后台服务架构高性能设计之道

极客重生 2022-09-05 21:27 Posted on 广东

The following article is from 腾讯技术工程 Author 腾讯程序员

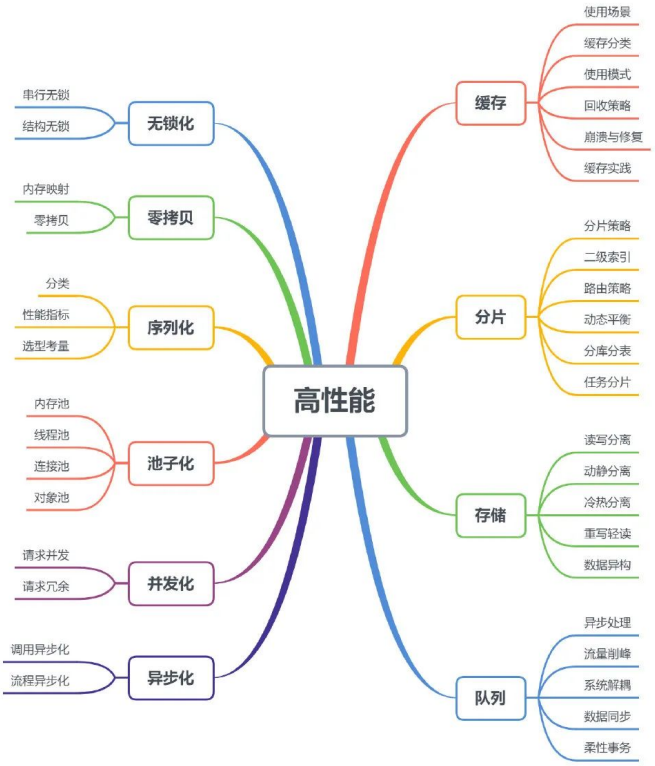
腾讯技术工程

腾讯技术官方号。腾讯技术创新、前沿领域发布解读平台。

> “N 高 N 可”，高性能、高并发、高可用、高可靠、可扩展、可维护、可用性等是后台开发耳熟能详的词了，它们中有些词在大部分情况下表达相近意思。本序列文章旨在探讨和总结后台架构设计中常用的技术和方法，并归纳成一套方法论。

前言

本文主要探讨和总结服务架构设计中高性能的技术和方法，如下图的思维导图所示，左边部分主要偏向于编程应用，右边部分偏向于组件应用，文章将按图中的内容展开。

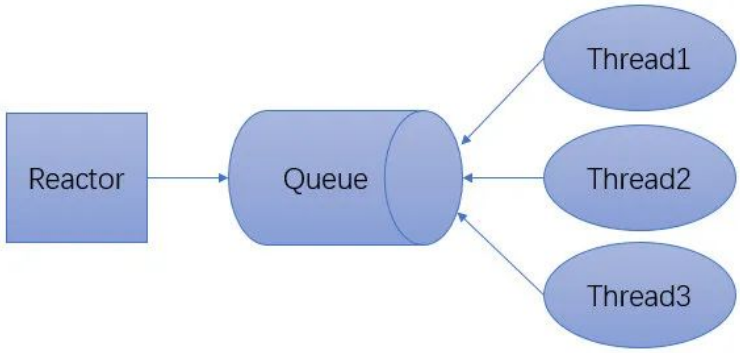


1 无锁化

大多数情况下，多线程处理可以提高并发性能，但如果对共享资源的处理不当，严重的锁竞争也会导致性能的下降。面对这种情况，有些场景采用了无锁化设计，特别是在底层框架上。无锁化主要有两种实现，串行无锁和数据结构无锁。

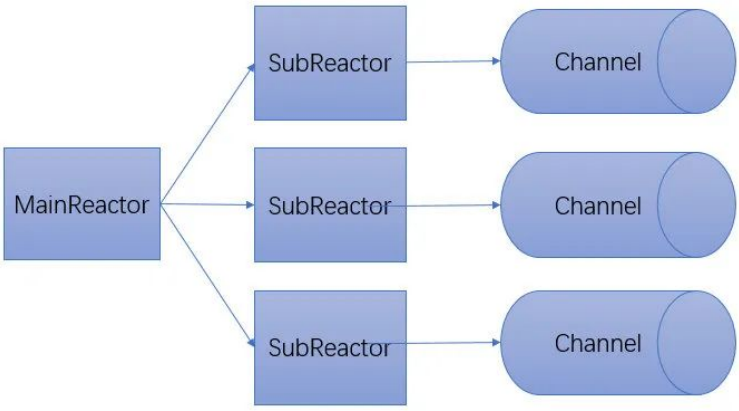
1.1 串行无锁

无锁串行最简单的实现方式可能就是单线程模型了，如 redis/Nginx 都采用了这种方式。在网络编程模型中，常规的方式是主线程负责处理 I/O 事件，并将读到的数据压入队列，工作线程则从队列中取出数据进行处理，这种半同步/半异步模型需要对队列进行加锁，如下图所示：



单Reactor多线程模型

上图的模式可以改成无锁串行的形式，当 MainReactor accept 一个新连接之后从众多的 SubReactor 选取一个进行注册，通过创建一个 Channel 与 I/O 线程进行绑定，此后该连接的读写都在同一个线程执行，无需进行同步。



主从Reactor职责链模型

1.2 结构无锁

利用硬件支持的原子操作可以实现无锁的数据结构，很多语言都提供 CAS 原子操作（如 go 中的 atomic 包和 C++11 中的 atomic 库），可以用于实现无锁队列。我们以一个简单的线程安全单链表的插入操作来看下无锁编程和普通加锁的区别。

```
template<typename T>
struct Node
{
    Node(const T &value) : data(value) { }
    T data;
    Node *next = nullptr;
};
```

有锁链表 WithLockList：

```
template<typename T>
class WithLockList
{
    mutex mtx;
    Node<T> *head;
public:
    void pushFront(const T &value)
    {
        auto *node = new Node<T>(value);
        lock_guard<mutex> lock(mtx); //①
        node->next = head;
        head = node;
    }
};
```

无锁链表 LockFreeList：

```
template<typename T>
class LockFreeList
{
```

```
{
    atomic<Node<T> *> head;
public:
    void pushFront(const T &value)
    {
        auto *node = new Node<T>(value);
        node->next = head.load();
        while(!head.compare_exchange_weak(node->next, node)); //②
    }
};
```

从代码可以看出，在有锁版本中 ① 进行了加锁。在无锁版本中，② 使用了原子 CAS 操作 compare_exchange_weak，该函数如果存储成功则返回 true，同时为了防止伪失败（即原始值等于期望值时也不一定存储成功，主要发生在缺少单条比较交换指令的硬件机器上），通常将 CAS 放在循环中。

下面对有锁和无锁版本进行简单的性能比较，分别执行 1000,000 次 push 操作。测试代码如下：

```
int main()
{
    const int SIZE = 1000000;
    //有锁测试
    auto start = chrono::steady_clock::now();
    WithLockList<int> wlList;
    for(int i = 0; i < SIZE; ++i)
    {
        wlList.pushFront(i);
    }
    auto end = chrono::steady_clock::now();
    chrono::duration<double, std::micro> micro = end - start;
    cout << "with lock list costs micro:" << micro.count() << endl;

    //无锁测试
    start = chrono::steady_clock::now();
    LockFreeList<int> lfList;
    for(int i = 0; i < SIZE; ++i)
    {
        lfList.pushFront(i);
    }
```

```
    end = chrono::steady_clock::now();
    micro = end - start;
    cout << "free lock list costs micro:" << micro.count() << endl;

    return 0;
}
```

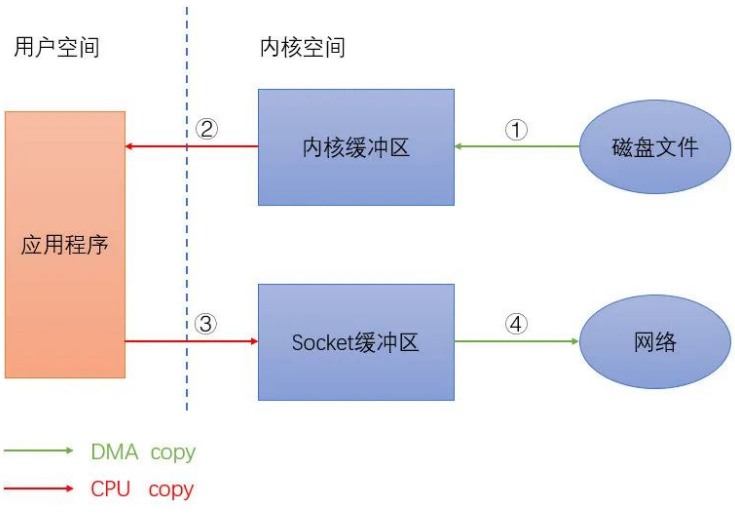
三次输出如下，可以看出无锁版本有锁版本性能高一些。with lock list costs micro:548118 free lock list costs micro:491570 with lock list costs micro:556037 free lock list costs micro:476045 with lock list costs micro:557451 free lock list costs micro:481470

2 零拷贝

这里的拷贝指的是数据在内核缓冲区和应用程序缓冲区直接的传输，并非指进程空间中的内存拷贝（当然这方面也可以实现零拷贝，如传引用和 C++中 move 操作）。现在假设我们有个服务，提供用户下载某个文件，当请求到来时，我们把服务器磁盘上的数据发送到网络中，这个流程伪代码如下：

```
filefd = open(...); //打开文件
sockfd = socket(...); //打开socket
buffer = new buffer(...); //创建buffer
read(filefd, buffer); //从文件内容读到buffer中
write(sockfd, buffer); //将buffer中的内容发送到网络
```

