HP

2021年11月18日　　11:02

==========

https://zhuanlan.zhihu.com/p/262946937

https://zhuanlan.zhihu.com/p/498546422

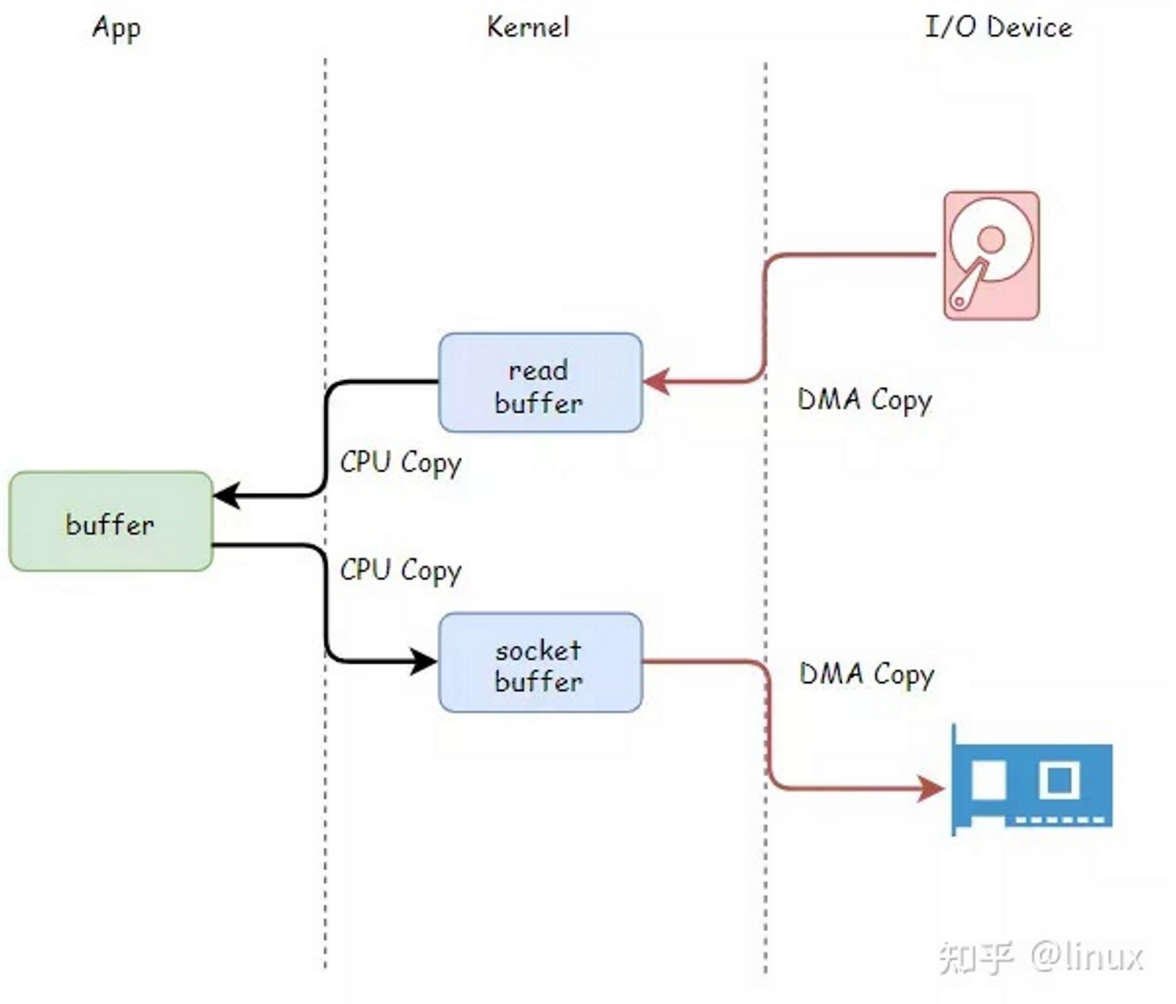
十大高性能开发

从内存、磁盘I/O、网络I/O、CPU、缓存、架构、算法等多层次递进，串联起高性能开发十大必须掌握的核心技术。

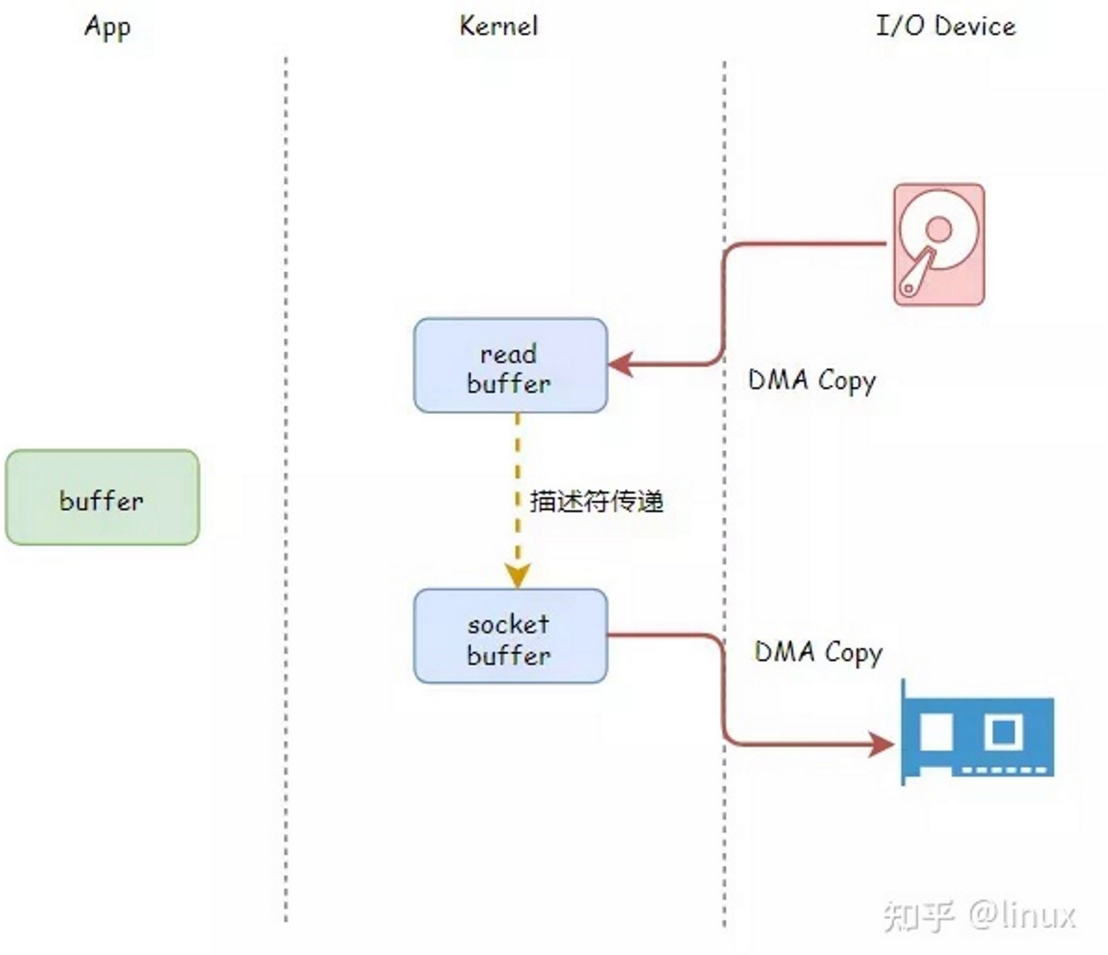
- I/O优化：零拷贝技术
- I/O优化：多路复用技术
- 线程池技术
- 无锁编程技术
- 进程间通信技术
- RPC && 序列化技术
- 数据库索引技术
- 缓存技术 && 布隆过滤器
- 全文搜索技术
- 负载均衡技术

需要开发一个静态web服务器，把磁盘文件(网页，图片)通过网络发出去。
最简单的：
　　主程序进入循环，等待连接
　　来一个连接 就启动一个 工作线程 来处理
　　工作线程 等待对方的请求，然后从磁盘 读取文件，往socket中发数据。

