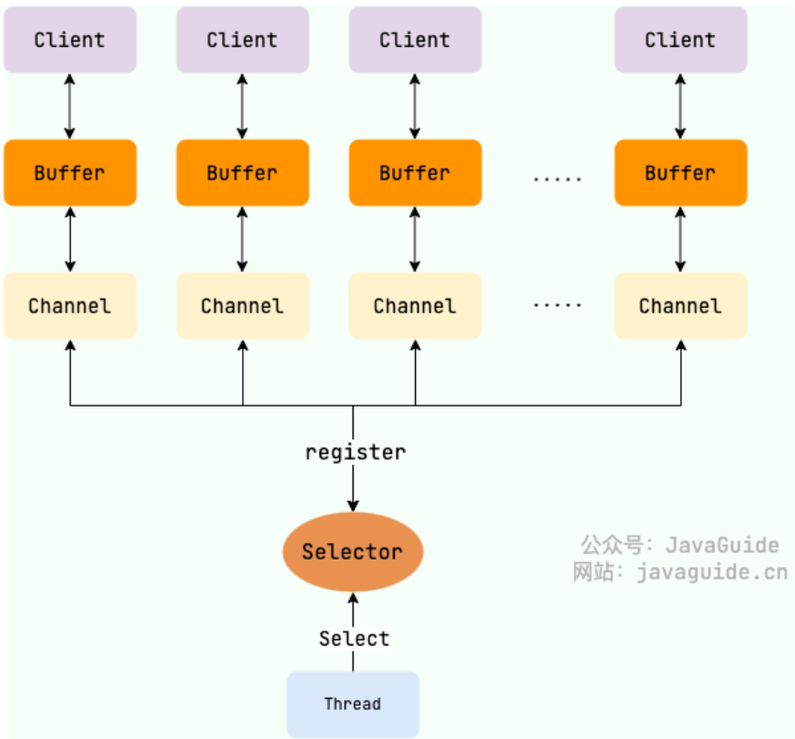
面经总结

Todo List
1. Java NIO
2. select、poll、epoll
    select 的原理和特点
    poll 的原理和特点
    epoll 的原理和特点
    区别总结
3. Java clone
4. Java treemap
5. jmap VS jstack命令
6.AOP
    Spring AOP（运行时织入）
        JDK动态代理示例
        CGLIB动态代理示例
    Aspectj（编译or类加载时织入）
    Java agent（更通用的字节码增强方式）
    Jvm SandBox

Todo List

1. Java NIO



Java NIO 的原理基于几个核心概念：缓冲区（Buffers）、通道（Channels）、选择器（Selectors）以及非阻塞 I/O。以下是这些概念的简要介绍和原理说明：

1. **缓冲区（Buffers）**：

NIO 读写数据都是通过缓冲区进行操作的。读操作的时候将 Channel 中的数据填充到 Buffer 中，而写操作时将 Buffer 中的数据写入到 Channel 中。
缓冲区是 NIO 中的一个关键概念，它是数据的容器，可以是字节、字符或其他类型的数据。缓冲区不是直接与流直接交互，而是作为数据的临时存储区域，用于在读取和写入操作中暂存数据。缓冲区有多种类型，如 ByteBuffer、CharBuffer 等，每种类型都有其特定的操作和行为。缓冲区具有容量（capacity）、限制（limit）和位置（position）三个属性，这些属性共同定义了缓冲区中数据的存储和访问方式。

2. **通道（Channels）**：

   Channel 是一个双向的、可读可写的数据传输通道，NIO 通过 Channel 来实现数据的输入输出。通道是一个抽象的概念，它可以代表文件、套接字或者其他数据源之间的连接。
   通道是 NIO 中用于数据传输的对象。与传统的流（Streams）不同，流是单向的（只能读或只能写），而通道可以是双向的，即可以同时进行读写操作。通道可以是文件通道（FileChannel），也可以是套接字通道（SocketChannel 和 ServerSocketChannel），用于网络通信。通道是非阻塞的，这意味着它们可以同时处理多个连接，而不会因为一个阻塞操作而影响其他操作。

3. **选择器（Selectors）**：
   选择器是 NIO 中用于多路复用 I/O 的工具。它允许单个线程监控多个输入通道，判断通道是否有事件发生（如连接请求、数据到达等），并根据事件类型进行相应的处理。这种方式极大地提高了程序在处理多个 I/O 通道时的效率，因为它避免了为每个通道分配单独的线程或使用多线程模型，从而减少了资源消耗和上下文切换的开销。

