SpringBoot 概述

什么是 Spring Boot?

Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。

Spring Boot、Spring MVC 和 Spring 有什么区别?

SpringFrame

SpringFramework 最重要的特征是依赖注入。所有 SpringModules 不是依赖注入就是 IOC 控制反转。

当我们恰当的使用 DI 或者是 IOC 的时候，我们可以开发松耦合应用。松耦合应用的单元测试可以很容易的进行。

SpringMVC

Spring MVC 提供了一种分离式的方法来开发 Web 应用。通过运用像 DispatcherServelet，MoudlAndView 和 ViewResolver 等一些简单的概念，开发 Web 应用将会变的非常简单。

SpringBoot

Spring 和 SpringMVC 的问题在于：需要配置大量的参数。
Spring Boot 通过一个自动配置和启动的项来目解决这个问题。为了更快的构建产品就绪应用程序，Spring Boot 提供了一些非功能性特征。

Spring Boot 有哪些优点?

Spring Boot 主要有如下优点：

容易上手，提升开发效率，为 Spring 开发提供一个更快、更广泛的入门体验。
开箱即用，远离繁琐的配置。
提供了一系列大型项目通用的非业务性功能，例如：内嵌服务器、安全管理、运行数据监控、运行状况检查和外部化配置等。
没有代码生成，也不需要XML配置。
避免大量的 Maven 导入和各种版本冲突。

为什么要用SpringBoot?

为了解决java开发中的，繁多的配置、底下的开发效率，复杂的部署流程，和第三方技术集成难度大的问题，产生了spring boot。

springboot 使用 “习惯优于配置”的理念让项目快速运行起来，使用springboot很容易创建一个独立运行的jar，内嵌servlet容器

springboot的核心功能一：独立运行spring项目，springboot可以以jar包的形式独立运行，运行一个springboot项目只需要 java -jar xxx.jar 来运行

springboot的核心功能二：内嵌servlet容器，可以内嵌tomcat，接天jetty，或者undertow，这样我们就可以不用war包形式部署项目

springboot的核心功能三，提供starter简化maven配置，spring提供了一系列starter pom 来简化maven的依赖加载， 当使用了 spring-boot-starter-web时，会自动加载所需要的依赖包

springboot的核心功能三：自动配置spring sprintboot 会根据在类路径的jar包，类，为jar包中的类自动配置bean，这样会极大的减少使用的配置，会根据启动类所在的目录，自动配置bean

什么是 Spring Boot Stater ?

启动器是一套方便的依赖没描述符，它可以放在自己的程序中。你可以一站式的获取你所需要的 Spring 和相关技术，而不需要依赖描述符的通过示例代码搜索和复制黏贴的负载。

例如，如果你想使用 Sping 和 JPA 访问数据库，只需要你的项目包含 spring-boot-starter-data-jpa 依赖项，你就可以完美进行。

Spring Boot 还提供了其它的哪些 Starter 选项?

Spring Boot 也提供了其它的启动器项目包括，包括用于开发特定类型应用程序的典型依赖项。

spring-boot-starter-web-services - SOAP Web Services

spring-boot-starter-web - Web 和 RESTful 应用程序

spring-boot-starter-test - 单元测试和集成测试

spring-boot-starter-jdbc - 传统的 JDBC

spring-boot-starter-hateoas - 为服务添加 HATEOAS 功能

spring-boot-starter-security - 使用 SpringSecurity 进行身份验证和授权

spring-boot-starter-data-jpa - 带有 Hibeernate 的 Spring Data JPA

spring-boot-starter-data-rest - 使用 Spring Data REST 公布简单的 REST 服务

Spring 是如何快速创建产品就绪应用程序的?

Spring Boot 致力于快速产品就绪应用程序。为此，它提供了一些譬如高速缓存，日志记录，监控和嵌入式服务器等开箱即用的非功能性特征。

spring-boot-starter-actuator - 使用一些如监控和跟踪应用的高级功能

spring-boot-starter-undertow, spring-boot-starter-jetty, spring-boot-starter-tomcat - 选择您的特定嵌入式 Servlet 容器

spring-boot-starter-logging - 使用 logback 进行日志记录

spring-boot-starter-cache - 启用 Spring Framework 的缓存支持

Spring2 和 Spring5 所需要的最低 Java 版本是什么?

Spring Boot 2.0 需要 Java8 或者更新的版本。Java6 和 Java7 已经不再支持。

Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的?

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：

SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。

EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。

ComponentScan：Spring组件扫描。

自动配置

什么是 JavaConfig?

Spring JavaConfig 是 Spring 社区的产品，它提供了配置 Spring IoC 容器的纯Java 方法。因此它有助于避免使用 XML 配置。使用 JavaConfig 的优点在于：

（1）面向对象的配置。由于配置被定义为 JavaConfig 中的类，因此用户可以充分利用 Java 中的面向对象功能。一个配置类可以继承另一个，重写它的@Bean 方法等。

（2）减少或消除 XML 配置。基于依赖注入原则的外化配置的好处已被证明。但是，许多开发人员不希望在 XML 和 Java 之间来回切换。JavaConfig 为开发人员提供了一种纯 Java 方法来配置与 XML 配置概念相似的 Spring 容器。从技术角度来讲，只使用 JavaConfig 配置类来配置容器是可行的，但实际上很多人认为将JavaConfig 与 XML 混合匹配是理想的。

（3）类型安全和重构友好。JavaConfig 提供了一种类型安全的方法来配置 Spring容器。由于 Java 5.0 对泛型的支持，现在可以按类型而不是按名称检索 bean，不需要任何强制转换或基于字符串的查找。

Spring Boot 自动配置原理是什么?

注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是自动配置的核心，

EnableAutoConfiguration 给容器导入META-INF/spring.factories 里定义的自动配置类。

筛选有效的自动配置类。

每一个自动配置类结合对应的 xxxProperties.java 读取配置文件进行自动配置功能

如何理解 Spring Boot 配置加载顺序?

在 Spring Boot 里面，可以使用以下几种方式来加载配置。

1）properties文件；

2）YAML文件；

3）系统环境变量；

4）命令行参数；

等等……

我们可以在 Spring Beans 里面直接使用这些配置文件中加载的值，如：

1、使用 @Value 注解直接注入对应的值，这能获取到 Spring 中 Environment 的值；

2、使用 @ConfigurationProperties 注解把对应的值绑定到一个对象；

3、直接获取注入 Environment 进行获取；

配置属性的方式很多，Spring boot使用了一种独有的 PropertySource 可以很方便的覆盖属性的值。

配置属性加载的顺序如下：

1、开发者工具 Devtools 全局配置参数；

2、单元测试上的 @TestPropertySource` 注解指定的参数；

3、单元测试上的 @SpringBootTest` 注解指定的参数；

4、命令行指定的参数，如 java -jar springboot.jar –name=”Java技术栈”；

5、命令行中的 SPRING_APPLICATION_JSONJSON 指定参数, 如 java -Dspring.application.json=’{“name”:”Java技术栈”}’ -jar springboot.jar

6、ServletConfig 初始化参数；

7、ServletContext 初始化参数；

8、JNDI参数（如 java:comp/env/spring.application.json）；

9、Java系统参数（来源：System.getProperties()）；

10、操作系统环境变量参数；

11、RandomValuePropertySource 随机数，仅匹配：ramdom.*；

12、JAR包外面的配置文件参数（application-{profile}.properties（YAML））

13、JAR包里面的配置文件参数（application-{profile}.properties（YAML））

14、JAR包外面的配置文件参数（application.properties（YAML））

15、JAR包里面的配置文件参数（application.properties（YAML））

16、@Configuration](mailto:@Configuration)配置文件上 [@PropertySource` 注解加载的参数；

17、默认参数（通过 SpringApplication.setDefaultProperties 指定）；

数字小的优先级越高，即数字小的会覆盖数字大的参数值，我们来实践下，验证以上配置参数的加载顺序。

什么是 YAML?

YAML 是一种人类可读的数据序列化语言。它通常用于配置文件。与属性文件相比，如果我们想要在配置文件中添加复杂的属性，YAML 文件就更加结构化，而且更少混淆。可以看出 YAML 具有分层配置数据。

YAML 配置的优势在哪里？

YAML 现在可以算是非常流行的一种配置文件格式了，无论是前端还是后端，都可以见到 YAML 配置。那么 YAML 配置和传统的 properties 配置相比到底有哪些优势呢?

配置有序，在一些特殊的场景下，配置有序很关键
支持数组，数组中的元素可以是基本数据类型也可以是对象
简洁
相比 properties 配置文件，YAML 还有一个缺点，就是不支持 @PropertySource 注解导入自定义的 YAML 配置。

Spring Boot 是否可以使用 XML 配置？

Spring Boot 推荐使用 Java 配置而非 XML 配置，但是 Spring Boot 中也可以使用 XML 配置，通过 @ImportResource 注解可以引入一个 XML 配置。

eg:

ImportResource({“classpath:some-context.xml”,”classpath:another-context.xml”})

spring boot 核心配置文件是什么？bootstrap.properties 和 application.properties 有何区别？

单纯做 Spring Boot 开发，可能不太容易遇到 bootstrap.properties 配置文件，但是在结合 Spring Cloud 时，这个配置就会经常遇到了，特别是在需要加载一些远程配置文件的时侯。

spring boot 核心的两个配置文件:

bootstrap (. yml 或者 . properties)：boostrap 由父 ApplicationContext 加载的，比 applicaton 优先加载，配置在应用程序上下文的引导阶段生效。一般来说我们在 Spring Cloud Config 或者 Nacos 中会用到它。且 boostrap 里面的属性不能被覆盖；
application (. yml 或者 . properties)： 由ApplicatonContext 加载，用于 spring boot 项目的自动化配置。

什么是 Spring Profiles?

Spring Profiles 允许用户根据不同配置文件（dev，test，prod 等）来注册 bean。因此，当应用程序在开发中运行时，只有某些 bean 可以加载，而在 PRODUCTION中，某些其他 bean 可以加载。假设我们的要求是 Swagger 文档仅适用于 QA 环境，并且禁用所有其他文档。这可以使用配置文件来完成。

Spring Boot 使得使用配置文件非常简单。

如何在自定义端口上运行 Spring Boot 应用程序?

为了在自定义端口上运行 Spring Boot 应用程序，您可以在application.properties 中指定端口。

```
server.port = 8090
```

安全

如何实现 Spring Boot 应用程序的安全性?

为了实现 Spring Boot 的安全性，我们使用 spring-boot-starter-security 依赖项，并且必须添加安全配置。它只需要很少的代码。配置类将必须扩展WebSecurityConfigurerAdapter 并覆盖其方法。

比较一下 Spring Security 和 Shiro 各自的优缺点？

由于 Spring Boot 官方提供了大量的非常方便的开箱即用的 Starter ，包括 Spring Security 的 Starter ，使得在 Spring Boot 中使用 Spring Security 变得更加容易，甚至只需要添加一个依赖就可以保护所有的接口，所以，如果是 Spring Boot 项目，一般选择 Spring Security 。当然这只是一个建议的组合，单纯从技术上来说，无论怎么组合，都是没有问题的。Shiro 和 Spring Security 相比，主要有如下一些特点：

Spring Security 是一个重量级的安全管理框架；Shiro 则是一个轻量级的安全管理框架

Spring Security 概念复杂，配置繁琐；Shiro 概念简单、配置简单

Spring Security 功能强大；Shiro 功能简单

Spring Boot 中如何解决跨域问题？

跨域可以在前端通过 JSONP 来解决，但是 JSONP 只可以发送 GET 请求，无法发送其他类型的请求，在 RESTful 风格的应用中，就显得非常鸡肋，因此我们推荐在后端通过 （CORS，Cross-origin resource sharing） 来解决跨域问题。这种解决方案并非 Spring Boot 特有的，在传统的 SSM 框架中，就可以通过 CORS 来解决跨域问题，只不过之前我们是在 XML 文件中配置 CORS ，现在可以通过实现WebMvcConfigurer接口然后重写addCorsMappings方法解决跨域问题。

```
@Configuration
public class CorsConfig implements WebMvcConfigurer {

    @Override
```

```
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("*")
                .allowCredentials(true)
                .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS"
)
                .maxAge(3600);
    }

}

12345678910111213
```

项目中前后端分离部署，所以需要解决跨域的问题。
我们使用cookie存放用户登录的信息，在spring拦截器进行权限控制，当权限不符合时，直接返回给用户固定的json结果。
当用户登录以后，正常使用；当用户退出登录状态时或者token过期时，由于拦截器和跨域的顺序有问题，出现了跨域的现象。
我们知道一个http请求，先走filter，到达servlet后才进行拦截器的处理，如果我们把cors放在filter里，就可以优先于权限拦截器执行。

```
@Configuration
public class CorsConfig {

    @Bean
    public CorsFilter corsFilter() {
        CorsConfiguration corsConfiguration = new CorsConfiguration();
        corsConfiguration.addAllowedOrigin("*");
        corsConfiguration.addAllowedHeader("*");
        corsConfiguration.addAllowedMethod("*");
        corsConfiguration.setAllowCredentials(true);
        UrlBasedCorsConfigurationSource urlBasedCorsConfigurationSource =
new UrlBasedCorsConfigurationSource();
        urlBasedCorsConfigurationSource.registerCorsConfiguration("/**",
 corsConfiguration);
        return new CorsFilter(urlBasedCorsConfigurationSource);
    }

}
1234567891011121314151 6
```

什么是 CSRF 攻击?