IO优化：零拷贝技术
上面的工作线程，从磁盘读文件，然后通过网络发送数据，数据 从 磁盘 到网络，需要 拷贝4次。 其中 CPU 需要亲自搬运2次。



零拷贝技术，释放 CPU，文件数据直接从 内核发出去，无需再拷贝到 应用程序 缓冲区。



Linux API：

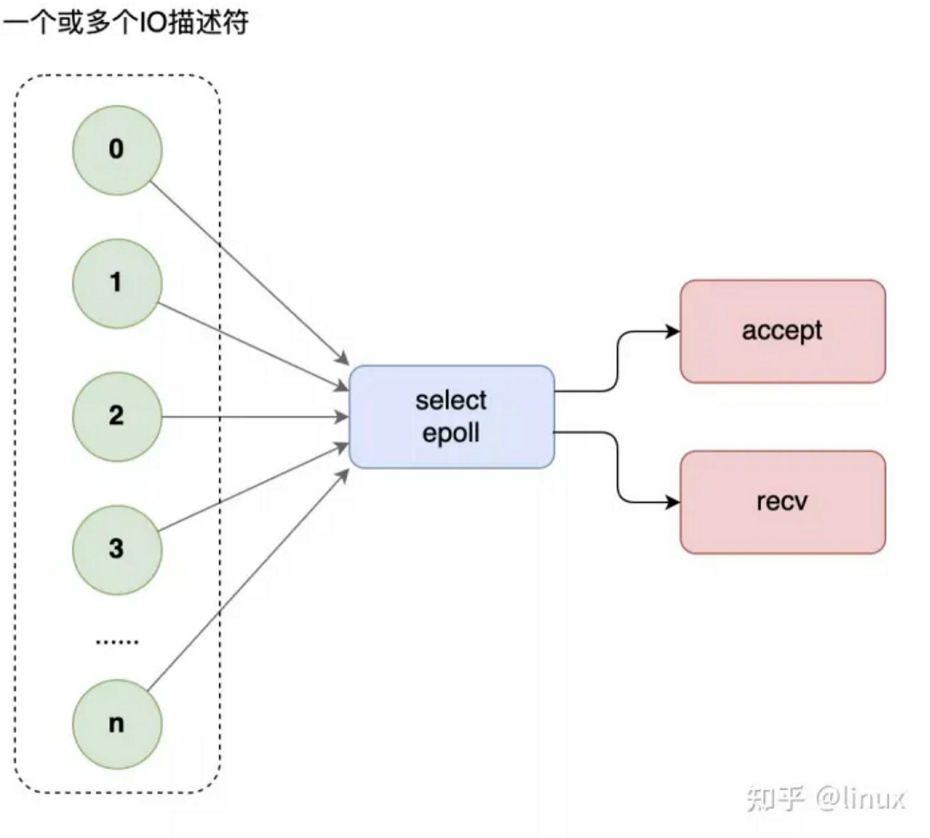
```
ssize_t sendfile(
  int out_fd,
  int in_fd,
  off_t *offset,
  size_t count
);
```

指定要发送的文件描述符 和 网络套接字描述符。
。。。window 有吗？

访问人多以后，又变慢了。这时需要
IO优化：多路复用技术

前面的版本中，每个线程都要阻塞在 recv 等待 对方的请求，访问人数多了，线程 就多了。大量线程 都在 阻塞，系统运算速度 就随之下降。

这时需要 多路复用技术， 使用 select 模型，将所有等待 (accept, recv) 都放在 主线程中，工作线程 不需要等待。



人数越来越多，select 也不行的时候，就需要 升级 多路复用模型为 epoll

select有 3弊， epoll有 3优

select 底层使用 数组 来管理 套接字描述符，同时管理的 数量有上限， 一般不超过几千个，epoll 使用 树 和 链表 来管理，同时管理数量可以很大。

select 不会告诉你 到底哪个套接字 来了信息，你需要一个个去询问。 epoll 直接告诉你 谁来了 消息，不用轮询。

select 进行系统调用时 还需要 把 套接字列表 在用户控件 和 内核空间 来回 拷贝，循环中 调用 select 时 简直浪费。 epoll 统一在 内核管理套接字 描述符，无需来回拷贝。

之前，工作线程 总是 用的时候创建，用完关闭。 大量请求来的时候，线程不断创建，关闭。 这时候需要

线程池技术

我们可以在程序 启动的时候 就批量启动一批 工作线程，而不是由请求的时候 才创建，使用一个 公共的任务队列， 请求来了 以后 放入 队列中， 各个工作线程 统一从 队列中 不断地 取出任务 来处理，这就是 线程池技术。

多线程技术 的 使用 一定程序 提升了 服务器的 并发能力， 但同时，多个线程 之间 为了数据同步，常常需要 使用 互斥体，信号，条件变量 等 手段 来同步 多个线程。这些 重量级 的同步手段 往往 会导致 线程 在 用户态/内核态 多次切换， 系统调用，线程切换 都是不小的开销。

在线程池技术中，提高了 一个 公共的 任务队列，各个工作线程 要从中 提取 任务 进行处理，这里 就 涉及到 多个工作线程 对这个 公共队列的 同步操作。

无锁编程技术
多线程 并发 编程中，遇到公共数据 就需要 进行 线程同步。 这里 同步 又分为 阻塞型同步 和 非阻塞型同步。

阻塞型同步 好理解，我们常用的 互斥体，信号，条件变量 等 这些操作 系统提供的机制 都属于 阻塞型同步， 其本质 都是 要加 "锁"

与之对应的 非阻塞型同步，就是 在无锁的情况下实现同步， 目前有 3类技术方案：

- wait-free
- lock-free
- obstruction-free

3类技术 方案 都是通过 一定的 算法 和 技术手段 来实现 不用 阻塞等待 而实现 同步，其中 lock-free 最为广泛。

lock-free 能广泛 应用，得益于 目前主流的 CPU 都提供了 原子级别的 read-modify-write 原语，这就是 著名的 CAS 操作。在 Interl X86 系列处理器上，就是 cmpxchg 系列指令

```
// 通过CAS操作实现Lock-free
do {
  ...
} while(!CAS(ptr, old_data, new_data ))
```

我们常见到的 无锁队列，无锁列表，无锁HashMap 等，其无锁的核心大多来源于此。

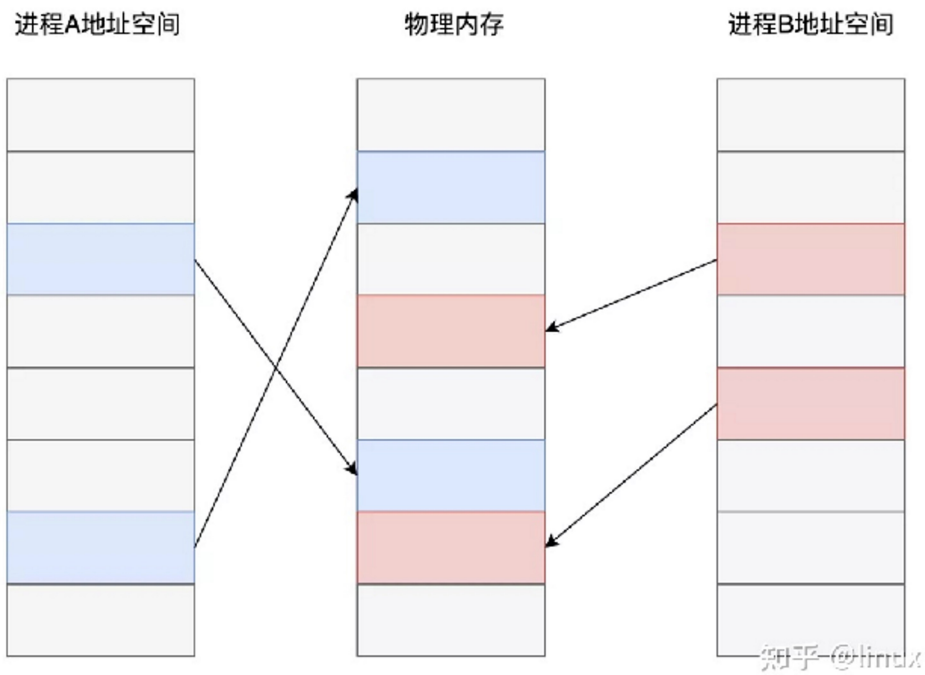
发现服务进程崩溃，排查发现是 工作线程 代码bug， 一崩溃 整个服务就不可用了， 所以决定 把 工作线程 和 主线程 拆开 到不同的 进程中， 工作线程的崩溃 不影响 整体的服务。这个时候 出现了 多线程，你需要

进程间通信技术
进程间通信，你能想到什么？

- 管道
- 命名管道
- socket
- 消息队列
- 信号
- 信号量
- 共享内存

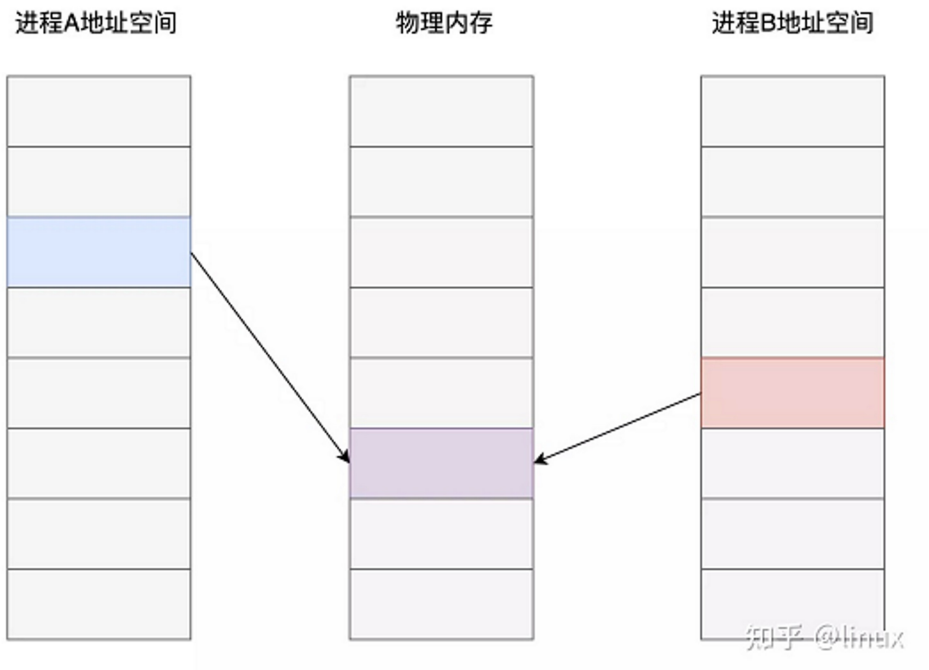
对于 本地进程间 需要 高频次 的 大量数据交互，首推 共享内存这种方案。

现代 OS 普遍采用了 基于 虚拟内存的 管理方案，在这种内存管理方式下，各个线程间 进行了 强制隔离。
程序代码中 使用的 内存地址 均是 一个 虚拟地址，由 OS 的内存管理 算法 提前 分配 映射到 对应的 物理内存页面， CPU 在执行代码指令时， 对访问到的 内存地址 再进行 实时的 转换翻译。