数据拷贝流程如下图：



上图中绿色箭头表示 DMA copy，DMA（Direct Memory Access）即直接存储器存取，是一种快速传送数据的机制，指外部设备不通过 CPU 而直接与系统内存交换数据的接口技术。红色箭头表示 CPU copy。即使在有 DMA 技术的情况下还是存在 4 次拷贝，DMA copy 和 CPU copy 各 2 次。

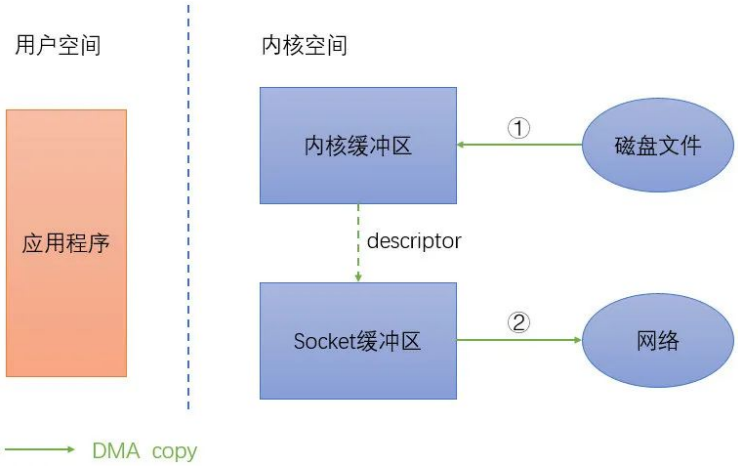
2.1 内存映射

内存映射将用户空间的一段内存区域映射到内核空间，用户对这段内存区域的修改可以直接反映到内核空间，同样，内核空间对这段区域的修改也直接反映用户空间，简单来说就是用户空间共享这个内核缓冲区。

使用内存映射来改写后的伪代码如下：

```
filefd = open(...); //打开文件
sockfd = socket(...); //打开socket
buffer = mmap(filefd); //将文件映射到进程空间
write(sockfd, buffer); //将buffer中的内容发送到网络
```

使用内存映射后数据拷贝流如下图所示：



内存映射

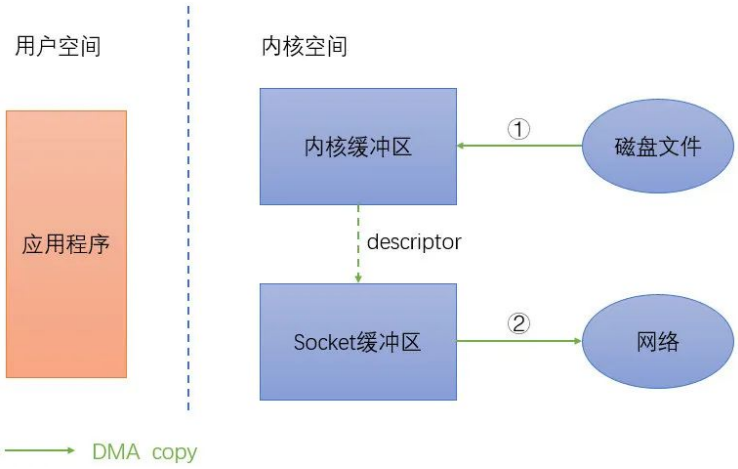
从图中可以看出，采用内存映射后数据拷贝减少为 3 次，不再经过应用程序直接将内核缓冲区中的数据拷贝到 Socket 缓冲区中。RocketMQ 为了消息存储高性能，就使用了内存映射机制，将存储文件分割成多个大小固定的文件，基于内存映射执行顺序写。

2.2 零拷贝

零拷贝就是一种避免 CPU 将数据从一块存储拷贝到另外一块存储，从而有效地提高数据传输效率的技术。Linux 内核 2.4 以后，支持带有 DMA 收集拷贝功能的传输，将内核页缓存中的数据直接打包发到网络上，伪代码如下：

```
filefd = open(...); //打开文件
sockfd = socket(...); //打开socket
sendfile(sockfd, filefd); //将文件内容发送到网络
```

使用零拷贝后流程如下图：



零拷贝

零拷贝的步骤为：1）DMA 将数据拷贝到 DMA 引擎的内核缓冲区中；2）将数据的位置和长度的信息的描述符加到套接字缓冲区；3）DMA 引擎直接将数据从内核缓冲区传递到协议引擎；

可以看出，零拷贝并非真正的没有拷贝，还是有 2 次内核缓冲区的 DMA 拷贝，只是消除了内核缓冲区和用户缓冲区之间的 CPU 拷贝。Linux 中主要的零拷贝函数有 sendfile、splice、tee 等。下图是来住 IBM 官网上普通传输和零拷贝传输的性能对比，可以看出零拷贝比普通传输快了 3 倍左右，Kafka 也使用零拷贝技术。

File size	Normal file transfer (ms)	transferTo (ms)
7MB	156	45
21MB	337	128
63MB	843	387
98MB	1320	617
200MB	2124	1150
350MB	3631	1762

700MB	13498	4422

普通读写和零拷贝性能对比

3 序列化

当将数据写入文件、发送到网络、写入到存储时通常需要序列化（serialization）技术，从其读取时需要进行反序列化（deserialization），又称编码（encode）和解码（decode）。序列化作为传输数据的表示形式，与网络框架和通信协议是解耦的。如网络框架 taf 支持 jce、json 和自定义序列化，HTTP 协议支持 XML、JSON 和流媒体传输等。

序列化的方式很多，作为数据传输和存储的基础，如何选择合适的序列化方式尤其重要。

3.1 分类

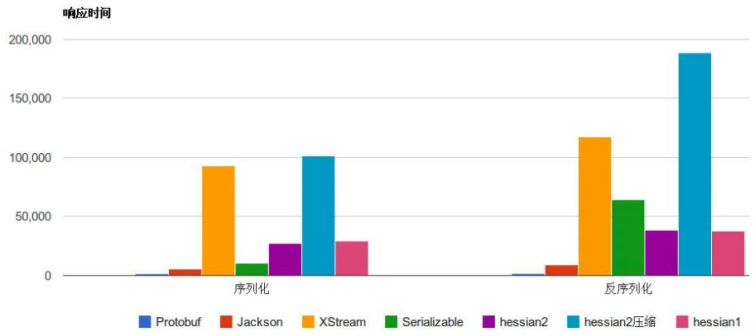
通常而言，序列化技术可以大致分为以下三种类型：

- **内置类型**：指编程语言内置支持的类型，如 java 的 java.io.Serializable。这种类型由于与语言绑定，不具有通用性，而且一般性能不佳，一般只在局部范围内使用。
- **文本类型**：一般是标准化的文本格式，如 XML、JSON。这种类型可读性较好，且支持跨平台，具有广泛的应用。主要缺点是比较臃肿，网络传输占用带宽大。
- **二进制类型**：采用二进制编码，数据组织更加紧凑，支持多语言和多平台。常见的有 Protocol Buffer/Thrift/MessagePack/FlatBuffer 等。

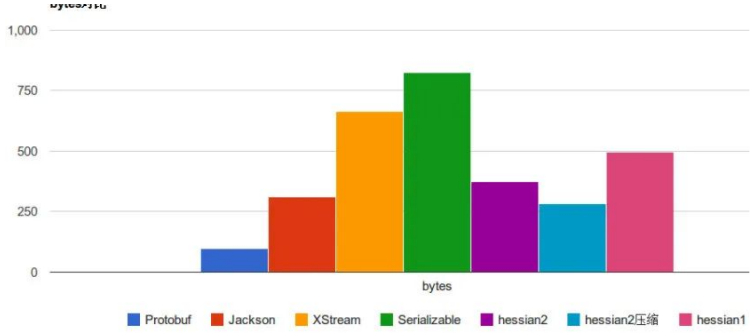
3.2 性能指标

衡量序列化/反序列化主要有三个指标：1）序列化之后的字节大小；2）序列化/反序列化的速度；3）CPU 和内存消耗；

下图是一些常见的序列化框架性能对比：



序列化和反序列化速度对比



序列化字节占用对比

可以看出 Protobuf 无论是在序列化速度上还是字节占比上可以说是完爆同行。不过人外有人，天外有天，听说 FlatBuffer 比 Protobuf 更加无敌，下图是来自 Google 的 FlatBuffer 和其他序列化性能对比，光看图中数据 FB 貌似秒杀 PB 的存在。

	FlatBuffers (binary)	Protocol Buffers LITE	Rapid JSON	FlatBuffers (JSON)	pugixml	Raw structs
Decode + Traverse + Dealloc (1 million times, seconds)	0.08	302	583	105	196	0.02
Decode / Traverse / Dealloc (breakdown)	0 / 0.08 / 0	220 / 0.15 / 81	294 / 0.9 / 287	70 / 0.08 / 35	41 / 3.9 / 150	0 / 0.02 / 0
Encode (1 million times, seconds)	3.2	185	650	169	273	0.15
Wire format size (normal / zlib, bytes)	344 / 220	228 / 174	1475 / 322	1029 / 298	1137 / 341	312 / 187
Memory needed to store decoded wire (bytes / blocks)	0 / 0	760 / 20	65689 / 4	328 / 1	34194 / 3	0 / 0
Transient memory allocated during decode (KB)	0	1	131	4	34	0
Generated source code size (KB)	4	61	0	4	0	0
Field access in handwritten traversal code	typed accessors	typed accessors	manual error checking	typed accessors	manual error checking	typed but no safety
Library source code (KB)	15	some subset of 3800	87	43	327	0