CSRF 代表跨站请求伪造。这是一种攻击，迫使最终用户在当前通过身份验证的Web 应用程序上执行不需要的操作。CSRF 攻击专门针对状态改变请求，而不是数据窃取，因为攻击者无法查看对伪造请求的响应。

监视器

Spring Boot 中的监视器是什么?

Spring boot actuator 是 spring 启动框架中的重要功能之一。Spring boot 监视器可帮助您访问生产环境中正在运行的应用程序的当前状态。有几个指标必须在生产环境中进行检查和监控。即使一些外部应用程序可能正在使用这些服务来向相关人员触发警报消息。监视器模块公开了一组可直接作为 HTTP URL 访问的REST 端点来检查状态。

如何在 Spring Boot 中禁用 Actuator 端点安全性?

默认情况下，所有敏感的 HTTP 端点都是安全的，只有具有 ACTUATOR 角色的用户才能访问它们。安全性是使用标准的 HttpServletRequest.isUserInRole 方法实施的。 我们可以使用来禁用安全性。只有在执行机构端点在防火墙后访问时，才建议禁用安全性。

我们如何监视所有 Spring Boot 微服务?

Spring Boot 提供监视器端点以监控各个微服务的度量。这些端点对于获取有关应用程序的信息（如它们是否已启动）以及它们的组件（如数据库等）是否正常运行很有帮助。但是，使用监视器的一个主要缺点或困难是，我们必须单独打开应用程序的知识点以了解其状态或健康状况。想象一下涉及 50 个应用程序的微服务，管理员将不得不击中所有 50 个应用程序的执行终端。为了帮助我们处理这种情况，我们将使用位于的开源项目。 它建立在 Spring Boot Actuator 之上，它提供了一个 Web UI，使我们能够可视化多个应用程序的度量。

前后端分离，如何维护Rest API接口文档？

前后端分离开发日益流行，大部分情况下，我们都是通过 Spring Boot 做前后端分离开发，前后端分离一定会有接口文档，不然会前后端会深深陷入到扯皮中。一个比较笨的方法就是使用 word 或者 md 来维护接口文档，但是效率太低，接口一变，所有人手上的文档都得变。在 Spring Boot 中，这个问题常见的解决方案是 Swagger ，使用 Swagger 我们可以快速生成一个接口文档网站，接口一旦发生变化，文档就会自动更新，所有开发工程师访问这一个在线网站就可以获取到最新的接口文档，非常方便。

整合第三方项目

什么是 WebSockets?

WebSocket 是一种计算机通信协议，通过单个 TCP 连接提供全双工通信信道。

1、WebSocket 是双向的 -使用 WebSocket 客户端或服务器可以发起消息发送。

2、WebSocket 是全双工的 -客户端和服务器通信是相互独立的。

3、单个 TCP 连接 -初始连接使用 HTTP，然后将此连接升级到基于套接字的连接。然后这个单一连接用于所有未来的通信

4、Light -与 http 相比，WebSocket 消息数据交换要轻得多。

什么是 Spring Data ?

Spring Data 是 Spring 的一个子项目。用于简化数据库访问，支持NoSQL 和 关系数据存储。其主要目标是使数据库的访问变得方便快捷。Spring Data 具有如下特点：

SpringData 项目支持 NoSQL 存储：

MongoDB （文档数据库）
Neo4j（图形数据库）
Redis（键/值存储）
Hbase（列族数据库）
SpringData 项目所支持的关系数据存储技术：

JDBC
JPA
Spring Data Jpa 致力于减少数据访问层 (DAO) 的开发量. 开发者唯一要做的，就是声明持久层的接口，其他都交给 Spring Data JPA 来帮你完成！Spring Data JPA 通过规范方法的名字，根据符合规范的名字来确定方法需要实现什么样的逻辑。

什么是 Spring Batch?

Spring Boot Batch 提供可重用的函数，这些函数在处理大量记录时非常重要，包括日志/跟踪，事务管理，作业处理统计信息，作业重新启动，跳过和资源管理。它还提供了更先进的技术服务和功能，通过优化和分区技术，可以实现极高批量和高性能批处理作业。简单以及复杂的大批量批处理作业可以高度可扩展的方式利用框架处理重要大量的信息。

什么是 FreeMarker 模板?

FreeMarker 是一个基于 Java 的模板引擎，最初专注于使用 MVC 软件架构进行动态网页生成。使用 Freemarker 的主要优点是表示层和业务层的完全分离。程序员可以处理应用程序代码，而设计人员可以处理 html 页面设计。最后使用freemarker 可以将这些结合起来，给出最终的输出页面。

如何集成 Spring Boot 和 ActiveMQ?

对于集成 Spring Boot 和 ActiveMQ，我们使用依赖关系。 它只需要很少的配置，并且不需要样板代码。

什么是 Apache Kafka?

Apache Kafka 是一个分布式发布 - 订阅消息系统。它是一个可扩展的，容错的发布 - 订阅消息系统，它使我们能够构建分布式应用程序。这是一个 Apache 顶级项目。Kafka 适合离线和在线消息消费。

什么是 Swagger？你用 Spring Boot 实现了它吗?

Swagger 广泛用于可视化 API，使用 Swagger UI 为前端开发人员提供在线沙箱。Swagger 是用于生成 RESTful Web 服务的可视化表示的工具，规范和完整框架实现。它使文档能够以与服务器相同的速度更新。当通过 Swagger 正确定义时，消费者可以使用最少量的实现逻辑来理解远程服务并与其进行交互。因此，Swagger消除了调用服务时的猜测。

前后端分离，如何维护接口文档？

前后端分离开发日益流行，大部分情况下，我们都是通过 Spring Boot 做前后端分离开发，前后端分离一定会有接口文档，不然会前后端会深深陷入到扯皮中。一个比较笨的方法就是使用 word 或者 md 来维护接口文档，但是效率太低，接口一变，所有人手上的文档都得变。在 Spring Boot 中，这个问题常见的解决方案是 Swagger ，使用 Swagger 我们可以快速生成一个接口文档网站，接口一旦发生变化，文档就会自动更新，所有开发工程师访问这一个在线网站就可以获取到最新的接口文档，非常方便。

其他

您使用了哪些 starter maven 依赖项?

使用了下面的一些依赖项

spring-boot-starter-activemq

spring-boot-starter-security

这有助于增加更少的依赖关系，并减少版本的冲突。

Spring Boot 中的 starter 到底是什么？

首先，这个 Starter 并非什么新的技术点，基本上还是基于 Spring 已有功能来实现的。首先它提供了一个自动化配置类，一般命名为 XXXAutoConfiguration ，在这个配置类中通过条件注解来决定一个配置是否生效（条件注解就是 Spring 中原本就有的），然后它还会提供一系列的默认配置，也允许开发者根据实际情况自定义相关配置，然后通过类型安全的属性注入将这些配置属性注入进来，新注入的属性会代替掉默认属性。正因为如此，很多第三方框架，我们只需要引入依赖就可以直接使用了。当然，开发者也可以自定义 Starter

spring-boot-starter-parent 有什么用？

我们都知道，新创建一个 Spring Boot 项目，默认都是有 parent 的，这个 parent 就是 spring-boot-starter-parent ，spring-boot-starter-parent 主要有如下作用：

定义了 Java 编译版本为 1.8 。
使用 UTF-8 格式编码。
继承自 spring-boot-dependencies，这个里边定义了依赖的版本，也正是因为继承了这个依赖，所以我们在写依赖时才不需要写版本号。
执行打包操作的配置。
自动化的资源过滤。
自动化的插件配置。
针对 application.properties 和 application.yml 的资源过滤，包括通过 profile 定义的不同环境的配置文件，例如 application-dev.properties 和 application-dev.yml。

Spring Boot 打成的 jar 和普通的 jar 有什么区别？

Spring Boot 项目最终打包成的 jar 是可执行 jar ，这种 jar 可以直接通过 java -jar xxx.jar 命令来运行，这种 jar 不可以作为普通的 jar 被其他项目依赖，即使依赖了也无法使用其中的类。

Spring Boot 的 jar 无法被其他项目依赖，主要还是他和普通 jar 的结构不同。普通的 jar 包，解压后直接就是包名，包里就是我们的代码，而 Spring Boot 打包成的可执行 jar 解压后，在 \BOOT-INF\classes 目录下才是我们的代码，因此无法被直接引用。如果非要引用，可以在 pom.xml 文件中增加配置，将 Spring Boot 项目打包成两个 jar ，一个可执行，一个可引用。

运行 Spring Boot 有哪几种方式?

1）打包用命令或者放到容器中运行

2）用 Maven/ Gradle 插件运行

3）直接执行 main 方法运行

Spring Boot 需要独立的容器运行吗?

可以不需要，内置了 Tomcat/ Jetty 等容器。

开启 Spring Boot 特性有哪几种方式?

1）继承spring-boot-starter-parent项目

2）导入spring-boot-dependencies项目依赖

如何使用 Spring Boot 实现异常处理?

Spring 提供了一种使用 ControllerAdvice 处理异常的非常有用的方法。 我们通过实现一个 ControlerAdvice 类，来处理控制器类抛出的所有异常。

如何使用 Spring Boot 实现分页和排序?

使用 Spring Boot 实现分页非常简单。使用 Spring Data-JPA 可以实现将可分页的传递给存储库方法。

微服务中如何实现 session 共享 ？

在微服务中，一个完整的项目被拆分成多个不相同的独立的服务，各个服务独立部署在不同的服务器上，各自的 session 被从物理空间上隔离开了，但是经常，我们需要在不同微服务之间共享 session ，常见的方案就是 Spring Session + Redis 来实现 session 共享。将所有微服务的 session 统一保存在 Redis 上，当各个微服务对 session 有相关的读写操作时，都去操作 Redis 上的 session 。这样就实现了 session 共享，Spring Session 基于 Spring 中的代理过滤器实现，使得 session 的同步操作对开发人员而言是透明的，非常简便。

Spring Boot 中如何实现定时任务 ？

定时任务也是一个常见的需求，Spring Boot 中对于定时任务的支持主要还是来自 Spring 框架。

在 Spring Boot 中使用定时任务主要有两种不同的方式，一个就是使用 Spring 中的 @Scheduled 注解，另一个则是使用第三方框架 Quartz。

使用 Spring 中的 @Scheduled 的方式主要通过 @Scheduled 注解来实现。

使用 Quartz ，则按照 Quartz 的方式，定义 Job 和 Trigger 即可。

微服务中如何实现 session 共享？

在微服务中，一个完整的项目被拆分成多个不相同的独立的服务，各个服务独立部署在不同的服务器上，各自的 session 被从物理空间上隔离开了，但是经常，我们需要在不同微服务之间共享 session ，常见的方案就是 Spring Session + Redis 来实现 session 共享。将所有微服务的 session 统一保存在 Redis 上，当各个微服务对 session 有相关的读写操作时，都去操作 Redis 上的 session 。这样就实现了 session 共享，Spring Session 基于 Spring 中的代理过滤器实现，使得 session 的同步操作对开发人员而言是透明的，非常简便。 session 共享大家可以参考：Spring Boot 一个依赖搞定 session 共享，没有比这更简单的方案了！

打包部署

什么是嵌入式服务器？我们为什么要使用嵌入式服务器呢？

思考一下在你的虚拟机上部署应用程序需要些什么。

第一步： 安装 Java

第二部： 安装 Web 或者是应用程序的服务器（Tomat/Wbesphere/Weblogic 等等）

第三部： 部署应用程序 war 包

如果我们想简化这些步骤，应该如何做呢？

让我们来思考如何使服务器成为应用程序的一部分？

你只需要一个安装了 Java 的虚拟机，就可以直接在上面部署应用程序了，

是不是很爽？

这个想法是嵌入式服务器的起源。

当我们创建一个可以部署的应用程序的时候，我们将会把服务器（例如，tomcat）嵌入到可部署的服务器中。

例如，对于一个 Spring Boot 应用程序来说，你可以生成一个包含 Embedded Tomcat 的应用程序 jar。你就可以想运行正常 Java 应用程序一样来运行 web 应用程序了。

嵌入式服务器就是我们的可执行单元包含服务器的二进制文件（例如，tomcat.jar）。

Spring Boot 如何实现热部署？

Spring Boot 实现热部署其实很容易，引入 devtools 依赖即可，这样当编译文件发生变化时，Spring Boot 就会自动重启。在 Eclipse 中，用户按下保存按键，就会自动编译进而重启 Spring Boot，IDEA 中由于是自动保存的，自动保存时并未编译，所以需要开发者按下 Ctrl+F9 进行编译，编译完成后，项目就自动重启了。

如果仅仅只是页面模板发生变化，Java 类并未发生变化，此时可以不用重启 Spring Boot，使用 LiveReload 插件就可以轻松实现热部署。

如何重新加载 Spring Boot 上的更改，而无需重新启动服务器? Spring Boot项目如何热部署?