从上图可以看出，不同的进程之间，虽然是 同一个内存地址，最终 在 OS 和 CPU 的 配合下， 实际 存储数据的 内存页面 是不同的。
。。really？ 感觉是 什么 用户态，或者说 线/进程上下文 ,但是 CPP直接操作内存地址的啊。

共享内存 这种 进程间 通信 方案的 核心在于： 如果 让同一个 物理内存page 映射到 2个进程地址空间中， 双方就可以直接 读写，无需拷贝了。



当然，共享内存 只是 最终的 数据传输载体，双方要实现通信 还是得 借助 信号，信号量等其他 通知机制。

增加了一台服务器 用来 提供 交互服务， 原先的服务器 还是 静态内容 的服务器。
动态服务 和 静态服务 之间 进程需要通信。

使用 RESTful 的话，JSON 传输效率低。 你需要 更高效的 通信方案

RPC && 序列化技术

RPC， remote procedure call， 远程过程调用。

通过网络 进行 功能调用，涉及 参数的 打包 解包，网络的 传输，结果的 打包解包 等。
其中 对 数据 进行 打包 解包 就需要 序列化技术。

序列化，简单来说，就是，将内存中的对象转换成 可以传输 和 存储的数据，而这个过程的逆向操作就是 反序列化

衡量序列化框架的指标有：
- 是否跨语言支持，支持哪些语言
- 是否单纯的 序列化功能，包不包含 RPC 框架
- 序列化传输能力
- 扩展支持能力(数据对象 增删字段后，前后的 兼容性)
- 是否支持动态解析(动态解析是指，不需要提前编译，根据拿到的数据 格式定义文件 立刻就能解析)

下面是 3大流行的序列化框架：Protobuf，Thrift，Avro 的对比

Protobuf
Google出品
支持 C++ Java Py 等
动态性较差，一般需要提前编译
不包含RPC，不过可以和 同为 Google 的 gRPC 一起使用。

Thrift
Facebook 出品
支持 C++ Java Py PHP Go C# JS 等
动态性差
包含RPC
这是 facebook 出品的 RPC 框架，本身内含了 二进制序列化方案，但 Thrift 本身 的RPC 和 数据序列化 是解耦的，你甚至可以 选择 XML JSON 等数据格式。

Avro
支持 C C++ Java Py C# 等
动态性 好
包含RPC
源自 Hadoop 生态中的 序列化框架，自带 RPC 框架，也可以独立使用。
最大的优势就是 支持 动态数据解析。

为什么一直提到 动态解析呢？因为 之前的 一段项目经历中，就遇到了 3种技术的选型。需

要 一个 C++ 开发的服务 和 一个 Java 开发的服务进行 RPC

Protobuf，Thrift 都需要 通过 "编译" 将对应的 数据协议 定义文件 编译成 C++ Java 源码，然后 放入工程中 一起编译，从而进行解析。 项目方 不愿意 使用 编译出来的源码，就需要使用 Avro

数据查询速度越来越慢
需要
数据库索引技术
索引分类
- 主键索引
- 聚集索引
- 非聚集索引

聚集索引是指 索引的 逻辑顺序 与 表记录的 物理存储 顺序一致的 索引，一般情况下，主键索引 符合这个定义，所以一般来说 主键索引 也是 聚集索引。但是，这不是绝对的，在不同 的数据库中，或者 在 同一个数据库下的 不同存储引擎 中 还是有不同。

聚集索引的叶子节点 直接存储了 数据，也是 数据节点， 而 非聚集索引 的叶子节点 没有存储 实际的 数据，需要二次查询。

索引的实现原理
索引的实现主要有3种
- B+ 树
- 哈希表
- 位图

B+树最多，特点是 树的节点众多，相较于 二叉树，这是一颗多叉树，是一个 扁平的 胖树，减少树的深度有利于减少 磁盘 IO次数。

hash表实现的索引 也叫做 散列索引，通过 hash 函数 来实现 数据的定位。hash算法的特点是 快，常数级别的 时间复杂度，缺点是 只适合 精确匹配，不适合 模糊匹配 和 范围搜索。

位图索引 相对少见。想象这么一个场景，如果 某个字段的 取值 只有 有限的几种，如 性别，省份，血型 等，针对这样的 字段 如果用 B+树 作为索引的话，会出现 大量 索引值相同的叶子结点，是一种 存储的浪费。

位图索引 正是基于 这一点进行优化， 针对 字段取值 只有少量 有限项，数据表中该列字段 出现 大量重复时， 就是位图索引 一展身手的 实际。

位图，就是 Bitmap，基本思想是 对该字段的 每一个取值 建立一个 二进制位图 来标记 数据表的 每一条记录 的该列 字段 是否是 对应取值。

用户增多后，数据库的 瓶颈出现了。
缓存技术 && 布隆过滤器

从物理 CPU 对内存数据的 缓存 到 浏览器 对网页内容的 缓存， 缓存技术 遍布于 计算机世界的 每个角落。
面对当前出现的数据库瓶颈，同样可以用缓存技术来解决。
每次访问数据库都需要 数据库进行查表(当然，数据库自身也有优化措施)，反映到底层就是进行一次 或多次 的磁盘 IO，但凡涉及 IO 的就会慢下来。如果是一些频繁用到 但又不会经常变化 的数据，何不将其缓存在内存中，不必每一次都要 找到 数据库要，从而减轻对数据库的压力呢？

memcached 和 redis 为代表的 内存对象 缓存系统 应运而生：
缓存系统的三个著名问题
缓存穿透：数据库中不存在，所以没有缓存，所以 始终会请求数据库
缓存击穿：热点数据过期，大量请求 直接到 数据库
缓存雪崩：大量数据过期，大量请求 直接到 数据库

如何判断我们要的数据是不是在缓存系统中呢？
即，如何快速判断一个数据量很大的 集合 中是否包含 我们指定的数据？

布隆过滤器
布隆过滤器 说 不存在 就肯定不存在， 说存在，可能存在 也可能不存在

网站内容增多，用户对于 快速全站搜索 的需求 日益强烈。需要：
全文搜索技术
简单的查询条件，传统的关系型数据库还可以应付。
但搜索需求一旦复杂起来，比如根据 文章内容关键字，多个搜索条件的逻辑组合 等情况下，数据库就 捉襟见肘了，这时需要 单独的 索引系统 来进行支持

广泛使用的 ElasticSearch 就是一套强大的 搜索引擎。集 全文检索，数据分析，分布式部署 等优点于一身，成为企业级搜索技术首选。

ES使用 RESTful 接口，使用JSON 作为数据传输格式，支持多种查询匹配，为各 主流语言 都提供了 SDK。

ES 常常和另外2个 开源软件： Logstash， Kibana 一起，形成一套 日志 收集，分析，展现的 完整解决方案： ELK 架构

Logstash 负责 数据的收集，解析， ElasticSearch 负责搜索，Kibana 负责 可视化交互

一台服务器的力量终究是有限的，所以 一个服务 由 多台服务器来提供服务，需要将 用户的请求 均衡的分摊到 各个服务器上，这时，需要

负载均衡技术
将负载 均匀地 分配到 各个业务节点上去。
和缓存一样，负载均衡 同样 存在于 计算机世界 各个角落。

按照均衡实现实体，可以分为 软件负载均衡(如 LVS，Nginx，HAProxy) 和 硬件负载均衡(如

A10， F5)
按照网络层次，可以分为 四层负载均衡(基于网络连接) 和 七层负载均衡(基于应用内容)
按照均衡策略算法，可以分为 轮询，哈希，权重，随机 均衡 或 几种算法的结合

目前使用 Nginx 来实现负载均衡，Nginx 支持 轮询，权重，IP哈希，最少连接数，最短响应时间 等 多种方式的 负载均衡配置。

轮询

```
upstream web-server {
    server 192.168.1.100;
    server 192.168.1.101;
}
```

权重

```
upstream web-server {
    server 192.168.1.100 weight=1;
    server 192.168.1.101 weight=2;
}
```

IP哈希

```
upstream web-server {
    ip_hash;
    server 192.168.1.100 weight=1;
    server 192.168.1.101 weight=2;
}
```

最少连接数

```
upstream web-server {
    least_conn;
    server 192.168.1.100 weight=1;
    server 192.168.1.101 weight=2;
}
```

最短响应时间

```
upstream web-server {
    server 192.168.1.100 weight=1;
    server 192.168.1.101 weight=2;
    fair;
}
```

==========

用户空间，内核空间
user land, kernel land

==========

高并发 解决方法(百度搜索的)

系统拆分
将一个系统拆分为 多个子系统，使用dubbo通信。 每个子系统 连一个数据库，这样从原先的单个库，到现在的 多个数据库。可以抗住高并发

cache
大部分高并发场景，都是 读多写少。

要考虑 哪些代码承担了 大量的读。

MQ
高并发写， 无法使用cache。
。。。感觉可以使用异步型cache(就是读写都在cache上，有专门线程来 将cache 回写到DB，不过 只适用于 对数据安全性不高的场景，因为cache崩溃，会丢失数据)

使用MQ，大量的写请求 都发送到 MQ中，MQ的消费者 执行 写。
。。异步写

分库分表
分库 提高并发
分表 减少sql耗时

读写分离
读多写少，主从架构，主库写入，从库读取。 读请求多，可以增加 从库。

横向扩展
分布式，将流量分开，让每个服务器都承担 一部分的 并发和流量。

异步
某些场景下，未处理完之前，可以向让 请求返回，在 数据准备好后 再通知 请求方。

限流

服务降级，熔断。

==========

epoll

==========

==========

无锁同步

wait-free
lock-free
obstruction-free

==========

零拷贝

==========

线程间通信

==========

进程间通信

https://zhuanlan.zhihu.com/p/94856678

==========

==========

==========

==========

【高并发系统设计原则（一）】
https://www.bilibili.com/video/BV1aP4y1X7AE?vd_source=ef3f191195033676cbe884d8ec6cca51

。。vd_source 是不是我的ID?