4. **非阻塞 I/O**：
   NIO 的通道默认是非阻塞的，这意味着在进行读取或写入操作时，如果没有数据可用，通道不会阻塞当前线程，而是立即返回。这允许线程在等待数据的同时处理其他任务。非阻塞 I/O 是通过底层的操作系统支持实现的，它依赖于操作系统提供的异步事件通知机制。

这些概念共同构成了 Java NIO 的基础，使得 NIO 能够提供高性能的 I/O 操作，特别是在高并发和大量数据传输的场景中。通过使用缓冲区来优化数据读写，通道来进行双向通信，选择器来实现多路复用，以及非阻塞 I/O 来提高线程的利用率，Java NIO 提供了一种与传统 I/O 完全不同的高效数据处理方式。

2. select、poll、epoll

> 文件描述符（File Descriptor）是操作系统中用于标识一个打开的文件、目录、管道、套接字等文件类型的对象的非负整数。在类Unix操作系统中，几乎所有的I/O操作都是通过文件描述符来完成的。文件描述符的主要用途和特点包括：
>
> 1. **唯一标识**：每个打开的文件或资源都有一个唯一的文件描述符，即使是同名的文件，每次打开也会得到不同的文件描述符。
> 2. **抽象化**：文件描述符提供了一个抽象层，使得不同类型的文件和资源可以通过统一的接口进行操作。例如，你可以对一个普通的文件和一个网络套接字使用相同的读取和写入系统调用。

3. **I/O 操作**：文件描述符是执行I/O操作的关键。通过文件描述符，应用程序可以使用如 `read`、`write`、`close` 等系统调用来读取数据、写入数据和关闭文件。
4. **标准I/O**：文件描述符也用于标准输入（stdin，通常为0）、标准输出（stdout，通常为1）和标准错误（stderr，通常为2）。这些是程序运行时默认打开的文件描述符，分别用于程序的输入、输出和错误信息的输出。
5. **多路复用**：在使用如 `select`、`poll` 或 `epoll` 等I/O多路复用技术时，文件描述符用于指定哪些文件或套接字需要监控，以及监控它们的哪些事件（如可读、可写、异常等）。
6. **继承性**：在创建子进程时，父进程可以决定哪些文件描述符要传递给子进程。这允许子进程访问父进程打开的资源。
7. **资源管理**：通过文件描述符，操作系统可以跟踪和管理所有打开的文件和其他资源，确保资源被正确关闭和释放。

文件描述符是操作系统提供的一种高效、灵活的资源管理方式，它使得程序员可以通过统一的接口处理各种类型的文件和资源，极大地简化了文件I/O操作的编程模型。

`select`、`poll` 和 `epoll` 都是 I/O 多路复用的技术，它们允许单个线程同时监控多个文件描述符（通常是套接字），以便在这些文件描述符中的任何一个变得可读或可写时得到通知。下面是它们的原理和区别：

select 的原理和特点

- **位图机制**：`select` 使用三张位图（读、写、异常），分别对应不同类型的事件。应用程序首先将需要监控的文件描述符集合设置到这些位图中，然后调用 `select` 系统调用。`select` 会阻塞等待，直到某个文件描述符就绪或者超时。
- **效率问题**：`select` 的效率随着监控的文件描述符数量增加而降低，因为它需要复制文件描述符集合到内核和用户空间，并且每次调用都会检查所有文件描述符，不管它们是否就绪。

poll 的原理和特点

- **pollfd 数组**：`poll` 使用一个 `pollfd` 结构体数组来监控每个文件描述符的状态。这个数组在用户空间，`poll` 函数会轮询这个数组，直到某个文件描述符就绪或者超时。
- **改进**：与 `select` 相比，`poll` 没有文件描述符数量的限制，因为它不依赖于位图，而是直接在用户空间操作。但是，`poll` 仍然需要遍历整个数组来查找就绪的文件描述符，这在大量文件描述符时效率仍然不高。

epoll 的原理和特点

- **事件通知机制**：`epoll` 是 Linux 特有的 I/O 多路复用技术，它使用事件通知机制，只返回那些真正发生的事件的文件描述符，而不是像 `select` 和 `poll` 那样轮询所有文件描述符。
- **效率和扩展性**：`epoll` 在处理大量文件描述符时效率更高，因为它避免了不必要的遍历。`epoll` 有两种工作模式：`EPOLL_LT`（水平触发）和 `EPOLL_ET`（边缘触发）。边缘触发模式可以进一步提高效率，因为它只在状态发生变化时通知，而不是像水平触发模式那样只要条件满足就通知。
- **内核与用户空间**：`epoll` 通过 `epoll_ctl` 函数管理监控的文件描述符，并且维护一个事件列表，这个列表存储在内核空间，减少了用户空间和内核空间之间的数据复制。

