalBinding;
    {
      const bindingObj = ObjectCreate(null);
      internalBinding = function internalBinding(module)
{
        let mod = bindingObj[module];
        if (typeof mod !== 'object') {
          mod = bindingObj[module] =
getInternalBinding(module);
          moduleLoadList.push(`Internal Binding
${module}`);
        }
        return mod;
      };
    }
```

Node.js在JS对getInternalBinding进行了一个封装，主要是加了缓存处理。

```
    const internalBindingWhitelist = new SafeSet([,
      'tcp_wrap',
      // 一系列C++内置模块名
    ]);

    {
      const bindingObj = ObjectCreate(null);
      process.binding = function binding(module) {
```

```
        module = String(module);
        if (internalBindingWhitelist.has(module)) {
          return internalBinding(module);
        }
        throw new Error(`No such module: ${module}`);
      };
    }
```

在process对象（就是我们平时使用的process对象）中挂载binding函数，这个函数主要用于内置的JS模块，后面我们会经常看到。binding的逻辑就是根据模块名查找对应的C++模块。上面的处理是为了Node.js能在JS层通过binding函数加载C++模块，我们知道Node.js中还有原生的JS模块（lib文件夹下的JS文件）。接下来我们看一下，对于加载原生JS模块的处理。Node.js定义了一个NativeModule类负责原生JS模块的加载。还定义了一个变量保存了原生JS模块的名称列表。

```
static map = new Map(moduleIds.map((id) => [id, new
NativeModule(id)]));
```

NativeModule主要的逻辑如下

1 原生JS模块的代码是转成字符存在node_javascript.cc文件的，NativeModule负责原生JS模块的加载，即编译和执行。 2 提供一个require函数，加载原生JS模块，对于文件路径以internal开头的模块，是不能被用户require使用的。

这是原生JS模块加载的大概逻辑，具体的我们在Node.js模块加载章节具体分析。执行完internal/bootstrap/loaders.js，最后返回三个变量给C++层。

```
    return {
      internalBinding,
      NativeModule,
      require: nativeModuleRequire
    };
```

C++层保存其中两个函数，分别用于加载内置C++模块和原生JS模块的函数。

```
1
2   set_internal_binding_loader(internal_binding_loader.As<Funct
    ());
        set_native_module_require(require.As<Function>());
```

至此，internal/bootstrap/loaders.js分析完了。

2 初始化执行上下文

BootstrapNode负责初始化执行上下文，代码如下

```
1       EscapableHandleScope scope(isolate_);
2       // 获取全局变量并设置global属性
3       Local<Object> global = context()->Global();
4       global->Set(context(),
5   FIXED_ONE_BYTE_STRING(isolate_, "global"),
6   global).Check();
7       /*
8         执行internal/bootstrap/node.js时的参数
9         process, require, internalBinding, primordials
10      */
11      std::vector<Local<String>> node_params = {
12          process_string(),
13          require_string(),
14          internal_binding_string(),
15          primordials_string()};
16      std::vector<Local<Value>> node_args = {
17          process_object(),
18          // 原生模块加载器
19          native_module_require(),
20          // C++模块加载器
21          internal_binding_loader(),
22          primordials()};
23
        MaybeLocal<Value> result = ExecuteBootstrapper(
            this, "internal/bootstrap/node", &node_params,
    &node_args);
```

在全局对象上设置一个global属性，这就是我们在Node.js中使用的global对象。接着执行internal/bootstrap/node.js设置一些变量（具体可以参考nternal/bootstrap/node.js）。

```
    process.cpuUsage = wrapped.cpuUsage;
    process.resourceUsage = wrapped.resourceUsage;
    process.memoryUsage = wrapped.memoryUsage;
    process.kill = wrapped.kill;
    process.exit = wrapped.exit;
```

设置全局变量