FlatBuffer性能对比

3.3 选型考量

在设计和选择序列化技术时，要进行多方面的考量，主要有以下几个方面：1）**性能**：CPU 和字节占用大小是序列化的主要开销。在基础的 RPC 通信、存储系统和高并发业务上应该选择高性能高压缩的二进制序列化。一些内部服务、请求较少 Web 的应用可以采用文本的 JSON，浏览器直接内置支持 JSON。2）**易用性**：丰富数据结构和辅助工具能提高易用性，减少业务代码的开发量。现在很多序列化框架都支持 List、Map 等多种结构和可读的打印。3）**通用性**：现代的服务往往涉及多语言、多平台，能否支持跨平台跨语言的互通是序列化选型的基本条件。4）**兼容性**：现代的服务都是快速迭代和升级，一个好的序列化框架应该有良好的向前兼容性，支持字段的增减和修改等。5）**扩展性**：序列化框架能否低门槛的支持自定义的格式有时候也是一个比较重要的考虑因素。

4 池子化

池化恐怕是最常用的一种技术了，其本质就是通过创建池子来提高对象复用，减少重复创建、销毁的开销。常用的池化技术有内存池、线程池、连接池、对象池等。

4.1 内存池

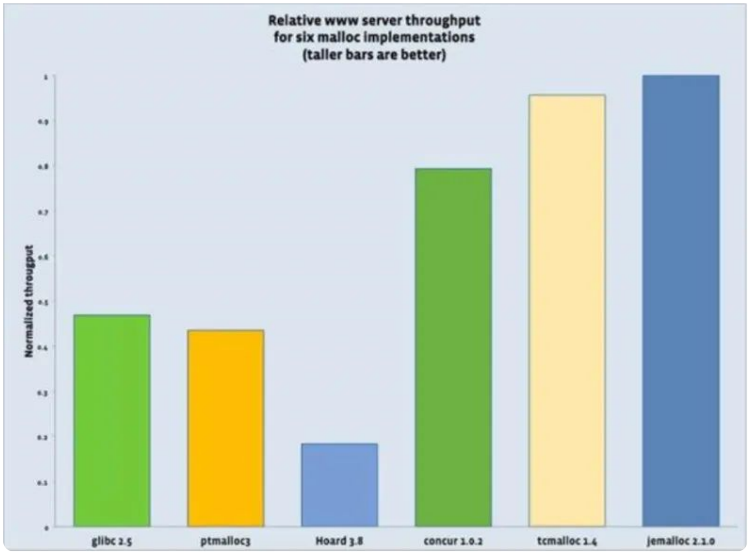
我们都知道，在 C/C++中分别使用 malloc/free 和 new/delete 进行内存的分配，其底层调用系统调用 sbrk/brk。频繁的调用系统调用分配释放内存不但影响性能还容易造成内存碎片，内存池技术旨在解决这些问题。正是这些原因，C/C++中的内存操作并不是直接调用系统调用，而是已经实现了自己的一套内存管理，malloc 的实现主要有三大实现。

1）ptmalloc：glibc 的实现。

2）tcmalloc：Google 的实现。

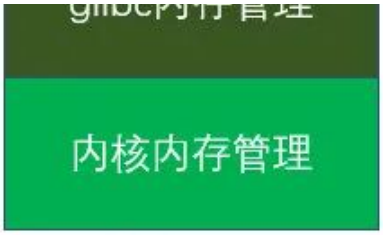
3）jemalloc：Facebook 的实现。

下面是来自网上的三种 malloc 的比较图，tcmalloc 和 jemalloc 性能差不多，ptmalloc 的性能不如两者，我们可以根据需要选用更适合的 malloc，如 redis 和 mysl 都可以指定使用哪个 malloc。至于三者的实现和差异，可以网上查阅。



内存分配器性能对比

虽然标准库的实现在操作系统内存管理的基础上再加了一层内存管理，但应用程序通常也会实现自己特定的内存池，如为了引用计数或者专门用于小对象分配。所以看起来内存管理一般分为三个层次。



内存管理三个层次

4.2 线程池

线程创建是需要分配资源的，这存在一定的开销，如果我们一个任务就创建一个线程去处理，这必然会影响系统的性能。线程池的可以限制线程的创建数量并重复使用，从而提高系统的性能。

线程池可以分类或者分组，不同的任务可以使用不同的线程组，可以进行隔离以免互相影响。对于分类，可以分为核心和非核心，核心线程池一直存在不会被回收，非核心可能对空闲一段时间后的线程进行回收，从而节省系统资源，等到需要时在按需创建放入池子中。

4.3 连接池

常用的连接池有数据库连接池、redis 连接池、TCP 连接池等等，其主要目的是通过复用来减少创建和释放连接的开销。连接池实现通常需要考虑以下几个问题：

1）初始化：启动即初始化和惰性初始化。启动初始化可以减少一些加锁操作和需要时可直接使用，缺点是可能造成服务启动缓慢或者启动后没有任务处理，造成资源浪费。惰性初始化是真正有需要的时候再去创建，这种方式可能有助于减少资源占用，但是如果面对突发的任务请求，然后瞬间去创建一堆连接，可能会造成系统响应慢或者响应失败，通常我们会采用启动即初始化的方式。

2）连接数目：权衡所需的连接数，连接数太少则可能造成任务处理缓慢，太多不但使任务处理慢还会过度消耗系统资源。

3）连接取出：当连接池已经无可用连接时，是一直等待直到有可用连接还是分配一个新的临时连接。

4）连接放入：当连接使用完毕且连接池未满时，将连接放入连接池（包括 3 中创建的临时连接），否则关闭。

5）连接检测：长时间空闲连接和失效连接需要关闭并从连接池移除。常用的检测方法有：使用时检测和定期检测。

4.4 对象池

严格来说，各种池都是对象池模式的应用，包括前面的这三哥们。对象池跟各种池一样，也是缓存一些对象从而避免大量创建同一个类型的对象，同时限制了实例的个数。如 redis 中 0-9999 整数对象就通过采用对象池进行共享。在游戏开发中对象池模式经常使用，如进入地图时怪物和 NPC 的出现并不是每次都是重新创建，而是从对象池中取出。

5 并发化

5.1 请求并发

如果一个任务需要处理多个子任务，可以将没有依赖关系的子任务并发化，这种场景在后台开发很常见。如一个请求需要查询 3 个数据，分别耗时 T1、T2、T3，如果串行调用总耗时 T=T1+T2+T3。对三个任务执行并发，总耗时 T=max(T1,T 2,T3)。同理，写操作也如此。对于同种请求，还可以同时进行批量合并，减少 RPC 调用次数。

5.2 冗余请求

冗余请求指的是同时向后端服务发送多个同样的请求，谁响应快就是使用谁，其他的则丢弃。这种策略缩短了客户端的等待时间，但也使整个系统调用量猛增，一般适用于初始化或者请求少的场景。公司 WNS 的跑马模块其实就是这种机制，跑马模块为了快速建立长连接同时向后台多个 ip/port 发起请求，谁快就用谁，这在弱网的移动设备上特别有用，如果使用等待超时再重试的机制，无疑将大大增加用户的等待时间。

6 异步化

对于处理耗时的任务，如果采用同步等待的方式，会严重降低系统的吞吐量，可以通过异步化进行解决。异步在不同层面概念是有一些差异的，在这里我们不讨论异步 I/O。

6.1 调用异步化

在进行一个耗时的 RPC 调用或者任务处理时，常用的异步化方式如下：

- **Callback**：异步回调通过注册一个回调函数，然后发起异步任务，当任务执行完毕时会回调用户注册的回调函数，从而减少调用端等待时间。这种方式会造成代码分散难以维护，定位问题也相对困难。

- **Future**：当用户提交一个任务时会立刻先返回一个 Future，然后任务异步执行，后续可以通过 Future 获取执行结果。对 1.4.1 中请求并发，我们可以使用 Future 实现，伪代码如下：

```
//异步并发任务
Future<Response> f1 = Executor.submit(query1);
Future<Response> f2 = Executor.submit(query2);
Future<Response> f3 = Executor.submit(query3);

//处理其他事情
doSomething();

//获取结果
Response res1 = f1.getResult();
Response res2 = f2.getResult();
Response res3 = f3.getResult();
```

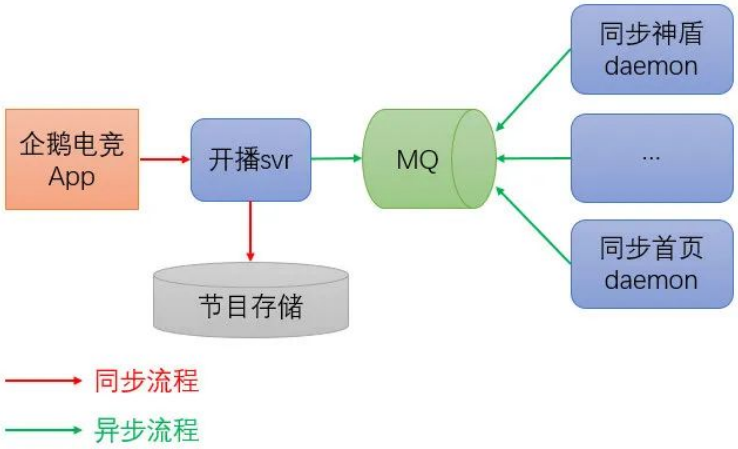
- CPS

（Continuation-passing style）可以对多个异步编程进行编排，组成更复杂的异步处理，并以同步的代码调用形式实现异步效果。CPS 将后续的处理逻辑当作参数传递给 Then 并可以最终捕获异常，解决了异步回调代码散乱和异常跟踪难的问题。Java 中的 CompletableFuture 和 C++ PPL 基本支持这一特性。典型的调用形式如下：

```
void handleRequest(const Request &req)
{
 return req.Read().Then([](Buffer &inbuf){
    return handleData(inbuf);
 }).Then([](Buffer &outbuf){
    return handleWrite(outbuf);
 }).Finally(){
    return cleanUp();
 });
}
```

6.2 流程异步化

一个业务流程往往伴随着调用链路长、后置依赖多等特点，这会同时降低系统的可用性和并发处理能力。可以采用对非关键依赖进行异步化解决。如企鹅电竞开播服务，除了开播写节目存储以外，还需要将节目信息同步到神盾推荐平台、App 首页和二级页等。由于同步到外部都不是开播的关键逻辑且对一致性要求不是很高，可以对这些后置的同步操作进行异步化，写完存储即向 App 返回响应，如下图所示：