这可以使用 DEV 工具来实现。通过这种依赖关系，您可以节省任何更改，嵌入式tomcat 将重新启动。Spring Boot 有一个开发工具（DevTools）模块，它有助于提高开发人员的生产力。Java 开发人员面临的一个主要挑战是将文件更改自动部署到服务器并自动重启服务器。开发人员可以重新加载 Spring Boot 上的更改，而无需重新启动服务器。这将消除每次手动部署更改的需要。Spring Boot 在发布它的第一个版本时没有这个功能。这是开发人员最需要的功能。DevTools 模块完全满足开发人员的需求。该模块将在生产环境中被禁用。它还提供 H2 数据库控制台以更好地测试应用程序。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

为什么我们需要 spring-boot-maven-plugin?

spring-boot-maven-plugin 提供了一些像 jar 一样打包或者运行应用程序的命令。

spring-boot:run 运行你的 SpringBooty 应用程序。

spring-boot：repackage 重新打包你的 jar 包或者是 war 包使其可执行

spring-boot：start 和 spring-boot：stop 管理 Spring Boot 应用程序的生命周期（也可以说是为了集成测试）。

spring-boot:build-info 生成执行器可以使用的构造信息。

如何使用 Spring Boot 部署到不同的服务器?

你需要做下面两个步骤：

在一个项目中生成一个 war 文件。
将它部署到你最喜欢的服务器（websphere 或者 Weblogic 或者 Tomcat and so on）。
第二步：取决于你的服务器。

Spring Boot 打成的 jar 和普通的 jar 有什么区别 ?

Spring Boot 项目最终打包成的 jar 是可执行 jar ，这种 jar 可以直接通过 java -jar xxx.jar 命令来运行，这种 jar 不可以作为普通的 jar 被其他项目依赖，即使依赖了也无法使用其中的类。

Spring Boot 的 jar 无法被其他项目依赖，主要还是他和普通 jar 的结构不同。普通的 jar 包，解压后直接就是包名，包里就是我们的代码，而 Spring Boot 打包成的可执行 jar 解压后，在 \BOOT-INF\classes 目录下才是我们的代码，因此无法被直接引用。如果非要引用，可以在 pom.xml 文件中增加配置，将 Spring Boot 项目打包成两个 jar ，一个可执行，一个可引用。

Redis提升试题

1、在项目中缓存是如何使用的？为什么要用缓存？缓存使用不当会造成什么后果？

面试官心理分析

这个问题，互联网公司必问，要是一个人连缓存都不太清楚，那确实比较尴尬。

只要问到缓存，上来第一个问题，肯定是先问问你项目哪里用了缓存？为啥要用？不用行不行？如果用了以后可能会有什么不良的后果？

这就是看看你对缓存这个东西背后有没有思考，如果你就是傻乎乎的瞎用，没法给面试官一个合理的解答，那面试官对你印象肯定不太好，觉得你平时思考太少，就知道干活儿。

面试题剖析

项目中缓存是如何使用的？

这个，需要结合自己项目的业务来。

为什么要用缓存？

用缓存，主要有两个用途：高性能、高并发。

高性能

假设这么个场景，你有个操作，一个请求过来，吭哧吭哧你各种乱七八糟操作 mysql，半天查出来一个结果，耗时 600ms。但是这个结果可能接下来几个小时都不会变了，或者变了也可以不用立即反馈给用户。那么此时咋办？

缓存啊，折腾 600ms 查出来的结果，扔缓存里，一个 key 对应一个 value，下次再有人查，别走 mysql折腾 600ms 了，直接从缓存里，通过一个 key 查出来一个 value，2ms 搞定。性能提升 300 倍。

就是说对于一些需要复杂操作耗时查出来的结果，且确定后面不怎么变化，但是有很多读请求，那么直接将查询出来的结果放在缓存中，后面直接读缓存就好。

高并发

所以要是你有个系统，高峰期一秒钟过来的请求有 1 万，那一个 mysql 单机绝对会死掉。你这个时候就只能上缓存，把很多数据放缓存，别放 mysql。缓存功能简单，说白了就是 key-value 式操作，单机支撑的并发量轻松一秒几万十几万，支撑高并发 so easy。单机承载并发量是 mysql 单机的几十倍。

缓存是走内存的，内存天然就支撑高并发。

用了缓存之后会有什么不良后果?

常见的缓存问题有以下几个:

缓存与数据库双写不一致 、缓存雪崩、缓存穿透、缓存并发竞争后面再详细说明。

2、redis 和 memcached 有什么区别？redis 的线程模型是什么？为什么 redis 单线程却能支撑高并发?

面试官心理分析

这个是问 redis 的时候，最基本的问题吧，redis 最基本的一个内部原理和特点，就是 redis 实际上是个单线程工作模型，你要是这个都不知道，那后面玩儿 redis 的时候，出了问题岂不是什么都不知道?

还有可能面试官会问问你 redis 和 memcached 的区别，但是 memcached 是早些年各大互联网公司常用的缓存方案，但是现在近几年基本都是 redis，没什么公司用 memcached 了。

面试题剖析

redis 和 memcached 有啥区别?

redis 支持复杂的数据结构
redis 相比 memcached 来说，拥有更多的数据结构，能支持更丰富的数据操作。如果需要缓存能够支持更复杂的结构和操作， redis 会是不错的选择。

redis 原生支持集群模式
在 redis3.x 版本中，便能支持 cluster 模式，而 memcached 没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据。

性能对比

由于 redis 只使用单核，而 memcached 可以使用多核，所以平均每一个核上 redis 在存储小数据时比memcached 性能更高。而在 100k 以上的数据中，memcached 性能要高于 redis。虽然 redis 最近也在存储大数据的性能上进行优化，但是比起 memcached，还是稍有逊色。

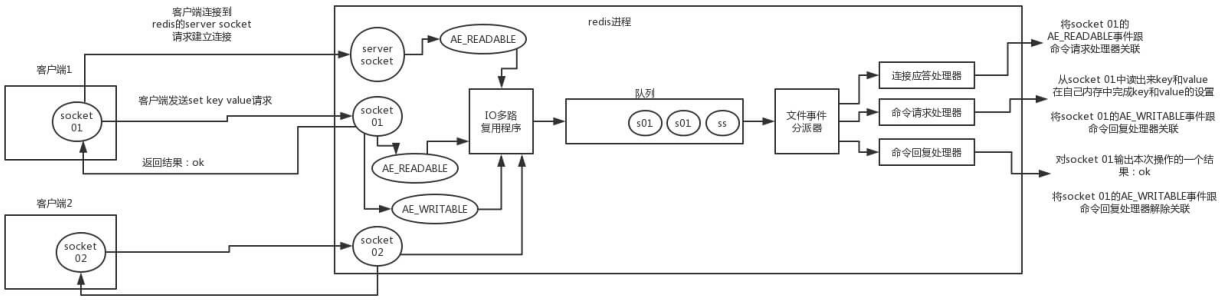
redis 的线程模型

redis 内部使用文件事件处理器 file event handler，这个文件事件处理器是单线程的，所以 redis 才叫做单线程的模型。它采用 IO 多路复用机制同时监听多个 socket，将产生事件的 socket 压入内存队列中，事件分派器根据 socket 上的事件类型来选择对应的事件处理器进行处理。

文件事件处理器的结构包含 4 个部分:

多个 socket
IO 多路复用程序
文件事件分派器
事件处理器（连接应答处理器、命令请求处理器、命令回复处理器）
多个 socket 可能会并发产生不同的操作，每个操作对应不同的文件事件，但是 IO 多路复用程序会监听多个 socket，会将产生事件的 socket 放入队列中排队，事件分派器每次从队列中取出一个 socket，根据 socket 的事件类型交给对应的事件处理器进行处理。

来看客户端与 redis 的一次通信过程：



要明白，通信是通过 socket 来完成的，不懂的同学可以先去看一看 socket 网络编程。
客户端 socket01 向 redis 的 server socket 请求建立连接，此时 server socket 会产生一个 AE_READABLE 事件，IO 多路复用程序监听到 server socket 产生的事件后，将该事件压入队列中。文件事件分派器从队列中获取该事件，交给连接应答处理器。连接应答处理器会创建一个能与客户端通信的 socket01，并将该 socket01 的 AE_READABLE 事件与命令请求处理器关联。
假设此时客户端发送了一个 set key value 请求，此时 redis 中的 socket01 会产生 AE_READABLE 事件，IO 多路复用程序将事件压入队列，此时事件分派器从队列中获取到该事件，由于前面 socket01 的 AE_READABLE 事件已经与命令请求处理器关联，因此事件分派器将事件交给命令请求处理器来处理。命令请求处理器读取 socket01 的 key value 并在自己内存中完成 key value 的设置。操作完成后，它会将 socket01 的 AE_WRITABLE 事件与令回复处理器关联。
如果此时客户端准备好接收返回结果了，那么 redis 中的 socket01 会产生一个 AE_WRITABLE 事件，同样压入队列中，事件分派器找到相关联的命令回复处理器，由命令回复处理器对 socket01 输入本次操作的一个结果，比如 ok，之后解除 socket01 的 AE_WRITABLE 事件与命令回复处理器的关联。

这样便完成了一次通信。

为啥 redis 单线程模型也能效率这么高？

纯内存操作
核心是基于非阻塞的 IO 多路复用机制
单线程反而避免了多线程的频繁上下文切换问题

3、redis 都有哪些数据类型？分别在哪些场景下使用比较合适？

面试官心理分析

除非是面试官感觉看你简历，是工作 3 年以内的比较初级的同学，可能对技术没有很深入的研究，面试官才会问这类问题。否则，在宝贵的面试时间里，面试官实在不想多问。

其实问这个问题，主要有两个原因：

看看你到底有没有全面的了解 redis 有哪些功能，一般怎么来用，啥场景用什么，就怕你别就会最简单的 KV 操作；
看看你在实际项目里都怎么玩儿过 redis。
要是你回答的不好，没说出几种数据类型，也没说什么场景，你完了，面试官对你印象肯定不好，觉得你平时就是做个简单的 set 和 get。

面试题剖析

redis 主要有以下几种数据类型：

string
hash
list
set
sorted set

string

这是最简单的类型，就是普通的 set 和 get，做简单的 KV 缓存。

```
set college szu
```

hash

这个是类似 map 的一种结构，这个一般就是可以将结构化的数据，比如一个对象（前提是这个对象没嵌套其他的对象）给缓存在 redis 里，然后每次读写缓存的时候，可以就操作 hash 里的某个字段。

```
hset person name bingo
hset person age 20
hset person id 1
hget person name
person = {
    "name": "bingo",
    "age": 20,
    "id": 1
}
```

list

list 是有序列表，这个可以玩儿出很多花样。

比如可以通过 list 存储一些列表型的数据结构，类似粉丝列表、文章的评论列表之类的东西。

比如可以通过 lrange 命令，读取某个闭区间内的元素，可以基于 list 实现分页查询，这个是很棒的一个功能，基于 redis 实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西，性能高，就一页一页走。

```
# 0开始位置，-1结束位置，结束位置为-1时，表示列表的最后一个位置，即查看所有。
lrange mylist 0 -1
```

比如可以搞个简单的消息队列，从 list 头怼进去，从 list 尾巴那里弄出来。

```
lpush mylist 1
lpush mylist 2
lpush mylist 3 4 5

# 1
rpop mylist
```

set

set 是无序集合，自动去重。

直接基于 set 将系统里需要去重的数据扔进去，自动就给去重了，如果你需要对一些数据进行快速的全局去重，你当然也可以基于 jvm 内存里的 HashSet 进行去重，但是如果你的某个系统部署在多台机器上呢？得基于 redis 进行全局的 set 去重。

可以基于 set 玩儿交集、并集、差集的操作，比如交集吧，可以把两个人的粉丝列表整一个交集，看看俩人的共同好友是谁？对吧。

把两个大 V 的粉丝都放在两个 set 中，对两个 set 做交集。

```
#-------操作一个set-------
# 添加元素
sadd mySet 1

# 查看全部元素
smembers mySet

# 判断是否包含某个值
sismember mySet 3

# 删除某个/些元素
srem mySet 1
srem mySet 2 4

# 查看元素个数
scard mySet

# 随机删除一个元素
spop mySet

#-------操作多个set-------
# 将一个set的元素移动到另外一个set
smove yourSet mySet 2
```

```
# 求两set的交集
sinter yourSet mySet

# 求两set的并集
sunion yourSet mySet

# 求在yourSet中而不在mySet中的元素
sdiff yourSet mySet
```

sorted set

sorted set 是排序的 set，去重但可以排序，写进去的时候给一个分数，自动根据分数排序。

```
zadd board 85 zhangsan
zadd board 72 lisi
zadd board 96 wangwu
zadd board 63 zhaoliu

# 获取排名前三的用户（默认是升序，所以需要 rev 改为降序）
zrevrange board 0 3

# 获取某用户的排名
zrank board zhaoliu
```

4、redis 的过期策略都有哪些？内存淘汰机制都有哪些？手写一下LRU代码实现？

面试官心理分析

如果你连这个问题都不知道，上来就懵了，回答不出来，那线上你写代码的时候，想当然的认为写进 redis的数据就一定会存在，后面导致系统各种 bug，谁来负责？

常见的有两个问题：

（1）往 redis 写入的数据怎么没了？

可能有同学会遇到，在生产环境的 redis 经常会丢掉一些数据，写进去了，过一会儿可能就没了。我的天，同学，你问这个问题就说明 redis 你就没用对啊。redis 是缓存，你给当存储了是吧？

啥叫缓存？用内存当缓存。内存是无限的吗，内存是很宝贵而且是有限的，磁盘是廉价而且是大量的。可能一台机器就几十个 G 的内存，但是可以有几个 T 的硬盘空间。redis 主要是基于内存来进行高性能、高并发的读写操作的。

那既然内存是有限的，比如 redis 就只能用 10G，你要是往里面写了 20G 的数据，会咋办？当然会干掉10G 的数据，然后就保留 10G 的数据了。那干掉哪些数据？保留哪些数据？当然是干掉不常用的数据，保留常用的数据了。

（2）数据明明过期了，怎么还占用着内存?