许多并发系统 基本都是 基于 线程 或 消息驱动。 这2个是比较极端的。
这里提出一个 比较 大众/折中 的 架构。
基于3个组件：tasks，queues，thread pools

大型互联网服务 必须能够 处理 意料之外的 高并发情况。
高并发 会造成 高的 IO 和 网络 请求。

除了高并发，还要其他的的指标(比如响应时间)， 用来 评估架构
带宽，某段时间访问量大，需要更多的服务器 和 带宽。
延迟，用户的耐心
可用性，每年几分钟不可能用。

需要知道 指标 的瓶颈点 在哪里，比例通过互联网访问，如果网络很差，那么 服务器处理再快，用户还是 要等很久。 或者 下载，如果带宽很低，那么需要很久。

多线程
    操作系统支持好
    CPU多核利用率高
    实现(代码)简单 。。。不过MQ现在也不难
上下文切换的开销
服务器伸缩比较难

事件驱动
    erlang，whatsapp使用了这种语言。
编程语言 对底层的支持有限，线程的话，Thread有 操作系统帮助控制。 事件驱动 得自己写。
debug 难

web proxy cache
服务器代理的缓存，如果有，则直接返回，如果没有，就去 后台服务器 找数据。

每秒 A个task，每个任务需要 L秒处理

多线程模型：
前面一个thread，来处理 dispatch task 给 线程池的某个线程。。。。 这个就是那个执行 threadpool.execute() 这个代码的 那个线程。

线程的创建 销毁 需要资源，所以都是 thread pool

线程受限于 共享资源， 需要 锁，互斥量，竞争 会导致 吞吐量 下降

共享数据的 读 写 。 写只能一个线程独占，只能一个核工作，其他的核 不能工作。吞吐量下降。

。。 记不清了。多线程的数量的公式： 一个任务的总时间 / CPU处理时间 。 前者是从 收到任务 到 返回结果 这一段 持续的 现实时间。 后者是 上下文切入+处理+上下文切出 的时间。
。。应该考虑 锁的独占问题。

线程 太 多 了以后，会导致 性能下降，因为 锁 和 上下文切换。

Event
。。里面是单线程的。
一个任务会被分成几段，出队入队，消耗IO，可能阻塞； 分段后 难以debug
框架不成熟，不像操作系统的线程。
无法重复利用 SMP 架构，无法充分利用多核
处理 阻塞代码 很麻烦。要拆段，每段一个thread。 Multi-Thread可以处理阻塞代码(OS自动切换线程)

对消息的处理，对任务状态的处理 比较高效， Event的吞吐量更高， 因为没有 线程切换。

Queue满。。。 但是 线程池也有这个

可以对 task 再切分，更细致处理。
生产者，消费者 的 处理速率。 设置Queue的大小

设计高并发系统时 还需要考虑很多其他的 参数：错误隔离，数据共享，编程难度

---

https://www2.eecs.berkeley.edu/Pubs/TechRpts/2000/CSD-00-1108.pdf
A Design Framework for Highly Concurrent Systems

。。。。2000年的。。

高并发系统的架构 最终分为 2大类： threaded，event driven。

我们现在有一个 通用的 设计 来构建 高并发系统， 它基于 3个组件：tasks，queues，

thread pools， 包含了 thread 和 event driven 的 并发性，性能，故障隔离，软件工程的 优势。

除了高并发，互联网服务 在设计时 还需要考虑：突发流量，持续服务(不能宕机，不能停机维护)，人类可接受的延迟

因为 延迟是有由 网络 决定的，一个重要的工程权衡 是 优化 高吞吐量 而不是 低延迟。

现有的编程模型 对 高吞吐量 的 code 支持不是很好。
虽然线程 是 高并发的 通常策略， 但 资源消耗大，伸缩性不够 这些限制 使得 开发人员更倾向于 event driven。
。。。我觉得 线程的资源消耗不大吧。 伸缩性确实不太行，要靠外部的负载均衡才能多服务器。。 但是 资源消耗。 我觉得比 事件驱动(MQ) 要小很多。资源指 服务器数量 和 CPU(电量)

不过，event driven 系统 通常是 针对特定应用 从头开始构建的，并且 依赖于 大多数语言和 操作系统 都不支持的 机制。
此外，event driven 的 开发 和 调试 比线程 更复杂。

。。感觉难在 设计，要 抽取出 事件。
。。本来想的是 MQ 不就是 读一个消息，然后处理， 有什么难的， 后来想到了 它也得考虑 锁。。 并且 (同样思路) 线程有什么难的，不就是 从线程池取一个线程(或new 一个线程)，然后 处理。
。。感觉 多线程 和 事件驱动， 两者 本身 没有 可比性。
。。。。事件驱动 感觉 是： MQ处理分发，单台服务器 多线程。
。。。。多线程 感觉是： (负载均衡分发)，单台服务器 多线程。
。。事件驱动 和 多线程 并不是 一个 层次的。

。。感觉 文章的 意义可能不太大。 毕竟 2000年， 那个时代 应该还是 单核 双核 的时代吧。
。。。。现在 8核 16核，(不知道服务器级别的多少核)， 多线程 是 很普遍的。

event driven 技术 对于 高并发 是 useful， 但是 当构建 真实系统时， thread 是有价值的， 对于 多CPU 并行，处理blocking IO

从许多客户端 以 每秒 A 个task 的速度 接收， 每个task 在返回 response 前 服务器端会 delay L秒，但是 可以 多个task 重叠。
我们定义 服务器的 task completion rate 是 S
。。根据下面，应该是 考虑了 cache。感觉 cache命中的话 就不算， 所以 S 应该是 100-cache命中率 ？

这种服务器的 一个例子 是 web proxy cache， 如果 请求 不在cache 中，从 后台服务器获得 page 会有一个 巨大的 延迟， 但是这段时间内，task 不消耗 CPU周期。

客户端收到 服务器的response 后， 立刻 再发出一个 task， 所以 这是一个 闭循环系统。

thread 允许 程序员 编写 straight-line code (。。直线代码。。感觉是 直白/直接 的代码)，依赖操作系统 透明地切换线程 来 overlap computation(重叠计算) 和 IO。

event，允许 程序员 显式 管理 并发 by 组织代码来对event做出react（比如 非IO阻塞的计算，APP特定的消息，timer event）

thread 对于 表示并发的form 占据了 统治地位。
大部分OS 都支持 线程。并且 现代语言(如Java) 对线程有 源生支持
程序员 对 线程的 按顺序的编程风格 感到舒服。
工具相对成熟。
thread 允许 APP 在 多核 上 进行 伸缩。(。。核多，线程多一点)

thread 编程 带来了 许多 正确性 和 调优 挑战。

sync时使用的 lock，mutex，condition variable，是 bug 的 普遍来源。

随着 争用锁 的线程的 增加，锁竞争 可能导致 严重的 性能下降。

event driven 使用 单线程， 非阻塞接口访问IO子系统，或 使用timer工具来在 并发任务间切换。

event driven 系统 通常 组织成 一个线程，死循环，处理 从 queue中 收到的 不同类型的 event。

event driven 编程 有 它自己的挑战， 每个 task的 处理流程 不再是由 一个 thread 来完成，而是 一个thread 完成 所有的 task。
这 带来了 debug 难度，因为 堆栈跟踪 不再表示 特定任务的 控制流程。
同时， task 状态 必须 绑定到 task 它自身， 而不是 存储在 本地变量中。
event 没有标准化，很少有 debug 工具。
但是 event 编程 避免了 很多 sync 的bug，比如 竞争条件 和 死锁。

event 通常 无法 获得 SMP(symmetrical multi-processing 对称多处理) 系统的 优势，除非使用 多个 event-processing thread

。。。
对称式多处理就是每个处理器的行为都是对等的，所以各处理器都自行调度，当处理器空闲时就到共享的就绪队列中挑选一个进程来执行。
非对称式多处理指定由某一个处理器来帮其他处理器进行调度，这种结构称为客户/服务器。这个指定的处理器称为主服务器，不仅处理所有调度，也处理 I/O 和其他系统响应的响应，而其他的处理器就只简单负责执行程序。
。。。

此外，无论使用何种 IO机制，event处理线程 都可以阻塞。页面错误 和 垃圾收集 是 线程暂停的 常见来源，无法避免。
此外，不可能所有 APP 代码都是 非阻塞的，通常，标准库组件 和 第三方代码 导出阻止接口。在这种情况下，线程 很有价值，因为 它们提供了 通过这些 阻塞接口 获得 并发性的机制。

由于队列 暴露了 task 的不同阶段，所以 程序员可以 利用 特定于APP的 知识 重排序 事件处理，以确定优先级 或 提高效率。

3 Design Framework
框架组件
4个通用构建块：tasks, thread pools, queues。
。。。我不知道为什么 是3个，但是 文章里是 four。 看后面，应该是 3个。

task 是我们的框架的基本单元。 它是 有类型的消息，包含了 要完成的一些工作的 描述，以及完成 该task 所需的数据。

task
由 APP 的各个组件 在一系列 阶段中 处理。
例如，web server 收到了一个 静态HTML页面的 请求，task 的 URL必须先被 解析，然后 在本地cache 中查询 这个 page，最后 (如果有需要的话) 从 磁盘读取。
task 的 各个阶段 可以顺序执行，也可以并行，或者2者的 组合
。。。顺序执行 + 并行 怎么混合，一混合 就是 并行了吧。