```
    defineOperation(global, 'clearInterval',
timers.clearInterval);
    defineOperation(global, 'clearTimeout',
timers.clearTimeout);
    defineOperation(global, 'setInterval',
timers.setInterval);
    defineOperation(global, 'setTimeout',
timers.setTimeout);
    ObjectDefineProperty(global, 'process', {
      value: process,
      enumerable: false,
      writable: true,
      configurable: true
    });
```

1.3.5 执行用户JS文件

StartMainThreadExecution进行一些初始化工作，然后执行用户JS代码。

1 给process对象挂载属性

执行patchProcessObject函数（在node_process_methods.cc中导出）给process对象挂载一些列属性，不一一列举。

```
```

```
    // process.argv
    process->Set(context,
                         FIXED_ONE_BYTE_STRING(isolate,
"argv"),
                 ToV8Value(context, env-
>argv()).ToLocalChecked()).Check();

    READONLY_PROPERTY(process,
                      "pid",
                 Integer::New(isolate, uv_os_getpid()));
```

因为Node.js增加了对线程的支持，有些属性需要hack一下，比如在线程里使用process.exit的时候，退出的是单个线程，而不是整个进程，exit等函数需要特殊处理。后面章节会详细讲解。

2 处理进程间通信

```
    function setupChildProcessIpcChannel() {
      if (process.env.NODE_CHANNEL_FD) {
        const fd = parseInt(process.env.NODE_CHANNEL_FD,
10);
        delete process.env.NODE_CHANNEL_FD;
        const serializationMode =
    process.env.NODE_CHANNEL_SERIALIZATION_MODE ||
'json';
        delete
process.env.NODE_CHANNEL_SERIALIZATION_MODE;
        require('child_process')._forkChild(fd,
serializationMode);
      }
    }
```

环境变量NODE_CHANNEL_FD是在创建子进程的时候设置的，如果有说明当前启动的进程是子进程，则需要处理进程间通信。

3 处理cluster模块的进程间通信

```
```

```
    function initializeclusterIPC() {
      if (process.argv[1] && process.env.NODE_UNIQUE_ID)
{
        const cluster = require('cluster');
        cluster._setupWorker();
        delete process.env.NODE_UNIQUE_ID;
      }
    }
```

4 执行用户JS代码

```
require('internal/modules/cjs/loader').Module.runMain(proces
```

internal/modules/cjs/loader.js是负责加载用户JS的模块，runMain函数在pre_execution.js被挂载，runMain做的事情是加载用户的JS，然后执行。具体的过程在后面章节详细分析。

1.3.6 进入Libuv事件循环

执行完所有的初始化后，Node.js执行了用户的JS代码，用户的JS代码会往Libuv注册一些任务，比如创建一个服务器，最后Node.js进入Libuv的事件循环中，开始一轮又一轮的事件循环处理。如果没有需要处理的任务，Libuv会退出，从而Node.js退出。

```
    do {
        uv_run(env->event_loop(), UV_RUN_DEFAULT);
        per_process::v8_platform.DrainVMTasks(isolate_);
        more = uv_loop_alive(env->event_loop());
        if (more && !env->is_stopping()) continue;
        if (!uv_loop_alive(env->event_loop())) {
            EmitBeforeExit(env.get());
        }
        more = uv_loop_alive(env->event_loop());
     } while (more == true && !env->is_stopping());
```

1.4 Node.js和其它服务器的比较

服务器是现代软件中非常重要的一个组成，我们看一下服务器发展的过程中，都有哪些设计架构。一个基于TCP协议的服务器，基本的流程如下（伪代码）。

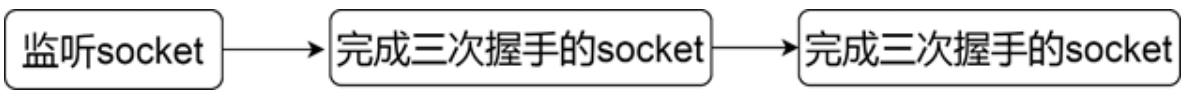
```
    // 拿到一个socket用于监听
    const socketfd = socket(协议类型等配置);
    // 监听本机的地址（ip+端口）
    bind(socketfd,  监听地址)
    // 标记该socket是监听型socket
    listen(socketfd)
```

执行完以上步骤，一个服务器正式开始服务。下面我们看一下基于上面的模型，分析各种各样的处理方法。

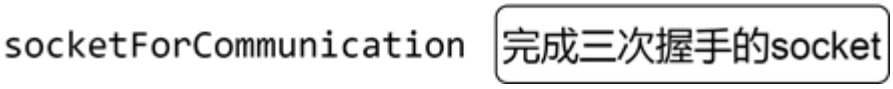
1.4.1 串行处理请求

