Programming Clouds with GreatFree:

An Unwrapped Open Source Methodology

Bing Li

To my mother, Huling Xue

Chapter 1 Introduction to GreatFree

Abstract

This paper introduces a series of APIs and idioms in Java SE (Java Standard Edition), GreatFree, to program large-scale distributed systems from scratch without adopting any third party frameworks. When programming with GreatFree, developers are required to take care of rather than be invisible to most of the implementation issues in a distributed system. It not only strengthens developers' skills to polish a system but also provides them with the techniques to create brand new and creative systems. However, taking care of many such issues is a heavy load because of the low-level of Java SE. To alleviate the burden to program with Java SE directly, GreatFree provides numerous APIs and idioms in Java SE to help programmers resolve indispensable distributed problems, such as communication programming, serialization, asynchronous and synchronous programming, resource management, load balancing, caching, eventing, requesting/responding, multicasting, and so forth. Additionally, as an open source tool to program, developers are able to strengthen their systems through not only adjusting GreatFree parameters but also upgrading GreatFree APIs and idioms themselves. According to the current intensive experiments, it is convenient for developers to program an ordinary or a large-scale distributed system from scratch with GreatFree.

1. Introduction

The paper exhibits an open-source programming tool, GreatFree, to assist developers to program large-scale distributed systems from scratch. Based on Java SE, GreatFree includes a series of APIs and idioms to ease the programming procedure. Nowadays most developers avoid programming a distributed system directly because of many issues to be resolved. In contrast, it is more convenient to configure mature open sources and commercial frameworks than to program with generic low-level programming languages like Java SE. However, in many specific cases it needs to implement a system by programming rather than configuring. Professional developers are required to be competent with dealing with complicated problems in a distributed computing environment through programming.

To implement a distributed system by programming, it encounters many implementation barriers, such as communication [1], serialization [1], asynchronous and synchronous management [2][3], resource management [4], load balancing [5], caching [6], eventing and requesting/responding [7], multicasting [8], and so forth. It is tough to implement such a system with generic fundamental programming languages like Java SE. Nevertheless, programming a distributed system from scratch is required in many specific cases. First of all, a brand new creative system requires developers to program the low-level models themselves to construct a self-contained system other than to resolve high-level problems only. The goal can never be achieved with the approaches of configuring and customizing. Second, a system needs to be flexible enough to be customized for a specific issue by programming. Though it

is seldom that all of the distributed features be implemented from scratch, a limited change is frequently required in a system. It is convenient for programmers to handle that only if all of the code is open and easy to be changed. Third, for professional developers, it is required for them to be competent with the task of programming a complicated system without the assistance of mature frameworks. Fourth, for security reasons, some systems, such as a creative one in a new environment, are forced to implement from an initial level. For those considerations, a bunch of APIs and idioms, which is named GreatFree, are proposed to assist developers to program distributed systems, specially the large-scale ones.

The methodology GreatFree represents encourages proficient developers to program their own distributed systems rapidly rather than to configure a mature framework. It provides developers with a new programming tool that focuses on the large-scale distributed computing environment. To achieve the goal, it prevents developers from heavy loads of programming a complicated system with fundamental programming languages, such as Java SE (Java Standard Edition) [9]. For that, it comes up with a bunch of APIs (Application Program Interfaces) and design patterns as idioms to solve the specific distributed issues such as client/server [10] and peer-to-peer models [10], distributing eventing [7] and polling [7], resource management [4], distributed concurrency and synchronization [2][3], distributed multicasting [8], distributed clusters [10] for computing and memorizing, and so forth. With the support of the GreatFree APIs and idioms, it is convenient for developers to program a large-scale distributed system from scratch. In brief, GreatFree is a programming environment that resides between the mature configurable systems and the naked generic programming languages. Using it, a high quality distributed system can be conveniently programmed from scratch.

When using GreatFree, it is required for developers to keep most of the distributed issues in mind. That is different from the traditional opinions of software engineering, which emphasizes the goal to hide all of the details of low-level systems such that developers are invisible to the specific computing environments. They need only to change limited parameters or configurations following tutorials and guiding books without worrying about what happens underlying high-level applications. On the contrary, the methodology of GreatFree proposes that most important techniques should be discernible to developers such that they need to resolve them by programming with relevant APIs and patterns. Developers are required to compose those stuffs to form an arbitrary application. Although mature systems minimize developers' effort, they also lower developers' abilities such that they can hardly deal with changes if they do not program for a long-term. Such developers without sufficient knowledge might be able to customize a high quality system by configuring mature systems. But they will never be able to implement a creative system in a new domain. Therefore, using GreatFree, developers always need to learn and keep in mind the relevant technologies. And then, when they would like to implement a brand new system, the methodology of GreatFree become their potential silver bullet.

On the other hand, GreatFree is different from the naked generic programming languages like Java SE. First of all, it contains a series of domain-specific APIs and idioms, which focus on the area of distributed systems, especially the large-scale ones whereas a generic programming language is required to deal with all of the issues in any domains. The convenience of GreatFree is that developers' programming loads

are reduced magnificently for its well-defined APIs and idioms. Although they need to be aware of the issues of distributed systems, they are not required to implement them using Java SE. On the contrary, they just need to program in a composing manner with the APIs and idioms from GreatFree. More important, proficient programmers are encouraged to revise the code of the APIs and idioms because all of the source code of GreatFree is open [11].

For the distinguished character of GreatFree, developers of the programming tool need to learn the distributed issues at first. To be invisible to the problems, they must have no idea which APIs and idioms should be chosen to program with a composing style. Fortunately, the programming procedure looks like constructing a tower with high quality building blocks rather than sands and stones directly.

The main contributions of the paper are summarized as follows.

As a methodology, a developer needs to learn the core techniques of the system they are programming rather than to configure a mature system without knowing anything about its internal principles. This is especially important to design a brand new and creative system.

To lower the burden of developers' effort, open source based APIs and idioms are effective in terms of its high quality and high changeability.

According to the above opinions, a series of APIs and idioms are proposed to help developers program a large-scale distributed system.

2. Related Work

In the domain of software engineering, to lower the burden to implement a distributed system, many mature frameworks are put forward as open sources [12][13][14][15] or commercial products [16] for specific application environments. Thus, developers are not required to program but configure those systems to fulfill their requirements. In addition, although some generic programming languages [9] are strengthened gradually to lower the cost to program, their distances to a distributed system are still too long to be reached by coding directly for most programmers. Finally, design patterns [17][18][19][20][21][22] are also believed to be one approach to implement a complicated system like a distributed one. However, until now, most design patterns are not the code in specific languages to be reused conveniently. Although idioms, as small-scale design patterns, intend to resolve the issue, a comprehensive solution of idioms is not available in the domain of distributed systems.

2.1 Configuring Mature Systems

Nowadays when implementing a distributed system, a common phenomenon is that developers get accustomed to downloading existing mature open sources or commercial systems rather than programming themselves line by line. No matter whether a system is complicated enough like a so-called cloud [15] or even a fundamental one like a client/server model [10], most developers give up implementing a system through programming. Although each of them must take the

relevant classes or trainings in universities or other schools, they become scared about programming those systems themselves.

The primary reason is due to the fact that those mature systems could fulfill their requirements. For example, to set up a Web site, Tomcat [12] is one of the popular choices as a Web server. Even for a gaming system, a Web server like Tomcat is suitable to support its centralized information exchanging among users. To initiate an e-commerce online system, developers prefer the Java EE (Java Enterprise Edition) [13] to the generic programming language, Java SE (Java Standard Edition) [9], since the former one hides most details of distributed techniques for enterprises from developers.

In addition, it is difficult to program high quality distributed systems by programming from scratch. Developers have to be aware of numerous technical details, such as network communication [1], serialization [1], asynchronous and synchronous programming [2][3], caching [6], eventing and requesting/responding [7], resource management [4], load balancing [5], multicasting [8], and so forth. All of the issues turn out to be a heavy workload to program.

Finally, most mature systems are adaptable to subtle changes in their specific environments. Those systems allow developers to configure in the event that some requirements cannot be fulfilled by default. They provide developers with configuration files in an XML [23] format such that they can set up their own preferences through modifying relevant parameters. For example, to update the Solr [24] configuration file, developers are able to construct a tree-like structure to distribute searching loads. When using JBoss [25], WebSphere [26] or other application servers, business transactions [6] are required to be specified in configuration files.

In short, when the requirements are clear and the relevant mature frameworks are available to the specific environment, it is reasonable to select an appropriate commercial framework or download an open source system to accomplish the goal rapidly.

2.2 Disadvantages of Configuring Mature Systems

However, besides the advantages of mature frameworks, developers should be aware of the disadvantages when using them to implement their systems. First of all, all of those mature systems are specific to their respective particular domains and their adaptability is restricted to narrow scopes. Although most existing frameworks claim they are adjustable to different environments, it is always an impossible mission when using the systems to a domain out of their ranges. For example, no one ever believes Java EE based enterprise systems could be used to take the load of sharing online videos. If a conventional Web server, like Tomcat [12], could work efficiently to transmit high-volume data, why some browser plug-in systems, like FlashGet [27], exist? Although online Web chatting systems were popular in 1990s, they were completely replaced by instant messaging [28]. Even though instant messaging systems dominate the market of lightweight information exchanging, they are almost beat by socialized systems, like Twitter [29], eventually. Therefore, once if a

framework is designed, its adaptation is usually constrained tightly within its domain. In brief, mature frameworks can hardly adapt to the updates in their preferred computing environment.

Second, even though within the preferred specific domain for their designs, their adaptability is limited in terms of the obvious differences between configuring and programming. For example, in Solr, using its configuration files, developers are offered the privilege to customize a hierarchical distributed structure to achieve the goal of load balancing. Indexed data is transmitted among the nodes by the protocol of HTTP [30]. Each node in the structure has to pull its parent node periodically to obtain latest data. However, the drawback is apparent in the case. First, developers are not free enough to construct an arbitrary topology other than the hierarchical one. Moreover, only the protocol of HTTP is used between the nodes. The periodically pulling is not efficient as an eventing and streaming based protocol [10], especially when the timing issue is critical in a particular application. Because of the limitations of the Solr, developers can do nothing. Only programming can achieve the goal to upgrade underlying protocols and algorithms. Mature systems are not adaptable enough to fulfill the requirements. Compared with programming, the flexibility upon configuring is believed to be extremely limited.

Third, developers probably rely on those systems such that it is difficult for them to keep and improve their skills. Using them, developers need only to describe requirements and change configurations. It can be achieved with few programming experiences. For example, to implement an e-commerce system with a Java EE application server [13][26], only business logic is required for developers to write code in the object-oriented model [17]. It is unnecessary for them to worry about threading [2][3], data transmission [1], synchronizing [6], resource management [4], state management [6], transaction management [6] and so forth. From the perspective of software engineering, it lowers developers' cost and speed up the system development. However, those developers always lose the indispensable expertise during the procedure. For some beginners, they even ignore the required knowledge and skills. After a period, it is impossible for them to be competent with implementing a creative system from scratch.

Fourth, sometimes those mature systems are easily misused and excessively utilized in inappropriate environments. The problem often happens when developers deploy those systems to unsuitable domains. Even though they become aware of the potential problems, the convenience of deploying and configuring attracts them to avoid the troubles of programming. Sometimes such a misusing might work, but that is usually a high cost and cumbersome solution. For example, to set up a peer-to-peer (P2P) [10] model in a distributed environment, developers would rather install Tomcat [12] on each node as the server to receive data and then find a HTTP client [30] for each of them to send data. The solution works since data can be sent from the client to the server by the HTTP request and any nodes are able to play the role of a server to receive remote requests and notifications. However, Tomcat is a huge stuff itself. It contains many other irrelevant components that are never used in the P2P system, such as JavaServer Pages [31]. Moreover, Tomcat consumes a large portion of resources as a Web based application platform on a server that has rich computing resources. Thus, it influences the quality of the P2P system, especially when it is deployed to the devices whose resources are limited. In most time, a P2P system

needs to deal with a large number of heterogeneous computing devices. Most of them lack computing resources. Therefore, Tomcat is unsuitable to the case. To some extent, the above solution is not bad although it is far from perfect. The most typical mistake is that developers would rather use Web servers anywhere for communications. The problem is often seen in the case that developers lack rich experiences, especially who scares about the TCP programming and the concurrency programming.

Finally, there are always new domains and novel ideas that are not covered by those mature systems. A mature system emerges only when the specific application environment becomes dominant and its primary requirements keep steady for a long period. For example, the so-called application server is usually suitable to the centralized mission-critical enterprise distributed environment to support e-commerce. Hadoop is proposed for the high concurrency and large-scale distributed clusters that primarily deal with high volume read-only data. Fortunately, with the progress of the computing world, new domains or computing environments always come out. The mobile Internet is apparently such an instance. Many fancy applications are to be implemented to accommodate the specific context. For example, when developers are designing an application to present Web pages on a smart phone, they have to take into account to accomplish the task with programming rather than simply to embed a browser into their applications. Thus, many Web sites nowadays design their own clients on smart phones and encourage users to access their sites using the specially customized clients rather than using traditional generic browsers. Because of the specific designs, the clients provide users with high quality accessing experiences. In addition, when new ideas are available about a specific topic, traditional approaches must be out of date. It is required for developers to keep programming to implement additional modules and integrate them with others. For instance, when a new routing algorithm [32] is available for a P2P system [33], it is impossible to exploit it by configuring existing systems. Developers have to implement the algorithm by programming.

In brief, although mature software systems are convenient for developers to deal with requirements in a particular computing environment, for a software developer, it is required to keep in mind that programming instead of configuring is the only way to raise their ability to deal with all kinds of difficult problems. In some specific scenarios, programming line-by-line is mandatory to the success of a system, especially when it is a brand new one in terms of scientific or business creations.

2.3 Design Patterns

For the problem of programming languages, some senior researchers and engineers propose design patterns [17][18][19][20][21][22] to assist junior programmers. Design patterns aim to provide developers with a bunch of mature solutions written in pattern languages [21][22] to resolve recurring problems in various domains, such as the object-oriented (OO) programming context [17], the distributed computing [18] and the enterprise systems [18]. The patterns are proved to be effective. For example, to the issues of OO programming, patterns like creational, structural and behavioral [17] ones lower developers' effort to reinvent them themselves through a long-term practice. Unfortunately, those stuffs retain to be generic in all of the above domains.

Thus, design patterns are not regarded as the final solution that can be reused rapidly to solve the problems of specific computing environments. Instead, they need to spend high effort to be aware the contexts where those abstract patterns are suitable. After that, those patterns have to be transformed into specific code. During the process, programmers are never isolated from the implementation details. It is still a tough job. In the domain of distributed systems, the burden becomes severely heavier.

For a particular type of patterns, idioms are defined as the ones written in specific languages. If so, the effort to program with them must be low. However, a comprehensive solution to the topic of distributed systems is not available to the best of my knowledge. In the Java world, such a solution is also almost unavailable. Even though some exists, it is still fine for developers to taste the solution of GreatFree and make a better choice.

2.4 Programming From Scratch

Most developers who lack rich experiences are scared about programming directly, especially when implementing a distributed system from scratch. Programming from scratch is designated as the developing approach through which developers take into account most of the specific issues of a particular application domain and solve them by coding themselves rather than importing existing modules from third parties. It seems that the approach violates the convention of software engineering that encourages developers to reuse mature legacy code and systems. In fact, that is not the truth all the time.

In the following three cases, programming is still required. First, reusable legacy resources are unavailable in a specific domain. Second, changes are required to fit a particular situation. Third, new ideas or new environments emerge such that it is required to come up with a relatively independent system. To deal with the issues of the above cases, programming becomes the indispensible skills for developers.
The primary reason that developers dislike programming from scratch is that most languages, such as Java SE, are generic to various application domains such that it is short of effective modules, fully grown APIs and compelling patterns to deal with specific issues. Even though developers grasp all of the details of a language and learn the requirements and solutions of the domain, it still takes them a high cost to program a high quality system in a specific area directly.

2.5 Cloud Programming Languages

Scala [39] is a famous programming language for clouds. It claims it provides more concise expressions when coding. It also provide static typing, lightweight syntax, object-oriented and functional languages. The most important updates that are potentially related to distributed computing is the technology called the actors [40], which is an improvement to Java concurrency technology from Akka [40]. However, it is still far from a straightforward technique support to the topic of distributed systems programming. Compared with GreatFree, most of essential issues are ignored by Scala. Although it is said some cloud systems are developed by Scala, it does not mean that Scala itself is a qualified programming language for the specific domain of cloud-based distributed systems.

Go [41] is another well-known progress, which is achieved by engineers of Google, in the form of programming language for cloud-based distributed systems. It claims that traditional languages, such as C and C++, are not satisfied with the requirements to develop distributed systems in terms of the lack of the supports for the large scale and concurrency. However, according to the latest version of Go, it still looks more like a generic programming language than that for distributed systems over the Internet. Although it provides new concurrency mechanisms such as Goroutines [41] and Channels [41], they are far from the specific design for cloud-based distributed systems. Furthermore, all of issues covered by GreatFree are not taken into account either.

2.6 Wrapped Open Source Solutions

Most distributed application frameworks claim they support open source. The new proposed languages such as Go [41] and Scala [39] also encourage open source. However, each of them separate their open source from the development interfaces for developers. When developing applications with frameworks, programmers work in a new context, which are either script languages [13][17][24][25][26] or configuration files [12][23][24][25][26]. Both of them are independent of the underlying source code. At least, when developers implement their applications, they do not need to care about the code. For the new programming languages, like Go and Scala, programmers work with their simplified syntax or improved concurrency without being aware of the open source. Because of the segregation, it results in the situation that makes it difficult for developers to revise the open source. Even though their open source is organized in a developer-friendly way, it contains excessive code that implements the separation or the syntax processing, which has nothing to do with distributed computing. It is tough for them to find the code to change so as to meet their upper level requirements. In fact, only a limited number of developers are interested in doing that. Most people are afraid of the task. This is the reason why the current solutions are called the wrapped open sources.

3. The Goals of GreatFree

GreatFree is made up with a series of APIs and idioms to overcome the difficulties of programming large-scale distributed systems. It believes the reusable code is the most convenient tool to speed up the implementation procedure. For the domain of distributed environment, the APIs and idioms play the role of building blocks to deal with most common problems. Those components are neither conceptual models like regular design patterns for developers to refer to nor black boxes like mature frameworks for configuring and customizing. Rather, the solution of GreatFree turns out to be constructible and revisable high quality resources for programming. For that, developers are required to be familiar with the relevant knowledge to come up with a distributed system rather than to follow a tutorial only and forget about the critical technical issues. A high quality, creative and flexible system cannot be implemented unless programmers understand the essences and details. Meanwhile, developers' effort is still lowered with the APIs and idioms. Inspired by developers' knowledge, the components can take the role to program a system conveniently in terms of their high readability, constructability and changeability.

3.1 Specific Code is the Silver Bullet

Specific code is the silver bullet to speed up the development of a self-contained system compared with other resources like design patterns and frameworks. Although they cover common solutions in either the manner of object-oriented programming [17] or even in distributed environments [18], design patterns belong to conceptual models for references rather than the ones that can be embedded into an existing system directly for execution. On the other hand, a mature framework is an established system for customization instead of a self-contained one in which each detail is visible to developers.

Specific code is the reusable program that is proved to resolve one typical issue in a high quality state in a particular computing environment. Once if a bunch of such code is available, it is convenient for developers to either weave them to their own programs following straightforward patterns or reconstruct them to implement other systems further. During the procedure, developers are only required to transform the code to their own particular programs in accordance with the specific code as samples. For the maturity of the specific code, the transforming is similar to a mapping process rather than the conventional one to learn requirements, propose solutions, design and test programs. Developers take the task without considering the details of code themselves in most cases such that the effort is apparently lower than programming from scratch.

3.2 Knowledge of Distributed Systems is Required

GreatFree provides a series of APIs and idioms to assist developers to implement a self-contained distributed system rapidly. Different from the transparency perspective of the traditional software engineering, GreatFree developers are required to learn the specific issues of the system they are implementing. Although it is unnecessary to resolve the problems by programming themselves, they ought to be aware of what components are needed, where to place them in the program and what effects the components take. As a common sense, it is unreasonable that developers know little about the principles of the system they are implementing. As a matter of fact, the programming environment supported by GreatFree is not a black box to hide details from programmers. Instead, it is the composing-enabled open source for them to follow and speed up the distributed system development. Only then, they are able to understand what they are doing such that it is possible for them to raise the quality of their systems and improve GreatFree through revising the open source.

3.3 Changeable APIs and Idioms

GreatFree is an open source development tool that consists of a series of APIs and idioms. Although it is tested, it does not mean that the exact solutions are not replaceable and changeable. With the support of the idioms, the overall structure of the system is stable and developer-friendly. However, it allows programmers to update the internal algorithms, i.e., the APIs, without ruining the code organization upon GreatFree idioms. For example, developers could upgrade the issues like

resource management, thread management, multicasting approaches, remote eventing and pulling and so forth. In addition, although the system implemented by GreatFree is scalable, it does not guarantee that it fits in all of the heterogeneous distributed environments. It encourages developers to follow the current version and make changes for their own requirements in either the APIs or the idioms.

4. The GreatFree Software Development Methodology

The methodology of GreatFree software development believes in the creed that software developers should retain the indispensable knowledge in the domain they work on. Based on the opinion, it is still necessary and feasible to establish a rapid software development tool, which support rapid development. Furthermore, the primary components ought to be visible to developers such that they are aware of the knowledge they retain. To lower the effort, the relevant techniques have to be constructed in unambiguous APIs for invocation and highly-patterned idioms for composition. When programming in practice, developers can either perform simplified behaviors to develop upper level applications or navigate into open source with their knowledge to upgrade underlying code. For the above considerations, an unwrapped open source tool that is domain-specific, technique-discernable and APIs-highly-patterned is an appropriate choice for developers to work with.

4.1 Retaining Knowledge

The methodology of GreatFree believes that qualified developers should be familiar with specific domain knowledge when developing software for one particular computing environment. Although there are many existing frameworks to minimize their effort by hiding them from underlying techniques in that domain, it should be preferred for them to uncover the wrapper and learn the details as much as possible. That opinion can never be the most efficient way to develop software rapidly enough, but it helps them be independent of the frameworks. Thus, once if they encounter a new environment, they can program with generic programming languages to implement their applications. In addition, if they find out that some drawbacks of the frameworks, they can improve them with their skills. At least, they can make a proper choice from those frameworks to help them implement applications in one particular computing environment without misuse or excessive use. If a development tool is properly designed, programmers can deal with new circumstances conveniently with their knowledge when no one ever handles.

4.2 Rapid Development

The methodology of GreatFree urges that a rapid development tool is always preferred although it encourages developers to retain knowledge about the computing environment they work on. Since it is impossible to implement an application from the bottommost level, the goals of most methodologies of software engineering are identical, i.e., lowering developers' effort and speeding up the development procedure. To the extreme extent, some researchers focus on the goal of automated code generation to replace the role of software developers. Usually, a suitable solution is to provide a script language for developers to model their requirements without

taking care of underlying techniques. It is feasible since developers are required to take into account modeling languages only and sometimes they just need to configure other than program with script languages. However, GreatFree does not accept the opinion to minimize developers' cost in any contexts since such a solution degenerates developers' abilities if they work with the tools for some time. Additionally, it loses the flexibility since developers can hardly make any changes on those frameworks. In contrast, GreatFree claims that a more proper rapid development tool is the one that keeps two balances, i.e., the balance between lowering developers' effort and keeping their skills and the one between speeding up the development procedure and designing for change. For the first one, domain-specific techniques should be not only discernable but also highly-patterned. For the second one, the source code should be open and unwrapped.

4.3 Highly-Patterned Unambiguous Building Blocks

The methodology of GreatFree believes a software development environment should not always hide developers from underlying techniques. Instead, domain-specific techniques ought to be uncovered for developers. However, it is also unreasonable that all of those techniques are naked without any decorations. To speed up the development procedure, those techniques should be discernable and unambiguous such that developers' can invoke and compose them conveniently according to domain-specific knowledge they retain. For that, complicated implementations are encapsulated by straightforward APIs for invocation. Moreover, those APIs are weaved together by highly-patterned idioms for composition. In brief, properly designed APIs and idioms are the primary components for developers to work with. When implementing applications with them, developers only need to program with simplified behaviors repeatedly following their domain-specific knowledge since sophisticated techniques are turned into visible building blocks. When designing those APIs and idioms, the most important principle is that they should be consistent with the knowledge developers retain. It is not reasonable to virtualize an environment which is independent of the computing environment on which developers work.

4.4 Unwrapped Open Source

The methodology of GreatFree puts forward that developers should program with open source made up with straightforward APIs and highly-patterned idioms rather than work within an environment independent of open source. In general, open source aims to meet the requirements of technique-visible methodology such that developers can make updates when necessary. However, in practice, most developers do not touch open source even though they work with it for a long time. Although it is always tough to read code of others, the primary reason is due to the fact that most open source systems separate developers from the code. When developers implement an application with them, they need to know nothing about the open source itself but learn particular modeling languages or configuration scripts. Even though the segregation speeds up the development procedure, it conceals developers from the underlying domain-specific techniques such that it establishes a severe barrier for open source to achieve its original goal. For that, GreatFree urges the solution of unwrapped open source rather than the traditional one, i.e., the wrapped open source. When programming applications within GreatFree-driven environments, developers

work with the open source directly instead of any independent scripts or configuration circumstances. With the support of straightforward APIs and highly-patterned idioms, the procedure is rather rapid since programming is simplified as repeatable behaviors. More important, besides rapid development, the unwrapped character assists programmers to gain the advantages of keeping skills, adapting to different domains and updating code flexibly, which are missed in the wrapped open source methodologies.

API	Description
AsyncRemoteEventer	This class aims to send notifications to a remote server asynchronously without waiting for responses. The sending methods are nonblocking.
Eventer	This is a thread derived from NotificationObjectQueue. It keeps working until no objects are available in the queue. The thread keeps alive unless it is shutdown by a manager outside.
EventerIdleChecker	The class works with AsyncRemoteEventer to check whether an instance of Eventer is idle long enough so that it should be disposed.
FreeClient	This is a TCP client that encloses some details of TCP APIs such that it is convenient for developers to interact with remote servers. Moreover, the client is upgraded to fit the caching management.
FreeClientCreator	The class contains the method to create an instance of FreeClient by its IP address and port number. It extends the interface of Creatable and it is used as the resource creator in the RetrievablePool.
FreeClientDisposer	The class implements the interface of Disposable and aims to invoke the dispose method of an instance of FreeClient to collect the resource. It is used a resource disposer in RetrievablePool.
FreeClientPool	The pool, RetrievablePool, is mainly used to manage the resource of FreeClient. Some problems exist when instances of FreeClient are exposed outside since they might be disposed inside in the pool. It is a better solution to wrap the instances of FreeClient and the management on them. The stuffs should be invisible to outside. For that, a new pool, FreeClientPool, is proposed.
IPNotification	This is an object to contain the instance of IPPort and the message to be sent to it. It is used by the class of Eventer in most time.
IPPort	The class consists of all of the values to create an instance of FreeClient. For example, it is the source that is used to initialize resources in RetrievablePool.
OutMessageStream	The class consists of the output stream that responds a client. The lock is used to keep responding operations atomic. The request is any message that extends ServerMessage.
RemoteReader	The class is responsible for sending a request to the remote end, waiting for the response and returning it to the local end which sends the request.
ServerIO	The class encloses all the IO required details to receive and respond a client's requests.
ServerIORegistry	The class is used to keep all of the ServerIOs, which are assigned to each client for the interactions between the server and the corresponding client. This is a management approach for those instances of ServerIOs.
ServerListener	The class acts as the listener to wait for a client's connection. To be more efficient, it involves a thread pool and the concurrency control mechanism in its internal mechanism.
SyncRemoteEventer	The eventer sends notifications to remote servers in a synchronous manner without waiting for responses. The sending methods are blocking.

Table 1.1 GreatFree Remote Interactions

5. The GreatFree APIs

GreatFree is comprised of two portions, the APIs and the design patterns as idioms. GreatFree APIs attempt to provide developers with numerous mature encapsulated programs to resolve most distributed problems. The portion of GreatFree APIs covers five areas, including remote interactions, resource reusing, concurrency implementation and control, multicasting and some other utilities. It helps developers construct their systems without taking care about the details of each solution.

5.1 Remote Interactions

The APIs for remote interactions aim to support developers to accomplish four types of remote interactions tasks, including establishing remote connections, sending

synchronous/asynchronous notifications, performing remote reading and managing remote IO resources. The APIs for remote interactions consist of 15 classes, as shown in Table 1.

5.2 Resource Reusing

Resource reusing is a critical topic since a distributed system consists of some critical resources to be saved. In Java SE, it claims that memory is maintained by the underlying platform. However, other resources have to be taken care by developers themselves. GreatFree includes a bunch of APIs in 16 classes to ease the task, as listed in Table 2. They are roughly divided into three categories, i.e., pooling, caching and relevant interfaces.

API	Description
Creatable	This is an interface to define a resource creator that initiates an instance of the resource in the resource pool, such as RetrievablePool. The resource derives from FreeObject.
Disposable	The interface defines a method for the disposer that collects the resource in the resource pool, such as RetrievablePool. The resource derives from FreeObject.
FreeReaderIdleChecker	The class is used to call back the method of checkIdle of the instance of FreeReaderPool.
FreeReaderPool	This class is similar to RetrievablePool. In fact, it is a specific version of RetrievablePool. First, the resource managed by the pool is FreeClient. Second, it aims to initialize instances of FreeClient for not only output but also for input.
HashCreatable	The interface defines the method to create instances that extend HashFreeObject.
HashDisposable	The interface defines the method to dispose the objects that are derived from HashFreeObject.
IdleChecker	The idle checker works with the ResourcePool to check the idle states of resources.
MulticastMessageDisposer	This is an implementation of the MessageBindable interface. It is usually used by threads that share the same multicast messages.
QueuedIdleChecker	The class aims to check periodically whether a resource is idle for a long enough period. If so, the resource needs to be disposed.
QueuedPool	The pool aims to manage resources that are scheduled by their idle lengths. The one that is idle longer has the higher probability to be reused than the one that is idle for a shorter period.
ResourceCache	This class is a cache to save the resources that are used in a high probability. It is designed since the total amount of data is too large to be loaded into the memory. Therefore, only the ones that are used frequently are loaded into the cache. It is possible that some loaded ones are obsolete. It is necessary to load new ones that are used frequently into the cached and save the ones that are out of date into the database or the file system persistently.
ResourcePool	The pool aims to manage resources that are scheduled by their idle lengths. The one that is idle longer has the higher probability to be reused than the one that is idle for a shorter period. Different from QueuedPool, this pool does not care about the type of resources. It assumes that all of resources in the pool are classified in the same type.
RetrievableIdleChecker	The class runs periodically to check whether a resource being managed in a resource pool, such as RetrievablePool, is idle enough time. If so, it collects the resource.
RetrievablePool	The class is a resource pool that aims to utilize the resources sufficiently with a lower cost. The pool is usually used for the resource of FreeClient. For each remote end, multiple FreeClients are initialized and managed by the pool. When it is necessary to interact with one remote end, it is convenient to obtain a FreeClient by the key or the initial values, i.e., the IP address and the port, which represent the remote end uniquely. That is why the pool is named the RetrievablePool.