通过 将任务 分解为 一系列阶段，可以将 这些 阶段 分布在 多个 物理资源上，并对 这些任务 的流程 进行分析，以实现 负载均衡 和 故障隔离。

thread pool，
在一台机器上的 线程的集合，处理task 。
从逻辑上讲，线程池 和 一组任务类型 相关联，池中的每个线程 执行一段代码，该代码 消费 一个task，处理它，并将 一个或多个 outgoing 任务 分派给 线程池。
线程池 是 我们框架中 用于执行 的 唯一资源

模块化 是通过 将APP 分解为 一系列 线程池，每个线程池 处理 特定的task类型。
在多CPU系统中，每个线程 可以在 单独的 CPU上运行，因此 可以利用 并发性。

queue
是 线程池 间 交互的手段。
queue 逻辑上 由 task 列表组成， thread pool从 它们的 incoming task queue(传入task 队列) 拉取 task，通过 将 task 推送到 线程池的 incoming queue 来 分发它们。
2个线程池的 操作 可以组合在一起， 通过 在 它们之间 插入一个 queue，来允许 task 从一个线程池 传递到 另一个。

我们将 线程池 及 它的incoming task queue 称为 task handler

通过引入 显式控制边界，queue 充当 线程池 间的 隔离机制。这限制了 线程的执行 到 一个 给定的 task handler。 这是可取的，因为2个原因：

1. 使得 APP 更容易 debug，由于 线程池的 内部状态 对 其它线程池 不可见。
2. 它可以消除 线程"逃逸"到一段代码中 而永远不会返回的 情况 -- 例如，到 执行 阻塞 IO 的库

另外，queue 为 overflow absorption(。溢流吸收), backpressure(。背压), fairness(。

公平性）提供了一种机制。
当 输入的task 超过 线程处理速度时， queue 就像一个 buffer。
backpressure 可以通过 让queue 在满了的时候 拒绝新entry(比如，抛出异常）来实现。这很重要，因为它 允许 系统拒绝 多余的 负载，而不是 缓冲 任意数量的 task
可以通过 根据线程池的 传入队列 长度 来 调度 线程池 来 实现公平性
。。。我以为是FIFO。这里更像是 负载均衡。 不过搞不懂，怎么 实现 这种调度。是指 线程池的 扩大 和缩小？

3.2 Design Patterns
现在的问题是 如何 映射 APP 为 使用上面的组件的组合。
何处使用 线程池，queue 是对 性能，错误隔离 有很大的影响的。

下面有 4 种 设计模式 来使用我们的框架 构建 APP。
这些模式 封装了 框架组件的 基本属性，并描述了 如何使用它们来构建 APP。

Wrap
wrap 模式 用 queue接口 包装 一组线程。
每个线程 处理 一个任务 的 多个阶段， 可能阻塞 一次或多次。
使用 wrap，在 线程集合 前面 放置一个 single input queue，实际上就是 在它们外面创建了一个 task handler。
这个动作 使得 task handler 中的 处理过程 在 load 时 更加健壮，因为 现在 task handler 中的 线程数量 可以 是 固定值，来防止 线程过多导致 性能下降， 无法处理的 task 会在 queue中累积。
。。就是 原本 来一个 request，建一个 thread，
。。现在是 请求放到 queue中，然后 线程池运行。 实际上 Java的 线程池 自带queue，所以 一个 ThreadPoolExecutor 就等于这里的 queue+线程池

Pipeline(管道)
pipeline 拿到 一个 单线程的代码片段，通过 在不同的点 引入 一个 queue 和 线程池边界来 将 代码 拆分成 多个 pipeline stages。
例如，如果 对每个 阻塞IO call 都 引入一个 queue，这会使得 每个 call 变成 非阻塞，因为 单独的线程 会处理 call。

有2个用途。
1. 限制 处理低并发操作 时 申请的 线程的 数量。
   假如 一个 pipeline stage 的 task arrival rate 是 每秒A个task，每个task 需要L秒，保持该阶段的 A 的完成率 需要 维持 A*L 个线程。
   现在 考虑 一个 pipeline stage，它限制了 并发处理的 任务个数。例如，Unix 文件系统 一般 处理 固定数量(40-50之间)的 并发 读写。 我们将这个限制 称为 管道阶段的 宽度，并用 W 表示。
   管道阶段 的 width 代表了 这个阶段 的 并发 线程的 上限 （ 超过就性能下降)。

即，如果 W < (A*L)，没有必要 为 这个阶段提供 大于 W 的线程数量， 多余的线程只会空闲。 完成率 S 是 S = W/L，如果 W<(A*L)，那么 S < A。
带上限的 width 限制了 task handler 的 完成率， width 只能通过 replication(后续会讨论) 来增加。

pipeline 提供了一个 方法 来 限制 stage的线程 数量 为 这个stage的 width。
把一个task 的处理 分解为 多个独立的 stage，并为 每个 stage 设置 合适的 线程池 size，并允许 这个 stage 被复制到 独立的物理资源(后面会说) 来实现更高的并发

2. 增加locality(地点)。
Cache locality 是 构建高吞吐量系统的 越来越重要的 因素，由于 cache 和 主内存 之间的 性能差距 越来越大。
。。。这不会是在说 CPU cache吧。。。

另外，多核CPU 使得 可用内存带宽 压力增大，如果 cache miss，会消耗 长时间。
。。。 真的是 CPU cache。。这个。。。

在 每个task 一个thread 的系统中， cache指令会发生 很多 miss，如果 线程 使用了 许多 不相关的 代码模块 来 处理 task。
另外，当一个 context switch 发生(由于 thread preemption(优先权) 或 阻塞IO)， 其他线程 一定会 从 cache 中清楚 等待线程的 状态。 当原线程 恢复执行，它为了 将 代码 和 状态 保存到 cache 中，会收到 很多 cache miss。这种情况下，这个系统里的 线程 受限于 有限的cache 空间。

应用 pipeline 可以增加 data 和 instruction cache locality 来 避免 这个 性能问题。
每个 pipeline stage 都可以执行 一个 "convey(护送)" 为所有task，用自己的代码 保证 cache warm。
。。感觉是 for循环啊，一个线程 跑完所有任务。。好像也不行，切不切换 要看OS，或许可以把 优先级调高。
。。不，上面说到 上下文切换，是由于 阻塞IO，或 优先级， 那么 就应该是 把阻塞IO 的 代码 抽取出来，这样的话， 代码越短，我相信 上下文 越小。切换的代价也越小。 而且 可以 什么 多路复用什么的(不清楚是不是这个方面的。)。
。。就和 事务一样， 大事务 的速度 比 多个小事务 慢很多。

另外，每个 pipeline stage 有机会 以 data cache locality 利用率大小 的 顺序 来 处理 incoming task。例如，如果 queue 是 FILO (。。就是 stack)， 那么 最近达到的 task 它可能仍然在 data cache 中。
。。？共享内存啊， 但是 通过 queue后， 这是 2个 相同内容的 对象吧。不，这是 queue，不是 MQ。 但是 queue， 大型系统，现在靠 MQ了， 单机的queue。很少直接用了吧。 而且 线程池自带queue。。
。。而且 第一点的 复制到额外的 物理资源上，说明 就是 MQ。
。。突然想到一个操作： 直接提交 本机的 线程池中， 线程池的 等待队列满了以后的 abort policy 就是 放到 外部的MQ中。 好像很6 啊。

尽管 pipeline 增加了 任务处理的时间， 但是 我们框架的 目标是 优化 总吞吐量，不是每个单独任务的 处理时间。

Combine
合并2个 独立的task handler 为 一个 带有共享线程池的 task handler。
如果 系统中的 线程数量过多，性能会下降。
combine 用来 允许 多个 task handler 共享 一个 thread pool 来避免线程浪费。

考虑：使用 pipeline 获得 3个连续的 pipeline stage 来 隔离 阻塞操作 到它自己的线程池中。 如果 第一，第三个 stage 都是 CPU 限制的，它们的 width W 是 CPU 的数量。 比起 2个 W大小的 线程池，通过 combine 得到 一个 在stage间共享的 W大小的线程池 更好。
这种情况下， Combine 是 pipeline 的reverse。

Replicate
create 一个现有的 task handler 的 "copy"。
另外，它要么在 一组新物理资源上 实例化 新的task handler， 要么 在 2个 copy 之间 设置 failure boundary (失败隔离)，要么 both。
replicate 用来 实现 并行 和 错误隔离

通过在 跨物理资源 复制 task handler，总宽度 W 增加了。
这可以用于 消除 任务处理阶段 流水线 的 瓶颈。
在 不同的地址 或 不同的机器上 运行 2个副本，可以在 2个副本将 引入 故障边界。这个使得 副本 具有 高可用，如果一个失败了，另一个还可以继续处理task。

复制引发了 分布式状态管理的 担忧。
集群中的 网络链接的失败 会导致 分区，如果 各个集群节点上的 task handler 需要 维持一致的状态，这是很麻烦的。

有几个方法来避免这个问题：
1. 使用 分布式一致性 或 group membership 协议。
2. 设计 集群互连 以消除分区，这是 DDS 和 Inktomi 搜索引擎 使用的方法。