```
    while(1) {
        const socketForCommunication = accept(socket);
        const data = read(socketForCommunication);
        handle(data);
        write(socketForCommunication, data );
    }
```

我们看看这种模式的处理过程，假设有n个请求到来。那么socket的结构如下图所示。



这时候进程从accept中被唤醒。然后拿到一个新的socket用于通信。结构如下图所示。



accept就是从已完成三次握手的连接队列里，摘下一个节点。很多同学都了解三次握手是什么，但是可能很少同学会深入思考或者看它的实现，众所周知，一个服务器启动的时候，会监听一个端口，其实就是新建了一个socket。那么如果有一个连接到来的时候，我们通过accept就能拿到这个新连接对应的socket，那这个socket和监听的socket是不是同一个呢？其实socket分为监听型和通信型的，表面上，服务器用一个端口实现了多个连接，但是这个端口是用于监听的，底层用于和客户端通信的其实是另一个socket。所以每一个连接过来，负责监听的socket发现是一个建立连接的包（syn包），它就会生成一个新的socket与之通信（accept的时候返回的那个）。监听socket里只保存了它监听的IP和端口，通信socket首先从监听socket中复制IP和端口，然后把客户端的IP和端口也记录下来，当下次收到一个数据包的时候，操作系统就会根据四元组从socket池子里找到该socket，从而完成数据的处理。

串行这种模式就是从已完成三次握手的队列里摘下一个节点，然后处理。再摘下一个节点，再处理。如果处理的过程中有阻塞式IO，可想而知，效率是有多低。而且并发量比较大的时候，监听socket对应的队列很快就会被占满（已完成连接队列有一个最大长度）。这是最简单的模式，虽然服务器的设计中肯定不会使用这种模式，但是它让我们了解了一个服务器处理请求的整体过程。

1.4.2 多进程模式

串行模式中，所有请求都在一个进程中排队被处理，这是效率低下的原因。这时候我们可以把请求分给多个进程处理来提供效率，因为在串行处理的模式中，如果有阻塞式IO操作，它就会阻塞主进程，从而阻塞后续请求的处理。在多进程的模式下，一个请求如果阻塞了进程，那么操作系统会挂起该进程，接

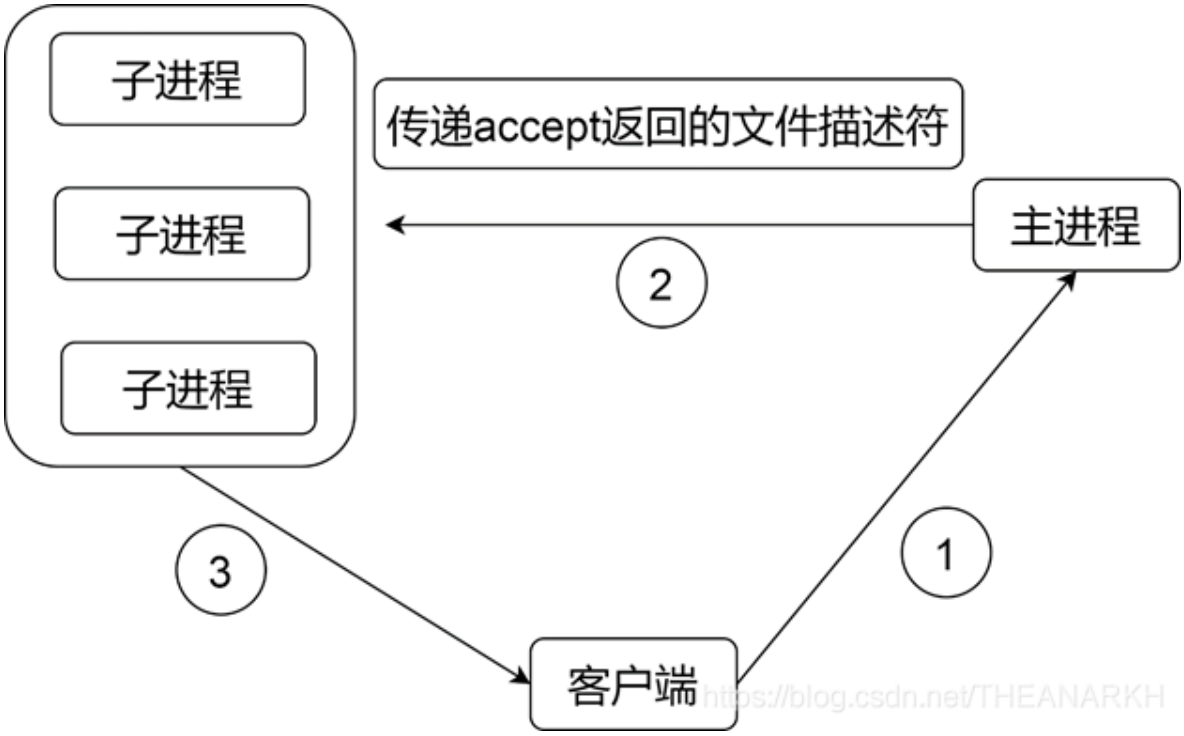
着调度其它进程执行，那么其它进程就可以执行新的任务。多进程模式下分为几种。

1 主进程accept，子进程处理请求 这种模式下，主进程负责摘取已完成连接的节点，然后把这个节点对应的请求交给子进程处理，逻辑如下。