企鹅电竞开播流程异步化

7 缓存

从单核 CPU 到分布式系统，从前端到后台，缓存无处不在。古有朱元璋“缓称王”而终得天下，今有不论是芯片制造商还是互联网公司都同样采取了“缓称王”（缓存称王）的政策才能占据一席之地。缓存是原始数据的一个复制集，其本质就是空间换时间，主要是为了解决高并发读。

7.1 缓存的使用场景

缓存是空间换时间的艺术，使用缓存能提高系统的性能。“劲酒虽好，可不要贪杯”，使用缓存的目的是为了提高性价比，而不是一上来就为了所谓的提高性能不计成本的使用缓存，而是要看场景。

适合使用缓存的场景，以之前参与过的项目企鹅电竞为例：

1）**一旦生成后基本不会变化的数据**：如企鹅电竞的游戏列表，在后台创建一个游戏之后基本很少变化，可直接缓存整个游戏列表；

2）**读密集型或存在热点的数据**：典型的就是各种 App 的首页，如企鹅电竞首页直播列表；

3）**计算代价大的数据**：如企鹅电竞的 Top 热榜视频，如 7 天榜在每天凌晨根据各种指标计算好之后缓存排序列表；

4）**千人一面的数据**：同样是企鹅电竞的 Top 热榜视频，除了缓存的整个排序列表，同时直接在进程内按页缓存了前 N 页数据组装后的最终回包结果；

不适合使用缓存的场景：

1）**写多读少，更新频繁**；

2）**对数据一致性要求严格**；

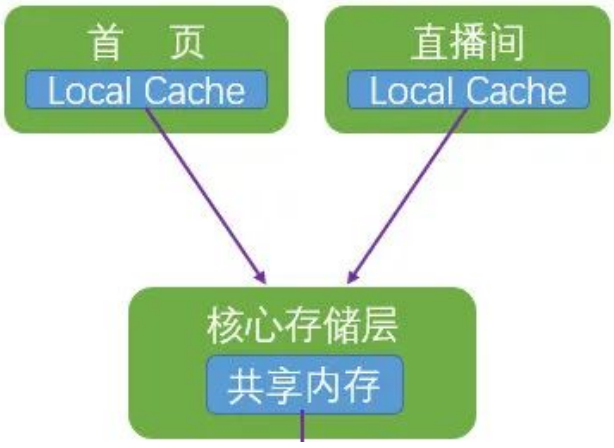
7.2 缓存的分类

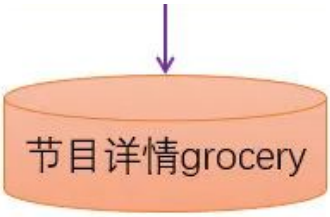
- **进程级缓存**：缓存的数据直接在进程地址空间内，这可能是访问速度最快使用最简单的缓存方式了。主要缺点是受制于进程空间大小，能缓存的数据量有限，进程重启缓存数据会丢失。一般通常用于缓存数据量不大的场景。

- **集中式缓存**：缓存的数据集中在一台机器上，如共享内存。这类缓存容量主要受制于机器内存大小，而且进程重启后数据不丢失。常用的集中式缓存中间件有单机版 redis、memcache 等。

- **分布式缓存**：缓存的数据分布在多台机器上，通常需要采用特定算法（如 Hash）进行数据分片，将海量的缓存数据均匀的分布在每个机器节点上。常用的组件有：Memcache（客户端分片）、Codis（代理分片）、Redis Cluster（集群分片）。

- **多级缓存**：指在系统中的不同层级的进行数据缓存，以提高访问效率和减少对后端存储的冲击。以下图的企鹅电竞的一个多级缓存应用，根据我们的现网统计，在第一级缓存的命中率就已经达 94%，穿透到 grocery 的请求量很小。





企鹅电竞首页多级缓存

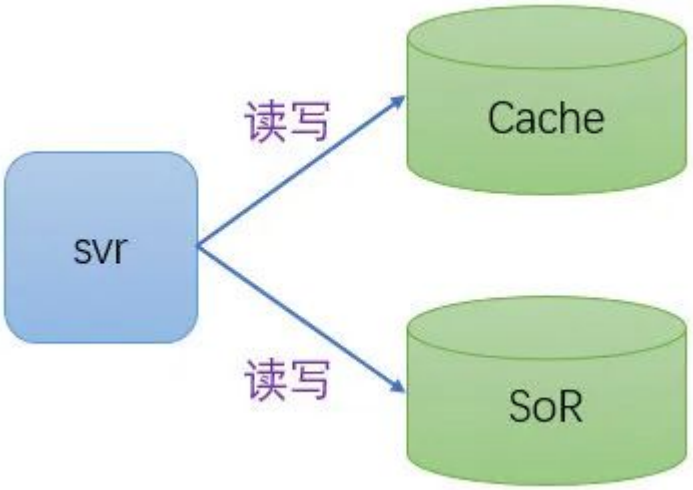
整体工作流程如下：

- 1）请求到达首页或者直播间服务后，如果在本地缓存命中则直接返回，否则从下一级缓存核心存储进行查询并更新本地缓存；
- 2）前端服务缓存没有命中穿透到核心存储服务，如果命中则直接返回给前端服务，没有则请求存储层 grocery 并更新缓存；
- 3）前两级 Cache 都没有命中回源到存储层 grocery。

7.3 缓存的模式

关于缓存的使用，已经有人总结出了一些模式，主要分为 Cache-Aside 和 Cache-As-SoR 两类。其中 SoR（system-of-record）：表示记录系统，即数据源，而 Cache 正是 SoR 的复制集。

Cache-Aside：旁路缓存，这应该是最常见的缓存模式了。对于读，首先从缓存读取数据，如果没有命中则回源 SoR 读取并更新缓存。对于写操作，先写 SoR，再写缓存。这种模式架构图如下：



Cache-Aside结构图

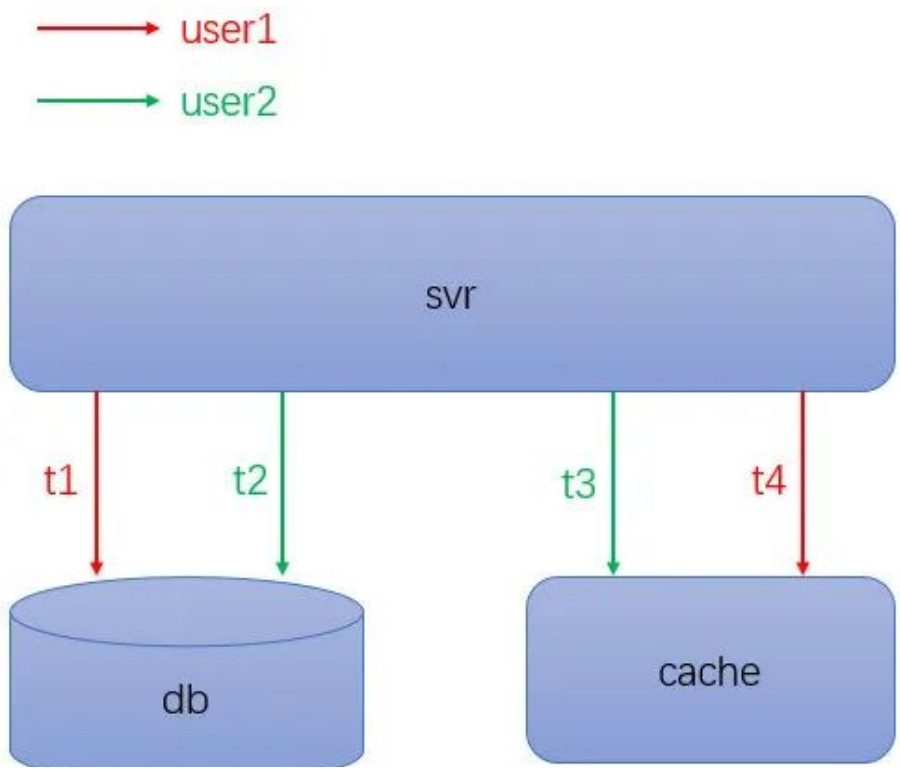
逻辑代码：

```
//读操作
data = Cache.get(key);
if(data == NULL)
{
    data = SoR.load(key);
    Cache.set(key, data);
}

//写操作
if(SoR.save(key, data))
{
    Cache.set(key, data);
}
```

这种模式用起来简单，但对应用层不透明，需要业务代码完成读写逻辑。同时对于写来说，写数据源和写缓存不是一个原子操作，可能出现以下情况导致两者数据不一致：

1）在并发写时，可能出现数据不一致。如下图所示，user1 和 user2 几乎同时进行读写。在 t1 时刻 user1 写 db，t2 时刻 user2 写 db，紧接着在 t3 时刻 user2 写缓存，t4 时刻 user1 写缓存。这种情况导致 db 是 user2 的数据，缓存是 user1 的数据，两者不一致。



Cache-Aside并发读写

2）先写数据源成功，但是接着写缓存失败，两者数据不一致。对于这两种情况如果业务不能忍受，可简单的通过先 delete 缓存然后再写 db 解决，其代价就是下一次读请求的 cache miss。

Cache-As-SoR：缓存即数据源，该模式把 Cache 当作 SoR，所以读写操作都是针对 Cache，然后 Cache 再将读写操作委托给 SoR，即 Cache 是一个代理。如下图所示：

Cache-As-SoR结构图

Cache-As-SoR 有三种实现：

1）**Read-Through**：发生读操作时，首先查询 Cache，如果不命中则再由 Cache 回源到 SoR 即存储端实现 Cache-Aside 而不是业务）。