Table 1.2 GreatFree Resource Reusing

5.3 Concurrency Implementation and Control

The issue of concurrency implementation and control is particularly critical to a distributed system since it is required to deal with potential accessing from a huge number of users and other remote clients. GreatFree contains rich solutions to the problem. 35 classes are designed to provide developers with a convenient

environment to program a high concurrency and low cost distributed system. Table 3 lists the APIs and simple descriptions of them.

To implement the concurrency, two primary situations need to be dealt with. The first one is to receive concurrent notifications and the second one is to receive concurrent requests and then generate responses concurrently. In details, each of them needs to take into account the issue of multicasting. That is, the notifications and requests/responses are performed within a large-scale environment, which consists of numerous nodes, rather than between two nodes. It proposes a couple of threading queues that encloses incoming messages and dispatchers to manage those threads for specific cases.

For the issue of concurrency control, first of all, it is embedded into the concurrency APIs. Moreover, a new class, Collaborator, which encloses locking and waiting/notifying mechanisms of Java SE, is proposed for developers to use conveniently.

API	Description
AnycastRequestDispatcher	This is a class that enqueues requests and creates anycast queue threads to respond concurrently. If the current host does not contain the requested data, it is necessary to forward the request to the host's children.
AnycastRequestQueue	This is a thread that possesses a request queue and other remote communication resources. It is the base one to support implementing anycast requests in a concurrent way.
AnycastRequestThreadCreatable	This is an interface to define the method to create a thread for processing anycast requests concurrently.
BoundBroadcastRequestDispatcher	This is a dispatcher to manage broadcast request threads that need to share requests. The threads must be synchronized by a binder.
BoundBroadcastRequestQueue	When processing broadcast requests, no matter whether the current host contains the matched data, it is required to forward the request to its children. That is the difference between the anycast and the broadcast. However, it is suggested that the retrieval and data forwarding can be done concurrently and they do not affect with one another. The thread is designed for the goal since they are synchronized once after both of them finish their critical tasks. Therefore, they do not affect each other.
BoundBroadcastRequestThreadCreatable	The interface defines the method to create the bound broadcast request thread.
BoundNotificationDispatcher	This is a dispatcher to manage threads that need to share notifications.
BoundNotificationQueue	The thread is different from NotificationQueue in the sense that it deals with the case when a notification is shared by multiple threads rather than just one. Therefore, it is necessary to implement a synchronization mechanism among those threads.
BoundNotificationThreadCreatable	The interface defines the method to create the instance of BoundNotificationQueue. It is managed by BoundNotificationDispatcher.
BroadcastRequestDispatcher	This is a class that enqueues requests and creates broadcast queue threads to respond them concurrently. If the current host does not contain the requested data, it is necessary to forward the request to the host's children.
BroadcastRequestQueue	This is a thread that possesses a request queue and other remote communication resources. It is the base one to support implementing broadcast requests in a concurrent way.
BroadcastRequestThreadCreatable	This is an interface to define the method to create a thread for processing broadcast requests concurrently.
CheckIdleable	This is an interface to define the signatures of two methods for a thread's idle checking.
Collaborator	The class encloses locking and notify/wait APIs to help developers control concurrency.
Consumable	The interface defines the method for a consumer in the producer/consumer pattern.
ConsumerThread	This is an implementation of the pattern of producer/consumer.
Dispatchable	It defines some interfaces that are needed in ServerMessageDispatcher.
Interactable	The interface defines a few method signatures for the interaction between a caller and a callee in a concurrent environment. The caller notifies the callee by calling the methods provided by the callee such that the callee can respond to the caller. The caller does that when its running circumstance is changed in a certain situation.
InteractiveDispatcher	This is a task dispatcher that schedules tasks to the special type of threads derived from InteractiveQueue. For the distinct designs, instances of managed

	threads can interact with the dispatcher for high quality management.
InteractiveQueue	This is the base class that must be derived to implement a thread that holds the methods that can be called to notify the interactive dispatcher.
InteractiveThreadCreatable	In general, a pool needs to have the ability to create instances of managed resources. The Creatable interface is responsible for that. The interface defines the method to create instances of InteractiveThread, which is derived from InteractiveQueue.
MapReduce	This is a high concurrency processing class. Numerous threads that take task queues are the input of the class. It is able to execute those threads concurrently and merge the results from them together.
MapReduceQueue	This is a thread to implement the mechanism of map/reduce.
MessageBindable	Some behaviors, such as disposing, on the messages must be synchronized among threads. If no synchronization, it is possible that a message is disposed while it is consumed in another one. The interface defines the relevant method signatures.
MessageProducer	This is a producer/consumer pattern class to input received messages into a concurrency mechanism, the server dispatcher, smoothly.
NotificationDispatcher	This is a class that enqueues notifications and creates threads to process them concurrently. It works in the way like a dispatcher. That is why it is named.
NotificationObjectQueue	This is a fundamental thread that receives and processes notifications as an object concurrently. Notifications are put into a queue and prepare for further processing. It must be derived by sub classes to process specific notifications.
NotificationQueue	This is a fundamental thread that receives and processes notifications in the form of messages concurrently. Notifications are put into a queue and prepare for further processing. It must be derived by sub classes to process specific notifications.
NotificationThreadCreatable	This is an interface defines the method signature to create the instances of NotificationQueue.
RequestDispatcher	This is a class that enqueues requests and creates threads to respond them concurrently. It works in the way like a dispatcher. That is why it is named.
RequestQueue	This is a thread that receives requests from a client, puts those messages into a queue and prepares for further processing. It must be derived by sub classes to provide the real responses for the requests.
RequestThreadCreatable	This is the interface to define a method signature that creates a thread to respond users' requests.
Runner	This is a class to simplify the procedure to invoke a single thread that implements the interface of Runnable of Java SE.
ServerMessageDispatcher	This is the base of a server message dispatcher. All of the messages sent to the server are dispatched by the class concurrently.
Threader	This is a class to simplify the procedure to invoke a single thread that is derived from the class of Thread of Java SE.
ThreadIdleChecker	This is a callback thread that runs periodically to call the idle checking method of the thread being monitored.

Table 1.3 GreatFree Concurrency Implementation and Control

5.4 GreatFree Multicasting

To implement a large-scale distributed system, it is indispensible to employ high efficient multicasting mechanisms. GreatFree provides developers with 23 classes to achieve the goals. Table 4 lists all of the APIs. In the current status, GreatFree supports a tree-based multicasting.

All of the nodes are organized as a tree to transmit data as messages or objects. The multicasting is divided into the notification one and the request/response one. For the later case, it is further categorized into the one of broadcast request/response and the one of anycast request/response.

API	Description
AnycastRequest	The request is a multicast one that is sent to all of the nodes in a cluster. However, once if one node at least responds the request positively, the multicast requesting is terminated. That is the difference from the broadcast requesting.
AnycastResponse	The message is an anycast response to be responded to the initial requester after retrieving the required data.
BroadcastRequest	The message is a broadcast request to be sent through all of the distributed nodes to retrieve required data. For multicasting is required, it extends

	ServerMulticastMessage.
BroadcastResponse	The message is a broadcast response to be responded to the initial requester after retrieving the required data.
ChildMessageCreatable	The interface defines the method that returns the message creator to generate multicast messages to children nodes in the multicasting topology. It is used to define the instance of children multicastor source.
ChildMulticastMessageCreatable	The interface defines the method to create a multicast message on a child node rather than the root one.
ChildMulticastor	This is the multicasting class to run on a child node in the multicasting topology.
ChildMulticastorSource	The class contains all of the initial values to create an instance of ChildMulticastor. That is why it is named ChildMulticastorSource. It is used by the resource pool to manage resources efficiently.
ObjectMulticastCreatable	The interface defines the methods to create multicast messages to be sent.
RootAnycastReaderSource	This class assists the resource pool to create instances of anycast readers. Thus, it contains all of required arguments to do that.
RootAnycastRequestCreatable	The interface defines the methods to create requests in the anycastor.
RootAnyRequestCreatable	The interface returns an instance of the anycast request creator. It is employed by the instance of RootAnycastReaderSource to provide the method for the resource pool to manage anycastors.
RootBroadcastReaderSource	This class provides the resource pool with initial values to create instances of broadcast readers. That is, it is a class that contains all of required arguments to do that.
RootBroadcastRequestCreatable	The interface defines the methods to create requests in the broadcast requestor.
RootBroadRequestCreatable	The interface returns the broadcast request creator. It is employed by the instance of RootBroadcastReaderSource to provide the method for the resource pool to manage broadcastors.
RootMessageCreatable	The interface defines the method that returns the message creator to generate multicast messages. It is used to define the instance of multicastor source.
RootMulticastorSource	The class contains all of the initial values that are required to create an instance of RootObjectMulticastor. The source is needed in the multicastor pool.
RootObjectMulticastor	The code is a core component to achieve the multicasting among a bunch of nodes. The nodes are usually organized into a particular topology to raise the multicast efficiency. In the case, a tree is constructed for the nodes. For different situations, a more appropriate topology can be selected by programmers.
RootRequestAnycastor	This class is the implementation to send an anycast request to all of the nodes in a particular cluster to retrieve data on each of them. It is also required to collect the results and then form a response to return the root. However, only one response is good enough for anycast.
RootRequestBroadcastor	This class is the implementation to send a broadcast request to all of the nodes in a particular cluster to retrieve data on each of them. It is also required to collect the results and then form a response to return the root.
ServerMessage	The class is the base for all messages transmitted between remote clients/servers.
ServerMulticastMessage	This is the base class to implement the message that can be multicast among a bunch of nodes.
Tree	It aims to construct a tree to raise the quality of multicasting. The tree in the case is simple, such as each node having an equal number of children. It is acceptable when all of the nodes have the similar computing capacity and most of them run within a stable computing environment. For a heterogeneous environment, a more complicated tree or other topologies must be applied.

Table 1.4 GreatFree Multicasting

5.4 GreatFree Utilities

GreatFree provides some utilities to assist the programming for sorting, file input/output, timing, XML and so forth. Although those APIs do not contribute to the distributed programming directly, it is useful in their specific cases, as shown in Table 5.

API	Description
CollectionSorter	The class aims to sort a collection, a list or a map, in the ascending or descending manner. It is also able to select the maximum or minimum value from the collection.
FileManager	The class provides some fundamental file operations based on the API from File of JDK.

FreeObject	The class is designed in the system to fit object reusing, caching and so on.
HashFreeObject	This is another general object that defines some fundamental information that is required to manage in a pool. Different from the one, FreeObject, this object is managed by the hash key rather than the object key.
NodeID	This singleton is used to save a node's unique ID only.
NullObject	The class represents nothing. It is used when an object needs to fill the placeholder of generics, but it does not matter what should be put there.
Prompts	It contains prompting messages on screen.
Rand	This code is used to generate different types of random number in integer, float and double by enclosing the one, Random, in JDK.
StringObj	This is an object that can be compared by its key in String.
Symbols	The class defines some frequently-used symbols.
TerminateSignal	The class is a flag that represents whether the node process is set to be terminated or not. For some long running threads, they can check the flag to stop their tasks immediately.
Time	The class consists of some common constants and methods to process timing values.
Tools	The class contains some methods that provide other classes with some generic services.
UtilConfig	The class keeps relevant configurations and constants of the solution.

Table 1.5 GreatFree Utilities

6. The GreatFree Idioms

Another portion of GreatFree is the design patterns as idioms. The design patterns help developers compose those APIs together in a certain structure to handle specific problems. In essence, they are regarded as the sample code for developers to follow. Because of the maturity of idioms, programming distributed systems become a lower cost task than doing that from scratch.

6.1 Terminator

The idiom of Terminator, which is implemented by a singleton, encloses a flag that represents whether an entire process is terminated or not. It is usually embedded into a long running thread, which detects the flag within a loop or periodically. When the flag is set to the state of being terminated, the thread is ended before the next step of the loop.

At the entry of the entire process, a while loop is designed in the way similar to the above manner. The loop does nothing but sleeps for a certain period in each step. The loop exits after the flag is set to the state of being terminated. The primary API the idiom uses is the TerminateSignal in the GreatFree utility.

6.2 Coordinator

For a large-scale distributed system, a centralized Coordinator is required to manage the entire system for the system initialization, the resource registry, the task and memory distribution, the message bus, the resource disposal and so forth. It is also implemented in a singleton. It contains two primary methods, start() and stop(). Thus, the coordinator is the entry and the exit of a distributed node.

6.3 Server Listener

The idiom of Server Listener is a component that plays the role the remote connection listener as the TCP (Transmission Control Protocol) [1] and accomplishes relevant

initialization tasks. For the various goals, it contains the TCP ServerSocket [1], an instance of ThreadPool, ServerIO and its registry, ServerIORegistry. The ServerSocket is used to accept remote TCP connections. When a new connection is constructed and the upper limit is not reached, an instance of ServerIO is created and kept in the registry. Then, the ThreadPool starts a thread to run the newly created ServerIO such that the remote client is served with a concurrency mechanism. In addition, if a peer-to-peer distributed model needs to be implemented, a remote client pool, FreeClientPool, is also taken into account to put in the listener. It needs to initialize an instance of FreeClient for the incoming remote node. For a large-scale distributed system, a peer-to-peer architecture is flexible in terms of implementing highly efficient interactions. Incidentally, each listener maps to one particular port number uniquely. To raise the performance, multiple threads are required to monitor the port. It is initialized and disposed in the Coordinator.

6.4 Remote Eventer

The idiom of Remote Eventer is responsible for sending notifications to a remote node at any moment. As a general eventing mechanism, the notification processes are performed in either a synchronous or an asynchronous manner such that both of the APIs, SyncRemoteEventer and AsyncRemoteEventer, need to be enclosed. In a specific application, all of the eventers are collected in a singleton for convenience. An instance of ThreadPool is also required in the singleton to initialize the asynchronous eventers. It is initialized and disposed in the Coordinator.

6.5 Remote Reader

The idiom of Remote Reader supports another manner to interact with a remote node. It sends a request and waits until a response is received. As a generic module, a singleton API, RemoteReader, is implemented in GreatFree APIs to accomplish the task. For a specific distributed node, it is suggested to enclose all of requests and their calls into a singleton for convenience. It is initialized and disposed in the Coordinator.

6.6 Server IO

Each distributed node needs to derive the API of ServerIO to establish the fundamental input/output streams as a server. A while-loop is needed in the derived idiom and the loop is exited until the remote node closes the connection. In the loop, remote messages are received and then they are assembled with the output stream and the locking for further processing. In the procedure, it employs the API of OutMessageStream.

6.7 Message Producer

The idiom of Message Producer is responsible for delivering messages to the idiom of Server Message Dispatcher for concurrently processing. It is derived from general class, MessageProducer, in GreatFree APIs. Additionally, it needs to be implemented with an instance of ServerMessageDispatcher. To keep concurrent, an instance of

Threader is also needed to start the idiom. As required by the Threader, a disposer is defined for the instance of ServerMessageDispatcher. For any distributed applications, only one message producer is needed to dispatch received message for further processing. Thus, a singleton is required to wrap the instance of MessageProducer. When a message is received by the Server IO, it is forwarded to the Message Producer after being assembled with the output stream and the locking using OutMessageStream. The idiom is initialized and disposed in the Coordinator.

6.8 Server Message Dispatcher

Derived from the API, ServerMessageDispatcher, the idiom of Server Message Dispatcher consists of all of the thread management pools to deal with incoming messages concurrently. According to the types of those messages, the thread management pools are divided into different groups for notifications, requests, multicast notifications, anycast requests, broadcast requests and so forth. For each type of messages, it is possible to define some exactly specific sub types of messages, such as the request for sign-in or the notification for online. If the sub types of incoming messages are rich, it is allowed to define multiple Server Message Dispatchers and initialized in the Message Producer. Furthermore, it needs to create a specific thread management idiom for each sub type message. All of them are enclosed, initialized and shutdown in one particular Server Message Dispatcher. The typical structure for the idiom is a switch statement that dispatches incoming messages to each of the thread management idiom according to the message identifications.

6.9 Notification Queue

The idiom of Notification Queue is derived from the API, NotificationQueue, with a specific notification message. In the idiom, one two-level nested while-loop is needed to enclose the operations, which are executed immediately after receiving one notification. The outer while-loop detects whether the thread is shutdown. The inner while-loop detects whether the queue to keep notifications in the idiom is empty or not. If the queue is empty, the thread does not terminate immediately. It needs to be waiting for a period such that it avoids high CPU usages for the long-running loop and it also reduces the cost to create a thread when new notifications are received. Usually, it is a high cost solution to collect the thread immediately when the queue is empty. To save memory, it is also suggested to dispose notifications after it is processed. For that, developers are required to invoke the disposing method.

6.10 Notification Object Queue

The idiom of Notification Object Queue is derived from the API, NotificationObjectQueue. Its structure is identical to that of the Notification Queue. The only difference is that data in the queue is derived from the Java base class, object, whereas the queue in the Notification Queue keeps data derived from ServerMessage of GreatFree. For that, the Notification Object Queue is usually not used as remote concurrent processing unless the object implements the interface of serializable of Java.

6.11 Bound Notification Queue

When a notification needs to be handled by multiple threads concurrently after it is received, it needs to use the idiom of Bound Notification Queue to define them if the notification is probably updated in any one of them. The idiom is derived from the API, BoundNotificationQueue, in which an interface, MessageBindable, should be implemented to enclose the operations to be synchronized.

6.12 Request Queue

As a child of the API, RequestQueue, the idiom of Request Queue has the similar structure to that of Notification Queue, such as the two-level nested while-loop, the message queue, the waiting control to save resources and the message disposing methods. Besides those, a responding method needs to be invoked in the inner loop immediately after the response is created.

6.13 Bound Broadcast Request Queue

The idiom of Bound Broadcast Request Queue is another case to send a request and receive a response from a remote system that is made up with a numerous distributed nodes. It is derived from the API, BoundBroadcastRequestQueue. Different from that of the Request Queue, the request is transmitted to each node in the system rather than to a single one. Additionally, since the request is handled by two concurrent threads, i.e., one for generating the response and another for forwarding it continually, and both of them might change it in some cases if applicable, such as disposing the request, it is necessary to synchronize the relevant operations. That is why the idiom is named Bound. The synchronized operations are enclosed in the unique instance of a class implementing the interface of MessageBindable.

6.14 Anycast Request Queue

Different from that of Bound Broadcast Request Queue, it is not required to get responses from each of them when a request is sent to a large number of potential distributed nodes. If anyone of them responds, the entire procedure is terminated. Derived from the API, AnycastRequestQueue, the idiom of Anycast Request Queue takes the task. It notes that it is impossible that the request is handled by the two operations concurrently, i.e., the one forwarding it and the one responding it. Because the forwarding is probably terminated in anycast, it is not necessary to synchronize the above two threads since the two operations are performed in a sequential order in the idiom. Thus, the idiom is not named as Bound as what it does on that of Bound Broadcast Request Queue.

6.15 Root Multicastor

Being situated at the coordinator node of a distributed system, the Root Multicastor is an idiom for multicasting in the system that consists of numerous distributed nodes. By default, the multicasting is performed within a tree topology and the coordinator plays the role of the root. It is a singleton that includes a bunch of multicastor resource pools, which are implemented by the API of ResourcePool. Each of the pools is able to create instances of specific multicastors derived from RootObjectMulticastor. They send data to all of the nodes in the tree. Each of the methods of the Root Multicastor contains four components, i.e., data to be transmitted, the initialization of a multicastor, the transmission invocation and the collection of the multicastor for reuse. The idiom is responsible for sending notifications only.

6.16 Child Multicastor

The partner of the Root Multicastor is the Child Multicastor, which is located at the children nodes of the multicasting tree. Each of the child multicastors is derived from ChildMulticastor to send data its immediate children to keep the multicasting go ahead. Except that, the structure of the Child Multicastor is identical to the root one. The multicastors are also managed by the instances of ResourcePool.

6.17 Root Anycast Eventer

Sometimes it is necessary to send notifications to some of the nodes in a system instead of all of them. In this case, the Root Anycast Eventer takes the responsibility. It is also situated at the root of the multicasting tree. There are no any differences from the structure of the Root Multicastor. The RootObjectMulticastor is the parent API of each multicastor. It only needs to send notifications to its immediate children.

6.18 Child Anycast Eventer

As the partner of the Root Anycast Eventer, the Child Anycast Eventer keeps sending notifications to its immediate children. But the operation is terminated if the notification is needed by the local node. The judgment is carried out by the idiom of Notification Queue, which receives the notification. The structure of the idiom is identical to that of the Child Multicastor. Each multicastor is also derived from ChildMulticastor and managed by the instances of ResourcePool.

6.19 Root Broadcast Reader

Sometimes, it is required to query all of the distributed nodes in a system. In the case, the idiom of Root Broadcast Reader is competent with the task. Different from those in the Root Multicastor, the multicastors are derived from RootRequestBroadcastor. Besides the multicastor pools implemented by ResourcePool, the idiom needs to implement a mechanism to collect the responses. That is, until the number of

responses reaches to that of the total of the distributed nodes, the method of the broadcast requesting is blocked.

6.20 Child Broadcast Reader

As the partner of the Root Broadcast Reader, the Child Broadcast Reader is easier to be implemented since it is unnecessary to take into account any additional issues. Its primary task is the same as the Child Multicastor. The change happens in the relevant threads, i.e., the Bound Notification Queue and the Bound Broadcast Request Queue, to process the broadcast requests. The first queue forwards the requests to its immediate children and the second one creates the relevant response to send it back to the root.

6.21 Root Anycast Reader

Different from the Root Broadcast Reader, another multicast requesting is accomplished by the idiom of Root Anycast Reader if one response at least is received by the root. It is unnecessary to send the request to all of the nodes in the distributed environment if one response at least is received. Similar to the Root Broadcast Reader, the idiom needs to implement a response collecting mechanism besides the multicastor pools. But it only needs to obtain one response and then returns it to the requestor instead of waiting for all of the requests from each node in the system.

6.22 Child Anycast Reader

The partner of the Root Anycast Reader is the Child Anycast Reader. The request from the root needs to be forwarded to the children of the current local node if the local node cannot generate the required response. The implementation of the idiom is identical to the Child Multicastor. The difference happens in the threads that invoke the methods of the idiom. In the case, the corresponding thread idiom is the Anycast Request Queue.

6.23 Interactive Queue

Different from traditional threads, the idiom of Interactive Queue contains a number of callback methods such that the running threads are able to interact with the pool that manages them. The interactions are realized through invoking the callbacks. According to the interactions, it is possible for the pool to notice the status of those threads such that it can make a proper decision to manage those threads effectively and efficiently.

6.24 Map Reducer

The idiom of Map Reducer is an important idiom to implement the high concurrency mechanism. When a large number of parallel tasks are available to be processed and

their results need to be merged, it is highly efficient to put them into the Map Reducer if they do not need to synchronize during the procedure to accomplish each of them. The Map Reducer is made up with the pools to manage the resources of MapReduce. Each of its public methods takes numerous tasks as input and the merged results as return values. Inside the methods, it needs to specify the reducer, initialize the algorithm object and invoke the concurrent mapping and reducing.

7. The Experimental Environment

GreatFree APIs and idioms were proposed during the procedure to implement a new infrastructure of World Wide Web [11][2]. It believes the social capital [36] is the driver to associate the human capital [37] over the Web. For the understanding, it needs to take a heavy load to program from scratch, including the issues of routing, multicasting, persisting, presenting and so forth. It attempts to build a brand new large-scale heterogeneous information system such that users are able to perform various information accessing behaviors, such as publishing, forwarding, commenting, browsing, navigating, searching and following.

The current version is implemented with Java SE 7. It reaches the total lines of code, 466,037, at the server side. In addition, its client side has 11,662 lines of code, which are implemented on Android 5.0. All of APIs and idioms were proposed during the procedure to implement the server side.

The server side is made up with the coordinator, the crawlers, the data rankers, the publisher rankers, the memory nodes and the interactive nodes. In the current version, except that the coordinator is implemented with a single computer, other types nodes are comprised of multiple computers without the upper limit. As the center of the system, the coordinator is responsible for integrating all of the nodes together, such as distributing tasks, disseminating data, forwarding queries and so forth. Each of the crawlers receives tasks from the coordinator, collects Web pages from the Web and then injects them into the system for sharing in a concurrent manner. The data rankers evaluate and sort data in a certain order such that it can be presented to users in a high quality form. The node rankers are responsible for sorting publishers of data and the consequences are useful to construct the graphs between publishing organizations and individuals. The memory nodes achieve the goal to keep important data in memory as cache and the rest data on disk for persistence with a distributed cluster. The interactive nodes connect with users' devices directly and respond to users' requests. Each of the above nodes is implemented from scratch with Java SE initially for the proposals of new distributed models and data transmission protocols. For that, the relevant APIs and idioms are proposed after a long-term accumulation. Nowadays the backend of the system is accomplished, but its clients are still under construction. It will be launched as a commercial system in one or two years.

8. Future Work

As a comprehensive solution to the large-scale distributed system, it can hardly claim the APIs and idioms discussed in the paper cover all of the issues. With the progress of the development of the system, more APIs and idioms must be put forward. In accordance with the domain of the social computing, it is expected to invent some

patterns that fit in the unstable environment. In addition, it needs to improve the thread management approaches in the current version since it notes that the number of raised threads is high. One reason for that is due to that each dispatcher has a parent thread that is always alive. For such dispatchers are numerously utilized, it leads to the growth of the number of threads. It can be solved through the lazy initialization and the parent thread is killed after a certain period. Finally, all of the algorithms in each APIs and idioms need to be improved. For the limited time, the algorithms are still rough. For example, the caching and pooling algorithms are to be optimized. Potential developers are able to strengthen those algorithms with the open source.

References

[1] Elliotte Rusty Harold. 2014. Java Network Programming. O'Reilly Media, ISBN: 978-1-449-35767-2.

[2] Doug Lea. 1999. Concurrent Programming in Java: Design Principles and Patterns, Second Edition. Addison Wesley, ISBN: 0-201-31009-0.

[3] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes and Doug Lea. 2006. Java Concurrency in Practice. Addison-Wesley, ISBN: 978-0-321-34960-6.

[4] Joshua Bloch. 2008. Creating and Destroying Objects, Chapter 2, Effective Java. Addison-Wesley, ISBN: 978-0-321-35668-0.

[5] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. 2011. Architectures, Chapter 2, Distributed Systems: Concepts and Design, the Fifth Edition. Addison-Wesley, ISBN: 0-13-239227-5.

[6] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. 2011. Consistency and Replication, Chapter 7, Distributed Systems: Concepts and Design, the Fifth Edition. Addison-Wesley, ISBN: 0-13-239227-5.

[7] Jim Farley. 2001. Message-Passing Systems, Chapter 6, Java Distributed Computing. O'Reilly Media, ISBN: 1-56592-206-9E.

[8] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. 2011. Communication, Chapter 4, Distributed Systems: Concepts and Design, the Fifth Edition. Addison-Wesley, ISBN: 0-13-239227-5.

[9] Ken Arnold, James Gosling and David Holmes. 2005. The Java Programming Language. Addison-Wesley, ISBN: 0-321-34980-6.

[10] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. 2011. Processes, Chapter 3, Distributed Systems: Concepts and Design, the Fifth Edition. Addison-Wesley, ISBN: 0-13-239227-5.

[11] Bing Li. 2015. DOI: https://github.com/greatfree/Programming-Clouds.

[12] Tomcat. 2015. Apache Tomcat. DOI: http://tomcat.apache.org/.

[13] Danny Coward. 2015. Java EE 7: The Big Picture. McGraw-Hill Education, ISBN: 978-0-07-183734-7.

[14] Chuck Lam. 2011. Hadoop In Action. Manning Publications, ISBN: 978-1-93518-219-1.

[15] Dan Radez. 2015. OpenStack Essentials. Packt Publishing, ISBN: 978-1-78398-708-5.

[16] Juval Lowy and Michael Montgomery. 2015. Programming WCF Services: Design and Build Maintainable Service-Oriented Systems. O'Reilly Media, ISBN: 978-1-491-94483-7.

[17] Steven John Metsker. 2002. Design Patterns Java Workbook. Addison Wesley, ISBN: 0-201-74397-3.

[18] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. 1996. Pattern-Oriented Software Architecture, A System of Patterns, Volume 1. John Wiley & Sons Ltd., ISBN: 0-471-95869-7.

[19] Douglas Schmidt, Michael Stal, Hand Rohnert and Frank Buschmann. 2000. Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2. John Wiley & Sons Ltd., ISBN: 0-471-60695-2.

[20] Michael Kircher and Prashant Jain. 2004. Pattern-Oriented Software Architecture, Patterns for Resource Management, Volume 3. John Wiley & Sons Ltd., ISBN: 0-470-84525-2.

[21] Frank Buschmann, Kevlin Henney and Douglas Schmidt. 2007. Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing, Volume 4. John Wiley & Sons Ltd., ISBN: 978-0-470-05902-9.

[22] Frank Buschmann, Kevlin Henney and Douglas Schmidt. 2007. Pattern-Oriented Software Architecture, On Patterns and Pattern Languages, Volume 5. John Wiley & Sons Ltd., ISBN: 978-0-471-05902-9.

[23] Paul Whitehead, Ernest Friedman-Hill and Emily Vander Veer. 2002. Java and XML. Wiley Publishing, ISBN: 0-7645-3683-4.

[24] Solr. 2015. DOI: http://lucene.apache.org/solr/.