```
    while(1) {
        const socketForCommunication = accept(socket);
        if (fork() > 0) {
             continue;
            // 父进程
        } else {
            // 子进程
            handle(socketForCommunication);
        }
    }
```

这种模式下，每次来一个请求，就会新建一个进程去处理。这种模式比串行的稍微好了一点，每个请求独立处理，假设a请求阻塞在文件IO，那么不会影响b请求的处理，尽可能地做到了并发。它的瓶颈就是系统的进程数有限，如果有大量的请求，系统无法扛得住，再者，进程的开销很大，对于系统来说是一个沉重的负担。

2 进程池模式 实时创建和销毁进程开销大，效率低，所以衍生了进程池模式，进程池模式就是服务器启动的时候，预先创建一定数量的进程，但是这些进程是worker进程。它不负责accept请求。它只负责处理请求。主进程负责accept，它把accept返回的socket交给worker进程处理，模式如下图所示。



但是和1中的模式相比，进程池模式相对比较复杂，因为在模式1中，当主进程收到一个请求的时候，实时fork一个子进程，这时候，这个子进程会继承主进程中新请求对应的fd，所以它可以直接处理该fd对应的请求，在进程池的模式中，子进程是预先创建的，当主进程收到一个请求的时候，子进程中是无法拿得到该请求对应的fd的。这时候，需要主进程使用传递文件描述符的技术把这个请求对应的fd传给子进程。一个进程其实就是一个结构体task_struct，在JS里我们可以说是一个对象，它有一个字段记录了打开的文件描述符，当我们访问一个文件描述符的时候，操作系统就会根据fd的值，从task_struct中找到fd对应的底层资源，所以主进程给子进程传递文件描述符的时候，传递的不仅仅是一个数字fd，因为如果仅仅这样做，在子进程中该fd可能没有对应任何资源，或者对应的资源和主进程中的是不一致的。这其中操作系统帮我们做了很多事情。让我们在子进程中可以通过fd访问到正确的资源，即主进程中收到的请求。

3 子进程accept

这种模式不是等到请求来的时候再创建进程。而是在服务器启动的时候，就会创建多个进程。然后多个进程分别调用accept。这种模式的架构如图1-8所示。