2）**Write-Through**：称为穿透写模式，由业务先调用写操作，然后由 Cache 负责写缓存和 SoR。

3）**Write-Behind**：称为回写模式，发生写操作时业务只更新缓存并立即返回，然后异步写 SoR，这样可以利用合并写/批量写提高性能。

7.4 缓存的回收策略

在空间有限、低频热点访问或者无主动更新通知的情况下，需要对缓存数据进行回收，常用的回收策略有以下几种：

1）**基于时间**：基于时间的策略主要可以分两种：

- 基于 TTL（Time To Live）：即存活期，从缓存数据创建开始到指定的过期时间段，不管有没有访问缓存都会过期。如 redis 的 EXPIRE。
- 基于 TTI（Time To Idle）：即空闲期，缓存在指定的时间没有被访问将会被回收。

2）**基于空间**：缓存设置了存储空间上限，当达到上限时按照一定的策略移除数据。

3）**基于容量**：缓存设置了存储条目上限，当达到上限时按照一定的策略移除数据。

4）**基于引用**：基于引用计数或者强弱引用的一些策略进行回收。

缓存的常见回收算法如下：

- FIFO（First In First Out）：先进选出原则，先进入缓存的数据先被移除。
- LRU（Least Recently Used）：最基于局部性原理，即如果数据最近被使用，那么它在未来也极有可能被使用，反之，如果数据很久未使用，那么未来被使用的概率也较。
- LFU：（Least Frequently Used）：最近最少被使用的数据最先被淘汰，即统计每个对象的使用次数，当需要淘汰时，选择被使用次数最少的淘汰。

7.5 缓存的崩溃与修复

由于在设计不足、请求攻击（并不一定是恶意攻击）等会造成一些缓存问题，下面列出了常见的缓存问题和解决方案。

缓存穿透：大量使用不存在的 key 进行查询时，缓存没有命中，这些请求都穿透到后端的存储，最终导致后端存储压力过大甚至被压垮。这种情况原因一般是存储中数据不存在，主要有两个解决办法。

- 1）设置空置或默认值：如果存储中没有数据，则设置一个空置或者默认值缓存起来，这样下次请求时就不会穿透到后端存储。但这种情况如果遇到恶意攻击，不断的伪造不同的 key 来查询时并不能很好的应对，这时候需要引入一些安全策略对请求进行过滤。
- 2）布隆过滤器：采用布隆过滤器将，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层数据库的查询压力。

缓存雪崩：指大量的缓存在某一段时间内集体失效，导致后端存储负载瞬间升高甚至被压垮。通常是以下原因造成：

- 1）缓存失效时间集中在某段时间，对于这种情况可以采取对不同的 key 使用不同的过期时间，在原来基础失效时间的基础上再加上不同的随机时间；
- 2）采用取模机制的某缓存实例宕机，这种情况移除故障实例后会导致大量的缓存不命中。有两种解决方案：① 采取主从备份，主节点故障时直接将从实例替换主；② 使用一致性哈希替代取模，这样即使有实例崩溃也只是少部分缓存不命中。

缓存热点：虽然缓存系统本身性能很高，但也架不住某些热点数据的高并发访问从而造成缓存服务本身过载。假设一下微博以用户 id 作为哈希 key，突然有一天志玲姐姐宣布结婚了，如果

她的微博内容按照用户 id 缓存在某个节点上，当她的万千粉丝查看她的微博时必然会压垮这个缓存节点，因为这个 key 太热了。这种情况可以通过生成多份缓存到不同节点上，每份缓存的内容一样，减轻单个节点访问的压力。

7.6 缓存的一些好实践

1）**动静分离**：对于一个缓存对象，可能分为很多种属性，这些属性中有的是静态的，有的是动态的。在缓存的时候最好采用动静分离的方式。如企鹅电竞的视频详情分为标题、时长、清晰度、封面 URL、点赞数、评论数等，其中标题、时长等属于静态属性，基本不会改变，而点赞数、评论数经常改变，在缓存时这两部分开，以免因为动态属性每次的变更要把整个视频缓存拉出来进行更新一遍，成本很高。

2）**慎用大对象**：如果缓存对象过大，每次读写开销非常大并且可能会卡住其他请求，特别是在 redis 这种单线程的架构中。典型的情况是将一堆列表挂在某个 value 的字段上或者存储一个没有边界的列表，这种情况下需要重新设计数据结构或者分割 value 再由客户端聚合。

3）**过期设置**：尽量设置过期时间减少脏数据和存储占用，但要注意过期时间不能集中在某个时间段。

4）**超时设置**：缓存作为加速数据访问的手段，通常需要设置超时时间而且超时时间不能过长（如 100ms 左右），否则会导致整个请求超时连回源访问的机会都没有。

5）**缓存隔离**：首先，不同的业务使用不同的 key，防止出现冲突或者互相覆盖。其次，核心和非核心业务进行通过不同的缓存实例进行物理上的隔离。

6）**失败降级**：使用缓存需要有一定的降级预案，缓存通常不是关键逻辑，特别是对于核心服务，如果缓存部分失效或者失败，应该继续回源处理，不应该直接中断返回。

7）**容量控制**：使用缓存要进行容量控制，特别是本地缓存，缓存数量太多内存紧张时会频繁的 swap 存储空间或 GC 操作，从而降低响应速度。

8）**业务导向**：以业务为导向，不要为了缓存而缓存。对性能要求不高或请求量不大，分布式缓存甚至数据库都足以应对时，就不需要增加本地缓存，否则可能因为引入数据节点复制和幂等处理逻辑反而得不偿失。

9）**监控告警**：跟妹纸永远是对的一样，总不会错。对大对象、慢查询、内存占用等进行监控。

8 分片

分片即将一个较大的部分分成多个较小的部分，在这里我们分为数据分片和任务分片。对于数据分片，在本文将不同系统的拆分技术术语（如 region、shard、vnode、partition）等统称为分片。分片可以说是一箭三雕的技术，将一个大数据集分散在更多节点上，单点的读写负载随之也分散到了多个节点上，同时还提高了扩展性和可用性。

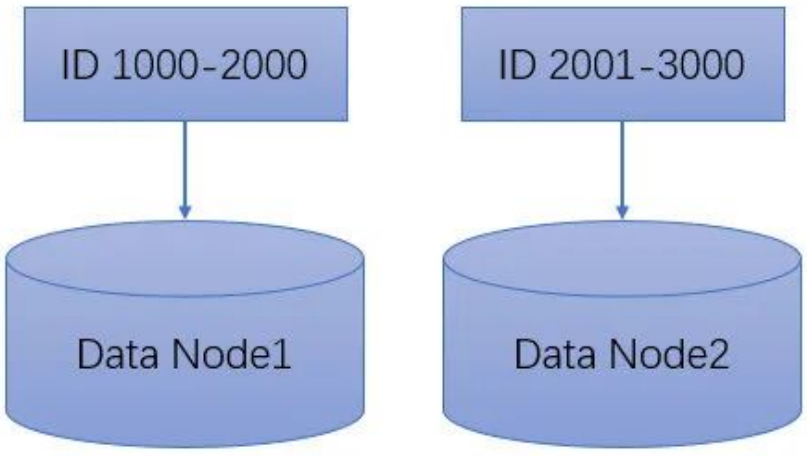
数据分片，小到编程语言标准库里的集合，大到分布式中间件，无所不在。如我曾经写过一个线程安全的容器以放置各种对象时，为了减少锁争用，对容器进行了分段，每个分段一个锁，按照哈希或者取模将对象放置到某个分段中，如 Java 中的 ConcurrentHashMap 也采取了分段的机制。分布式消息中间件 Kafka 中对 topic 也分成了多个 partition，每个 partition 互相独立可以比并发读写。

8.1 分片策略

进行分片时，要尽量均匀的将数据分布在所有节点上以平摊负载。如果分布不均，会导致倾斜使得整个系统性能的下降。常见的分片策略如下：

- 区间分片

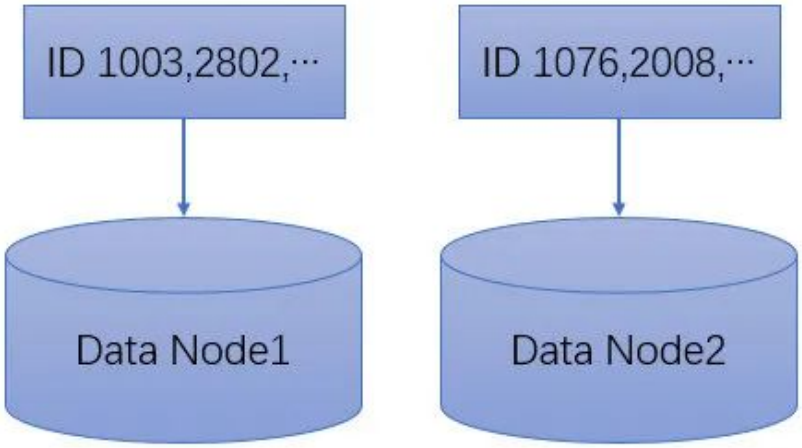
  基于一段连续关键字的分片，保持了排序，适合进行范围查找，减少了垮分片读写。区间分片的缺点是容易造成数据分布不均匀，导致热点。如直播平台，如果按 ID 进行区间分片，通常短位 ID 都是一些大主播，如在 100-1000 内 ID 的访问肯定比十位以上 ID 频繁。常见的还有按时间范围分片，则最近时间段的读写操作通常比很久之前的时间段频繁。



区间分片

- 随机分片

按照一定的方式（如哈希取模）进行分片，这种方式数据分布比较均匀，不容易出现热点和并发瓶颈。缺点就是失去了有序相邻的特性，如进行范围查询时会向多个节点发起请求。



随机分片

- **组合分片**：对区间分片和随机分片的一种折中，采取了两种方式的组合。通过多个键组成复合键，其中第一个键用于做哈希随机，其余键用于进行区间排序。如直播平台以主播id+开播时间（anchor_id,live_time）作为组合键，那么可以高效的查询某主播在某个时间段内的开播记录。社交场景，如微信朋友圈、QQ 说说、微博等以用户 id+发布时间(user_id,pub_time)的组合找到用户某段时间的发表记录。

