第 1 章
性能测试、分析与调优基础

随着互联网的高速发展，无论过去、现在还是将来，性能测试和性能分析永远都是软件开发中一个无法回避的话题。一个网站在上线后，其性能的好坏会直接影响用户的体验，没有哪个用户可以忍受打开一个网站需要很长时间才有响应。所以性能测试和性能分析是任何一个网站、系统或者软件在上线前都需要去关注的核心问题。

性能测试除了为获取性能指标外，更多是为了发现性能瓶颈和性能问题，然后针对性能问题和性能瓶颈进行分析和调优。

1.1 性能测试的基础

性能可以理解为一个系统实现其功能的能力，从宏观上可以描述为系统能够稳定运行、高并发访问时系统不会出现宕机、系统处理完成用户请求需要的时间、系统能够同时支撑的并发访问量、系统每秒可以处理完成的事务数等；从微观上可以描述为处理每个事务的资源开销，资源的开销可以包括 CPU、磁盘 I/O、内存、网络传输带宽等，甚至可以体现为服务器连接数、线程数、JVM Heap 等的使用情况，也可以表现为内存的分配回收是否及时、缓存规则的命中率等。

性能到底有多重要呢？我们可以举一个网站访问的例子来说明，一个网页的加载速度如果超过 4~5 秒，可能 25%的人会选择放弃。百度的搜索结果响应时间慢 0.4 秒，一天的搜索量可能会减少千万次左右。所以一个系统、一个网站的性能决定了其能够支撑业务的能力。

不同的群体对性能的理解可能会存在很大的差异。

（1）普通的用户更加关心响应时间和稳定性：

- 要访问的页面还要等多久才能加载出来？
- 为什么有时候会访问失败？为什么会出现 502 错误？

（2）架构师和工程师可能更加关心架构设计和代码编写的性能：

- 应用架构设计是否合理？
- 技术架构设计是否合理？

- 数据架构设计是否合理？
- 部署架构设计是否合理？
- 代码是否存在性能问题？
- JVM 中是否有不合理的内存分配和使用？
- 线程同步和线程锁是否合理？
- 代码的计算算法是否可以进一步优化以减少 CPU 的消耗时间？

（3）运维工程师可能更加关心系统的监控以及稳定性情况：

- 服务器各项资源使用率在正常范围内吗？
- 数据库的连接数在正常范围内吗？
- SQL 执行时间正常吗，是否存在慢查询日志？
- 系统能够支撑 7 × 24 小时连续不间断的业务访问吗？
- 系统是高可用的吗，服务器节点宕机了会影响用户使用吗？
- 对节点扩容后，可以提高系统的性能吗？

1.1.1 性能测试的分类

性能测试的类型通常包括如下几种：

- 性能测试：寻求系统在正常负载下的各项性能指标，或者通过调整并发用户数，使系统资源的利用率处于正常水平时获取到系统的各项性能指标。
- 负载测试：系统在不同负载下的性能表现，通过该项测试可以寻求到系统在不同负载下的性能变化曲线，从而寻求到性能的拐点。例如负载测试时，在只不断递增并发用户数时，观察各项性能指标的变化规律，找到系统能到达的最大 TPS，并且观察此时系统处理的平均响应时间、各项系统资源和硬件资源的消耗情况。
- 压力测试（在后文也简称为压测）：系统在高负载下的性能表现，该项测试主要为了寻求系统能够承受的最大负载以及此时系统的吞吐率，通过该测试也可以发现系统在超高负载下是否会出现崩溃而无法访问，以及在系统负载减小后，系统性能能否自动恢复。
- 基准测试：针对待测系统开发中的版本执行的测试，采集各项性能指标作为后期版本性能的对比。
- 稳定性测试：以正常负载或者略高于正常负载来对系统进行长时间的测试，检测系统是否可以长久稳定运行，以及检查系统的各项性能指标会不会随着时间而发生明显变化。
- 扩展性测试：通常用于新上线的系统或者新搭建的系统环境，通过先测试单台服务器的处理能力，然后慢慢增加服务器的数量，测试集群环境下单台服务器的处理能力是否有损耗，集群环境的处理能力是否可以呈现稳定增加。

1.1.2　性能测试的场景

性能测试的场景类型通常包含如下几种：

- 业务场景：通常指的是系统的业务处理流程，描述具体的用户行为，通过对用户行为进行分析来划分不同的业务场景，是性能测试时测试场景设计的重要来源。
- 测试场景：测试场景是对业务场景的真实模拟，测试场景的设计应该尽可能贴近真实的业务场景，有时候由于测试条件的限制，可以适当做一些调整和特殊的设置等。
- 单场景：指的是只涉及单个业务流程的测试场景，目的是测试系统的单个业务处理能力是否达到预期，并且得到系统资源利用正常情况下的最大 TPS、平均响应时间等性能指标。
- 混合场景：测试场景中涉及多个业务流程，并且每个业务流程在混合的业务流程中占的比重会不同，该比重一般根据实际的业务流程来设定，尽可能符合实际的业务需要。该测试场景的目的是测试系统的混合业务处理能力是否满足预期要求，并且评估系统的混合业务处理容量最大能达到多少。

1.2　常见的性能测试指标

要衡量一个系统性能的好坏，在性能测试中通常会使用一些性能指标来进行分析和描述，以下是一些常用的性能指标。

1.2.1　响应时间

请求或者某个操作从发出的时间到收到服务器响应的时间的差值就是响应时间。在性能测试中，一般统计的是事务处理的响应时间。

图 1-2-1 是一次标准 HTTP 请求可能经过的处理路径和节点，那么响应时间的计算方式就是所有路径消耗的时间和每个服务器节点处理时间的累加，通常是网络时间+应用程序的处理时间。

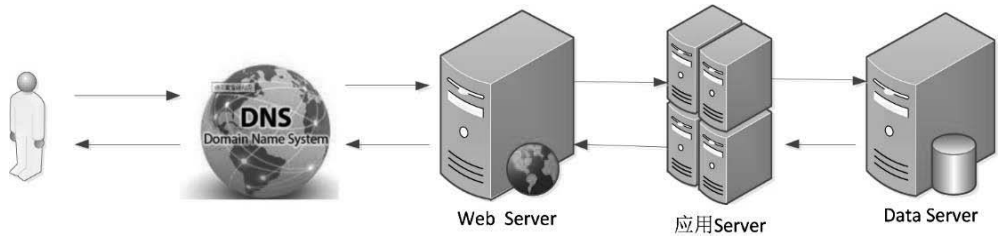

图 1-2-1

1.2.2　TPS/QPS

事务是自定义的某个操作或者一组操作的集合。例如在一个系统的登录页面，输入用户

名和密码，从单击“登录”按钮开始到登录完成跳转到页面，并且新的页面完全加载完成，这一个操作就可以定义为一个事务。

TPS 是 Transaction Per Second 的缩写，即系统每秒能够处理的交易和事务的数量，一般统计的是每秒处理的事务数。

QPS 是 Query Per Second（每秒查询率）的缩写，是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准。