[25] JBoss. 2015. DOI: http://www.jboss.org/.

[26] WebSphere. 2015. DOI: http://www.ibm.com/software/websphere.

[27] FlashGet. 2015. DOI: http://www.flashget.com/index_en.html.

[28] Skype. 2015. DOI: http://www.skype.com.

[29] Twitter. 2015. DOI: http://www.twitter.com.

[30] HTTP. 1996. HTTP – Hypertext Transfer Protocol Overview. DOI: .http://www.w3.org/Protocols/.

[31] JavaServer Pages. 2015. DOI: http://www.oracle.com/technetwork/java/javaee/jsp/index.html.

[32] Ian Clarke, Oskar Sandberg, Brandon Wiley and Theodore W. Hong. 2001. Freenet: A Distributed Anonymous Information Storage and Retrieval System. International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability, 2001.

[33] Matei Ripeanu. 2001. Peer-to-Peer Architecture Case Study: Gnutella Network. Proceedings of 1st International Conference on Peer-to-Peer Computing, 2001, pp. 99-100.

[34] Time-Berners Lee. 1989. The Original Proposal of WWW and HTML. DOI: http://www.w3.org/History/1989/proposal.html.

[35] Bing Li. 2015. The New Infrastructure of WWW. DOI: http://greatfree.blog.163.com/.

[36] Nan Lin. 2001. Social Capital: Capital Captured Through Social Relations, Chapter 2. Social Capital – A Theory of Social Structure and Action, Cambridge University Press, ISBN: 0-521-47431-0.

[37] Nan Lin. 2001. Theories of Capital: The Historical Foundation, Chapter 1. Social Capital – A Theory of Social Structure and Action, Cambridge University Press 2001, ISBN: 0-521-47431-0.

[39] Martin Odersky, Lex Spoon, Bill Venners. Programming in Scala, Second Edition. Artima Press 2010.

[40] Munish K. Gupta. Akka Essentials. Pakct Publishing, ISBN: 978-1-84951-828-4

[41] Alan Donovan, Brian Kernighan. The Go Programming Language. Addison-Wesley, ISBN-13: 978-0-13-419044-0; ISBN-10: 0-13-419044-0.

Chapter 2 Setting Up the Environment

Abstract

This chapter aims to explain in detail about the programming environment to be set up before a programmer starts to learn programming with GreatFree. I need to emphasize that as a professional software engineer, the Unix-like environment [1][2][3] is the most vital one for programming rather than any others, such as Windows [4]. It is just a common sense, which is not necessary to be explained further. Unfortunately, I noticed that many students or engineers would like to program or only know how to program on Windows. After five decades, Unix-like systems, such as Solaris [5], Mac OS X [6], Linux [7][8], iOS [9] and Android [10], almost occupy most types of computing devices, like backend servers, main frames, distributed clusters and even smart phones, except the PC client side. Without knowledge about Unix programming [1][2][3], a programmer must lose the future.

Another point I have to indicate is that students should not rely on the Integrated Development Environment (IDE) [11] heavily, especially during the phase of testing. The command-line environment is always the first priority choice for a professional programmer. The best benefit we can obtain from an IDE is that it can help us type, parse or compile code. However, when debugging is started, especially when it is performed in a distributed environment with many concurrent threads, I highly suggest you to abandon any IDE and accomplish the task with a Unix terminal [1]. Each terminal holds one process which contains multiple threads. The advantage to do that is that you can save resources when debugging. Otherwise, you have to start one instance of IDE for each process. Usually, it is a way to waste computing resources because IDE itself needs to consume many resources. Compared with IDE, Unix terminals are lightweight in terms of resource consumption. Moreover, a distributed system is always a concurrent one that contains many threads running asynchronously. Even though you choose IDE to debug them, the debugging functions supported by IDE cannot work properly since it must affect the asynchronous and synchronous characters of a program. For example, if is infeasible to debug a distributed system by inserting break points and running code line by line by pressing a function key. For that, a Unix terminal is always the best friend of a programmer to debug a distributed system.

Finally, I should suggest that multiple computers are highly suggested when you practice distributed programming. Although you can start multiple terminals to simulate a distributed environment when running your code, maybe it is fine when the system is small in terms of the amount of computing resource consumption. However, when you need to implement a practical system, you must notice that it is impossible to do that immediately on a single computer. The most important motivation to design distributed systems is that a single computer cannot afford the cost of computer resources. So why should we do that with a single one? Although many sample systems in the book can run over a single computer, there are some heavy burden ones that consume a lot of resources, such as the sample for a search engine [12]. On the other hand, I think programming with multiple computers together is much

funnier than a single one. If you have some colleagues, you can implement a system with GreatFree with your own computers and then connect them together for interactions. It is really like a game, right? Fortunately, computers become much cheaper at this time. You can order a highly-configured new one and get two or three old ones. That is what I did in my laboratory.

OK, you must be ready for setting up the environment. Let's go!

1. The System Environment

The section gives you a detailed introduction to set up the programming environment. I assume that you have multiple desktops or laptops with installed Unix/Linux. It does not mean that you cannot keep practicing GreatFree with a single PC with Windows. Yes, you can. For example, you can install a Java IDE to program with GreatFree on Windows. But it is really not so professional. For a practice, it is still fine. But when implementing a practical system, it becomes impossible.

On the other hand, for senior programmers, the below descriptions might not be the one choice since you must have your own favorite tools or options. You can decide by yourself for the environment.

1.1 Operating Systems

Since I highly suggest you to program over Unix-like operating systems, you must have desktops and laptops on which those systems are installed. If you have Apple [13] computers, such iMac [14] or Mac Book Pro [15], I have to say congratulations to you since you do not need to spend effort on operating systems. Mac OS X [6] for those computers are Unix inherently indeed.

Unfortunately, Apple stuffs are always expensive. That is what I hate as well. So, if you do not have Apple ones, it does not matter. You can still solve the problems smoothly in the current computer world. You must know about open source operating systems, Linux [7], which has multiple children, such as Ubuntu [8], RedHat [16], Debian [17] and CentOS [18]. Anyone of them is fine for you to learn Unix and practice distributed system programming. In my laboratory, I choose Ubuntu. I have to say that I am not an operating system (OS) geek. Although I spend a large amount of time on Linux, I never strive to investigate any piece of code of Linux. The operating system is a complete black box although I know it is must more robust and popular than Windows. Luckily, it does not affect me to program and implement my systems if you know the frequently-used Linux commands [19]. In the later sections, you must see such commands. You do not need to worry about since they are easy to learn if you have some experiences even on any other OS, such as Windows. Certainly, if you are an experienced guy in Linux, you must have much deeper understandings about the systems we will program on it than me. I am also trying to improve on that.

In my case, I have Apple machines and Linux ones. When I program, both Mac OS X and Linux are used. The Linux I select is Ubuntu 16.04.1 [8], which can be downloaded for free. You need to follow the instructions online to install it on your

desktop or laptop. I do not explain that in the book. I just want to tell you one thing. If your machine has Windows installed and you prefer having the operating systems on your computer, the installation software of Ubuntu has such an option. You just need to select before the installation is started. I believe most students in school would like to keep Windows since they need to write documents or play games on it. Figure 2.1 illustrates how to keep Windows and Ubuntu on your PC when starting to install Ubuntu. Make sure to select the option of "Install Ubuntu alongside Windows 7". After the installation, when you start your PC, you must see a new prompt interface shown in the Figure 2.2, in which you can make a selection to decide which operating system you would like to enter.

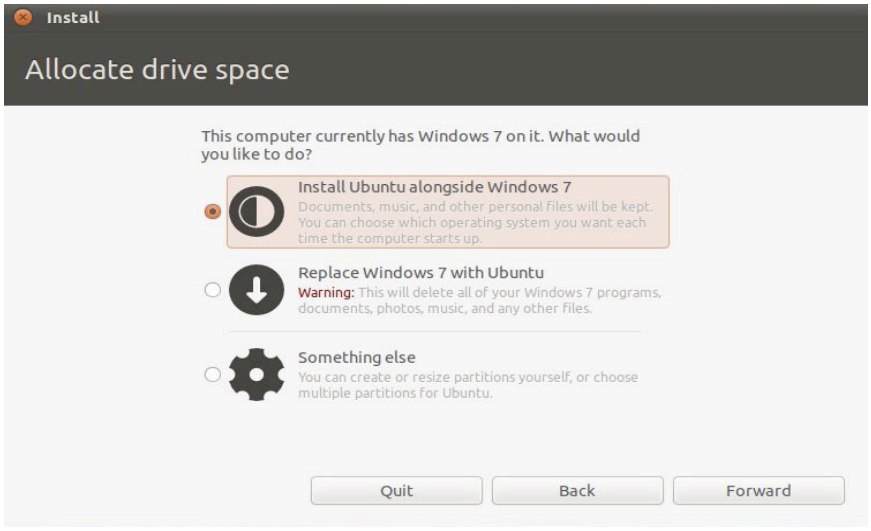


Figure 2.1 If Windows is located on your PC when installing Ubuntu, ensure to select the correct option if you want to keep both of them or not

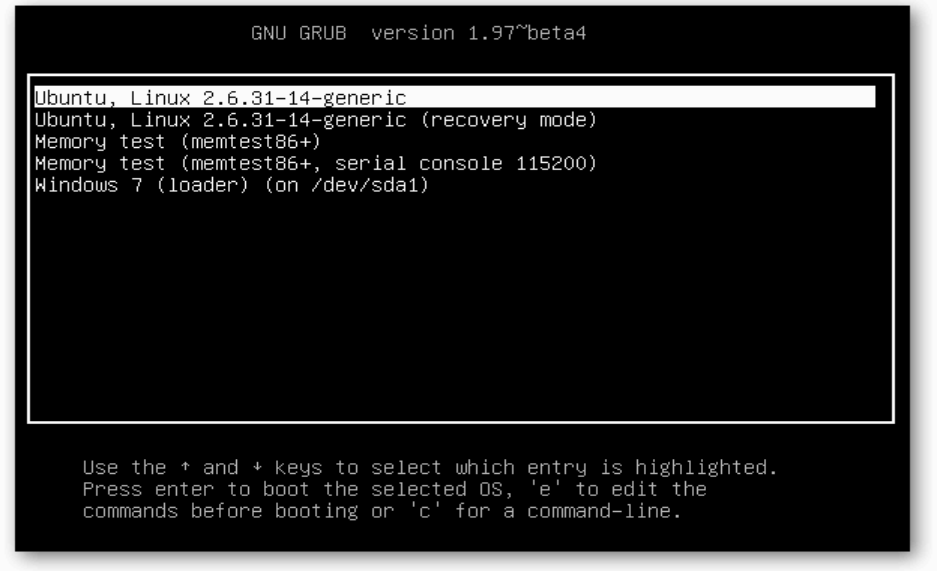


Figure 2.2 After the installation of Ubuntu, a menu is displayed if you decide to keep both Windows and Ubuntu together on your PC. You can make a selection to enter the operating system you like each time you start your PC

1.2 SSH Server/Client

Once if a Linux system, such as Ubuntu, is installed, you need to get ready to install SSH Server/Client [20], which is useful to transmit source code between your multiple computers for debug and control all of the computers remotely. You might not understand why we should transmit source code. I will describe and show how to do that later in detail. Now I just explain simply that you have to deploy your programmed code to other machines through SSH because we are programming distributed systems.

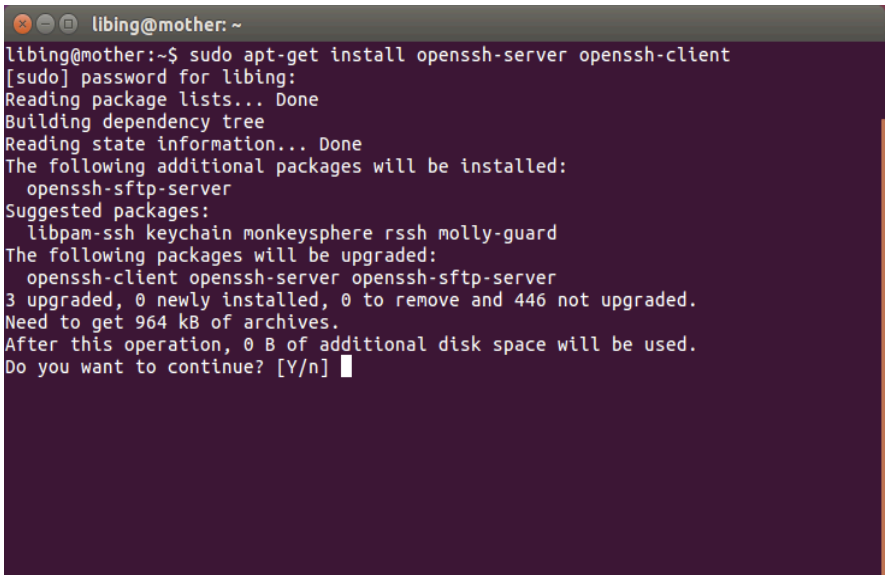


Figure 2.3 Install SSH Server/Client in Ubuntu

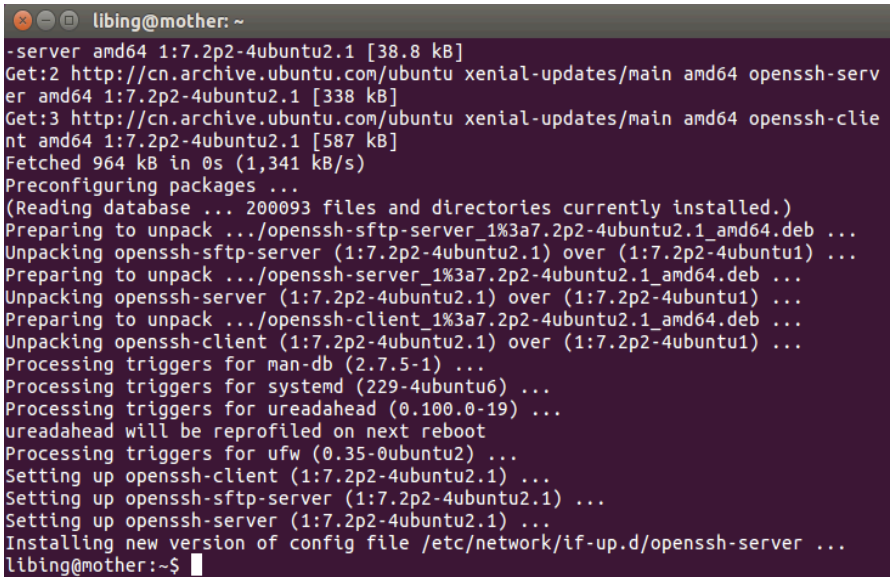


Figure 2.4 The SSH Server/Client is Installed Successfully

SSH Server/Client is a popular FTP-like [21] secure software. It is convenient to transmit files between Unix/Linux machines and manipulate remote computers in terminal. Since Ubuntu is the environment on which I program, I just show how to

install SSH Server/Client on such an OS. If you want to do that on others, you can get relevant information easily online.

To install SSH Server/Client on Ubuntu, you need to type the command as follows.

$ sudo apt-get install openssh-server openssh-client

After that, you must get prompted and just follow them. The entire procedure is pretty straightforward and it should be easy to complete. Figure 2.3 shows the terminal after typing the command to install the SSH server and client. And Figure 2.4 displays what the terminal looks like after the installation. I need to indicate that the SSH is already installed on my PC when I take the screenshots. So it prompts that the package will be upgraded. If you install SSH on a fresh PC, the message must be different. Anyway, the procedure is straightforward. So you just follow the prompts without worrying about the difference.

If you use Mac OS X, you do not need to install SSH since it is already built into the system. However, you have to enable it through the dialog of Sharing before it can work for you, as shown in Figure 2.5.

Figure 2.5 Enable SSH on Mac OS X

1.3 Java Development Toolkit (JDK)

Now you need to install the core stuff of the development environment. Since we program with Java, you are required to install Java Development Toolkit (JDK) [22][23]. It can also be downloaded for free from the Oracle Web site [23]. At present, you can download JDK 8 although JDK 7 is also accepted. JDK can be installed on most popular operating systems like Windows, Unix, Solaris, Mac OS X and Linux. You need to pay much attention to which operating system you work on before starting the download. Moreover, you also need to be aware of whether your

OS is 64 bit or 32 bit before downloading. All of the options are listed clearly on the Oracle site as shown in Figure 2.6.

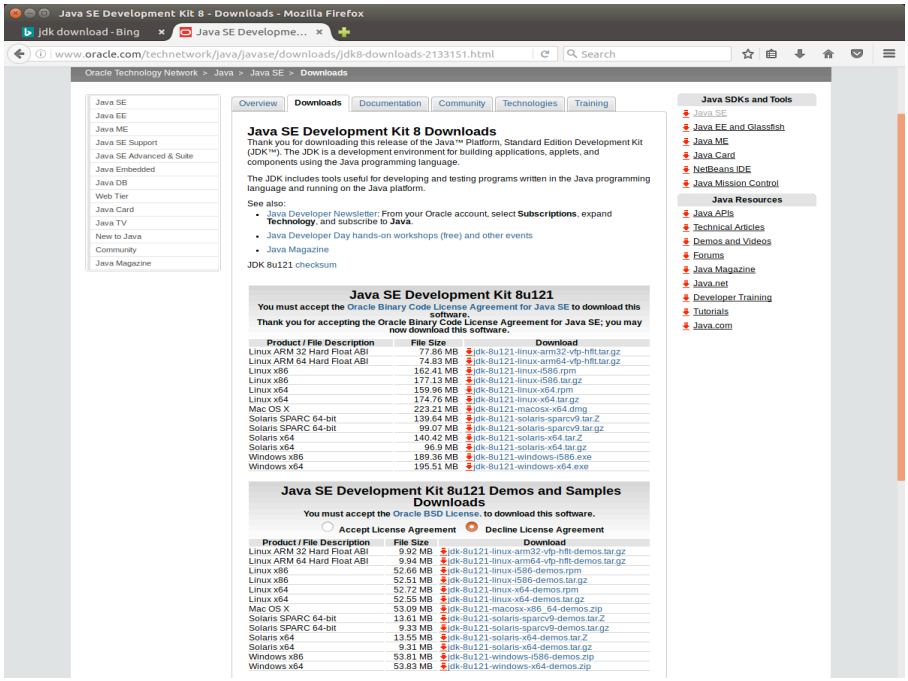


Figure 2.6 The Web site from Oracle.com for JDK downloading

After JDK is downloaded, it can be installed conveniently by extracting the downloaded package. If it is a compressed one, you are required to decompress it before running it. The entire procedure is also smooth and you are not required to intervene.

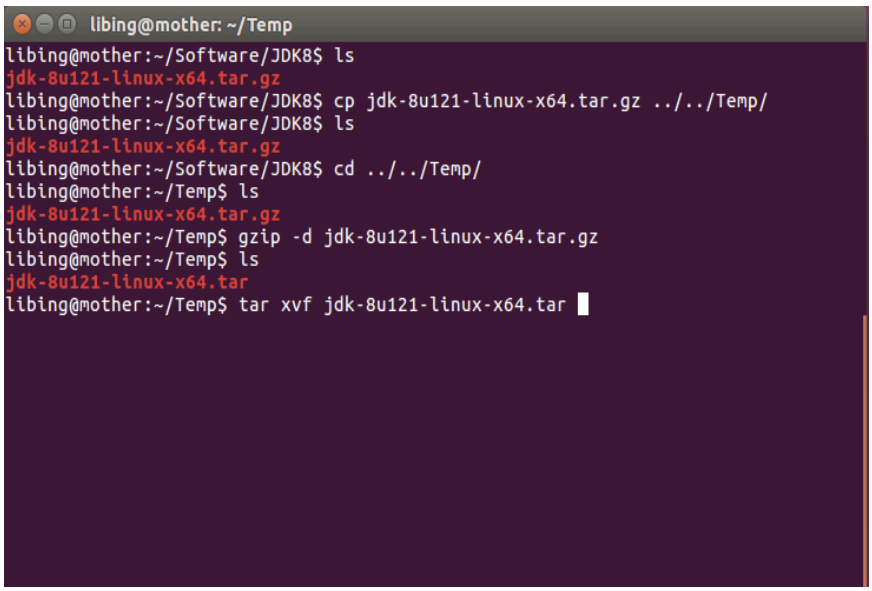


Figure 2.7 The manipulations in the terminal to decompress the download JDK package

For example, if you download the package, jdk-8u121-linux-x64.tar.gz, as shown in Figure 2.7, you are recommended to make a copy for it and move it to a clean

directory, e.g., ./Temp. In the directory, you need to decompress it first. To do that, just type the below command.

$ gzip –d jdk-8u121-linux-x64.tar.gz

After that, you get a tar file named, jdk-8u121-linux-x64.tar, which must be extracted by executing the below command.

$ tar xvf jdk-8u121-linux-x64.tar

Once if it is done, a directory named jdk1.8.0_121 is available under the current folder, which contains all of JDK files, as shown in Figure 2.8.

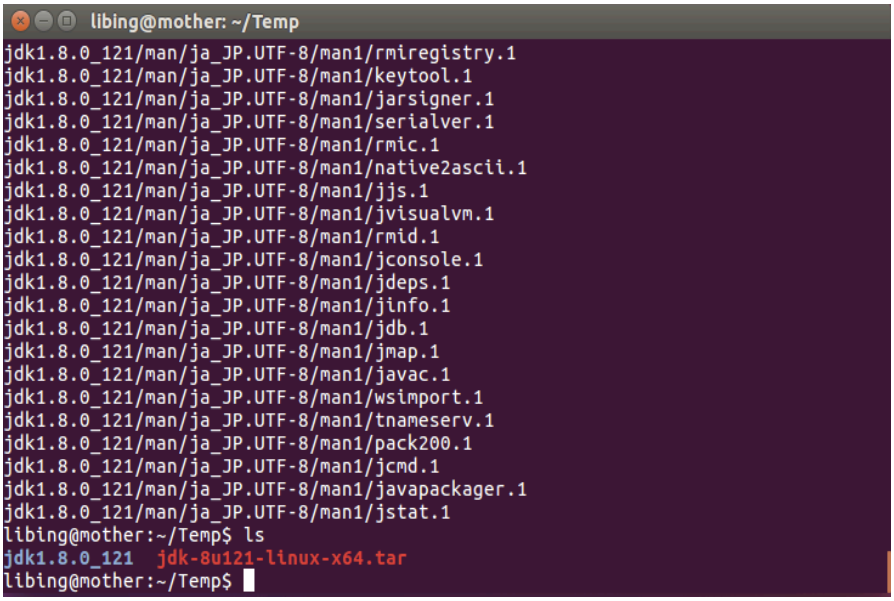


Figure 2.8 The directory extracted from the JDK tar file

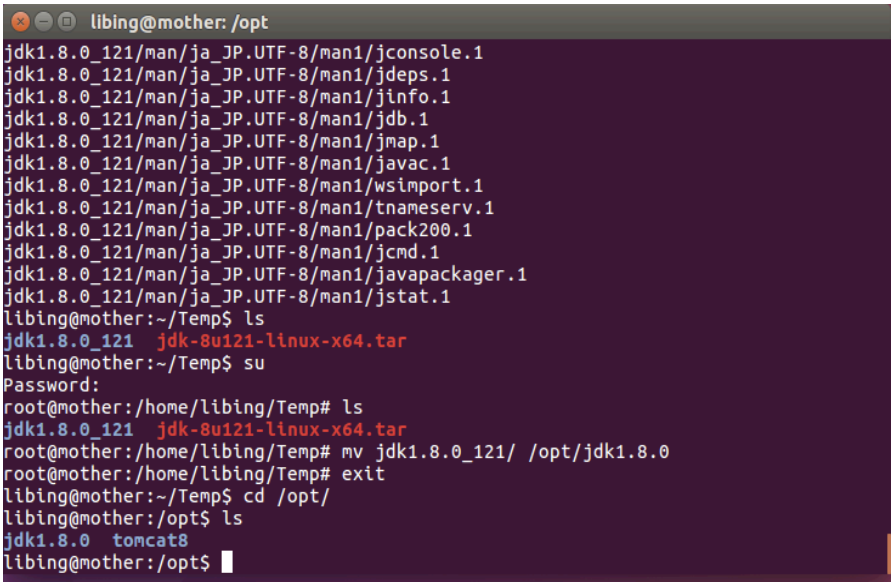


Figure 2.9 The JDK is moved under the directory, /opt, conventionally

Conventionally, it is required to put the installed software under the directory, /opt. That is also what I do for the JDK, as shown in Figure 2.9.

Since /opt is a directory belonged to root, you are required to do that with the administrator, root. Once if you switch to root, you can move the JDK directory from the current one to /opt. You should note that the new directory is renamed as jdk1.8.0 rather than the original one, jdk1.8.0_121 for the sake of cleanness.

```
# mv jdk1.8.0_121/ /opt/jdk1.8.0
```

After the moving, you need to switch back to your regular account.

After JDK is installed, you are required to revise your profile to specify the environment variables for your development as follows.

JAVA_HOME

What you need to do in the profile is to add a new variable, JAVA_HOME, which tells the system where the JDK is located.

PATH

Then, you are also required to add the path, $JAVA_HOME/bin, which contains JDK commands into the variable, PATH, such that you can type JDK commands under any directories.

CLASSPATH

In addition, another new variable is CLASSPATH, which is critical since it helps the development environment to find JDK libraries.

JAVA_OPTS

JAVA_OPTS is a variable to specify the size of memory to be used by JDK. When your Java is executed, the memory it consumes should not exceed the size. In the later sections, another preferred approach is to specify the option in the build.xml consumed by Ant [24].

The last task is to export the new environment variables in your profile. The revised profile should be like the one shown in Figure 2.10.

After the profile is revised, you can validate whether JDK is installed correctly or not though executing some commands of JDK, such as java and javac. The command of java is the one to run a Java application and the one of javac is responsible for compiling a Java program.

Before executing the JDK commands, you should ensure your profile takes effect in the environment of your current account. You just need to type the below command to update your computing environment.

$ . .profile

Finally, installing JDK over Mac OS X is a little bit different. According to Figure 2.6, the JDK package for Mac OS X is jdk-8u121-macosx-x64.dmg. After downloading it, it is installed automatically such that you cannot intervene it during the procedure. In my Mac OS X, JDK is put under the below directory. You have to specify them carefully in the profile, .bash_profile, for your account on Mac OS X.

JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home

To take effect for all of the setting up, you also need to execute the profile as follows. However, the command is .bash_profile, which is different from .profile on Ubuntu.

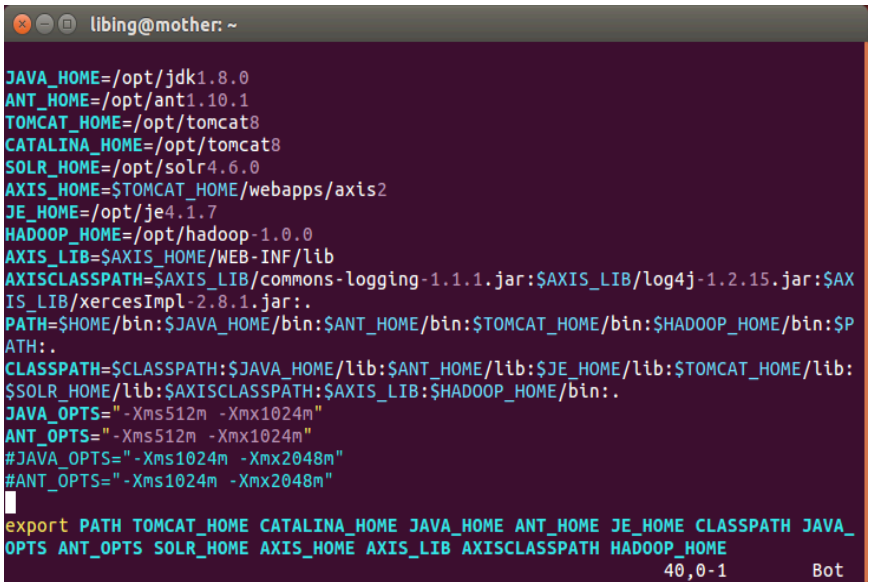


Figure 2.10 The revised profile for the installed JDK

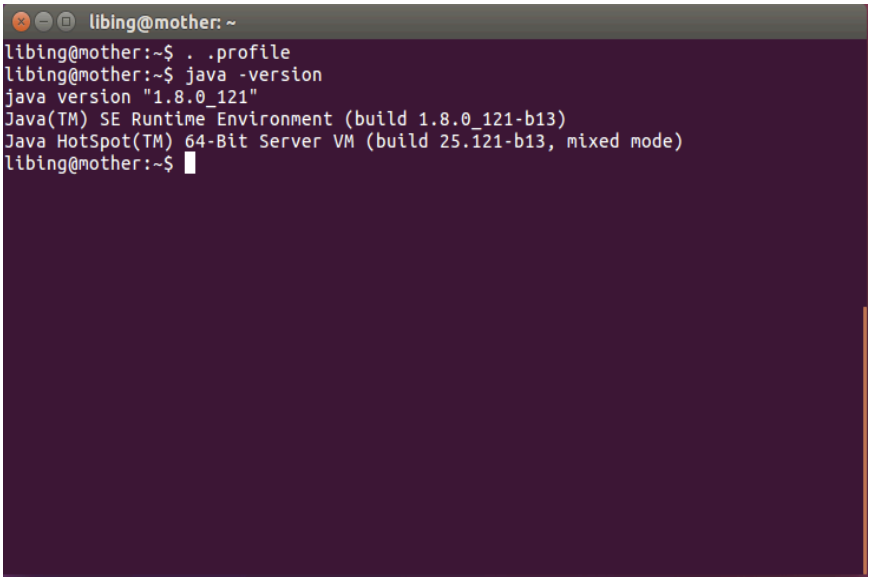


Figure 2.11 The version of JDK is displayed after executing the command of java without any parameters

$ . .bash_profile

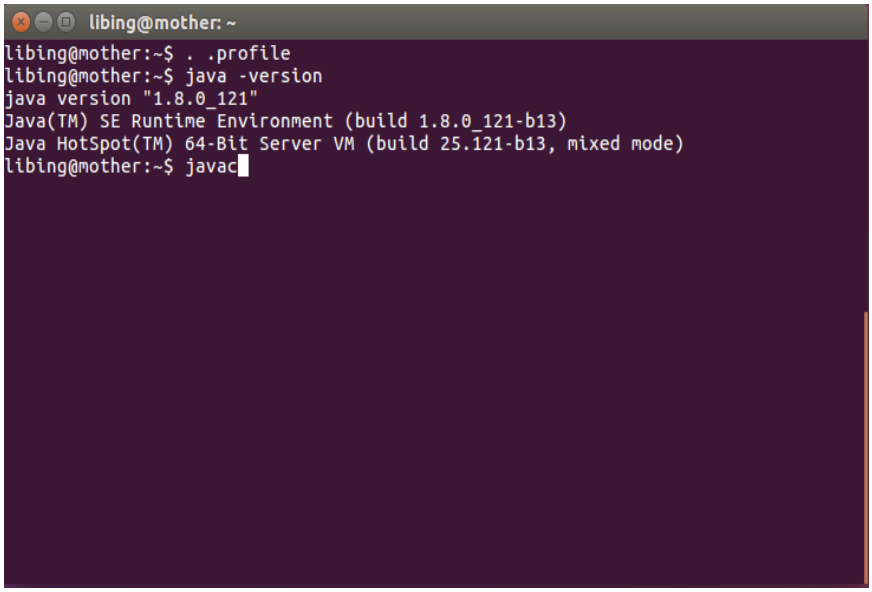


Figure 2.12 It is time to run javac to validate whether JDK is installed correctly

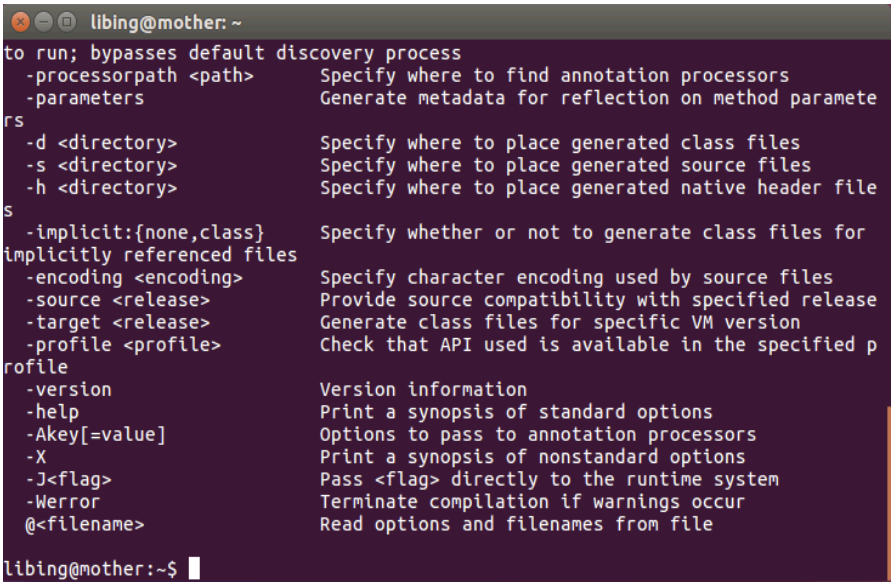


Figure 2.13 The help information is displayed after javac is executed without any parameters

After that, all of those environments, such as JAVA_HOME, CLASSPATH, PATH and so forth, become effective.