3.3 Applying the Design Patterns
需要考虑约束
1. pyhsical resource limits：假设 APP可用的 集群节点数量 是固定值。同样，CPU速度，内存大小，磁盘和网络 带宽，等其他参数 都被认为 是固定的。
2. thread limits：线程的实现 通常有 线程数量的 限制，超过这个限制，会使得 性能下降，我们称 这个限制 是 T'。 这个值 取决于 线程实现。
3. latency and width：给定task processing stage 的 L 和 W 决定了 应用上述设计模式的 大部分决策。

Apply Wrap to introduce load conditioning
将APP 线程放到 一个 task handler 中 来使得 线程的数量 限制到 某些 值(少于T')，额外的task 被queue abort。

Apply Pipeline to avoid wasting threads
在上面的转换后，可能增加吞吐量 by 将具有低width W 的代码 隔离到它自己的 task

handler，且 限制task handler 中的 线程数量 限制 为 W。
例如，可以在 进行文件系统 调用时 应用 pipeline，并将文件系统任务 处理程序 中的 线程数 限制为 文件系统 可以处理的 并发访问数。这有效地 "释放" 了 额外的线程，这些 线程 可以放到 其他 的处理 程序中。

Apply Pipeline for cache performance
pipeline 可以用来 增加 cache locality by 隔离 有关联的task processing stage 的代码 到 它们自己的 task handler 中。
以这种方式构架的 APP 的 cache locality 的优势 取决于 2个主要因素
1. 在进行 task 的 convoy 时，task handler 内部使用的 code 和 shared code 的总量。这决定了 task handler 中可以 实现的 潜在 cache locality优势
2. 当task 被push到queue中时，在 task handler 之间可以 传递的 数据的 总量。这决定了 在无序 处理任务时的 潜在数据 cache locality 优势。

pipelining task processing 的 性能影响 能 直接度量，可以用来 决定 一个 APP是否使用这种 结构

Apply Replicate for fault tolerance
在多个资源中使用 task handler 的 replication，通过增加冗余的 task processing stage， 增加了 APP整体的 可靠性。 一台机器故障的几率是 F，那么n台机器的 集群都故障的概率是 F^n

replication 的收益 依赖于几个因素：
1. 是否有足够的 物理资源 来支持 replication。
2. 要 replicate 的 task handler 是否依赖于 共享的 状态，这会引发几个 设计问题。

Apply Replicate to scale concurrency
replicate 可以有效地 增加 并发 width W， 通过 在 不同的资源上 运行 多个实例。
通常来说， 将一个 task handler 复制 n 次，会将 整体的 width 增加到 W * n。
对于一个 有着 L latency 的 task hander，这增加了 整体的 task completion rate，从 S=W/L 增加到 S=(W*n)/L， 消除了 吞吐量瓶颈。

Apply Combine to limit the number of threads per node
在对 每个task一个thread 的 应用 进行 pipeline 和 replicate 后，我们可能 获得 大量的 task handler，每个task handler 有它们自己的 thread pool。
如果一些 task handler 的 latency 是 低的，那么 它们 可以 "share" thread pool，来降低 总的 线程数量。
之前讨论过，限制 每个 node 的 thread 数量 为 值 T' 是重要的，这个值 对于系统来说是可以测量的。
Combine 可以合并 一个node 上的 不同的 task handler 使用的 线程池，来节约线程。

这些启发式 假设 APP 设计者 也在考虑 底层平台的 固有资源限制。
例如，一个 task handler 不应该 放到 node 上，如果 它会 使得 node 的线程数量超过 T'。

3.4 Principles

除了上面的设计模式，我们的框架 还有一些其他的 用来构建系统的 准则

1. task handler 尽量是 无状态的。这允许 task handler 变成 lock-free，因为 线程间没有 状态 在共享。 而且，这允许 task handler 更容易 被 create 和 restart。
2. task间 应该通过 值传递 来传递数据，而不是 ref。 2个task handler 间的 数据共享会 引发 许多问题。

   共享数据的 一致性 需要 使用 lock 或类似机制 来保证。lock 会导致 竞争，长时间等待，这些会降低 并发。

   通过 ref 传递 数据， 当2个 task handler 在不同的 位置空间 或 不同的 机器上时，这会是 问题。 DSM(distributed shared memory) 能用来 进行 跨地址空间 共享， DSM机制 是复杂的 且 它们自己就有 并发问题。

   数据共享 需要 task handler 商定 谁负责 回收 data。在有GC 的环境中，这个很简单。如果没有GC，需要明确的 配合。

   还有，数据共享 减少了 错误隔离。如果一个 task handler 失败了，使得 共享的数据进入了 不一致状态，共享这个数据的 任何其他的 task handler 必须能够 从这种情况下恢复 或 自己承担故障风险。

   值传递的 一种替代是： ref传递 且 数据发起者放弃访问。

   减少数据共享的 另一种方式是 对 APP 状态 进行 空间分区，在这种情况下，task handler 的 多个 pipeline stage 或 replica 处理 它们自己的 APP 状态 私有分区，而不是 共享状态+lock。

   Another means of reducing data sharing is to space-partition application state

3. 避免 fate sharing (命运共享)。 如果 2个task handler 共享了 物理资源，那么 它们共享了 命运：即，如果物理资源失败 (如 node崩溃)，这2个 task handler 都会失败。 所以 一个task handler 的 副本 应该 允许在 不同的 物理node 上， 或 至少在不同的 address space， 来避免 fate sharing。

   注意，task handler 可以通过 负载 进行 link，不仅仅是 failure。例如，如果一个 replica 失败了， 其他的replica 会 承担 它的负载，为了 维持吞吐量。

4. 权限控制 应该在 APP 的 前面执行完。 在最前面就 拒绝 或 延迟 任务，可以让 task handler 获得 尽可能多的 资源。

   这个要求 所有的 task handler 都 实现 backpressure，通过 当queue长度太长时，拒绝 新的 task。 不使用 额外的 权限控制机制，一个 slow task handler 会 过载 并且 许多task 在排队。

   这增加了 latency(延迟) 和 系统的资源需求。。。。估计就是说前面加了一层，增加了处理时间 和 系统资源，但是 可以防止 最坏情况的发生。

4 Analysis of the Framework

4.1 Distributed Data Structures

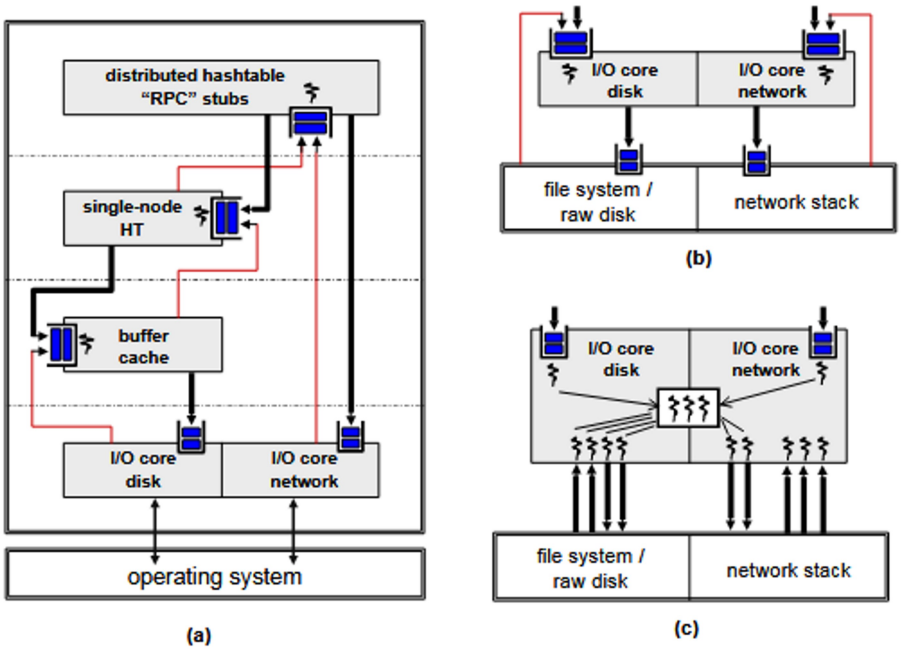


Figure 10: *Distributed hash tables:* (a) illustrates the structure of the distributed hash table "brick" process; the thick arrows represent request tasks, and the thin arrows represent completion tasks. (b) shows the ideal operating system and I/O core implementation, in which the OS exports a non-blocking interface. (c) shows our current implementation, in which blocking the I/O interfaces exposed by the Java class libraries force us to dispatch threads to handle I/O requests from a fixed-sized thread pool. There are also dedicated threads that block listening on incoming sockets.