1.2.3 并发用户

在真实的用户操作中，用户的每个相邻操作之间都会有一定的间隔时间（在性能测试中，我们称为用户思考时间）。因此，并发用户一般有绝对并发和相对并发之分，绝对并发是指某个时间点多个用户同时向服务器发出请求，相对并发是指一段时间多个用户向服务器发出请求。单就性能指标而言，系统的并发用户数是指系统可以同时承载的、正常使用系统功能的用户总数量。

针对并发用户，我们举例说明。在京东购物网站上购买一件商品的流程包括登录、浏览商品、把商品加入购物车、去购物车结算、确认商品清单、确认收货地址信息，最后提交订单并支付。如果 200 人同时按照这个流程购买一件商品，因为每个人购买商品的步骤有快有慢，所以在同一时间点向服务器发出请求的用户肯定不会有 200 人，会远远小于 200 人。如果我们假设为 20 人，那么上面说的 200 个用户就是相对并发用户数，而 20 个用户就是绝对并发用户数。

1.2.4 PV/UV

PV（Page View 的简写）即页面的浏览量或者点击量。用户每次对系统或者网站中任何页面的访问均会被记录一次，用户如果对同一页面进行多次访问，那么访问量会进行累加。PV 一般是衡量电子商务网站性能容量的重要指标。PV 的统计可以分为全天 PV、每个小时的 PV 以及峰值 PV（高峰 1 小时的 PV）。

UV（Unique Visitor 的简写）指系统的独立访客。访问系统网站的一台计算机客户端被称为一个访客，每天 00:00 点到次日 00:00 点内相同的客户端只能被计算一次。同样地，UV 的统计也分为全天 UV、每个小时的 UV 以及峰值 UV（高峰 1 小时的 UV）。

PV 和 UV 是衡量 Web 网站的两个非常重要的指标，PV/s 一般是由 TPS 通过一定的模型转换为 PV 的。例如，如果把每一个完整的页面都定义为一个事务，那么 TPS 就可以等同于 PV/s。PV 和 UV 之间一般存在一个比例，PV/UV 可以理解为每个用户平均浏览访问的页面数，这个比值在不同的时间点会有所波动，比如双 11 电商大促销时，PV/UV 的比重会比平时高很多。

1.2.5 点击率

每秒的页面点击数通常被称为点击率（也就是常说的 hit），该性能指标反映了客户端每秒向服务端提交的请求数。通常一个 hit 对应一次 HTTP 请求。在性能测试中，我们一般不发起静态请求（指的是对静态资源的请求，比如 JS、CSS、图片文件等），所以 hit 通常指的

是动态请求。在性能测试中，我们之所以不发起静态请求，是因为静态请求一般可以通过缓存处理，比如 CDN 等。很多静态请求一般都不需要经过应用服务器的处理，它们要么直接由 CDN 缓存处理，要么在 Web 服务器就已经被处理完成了。

1.2.6　吞吐量

吞吐量是指系统在单位时间内处理客户端请求的数量。从不同的角度看，吞吐量的计算方式可以不一样。

- 从业务角度：吞吐量可以用请求数/s、页面数/s 等来进行衡量计算。
- 从网络角度：吞吐量可以用字节/s 来进行衡量计算。
- 从应用角度：吞吐量指标反映的是服务器承受的压力，即系统的负载能力。

一个系统的吞吐量一般与一个请求处理对 CPU 的消耗、带宽的消耗、I/O 和内存资源的消耗情况等紧密相连。

1.2.7　资源开销

资源开销是每个请求或者事务对系统资源的消耗，用来衡量请求或者事务对资源的消耗程度。例如，对 CPU 的消耗可以用占用 CPU 的秒数或者核数来衡量，对内存的消耗可以用内存使用率来衡量，对 I/O 的消耗可以用每秒读写磁盘的字节数来衡量。在性能测试中，资源的开销是一个可以量化的概念。资源的开销情况对性能指标有着重要的影响，一般做性能优化时，都是尽可能让每一个请求或者事务对系统资源的消耗减少到最小。

1.3　性能测试的目标

性能测试可以发现的问题或者执行的目标描述如下：

（1）了解系统的各项性能指标，通过性能压测来了解系统能承受多大的并发访问量、系统的平均响应时间是多少、系统的 TPS 是多少等。

（2）发现系统中存在的性能问题，常见的性能问题如下：

- 系统中存在负载不均衡的情况。负载不均衡一般指的是在并发的情况下，每台服务器接收的并发压力不均匀，从而导致部分服务器因为压力过大而出现性能急剧下降，以及部分服务器因为并发过小而出现资源浪费的情况。
- 系统中存在内存泄漏问题。内存泄漏是指应用程序代码在每次执行完后，不会主动释放内存资源而导致内存使用一直增加，最终会使服务器物理内存全部耗光，程序运行逐渐变慢，并最终因为无法申请到内存而退出运行。内存泄漏多数情况下是一个渐进的过程，不易被立即发现，一般需要通过高并发性能压测才能暴露。

- 系统中存在连接泄漏问题。连接泄漏种类非常广泛，可以是数据库连接泄漏、HTTP 连接泄漏或者其他的 TCP/UDP 连接泄漏等。除了系统实际情况需要建立长连接外，一般短连接都应该用完就关闭和释放。
- 系统中存在线程安全问题。线程安全问题是在高并发访问的多线程处理系统中经常会出现的问题，如果系统中存在线程安全问题，就会出现多个线程先后更改数据的情况，导致所得到的数据全部是脏数据，有时候甚至会造成巨大的经济损失。
- 系统中存在死锁问题。死锁问题也是多线程系统中经常会遇到的一个经典问题，一般常见的有系统死锁、数据库死锁等。
- 系统中存在网络架构或者应用架构扩展性问题。扩展性问题一般是指在性能指标无法满足预期的情况下，通过横向或者纵向扩展硬件资源后，系统性能指标无法按照一定的线性规律进行快速递增。
- 系统中存在性能瓶颈。性能瓶颈一般是指因为某些因素而造成系统的性能无法持续上升。

（3）解决性能压测中存在的问题和性能瓶颈，通过不断地进行性能调优，使得系统可以满足预期的性能指标。

1.4 性能测试的基本流程

通常情况下，性能测试会经历如图 1-4-1 所示的多个阶段，这些阶段可以和很多性能测试工具对应起来，比如分析性能测试结果可以用 LoadRunner 的 Analysis 工具来实现。

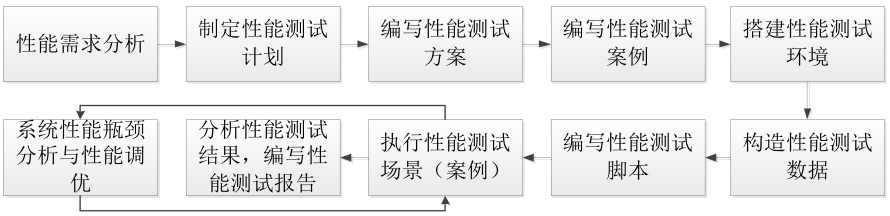

图 1-4-1

1.4.1 性能需求分析

（1）熟悉被压测系统的基本业务流程，明确此次性能测试要达到的目标，与产品经理、业务人员、架构师、技术经理一起沟通，找到业务需求的性能点。

（2）熟悉系统的应用架构、技术架构、数据架构、部署架构等，找到与其他系统的交互流程，明确系统部署的硬件配置信息、软件配置信息，把对性能测试有重要影响的关键点明确地列举出来，一般包括如下几点：