这是由 redis 的过期策略来决定。

面试题剖析

redis 过期策略

redis 过期策略是：定期删除+惰性删除。

所谓定期删除，指的是 redis 默认是每隔 100ms 就随机抽取一些设置了过期时间的 key，检查其是否过期，如果过期就删除。

假设 redis 里放了 10w 个 key，都设置了过期时间，你每隔几百毫秒，就检查 10w 个 key，那 redis 基本上就死了，cpu 负载会很高的，消耗在你的检查过期 key 上了。注意，这里可不是每隔 100ms 就遍历所有的设置过期时间的 key，那样就是一场性能上的灾难。实际上 redis 是每隔 100ms 随机抽取一些key 来检查和删除的。

但是问题是，定期删除可能会导致很多过期 key 到了时间并没有被删除掉，那咋整呢？所以就是惰性删除了。这就是说，在你获取某个 key 的时候，redis 会检查一下 ，这个 key 如果设置了过期时间那么是否过期了？如果过期了此时就会删除，不会给你返回任何东西。

获取 key 的时候，如果此时 key 已经过期，就删除，不会返回任何东西。

答案是：走内存淘汰机制。

内存淘汰机制

redis 内存淘汰机制有以下几个：

noeviction: 当内存不足以容纳新写入数据时，新写入操作会报错，这个一般没人用吧，实在是太恶心了。
allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的 key（这个是最常用的）。
allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个 key，这个一般没人用吧，为啥要随机，肯定是把最近最少使用的 key 给干掉啊。
volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的 key（这个一般不太合适）。
volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个 key。
volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的 key 优先移除。

手写一个 LRU 算法

你可以现场手写最原始的 LRU 算法，那个代码量太大了，似乎不太现实。

不求自己纯手工从底层开始打造出自己的 LRU，但是起码要知道如何利用已有的 JDK 数据结构实现一个Java 版的 LRU。

```
import java.util.LinkedHashMap;
```

```
import java.util.Map;

public class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private final int CACHE_SIZE;

    /**
     * 传递进来最多能缓存多少数据
     * @param cacheSize 缓存大小
     */
    public LRUCache(int cacheSize) {

        //true 表示让 linkedHashMap 按照访问顺序来进行排序，最近访问的放在头部，最老访问的的放在尾部
        super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true);
        CACHE_SIZE = cacheSize;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        //当map中的数据量大于制定的缓存个数的时候，就自动删除最老的数据
        return size() > CACHE_SIZE;
    }
}
```

父类的构造方法：LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder);

initialCapacity:初始容量

loadFactor:装载因子

accessOrder:访问顺序

问题1：关于装载因子的为什么是0.75

请参见尼恩的 《HashMap - 图解 -秒懂》 博文和配套视频

问题2：accessOrder访问顺序 的取值含义：

false，所有的Entry按照插入的顺序排列
true，所有的Entry按照访问的顺序排列

5、如何保证 redis 的高并发和高可用？redis 的主从复制原理能介绍一下么？redis 的哨兵原理能介绍一下么？

面试官心理分析
其实问这个问题，主要是考考你，redis 单机能承载多高并发？如果单机扛不住如何扩容扛更多的并发？redis 会不会挂？既然 redis 会挂那怎么保证 redis 是高可用的？

其实针对的都是项目中你肯定要考虑的一些问题，如果你没考虑过，那确实你对生产系统中的问题思考太少。

面试题剖析
如果你用 redis 缓存技术的话，肯定要考虑如何用 redis 来加多台机器，保证 redis 是高并发的，还有就是如何让 redis 保证自己不是挂掉以后就直接死掉了，即 redis 高可用。

由于此节内容较多，因此，会分为两个小节进行讲解。- redis 主从架构 - redis 基于哨兵实现高可用 redis 实现高并发主要依靠主从架构，一主多从，一般来说，很多项目其实就足够了，单主用来写入数据，单机几万 QPS，多从用来查询数据，多个从实例可以提供每秒 10w 的 QPS。

如果想要在实现高并发的同时，容纳大量的数据，那么就需要 redis 集群，使用 redis 集群之后，可以提供每秒几十万的读写并发。

redis 高可用，如果是做主从架构部署，那么加上哨兵就可以了，就可以实现，任何一个实例宕机，可以进行主备切换。

6、redis 的持久化有哪几种方式？不同的持久化机制都有什么 优缺点？持久化机制具体底层是如何实现的？

面试官心理分析

redis 如果仅仅只是将数据缓存在内存里面，如果 redis 宕机了再重启，内存里的数据就全部都弄丢了啊。

你必须得用 redis 的持久化机制，将数据写入内存的同时，异步的慢慢的将数据写入磁盘文件里，进行持久化。

如果 redis 宕机重启，自动从磁盘上加载之前持久化的一些数据就可以了，也许会丢失少许数据，但是至少不会将所有数据都弄丢。

这个其实一样，针对的都是 redis 的生产环境可能遇到的一些问题，就是 redis 要是挂了再重启，内存里的数据不就全丢了？能不能重启的时候把数据给恢复了？

面试题剖析

持久化主要是做灾难恢复、数据恢复，也可以归类到高可用的一个环节中去，比如你 redis 整个挂了，然后 redis 就不可用了，你要做的事情就是让 redis 变得可用，尽快变得可用。

重启 redis，尽快让它对外提供服务，如果没做数据备份，这时候 redis 启动了，也不可用啊，数据都没了。

很可能说，大量的请求过来，缓存全部无法命中，在 redis 里根本找不到数据，这个时候就死定了，出现缓存雪崩问题。所有请求没有在redis命中，就会去mysql数据库这种数据源头中去找，一下子mysql承接高并发，然后就挂了…

如果你把 redis 持久化做好，备份和恢复方案做到企业级的程度，那么即使你的 redis 故障了，也可以通过备份数据，快速恢复，一旦恢复立即对外提供服务。

redis 持久化的两种方式

RDB：RDB 持久化机制，是对 redis 中的数据执行周期性的持久化。
AOF：AOF 机制对每条写入命令作为日志，以 append-only 的模式写入一个日志文件中，在 redis

重启的时候，可以通过回放 AOF 日志中的写入指令来重新构建整个数据集。
通过 RDB 或 AOF，都可以将 redis 内存中的数据给持久化到磁盘上面来，然后可以将这些数据备份到别的地方去，比如说阿里云等云服务。

如果 redis 挂了，服务器上的内存和磁盘上的数据都丢了，可以从云服务上拷贝回来之前的数据，放到指定的目录中，然后重新启动 redis，redis 就会自动根据持久化数据文件中的数据，去恢复内存中的数据，继续对外提供服务。

如果同时使用 RDB 和 AOF 两种持久化机制，那么在 redis 重启的时候，会使用 AOF 来重新构建数据，因为 AOF 中的数据更加完整。

RDB 优缺点

RDB 会生成多个数据文件，每个数据文件都代表了某一个时刻中 redis 的数据，这种多个数据文件的方式，非常适合做冷备，可以将这种完整的数据文件发送到一些远程的安全存储上去，比如说 Amazon的 S3 云服务上去，在国内可以是阿里云的 ODPS 分布式存储上，以预定好的备份策略来定期备份 redis中的数据。
RDB 对 redis 对外提供的读写服务，影响非常小，可以让 redis 保持高性能，因为 redis 主进程只需要 fork 一个子进程，让子进程执行磁盘 IO 操作来进行 RDB 持久化即可。·
相对于 AOF 持久化机制来说，直接基于 RDB 数据文件来重启和恢复 redis 进程，更加快速。
如果想要在 redis 故障时，尽可能少的丢失数据，那么 RDB 没有 AOF 好。一般来说，RDB 数据快照文件，都是每隔 5 分钟，或者更长时间生成一次，这个时候就得接受一旦 redis 进程宕机，那么会丢失最近 5 分钟的数据。
RDB 每次在 fork 子进程来执行 RDB 快照数据文件生成的时候，如果数据文件特别大，可能会导致对客户端提供的服务暂停数毫秒，或者甚至数秒。

AOF 优缺点

AOF 可以更好的保护数据不丢失，一般 AOF 会每隔 1 秒，通过一个后台线程执行一次 fsync 操作，最多丢失 1 秒钟的数据。
AOF 日志文件以 append-only 模式写入，所以没有任何磁盘寻址的开销，写入性能非常高，而且文件不容易破损，即使文件尾部破损，也很容易修复。
AOF 日志文件即使过大的时候，出现后台重写操作，也不会影响客户端的读写。因为在 rewrite log 的时候，会对其中的指令进行压缩，创建出一份需要恢复数据的最小日志出来。在创建新日志文件的时候，老的日志文件还是照常写入。当新的merge后日志文件ready的时候，在交换新老日志文件即可。
AOF 日志文件的命令通过非常可读的方式进行记录，这个特性非常适合做灾难性的误删除的紧急恢复。比如某人不小心用 flushall 命令清空了所有数据，只要这个时候后台 rewrite 还没有发生，那么就可以立即拷贝 AOF 文件，将最后一条 flushall 命令给删了，然后再将该 AOF 文件放回去，就可以通过恢复机制，自动恢复所有数据。
对于同一份数据来说，AOF 日志文件通常比 RDB 数据快照文件更大。
AOF 开启后，支持的写 QPS 会比 RDB 支持的写 QPS 低，因为 AOF 一般会配置成每秒 fsync 一次日志文件，当然，每秒一次 fsync，性能也还是很高的。（如果实时写入，那么 QPS 会大降，redis 性 能会大大降低）
以前 AOF 发生过 bug，就是通过 AOF 记录的日志，进行数据恢复的时候，没有恢复一模一样的数据出来。所以说，类似 AOF 这种较为复杂的基于命令日志 / merge / 回放的方式，比基于 RDB 每次持久化一份完整的数据快照文件的方式，更加脆弱一些，容易有 bug。不过 AOF 就是为了避免 rewrite 过程导致的 bug，因此每次 rewrite 并不是基于旧的指令日志进行 merge 的，而是基于当时内存中的数据进行指令的重新构建，这样健壮性会好很多。

RDB 和 AOF 到底该如何选择

不要仅仅使用 RDB，因为那样会导致你丢失很多数据；
也不要仅仅使用 AOF，因为那样有两个问题：第一，你通过 AOF 做冷备，没有 RDB 做冷备来的恢复速度更快；第二，RDB 每次简单粗暴生成数据快照，更加健壮，可以避免 AOF 这种复杂的备份和恢复机制的 bug；

9、如何保证缓存与数据库的双写一致性?

面试官心理分析析

你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的问题，那么你如何解决一致性问题？#### 面试题剖析试题剖析

一般来说，如果允许缓存可以稍微的跟数据库偶尔有不一致的情况，也就是说如果你的系统不是严格要求“缓存+数据库” 必须保持一致性的话，最好不要做这个方案，即：读请求和写请求串行化，串到一个内存队列里去。

串行化可以保证一定不会出现不一致的情况，但是它也会导致系统的吞吐量大幅度降低，用比正常情况下多几倍的机器去支撑线上的一个请求。

Cache Aside Pattern

最经典的缓存+数据库读写的模式，就是 Cache Aside Pattern。- 读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。- 更新的时候，先更新数据库，然后再删除缓存。

为什么是删除缓存，而不是更新缓存?