Then, you can type the command of java to validate the installation of JDK.

$ java -version

If everything is fine, it prompts to you the information about the version information about the JDK you just installed. In this case, the version is 1.8.0_121. And, it also shows some other information related to your JDK, such as JRE and HotSpot.

If you run the command of javac without any parameters as follows, it displays the help information about it. At this time, you do not need to care about the information. What we are doing right now is to ensure JDK is installed properly.

$ javac

Figure 2.11 illustrates how to show the version of JDK by running java. Figure 2.12 and Figure 2.13 show the help information about javac if you type it in a terminal without any parameters.

1.4 Eclipse

An integrated development environment (IDE) is needed for sure when programming applications with Java. For Java programming, there are a couple of IDE candidates. In them, Eclipse [11][25] is the one I highly recommend. This is the most famous and popular one for Java programming.

I ever tasted others like NetBeans [26] and IntelliJ IDEA [27]. Although I cannot say the others are worse than Eclipse, I would like to suggest you to use Eclipse. The differences between them are tiny and I am not interested in spending much time clarifying that. At least, Eclipse makes me comfortable when programming with it. I suggest to select the option of automatic building when you type code in Eclipse. If so, you are prompted potential errors and warnings when you type. I love the feature a lot since it must avoid much effort to fix them together until compiling programs explicitly. As a man, it is always easy to make mistakes when typing code. An instant prompt must be very helpful.

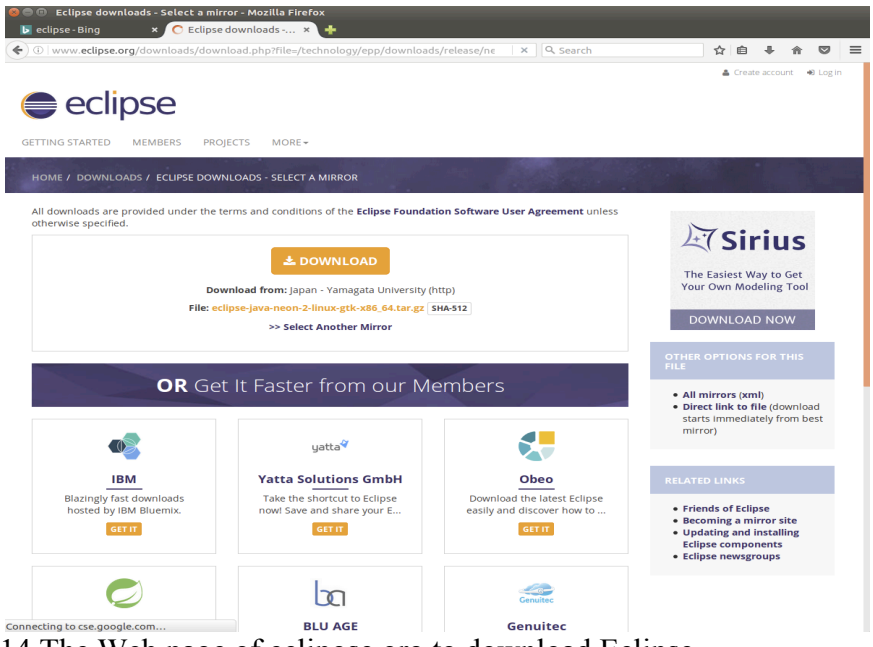


Figure 2.14 The Web page of eclipose.org to download Eclipse

Compared with IntelliJ IDEA, although Eclipse cannot offer programmers sufficient intelligent code options, I think it is good enough. IntelliJ IDEA does not support automatic building instantly when programming. If it does, I would rather use IntelliJ IDEA. So after comparing, I believe Eclipse is my final selection. You can determine which one you like the best. Anyway, an IDE is really an important tool for a programmer since you need to spend many long nights with it if you are ambitious to build a wonderful application like me.

If you think Eclipse is your choice as well, you can access the site, eclipse.org, to download it, as shown in Figure 2.14.

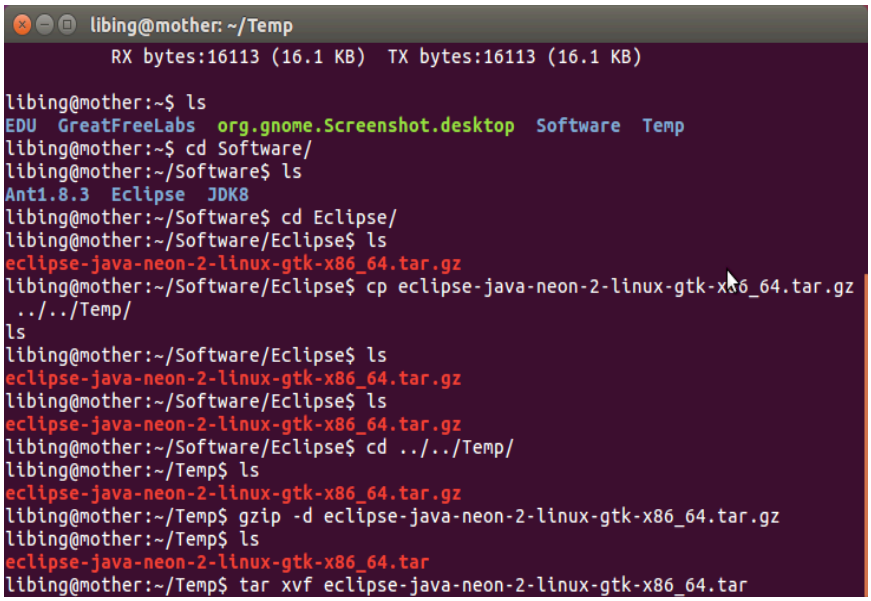


Figure 2.15 The manipulations to decompress and extract Eclipse

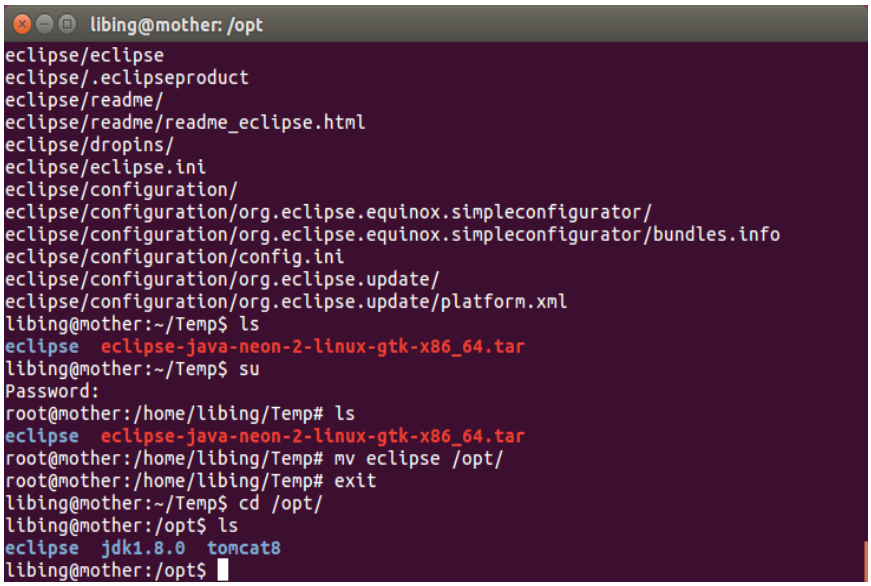


Figure 2.16 Eclipse is installed and put under the directory, /opt

After an appropriate Eclipse package, for example, eclipse-java-neon-2-linux-gtk-x86_64.tar.gz, is downloaded, you are also recommended to make a copy of it and move it a clean directory, such as ./Temp, as what I do in Figure 2.15.

For example, the software package of Eclipse is put into a folder named /home/libing/Software/Eclipse/, in which you would like to keep all your downloaded software. You can follow the following steps to install Eclipse. You are required to decompress as well as extract it. After that, it is time to move the directory, /opt, as shown in Figure 2.16. Assume your current directory is /home/libing/Software/Eclipse and the clean folder is located at /home/libing/Temp.

$ cp eclipse-java-neon-2-linux-gtk-x86_64.tar.gz ../../Temp/

$ cd ../../Temp/

$ gzip –d eclipse-java-neon-2-linux-gtk-x86_64.tar.gz

$ tar xvf eclipse-java-neon-2-linux-gtk-x86_64.tar

1.5 Apache Ant

Apache Ant [24] is also a mandatory software for you to install before moving forward. Ant helps us automate our source build processes just like Make [28] for C language [29]. As I say before, a programmer is required to be familiar with the command-line environment when debugging source code. Ant is the tool that builds and executes our source code in a terminal of Unix/Linux or Windows. If you are not familiar with how to use Ant, you have to spend some time on that. If you do not have much time, you just follow my samples of the book to learn the basics in the later sections.

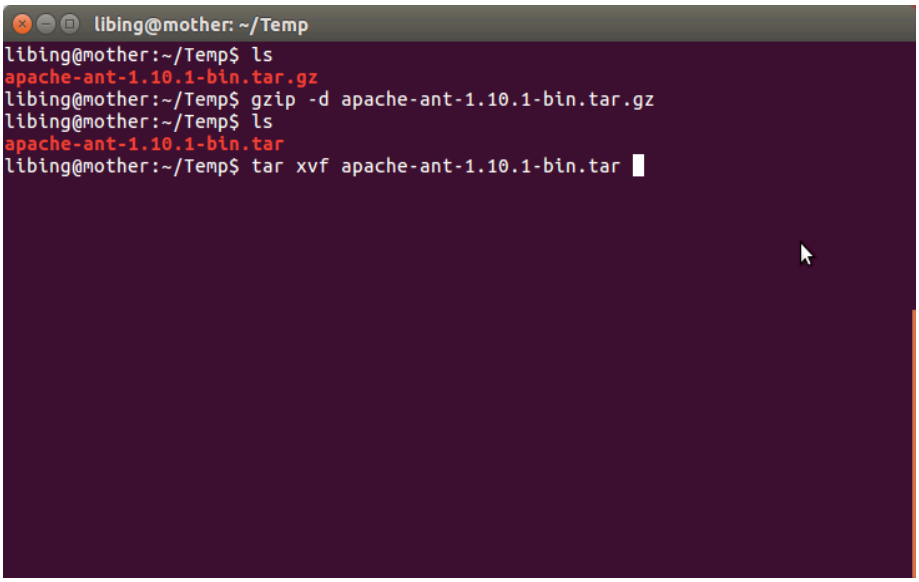


Figure 2.17 The steps to install Ant

Ant is easy to install as well. In fact, you just need to decompress its package and specify its path in Linux profile, as shown in Figure 2.17, Figure 2.18 and Figure 2.10.

It is necessary to emphasize that you are recommended to download the version of Ant at high as possible. With the continuously upgrading of JDK, the low version of Ant might not be compatible with it. For now, if you decide to program with JDK 8, you are forced to use Ant in the version of 1.9.0 at least. In our case, we choose the latest one, Ant 1.10.1.

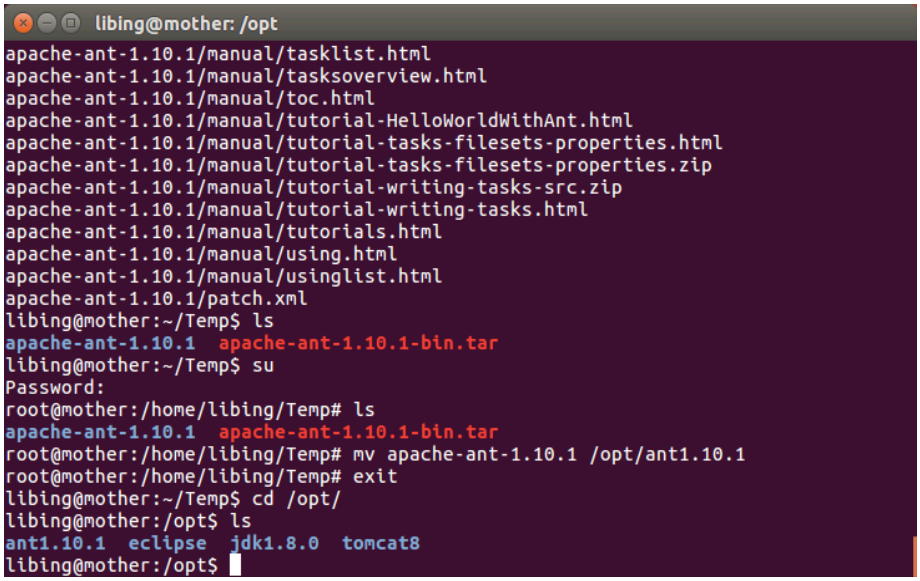


Figure 2.18 Ant is installed

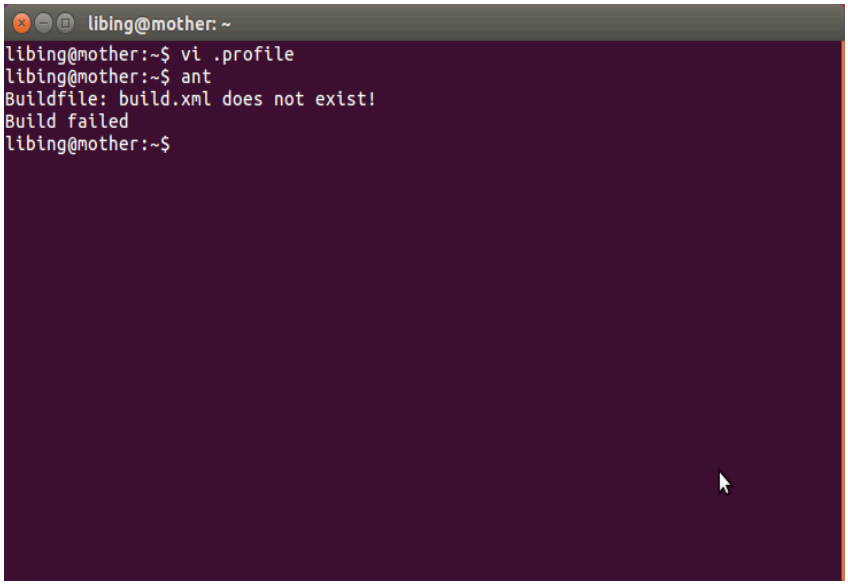


Figure 2.19 Ant is executed without build.xml

After downloading the Ant package, first you need to decompress it.

$ gzip -d apache-ant-1.10.1-bin.tar.gz

Then, extract the tar file by the below command.

$ tar xvf apache-ant-1.10.1-bin.tar

Similar to JDK and Eclipse, the installed Ant should be moved under /opt as well.

After Ant is installed, you need to revise your profile. When doing that, you are required to specify a new environment variable, ANT_HOME, in which Ant is located. In addition, the location of Ant commands must be added into PATH. For all of the Java systems, the variable CLASSPATH is required. For Ant, the libraries it provides is located at $ANT_HOME/lib. Moreover, the variable ANY_OPTS is also recommended to be added into the profile. Do not forget exporting those variables, finally.

After all of the above steps are performed, you can run the command, Ant, at an arbitrary location by typing ant within a terminal. Unless the configuration file, build.xml, is located in the same path, you cannot get a correct prompt. What you see is that the build.xml is not existed as shown in Figure 2.19. That means you are not ready to execute Ant since a build.xml is required to be located at the current directory where Ant is executed. You can find samples of build.xml later. Just follow those samples and work with most practical cases when programming clouds and distributed systems with GreatFree. Do not worry at this time.

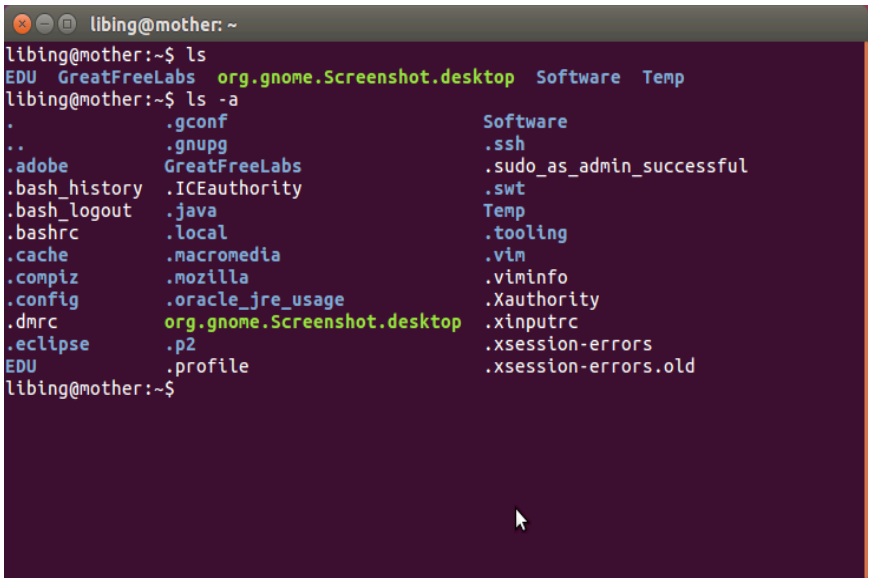


Figure 2.20 .profile is invisible by default

1.6 Your Profile

As mentioned previously, each account in Unix/Linux has a profile, which is used to define a specific environment for you when you enter Unix/Linux with the account. In our case, Ubuntu, the profile is named .profile, which is invisible by default. You can display by typing the below command and editing it by vi [30], as shown in Figure

2.20. Your profile should be roughly like the one shown in Figure 2.10. It is fine that some ones are different. But you need to be aware of the ones mentioned above should be listed in your profile, including JAVA_HOME, ANT_HOME and PATH.

2. The Coding Environment

Similar to the environment setting up, before discussing the programming procedure, it is necessary to indicate that those are the steps I follow when I program. If you are a senior programmer, you must have your own preferences. Then, you can ignore what I present. However, if you are a junior, I highly suggest you to follow them as me. Although you must find other better ways in your future professional days, what I discuss below is a good start for you to more forward.

As discussed before, programming GreatFree is a convenient procedure. In most time, you just need to work with the three behaviors, copy-paste-replace (CPR). You might not understand exactly what I am talking about at this moment. It is fine. You will know about that soon. But you must notice that the first behavior is copy. Have you raised a question in your mind? What should I copy? That is a good question. Now if you open Eclipse, nothing exists in it. You have no any sources to copy. That is what we should do in the next steps.

Finally, although Ubuntu is a best choice for your distributed systems or clouds to run for its cheapness and high quality, I suggest you to prepare a better machine to code. A programming machine should be different from a testing or running one since you need to sit down before it and you also need to think and type all the time. Therefore, it should be as comfortable as possible. For me, I choose iMac to program. So in the later sections or chapters, Mac OS X becomes the primary interface to show you how to program with GreatFree. The downfall of Mac OS X is that Apple machines are expensive. As a teacher, I always suggest students to buy used desktops or laptops instead of new ones. My advisor, Professor Wu [31], indicated in his book [31] that almost famous computing systems, such as Unix and Fortran [32], were implemented on used machines or even the ones abandoned by others. Thus, I will keep on setting up your coding environment over Ubuntu although most time I program with iMac.

2.1 Preparing the GreatFree Code and Packages

To start to learn programming with GreatFree, I have prepared for some fundamental code and packages for you to begin with. You can either download them from the link [33] or get them from the CD attached with the book. All of them is enclosed in a tar file, greatfree.tar, which you are required to extract. I need to say that this is just one portion of the code. Later, you will get more code to enjoy the programming methodology.

To extract it, type the command as follows. The tar file, greatfree.tar, should be located in the current path where you type the command. Otherwise, you are required to specify the location of the tar file, as shown in Figure 2.21.

```
$ tar xvf greatfree.tar
```

Figure 2.21 Extract greatfree.tar

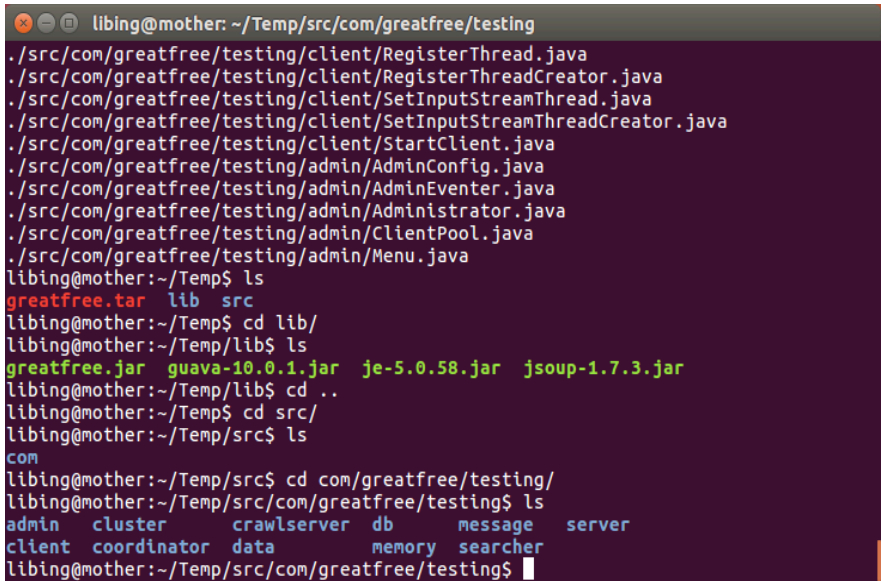


Figure 2.22 greatfree.tar is extracted

Once if it is extracted, as shown in Figure 2.22, two directories are emerged in the current path as follows.

./lib
./src

The directory, lib, contains four Java packages, which are specified as follows.

greatfree.jar, the library of GreatFree that contains the APIs to support your programming distributed systems or clouds

guava-10.0.1.jar, Google developed Java libraries that extend standard JDK

je-5.0.58.tar, Oracle Berkeley object-oriented database for Java

jsoup-1.7.3.jar, an HTML parser

If you are not familiar with them, do not worry and just follow my descriptions to set up your development environment. It does not affect your learning programming with GreatFree.

Just remember the directory, which is /home/libing/Temp/ in our case, that keeps GreatFree code and libraries. You need to import them into Eclipse soon.

2.2 Creating a Java Project in Eclipse

When you first run Eclipse, you can type the below command if you are in the path, /opt/eclipse, where Eclipse is installed, as shown in Figure 2.23. If not, you need to type the path first. The symbol, &, notifies the system that Eclipse is started asynchronously such that the current terminal is not blocked for the execution of Eclipse.

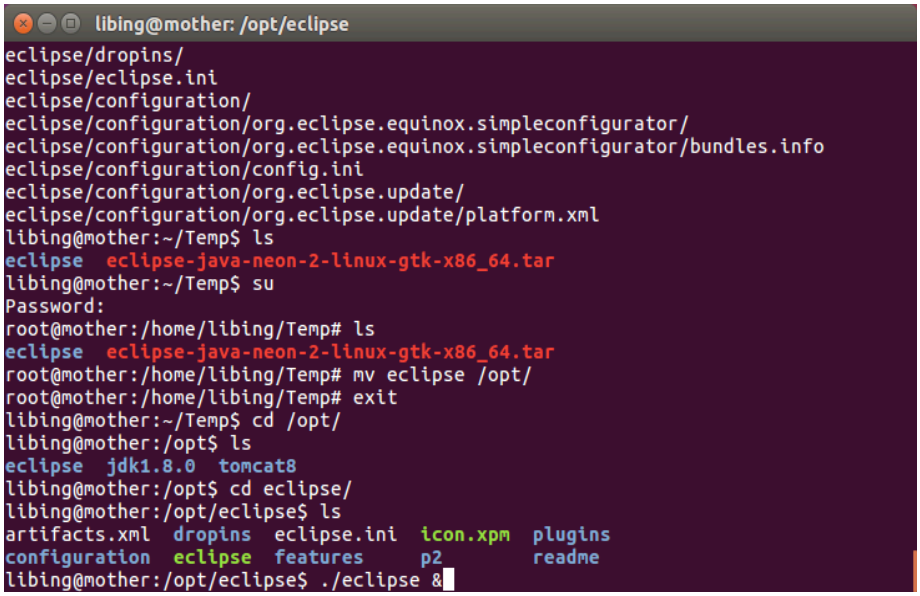


Figure 2.23 Execute Eclipse asynchronously

$ ./eclipse &

Or

$ /opt/eclipse &

When the above command is executed, you see a screen like Figure 2.24.

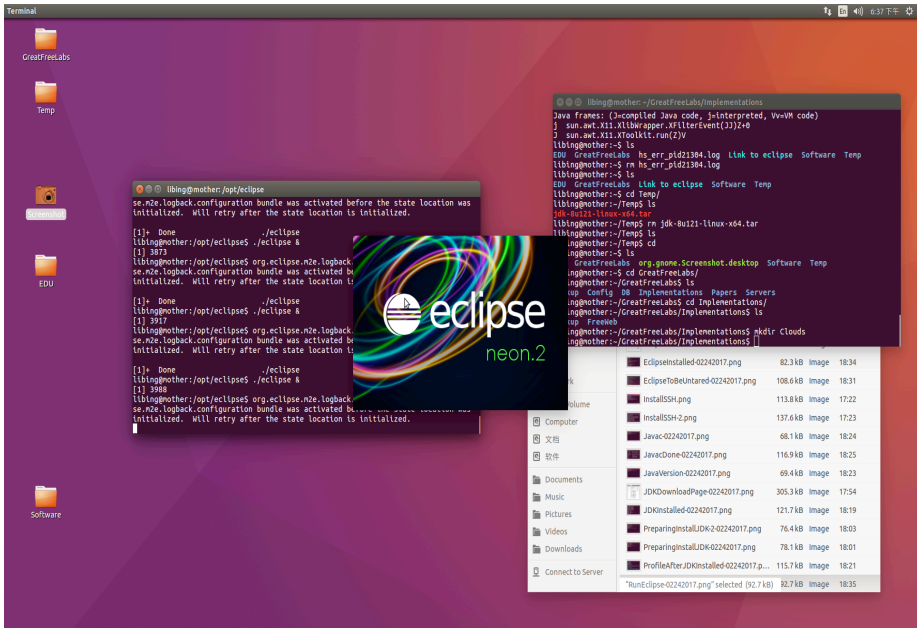


Figure 2.24 Eclipse is to be started

Then, you are prompted to specify the working space, where all of your source code and libraries are located, as shown in Figure 2.25.