- 用户发起请求的顺序、请求之间的相互调用关系。

- 业务数据流走向、数据是如何流转的、经过了哪些应用服务、经过了哪些存储服务。
- 评估被压测系统可能存在的重点资源消耗，是 I/O 消耗型、CPU 消耗型，还是内存消耗型，这样在压测执行时可以重点进行监控。
- 关注应用的部署架构。如果是集群部署，压测时需要关注应用的负载均衡转发是否均匀，每台应用服务器资源消耗是否大体一致。
- 和技术经理一起沟通，明确应用的并发架构是采用多线程处理还是多进程处理，重点需要关注是否会死锁、数据是否存在不一致、线程同步锁是否合理（锁的粒度一般不宜过大，过大时可能会影响并发线程的处理）等。

（3）明确系统上线后可能会达到的最大并发用户数、用户期望的平均响应时间以及峰值时的业务吞吐量，并将这些信息转换为性能需求指标。

1.4.2　制定性能测试计划

性能测试计划是性能测试的指导，是一系列测试活动的依据，在制定性能测试计划时，需要明确系统的上线时间点、当前项目的进度以及所处的阶段、可以供调配的硬件资源和性能测试人员。一个完整的性能测试计划一般包括如下几个部分：

（1）性能测试计划编写的目的：主要是作为整个性能测试过程的指导，让性能测试环境搭建、测试策略的选取、任务与进度事项跟踪、性能测试风险分析等事项有序地进行；同时也需要明确此次性能测试预期需要达到的标准，以及明确性能测试完成而退出测试所需的条件。

（2）明确各个阶段的具体执行时间点以及对应的责任人：

- 预计由谁何时开始性能需求分析，何时结束性能需求分析。
- 预计由谁何时开始性能测试方案的编写，何时结束性能测试方案的编写。
- 预计由谁何时开始性能测试案例的编写，何时结束性能测试案例的编写。
- 预计由谁何时开始搭建性能测试环境，何时结束性能测试环境的搭建。
- 预计由谁何时开始准备性能测试需要的数据，何时准备完毕。
- 预计由谁何时开始编写性能测试脚本，何时编写完毕。性能测试脚本的编写一般包含如下步骤：
  - 按照性能测试场景，开始录制性能测试脚本或者直接编写性能测试脚本，此时可能用到的常见性能测试工具包括 LoadRunner、BadBoy、JMeter、nGrinder 等。
  - 根据准备好的测试数据，对性能测试脚本进行参数化，添加集合点、事务分析点等。
  - 对性能脚本进行试运行调试，确保不出现报错，并且可以覆盖测试场景中的所有操作。
- 预计由谁何时开始性能测试的执行，何时完成性能测试的执行，此阶段一般需要完成如下事项：

- 完成每一个性能测试场景和案例的执行，记录相关的性能测试结果，明确性能曲线的变化趋势，获取性能的拐点等。
- 根据性能测试的结果，评估性能数据是否可以满足预期，从性能测试结果数据中分析存在的性能问题。
- 针对性能问题，进行性能定位和优化，然后进行二次压测，直至性能数据可以满足预期，性能测试问题得到解决。
- 完成性能测试分析报告的编写。

（3）性能测试风险的分析和控制：评估可能存在的风险和不可控的因素，以及这些风险和因素对性能测试可能产生的影响，针对这些风险因素需要给出对应的短期和长期的解决方案。性能测试风险一般包括如下几点：

- 性能测试环境因素：无法预期完成性能测试环境的搭建，这中间的原因可能是硬件引起也可能是软件引起，硬件原因一般可能包括性能压测服务器无法按时到位、服务器硬件配置无法满足预期（一般要求性能压测服务器硬件配置等同于生产环境，服务器的节点数可以少于生产环境，但是需要保证每个应用服务至少部署了两台节点服务器）。软件原因可能包括性能测试环境软件配置无法和生产环境保持一致（一般要求性能压测环境软件配置，比如软件版本、数据库版本、驱动版本等要和生产环境完全保持一致）。
- 性能测试人员因素：性能测试人员无法按时到位参与项目的性能测试，如果出现这样的风险，肯定会导致性能测试无法按预期进行，需要立即向项目经理进行汇报，以确保可以协调到合适的人员，因为这是一个非常严重的风险。
- 性能测试结果无法达到预期：即系统的性能无法达到生产预期上线要求或者存在的性能问题无法解决。性能调优其实本身就是一个长期的不断优化的过程，此时可以看是否能够通过服务器的横向或者纵向扩容来解决，如果还是无法解决，那么也需要提前上报风险。

1.4.3 编写性能测试方案

在有了性能测试计划后，我们就需要按照性能需求分析的结果来制定性能测试方案，即按照什么样的思路和策略去测试、需要设计哪些测试场景，以及测试场景执行的先后顺序、每个测试场景需要重点关注的性能点等，一般包括如下几个部分：

（1）测试场景的设计：

- 单场景设计：单一业务流程的处理模式设计。
- 混合场景设计：多个业务流程同时混合处理模式的设计。

（2）定义事务。测试方案中需要明确定义好压测事务，方便分析响应时间（特别是在混合场景中，事务的定义可以方便分析每一个场景响应时间的消耗）。比如我们对在淘宝网购买商品这一场景进行压测，可以把下订单定义为一个事务，把支付也定义为一个事务，在压

测结果中，如果响应时间较长，就可以对每一个事务进行分析，看哪个事务耗时最长。

（3）明确监控对象。针对每个场景，明确可能的性能瓶颈点，比如数据库查询、Web 服务器服务转发、应用服务器等；需要监控的对象，比如 TPS、平均响应时间、点击率、并发连接数、CPU、内存、IO；等等。

（4）定义测试策略：

- 明确性能测试的类型：需要进行哪些类型的性能测试，比如负载测试、压力测试、稳定性测试等。
- 明确性能测试场景的执行顺序，一般是先执行单场景测试，后执行混合场景测试。
- 如果是进行压力测试，还需要明确加压的方式，比如按照开始前 5 分钟、20 个用户，然后每隔 5 分钟增加 20 个用户的方式来进行加压。

（5）性能测试工具的选取。性能测试工具有很多，常见的有 LoadRunner、JMeter、nGrinder 等，那么如何来选取合适的性能测试工具呢？

- 一般性能测试工具都是基于网络协议开发的，所以我们需要明确待压测系统使用的协议，尽可能和被压测系统的协议保持一致，或者至少要支持被压测系统的协议。
- 理解每种工具实现的原理，比如哪些工具适用于同步请求的压测，哪些工具适用于异步请求的压测。
- 压测时明确连接的类型，比如属于长连接还是短连接、一般连接多久能释放。
- 明确性能测试工具并发加压的方式，比如是多线程加压还是多进程加压，一般采用的都是多线程加压。