区别总结

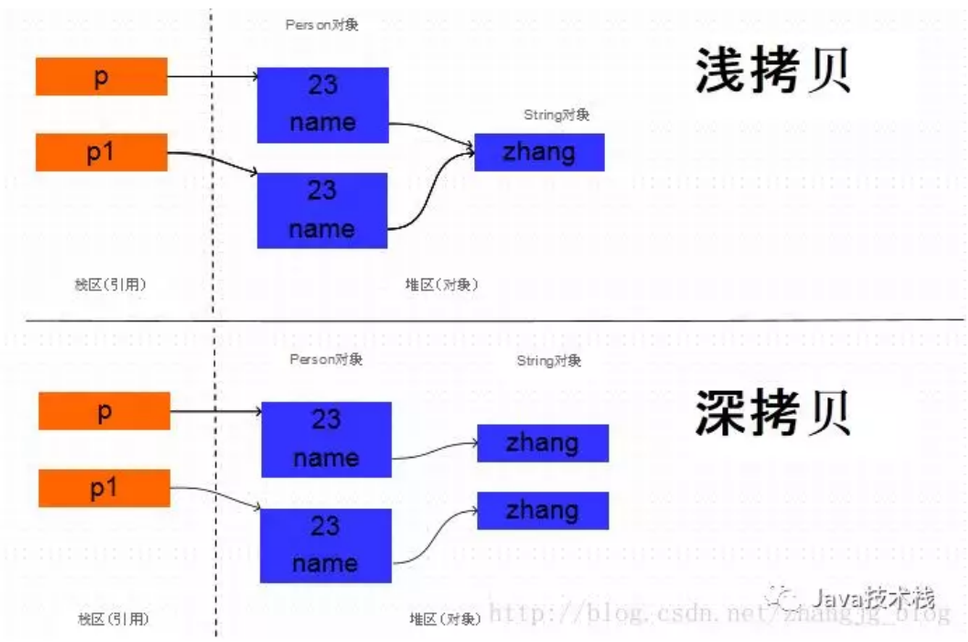
- **性能**：`epoll` 在处理大量并发连接时性能最佳，`poll` 次之，`select` 性能最差。
- **文件描述符限制**：`select` 有文件描述符数量的限制，`poll` 没有硬性限制，但实际应用中受内存限制，`epoll` 支持的文件描述符数量远超 `select` 和 `poll`。
- **事件处理方式**：`select` 和 `poll` 都是轮询机制，`epoll` 使用事件通知机制，只在事件发生时通知应用程序。
- **内核与用户空间交互**：`epoll` 减少了内核与用户空间的交互次数，因为它避免了每次调用都复制文件描述符集合。

在选择使用哪种技术时，需要根据应用程序的具体需求和运行环境来决定。对于需要处理大量并发连接的高性能服务器，`epoll` 通常是首选。

3. Java clone

https://www.cnblogs.com/javastack/p/12759352.html

调用clone方法时，分配的内存和源对象（即调用clone方法的对象）相同，然后再使用原对象中对应的各个域，填充新对象的域， 填充完成之后，clone方法返回，一个新的相同的对象被创建，同样可以把这个新对象的引用发布到外部 。



clone方法执行的是浅拷贝，**如果想要实现深拷贝，可以通过覆盖Object中的clone方法的方式来实现。**

```
static  class  Body  implements  Cloneable
{
    public  Head  head;
    public  Body()
    {}
    public  Body(Head  head)
    {
        this.head  =  head;
    }
    @Override
    protected  Object  clone()  throws  CloneNotSupportedExcepti
    {
        Body  newBody  =    (Body)  super.clone();
        newBody.head  =   (Head)  head.clone();
        return  newBody;
    }
}
static  class  Head  implements  Cloneable
{
    public   Face  face;
    public  Head()
    {}
    public  Head(Face  face)
    {
        this.face  =  face;
    }
    @Override
    protected  Object  clone()  throws  CloneNotSupportedExcepti
    {
        return  super.clone();
    }
}
public  static  void  main(String\[\]  args)  throws  CloneNotSu
{
    Body  body  =  new  Body(new  Head());
    Body  body1  =   (Body)  body.clone();
    System.out.println("body == body1 : "  +  (body  ==  body1)
    System.out.println("body.head == body1.head : "  +   (body.h
}
```