原因很简单，很多时候，在复杂点的缓存场景，缓存不单单是数据库中直接取出来的值。

比如可能更新了某个表的一个字段，然后其对应的缓存，是需要查询另外两个表的数据并进行运算，才能计算出缓存最新的值的。

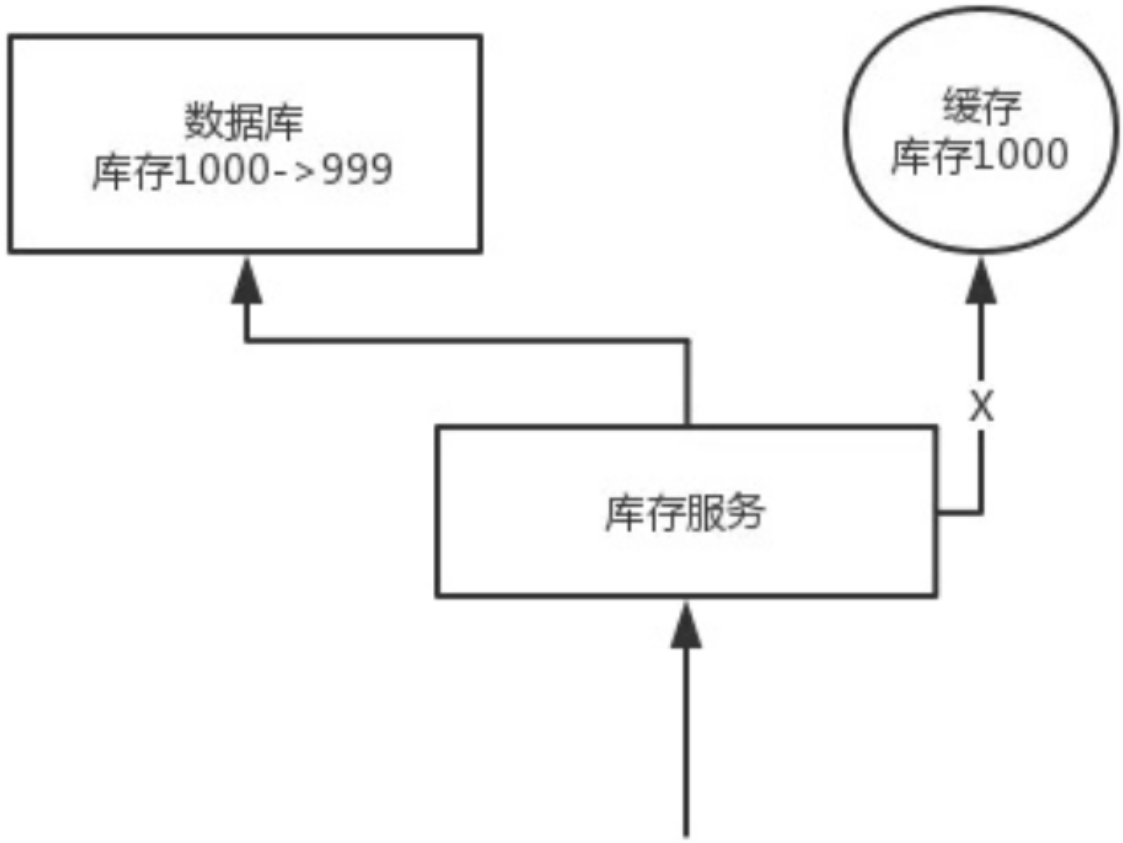
另外更新缓存的代价有时候是很高的。是不是说，每次修改数据库的时候，都一定要将其对应的缓存更新一份？也许有的场景是这样，但是对于比较复杂的缓存数据计算的场景，就不是这样了。如果你频繁修改一个缓存涉及的多个表，缓存也频繁更新。但是问题在于，这个缓存到底会不会被频繁访问到?

举个栗子，一个缓存涉及的表的字段，在 1 分钟内就修改了 20 次，或者是 100 次，那么缓存更新 20 次、100 次；但是这个缓存在 1 分钟内只被读取了 1 次，有大量的冷数据。实际上，如果你只是删除缓存的话，那么在 1 分钟内，这个缓存不过就重新计算一次而已，开销大幅度降低。用到缓存才去算缓存。

其实删除缓存，而不是更新缓存，就是一个 lazy 计算的思想，不要每次都重新做复杂的计算，不管它会不会用到，而是让它到需要被使用的时候再重新计算。像 mybatis，hibernate，都有懒加载思想。查询一个部门，部门带了一个员工的 list，没有必要说每次查询部门，都里面的 1000 个员工的数据也同时查出来啊。80%的情况，查这个部门，就只是要访问这个部门的信息就可以了。先查部门，同时要访问里面的员工，那么这个时候只有在你要访问里面的员工的时候，才会去数据库里面查询1000个员工。

最初级的缓存不一致问题及解决方案

问题：先更新数据库，再删除缓存。如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据就出现了不一致。



解决思路：先删除缓存，再更新数据库。如果数据库更新失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致。因为读的时候缓存没有，所以去读了数据库中的旧数据，然后更新到缓存中。

比较复杂的数据不一致问题分析

数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改。一个请求过来，去读缓存，发现缓存空了，去查询数据库，查到了修改前的旧数据，放到了缓存中。随后数据变更的程序完成了数据库的修改。完了，数据库和缓存中的数据不一样了…

为什么上亿流量高并发场景下，缓存会出现这个问题?

只有在对一个数据在并发的进行读写的时候，才可能会出现这种问题。其实如果说你的并发量很低的话，特别是读并发很低，每天访问量就 1 万次，那么很少的情况下，会出现刚才描述的那种不一致的场景。但是问题是，如果每天的是上亿的流量，每秒并发读是几万，每秒只要有数据更新的请求，就可能会出现上述的数据库+缓存不一致的情况。

解决方案如下：

更新数据的时候，根据数据的唯一标识，将操作路由之后，发送到一个 jvm 内部队列中。读取数据的时候，如果发现数据不在缓存中，那么将重新读取数据+更新缓存的操作，根据唯一标识路由之后，也发送同一个jvm 内部队列中。

一个队列对应一个工作线程，每个工作线程串行拿到对应的操作，然后一条一条的执行。这样的话一个数据变更的操作，先删除缓存，然后再去更新数据库，但是还没完成更新。此时如果一个读请求过来，没有读到缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，然后同步等待缓存更新完成。

这里有一个优化点，一个队列中，其实多个更新缓存请求串在一起是没意义的，因此可以做过滤，如果发现队列中已经有一个更新缓存的请求了，那么就不用再放个更新请求操作进去了，直接等待前面的更新操作请求完成即可。

待那个队列对应的工作线程完成了上一个操作的数据库的修改之后，才会去执行下一个操作，也就是缓存更新的操作，此时会从数据库中读取最新的值，然后写入缓存中。

如果请求还在等待时间范围内，不断轮询发现可以取到值了，那么就直接返回；如果请求等待的时间超过一定时长，那么这一次直接从数据库中读取当前的旧值。

高并发的场景下，该解决方案要注意的问题：

（1）读请求长时阻塞

由于读请求进行了非常轻度的异步化，所以一定要注意读超时的问题，每个读请求必须在超时时间范围内返回。该解决方案，最大的风险点在于说，可能数据更新很频繁，导致队列中积压了大量更新操作在里面，然后读请求会发生大量的超时，最后导致大量的请求直接走数据库。务必通过一些模拟真实的测试，看看更新数据的频率是怎样的。

另外一点，因为一个队列中，可能会积压针对多个数据项的更新操作，因此需要根据自己的业务情况进行测试，可能需要部署多个服务，每个服务分摊一些数据的更新操作。如果一个内存队列里居然会挤压 100 个商品的库存修改操作，每隔库存修改操作要耗费 10ms 去完成，那么最后一个商品的读请求，可能等待 10 *100 = 1000ms = 1s 后，才能得到数据，这个时候就导致读请求的长时阻塞。

一定要做根据实际业务系统的运行情况，去进行一些压力测试，和模拟线上环境，去看看最繁忙的时候，内存队列可能会挤压多少更新操作，可能会导致最后一个更新操作对应的读请求，会 hang 多少时间，如果读请求在 200ms 返回，如果你计算过后，哪怕是最繁忙的时候，积压 10 个更新操作，最多等待 200ms，那还可以的。

如果一个内存队列中可能积压的更新操作特别多，那么你就要加机器，让每个机器上部署的服务实例处理更少的数据，那么每个内存队列中积压的更新操作就会越少。

其实根据之前的项目经验，一般来说，数据的写频率是很低的，因此实际上正常来说，在队列中积压的更新操作应该是很少的。像这种针对读高并发、读缓存架构的项目，一般来说写请求是非常少的，每秒的 QPS 能到几百就不错了。

我们来实际粗略测算一下。

如果一秒有 500 的写操作，如果分成 5 个时间片，每 200ms 就 100 个写操作，放到 20 个内存队列中，每个内存队列，可能就积压 5 个写操作。每个写操作性能测试后，一般是在 20ms 左右就完成，那么针对每个内存队列的数据的读请求，也就最多 hang 一会儿，200ms 以内肯定能返回了。

经过刚才简单的测算，我们知道，单机支撑的写 QPS 在几百是没问题的，如果写 QPS 扩大了 10 倍，那么就扩容机器，扩容 10 倍的机器，每个机器 20 个队列。

（2）读请求并发量过高

这里还必须做好压力测试，确保恰巧碰上上述情况的时候，还有一个风险，就是突然间大量读请求会在几十 毫秒的延时 hang 在服务上，看服务能不能扛的住，需要多少机器才能扛住最大的极限情况的峰值。

但是因为并不是所有的数据都在同一时间更新，缓存也不会同一时间失效，所以每次可能也就是少数数据的缓存失效了，然后那些数据对应的读请求过来，并发量应该也不会特别大。

（3）多服务实例部署的请求路由

可能这个服务部署了多个实例，那么必须保证说，执行数据更新操作，以及执行缓存更新操作的请求，都通过 Nginx 服务器路由到相同的服务实例上。

比如说，对同一个商品的读写请求，全部路由到同一台机器上。可以自己去做服务间的按照某个请求参数的hash 路由，也可以用 Nginx 的 hash 路由功能等等。

（4）热点商品的路由问题，导致请求的倾斜

万一某个商品的读写请求特别高，全部打到相同的机器的相同的队列里面去了，可能会造成某台机器的压力过大。就是说，因为只有在商品数据更新的时候才会清空缓存，然后才会导致读写并发，所以其实要根据业务系统去看，如果更新频率不是太高的话，这个问题的影响并不是特别大，但是的确可能某些机器的负载会高一些。

10 为什么是删除缓存，而不是更新缓存?

原因很简单，很多时候，在复杂点的缓存场景，缓存不单单是数据库中直接取出来的值。

比如可能更新了某个表的一个字段，然后其对应的缓存，是需要查询另外两个表的数据并进行运算，才能计算出缓存最新的值的。

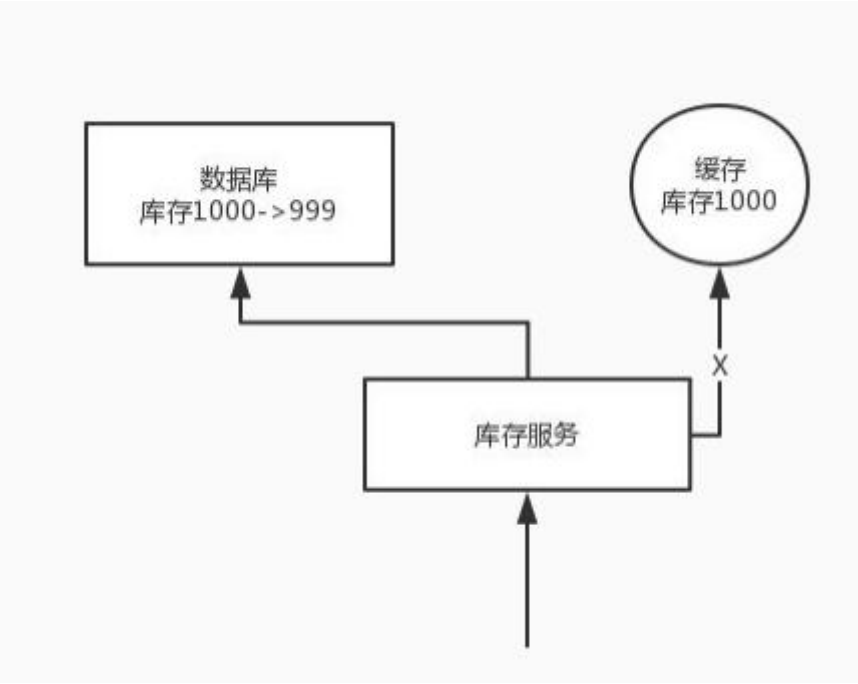
另外更新缓存的代价有时候是很高的。是不是说，每次修改数据库的时候，都一定要将其对应的缓存更新一份？也许有的场景是这样，但是对于比较复杂的缓存数据计算的场景，就不是这样了。
如果你频繁修改一个缓存涉及的多个表，缓存也频繁更新。但是问题在于，这个缓存到底会不会被频繁访问到?

举个栗子，一个缓存涉及的表的字段，在 1 分钟内就修改了 20 次，或者是 100 次，那么缓存更新 20 次、100 次；但是这个缓存在 1 分钟内只被读取了 1 次，有大量的冷数据。
实际上，如果你只是删除缓存的话，那么在 1 分钟内，这个缓存不过就重新计算一次而已，开销大幅度降低。用到缓存才去算缓存。

其实删除缓存，而不是更新缓存，就是一个 lazy 计算的思想，不要每次都重新做复杂的计算，不管它会不会用到，而是让它到需要被使用的时候再重新计算。像 mybatis，hibernate，都有懒加载思想。
查询一个部门，部门带了一个员工的 list，没有必要说每次查询部门，都里面的 1000 个员工的数据也同时查出来啊。80%的情况，查这个部门，就只是要访问这个部门的信息就可以了。先查部门，同时要访问里面的员工，那么这个时候只有在你要访问里面的员工的时候，才会去数据库里面查询1000个员工。

2）最初级的缓存不一致问题及解决方案
问题：先更新数据库，再删除缓存。如果删除缓存失败了，那么会导致数据库中是新数据，缓存中

是旧数据，数据就出现了不一致。



解决思路：先删除缓存，再更新数据库。如果数据库更新失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致。因为读的时候缓存没有，所以去读了数据库中的旧数据，然后更新到缓存中。

3）比较复杂的数据不一致问题分析
数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改。一个请求过来，去读缓存，发现缓存空了，去查询数据库，查到了修改前的旧数据，放到了缓存中。随后数据变更的程序完成了数据库的修改。
完了，数据库和缓存中的数据不一样了…

11、生产环境中的 redis 是怎么部署的?