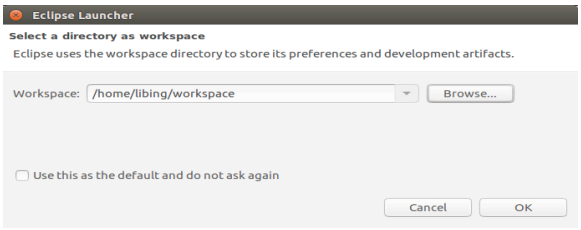


Figure 2.25 The dialog that allows you to specify your working space

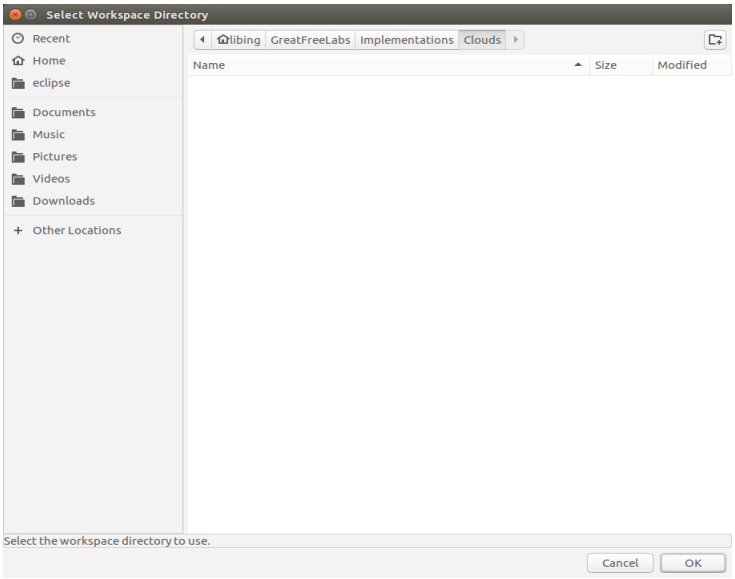


Figure 2.26 Specify the working space

In my case, a directory, /home/libing/GreatFreeLabs/Implementation/Clouds, is created prior to the starting up of Eclipse. You can also do that at this moment. Then, specify your working space to the directory you just created by clicking the button, Browse, as shown in Figure 2.26 and Figure 2.27. After that, just click the button of OK and your Eclipse is opened as shown in Figure 2.28.

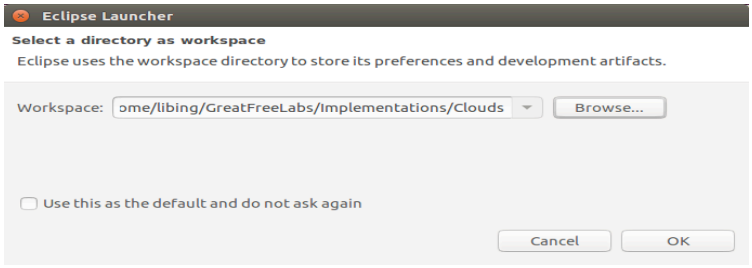


Figure 2.27 The working space is specified

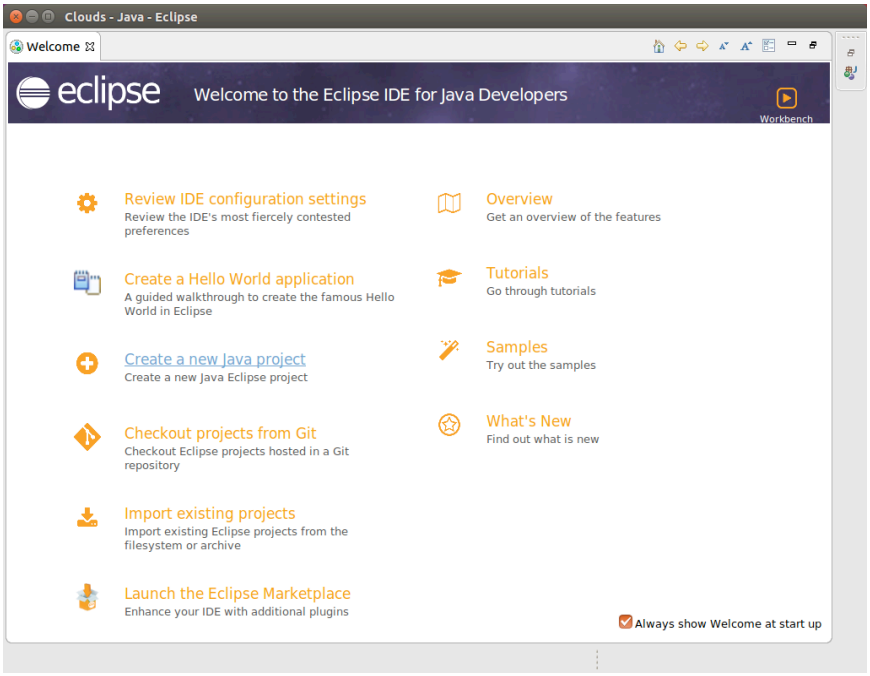


Figure 2.28 Eclipse is opened

Although Eclipse is opened, you can do nothing unless a project is created. In the book, all of the projects to be created are Java ones. To do that, you just right-click in the perspective of Package Explorer and then a hierarchical menu is popped up for you to create a new Java project, as shown in Figure 2.29.

Once if the option, Java Project, is selected, a dialog is popped up for you to specify details about the project. First, you need to fill the name of the project into the text field in the top of the dialog, as shown in Figure 2.30. What I did is to name the project as Clouds. You feel free to name your project as you like. Then, just click the next button without updating anything in the dialog, which is shown in Figure 2.31. Finally, you just click the button, Finish, to complete the project creation. That is what you see in Figure 2.32.

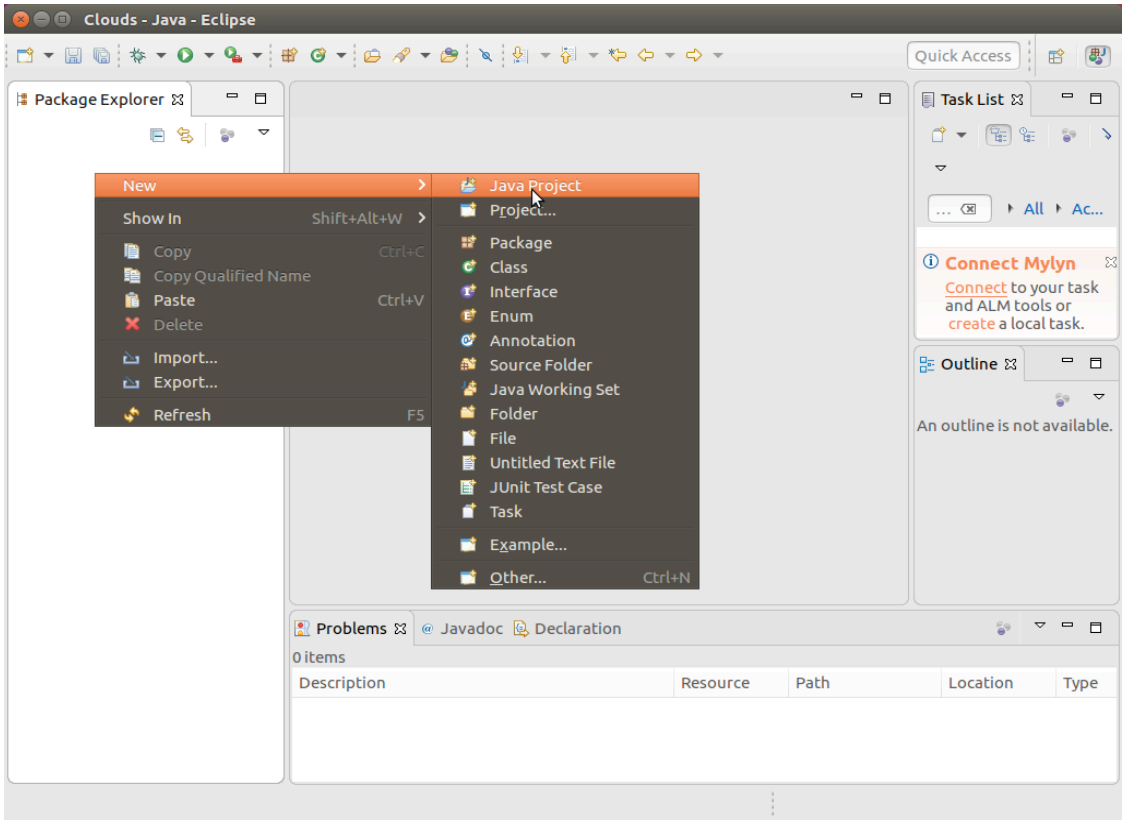


Figure 2.29 Right-click on the perspective of Package Explorer to create a new Java project by selecting the option from the hierarchical popped up menu

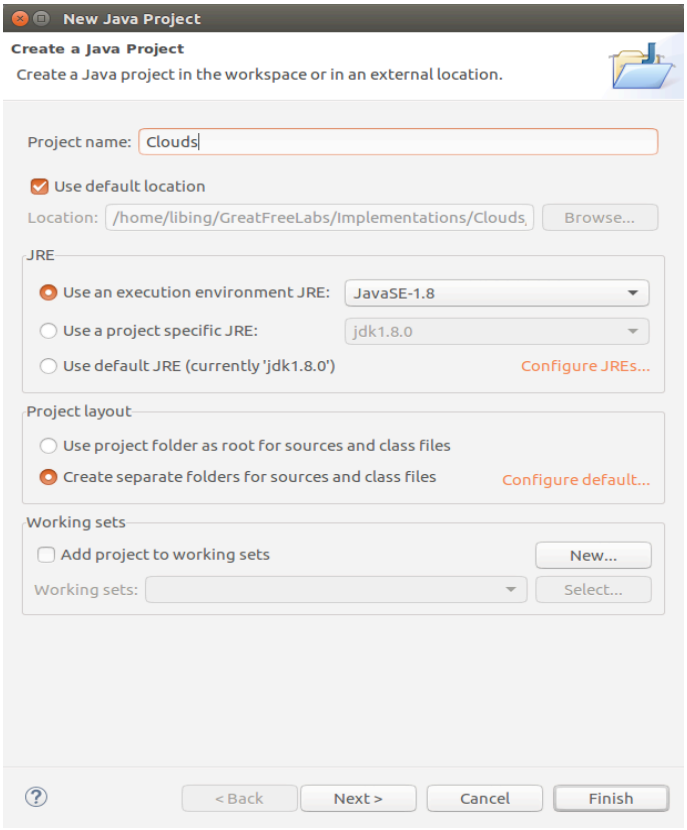


Figure 2.30 Fill the text field of Project name

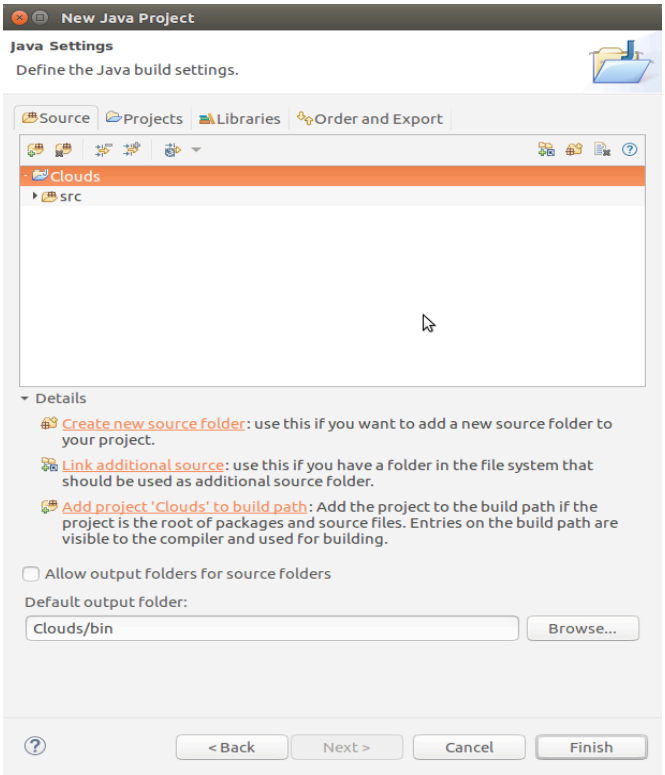


Figure 2.31 A Java project is to be created

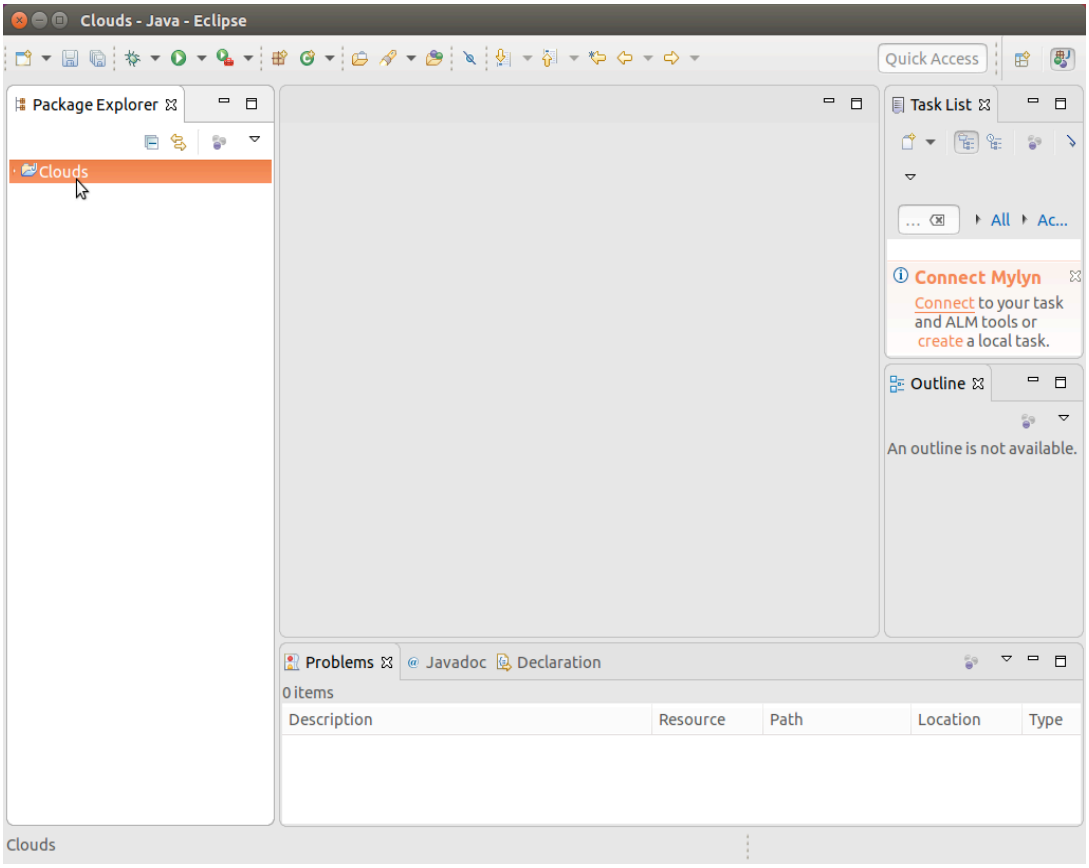


Figure 2.32 A Java project is created

2.3 Importing GreatFree Code and Libraries

After a Java project is created, you can start to program any applications with Java SE. Since we need to learn how to project distributed applications or clouds with Java, you still need to do additional setups for your development environment. Now you can start to import GreatFree Code and libraries, which ease the procedure to achieve the goals.

First, right-click on the icon, src, which is just beneath of the project name, Clouds. You should make sure to click on it. Otherwise, the imported files must be located in a wrong place. You can follow Figure 2.33 to do that.

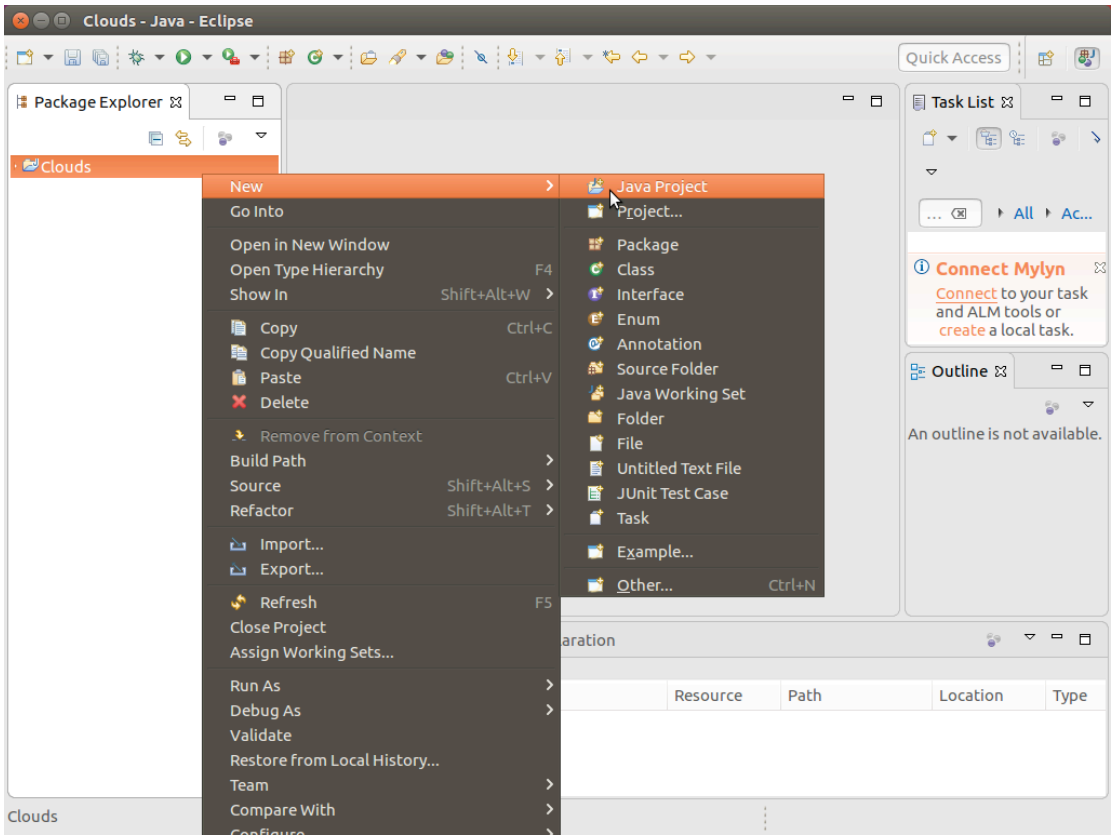


Figure 2.33 Right-click on the icon, src, and then select the option, Import, from the popped up menu

Once if the Import is clicked, a new dialog is popped up. You can follow Figure 2.34 to choose the option, File System, under the General option. That means you are ready to import source code from a certain directory of the current operating system, which is Ubuntu in our case.

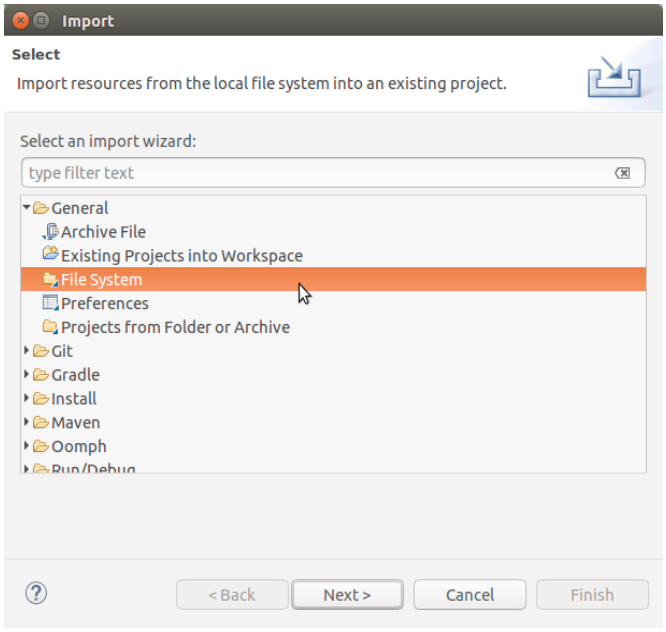


Figure 2.34 Select the option, File System, under General to import source code from

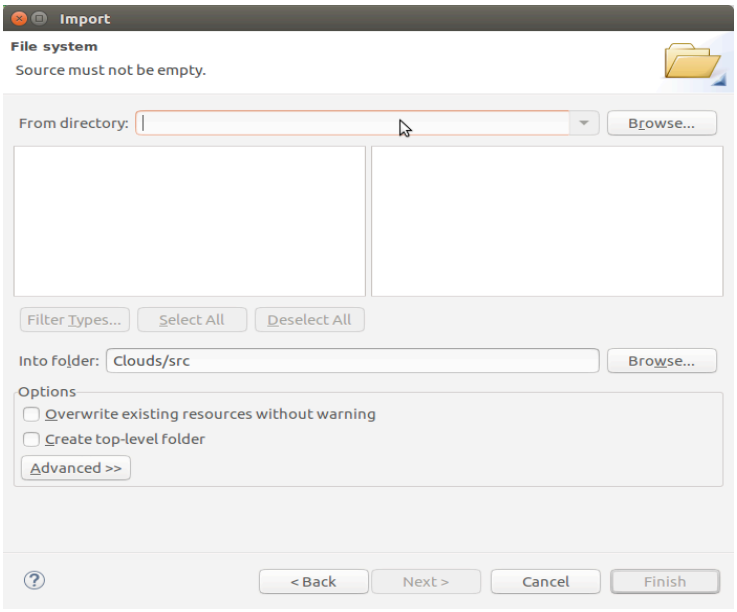


Figure 2.35 The dialog to browse the file system and locate the directory where the source code to be imported is saved

Then, click the button, Next, you can see the dialog as shown in Figure 2.35, in which you can browse the file system of Ubuntu and locate the directory in which the source code is saved. In our case, the directory to be located is /home/libing/Temp, in which GreatFree code and libraries are retained. You can refer to Section 2.1 and Figure 2.22 for the information. You just need to single-click on the directory of src which contains GreatFree code as shown in Figure 2.36. Do not enter into the directory.

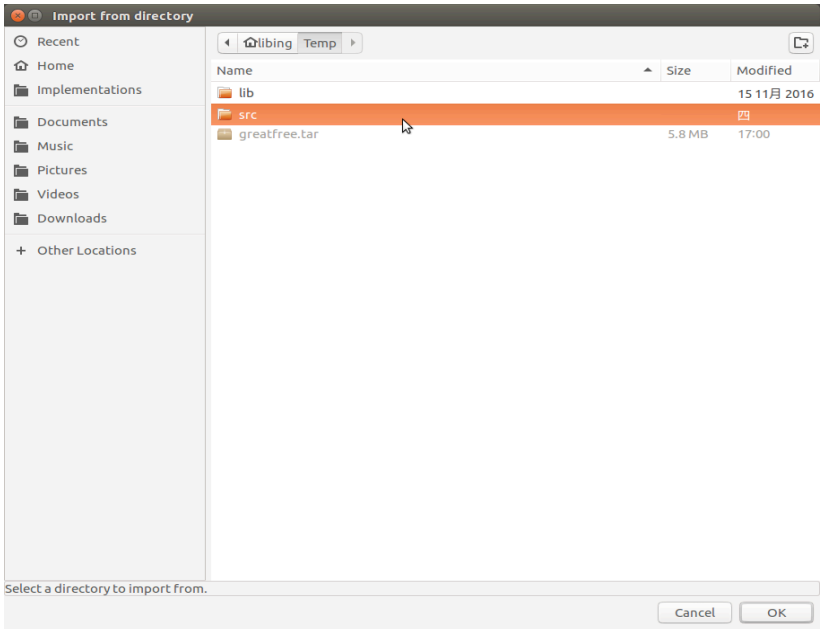


Figure 2.36 Locate the directory of src, which contains GreatFree source to be imported

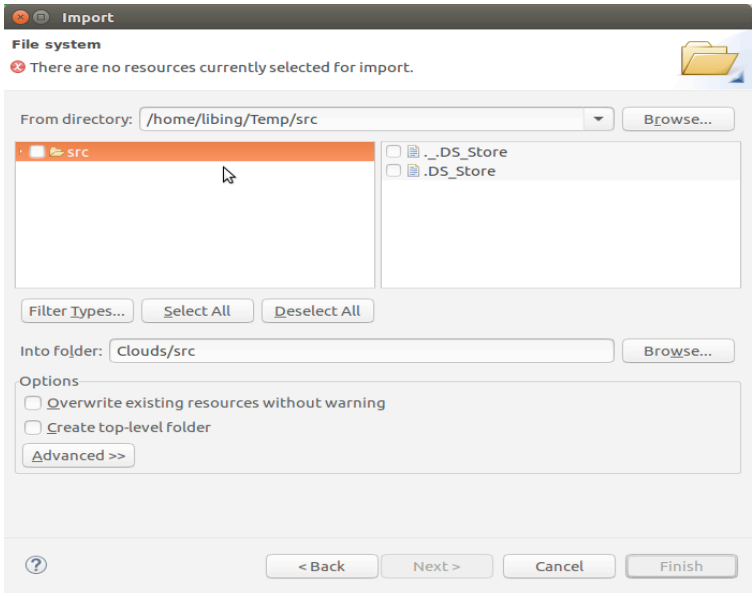


Figure 2.37 An error message is shown in the source import dialog

Thereafter, you just click the button, OK, to continue the import. After that, you return to the previous dialog for import, as shown Figure 2.37. Unfortunately, there is an error message in the dialog. It says “There are no resources currently selected for import”. To move it, you need to select the checkbox immediately left to src to remove the error message. It is possible that some other files are selected on the right window. In our case, they are .DS_Store something like that. No matter what the files are, all of them should be deselected. That is what you see in Figure 2.38. Finally, you need to click the button, Finish, to complete the source import procedure.

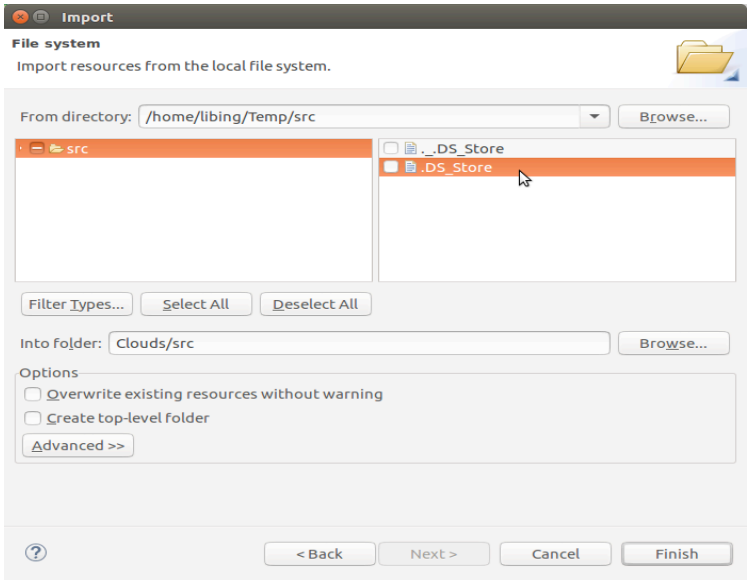


Figure 2.38 Finalize the options for the source import

Unfortunately, you must feel painful because you see each package imported has one red-cross icon that indicates the source code in the package contains compilation errors, as shown in Figure 2.39. Do not worry about that since one additional task is not done yet. To set up the coding environment, you are required to import external libraries as discussed in Section 2.1 and shown in Figure 2.22.

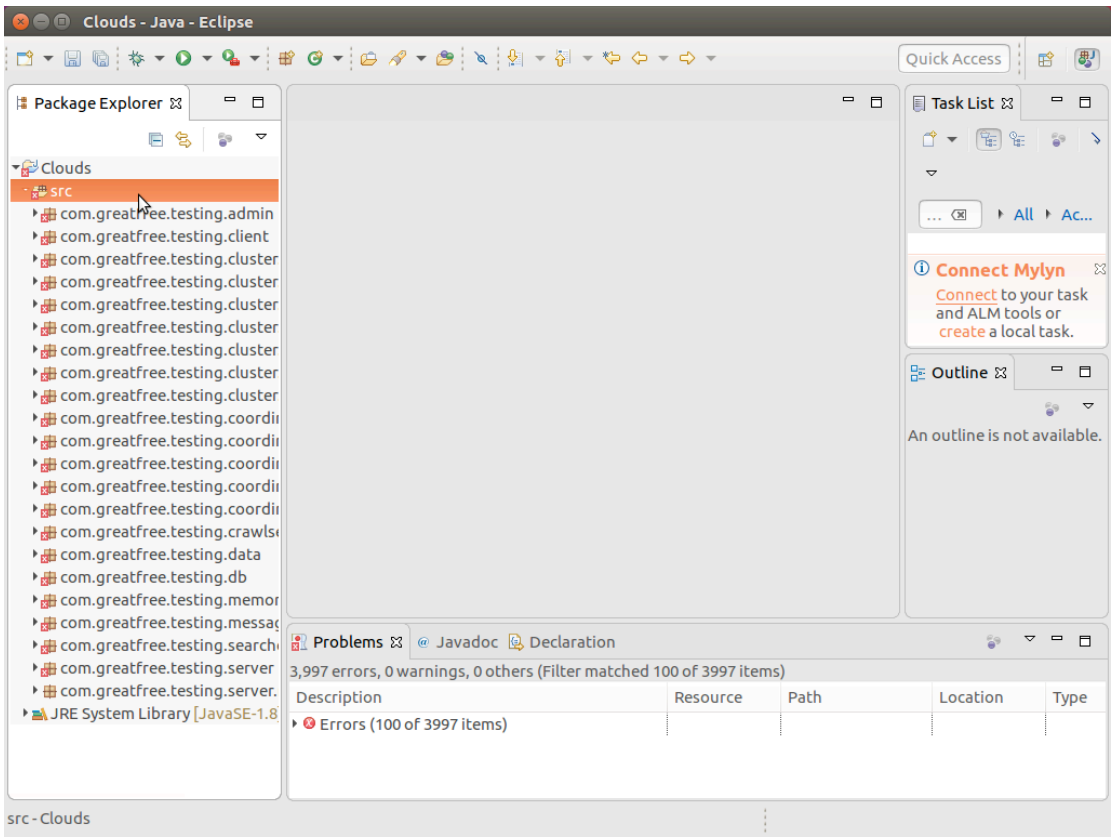


Figure 2.39 Each imported Java package contains compilation errors

Before explaining how to import GreatFree libraries, it is necessary to indicate that it is weird that the above steps do not work on Eclipse on Mac OS X. It is really annoyed, right? Fortunately, a convenient alternative approach can accomplish the task, i.e., drag-and-drop. Instead of selecting source code following the above steps, you can drag the com directory, which is immediately under the one of src, to the perspective of Package Explorer directly and drop it under the node of src, show in Figure 2.40 and Figure 2.41.

After the operation of drag-and-drop, you will be asked whether to link the source code files and folders or to copy them. You are required to copy them there. That is what you see in Figure 2.42. After that, you can see all of the source code is imported and the Eclipse interface is identical to what you see in Ubuntu, as shown in Figure 2.43.