打印结果:

```
body == body1 : false
body.head == body1.head : false
```

由此可见， body和body1内的head引用指向了不同的Head对象， 也就是说在clone Body对象的同时， 也拷贝了它所引用的Head对象， 进行了深拷贝。

4. Java treemap

https://segmentfault.com/a/1190000021434060

- `TreeMap`的底层结构是红黑树，因此可以依赖红黑树的特性保证`key`有序，所有操作其实就是对红黑树的操作。
- 而因为红黑树的数据结构，所以插入、查询和删除时间复杂度都是O(logn)，但是删除和插入后可能需要调整树结构满足红黑树的规则，需要耗费性能。
- `TreeMap`的效率很高，还支持各种条件查找，甚至是范围查找和范围替换等等。

5. jmap VS jstack命令

https://juejin.cn/post/6957903936657293319

jmap：**Memory Map for Java**用于生成堆转储快照**一般称为heapdump或dump文件**。同时它还可以查询finalize执行队列、Java堆和方法区的详细信息，如空间使用率、当前用的是哪种收集器等。说简单点就是它能用来**查看堆内存信息。**

表4-3　jmap工具主要选项

选　　项	作　　用
-dump	生成 Java 堆转储快照。格式为 -dump:[live,]format=b,file=<filename>，其中 live 子参数说明是否只 dump 出存活的对象
-finalizerinfo	显示在 F-Queue 中等待 Finalizer 线程执行 finalize 方法的对象。只在 Linux/Solaris 平台下有效
-heap	显示 Java 堆详细信息，如使用哪种回收器、参数配置、分代状况等。只在 Linux/Solaris 平台下有效
-histo	显示堆中对象统计信息，包括类、实例数量、合计容量
-permstat	以 ClassLoader 为统计口径显示永久代内存状态。只在 Linux/Solaris 平台下有效
-F	当虚拟机进程对 -dump 选项没有响应时，可使用这个选项强制生成 dump 快照。只在 Linux/Solaris 平台下有效

jstack：**Stack Trace for Java**命令用于生成JVM当前时刻的**线程快照。**线程快照就是当前JVM内每一条线程正在执行的方法堆栈的集合，生成线程快照的目的通常是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间挂起等，都是导致线程长时间停顿的常见原因。线程出现停顿时通过jstack来查看各个线程的调用堆栈， 就可以获知没有响应的线程到底在后台做些什么事情，或者等待着什么资源。

6.AOP

有哪几种实现方式？

Spring AOP（运行时织入）

https://www.cnblogs.com/tuyang1129/p/12878549.html

`Spring`的`AOP`实现原理其实很简单，就是通过动态代理实现的。如果我们为`Spring`的某个`bean`配置了切面，那么`Spring`在创建这个`bean`的时候，实际上创建的是这个`bean`的一个代理对象，我们后续对`bean`中方法的调用，实际上调用的是代理类重写的代理方法。而`Spring`的`AOP`使用了两种动态代理，分别是JDK的动态代理，以及CGLib的动态代理。

Spring默认使用JDK的动态代理实现AOP，类如果实现了接口，Spring就会使用这种方式实现动态代理。`JDK`的动态代理存在限制，那就是被代理的类必须是一个实现了接口的类，代理类需要实现相同的接口，代理接口中声明的方法。若需要代理的类没有实现接口，此时`JDK`的动态代理将没有办法使用，于是`Spring`会使用`CGLib`的动态代理来生成代理对象。`CGLib`直接操作字节码，生成类的子类，重写类的方法完成代理。

JDK动态代理在内部使用反射机制来拦截代理对象的方法调用，通过实现`InvocationHandler`接口的`invoke`方法，可以拦截并自定义代理对象的方法调用。CGLIB通过字节码技术在运行时生成一个新的子类，来实现对目标类的代理。与JDK动态代理类似，CGLIB也提供了方法拦截的功能，但需要使用`MethodInterceptor`接口。