（6）明确硬件配置和软件配置。

硬件配置一般包括服务器的 CPU 配置、内存配置、硬盘存储配置、集群环境下还要包括集群节点的数量配置等。

软件配置一般包括：

- 操作系统配置：操作系统的版本以及参数配置需要同线上保持一致。
- 应用版本配置：应用版本要和线上保持一致，特别是中间件、数据库组件等的版本，因为不同版本，其性能可能不一样。
- 参数配置：比如 Web 中间件服务器的负载均衡、反向代理参数配置、数据库服务器参数配置等。

（7）网络配置。一般为了排除网络瓶颈，除非有特殊要求，通常建议在局域网下进行性能测试，并且要明确压测服务器的网卡类型以及网络交换机的类型，比如网卡是否为千兆网卡，交换机属于百兆交换机还是千兆交换机等，这对我们以后分析性能瓶颈会有很大的帮助。在网络吞吐量较大的待压测系统中，网络有时候也很容易成为一个性能瓶颈。

1.4.4　编写性能测试案例

性能测试案例一般是对性能测试方案中性能压测场景的进一步细化，一般包括如下几点：

- 预置条件：一般指执行此性能测试案例需要满足的条件。比如性能测试数据需要准备到位、性能测试环境需要启动成功等。
- 执行步骤：详细描述案例执行的步骤，一般需要描述测试脚本的录制和编写、脚本的调试、脚本的执行过程（比如如何加压、每个加压的过程持续多久等）、需要观察和记录的性能指标、需要明确性能曲线的走势、需要监控哪些性能指标等。
- 性能预期结果：描述性能测试预期需要达到的结果，比如 TPS 需要达到多少、平均响应时间需要控制到多少以内、服务器资源的消耗需要控制在多少以内等。

1.4.5 搭建性能测试环境

性能测试环境搭建需要注意如下几点：

- 需要尽可能与实际生产环境的配置保持一致，不可减少其中的相关组件，实际生产环境中有的组件在性能测试环境中都必须部署。
- 一般生产环境的服务器数量和配置都很高，但是性能测试环境又不可能以这么高的成本去部署完全一样的硬件机器。性能测试环境上的机器数量和机器配置可以按照一定比例进行缩减，但是如果是分布式系统，那么服务的机器节点至少需要两个或者两个以上，并且节点资源配置不能太低。
- 操作系统版本以及在操作系统上部署的相关软件（比如中间件软件、应用容器软件等）版本，数据库软件版本必须与实际生产环境完全一致。
- 网络环境必须与实际生产环境保持一致，因为网络是一个容易成为性能瓶颈的地方。

1.4.6 构造性能测试数据

性能测试的结果是否准确，在一定程度上还取决于测试数据的数量和质量。

- 如果不是直接使用实际的生产环境进行线上的性能测试（比如淘宝、天猫、苏宁易购等网站可以在双 11 大促前，允许在特定的时间段进行生产环境的压力测试），那么在性能测试环境中，就需要提前导入能够模拟生产环境的基础数据，比如用户数据、角色权限数据等，并且基础数据需要有足够的量，或者尽可能和生产环境一致，或者只能略低于生产环境。
- 测试场景和测试案例中需要的数据，这块的数据可以通过从生产环境中拉取数据脱敏后导入，或者使用数据脚本进行批量构造。数据量需要和生产环境的数据库中的数据量尽可能在一个数量级上，或者根据性能测试环境硬件配置和生产环境硬件配置的比例差别来评估测试数据和生产数据的量级比例为多少是合适的。
- 数据的质量也非常重要，比如一个查询操作的性能测试场景，如果构造的数据都是查询参数无法查询到的数据，那这个性能测试的结果肯定不会准确。性能测试的数据一定是与性能测试案例中查询参数可以相互匹配的。

1.5 性能分析调优模型

性能测试除了为获取性能指标外，更多是为了发现性能问题和性能瓶颈，然后针对性能问题和性能瓶颈进行分析和调优。在当今互联网高速发展的时代，结合传统软件系统模型以及互联网网站特征，性能调优的模型可以归纳总结为如图 1-5-1 所示的知识框架。

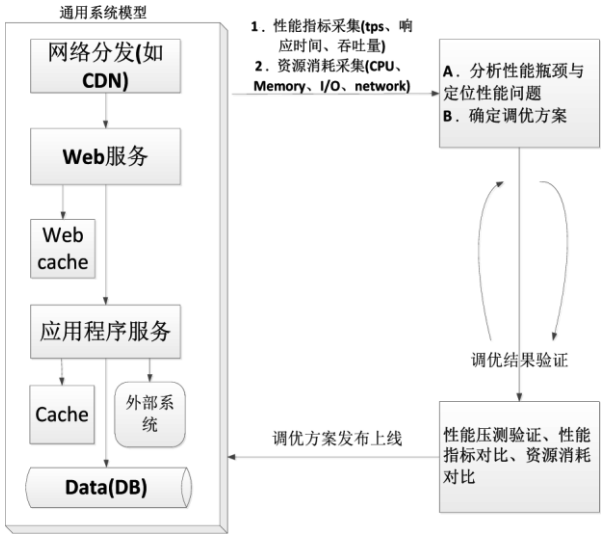

图 1-5-1

系统模型中相关的组件说明如下：

（1）网络分发：网络分发是高速发展的互联网时代常用的降低网络拥塞、快速响应用户请求的一种技术手段，最常用的网络分发就是 CDN（Content Delivery Network，即内容分发网络），它依靠部署在世界各地的边缘服务器，通过中心平台的负载均衡、源服务器内容分发、调度等功能模块，使世界各地用户就近获取所需内容，而不用每次都到中心平台的源服务器获取响应结果。比如，南京的用户直接访问部署在南京的边缘服务器，而不需要访问部署在遥远的北京的服务器。

（2）Web 服务器：Web 服务器用于部署 Web 服务。Web 服务器负责请求的响应和分发以及静态资源的处理。

（3）Web 服务：Web 服务指运行在 Web 服务器上的服务程序。最常见的 Web 服务就是 Nginx 和 Apache。

（4）Web Cache：Web Cache 指 Web 层的缓存。一般用来临时缓存 HTML、CSS、图像等静态资源文件。

（5）应用服务器：应用服务器用于部署应用程序，如 Tomcat、WildFly、普通的 Java 应用程序（如 jar 包服务）、IIS 等。

（6）应用程序服务：应用程序服务指运行在应用服务器上的程序，比如 Java 应用、C/C++应用、Python 应用。一般用于处理用户的动态请求。

（7）应用缓存：应用缓存指应用程序层的缓存服务，常用的应用缓存技术有 Redis、Memcached 等，这些技术手段也是动态扩展的高并发分布式应用架构中经常使用的技术手段。

（8）数据库（DB）：用于数据的存储，可以包括关系数据库以及 NoSQL 数据库（非关系数据库）。常见的关系数据库有 MySQL、Oracle、SQL Server、DB2 等，常见的 NoSQL 数据库有 HBase、MongoDB、ElasticSearch 等。