To import external libraries, you should right-click on the project name in the perspective of Package Explorer. In my case, the name of the project is called Clouds. Then, a menu is popped up such that you can choose the option, “Properties”, as shown in Figure 2.44. The next dialog is shown in Figure 2.45, in which you need to select the option, Java Build Path, from the left-hand side. Moreover, the tab, “Libraries”, should be clicked. To continue, you should click the button, Add External JARs.

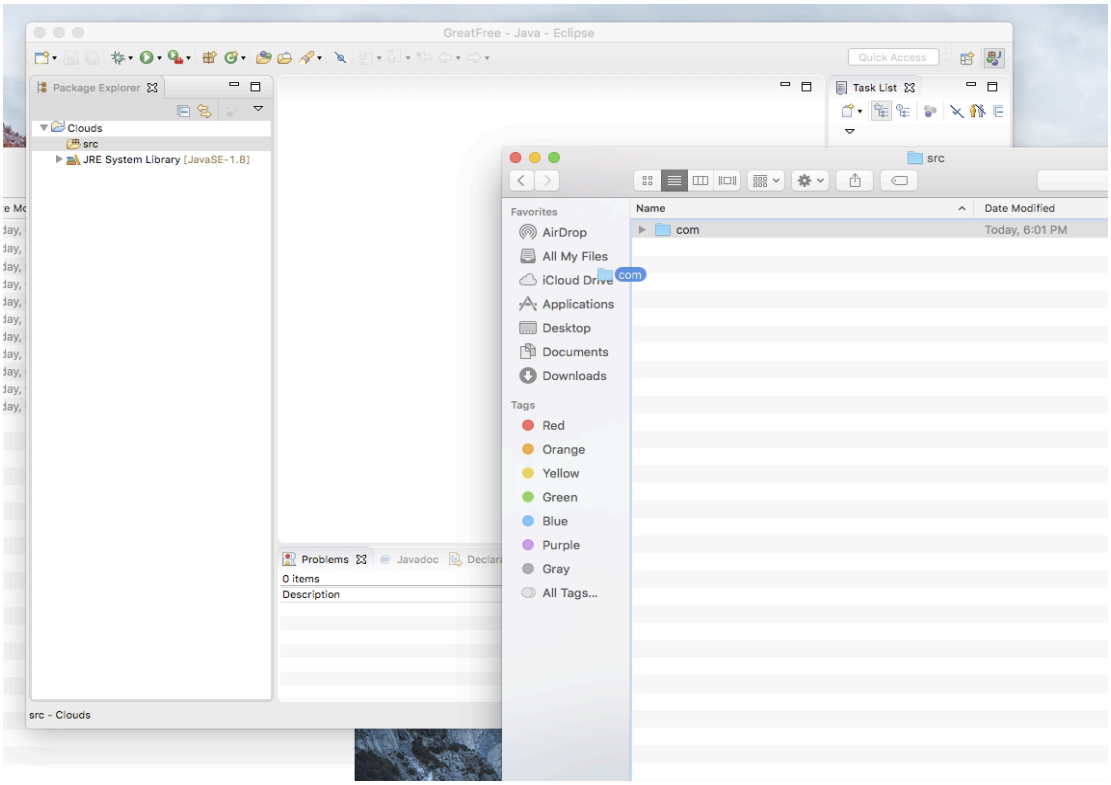


Figure 2.40 In Mac OS X, drag the source code to the perspective of Package Explorer

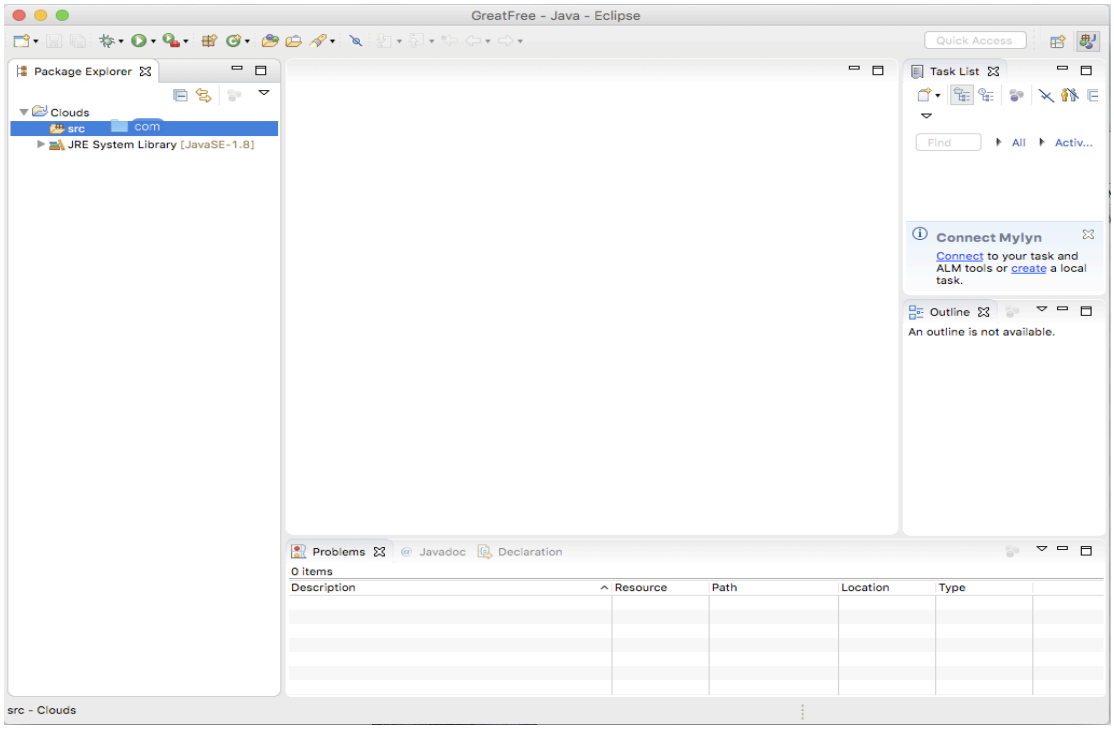


Figure 2.41 In Mac OS X, drop the source code directory, ./com, under the node of src in the perspective of Package Explorer

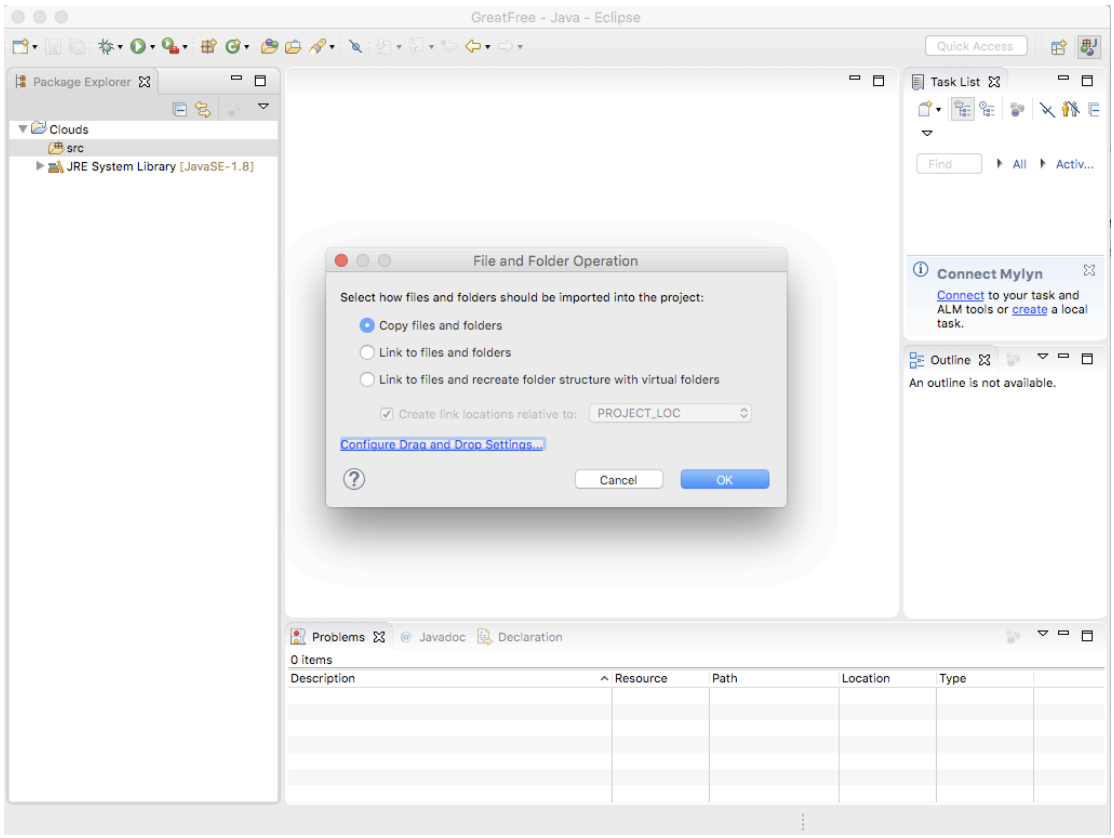


Figure 2.42. Confirm whether to link or copy the source code files and folders

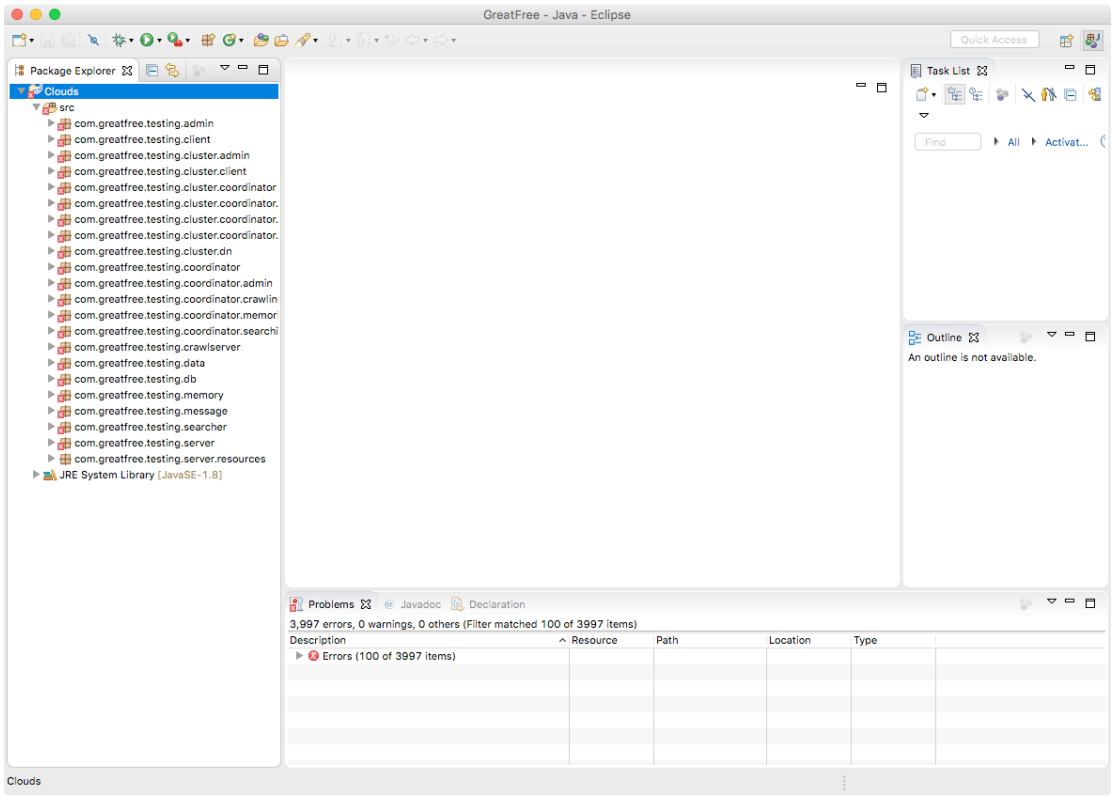


Figure 2.43 The source code is imported into Eclipse on Mac OS X

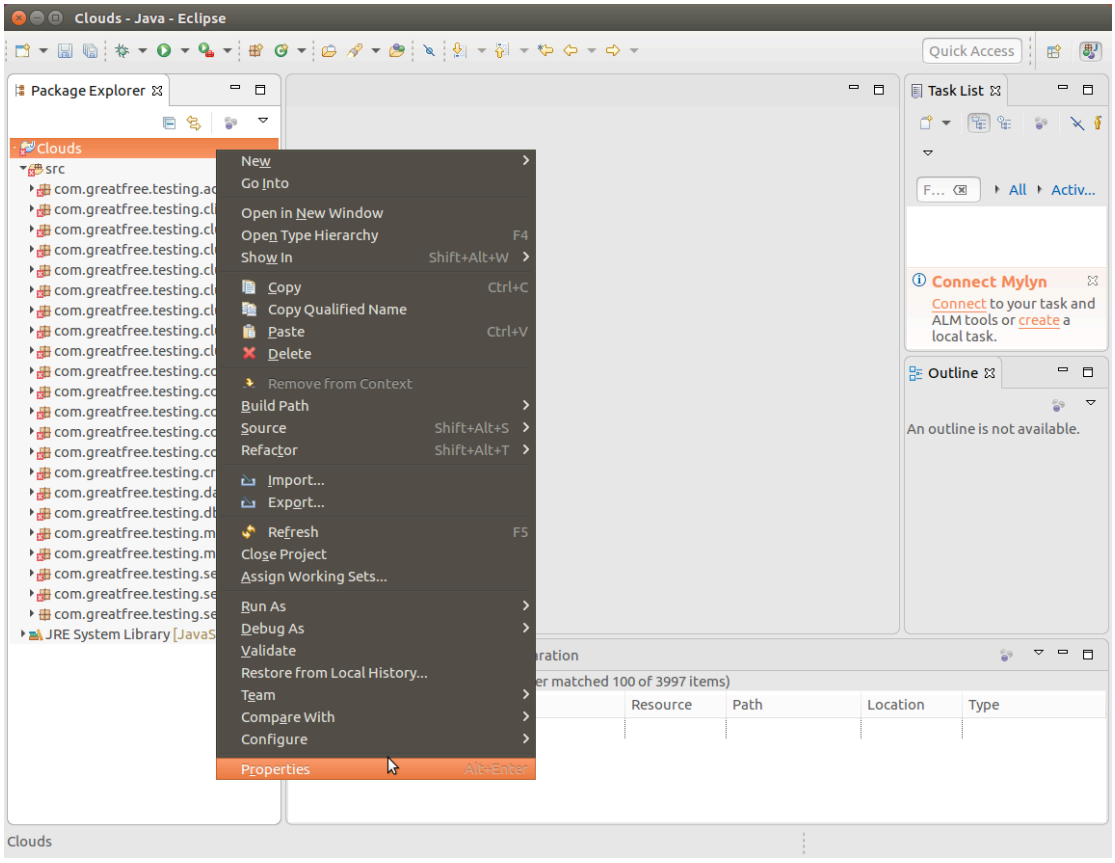


Figure 2.44 Select the option, “Properties”, to import external libraries

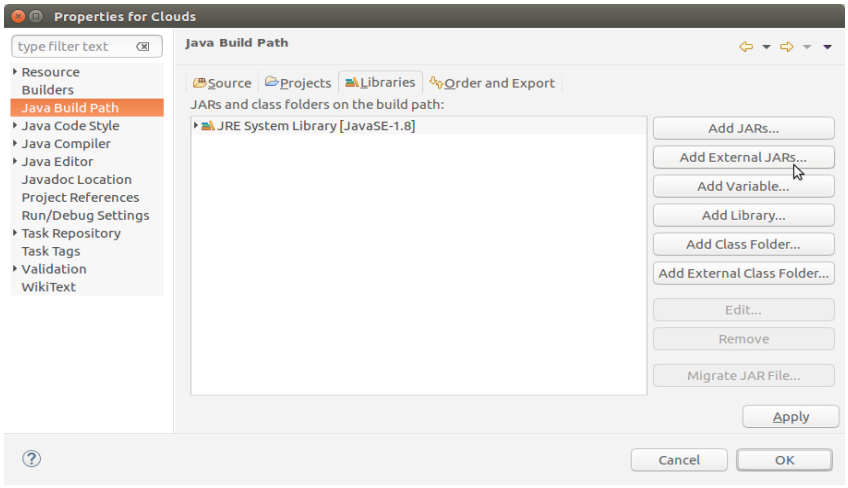


Figure 2.45 The operations to add external libraries to your current project

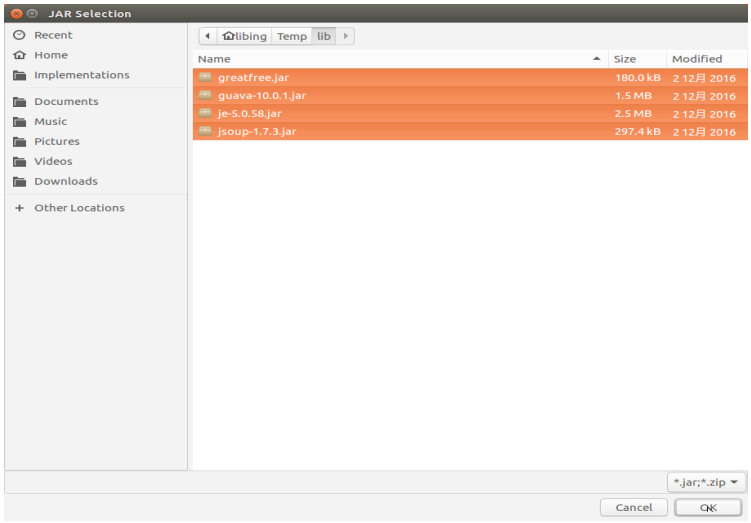


Figure 2.46 Select GreatFree libraries to import into your project

Now recall Section 2.1 or follow Figure 2.22 and go to the directory where GreatFree libraries are retained. In my case, the directory is /home/libing/Temp. All of four libraries should be selected for the import, as shown in Figure 2.46. Once if the button of OK is clicked, you return to the original dialog, Properties for Clouds. Different from the previous one, the four libraries to be imported are listed in the main view of the dialog. To finish the procedure, just click the button of OK again. That is the final step to import external libraries to your project, as shown in Figure 2.47.

Now you must be aware that the four libraries are listed under the node, Referenced Libraries, in the perspective of Package Explorer. More important, all of compilation errors are gone at this moment, as shown in Figure 2.48. Congratulations, you are ready to program distributed systems or Clouds with GreatFree at this moment!

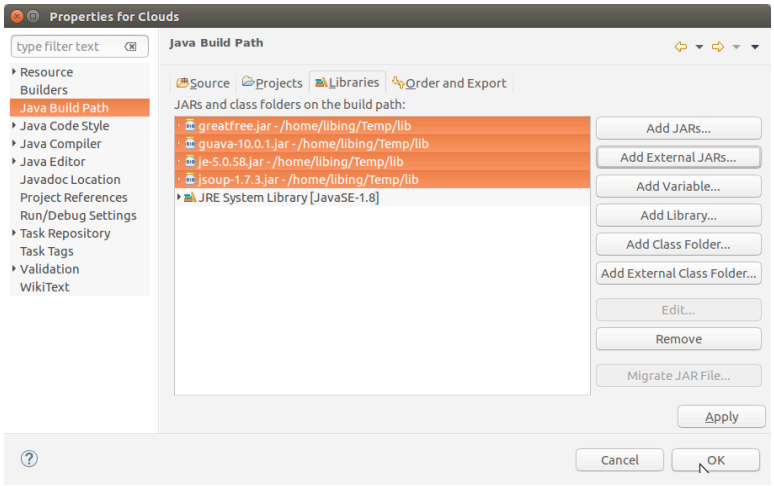


Figure 2.47 The libraries to be imported are listed in the dialog of Properties for Clouds

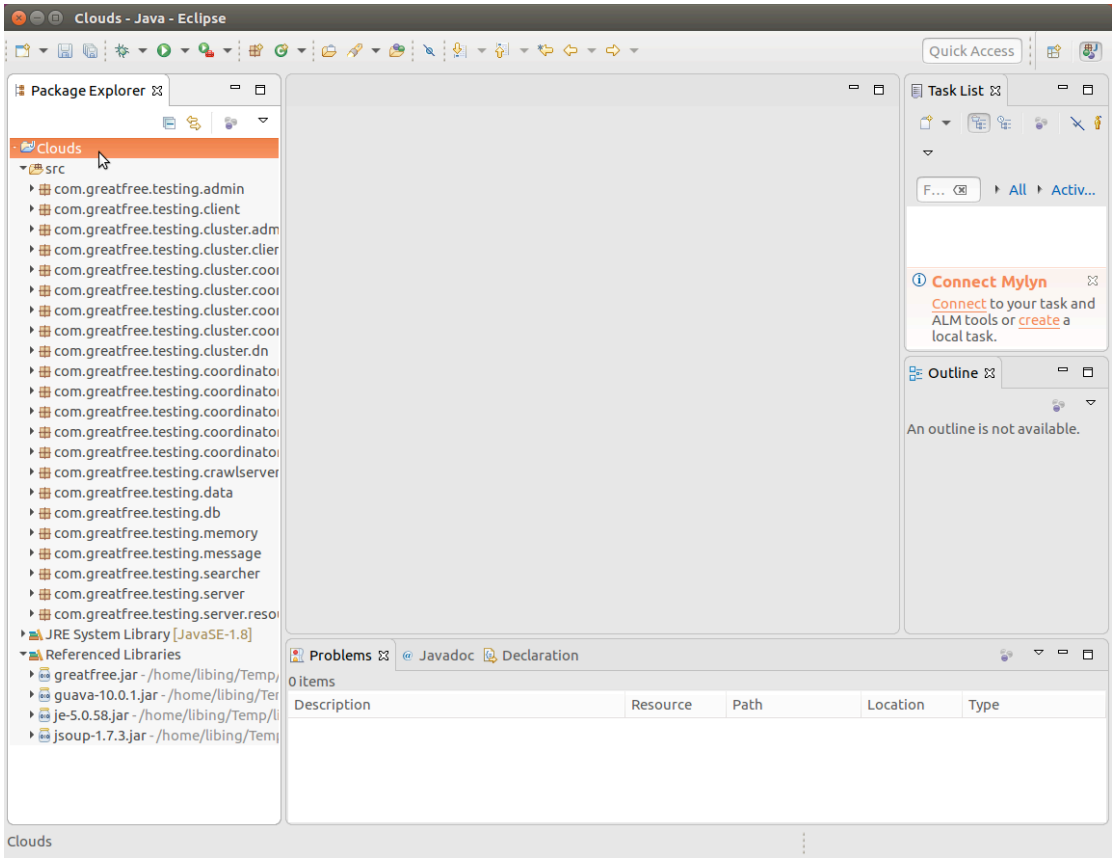


Figure 2.48 After GreatFree libraries is imported, all of compilation errors are gone

3. Testing

Although you have set up both of the system environment and the coding one, there is one important step for you to move forward. Now it is time for you to run the sample code you just imported and ensure further that you can really program. In addition, I should indicate that it is critical for you to learn how to debug and deploy in the environment that is just set. This is a professional circumstance that an excellent

programmer should be aware of. In the following chapters, all of our programming effort is performed in it. Finally, it is preferred that you prepare three PCs or more. That means a single computer is not good enough for you to taste how to program with GreatFree. As a professional engineer in Computer Science, multiple computers are the basic requirement, especially when you focus on distributed systems or clouds.

For me, I have more than ten computers that are under my control. The system environment is set up on all of them. Moreover, it is expected there are rich computing resources inside each of them since a distributed system or clouds might consume a lot. On the other hand, it is not necessary to set up the coding environment to all of them. As I mentioned before, when coding, you are recommended to have a desktop with a large screen although the CPU, memory and other configurations are not required to be too high. For me, an iMac with 27' screen is employed. With it, I can type all of my code, deploy programs and debug them remotely on other machines. Hence, you will see Mac OS X screen shots in the following sections or chapters. However, do not worry about that. When you program with Eclipse, everything is almost identical no matter whether you work on Mac OS X or Ubuntu. And even when deploying and debugging within a terminal, the command-line interfaces on any Unix versions are the same as well.

After you get ready for those machines, we will test a classic distributed system, i.e., Client/Server (C/S) [34], in the section. As a typical model, we will explain both of them in details in later chapters to help you understand how to program them with GreatFree. In the section, we focus on how to execute them to validate whether your environment is settled down.

3.1 The Ant Environment

In Section 1.5, although Ant is installed, it is executed without the file of build.xml. Actually, the file is the configuration one that defines JDK how to compile, execute and debug your Java source code. You are required to set up the environment and write a proper build.xml to test your code.

First, let us set up your Ant running environment. The Ant running environment usually contains three folders as follows.

build: The folder that contains the compiled Java byte code;

lib: The folder that contains all of the libraries that are used by source code;

src: The folder that contains all of source code to be compiled and executed.

Besides the above three folders, an XML file, build.xml, is also located in the Ant running environment. Figure 2.49 illustrates the environment, in which Ant is installed.

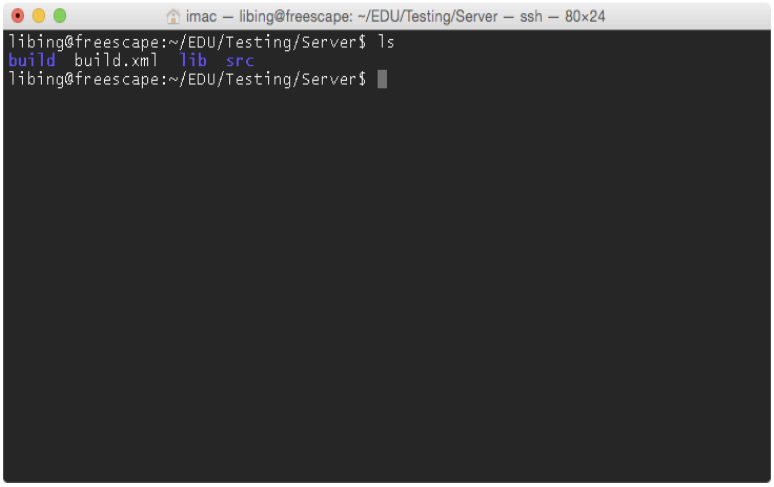


Figure 2.49 The environment in which Ant installed

Now let us install such an environment step by step and execute sample code to ensure it is installed successfully. Because we need to program distributed systems and multiple computers get involved in, we have to redo that on remote machines by typing commands rather than doing that on a local one. For that, the software, SSH, which we installed previously, has to be used.

In my case, I prepare for three PCs to do that. The first PC, Labs, is the one I need to program with Eclipse. It is a iMac desktop on which Mac OS X is installed.

The second one is named, Mum, which is also an Apple machine, MacBook Pro, with Mac OS X installed. Although Eclipse is installed on it, it is only used as one of remote testing nodes.

The third one is named, freescape, which is a ThinkPad laptop with Ubuntu. Eclipse is also installed on it. However, in the following procedure, it is used as another remote testing node only.

To interact with the remote testing nodes, Mum and freescape, SSH is the software we need to exploit. I need to indicate that as I said previously it does not matter even though you do not have multiple machines like me. At least, for the learning purpose, it is fine even if you have only one PC although it is not convenient for you to operate.

3.2 Accessing Remote Testing Nodes

First of all, you need to create a new directory, which is the work space for your future project deployment and testing, on both of the remote testing nodes, Mum and freescape. For example, we can name the folder as Testing. If we do that on the node, Mum, with SSH, we need to sign in it remotely by typing the following command. It assumes that 192.168.1.105 is the IP address [35] of Mum. In addition, we need to have an account on Mum. For example, the username of the account is greatfree.

$ ssh greatfree@192.168.1.105

Then, you are required to input the password to finish the sign-in procedure. After sign-in, you need to find a location on the disk space in which the account has full privileges. In this case, the folder, Testing, is created under the path /GreatFreeLabs, as shown in Figure 2.50.

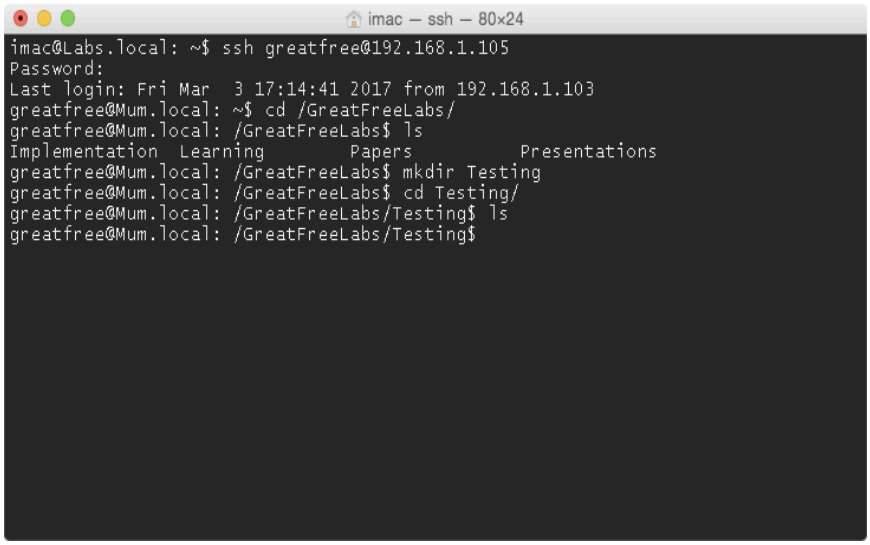


Figure 2.50 Create a new folder, Testing, on a remote testing node, Mum, on which Mac OS X is installed

For another testing node, freescape, you are required to do the same thing. Assume its IP address is 192.168.1.105 and then you sign in it with an account named, libing. The procedure is shown in Figure 2.51. Since we access both of the remote nodes from my local desktop, iMac with Mac OS X, you must notice that the terminals for both nodes have the same look-and-feel.