让我们通过一个简单的例子来展示JDK动态代理和CGLIB动态代理的使用。

JDK动态代理示例

假设我们有一个`Rental`接口和一个实现该接口的`Movie`类。我们将创建一个代理对象，用于记录`Movie`对象的方法调用信息。

首先，定义`Rental`接口和`Movie`实现类：

```
public interface Rental {
    void rent();
}

public class Movie implements Rental {
    @Override
    public void rent() {
        System.out.println("Renting a movie");
    }
}
```

接下来，使用JDK动态代理创建`Movie`的代理对象：

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class JDKProxyExample {
    public static void main(String[] args) {
        Rental movie = new Movie();
        Rental proxy = (Rental) Proxy.newProxyInstance(
                movie.getClass().getClassLoader(),
                new Class[]{Rental.class},
                new InvocationHandler() {
                    @Override
                    public Object invoke(Object proxyInstanc
```

```
e, Method method, Object[] args) throws Throwable {
                        System.out.println("Before method cal
l");
                        Object result = method.invoke(movie,
args);
                        System.out.println("After method cal
l");
                        return result;
                    }
                }
        );

        proxy.rent(); // 输出: "Before method call", "Renting
a movie", "After method call"
    }
}
```

CGLIB动态代理示例

现在，我们将使用CGLIB来创建`Movie`类的动态代理。首先，我们需要添加CGLIB库的依赖到项目中。以下是Maven的依赖配置示例：

我们创建一个`Movie`类的方法拦截器：

```
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

public class MovieInterceptor implements MethodInterceptor {
    @Override
    public Object intercept(Object obj, Method method, Object
[] args, MethodProxy proxy) throws Throwable {
        System.out.println("Before method call");
        Object result = method.invoke(obj, args);
        System.out.println("After method call");
        return result;
    }
}
```

最后，我们使用CGLIB创建`Movie`类的代理对象：

```
import net.sf.cglib.proxy.Enhancer;

public class CGLIBProxyExample {
    public static void main(String[] args) {
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(Movie.class);
        enhancer.setCallback(new MovieInterceptor());
        Movie proxy = (Movie) enhancer.create();
```

```
        proxy.rent(); // 输出: "Before method call", "Renting
a movie", "After method call"
    }
}
```

在这个例子中，我们首先定义了一个`Rental`接口和一个`Movie`实现类。然后，我们分别使用JDK动态代理和CGLIB动态代理创建了`Movie`的代理对象，并在代理对象的方法调用前后添加了日志输出。通过比较两个示例，你可以看到JDK动态代理和CGLIB动态代理在代码实现上的差异，以及它们如何用于创建代理对象并拦截方法调用。

Aspectj（编译or类加载时织入）

一种就是我们常见的**基于java注解**切面描述的方法，这种方法兼容java语法，写起来十分方便，不需要IDE的额外语法检测支持；另外一种是**基于aspect文件**的切面描述方法，这种语法本身并不是java语法，因此写的时候需要IDE的插件支持才能进行语法检查。

三种织入方式：

https://www.javadoop.com/post/aspectj

1. compile-time：这是最简单的使用方式，在编译期的时候进行织入（即在编译的时候先修改了代码再进行编译），这样编译出来的 .class 文件已经织入了我们的代码，在 JVM 运行的时候其实就是加载了一个普通的被织入了代码的类。可以运用`aspectj-maven-plugin`插件来实现编译。
2. post-compile：编译后织入，增强已经编译出来的类，如我们要增强依赖的 jar 包中的某个类的某个方法。可以运用`aspectj-maven-plugin`插件来实现编译。
3. load-time：在 JVM 进行类加载的时候进行织入。这种方式不需要使用AspectJ编译器，而是在类加载到JVM时，通过自定义的类加载器来实现。AspectJ的weaver会在类加载时读取类文件，应用切点和通知，然后生成新的类文件来替换原始的类。

Java agent（更通用的字节码增强方式）

Java Agent是通过`-javaagent`参数在JVM启动时或运行时附加到JVM上的一个组件。它使用JVM Tool Interface (JVMTI)来监听和修改JVM的行为。Java Agent可以在以下时机进行类文件的修改：

- **类加载前premain**：在类文件被加载到JVM之前，可以拦截并修改字节码。通过在 java 的启动参数中添加 -javaagent:/jar包路径来进行字节码增强，随着java进程启动而启动。
- **类加载后agentmain**：对于已经加载到JVM中的类，可以在其被重新定义（redefine）时修改字节码。在运行时通过JVM Attach机制进行字节码增强。

Jvm SandBox