```
    const socketfd = socket(协议类型等配置);
    bind(socketfd,  监听地址)

    for (let i = 0 ; i < 进程个数; i++) {
        if (fork() > 0) {
            // 父进程负责监控子进程
        } else {
            // 子进程处理请求
              listen(socketfd);
            while(1) {
                const socketForCommunication =
accept(socketfd);
                handle(socketForCommunication);
            }
        }
    }
```

这种模式下多个子进程都阻塞在accept。如果这时候有一个请求到来，那么所有的子进程都会被唤醒，但是首先被调度的子进程会首先摘下这个请求节点，后续的进程被唤醒后可能会遇到已经没有请求可以处理，又进入睡眠，进程被无效唤醒，这是著名的惊群现象。改进方式就是在accpet之前加锁，拿到锁的进程才能进行accept，这样就保证了只有一个进程会阻塞在accept，Nginx解决了这个问题，但是新版操作系统已经在内核层面解决了这个问题。每次只会唤醒一个进程。通常这种模式和事件驱动配合使用。

1.4.3 多线程模式

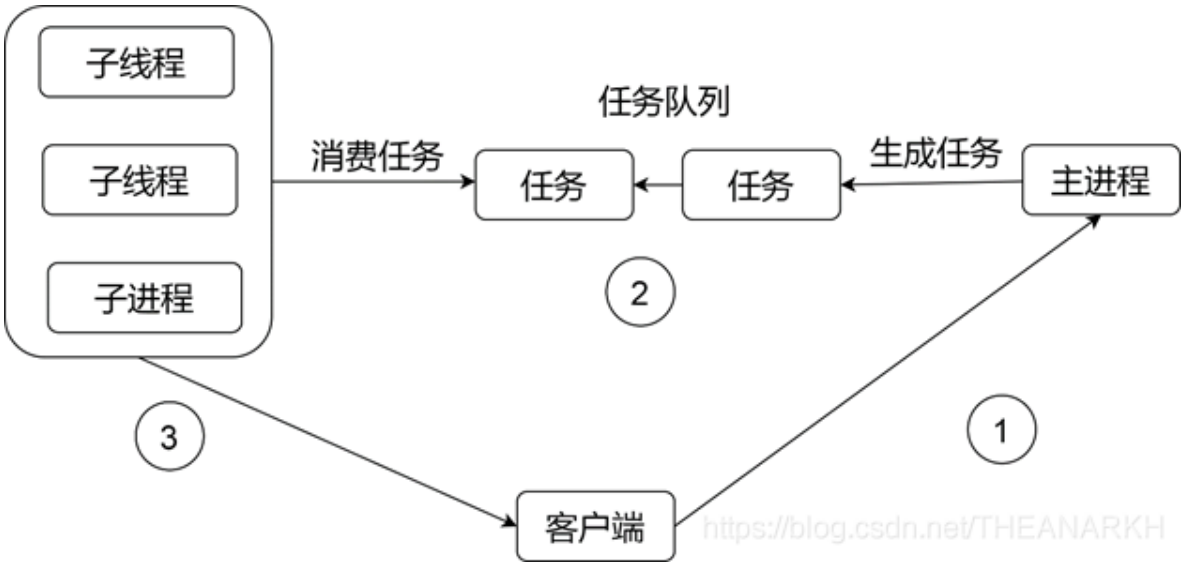
多线程模式和多进程模式是类似的，也是分为下面几种

1 主进程accept，创建子线程处理

2 子线程accept

3 线程池

前面两种和多进程模式中是一样的，但是第三种比较特别，我们主要介绍第三种。在子进程模式时，每个子进程都有自己的task_struct，这就意味着在fork之后，每个进程负责维护自己的数据，而线程则不一样，线程是共享主线程（主进程）的数据的，当主进程从accept中拿到一个fd的时候，传给线程的话，线程是可以直接操作的。所以在线程池模式时，架构如下图所示。



主进程负责accept请求，然后通过互斥的方式插入一个任务到共享队列中，线程池中的子线程同样是通过互斥的方式，从共享队列中摘取节点进行处理。

1.4.4 事件驱动

现在很多服务器（Nginx，Node.js，Redis）都开始使用事件驱动模式去设计。从之前的设计模式中我们知道，为了应对大量的请求，服务器需要大量的进程/线程。这个是个非常大的开销。而事件驱动模式，一般是配合单进程（单线程），再多的请求，也是在一个进程里处理的。但是因为是单进程，所以不适合CPU密集型，因为一个任务一直在占据CPU的话，后续的任务就无法执行

了。它更适合IO密集的（一般都会提供一个线程池，负责处理CPU或者阻塞型的任务）。而使用多进程/线程模式的时候，一个进程/线程是无法一直占据CPU的，执行一定时间后，操作系统会执行任务调度。让其它线程也有机会执行，这样就不会前面的任务阻塞后面的任务，出现饥饿情况。大部分操作系统都提供了事件驱动的API。但是事件驱动在不同系统中实现不一样。所以一般都会有一层抽象层抹平这个差异。这里以Linux的epoll为例子。