4.2 vSpace

4.3 Flash and Harvest
Flash web server
Harvest web cache

5 Related Work

6 Future Work and Conclusions

==========

==========

Scatter/Gather I/O，翻译过来是分散/聚集 I/O（又称为Vectored I/O）。是一种可以单次调用中对多个缓冲区进行输入/输出的方式，可以把多个缓冲区数据一次写到数据流中，也可以把一个数据流读取到多个缓冲区

---

https://blog.csdn.net/weixin_33910759/article/details/92527574

分散/聚集 I/O 是一种 可以在 单次系统调用中 对多个缓冲区输入输出的方法， 可以把多个缓冲区的 数据写到 单个数据流，也可以把 单个数据流读到 多个缓冲区中。
命名的原因 在于 数据会被 分散到指定缓冲区向量，或者从 指定缓冲区向量中 聚集数据。
这种输入输出方法 也称为 向量IO (vector I/O)。
与之不同，标准读写系统调用 可以称为 线性I/O (linear I/O)

与线性IO相比， 分散/聚集 IO 有如下优势
编码模式更自然
    如果数据本身是分段的 (比如 预定义的结构体的变量)，向量IO 提供了 直观的数据处理方式
效率更高
    单个向量IO 操作可以取代 多个线性IO 操作。
性能更好
    除了减少了 发起的系统调用次数，通过内部优化，向量IO 可以比线性 IO 提供更好的性能
支持原子性
    和多个线性IO 操作不同，一个进程可以执行 单个向量IO操作，避免了 和其他进程交叉操作的风险。

readv() 和 writev()
Linux 实现了 POSIX 1003.1-2001 中定义的一组 实现分散/聚集 IO 机制的系统调用。该实现满足了上面所述的 所有特性。

readv() 函数 从文件描述符 fd 中读取 count 个段(segment) (一个段 就是一个 iovec 结构体) 到参数 iov 所指定的缓冲区中。

```
#include <sys/uio.h>
ssize_t readv (int fd, const struct iovec *iov, int count)
```

writev() 函数 从参数 iov 指定的缓冲区中 读取 count个 段的值，并写入到 fd中：

```
#include <sys/uio.h>
ssize_t writev(int fd, const struct iovec *iov, int count)
```

除了同时操作多个缓冲区外，readv() 函数 和 writev() 函数的 功能和 read()，write() 的功能一致。

iovec 结构体描述了一个 独立的，物理不连续的 缓冲区，我们称为 段(segment)

```
#include <sys/uio.h>
struct iovec {
       void      *iov_base;/* pointer to start of buffer */
       size_t    iov_len;/* size of buffer in bytes */
};
```

一组段的集合称为向量(vector)。 每个段描述了内存中所要读写 的缓冲区的 地址和长度。
readv() 函数在处理 下个缓冲区之前，会填满 当前缓冲区的 iov_len 个字节。
write() 函数在处理下个缓冲区之前，会把当前缓冲区所有 iov_len 个字节数据输出，这2个函数都会顺序处理向量中的段，从iov[0] 开始，接着是 iov[1]，一直到 iov[count - 1]

返回值
操作成功时，readv() 和 writev() 分别返回 读写的字节数。 该返回值 应该等于 所有 count 个 iov_len 的和。
出错时，返回-1， 并相应设置 errno 值。
readv,writev 可能的错误包含了 read，write 可能的错误，并且 多了2个场景：

1. 由于返回值是 ssize_t， 如果所有 count 个 iov_len 的和 超出 SSIZE_MAX，则不会处理任何数据，返回-1，并把 errno 值 设置为 EINVAL。
2. POSIX 指出 count 必须大于0，且小于等于 IOV_MAX (在limits.h中定义)。在 Linux 中，IOV_MAX 的值是 1024. 如果count为0，该系统调用会返回0。如果 count 大于 IOV_MAX，不会处理任何数据，返回-1， 并把 error 设置为 EINVAL

优化count值
在向量IO中，Linux 内核必须分配 内部数据结构 来表示 每个段(segment)。一般来说，是基于count 的大小动态分配进行的。然而，为了优化，如果count 值足够小，内核会在 栈上 创建一个 很小的 段数组，避免通过动态分配段内存，从而获得性能上的一些提升。 count 的阈值一般是 8。 因此如果 count <= 8，向量IO操作 会以 一种高效的方式，在进程的 内核栈 中运行。

大多数情况下，无法选择在 指定的 向量IO操作 中一次同时 传递多少个段。当你认为可以调试一个较小值时，选择 8 或更小的值 肯定 会 得到性能的提示。

Linux内核把 readv writev 作为系统调用实现，在内部使用 分散聚合IO 模式。 实际上，linux内核中所有的 IO 都是向量IO，read write 是通过 向量IO 实现的，只不过 向量中只有一个 段。

==========

==========

IO多路复用

https://blog.csdn.net/m0_51319483/article/details/124264619

IO 多路复用（IO Multiplexing ）一种同步IO模型，单个进程/线程 就可以同时处理多个IO请求。 一个进程/线程 可以监视 多个文件句柄，一旦某个句柄就绪，就能够通知 应用程序进行相应的 读写操作， 没有文件句柄就绪时 会阻塞应用程序，交出CPU。

多路是指 网络连接，复用是指 同一个进程/线程。
一个进程/线程 虽然任一时刻只能处理 一个请求，但是处理 每个请求的 事件时，耗时控制再 1ms 以内，这样 1秒就可以 处理 上千个请求， 把时间拉长来看，多个请求复用了一个进程/线程，这就是 多路复用。 这种思想类似 一个CUP 并发多个进程，所以也叫做 时分多路复用。

在没有 IO多路复用时， 有 BIO，NIO 2种实现，但是会出现 阻塞 或 开销大的 问题

同步阻塞(BIO)

1. 服务器采用 单线程，当accept 一个请求后，在 recv 和 send 调用阻塞时，将无法 accept 其他请求 (必须等上一个请求 处理完 recv 或 send)， 不能处理并发
2. 服务器采用 多线程，当accept 一个请求后，开启线程进行 recv，可以完成并发处理，但随着请求数增加，需要增加系统线程，大量线程 占用 很大的内存空间，并且 线程切换 带来 开销。

同步非阻塞(NIO)
服务器accept 一个请求后，加入 fds 集合，每次 轮询以便 fds集合recv（非阻塞）数据，没有数据则 立即返回错误， 每次轮询所有 fd（包括没有发生 读写事件的 fd）会很浪费CPU。

IO多路复用
服务器端 采用 单线程 通过 select/epoll 等系统调用 获取 fd 列表，遍历 有事件的 fd 进行 accept/recv/send， 使其能支持更多的并发连接请求

IO多路复用的3种实现方式
select 函数

缺点：
每个进程所打开的 fd 是有限制的，通过 FD_SETSIZE 设置，默认 1024
每次调用 select，都需要把 fd集合 从用户态 拷贝到 内核态，fd多的话，开销很大。
对 socket扫描时 是 线性扫描，采用轮询的方法，效率较低（高并发时）

poll函数

缺点
每次调用 poll时，都需要把 fd集合 从用户态 拷贝到 内核态，这个开销在 fd很多时会很大。
对socket 扫描时 是线性扫描，采用轮询的方式，效率较低(高并发时)

epoll函数

缺点
只能在 Linux 下工作。

epoll LT和ET模式的区别
epoll 有 EPOLLLT 和 EPOLLET 2种触发模式， LT是默认模式， ET是高速模式
LT模式下，只要这个 fd 还有数据可读，每次 epoll_wait 都会返回它的事件，提醒用户程序去操作
ET模式下，它只会提示一次，直到 下次再有数据 流入之前 都不会 再提示，无论 fd 中是否还有数据可读。 所以在ET 模式下，read 一个 fd 的时候 一定要把 buffer 读完，或者 遇到 EGAIN 错误

select/poll/epoll 区别

	selec	poll	epoll
数据结构	bitmap	数组	红黑树
最大连接数	1024	无上限	无上限
fd拷贝	每次调用selec拷贝	每次调用poll拷贝	fd首次调用epoll_ctl拷贝，每次调用epoll_wait不拷贝

工作效率	轮询O：（n）	轮询：O(n)	回调：O(1)

---

https://www.jianshu.com/p/7835a916353b

用户进程缓冲区 和 内核缓冲区 介绍
系统调用需要保存之前的 进程数据 和 状态等信息，而结束调用之后 回来 还需要恢复之前的 信息，为了减少这种 损耗时间 和 损耗性能的 系统调用， 于是出现了 缓冲区。
缓冲区的目的是 为了减少 频繁的 系统IO调用。 有了缓冲区，操作系统 使用 read函数把数据从内核缓冲区 复制到 进程缓冲区， write 把数据从 进程缓冲区 复制到 内核缓冲区中。
等待缓冲区达到一定数量的时候，再进行IO的调用，提示性能。
至于什么时候读取和存储 由 内核来决定，用户程序不需要关心。
用户程序的 IO 读写程序，大多数情况下，并没有进行实际的 IO 操作，而是读写自己的进程缓冲区

五种IO模型
同步阻塞IO (Blocking IO)
阻塞IO，指的是需要 内核IO 操作 彻底完后后，才返回到 用户控件，执行用户的操作，在此期间请求的 用户 进程是挂起状态。
阻塞 指的是 用户空间程序的 执行状态，用户空间程序 需要等到 IO操作彻底完成。
传统的IO 模型都是 同步阻塞IO。 默认创建的 socket 都是阻塞的。
(使用 recvfrom 函数一直等待 数据 直到拷贝到 用户空间，这段时间内 进程 始终阻塞)

同步非阻塞IO (Non-blocking IO)
非阻塞IO， 指的是 用户程序 不需要等待内核IO 操作完成后，内核立即返回给 用户一个状态值，用户空间 无需等待 内核的 IO 操作彻底完成， 可以立即返回 用户空间，执行用户的操作，处于 非阻塞的状态。

IO多路复用(IO Multiplexing)
在调用 recv 前 先调用 select 或者 poll， 这2个系统 调用 都可以 在内核准备好数据(网络数据到达内核) 时 告知用户进程， 这个时候 再调用 recv 一定是有数据的。因此这个过程是 阻塞于 select 或 poll， 而没有阻塞于 recv。在数据到达的时候 依然需要 等待复制数据到 用户空间，因此它还是 同步IO。

信号驱动IO模型
通过调用 sigaction 注册信号函数， 等内核数据准备好的时候 系统 中断 当前程序，执行信号函数 返回数据 准备好的 信号，之后调用 recvfrom 同步复制数据。