（9）外部系统：指当前系统依赖于其他的外部系统，需要从其他外部系统中通过二次请求获取数据，外部系统有时候可能会存在很多个。

图 1-5-1 中所示的系统模型，是一个互联网中常见的用户请求的分层转发和处理的过程。这个性能调优就是不断采集系统中的性能指标，以及系统模型中各层的资源消耗，从中发现性能瓶颈和性能问题，然后对瓶颈和问题进行分析诊断来确定性能调优方案，最后通过性能压测来验证调优方案是否有效；如果无效，则继续重复这个过程进行性能分析，直到调优方案有效，瓶颈和问题得到解决。这个过程一般非常漫长，因为很多时候性能调优方案往往不是一次就能有效，或者一次就能解决所有的瓶颈和问题，或者一次就解决了当前的瓶颈和问题，但是继续执行性能压测又可能会出现新的瓶颈和问题。

1.6 性能分析调优思想

1.6.1 分层分析

分层分析指的是按照系统模型、系统架构以及调用链分层进行监控分析和问题排查。分层分析具有如下特点：

- 分层分析一般需要对系统的应用架构以及部署架构的层次非常熟悉，需要熟知请求的处理链过程。
- 分层分析一般需要对每一层建立 checklist（检查清单），然后按照每一层的 checklist 逐一进行分析。
- 通过分层分析来排查问题的效率虽然较低，但是往往能发现更多的性能问题。
- 分层分析可以自上而下，也可以自下而上。

对于一个应用系统，常见的分层分析思路如图 1-6-1 所示。分层分析自上而下说明如下：

- 网络分发。如 CDN，可以降低网络拥塞。
- Web 服务。除了其自身外，也包括 Web Cache。
- 应用程序服务。除了其自身外，也包括应用层 Cache 和同外部系统的交互。
- Data（DB）。即数据或数据库。

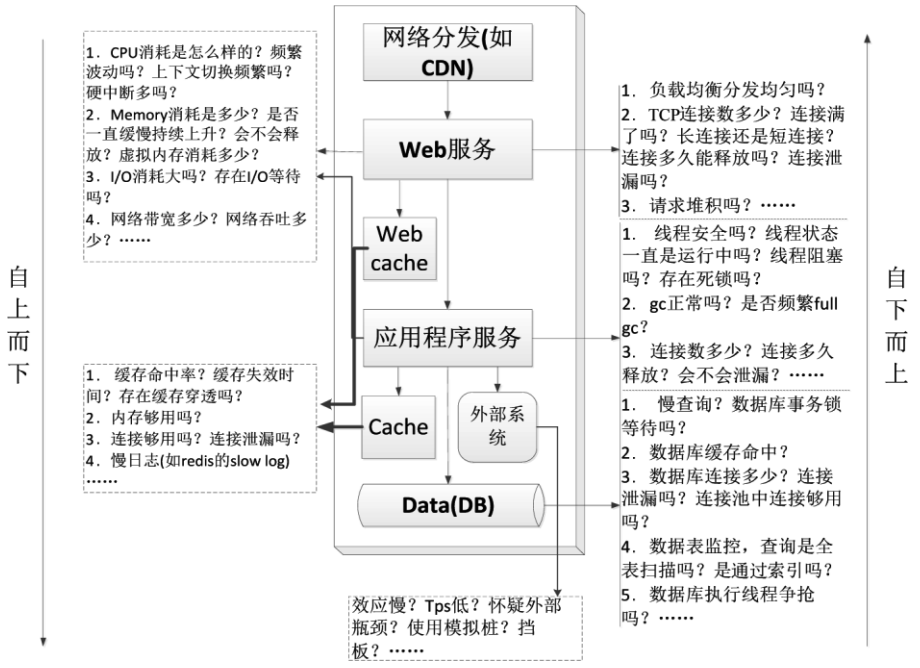

图 1-6-1

1.6.2　科学论证

科学论证是指通过一定的假设和逻辑思维推理来分析性能问题，一般包括发现问题、问题假设、预测、试验论证、分析这 5 个步骤，如图 1-6-2 所示。

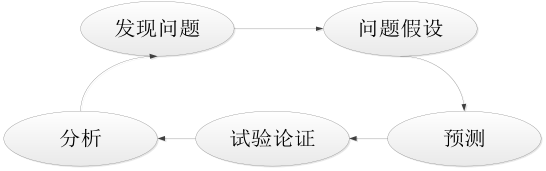

图 1-6-2

（1）发现问题：指通过性能采集和监控，发现性能瓶颈或者性能问题，比如并发用户数增大后 TPS 并不增加、每台应用服务器的 CPU 消耗相差特别大等。

（2）问题假设：指根据自己的经验判断，假设是某个因素导致出现了瓶颈和问题。

（3）预测：指根据问题假设，预测可能出现的一些现象或者特征。

（4）试验论证：根据预测，通过试验去检查预期可能出现的现象或者特征。

（5）分析：根据获取到的实际现象或者特征进行分析，判断假设是否正确，如果不正确，就重新按照这个流程进行分析论证。

科学论证法进行性能分析与调优的示例如图 1-6-3 所示。

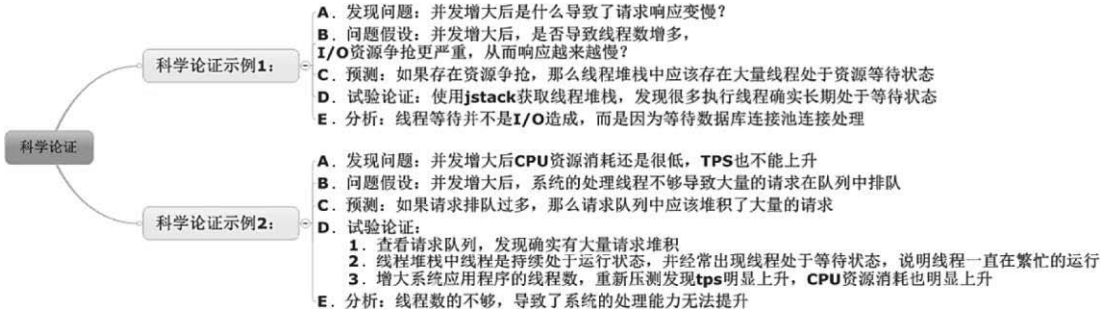

图 1-6-3

1.6.3 问题追溯与归纳总结

问题追溯指的是根据问题去追溯最近系统或者环境发生的变化，一般适用于已上线生产系统的版本发布或者环境变动导致的性能问题。问题追溯是通过不断向下追溯问题和根据问题描述去逐步排查可能导致问题的原因。问题追溯的步骤一般如下：

（1）是怎么觉得运行的系统或者软件有性能问题的？

（2）是用户反馈的吗？感觉系统或者软件很卡顿？

（3）问题能用具体的性能指标描述吗？打开网页响应时间很慢？

（4）问题影响其他的系统吗？是不是独立系统？

（5）最近有什么变动吗？代码有变动吗？数据库表有变动吗（有新创建的表吗？这些表有索引吗？）？服务器硬件有变动吗？网络有变动吗？配置有变动吗？

（6）是下游别的系统造成的影响吗？下游别的系统最近有发布吗？

归纳总结指在出现某种性能瓶颈或者性能问题时，根据以往的经验从总结的原因中进行逐一排查。

1.7 性能调优技术

1.7.1 缓存调优

在这个互联网高速发展的时代，为了缩短用户访问请求的响应时间，缓存的使用已经成为很多大型系统或者电商网站使用的一个关键技术。缓存的合理设计至关重要，它直接关系到一个系统或者网站的并发访问能力和用户体验。