```
    // 创建一个epoll
    var epollFD = epoll_create();
    /*
     在epoll给某个文件描述符注册感兴趣的事件，这里是监听的socket，注册可
     读事件，即连接到来
     event = {
        event: 可读
        fd:  监听socket
        // 一些上下文
     }
    */
    epoll_ctl(epollFD , EPOLL_CTL_ADD , socket, event);
    while(1) {
        // 阻塞等待事件就绪，events保存就绪事件的信息，total是个数
        var total= epoll_wait(epollFD , 保存就绪事件的结构events, 事件个数, timeout);
        for (let i = 0; i < total; i++) {
            if (events[i].fd === 监听socket) {
                var newSocket = accpet(socket);
                /*
                   把新的socket也注册到epoll，等待可读，
                   即可读取客户端数据
                 */
                epoll_ctl(epollFD,
                          EPOLL_CTL_ADD,
                          newSocket,
                          可读事件);
            } else {
                // 从events[i]中拿到一些上下文，执行相应的回调
```

```
                }
            }
        }
```

这就是事件驱动模式的大致过程，本质上是一个订阅/发布模式。服务器通过注册文件描述符和事件到epoll中，epoll开始阻塞，等到epoll返回的时候，它会告诉服务器哪些fd的哪些事件触发了，这时候服务器遍历就绪事件，然后执行对应的回调，在回调里可以再次注册新的事件，就是这样不断驱动着。epoll的原理其实也类似事件驱动，epoll底层维护用户注册的事件和文件描述符，epoll本身也会在文件描述符对应的文件/socket/管道处注册一个回调，然后自身进入阻塞，等到别人通知epoll有事件发生的时候，epoll就会把fd和事件返回给用户。

```
    function epoll_wait() {
        for 事件个数
            // 调用文件系统的函数判断
            if (事件[i]中对应的文件描述符中有某个用户感兴趣的事件发生?) {
                插入就绪事件队列
            } else {
              /*
                 在事件[i]中的文件描述符所对应的文件/socket/管道等indeo节
                 点注册回调。即感兴趣的事件触发后回调epoll，回调epoll后，
                epoll把该event[i]插入就绪事件队列返回给用户
              */
            }
    }
```

以上就是服务器设计的一些基本介绍。现在的服务器的设计中还会涉及到协程。不过目前还没有看过具体的实现，所以暂不展开介绍，有兴趣的通信可以看一下协程库libtask了解一下如何使用协程实现一个服务器。 Node.js是基于单进程（单线程）的事件驱动模式。这也是为什么Node.js擅长处理高并发IO型任务而不擅长处理CPU型任务的原因，Nginx、Redis也是这种模式。另外Node.js

是一个及web服务器和应用服务器于一身的服务器，像Nginx这种属于web服务器，它们只处理HTTP协议，不具备脚本语言来处理具体的业务逻辑。它需要把请求转发到真正的web服务器中去处理，比如PHP。而Node.js不仅可以解析HTTP协议，还可以处理具体的业务逻辑。