异步IO
调用 aio_read， 让内核等数据准备好 并且 复制到 用户进程空间后 执行 实现指定好的 函

数。

select, poll和 epoll 的区别和联系

1. poll本质上 和 select 没有区别，但是 它没有 最大连接数的 限制，因为它是 基于链表来存储的。
2. select,poll 需要 不断地轮询所有 fd集合，直到设备就绪，期间可能要 睡眠和唤醒多次 交替。而 epoll 其实也需要 调用 epoll_wait 不断轮询 就绪链表，期间也可能多次睡眠 和 唤醒 交替，但是 它是 设备就绪时，调用 回调函数，把 就绪 fd 放入 就绪链表中，并唤醒 在 epoll_wait 中进入睡眠的进程。
   虽然都要睡眠和交替，但是 select 和 poll 在"醒着" 的时候 要遍历 整个 fd 集合，而 epoll 在 "醒着" 的时候 只要判断下 就绪链表 是否为空 就行，节省了大量的 CPU 时间。这就是 回调机制 带来的 性能提升。
3. select, poll 每次调用时 都要把 fd 集合 从用户态 往内核态 拷贝一次，并且 要把 current 往设备等待 队列中 挂一次，而 epoll 只要一次拷贝，而且把 current 往等待队列上 挂也只挂一次。 这也能节省不少的开销。

==========

https://zhuanlan.zhihu.com/p/446607767

IO多路复用技术总结

。。很多代码。

==========

==========

https://www.cnblogs.com/maxigang/p/9041080.html
用户态和内核态

内核态：CPU可以访问 内存的所有数据，包括 外围设备 ，如 硬盘，网卡，CPU 也可以将 自己从一个程序 切换到 另一个程序。
用户态：只能受限的访问内存，且不允许访问 外围设备，占用 CPU 的能力 被剥夺，CPU 资源可以被 其他程序 获取。

由于需要限制 不同程序 之间的 访问能力，防止它们 获得 别的程序的 内存数据，或者 获取 外围设备的数据，并发送到网络， CPU 划分出 2个 权限等级 - 用户态和内核态。

用户态和内核态的切换
所有用户程序 都是 允许在 用户态的， 但是有时候 程序 确实 需要做一些 内核态的 事情，例如 从 硬盘读取数据，或者从 硬盘获取输入等，而唯一可以做这些事情的 就是 操作系统，所以 此时 程序 就需要 先 操作 系统请求 以 程序的名义 来执行这些操作。

这时需要一个这样的机制： 用户态 程序 切换到 内核态，但是不能控制 内核态 中执行的指令。
这种机制 叫做 系统调用，在 CPU 中 的实现 称之为 陷阱指令

工作流程如下：
1. 用户态程序 将 一些数据值 放到 寄存器中，或者 使用 参数 创建一个 堆栈，以此表明需要 操作系统提供的服务。
2. 用户态程序 执行 陷阱指令
3. CPU切换到 内核态，并跳到 位于 内存指定位置的 指令，这些指令 是操作系统的 一部分，它们具有 内存保护，不可被 用户态程序访问。
4. 这些指令 称之为 陷阱 或 系统调用处理器。 它们会读取 程序放入内存的 数据参数，并执行 程序请求的 服务
5. 系统调用完成后，操作系统会 重置 CPU 为 用户态 并返回 系统调用的 结果。

当一个任务(进程)执行系统调用 而陷入 内核代码中执行时，我们就称 进程 处于 内核运行态（或 简称 内核态）。此时处理器 处于 特权级 最高的（0级）内核代码中执行。
当进程 处于内核态时，执行的内核代码 会使用 当前进程的 内核栈。 每个进程都有自己的内核栈。
当进程 在执行 用户自己的代码时，则称其处于 用户运行态(用户态)。即此时 处理器在 特

权级 最低的(3级) 用户代码中运行。 当正在执行 用户程序 而突然 被 中断程序中断时，此时 用户程序也可以 象征性地 称为 处于进程的 内核态。 因为中断处理程序 将使用 当前进程的 内核栈。这与 处于内核态的进程 的状态有些类似。

内核态 和 用户态是 操作系统的 2种运行级别，跟 intel cpu 没有必然的联系，intel cpu 提供 Ring0 - Ring3 四种级别的运行模式，Ring0 最高。
Linux 使用 Ring3 级别 运行用户态，Ring0 作为 内核态， 没有使用 Ring1 和 Ring2.
Ring3 状态 不能访问 Ring0 的地址空间，包括 代码和数据。
Linux 进程的 4GB 地址空间，3G-4G部分 大家是共享的，是内核态的 地址空间，这里存放整个内核的 代码和所有的 内核模块，以及 内核所维护的数据。
用户运行一个程序，该程序 所创建的 进程开始是 运行在 用户态的，如果要执行文件操作，网络数据发送 等操作，必须通过 write，send 等系统调用，这些系统调用 会 调用 内核中的 代码来完成操作， 这时，必须切换到 Ring0，然后进入 3GB-4GB 中的 内核地址空间 去执行 这些代码完成操作，完成后，切换回 Ring3， 回到用户态。 这样，用户态的 程序 就不能 随意操作 内核地址空间，具有一定的 安全保护作用。

至于说 保护模式，是说 通过 内存页表操作 等机制，保证 进程间的地址空间不会 互相冲突，一个 进程的 操作 不会修改 另外一个 进程的 地址 空间中的 数据。

先看一个例子：

```
void testfork(){
    if(0 = = fork()){
        printf(“create new process success!\n”);
    }
    printf(“testfork ok\n”);
}
```

代码很简单：执行一个 fork()，生成一个 新的进程，从逻辑的 角度看， 就是 判断了 如果 fork() 返回 0 则 打印语句，然后 函数 最后再 打印一句 来表示 执行完整的 testfork() 函数。
无法看出 哪里体现了 用户态 和进程台的 概念。

fork 的工作 实际上是以 系统调用的 方式 完成 相应功能的，具体的工作由 sys_fork 负责实施。
对于任何操作系统，创建一个 新的进程 必然属于 核心功能， 因为它要 做很多 底层 的工作，消耗 系统的 物理资源，比如 分配 物理内存，从父进程 拷贝相关信息，拷贝设置页目录页表等，这些 显然不能虽然 让哪个线程 就能去做，于是 就自然 引出 特权级别的概念，显然，最关键性的 权力 必须由 最高级别 程序来执行，这样 才可以 做到 集中管理，减少有限资源 的访问 和 使用冲突。
特权级 显然是 非常有效的 管理 和 控制 程序执行的 手段，因此 硬件上 对 特权级 做了很多支持， 就 intel x86 架构 的 CPU 来说， 一共有 0-3 四个特权级，0最高， 硬件上在执行 每条指令时 都会对 指令 所具有的 特权级 做相应的检查，相关概念有 CPL，DPL，RPL。
硬件提供了 一套特权级使用的 相关机制，软件自然就会好好利用，这属于 OS 要做的事情，对于 Unix/Linux 来说，只使用了 0 和 3级特权级。

现在从特权级 的调度 来理解 用户态 和 内核态 就比较好理解了， 当 程序 运行在 3级特

权级上时， 就称之为 运行在 用户态， 因为这是 最低特权级， 是普通的 用户进程允许的特权级， 大部分用户 直接面对的 程序 都是 允许在 用户态。
当 程序运行在 0级特权级 上时， 就可以称之为 允许在 内核态。

运行在用户态下的 程序不能 直接访问 操作系统 内核数据机构 和 程序，比如上面的 testfork() 就不能 直接调用 sys_fork()，因为 前者是工作在 用户态，属于 用户态程序，而 sys_fork() 是工作在内核态，属于内核态程序。

当我们在程序中 执行一个程序时，大部分时间 是运行在用户态下的， 在其需要 操作系统帮助 完成 某些 它没有权利 和能力完成的 工作时 就会 切换到 内核态，比如 testfork() 最初运行在 用户态进程下，当它调用 fork() 最终触发 sys_fork() 的执行时， 就切换到 内核态。

用户态切换到 内核态的3种方式

1. 系统调用
   这是用户态进程 主动要求切换到 内核态的一种方式，用户态进程 通过系统调用申请 使用 OS提供的服务程序 完成工作，比如前面中的 fork() 实际上 就是 执行了一个 创建新进程的 系统调用。而系统调用 的机制 的核心还是 使用了 OS为用户特别开发的 一个中断 来实现，例如 Linux 的 int 80h 中断。
2. 异常
   当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。
3. 外围设备的中断
   当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。

这3种方式是系统在运行时由用户态转到内核态的最主要方式，其中系统调用可以认为是用户进程主动发起的，异常和外围设备中断则是被动的。

从触发方式上看，可以认为存在前述3种不同的类型，但是从最终实际完成由用户态到内核态的切换操作上来说，涉及的关键步骤是完全一致的，没有任何区别，都相当于执行了一个中断响应的过程，因为系统调用实际上最终是中断机制实现的，而异常和中断的处理机制基本上也是一致的
涉及到由用户态切换到内核态的步骤主要包括：
[1] 从当前进程的描述符中提取其内核栈的ss0及esp0信息。
[2] 使用ss0和esp0指向的内核栈将当前进程的cs,eip,eflags,ss,esp信息保存起来，这个过程也完成了由用户栈到内核栈的切换过程，同时保存了被暂停执行的程序的下一条指令。
[3] 将先前由中断向量检索得到的中断处理程序的cs,eip信息装入相应的寄存器，开始执行中断处理程序，这时就转到了内核态的程序执行了。

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========