面试官心理分析析

看看你了解不了解你们公司的 redis 生产集群的部署架构，如果你不了解，那么确实你就很失职了，你的redis 是主从架构？集群架构？用了哪种集群方案？有没有做高可用保证？有没有开启持久化机制确保可以进行数据恢复？线上 redis 给几个 G 的内存？设置了哪些参数？压测后你们 redis 集群承载多少QPS?

兄弟，这些你必须是门儿清的，否则你确实是没好好思考过。

面试题剖析

redis cluster，10 台机器，5 台机器部署了 redis 主实例，另外 5 台机器部署了 redis 的从实例，每个主实例挂了一个从实例，5 个节点对外提供读写服务，每个节点的读写高峰 qps 可能可以达到每秒 5 万，5 台机器最多是 25 万读写请求/s。

机器是什么配置？32G 内存+ 8 核 CPU + 1T 磁盘，但是分配给 redis 进程的是 10g 内存，一般线上生产环境，redis 的内存尽量不要超过 10g，超过 10g 可能会有问题。

5 台机器对外提供读写，一共有 50g 内存。

因为每个主实例都挂了一个从实例，所以是高可用的，任何一个主实例宕机，都会自动故障迁移，redis 从实例会自动变成主实例继续提供读写服务。

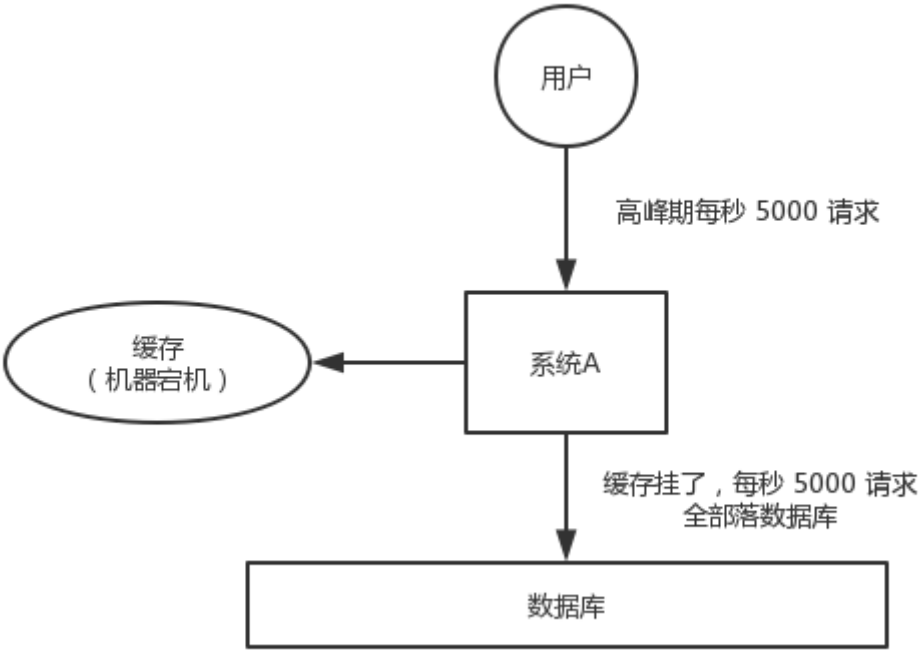
你往内存里写的是什么数据？每条数据的大小是多少？商品数据，每条数据是 10kb。100 条数据是 1mb，10 万条数据是 1g。常驻内存的是 200 万条商品数据，占用内存是 20g，仅仅不到总内存的 50%。目前高峰期每秒就是 3500 左右的请求量。

其实大型的公司，会有基础架构的 team 负责缓存集群的运维。

12.了解什么是 Redis 的雪崩、穿透和击穿？Redis 崩溃之后会怎么样？系统该如何应对这种情况？如何处理 Redis 的穿透？

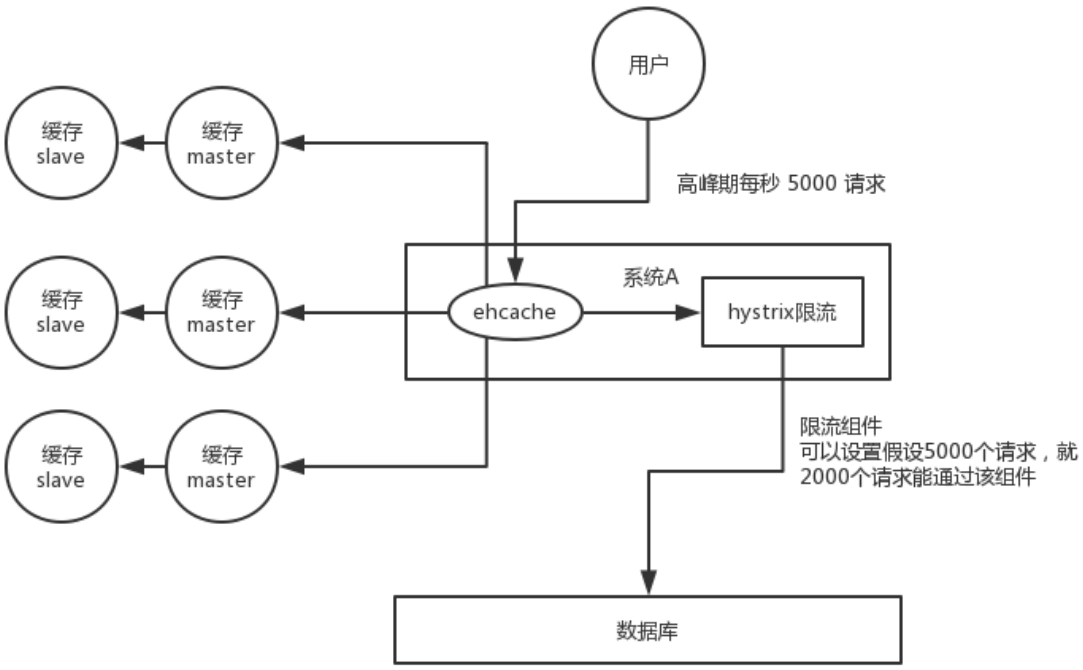
缓存雪崩

对于系统 A，假设每天高峰期每秒 5000 个请求，本来缓存在高峰期可以扛住每秒 4000 个请求，但是缓存机器意外发生了全盘宕机。缓存挂了，此时 1 秒 5000 个请求全部落数据库，数据库必然扛不住，它会报一下警，然后就挂了。此时，如果没有采用什么特别的方案来处理这个故障，DBA 很着急，重启数据库，但是数据库立马又被新的流量给打死了。



缓存雪崩的事前事中事后的解决方案如下：

事前：Redis 高可用，主从+哨兵，Redis cluster，避免全盘崩溃。
事中：本地 ehcache 缓存 + hystrix 限流&降级，避免 MySQL 被打死。
事后：Redis 持久化，一旦重启，自动从磁盘上加载数据，快速恢复缓存数据。



用户发送一个请求，系统 A 收到请求后，先查本地 ehcache 缓存，如果没查到再查 Redis。如果 ehcache 和 Redis 都没有，再查数据库，将数据库中的结果，写入 ehcache 和 Redis 中。

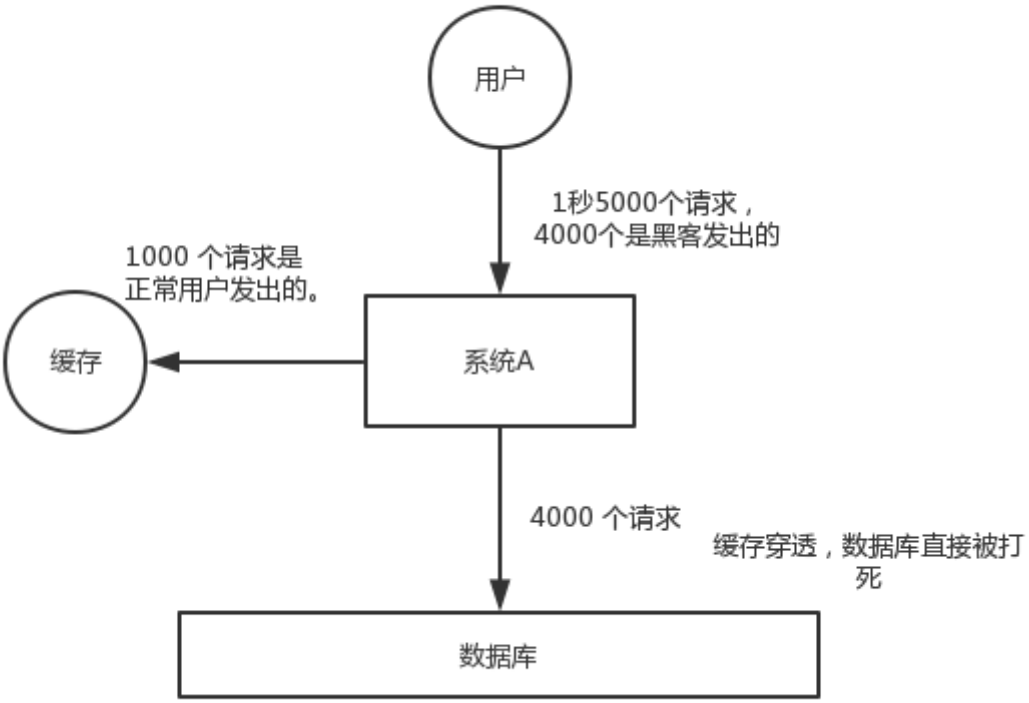
限流组件，可以设置每秒的请求，有多少能通过组件，剩余的未通过的请求，怎么办？走降级！可以返回一些默认的值，或者友情提示，或者空值。

好处：

数据库绝对不会死，限流组件确保了每秒只有多少个请求能通过。
只要数据库不死，就是说，对用户来说，2/5 的请求都是可以被处理的。
只要有 2/5 的请求可以被处理，就意味着你的系统没死，对用户来说，可能就是点击几次刷不出来页面，但是多点几次，就可以刷出来了。
缓存穿透
对于系统A，假设一秒 5000 个请求，结果其中 4000 个请求是黑客发出的恶意攻击。

黑客发出的那 4000 个攻击，缓存中查不到，每次你去数据库里查，也查不到。

举个栗子。数据库 id 是从 1 开始的，结果黑客发过来的请求 id 全部都是负数。这样的话，缓存中不会有，请求每次都“绕过缓存”，直接查询数据库。这种恶意攻击场景的缓存穿透就会直接把数据库给打死。



解决方式很简单，每次系统 A 从数据库中只要没查到，就写一个空值到缓存里去，比如 set -999 UNKNOWN 。然后设置一个过期时间，这样的话，下次有相同的 key 来访问的时候，在缓存失效之前，都可以直接从缓存中取数据。

这种方式虽然是简单，在某些场景（如数据量大的博客）下不优雅，还可能会缓存过多的空值，更加优雅的方式就是：使用 bitmap 布隆过滤

缓存击穿
缓存击穿，就是说某个 key 非常热点，访问非常频繁，处于集中式高并发访问的情况，当这个 key 在失效的瞬间，大量的请求就击穿了缓存，直接请求数据库，就像是在一道屏障上凿开了一个洞。

不同场景下的解决方式可如下：

若缓存的数据是基本不会发生更新的，则可尝试将该热点数据设置为永不过期。
若缓存的数据更新不频繁，且缓存刷新的整个流程耗时较少的情况下，则可以采用基于 Redis、zookeeper 等分布式中间件的分布式互斥锁，或者本地互斥锁以保证仅少量的请求能请求数据库并重新构建缓存，其余线程则在锁释放后能访问到新缓存。
若缓存的数据更新频繁或者在缓存刷新的流程耗时较长的情况下，可以利用定时线程在缓存过期前主动地重新构建缓存或者延后缓存的过期时间，以保证所有的请求能一直访问到对应的缓存。
缓存击穿和 缓存穿透这两者的区别：
缓存击穿重点在“击” 就是某个或者是几个热点 key 穿透了缓存层
缓存穿透重点在“透”：大量的请求绕过了缓存层