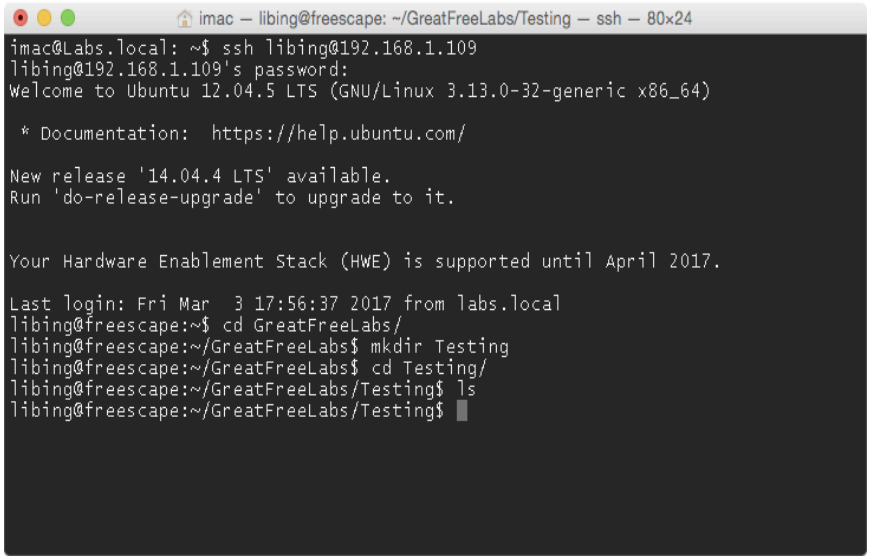


Figure 2.51 Create an empty folder, Testing, on a remote testing node, freescape, on which Ubuntu is installed

3.3 Setting Up a Server Node

Then, since we need to test our source code in the command-line environment, you need to set up projects under the folder, Testing, for the two remote nodes, respectively. In fact, it is identical to what we do in Eclipse although the procedure has no any GUIs (Graphical User Interfaces).

You need to create a new directory under Testing. Since we are preparing for testing a classic C/S system, a server node must be set up at first. To do that, create a new folder named Server under either of the remote nodes. Here, the term, Server, is the project name of your application to be programmed. For my case, I do that on the node, Mum, as shown in Figure 2.52.

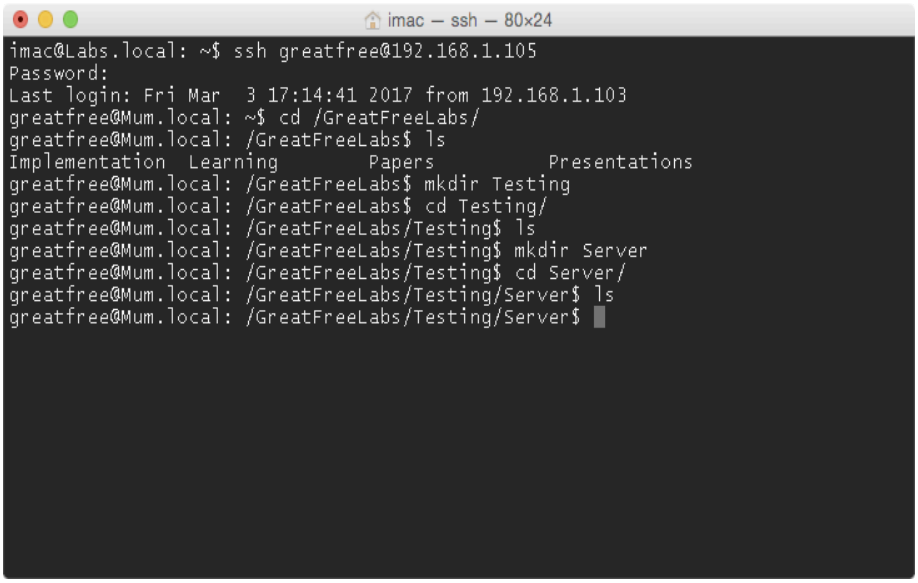


Figure 2.52 Create a new folder, Server, under Testing on the remote node, Mum

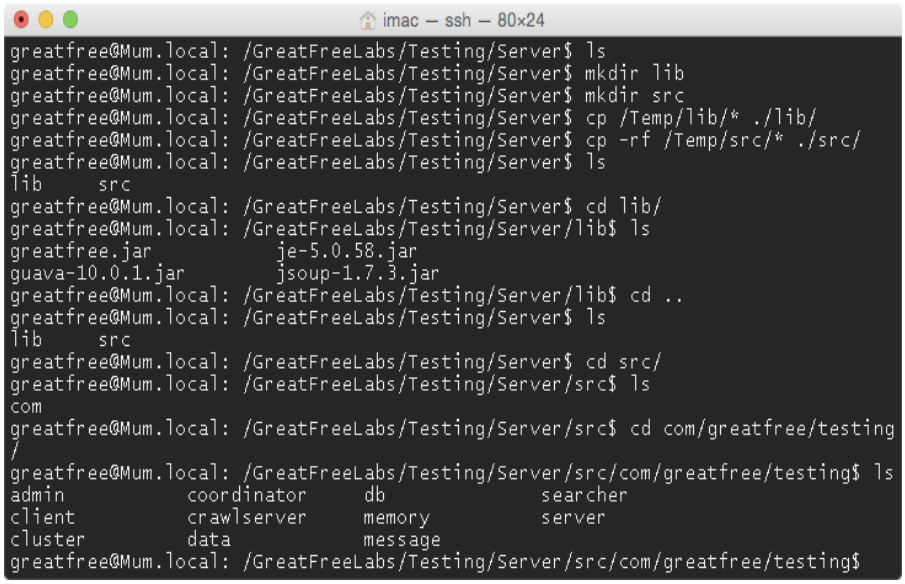


Figure 2.53 Copy libraries and source code to lib and src, respectively

Thereafter, create two folders, lib and src, under the folder, Server. According to their names, you must be able to guess their senses and usefulness. The one of lib is used for saving external libraries, which are the same as the ones we import in Eclipse. The one of src is used to save all of the source code. That is also included in the package, greatfree.tar. Hence, what you can do is just to simply copy the libraries and source code under the folders, lib and src, respectively. When you copy the source code, you should pay much attention that you should copy the source code with all of the directories under the one of com. When you finish it, you should have a terminal exactly like Figure 2.53.

You must now understand what is located inside the folder of src. Never mind at this moment. The structures of libraries and source code are actually identical to the ones in Eclipse.

For the server, you are still required to make a change to the source code since you just move to the code to a new directory. In my case, it is shown as follows.

/GreatFreeLabs/Testing/Server

Usually, it is possible for a system to operate a directory on the file system. It might get problems if the directory is updated when installing the system. To avoid the problem, you need to update the relevant configuration. In our case, Line 34 of the code, ServerConfig.java which is shown in List 2.1, should be updated. The line keeps the information about the path of the server. On the path, the server can access the local file system.

The source code, ServerConfig.java, is located under the path, ./src/com/greatfree/testing/data. You can update the code with any text editors. In my case, the server is located on a Mac OS X system such that I can update it with vi [30], a very classic text editor. I just go into the directory and update Line 34 with the correct value, as shown in Figure 2.54 and Figure 2.55.

Figure 2.54 Update the code, ServerConfig.java, with vi

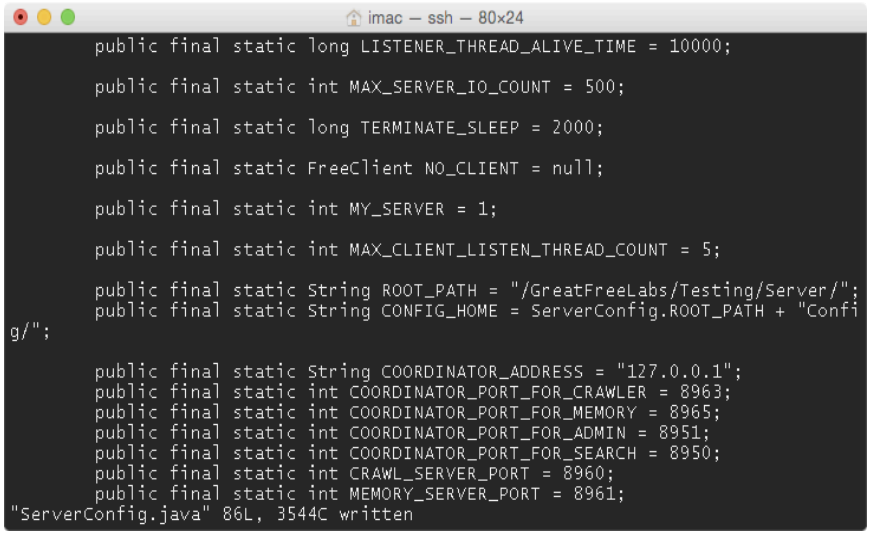


Figure 2.55 Line 34 is updated to the correct value

I do not recommend you to do that in Eclipse as we set up in Section 2.2. Since Eclipse manages a project in its own directory, which is different from those for debugging. In my environment, they even do not reside in the same PC. If you do that in Eclipse, you have to deploy the code to the remote node, like the one of Mum, again. It is so annoyed, especially when the change is so tiny as that case. Eclipse is exploited only until we need to write a lot of code and make a big change to the system.

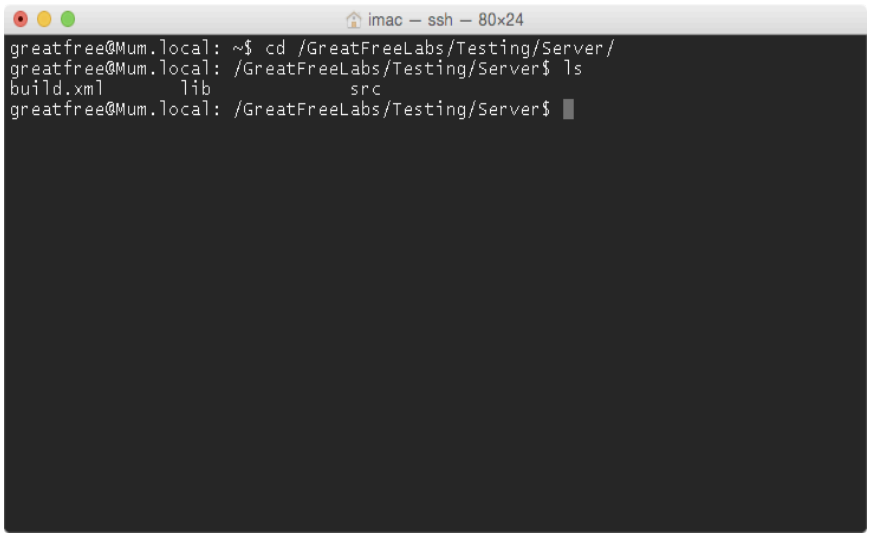


Figure 2.56 The Ant configuration file, build.xml, is saved immediately under the project directory, Server

```
1  package com.greatfree.testing.data;
2
3  import com.greatfree.remote.FreeClient;
4
5  /*
6  * The class keeps constants of the testing sample code. 08/04/2014, Bing Li
7  */
8
9  // Created: 08/04/2014, Bing Li
```

```
10  public class ServerConfig
11  {
12      public final static String SERVER_IP = "192.168.1.105";
13      public final static int SERVER_PORT = 8964;
14      public final static int ADMIN_PORT = 8947;
15      public final static int CLIENT_PORT = 8949;
16
17      public final static int DISPATCHER_POOL_SIZE = 500;
18      public final static long DISPATCHER_POOL_THREAD_POOL_ALIVE_TIME = 2000;
19
20      public final static int LISTENING_THREAD_COUNT = 5;
21      public final static int LISTENER_THREAD_POOL_SIZE = 50;
22      public final static long LISTENER_THREAD_ALIVE_TIME = 10000;
23
24      public final static int MAX_SERVER_IO_COUNT = 500;
25
26      public final static long TERMINATE_SLEEP = 2000;
27
28      public final static FreeClient NO_CLIENT = null;
29
30      public final static int MY_SERVER = 1;
31
32      public final static int MAX_CLIENT_LISTEN_THREAD_COUNT = 5;
33
34      public final static String ROOT_PATH = "/GreatFreeLabs/Testing/Server/";
35      public final static String CONFIG_HOME = ServerConfig.ROOT_PATH + "Config/";
36
37      public final static String COORDINATOR_ADDRESS = "127.0.0.1";
38      public final static int COORDINATOR_PORT_FOR_CRAWLER = 8963;
39      public final static int COORDINATOR_PORT_FOR_MEMORY = 8965;
40      public final static int COORDINATOR_PORT_FOR_ADMIN = 8951;
41      public final static int COORDINATOR_PORT_FOR_SEARCH = 8950;
42      public final static int CRAWL_SERVER_PORT = 8960;
43      public final static int MEMORY_SERVER_PORT = 8961;
44      public final static int SEARCH_CLIENT_PORT = 8948;
45      public final static int COORDINATOR_DN_PORT = 8946;
46      public final static int DN_PORT = 8945;
47
48      public final static long REQUEST_THREAD_WAIT_TIME = 1000;
49      public final static long NOTIFICATION_THREAD_WAIT_TIME = 1000;
50
51      public final static int REQUEST_DISPATCHER_POOL_SIZE = 50;
52      public final static long REQUEST_DISPATCHER_THREAD_ALIVE_TIME = 500;
53      public final static int MAX_REQUEST_TASK_SIZE = 500;
54      public final static int MAX_REQUEST_THREAD_SIZE = 50;
55      public final static long REQUEST_DISPATCHER_WAIT_TIME = 1000;
56      public final static int REQUEST_DISPATCHER_WAIT_ROUND = 5;
57      public final static long REQUEST_DISPATCHER_IDLE_CHECK_DELAY = 2000;
58      public final static long REQUEST_DISPATCHER_IDLE_CHECK_PERIOD = 2000;
59
60      public final static int NOTIFICATION_DISPATCHER_POOL_SIZE = 30;
61      public final static long
62          NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME = 2000;
63
64      public final static int MAX_NOTIFICATION_TASK_SIZE = 500;
65      public final static int MAX_NOTIFICATION_THREAD_SIZE = 10;
66
67      public final static long NOTIFICATION_DISPATCHER_WAIT_TIME = 1000;
68      public final static int NOTIFICATION_DISPATCHER_WAIT_ROUND = 5;
69      public final static long
70          NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY = 3000;
71      public final static long
72          NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD = 3000;
73
74      public final static int CLIENT_POOL_SIZE = 500;
75      public final static long CLIENT_IDLE_CHECK_DELAY = 3000;
76      public final static long CLIENT_IDLE_CHECK_PERIOD = 30000;
77      public final static long CLIENT_MAX_IDLE_TIME = 3000;
78
79      public final static long DISTRIBUTE_DATA_WAIT_TIME = 2000;
80      public final static int MULTICASTOR_POOL_SIZE = 100;
81      public final static long MULTICASTOR_POOL_WAIT_TIME = 1000;
82
83      public final static int ROOT_MULTICAST_BRANCH_COUNT = 100;
84      public final static int MULTICAST_BRANCH_COUNT = 16;
85
86      public final static int SCHEDULER_POOL_SIZE = 50;
```

```
87        public final static long SCHEDULER_KEEP_ALIVE_TIME = 5000;
88    }
```

List 2.1 ServerConfig.java

3.4 Creating the build.xml

One more step is required for the server node setting up. The file, build.xml, is mentioned a couple of times. Now it is time to create it such that you can execute the source code with Ant. You can download the build.xml for the server node from the link, [33], or get it from the CD attached with the book. You should be clear that each project has its unique build.xml. For the server node, the build.xml is presented in List 2.2. Be sure that the file, build.xml, should be saved immediately under the project directory, Server, as shown in Figure 2.56.

```
1     <project name="Clouds" default="main" basedir=".">
2         <description>This file is used to build the server node. 03/03/2017, Bing Li</description>
3         <!-- set global properties for this build -->
4         <property name="src.dir" value="src"/>
5         <property name="build.dir" value="build"/>
6         <property name="classes.dir" value="${build.dir}/classes"/>
7         <property name="jar.dir" value="${build.dir}/jar"/>
8         <property name="main-class" value="com.greatfree.testing.server.StartServer"/>
9         <property name="lib.dir" value="lib"/>
10        <property name="java.home" value="/opt/jdk1.8.0"/>
11
12        <path id="classpath">
13            <fileset dir="${java.home}" includes="**/*.jar"/>
14            <fileset dir="${lib.dir}" includes="**/*.jar"/>
15        </path>
16
17        <target name="init">
18            <!-- Create the time stamp -->
19            <tstamp/>
20            <!-- Create the build directory structure used by compile -->
21            <mkdir dir="${build.dir}"/>
22            <mkdir dir="${classes.dir}"/>
23            <mkdir dir="${jar.dir}"/>
24        </target>
25
26        <target name="compile" depends="init" description="compile the source">
27            <javac srcdir="${src.dir}" destdir="${classes.dir}" classpathref="classpath" debug="true"
28            debuglevel="source,lines,vars" includeantruntime="false"/>
29        </target>
30
31        <target name="jar" depends="compile" description="generate the distribution" >
32            <jar destfile="${jar.dir}/${ant.project.name}.jar" basedir="${classes.dir}">
33                <manifest>
34                    <attribute name="Main-Class" value="${main-class}"/>
35                </manifest>
36            </jar>
37        </target>
38
39        <target name="run" depends="jar">
40            <java fork="true" maxmemory="8192m" classname="${main-class}">
41                <classpath>
42                    <path refid="classpath"/>
43                    <path location="${jar.dir}/${ant.project.name}.jar"/>
44                </classpath>
45            </java>
46        </target>
47
48        <target name="clean" description="clean up" >
49            <delete dir="${build.dir}"/>
50        </target>
51
52        <target name="main" depends="clean,run"/>
```

```
53 </project>
```

List 2.2 The build.xml for the server

If you are not familiar with Ant, you must have no idea about the configuration file, build.xml. For details, you can visit the http://ant.apache.org to learn. To be simplified, I just give you a rough description about it. According to the explanations, you can even deal with most scenarios when debugging and testing your code.

Line 1~10 describes the configurations of your project as well as the JDK on your current machine. Line 8 is the one you should update for a particular project. You can see the value is equal to "com.greatfree.testing.server.StartServer" in that line. It defines the execution entry of the project. For any other projects, the entry must be different such that you need to update it according to every specific case. Line 10 tells Ant the location of JDK. In our case, the value is "/opt/jdk1.8.0". However, it is weird that it does not affect the execution of Ant even though you specify JDK wrongly at that line. I tested that on Ubuntu. If you specify the correct $JAVA_HOME in .profile, that is true. If not, Ant cannot execute even though you do that correctly in Line 10. It must make you confused. Anyway, as an engineering activity, it is easy to understand since it is not restricted as mathematics at all. The same phenomenon occurs on Mac OS X as well. In short, the environment variable, $JAVA_HOME, is critical for Ant to be executed correctly.

Line 12~15 specifies the class path for Ant to compile your source code. In this case, two types of libraries are included, the built-in ones from JDK and the external ones you import. They are counterparts to the JRE System Library and the Referenced Libraries, respectively, as you can see in Eclipse in Figure 2.48. If more libraries in different directories are needed, you can specify them here.

3.5 Initializing with Ant

Line 17~24 asks Ant to create a directory named build which is defined in Line 5. The compiled byte-code will be placed into the specified directory. In addition, two subdirectories are created, i.e., classes and jar. The one of classes is used to keep all of the compiled results, class files and the one of jar keeps the packaged jar file of those compiled class files. You must notice that the tag in Line 17 is target and it is named init. That means Ant can run the lines between the target independently. To do that, you can type the below command.

$ ant init

After the above command is executed, a new directory, build, is created. You can enter it and then you must find two subdirectories, classes and jar, are also there although each of them contains nothing. You can refer to Figure 2.57 for details.

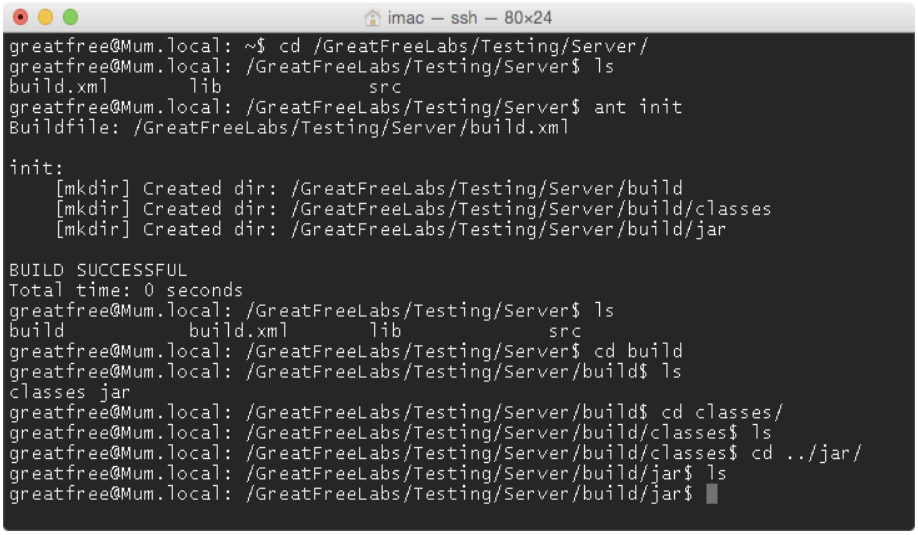


Figure 2.57 After ant init is executed, the build and its subdirectories are created

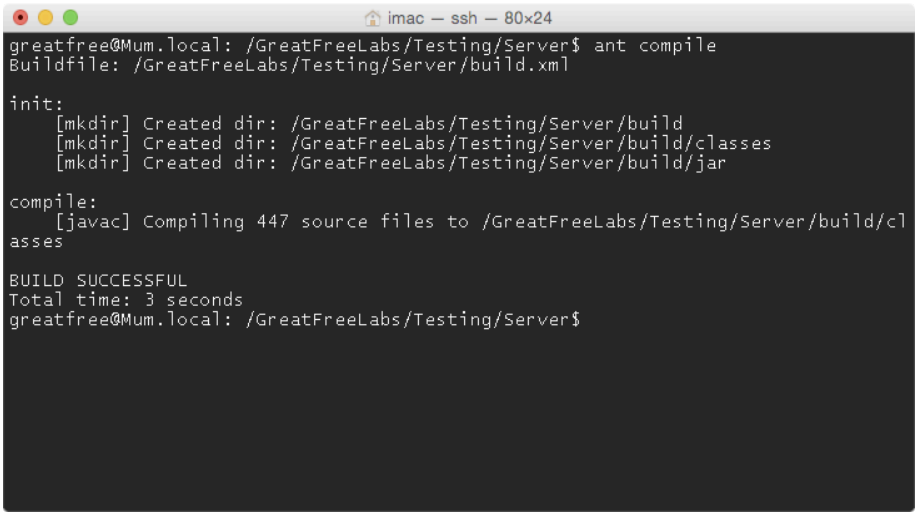


Figure 2.58 Compile Java source code with Ant

3.6 Compiling with Ant

Line 26~29 is also a target that can be executed by Ant independently. According to the target name, compile, in Line 26, you can guess that the lines aim to compile the source code. That is correct. However, before the compilation, you need to make the above directories since it depends on the target of init as it is defined by the value of depends in Line 26. You can take it try to execute by the below command.

$ ant compile

After the compilation, it must prompt that the compilation procedure is accomplished successfully, as shown in Figure 2.58. If so, you can find that the class files are located under the corresponding directories, ./build/classes, as shown in Figure 2.59.

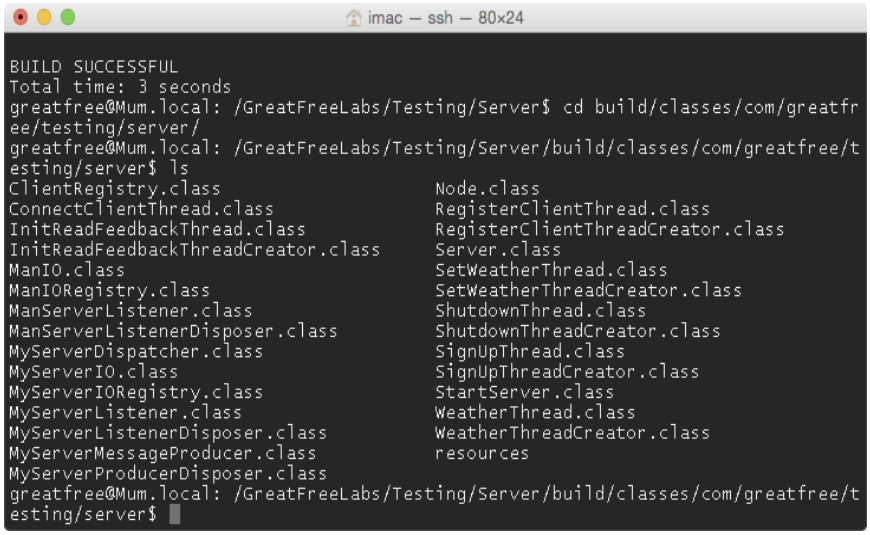


Figure 2.59 Class files in build/classes after compilation with Ant

3.7 Packaging with Ant

Line 31~37 defines the instructions to make a package for all of the results, class files, from compiled source code after compilation. The target name is called jar as shown in Line 31. To package the class files, it must depend on the completion of the compilation procedure such that the value of depends is compile. You can take a try to execute the target as follows. After that, a jar file is created based on those class files and it is persisted in the directory, ./build/jar. The name of jar file is identical to the project name defined in Line 32. You can go back to Line 1 and find that the name of the project is Clouds. Hence, you will get a jar file called, Clouds.jar, in the directory, as shown in Figure 2.60.

$ ant jar

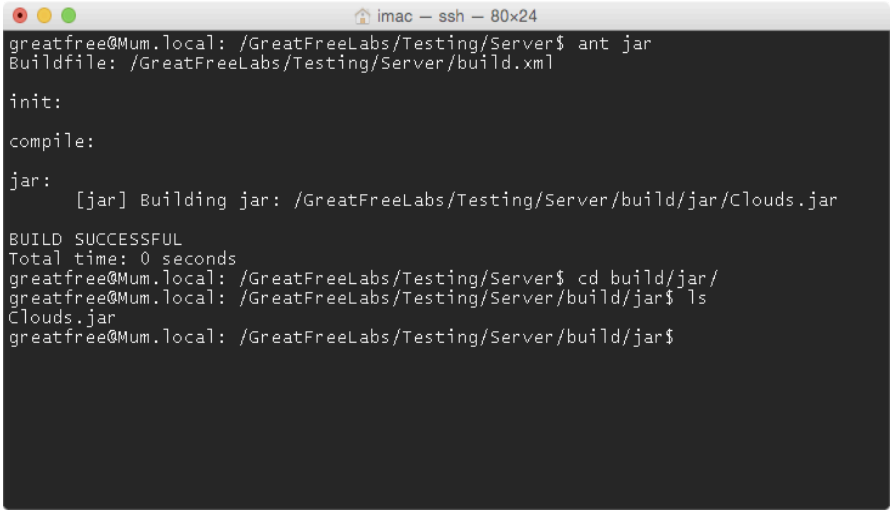


Figure 2.60 A jar is created by Ant under the directory, ./build/jar/.

3.8 Running the Server with Ant

Line 39~46 is the most important portion in the configuration file, build.xml. According to its target name, you must know it runs the source code in the project. To execute the project, the jar file must be created prior to the target such that it depends on the target of jar as defined in Line 39. Additionally, another important parameter is the value of maxmemory in Line 40, which defines the memory upper limit the project can consume. The current value is 8G (8192M), which is a big number in practice. Usually, you might not need such a high value. You can update it restrictedly according to your practical requirements.

To run the project, you just type the below command. Then, you will see Figure 2.61 which illustrates the project is running. Since it is a server application, it displays the prompts and waits for incoming messages for further interactions.

$ ant run

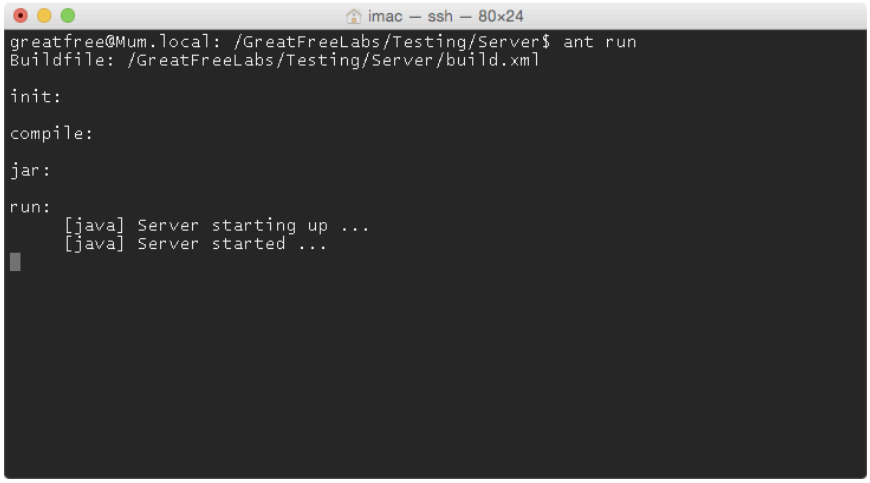


Figure 2.61 The server is running and waiting for incoming messages for further interactions

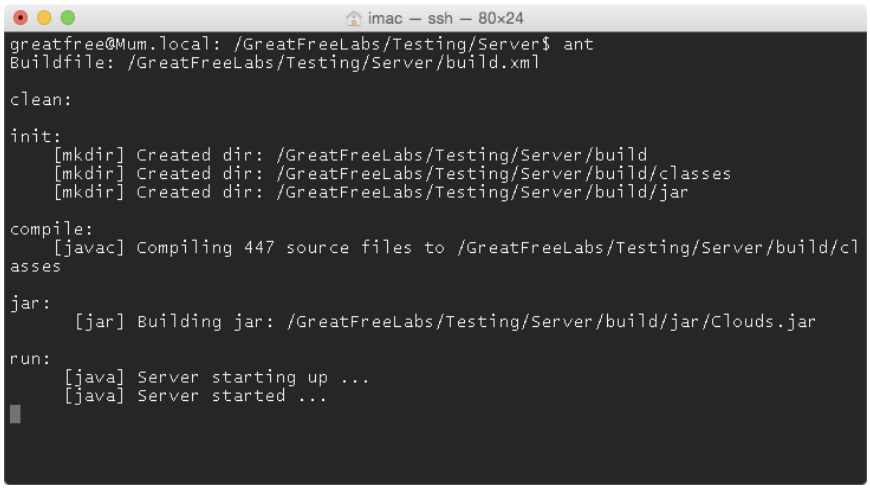


Figure 2.62 Run the project with the command, ant, only

In fact, you can do the above tasks in one step just by typing the command, ant, only as follows, like Figure 2.62. Then, you will get the same result as what you see in Figure 2.61.