1. 缓存分类

缓存按照存放地点的不同，可以分为用户端缓存和服务端缓存，如图 1-7-1 所示。

图 1-7-1

（1）用户端缓存：一般指的是个人计算机（PC）的浏览器缓存以及移动端手机 APP 的本地缓存等。例如，一些静态页面，除非重新部署否则平时一般不发生变更，那这种就可以设计到缓存中，而且一般静态页面如果每次访问的时候都需要去加载，那么在用户网络状况不佳时，会造成用户页面加载很慢，用户体验非常差。因此，所以我们一般看到的手机 APP 基本上都使用了大量的用户数据缓存。

（2）服务端缓存：可以包括 Web 中间件（比如 Nginx 或者 Apache）的缓存、应用数据的缓存（比如 Redis 或者 Memcached 等内存数据库）。Web 中间件的缓存主要用于存储一些前端的静态资源文件。应用数据缓存是指为了提高查询效率，避免每次查询相同的数据都需要去查询数据库，从而将数据缓存在内存数据库中。

应用数据缓存和数据库之间的交互一般如图 1-7-2 所示。从图中可以看到，应用程序服务器先查询缓存，如果缓存有数据，则直接返回；如果缓存中没有数据，则直接查询数据库，并将数据写入缓存中，以便下次直接访问缓存。

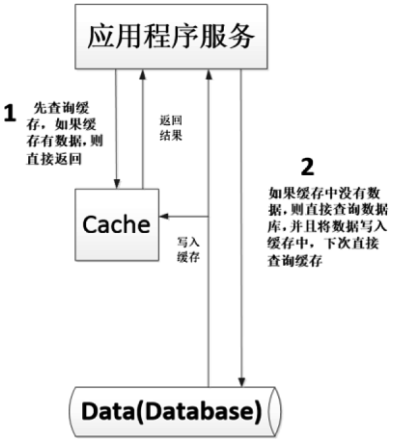

图 1-7-2

注意，Redis 是由 C 语言实现的一个内存数据库，相关的知识可以参考 Redis 官方网站：https://redis.io/。Memcached 也是由 C 语言实现的一个内存数据库，相关的知识可以参考 Memcached 官方网站：http://memcached.org/。

2. 缓存设计和调优的关键点

（1）如何让缓存的命中率更高？

- 缓存适合“读多写少”的业务场景。
- 根据实际业务设计最合适的缓存粒度，粒度过细则内存资源的成本会过高，粒度过粗则又会影响命中率。
- 设计最合适业务场景的缓存更新策略，如果缓存更新过慢，那势必会影响缓存的命中率，从而加大数据库的查询压力。

（2）如何防止缓存穿透？

缓存穿透是指用户查询时，总是存在大量的查询直接绕过缓存数据库而直接去查询底层数据库。要防止缓存穿透，可以参考以下几点：

- 数据空值缓存：将数据库中查询结果为空的缓存键也存储到缓存中，避免空值时重复查询数据库。
- 对于不存在的数据，也设置缓存：建议可以设置一个较短的过期时间，从而减轻数据库的查询压力，因为数据是否存在只有查询了数据库才能确定，如果提前就在缓存中设置了，那查询不存在的数据时，就可以在缓存层过滤掉。
- 避免大量的缓存数据同时失效或过期：如果大量的缓存数据同时失效或过期，就会导致大量请求直接查询数据库。

（3）如何控制好缓存的失效时间和失效策略？

由于缓存数据库一般都是将数据存储在内存中，但是内存一般都是有大小限制的，因此现实中需要根据实际的业务场景，选择最适合的缓存失效策略。常见的缓存失效策略说明如下：

- FIFO（先进先出）：在数据超出缓存数据库的最大容量时，优先清理最早进入缓存的数据。
- LIFO（后进后出）：与 FIFO 刚好相反，这个策略会优先清理最后进入缓存的数据，这种策略一般用得很少，只有存在特殊的业务场景才使用。
- LRU（最久不被使用）：在数据超出缓存数据库的最大容量时，优先清理最久时间未被访问的数据。
- MRU（最近使用）：与 LRU 刚好相反，MRU 会清理最近被使用的数据。
- LFU（最不常用）：在数据超出缓存数据库的最大容量时，优先清理总访问次数最少的数据。

（4）如何做好缓存的监控分析？

可以通过慢日志（Slow Log）分析、连接数监控、内存使用监控等多种方式，做好缓存的监控分析。比如，通过慢日志可以看到在查询哪条数据时，查询时间最久；通过监控内存的使用，可以看到哪些数据经常被访问，哪些数据不经常被访问，以及哪些数据正在从缓存中失效而被移出缓存。

（5）如何防止缓存雪崩？

缓存雪崩指的是服务器在出现断电等极端异常情况后，缓存中的数据全部丢失，导致大量的请求需要从数据库中直接获取数据，使数据库压力过大从而造成数据库崩溃。防止缓存雪崩需要

注意：

- 要处理好缓存数据全部丢失后，如何能快速地把数据重新加载到缓存中。
- 缓存数据的分布式冗余备份，当出现数据丢失时，可以迅速切换使用备份数据。
- 设置多级缓存机制，从而避免某级缓存雪崩后，所有的请求直接去查询数据库。

1.7.2　同步转异步推送

同步指的是系统（请求处理方）收到请求调用方的一个请求（Request）后，在该请求没有处理完成时，就一直不返回响应（Response）结果，直到处理完成后才返回响应结果，如图 1-7-3 所示。

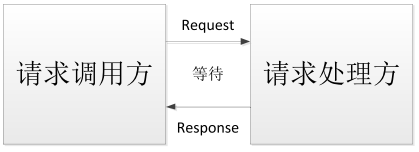

图 1-7-3

与同步相比，异步指的是系统收到一个请求后，立即把请求接收成功返回给请求调用方，在请求处理完成后，再异步推送处理结果给调用方，或者请求调用方间隔一定时间之后再重新获取请求结果，如图 1-7-4 所示。

图 1-7-4

同步转异步主要解决同步请求时的阻塞等待问题。一直处于阻塞等待的请求，往往会造成连接不能快速释放，从而导致在高并发处理时连接数不够用。通过队列异步接收请求后，请求处理方再进行分布式的并行处理，可以扩展处理能力，并且网络连接也可以快速释放。

1.7.3　削峰填谷

在同步转异步推送处理中，最常用到的就是消息队列，如图 1-7-5 所示。消息队列是一种临时存储消息和请求的存储介质，可以是内存，也可以是磁盘，主要用于系统或者模块间的消息传递。很多编程语言内部都有消息队列的实现，当然也有专门用于消息队列的技术组件，比如 Kafka、RabbitMQ 等。