Redis布隆过滤器防恶意流量击穿缓存

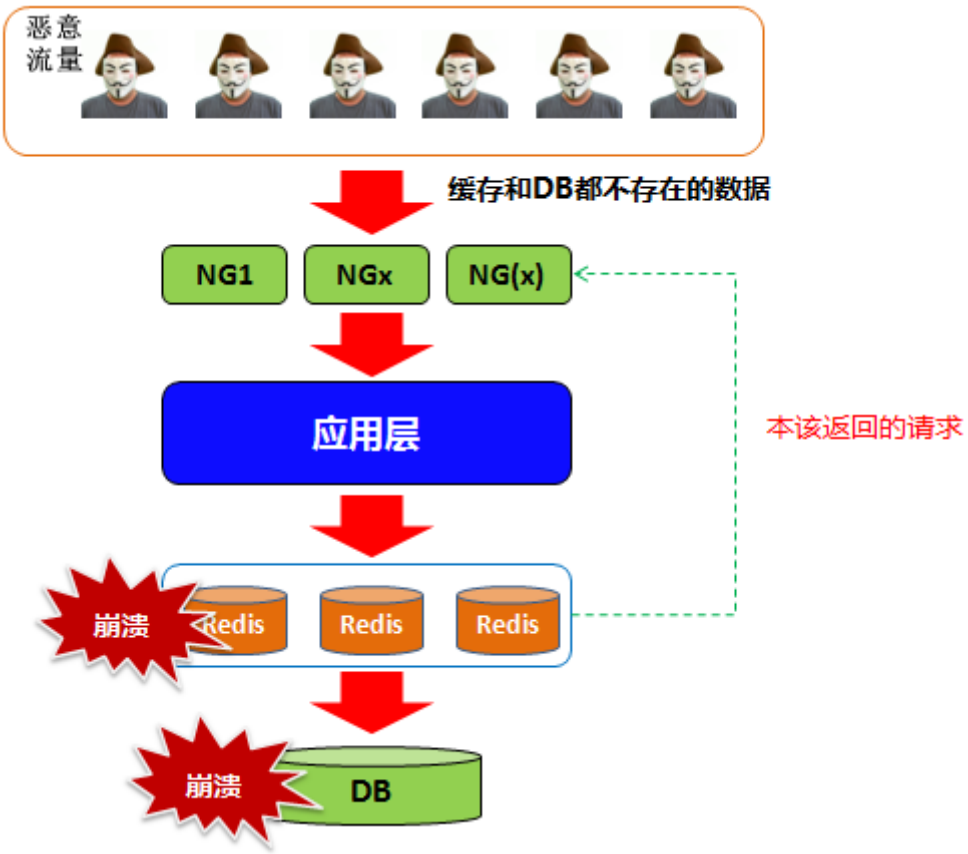
什么是恶意流量穿透

假设我们的Redis里存有一组用户的注册email，以email作为Key存在，同时它对应着DB里的User表的部分字段。

一般来说，一个合理的请求过来我们会先在Redis里判断这个用户是否是会员，因为从缓存里读数据返回快。如果这个会员在缓存中不存在那么我们会去DB中查询一下。

现在试想，有千万个不同IP的请求（不要以为没有，我们就在2018年和2019年碰到了，因为攻击的成本很低）带着Redis里根本不存在的key来访问你的网站，这时我们来设想一下：

请求到达Web服务器；
请求派发到应用层->微服务层；
请求去Redis捞数据，Redis内不存在这个Key；
于是请求到达DB层，在DB建立connection后进行一次查询
千万乃至上亿的DB连接请求，先不说Redis是否撑的住DB也会被瞬间打爆。这就是“Redis穿透”，它会打爆你的缓存或者是连DB一起打爆进而引起一系列的“雪崩效应”。



怎么防

那就是使用布隆过滤器，可以把所有的user表里的关键查询字段放于Redis的bloom过滤器内。有人会说，这不疯了，我有4000万会员？so what!

你把4000会员放在Redis里是比较夸张，有些网站有8000万、1亿会员呢?

因此我没让你直接放在Redis里，而是放在布隆过滤器内!

布隆过滤器内不是直接把key,value这样放进去的，它存放的内容是这么一个bitmap中。

bitmap

所谓的Bit-map就是用一个bit位来标记某个元素对应的Value，通过Bit为单位来存储数据，可以大大节省存储空间.
所以我们可以通过一个int型的整数的32比特位来存储32个10进制的数字，那么这样所带来的好处

是内存占用少、
效率很高（不需要比较和位移）比如我们要存储5(101)、3(11)四个数字，那么我们申请int型的内存空间，会有32
个比特位。这四个数字的二进制分别对应
从右往左开始数，比如第一个数字是5，对应的二进制数据是101, 那么从右往左数到第5位，把对应的二进制数据
存储到32个比特位上。

第一个5就是 00000000000000000000000000101000
输入3时候 00000000000000000000000000001100

Bloom Filte介绍

1. 含义

(1). 布隆过滤器（Bloom Filter）是由Howard Bloom在1970年提出的一种比较巧妙的概率型数据结构，它实际上是由一个很长的二进制(0或1)向量和一系列随机映射函数组成。

(2). 布隆过滤器可以用于检索一个元素是否在一个集合中。它可以告诉你某种东西一定不存在或者可能存在。当布隆过滤器说，某种东西存在时，这种东西可能不存在；当布隆过滤器说，某种东西不存在时，那么这种东西一定不存在。

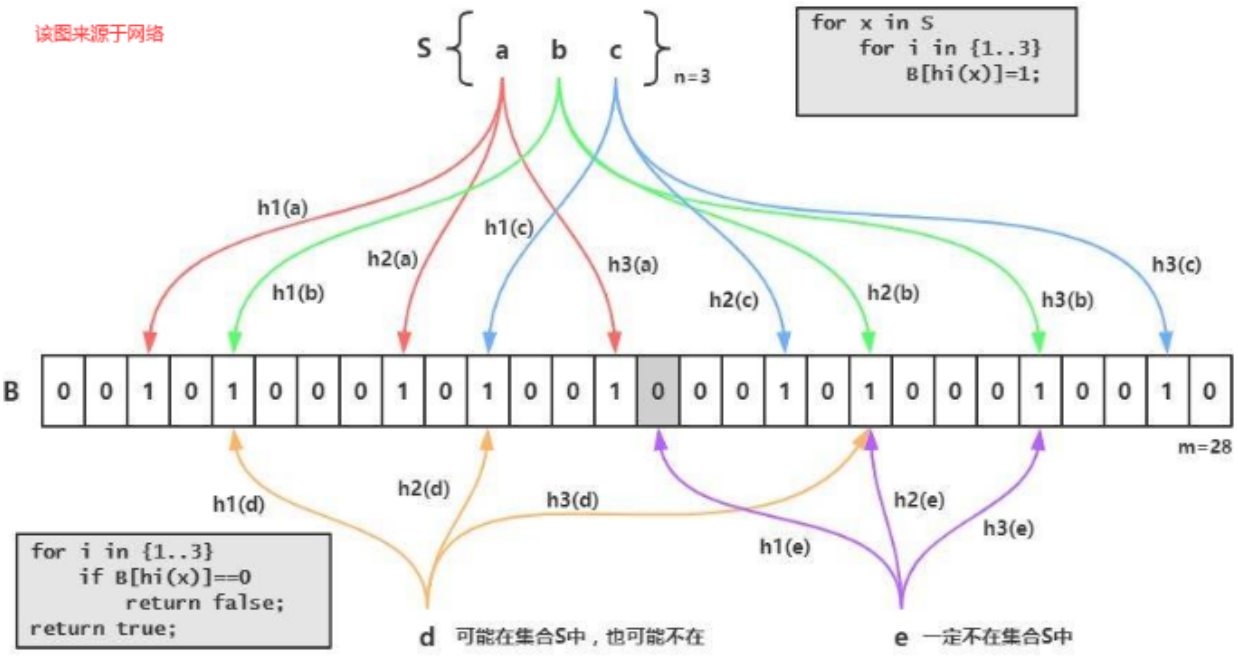
(3). 布隆过滤器 优点：A. 空间效率高，占用空间少 B. 查询时间短

缺点：A. 有一定的误判率 B. 元素不能删除

2. 原理

当一个元素被加入集合时，通过K个散列函数将这个元素映射成一个位数组中的K个点（使用多个哈希函数对元素key (bloom中不存value) 进行哈希，算出一个整数索引值，然后对位数组长度进行取模运算得到一个位置，每个无偏哈希函数都会得到一个不同的位置），把它们置为1。

检索时，我们只要看看这些点是不是都是1就（大约）知道集合中有没有它了：① 如果这些点有任何一个为0（如下图的e），则被检元素一定不在；如果都是1（如下图的d），并不能完全说明这个元素就一定存在其中，有可能这些位置为1是因为其他元素的存在，这就是布隆过滤器会出现误判的原因。



补充：

Bloom Filter跟 ‘’单哈希函数BitMap‘’ 不同之处在于：Bloom Filter使用了k个哈希函数，每个字符串跟k个bit对应，从而降低了冲突的概率。

3. 实现

(1). Redis的bitmap

基于redis的 bitmap数据结构 的相关指令来执行。

(2). RedisBloom （推荐）

布隆过滤器可以使用Redis中的位图(bitmap)操作实现，直到Redis4.0版本提供了插件功能，Redis官方提供的布隆过滤器才正式登场，布隆过滤器作为一个插件加载到Redis Server中，官网推荐了一个 RedisBloom 作为 Redis 布隆过滤器的 Module。

详细安装、指令操作参考：https://github.com/RedisBloom/RedisBloom

文档地址：https://oss.redislabs.com/redisbloom/

(3). PyreBloom

pyreBloom 是 Python 中 Redis + BloomFilter 模块，是c语言实现。如果觉得Redis module的形式部署很麻烦或者线上环境Redis版本不是 4.0 及以上，则可以采用这个，但是它是在 hiredis 基础上，需要安装hiredis，且不支持重连和重试。

(4). Lua脚本实现

详见：https://github.com/erikdubbelboer/redis-lua-scaling-bloom-filter

(5). guvua包自带的布隆过滤器

Bloom Filte 应用场景

1. 解决缓存穿透

(1). 含义

业务请求中数据缓存中没有，DB中也没有，导致类似请求直接跨过缓存，反复在DB中查询，与此同时缓存也不会得到更新。（详见：https://www.cnblogs.com/yaopengfei/p/13878124.html）

(2). 解决思路

事先把存在的key都放到redis的Bloom Filter 中，他的用途就是存在性检测，如果 BloomFilter 中不存在，那么数据一定不存在；如果 BloomFilter 中存在，实际数据也有可能会不存在。

剖析：**布隆过滤器可能会误判，放过部分请求，当不影响整体，所以目前该方案是处理此类问题最佳方案。

2. 黑名单校验

识别垃圾邮件，只要发送者在黑名单中的，就识别为垃圾邮件。假设黑名单的数量是数以亿计的，存放起来就是非常耗费存储空间的，布隆过滤器则是一个较好的解决方案。把所有黑名单都放在布隆过滤器中，再收到邮件时，判断邮件地址是否在布隆过滤器中即可。

ps：

如果用哈希表，每存储一亿个 email地址，就需要 1.6GB的内存（用哈希表实现的具体办法是将每一个 email地址对应成一个八字节的信息指纹，然后将这些信息指纹存入哈希表，由于哈希表的存储效率一般只有 50%，因此一个 email地址需要占用十六个字节。一亿个地址大约要 1.6GB，即十六亿字节的内存）。因此存贮几十亿个邮件地址可能需要上百 GB的内存。而Bloom Filter只需要哈希表 1/8到 1/4 的大小就能解决同样的问题。

3. Web拦截器

(1). 含义

如果相同请求则拦截，防止重复被攻击。

(2). 解决思路

用户第一次请求，将请求参数放入布隆过滤器中，当第二次请求时，先判断请求参数是否被布隆过滤器命中，从而提高缓存命中率。

布隆过滤器其他场景

比如有如下几个需求：

1、原本有10亿个号码，现在又来了10万个号码，要快速准确判断这10万个号码是否在10亿个号码库中?

解决办法一：将10亿个号码存入数据库中，进行数据库查询，准确性有了，但是速度会比较慢。

解决办法二：将10亿号码放入内存中，比如Redis缓存中，这里我们算一下占用内存大小：10亿*8字节=8GB，通过内存查询，准确性和速度都有了，但是大约8gb的内存空间，挺浪费内存空间的。

2、接触过爬虫的，应该有这么一个需求，需要爬虫的网站千千万万，对于一个新的网站url，我们如何判断这个url我们是否已经爬过了？

解决办法还是上面的两种，很显然，都不太好。

3、同理还有垃圾邮箱的过滤。

那么对于类似这种，大数据量集合，如何准确快速的判断某个数据是否在大数据量集合中，并且不占用内存，布隆过滤器应运而生了。

4、假设我用python爬虫爬了4亿条url，需要去重？

给Redis安装Bloom Filter

Redis从4.0才开始支持bloom filter，因此本例中我们使用的是Redis5.4。

Redis的bloom filter下载地址在这：https://github.com/RedisLabsModules/redisbloom.git

```
git clone https://github.com/RedisLabsModules/redisbloom.git
cd redisbloom
make # 编译
```

让Redis启动时可以加载bloom filter有两种方式：

手工加载式：

```
redis-server --loadmodule ./redisbloom/rebloom.so
```

每次启动自加载：

编辑Redis的redis.conf文件，加入：

```
loadmodule /soft/redisbloom/redisbloom.so
```

Like this:

```
#
# If instead you are interested in using includes to override configuration
# options, it is better to use include as the last line.
#
# include /path/to/local.conf
# include /path/to/other.conf

################################## MODULES #####################################

# Load modules at startup. If the server is not able to load modules
# it will abort. It is possible to use multiple loadmodule directives.
#
# loadmodule /path/to/my_module.so
# loadmodule /path/to/other_module.so
loadmodule /soft/redisbloom/redisbloom.so
################################## NETWORK #####################################
```