8.2 二级索引

二级索引通常用来加速特定值的查找，不能唯一标识一条记录，使用二级索引需要二次查找。关系型数据库和一些 K-V 数据库都支持二级索引，如 mysql 中的辅助索引（非聚簇索引），ES 倒排索引通过 term 找到文档。

- 本地索引

  索引存储在与关键字相同的分区中，即索引和记录在同一个分区，这样对于写操作时都在一个分区里进行，不需要跨分区操作。但是对于读操作，需要聚合其他分区上的数据。如以王者荣耀短视频为例，以视频 vid 作为关键索引，视频标签（如五杀、三杀、李白、阿珂）作为二级索引，本地索引如下图所示：

本地索引

- 全局索引

  按索引值本身进行分区，与关键字所以独立。这样对于读取某个索引的数据时，都在一个分区里进行，而对于写操作，需要跨多个分区。仍以上面的例子为例，全局索引如下图所示：

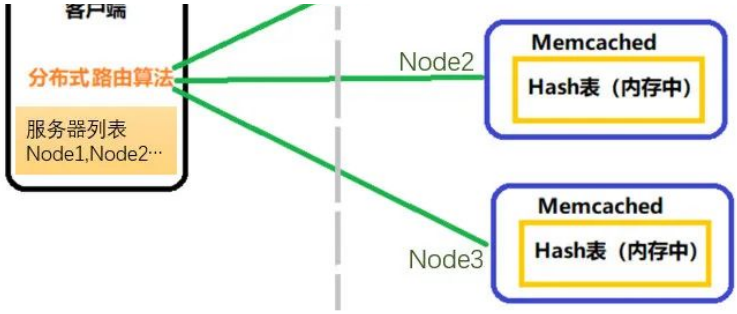
全局索引

8.3 路由策略

路由策略决定如何将数据请求发送到指定的节点，包括分片调整后的路由。通常有三种方式：客户端路由、代理路由和集群路由。

- 客户端路由

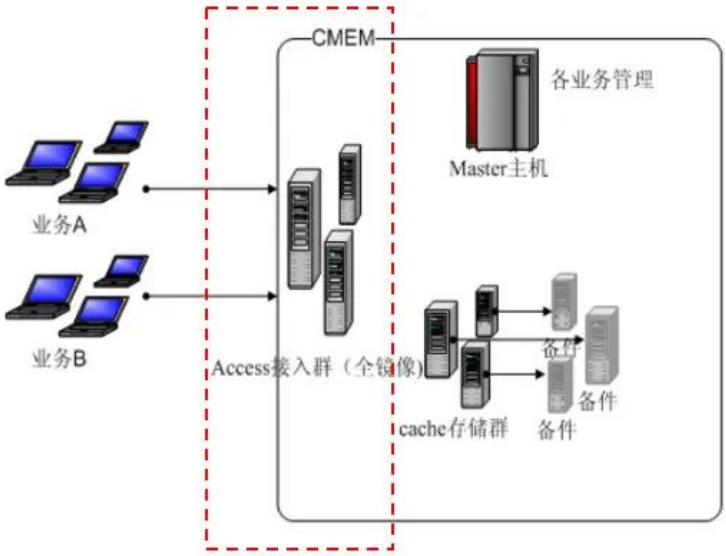
  客户端直接操作分片逻辑，感知分片和节点的分配关系并直接连接到目标节点。Memcache就是采用这种方式实现的分布式，如下图所示。



Memcache客户端路由

- 代理层路由

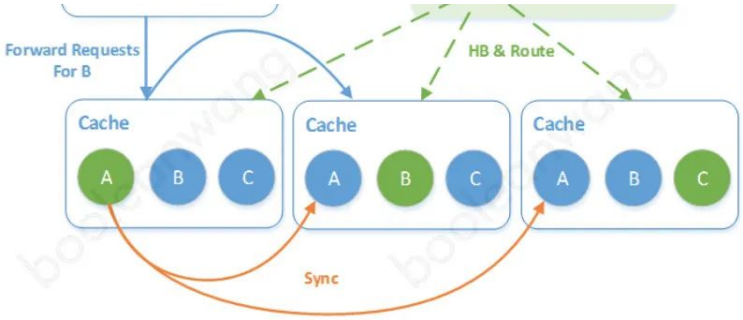
  客户端的请求到发送到代理层，由其将请求转发到对应的数据节点上。很多分布式系统都采取了这种方式，如业界的基于 redis 实现的分布式存储 codis（codis-proxy 层），公司内如 CMEM（Access 接入层）、DCache（Proxy+Router）等。如下图所示 CMEM 架构图，红色方框内的 Access 层就是路由代理层。



CMEM接入层路由

- 集群路由

  由集群实现分片路由，客户端连接任意节点，如果该节点存在请求的分片，则处理；否则将请求转发到合适的节点或者告诉客户端重定向到目标节点。如 redis cluster 和公司的 CKV+采用了这种方式，下图的 CKV+集群路由转发。



CKV+集群路由

以上三种路由方式都各优缺点，客户端路由实现相对简单但对业务入侵较强。代理层路由对业务透明，但增加了一层网络传输，对性能有一定影响，同时在部署维护上也相对复杂。集群路由对业务透明，且比代理路由少了一层结构，节约成本，但实现更复杂，且不合理的策略会增加多次网络传输。

8.4 动态平衡

在学习平衡二叉树和红黑树的时候我们都知道，由于数据的插入删除会破坏其平衡性。为了保持树的平衡，在插入删除后我们会通过左旋右旋动态调整树的高度以保持再平衡。在分布式数据存储也同样需要再平衡，只不过引起不平衡的因素更多了，主要有以下几个方面：

1）读写负载增加，需要更多 CPU；

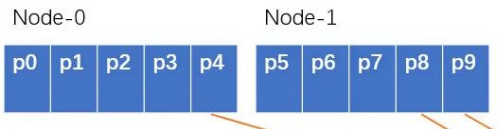
2）数据规模增加，需要更多磁盘和内存；

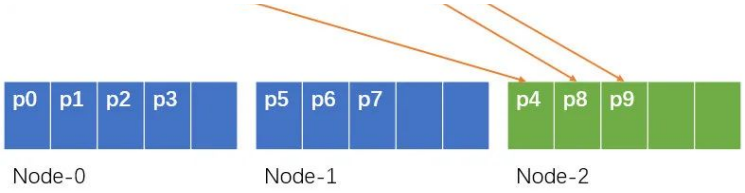
3）数据节点故障，需要其他节点接替；

业界和公司很多产品也都支持动态平衡调整，如 redis cluster 的 resharding，HDFS/kafka 的 rebalance。常见的方式如下：

- **固定分区**

  创建远超节点数的分区数，为每个节点分配多个分区。如果新增节点，可从现有的节点上均匀移走几个分区从而达到平衡，删除节点反之，如下图所示。典型的就是一致性哈希，创建 2^32-1 个虚拟节点（vnode）分布到物理节点上。该模式比较简单，需要在创建的时候就确定分区数，如果设置太小，数据迅速膨胀的话再平衡的代价就很大。如果分区数设置很大，则会有一定的管理开销。





固定分区再平衡

- **动态分区**

  自动增减分区数，当分区数据增长到一定阀值时，则对分区进行拆分。当分区数据缩小到一定阀值时，对分区进行合并。类似于 B+树的分裂删除操作。很多存储组件都采用了这种方式，如 HBase Region 的拆分合并，TDSQL 的 Set Shard。这种方式的优点是自动适配数据量，扩展性好。使用这种分区需要注意的一点，如果初始化分区为一个，刚上线请求量就很大的话会造成单点负载高，通常采取预先初始化多个分区的方式解决，如 HBase 的预分裂。

8.5 分库分表

当数据库的单表/单机数据量很大时，会造成性能瓶颈，为了分散数据库的压力，提高读写性能，需要采取分而治之的策略进行分库分表。通常，在以下情况下需要进行分库分表：

1）单表的数据量达到了一定的量级（如 mysql 一般为千万级），读写的性能会下降。这时索引也会很大，性能不佳，需要分解单表。

2）数据库吞吐量达到瓶颈，需要增加更多数据库实例来分担数据读写压力。

分库分表按照特定的条件将数据分散到多个数据库和表中，分为垂直切分和水平切分两种模式。

- **垂直切分**：按照一定规则，如业务或模块类型，将一个数据库中的多个表分布到不同的数据库上。以直播平台为例，将直播节目数据、视频点播数据、用户关注数据分别存储在不同的数据库上，如下图所示：

垂直切分

优点：

1）切分规则清晰，业务划分明确；

2）可以按照业务的类型、重要程度进行成本管理，扩展也方便；

3）数据维护简单；

缺点：

1）不同表分到了不同的库中，无法使用表连接 Join。不过在实际的业务设计中，也基本不会用到 join 操作，一般都会建立映射表通过两次查询或者写时构造好数据存到性能更高的存储系统中。

2）事务处理复杂，原本在事务中操作同一个库的不同表不再支持。如直播结束时更新直播节目同时生成一个直播的点播回放在分库之后就不能在一个事物中完成，这时可以采用柔性事务或者其他分布式事物方案。

- **水平切分**：按照一定规则，如哈希或取模，将同一个表中的数据拆分到多个数据库上。可以简单理解为按行拆分，拆分后的表结构是一样的。如直播系统的开播记录，日积月累，表会越来越大，可以按照主播 id 或者开播日期进行水平切分，存储到不同的数据库实例中。优点：1）切分后表结构一样，业务代码不需要改动；2）能控制单表数据量，有利于性能提升；缺点：1）Join、count、记录合并、排序、分页等问题需要跨节点处理；2）相对复杂，需要实现路由策略；综上所述，垂直切分和水平切分各有优缺点，通常情况下这两种模式会一起使用。