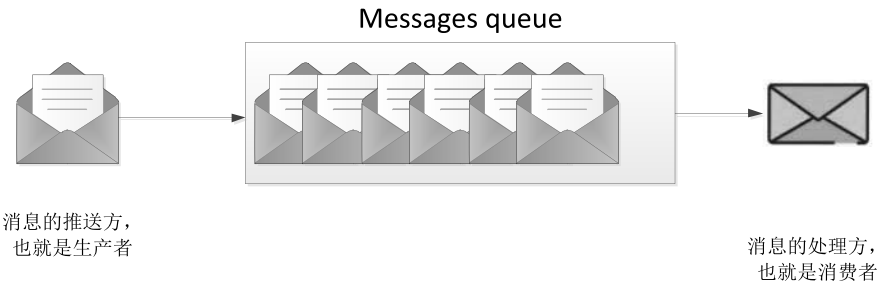

图 1-7-5

消息队列是在软件架构设计中用于系统或者模块间解耦的一种常见设计模式，在系统和模块之间建立了一个数据通道，实现了系统或者模块之间的解耦。通过削峰填谷和异步处理，避免了系统资源耗尽，尤其在高并发调用系统的情况下，系统可能会面临突然的大量请求，而消息队列可以缓冲这些请求，将它们临时存储在消息队列中，按照系统的处理能力按顺序逐步消费数据消息，平滑过渡高峰期的请求，避免系统超压崩溃或性能急剧下降。

我们以双 11 大促秒杀活动作为示例来说明，消息队列用来削峰填谷，对优化系统性能压力起到关键的作用，如图 1-7-6 所示。

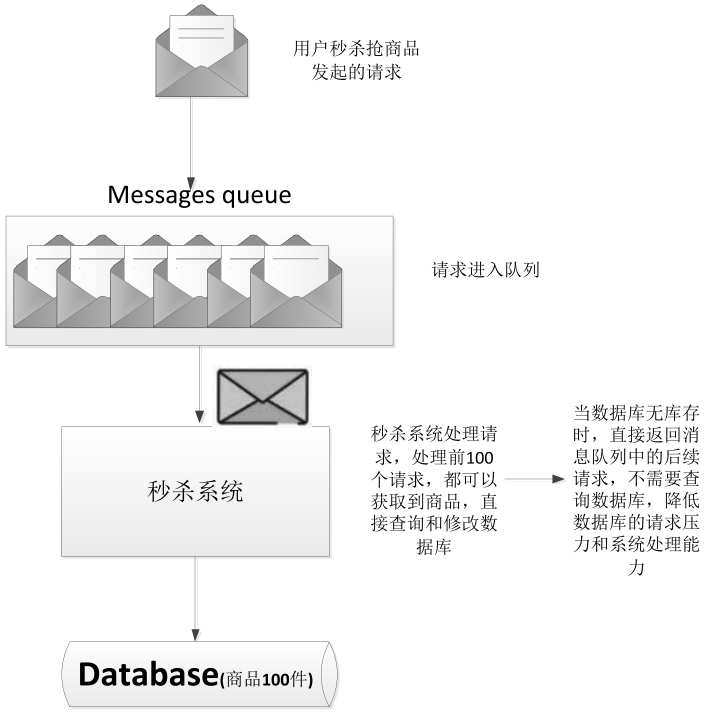

图 1-7-6

可以看到有了消息队列后：

- 很好地降低了数据库的压力，因为只有前 100 个请求需要查询和修改数据库的库存，当库存为空时，不需要做任何处理，直接返回用户。

- 秒杀系统可以根据自身的消费能力，合理处理数据。当库存为空时，后续收到的消息队列中的请求，可以直接返回，不需要做任何的逻辑处理，从而减少了系统资源的使用，提升了系统的并发处理能力。

1.7.4　拆分

拆分指的是将系统中的复杂业务调用拆分为多个简单的调用，如图 1-7-7 所示。拆分一般遵循的原则如下：

- 对于高并发的业务，请求调用单独拆分为单个的子系统应用。
- 对于并发访问量接近的业务，可以按照产品业务进行拆分，相同的产品业务都归类到一个新的子系统中。

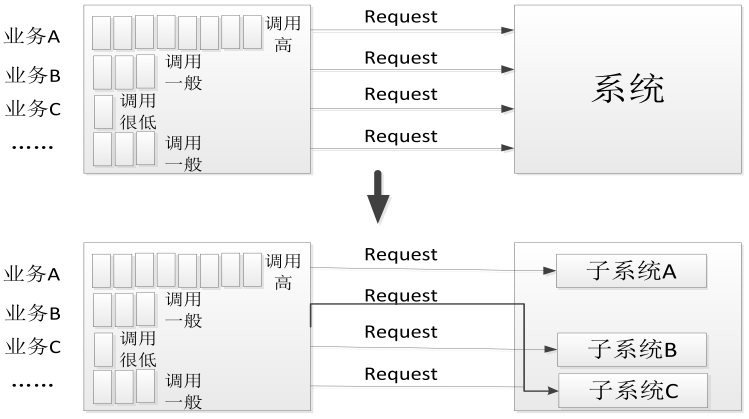

图 1-7-7

系统拆分带来的好处就是高并发的业务不会对低并发业务的性能造成影响，而且系统在硬件扩展时，也可以有针对性地进行扩展，避免资源的浪费。

1.7.5　任务分解与并行计算

任务分解与并行计算指的是将一个任务拆分为多个子任务，然后将多个子任务并行进行计算处理，最后只需要将并行计算的结果合并在一起返回即可，如图 1-7-8 所示。这样处理的目的是通过并行计算的方式来增加处理性能。

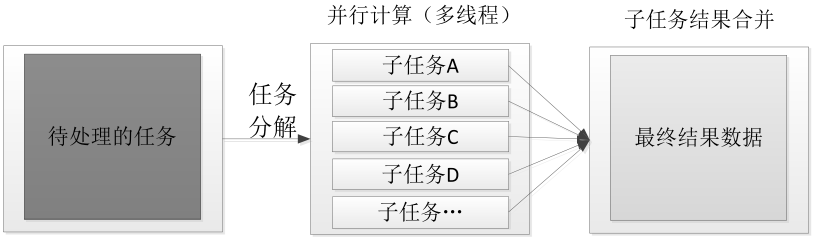

图 1-7-8

另外，对于包含多个处理步骤的串行任务，也可以尽量按照如图 1-7-9 所示的方式转换为并行计算处理。

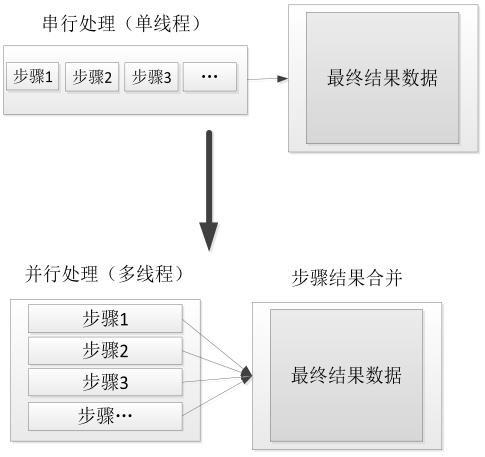

图 1-7-9

1.7.6 索引与分库分表

索引指应用程序在查询时，尽量使用数据库索引进行查询。数据库表在创建时也尽量对查询条件的字段建立合适的索引。这里强调一定是合适的索引，如果索引建立不合适，不仅对查询效率没有任何的帮助，反而会使数据库表在插入数据时变得更慢，因为一旦建立了索引，数据在插入时，索引也会自动更新，这样就加大了数据库数据插入时的资源消耗。例如，数据库表中有一个字段为 status，而 status 的取值只有 0、1、2 三个值，这时候如果对 status 建立索引，对查询效率没有任何帮助，因为 status 字段的值只有 0、1、2 这三个值，取值范围太少，建立索引后根据 status 去检索时，需要扫描的数据量还是非常大的。