在Redis里使用Bloom Filter

基本指令：

bf.reserve {key} {error_rate} {size}

```
127.0.0.1:6379> bf.reserve userid 0.01 100000
OK
```

上面这条命令就是：创建一个空的布隆过滤器，并设置一个期望的错误率和初始大小。{error_rate}过滤器的错误率在0-1之间，如果要设置0.1%，则应该是0.001。该数值越接近0，内存消耗越大，对cpu利用率越高。

bf.add {key} {item}

```
127.0.0.1:6379> bf.add userid '181920'
(integer) 1
```

上面这条命令就是：往过滤器中添加元素。如果key不存在，过滤器会自动创建。

bf.exists {key} {item}

```
127.0.0.1:6379> bf.exists userid '101310299'
(integer) 1
```

上面这条命令就是：判断指定key的value是否在bloomfilter里存在。存在：返回1，不存在：返回0。

引入Redis布隆过滤器防止缓存穿透

缓存穿透（大量查询一个不存在的key）定义：
缓存穿透，是指查询一个数据库中不一定存在的数据；
正常使用缓存查询数据的流程是，依据key去查询value，数据查询先进行缓存查询，如果key不存在或者key已经过期，再对数据库进行查询，并把查询到的对象，放进缓存。如果数据库查询对象为空，则不放进缓存。

如果每次都查询一个不存在value的key，由于缓存中没有数据，所以每次都会去查询数据库；当对key查询的并发请求量很大时，每次都访问DB，很可能对DB造成影响；并且由于缓存不命中，每次都查询持久层，那么也失去了缓存的意义。
缓存穿透　解决方法

第一种是缓存层缓存空值

将数据库中的空值也缓存到缓存层中，这样查询该空值就不会再访问DB，而是直接在缓存层访问就行。

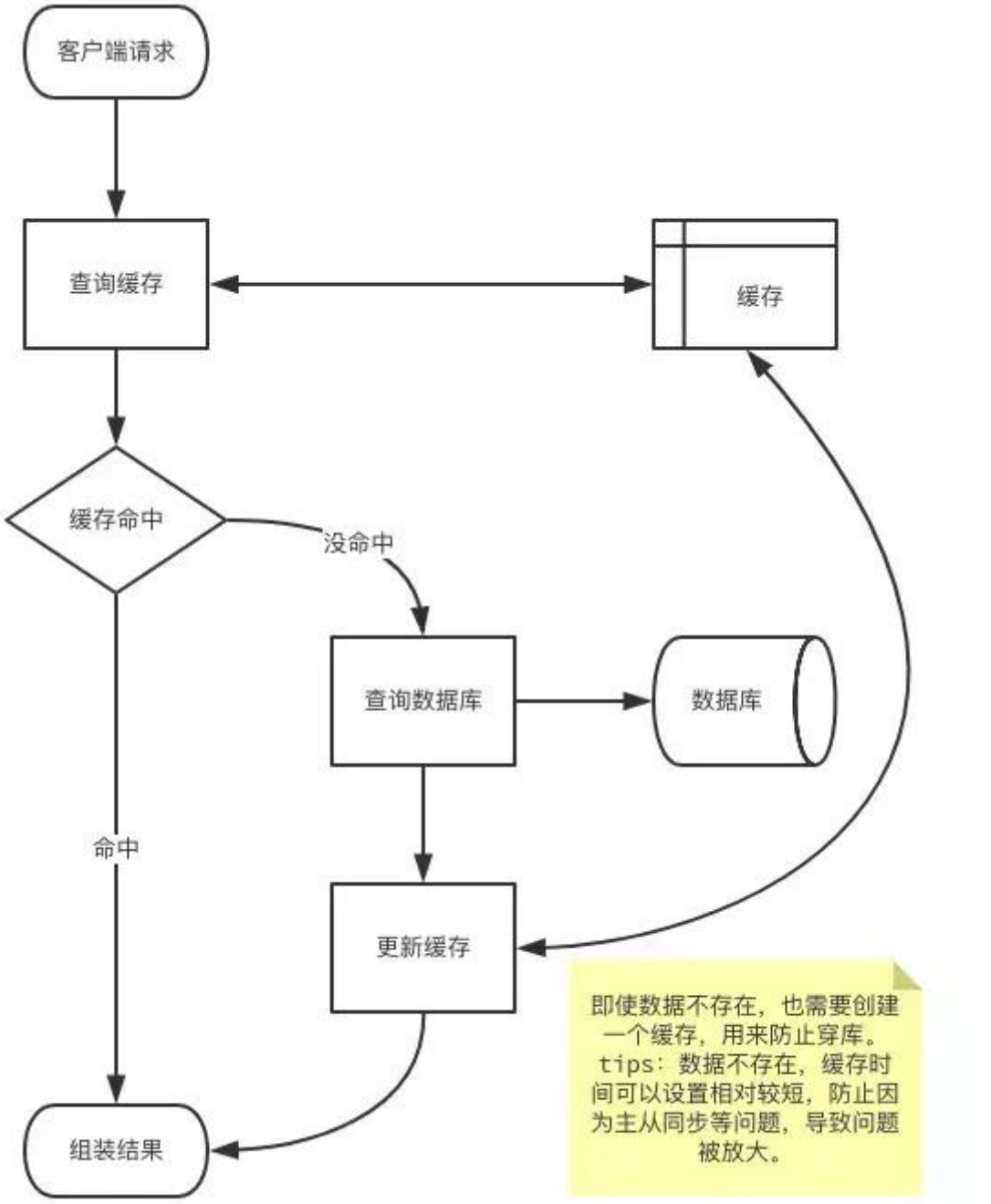
但是这样有个弊端就是缓存太多空值占用了更多的空间，可以通过给缓存层空值设立一个较短的过期时间来解决，例如60s。

第二种是布隆过滤器

将数据库中所有的查询条件，放入布隆过滤器中，

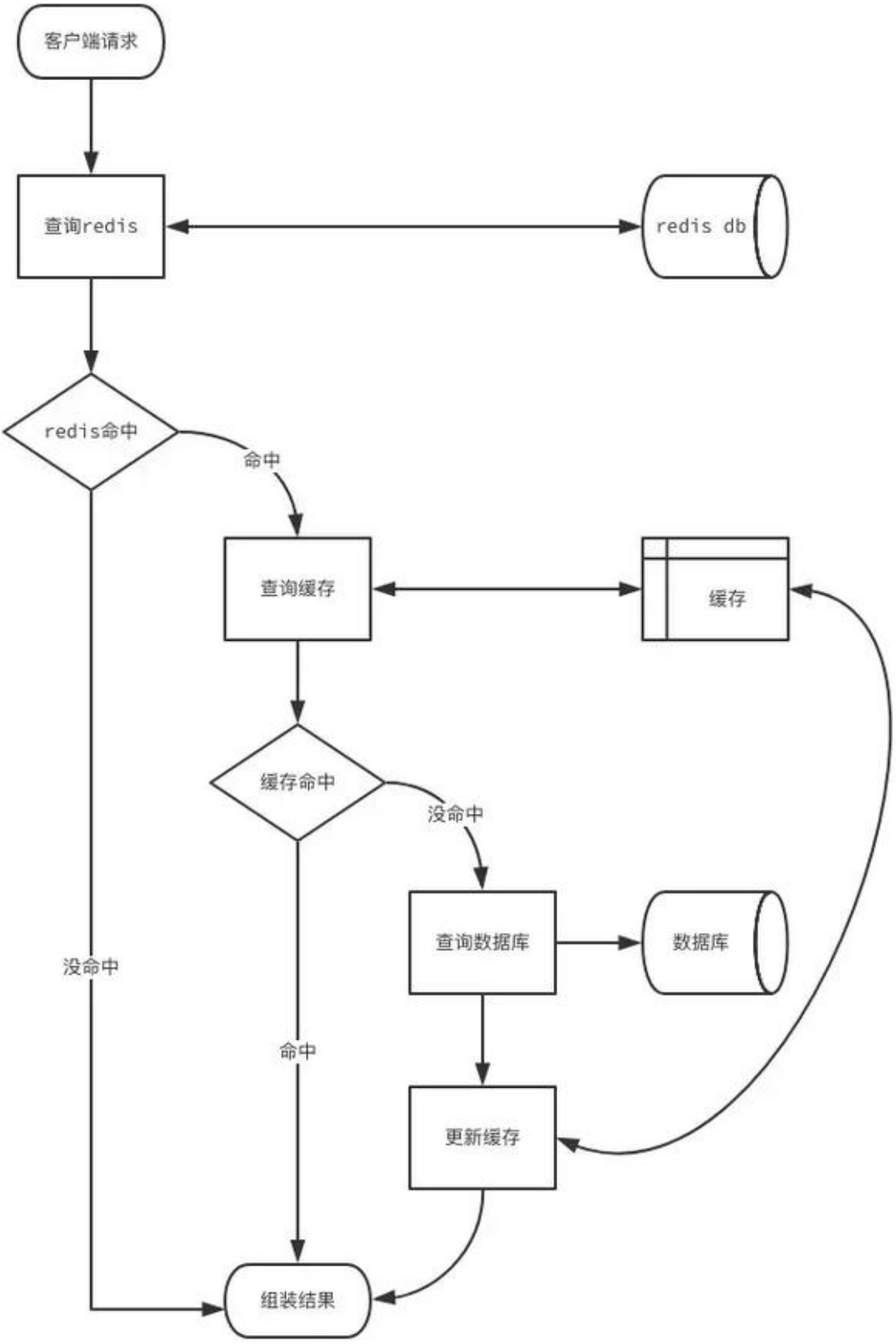
当一个查询请求过来时，先经过布隆过滤器进行查，如果判断请求查询值存在，则继续查；如果判断请求查询不存在，直接丢弃。

这里看Bloom Filter。
我们先看看一般业务缓存流程：



先查询缓存，缓存不命中再查询数据库。 然后将查询结果放在缓存中即使数据不存在，也需要创建一个缓存，用来防止穿库。这里需要区分一下数据是否存在。 如果数据不存在，缓存时间可以设置相对较短，防止因为主从同步等问题，导致问题被放大。

这个流程中存在薄弱的问题是，当用户量太大时，我们会缓存大量数据空数据，并且一旦来一波冷用户，会造成雪崩效应。 对于这种情况，我们产生第二个版本流程:redis过滤冷用户缓存流程



我们将数据库里面，命中的用户放在redis的set类型中，设置不过期。 这样相当把redis当作数据库的索引，只要查询redis，就可以知道是否数据存在。 redis中不存在就可以直接返回结果。 如果存在就按照上面提到一般业务缓存流程处理。

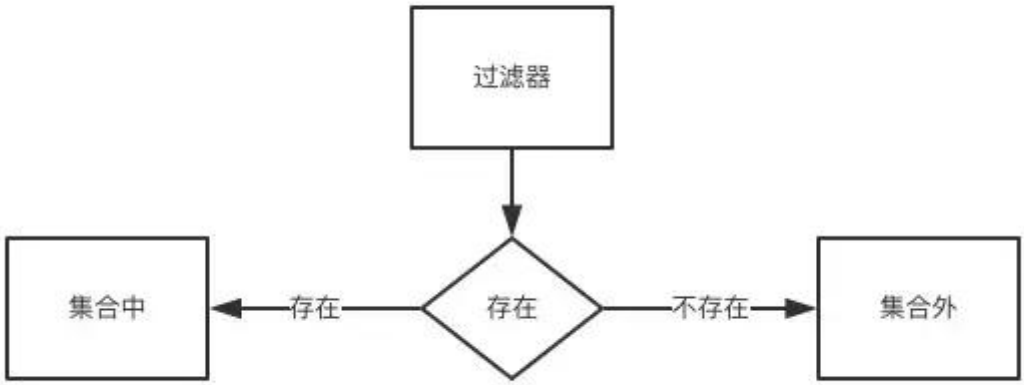
聪明的你肯定会想到更多的问题：

redis本身可以做缓存，为什么不直接返回数据呢？
如果数据量比较大，单个set，会有性能问题？

业务不重要，将全量数据放在redis中，占用服务器大量内存。投入产出不成比例？
问题1需要区分业务场景，结果数据少，我们是可以直接使用redis作为缓存，直接返回数据。 结果比较大就不太适合用redis存放了。比如ugc内容，一个评论里面可能存在上万字，业务字段多。

redis使用有很多技巧。bigkey 危害比较大，无论是扩容或缩容带来的内存申请释放， 还是查询命令使用不当导致大量数据返回，都会影响redis的稳定。这里就不细谈原因及危害了。 解决bigkey方法很简单。我们可以使用hash函数来分桶，将数据分散到多个key中。 减少单个key的大小，同时不影响查询效率。

问题3是redis存储占用内存太大。因此我们需要减少内存使用。 重新思考一下引入redis的目的。redis像一个集合，整个业务就是验证请求的参数是否在集合中。



Redis 的并发竞争问题是什么？如何解决这个问题？了解 Redis 事务的 CAS 方案吗？

简单的讲：就是多客户端同时并发写一个 key，可能本来应该先到的数据后到了，导致数据版本错了；或者是多客户端同时获取一个 key，修改值之后再写回去，只要顺序错了，数据就错了。

而且 Redis 自己就有天然解决这个问题的 CAS 类的乐观锁方案，使用版本号进行控制，cas的思想这里就不详细说了。