8.6 任务分片

记得小时候发新书，老师抱了一堆堆的新书到教室，然后找几个同学一起分发下去，有的发语文，有的发数学，有的发自然，这就是一种任务分片。车间中的流水线，经过每道工序的并行后最终合成最终的产品，也是一种任务分片。

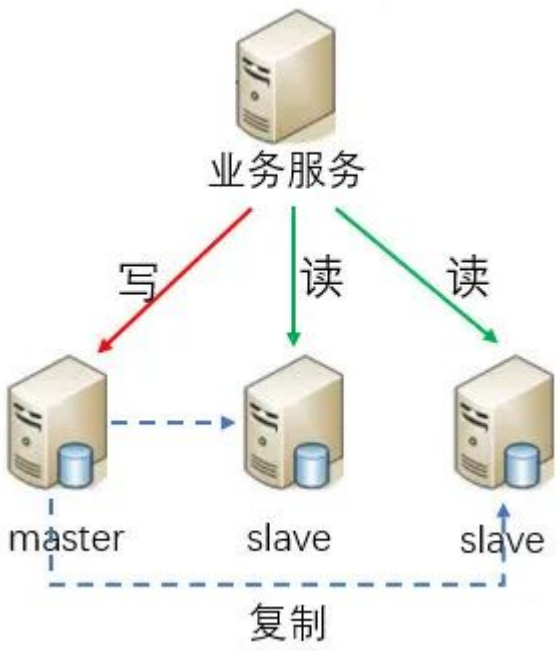
任务分片将一个任务分成多个子任务并行处理，加速任务的执行，通常涉及到数据分片，如归并排序首先将数据分成多个子序列，先对每个子序列排序，最终合成一个有序序列。在大数据处理中，Map/Reduce 就是数据分片和任务分片的经典结合。

9 存储

任何一个系统，从单核 CPU 到分布式，从前端到后台，要实现各式各样的功能和逻辑，只有读和写两种操作。而每个系统的业务特性可能都不一样，有的侧重读、有的侧重写，有的两者兼备，本节主要探讨在不同业务场景下存储读写的一些方法论。

9.1 读写分离

大多数业务都是读多写少，为了提高系统处理能力，可以采用读写分离的方式将主节点用于写，从节点用于读，如下图所示。



读写分离架构

读写分离架构有以下几个特点：1）数据库服务为主从架构，可以为一主一从或者一主多从；2）主节点负责写操作，从节点负责读操作；3）主节点将数据复制到从节点；基于基本架构，可以变种出多种读写分离的架构，如主-主-从、主-从-从。主从节点也可以是不同的存储，如 mysql+redis。

读写分离的主从架构一般采用异步复制，会存在数据复制延迟的问题，适用于对数据一致性要求不高的业务。可采用以下几个方式尽量避免复制滞后带来的问题。

1）写后读一致性：即读自己的写，适用于用户写操作后要求实时看到更新。典型的场景是，用户注册账号或者修改账户密码后，紧接着登录，此时如果读请求发送到从节点，由于数据可能还没同步完成，用户登录失败，这是不可接受的。针对这种情况，可以将自己的读请求发送到主节点上，查看其他用户信息的请求依然发送到从节点。

2）二次读取：优先读取从节点，如果读取失败或者跟踪的更新时间小于某个阀值，则再从主节点读取。

3）关键业务读写主节点，非关键业务读写分离。

4）单调读：保证用户的读请求都发到同一个从节点，避免出现回滚的现象。如用户在 M 主节点更新信息后，数据很快同步到了从节点 S1，用户查询时请求发往 S1，看到了更新的信息。接着用户再一次查询，此时请求发到数据同步没有完成的从节点 S2，用户看到的现象是刚才的更新的信息又消失了，即以为数据回滚了。

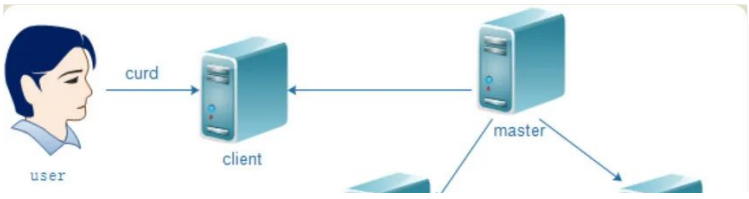
9.2 动静分离

动静分离将经常更新的数据和更新频率低的数据进行分离。最常见于 CDN，一个网页通常分为静态资源（图片/js/css 等）和动态资源（JSP、PHP 等），采取动静分离的方式将静态资源缓存在 CDN 边缘节点上，只需请求动态资源即可，减少网络传输和服务负载。

在数据库和 KV 存储上也可以采取动态分离的方式，如 7.6 提到的点播视频缓存的动静分离。在数据库中，动静分离更像是一种垂直切分，将动态和静态的字段分别存储在不同的库表中，减小数据库锁的粒度，同时可以分配不同的数据库资源来合理提升利用率。

9.3 冷热分离

冷热分离可以说是每个存储产品和海量业务的必备功能，Mysql、ElasticSearch、CMEM、Grocery 等都直接或间接支持冷热分离。将热数据放到性能更好的存储设备上，冷数据下沉到廉价的磁盘，从而节约成本。企鹅电竞为了节省在腾讯云成本，直播回放按照主播粉丝数和时间等条件也采用了冷热分离，下图是 ES 冷热分离的一个实现架构图。



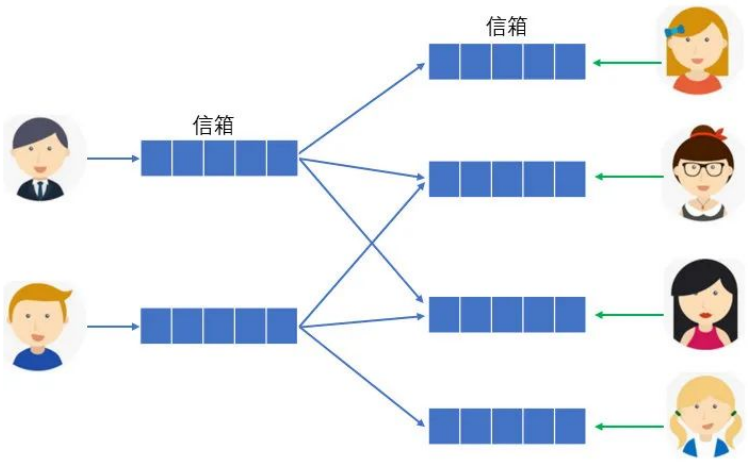
ES冷热分离架构图

9.4 重写轻读

重写轻度个人理解可能有两个含义：1）关键写，降低读的关键性，如异步复制，保证主节点写成功即可，从节点的读可容忍同步延迟。2）写重逻辑，读轻逻辑，将计算的逻辑从读转移到写。适用于读请求的时候还要进行计算的场景，常见的如排行榜是在写的时候构建而不是在读请求的时候再构建。

在微博、朋友圈等社交产品场景中都有类似关注或朋友的功能。以朋友圈模拟为例（具体我也不知道朋友圈是怎么做的），如果用户进入朋友圈时看到的朋友消息列表是在请求的时候遍历其朋友的新消息再按时间排序组装出来的，这显然很难满足朋友圈这么大的海量请求。可以采取重写轻读的方式，在发朋友圈的时候就把列表构造好，然后直接读就可以了。

仿照 Actor 模型，为用户建立一个信箱，用户发朋友圈后写完自己的信箱就返回，然后异步的将消息推送到其朋友的信箱，这样朋友读取他的信箱时就是其朋友圈的消息列表，如下图所示：



重写轻读流程

上图仅仅是为了展示重写轻度的思路，在实际应用中还有些其他问题。如：1）写扩散：这是个写扩散的行为，如果一个大户的朋友很多，这写扩散的代价也是很大的，而且可能有些人万年不看朋友圈甚至屏蔽了朋友。需要采取一些其他的策略，如朋友数在某个范围内是才采取这种方式，数量太多采取推拉结合和分析一些活跃指标等。2）信箱容量：一般来说查看朋友圈不会不断的往下翻页查看，这时候应该限制信箱存储条目数，超出的条目从其他存储查询。

9.5 数据异构

数据异构主要是按照不同的维度建立索引关系以加速查询。如京东、天猫等网上商城，一般按照订单号进行了分库分表。由于订单号不在同一个表中，要查询一个买家或者商家的订单列表，就需要查询所有分库然后进行数据聚合。可以采取构建异构索引，在生成订单的时同时创建买家和商家到订单的索引表，这个表可以按照用户 id 进行分库分表。

10 队列

在系统应用中，不是所有的任务和请求必须实时处理，很多时候数据也不需要强一致性而只需保持最终一致性，有时候我们也不需要知道系统模块间的依赖，在这些场景下队列技术大有可为。

10.1 应用场景

队列的应用场景很广泛，总结起来主要有以下几个方面：

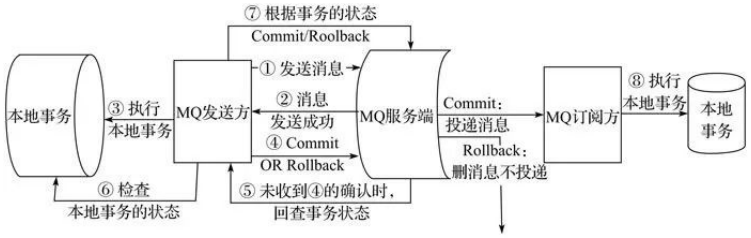
- **异步处理**：业务请求的处理流程通常很多，有些流程并不需要在本次请求中立即处理，这时就可以采用异步处理。如直播平台中，主播开播后需要给粉丝发送开播通知，可以将开播事件写入到消息队列中，然后由专门的 daemon 来处理发送开播通知，从而提高开播的响应速度。