正确使用索引可以很好地提高查询效率，但是如果一个表的数据量非常庞大，比如达到了亿万级别，此时索引查询很慢，并且新数据插入时也很慢，这就需要对数据进行分表或者分库。分库一般指的是一个数据库的存储量已经很大了，查询和插入时 I/O 消耗非常大，此时就需要将数据库拆分成 2 个库，以减轻读写时 I/O 的压力，如图 1-7-10 所示。

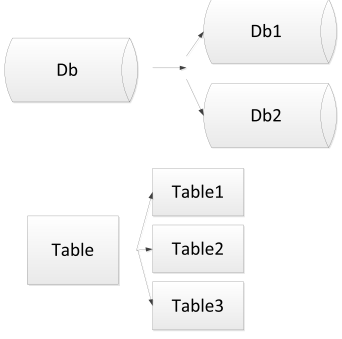

图 1-7-10

常见的分库分表方式如下：

- 按照冷热数据分离的方式：一般将查询频率较高的数据称为热数据，查询频率较低或者几乎不被查询的数据称为冷数据。冷热数据分离后，热数据单独存储，这样数据量就会下降，查询的性能自然也就提升了，而且还可以更方便地单独针对热数据进行 I/O 性能调优。
- 按照时间维度的方式：比如可以按照实时数据和历史数据分库分表，也可以按照年份、月份等时间区间进行分库分表，目的是尽可能地减少每个库表中的数据量。
- 按照一定的算法计算的方式：此种方式一般适用于数据都是热数据的情况，比如数据无法做冷热分离，所有的数据都经常被查询到，而且数据量又非常大，此时就可以根据数据中的某个字段执行算法（注意：这个字段一般是数据查询时的检索条件字段），使得数据插入后能均匀地落到不同的分表中去（由算法决定每条数据进入哪个分表），查询时再根据查询条件字段执行同样的算法，就可以快速定位到是需要到哪个分表中去进行数据查询。

在分库分表后，写入数据时就可以按照如图 1-7-11 所示的方式进行分布式写入。数据分库分表带来的另一个好处就是：如果在单次查询时需要查询多个分表，那么就可以通过多线程并行的方式去查询每个分表，最后合并每个分表的查询结果即可，这样也可以使得查询的效率更高，如图 1-7-12 所示。

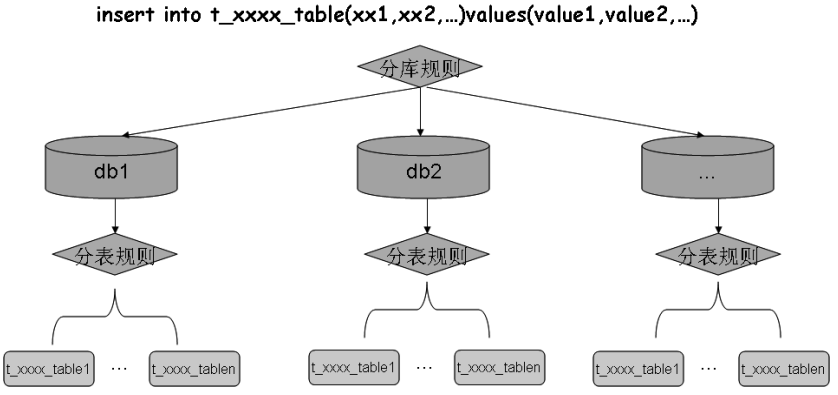

图 1-7-11

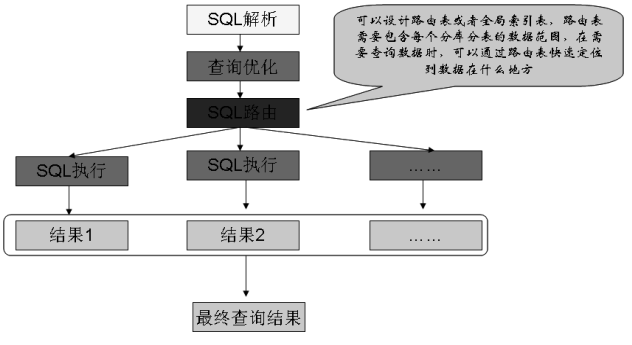

图 1-7-12

数据分库分表的关键点如下：

- 合理设计路由表，当需要查询数据时，能快速地定位到需要查询的数据是分布在哪些库的哪些表中，这样就能快速到对应的库和表中去做查询。
- 设计分库时，根据实际业务场景，尽量减少跨库之间的关联查询，因为不同的库的数据一般分布在不同的存储上，如果做关联查询，那么就会涉及大量数据的网络传输，从而影响查询性能。
- 设计分表时，尽可能根据实际业务场景让每个分表的数据分布相对比较均匀。

1.7.7 层层过滤

层层过滤是指从系统的前端到后端处理的过程中，层层拦截不合理的请求，尽量将请求拦截在上游，降低下游的压力，从而减少系统和底层数据库的并发处理压力，达到性能提升的目的，如图 1-7-13 所示。

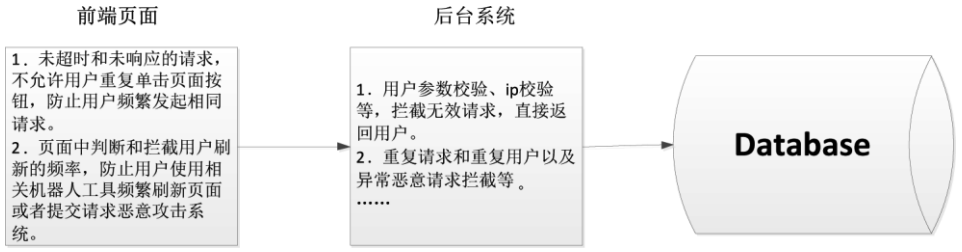

图 1-7-13

层层过滤的关键点如下：

- 在不同的层级尽可能过滤掉属于该层级的应当被过滤的无效请求，让最末端进入数据库中的请求都是有效的请求。
- 错误前置，提前抛出异常。对于异常的请求，越早抛出异常，越有利于减轻系统的负载和节省资源的占用。
- 避免重复请求以及通过机器人的恶意请求，从而降低系统的处理压力，更好地保护系统。