- **流量削峰**：高并发系统的性能瓶颈一般在 I/O 操作上，如读写数据库。面对突发的流量，可以使用消息队列进行排队缓冲。以企鹅电竞为例，每隔一段时间就会有大主播入驻，如梦泪等。这个时候会有大量用户的订阅主播，订阅的流程需要进行多个写操作，这时先只写用户关注了哪个主播存储。然后在进入消息队列暂存，后续再写主播被谁关注和其他存储。

- **系统解耦**：有些基础服务被很多其他服务依赖，如企鹅电竞的搜索、推荐等系统需要开播事件。而开播服务本身并不关心谁需要这些数据，只需处理开播的事情就行了，依赖服务（包括第一点说的发送开播通知的 daemon）可以订阅开播事件的消息队列进行解耦。

- **数据同步**：消息队列可以起到数据总线的作用，特别是在跨系统进行数据同步时。拿我以前参与过开发的一个分布式缓存系统为例，通过 RabbitMQ 在写 Mysql 时将数据同步到 Redis，从而实现一个最终一致性的分布式缓存。

- **柔性事务**：传统的分布式事务采用两阶段协议或者其优化变种实现，当事务执行时都需要争抢锁资源和等待，在高并发场景下会严重降低系统的性能和吞吐量，甚至出现死锁。互联网的核心是高并发和高可用，一般将传统的事务问题转换为柔性事务。下图是阿里基于消息队列的一种分布式事务实现（详情查看：企业 IT 架构转型之道 阿里巴巴中台战略思想与架构实战，微信读书有电子版）：



基于MQ的分布式柔性事务

其核心原理和流程是：

1）分布式事务发起方在执行第一个本地事务前，向 MQ 发送一条事务消息并保存到服务端，MQ 消费者无法感知和消费该消息 ①②。

2）事务消息发送成功后开始进行单机事务操作 ③：

a）如果本地事务执行成功，则将 MQ 服务端的事务消息更新为正常状态 ④；

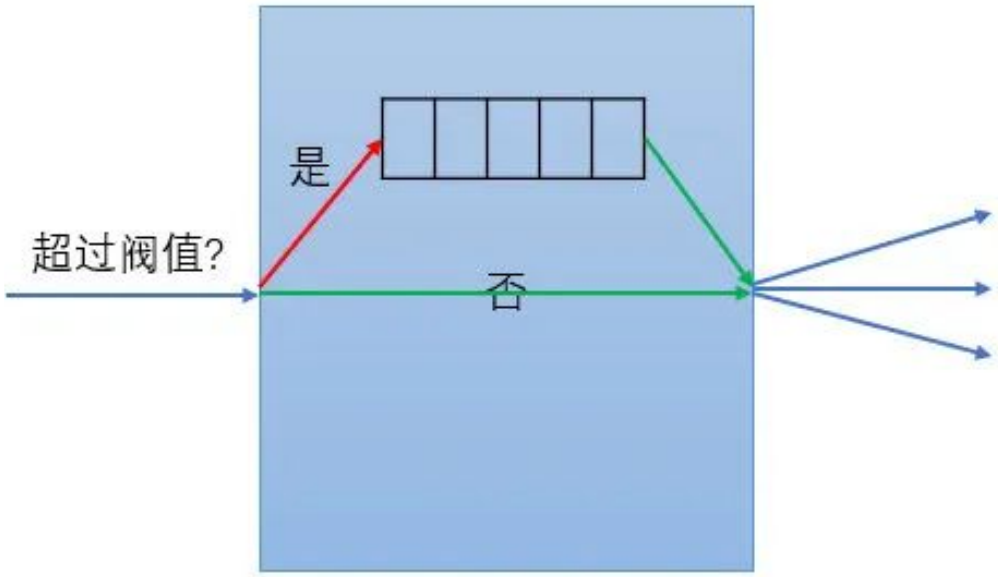
b）如果本地事务执行时因为宕机或者网络问题没有及时向 MQ 服务端反馈，则之前的事务消息会一直保存在 MQ。MQ 服务端会对事务消息进行定期扫描，如果发现有消息保存时间超过了一定的时间阀值，则向 MQ 生产端发送检查事务执行状态的请求 ⑤；

c）检查本地事务结果后 ⑥，如果事务执行成功，则将之前保存的事务消息更新为正常状态，否则告知 MQ 服务端进行丢弃；

3）消费者获取到事务消息设置为正常状态后，则执行第二个本地事务 ⑧。如果执行失败则通知 MQ 发送方对第一个本地事务进行回滚或正向补偿。

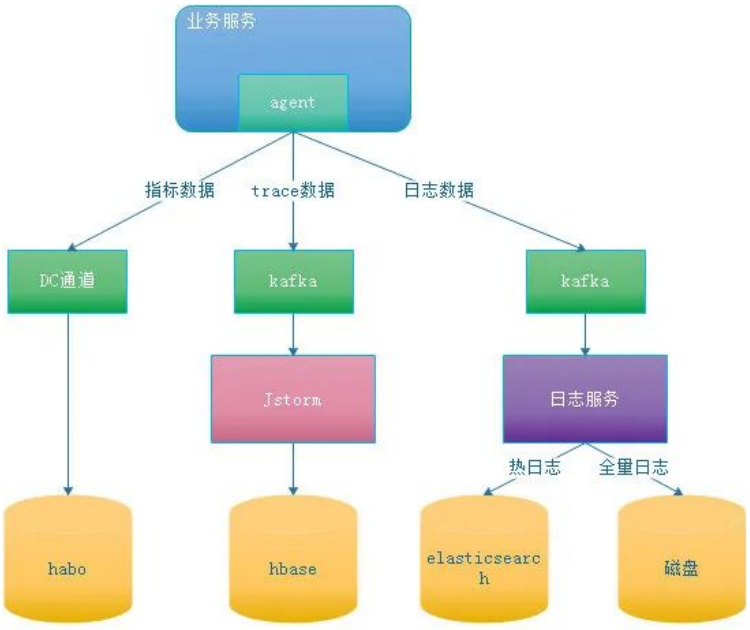
10.2 应用分类

- **缓冲队列**：队列的基本功能就是缓冲排队，如 TCP 的发送缓冲区，网络框架通常还会再加上应用层的缓冲区。使用缓冲队列应对突发流量时，使处理更加平滑，从而保护系统，上过 12306 买票的都懂。



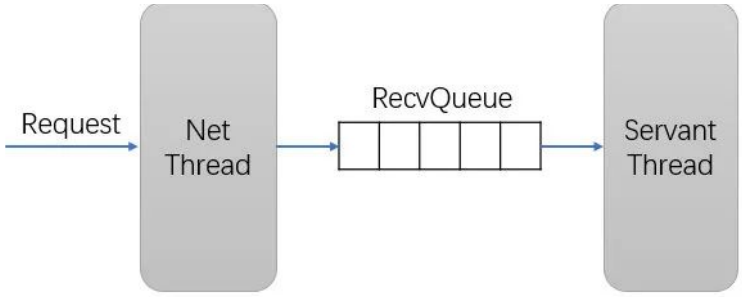
缓冲队列

在大数据日志系统中，通常需要在日志采集系统和日志解析系统之间增加日志缓冲队列，以防止解析系统高负载时阻塞采集系统甚至造成日志丢弃，同时便于各自升级维护。下图天机阁数据采集系统中，就采用 Kafka 作为日志缓冲队列。



天机阁数据采集系统

- **请求队列**：对用户的请求进行排队，网络框架一般都有请求队列，如 spp 在 proxy 进程和 work 进程之间有共享内存队列，taf 在网络线程和 Servant 线程之间也有队列，主要用于流量控制、过载保护和超时丢弃等。



TAF请求接收队列

- **任务队列**：将任务提交到队列中异步执行，最常见的就是线程池的任务队列。
- 消息队列

  用于消息投递，主要有点对点和发布订阅两种模式，常见的有 RabbitMQ、RocketMQ、Kafka 等，下图是常用消息队列的对比：

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
单机吞吐量	万级，比 RocketMQ、Kafka 低一个数量级	同 ActiveMQ	10 万级，支撑高吞吐	10 万级，高吞吐，一般配合大数据类的系统来进行实时数据计算、日志采集等场景
topic 数量对吞吐量的影响			topic 可以达到几百/几千的级别，吞吐量会有较小幅度的下降，这是 RocketMQ 的一大优势，在同等机器下，可以支撑大量的 topic	topic 从几十到几百个时候，吞吐量会大幅度下降，在同等机器下，Kafka 尽量保证 topic 数量不要过多，如果要支撑大规模的 topic，需要增加更多的机器资源
时效性	ms 级	微秒级，这是 RabbitMQ 的一大特点，延迟最低	ms 级	延迟在 ms 级以内
可用性	高，基于主从架构实现高可用	同 ActiveMQ	非常高，分布式架构	非常高，分布式，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用
消息可靠性	有较低的概率丢失数据		经过参数优化配置，可以做到 0 丢失	同 RocketMQ
功能支持	MQ 领域的功能极其完备	基于 erlang 开发，并发能力很强，性能极好，延时很低	MQ 功能较为完善，还是分布式的，扩展性好	功能较为简单，主要支持简单的 MQ 功能，在大数据领域的实时计算以及日志采集被大规模使用

常用消息队列

总结

本文探讨和总结了后台开发设计高性能服务的常用方法和技术，并通过思维导图总结了成一套方法论。当然这不是高性能的全部，甚至只是凤毛菱角。每个具体的领域都有自己的高性能之道，如网络编程的 I/O 模型和 C10K 问题，业务逻辑的数据结构和算法设计，各种中间件的参数调优等。文中也描述了一些项目的实践，如有不合理的地方或者有更好的解决方案，请各位同仁赐教。