$ ant

By the way, at this moment, you can shutdown the running server by typing Ctrl+C since it is just a sample code. In practice, it is not a good habit since it might result in loss of data and the failure of the system. A qualified system always provides users with a graceful way to quit.

3.9 Cleaning Up with Ant

Usually, a project needs to be rebuilt and updated all the time. Therefore, it is required for a project management tool to provide the way to clean up the out-of-date built results. That is what is defined in Line 48-50. To do that, you can type the below command. After the command is executed, you can see that the directory of build is removed, as shown in Figure 2.63. That means you can rebuild the entire project based on the updated source code.

$ ant clean

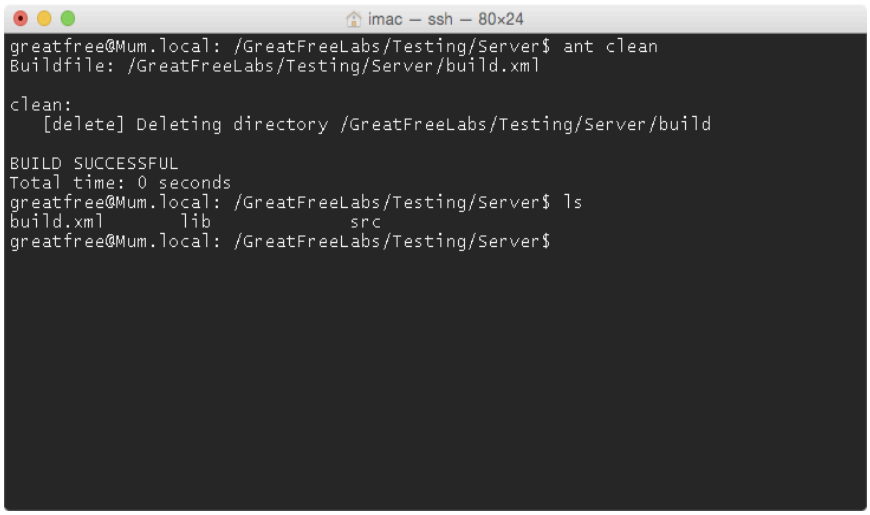


Figure 2.63 Cleaning up the out-of-date built results with ant

3.10 The Main Task of Ant

The last target, called main, in the build.xml is defined in Line 52. It actually does nothing but executes the target clean first and then invokes the one of run. It is exactly the same functions as those of rebuilding and running in Eclipse. If you run it as follows in Figure 2.64, the result is equivalent to that resulted from the command, ant, only, as shown in Figure 2.62. In practice, the most-often typed command is ant only. In fact, it runs the target of main.

$ ant main

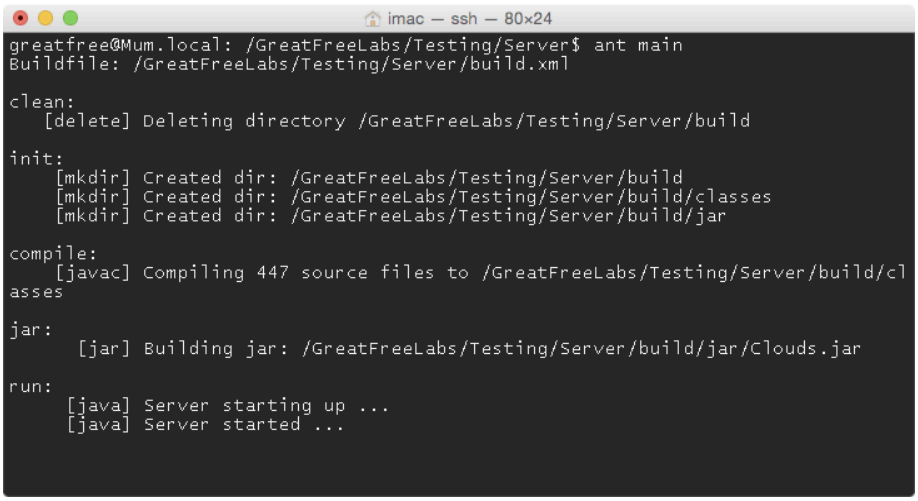


Figure 2.64 The target of main is equivalent to the command, ant, only

3.11 Setting Up a Client Node

If the server node runs successfully, you need to set up another node, the client one, to test whether your environment is really set up. To do that, you can almost do the same thing from Section 3.1 to Section 3.4 on another PC, which is called freescape in my environment. One tiny difference is that you need to create a directory, Client, under the directory of Testing. It makes sense since we need to set up the client. After that, you can have an environment like the one shown in Figure 2.65.

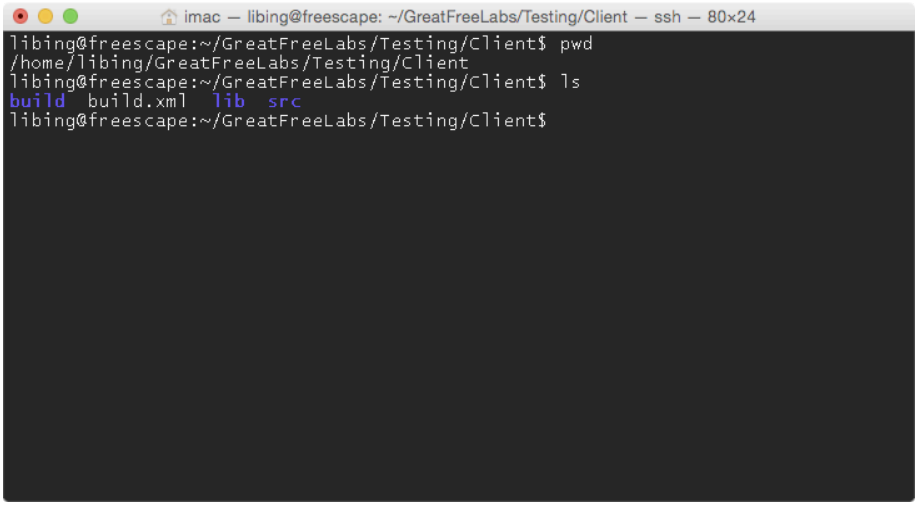


Figure 2.65 The client for testing is set up

Before you can invoke the ant, you need to do some important updates. First, the server IP address should be updated at your client side. As discussed previously, for example, the address in my environment is 192.168.1.105. Thus, you need to update the code, Line 12 of ServerConfig.java, as shown in List 2.1. The code can be found in the directory of ./src/com/greatfree/testing/data/ServerConfig.java. You can also do that with the built-in text editor, vi, or any other ones. In my environment, since the client is located on a Ubuntu PC, freescape, I need to update the line with vi, as

shown in Figure 2.66. Again, remember you do not need to do that with Eclipse since the update is also small.

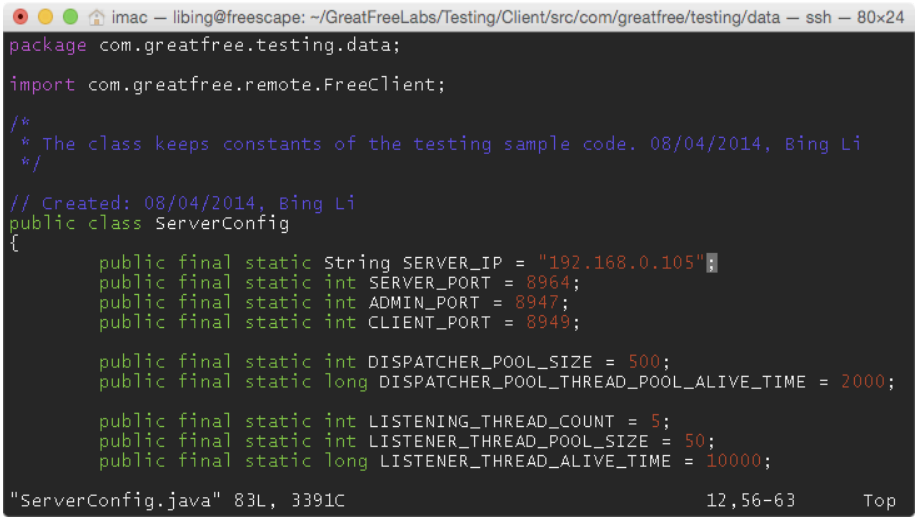


Figure 2.66 Update Line 12 using vi on Ubuntu

Another update before the testing can be proceeded is to ensure the build.xml for the client is correctly written. For the client, Line 8 of the build.xml should be exactly as the one shown as below.

<property name="main-class" value="com.greatfree.testing.client.StartClient"/>

Compared with the one in List 2.2, the StartServer is updated to StartClient, which is the unique entry for executing the client. You can also update the value by vi or any other text editors, as shown in Figure 2.67.

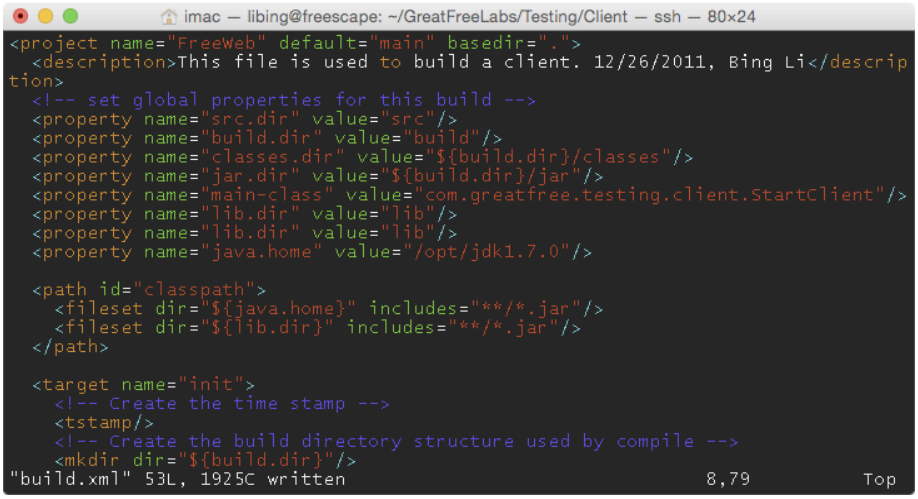


Figure 2.67 Update the unique entry, Line 8, for executing the client

3.12 Running the Client

After the above tasks are finished, you are ready to test your debugging and testing environment. Before running your client, make sure the server is started on the server

node as shown in Figure 2.61. Thereafter, you can run Ant to execute the client by typing the below command, shown in Figure 2.68.

$ ant

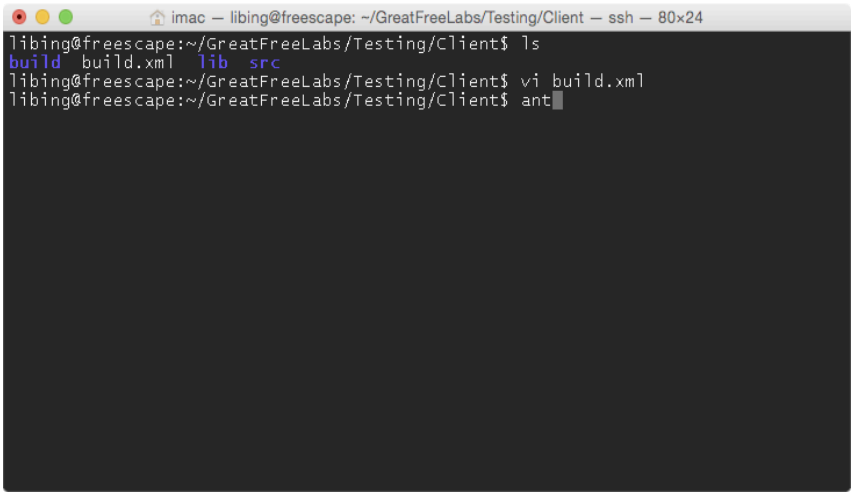


Figure 2.68 Run the client with Ant

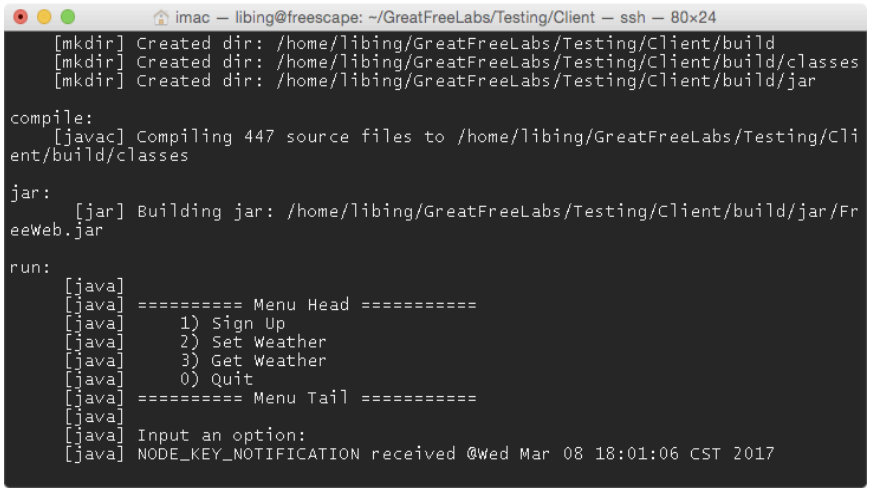


Figure 2.69 The client runs at the moment when it is started up

Once if the client runs, a simple menu is displayed in the terminal, as shown in Figure 2.69. At the same time, the server side gets updated as well, as shown in Figure 2.70. You do not need to care about what the prompt message means at this moment. I will explain that later.

Before moving forward, I just give a brief introduction to the application with which you are testing the environment. Actually, the application shows you how to transmit information to the server and then get it back and show at the client side. In this case, the client sends the weather information to the server by the second menu option, i.e., “Set Weather”. Then, it gets the data back by the third option, “Get Weather”. Besides those two options, it has the first one which simulates how to sign up the server, i.e., “Sign Up”, and the last one to quit the system, i.e., “Quit”.

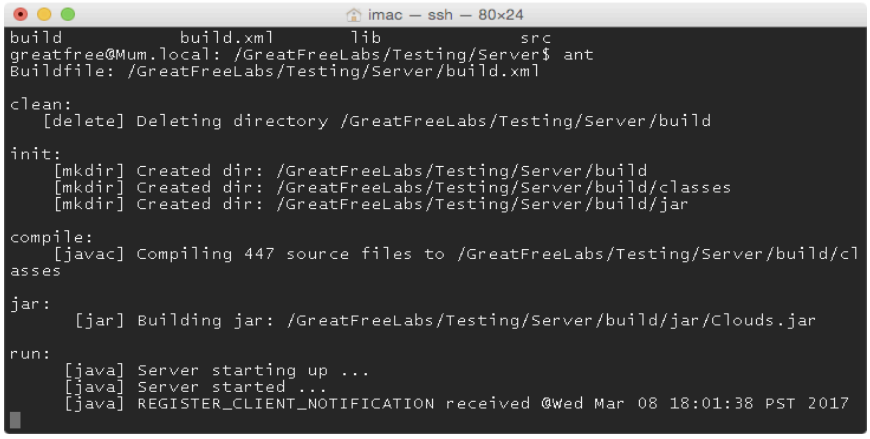


Figure 2.70 The server gets updated when the client just starts

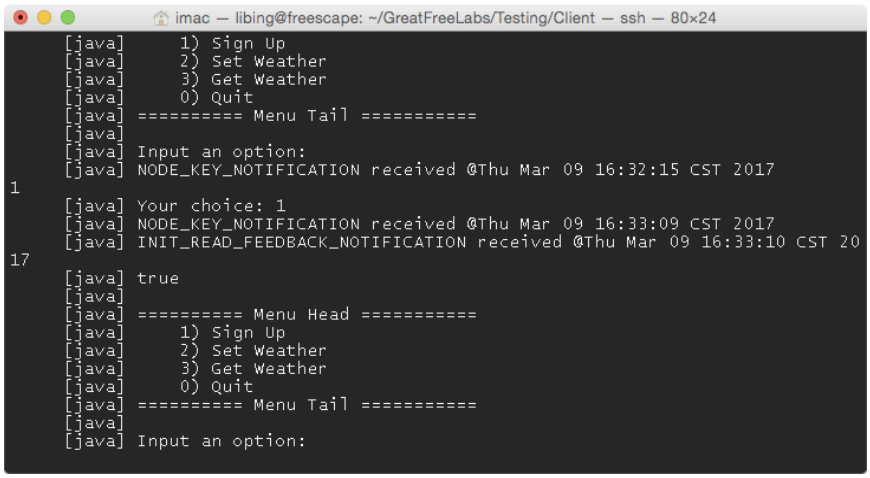


Figure 2.71 The terminal at the client side after the first option, Sign Up, is selected

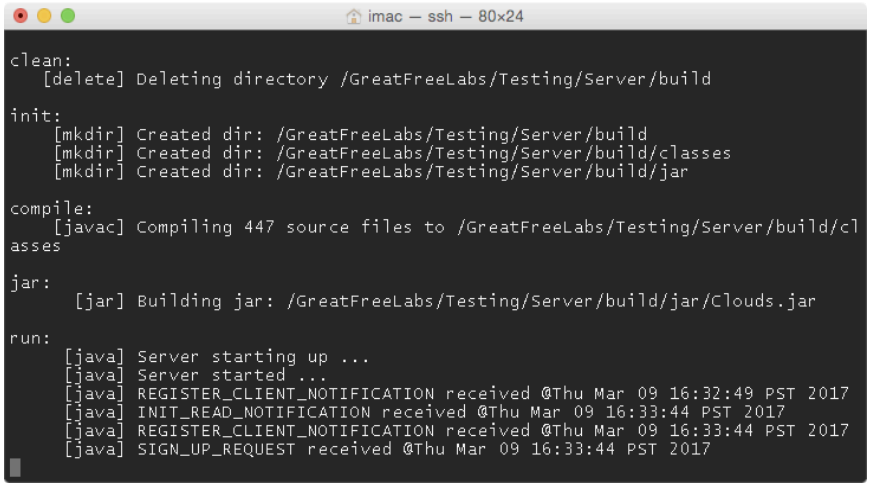


Figure 2.72 The terminal at the server side after the first option, Sign Up, is selected

You can then follow the order of the menu to keep running the client. First, you select the first option, “Sign Up”, of the menu at the client side. After that, the client side and the server one change as shown in Figure 2.71 and Figure 2.72. At this moment, you do need to care about the prompts. What you need to do is to keep the sample

running fine. One interesting thing is that “SIGN_UP_REQUEST received …” is shown at the server side. According to that, you can guess that the server receives the message from the client.

You can move forward continually by selecting the second option, Set Weather, from the client side. That is, it sends the weather information to the server. You can see one prompt at the server side, “SET_WEATHR_NOTIFICATION”. That means the weather information is received by the server. Your current terminals should be like Figure 2.73 and Figure 2.74.

The third option of the menu is Get Weather, which tries to get the weather information you just sent to the server back to the client. After you select it, you can see that data is displayed at the client side as follows. On the server side, it prompts that “WEATHER_REQUEST” is received. Both of the terminals should be like Figure 2.75 and Figure 2.76.

……
Temperature: 20.4
Forcast: Sunshine
Rain: false
How much rain: 10.0
……

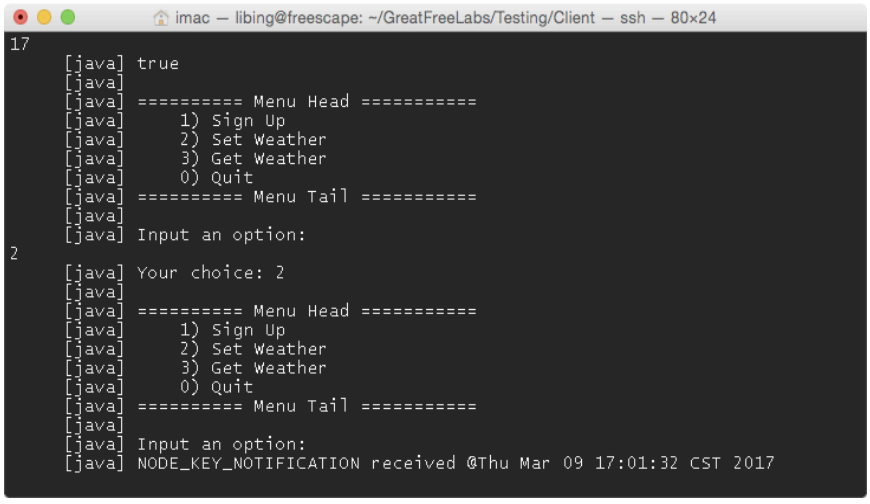


Figure 2.73 The terminal at the client side after the second option, Set Weather, is selected

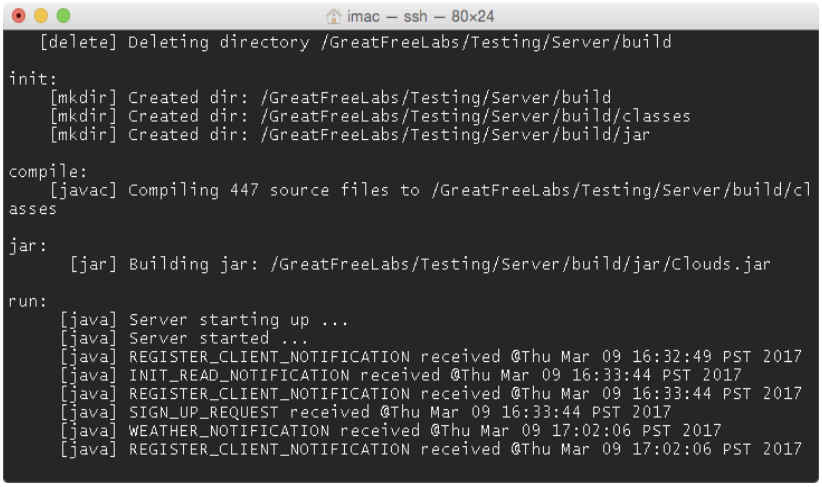


Figure 2.74 The terminal at the server side after the second option, Set Weather, is selected

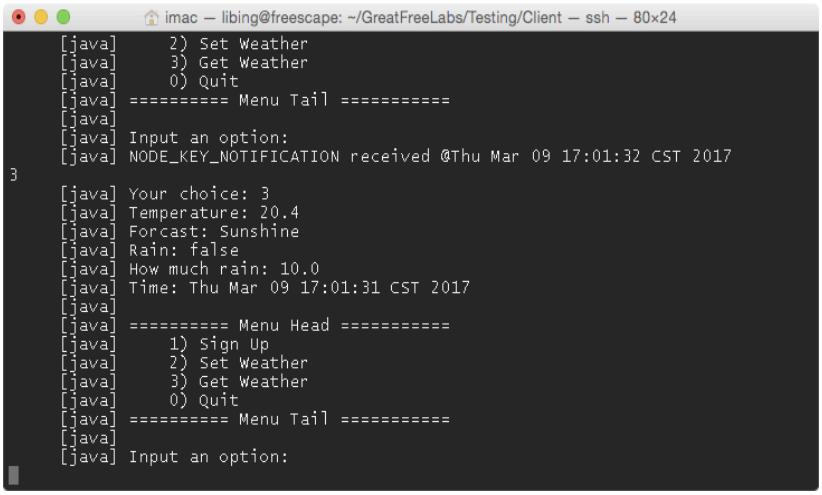


Figure 2.75 The terminal at the client side after the third option, Get Weather, is selected

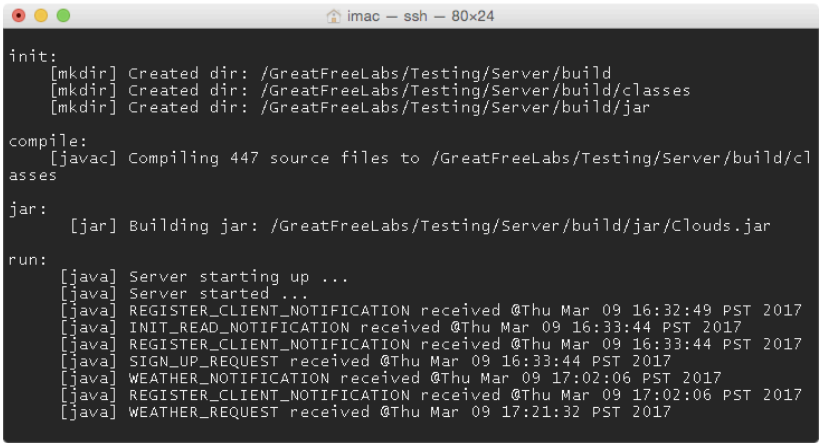


Figure 2.76 The terminal at the server side after the third option, Get Weather, is selected

The last option of the menu at the client side forces the client application to quit, as shown in Figure 2.77. The server side can quit only by typing Ctrl+C (Figure 2.78). After the last option is selecte, the testing is accomplished. If there are no any exceptions during the procedure, that means your development is set up successfully. You can more further to what you expect, how to program large-scale distributed systems with GreatFree!

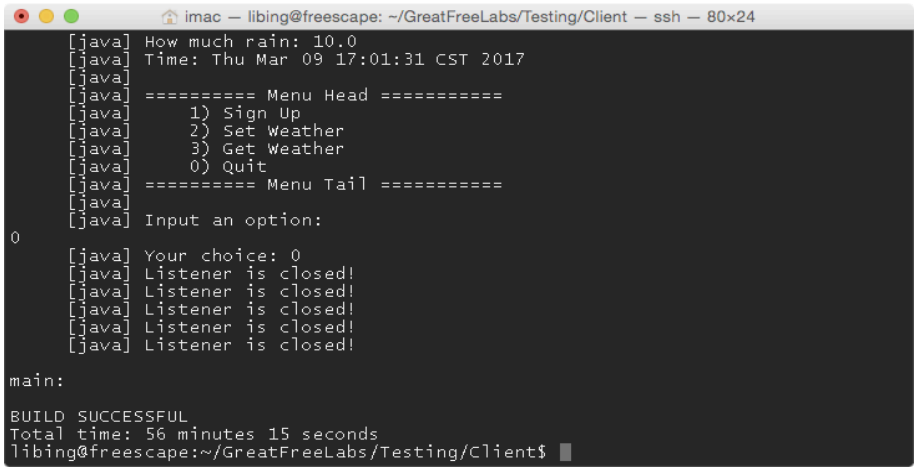


Figure 2.77 The client quits

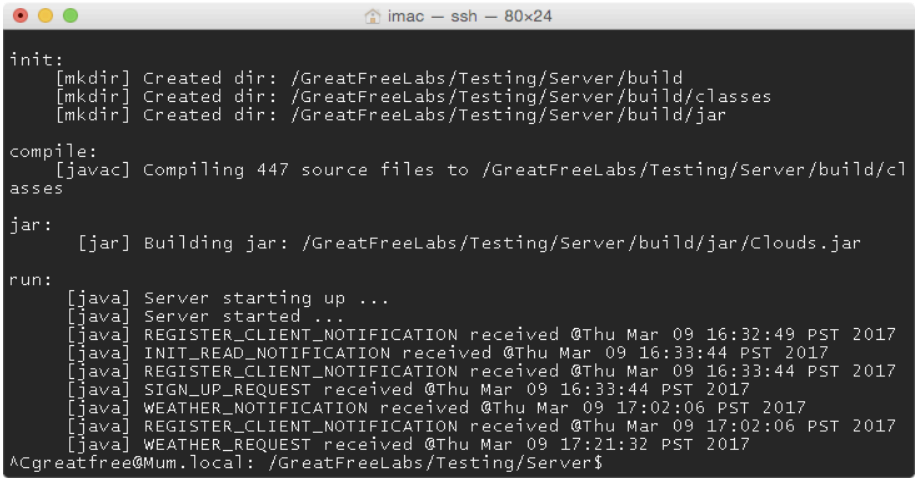


Figure 2.78 The server quits

4. The Last Setting Up

Now all of your environments are set and you are ready to move forward to learn programming with GreatFree. However, I still need to mention the last setting up, i.e., the word-wrap. It is weird that the function is not included in Eclipse. Thus, if you open one code, for example, com.greatfree.testing.server.MyServerDispatcher, it must look like the one shown in Figure 2.79.

```
36
37  // Created: 09/20/2014, Bing Li
38  public class MyServerDispatcher extends ServerMessageDispatcher<ServerMessage>
39  {
40      // Declare a notification dispatcher to process the registration notification concurrently. 11/04/2014, Bir
41      private NotificationDispatcher<RegisterClientNotification, RegisterClientThread, RegisterClientThread
42      // Declare a request dispatcher to respond users sign-up requests concurrently. 11/04/2014, Bing Li
43      private RequestDispatcher<SignUpRequest, SignUpStream, SignUpResponse, SignUpThread, SignU
44      // Declare a notification dispatcher to set the value of Weather when an instance of WeatherNotificatio
45      private NotificationDispatcher<WeatherNotification, SetWeatherThread, SetWeatherThreadCreator> s
46      private NotificationDispatcher<TestNotification, TestNotificationThread, TestNotificationThreadCreato
47      // Declare a request dispatcher to respond an instance of WeatherResponse to the relevant remote cli
48      private RequestDispatcher<WeatherRequest, WeatherStream, WeatherResponse, WeatherThread, W
49      // Declare a notification dispatcher to deal with instances of InitReadNotification from a client concurre
50      private NotificationDispatcher<InitReadNotification, InitReadFeedbackThread, InitReadFeedbackThre
51      // Declare a notification dispatcher to shutdown the server when such a notification is received. 02/15/2
52      private NotificationDispatcher<ShutdownServerNotification, ShutdownThread, ShutdownThreadCreat
53
54      /*
55       * Initialize. 09/20/2014, Bing Li
56       */
57      public MyServerDispatcher(int corePoolSize, long keepAliveTime)
58      {
59          // Set the pool size and threads' alive time. 11/04/2014, Bing Li
60          super(corePoolSize, keepAliveTime);
61
62          // Initialize the client registration notification dispatcher. 11/30/2014, Bing Li
63          this.registerClientNotificationDispatcher = new NotificationDispatcher<RegisterClientNotification, Re
64
65          // Initialize the sign up request dispatcher. 11/04/2014, Bing Li
66          this.signUpRequestDispatcher = new RequestDispatcher<SignUpRequest, SignUpStream, SignUpl
67
```

Figure 2.79 The code of MyServerDispatcher is presented without word-wrap

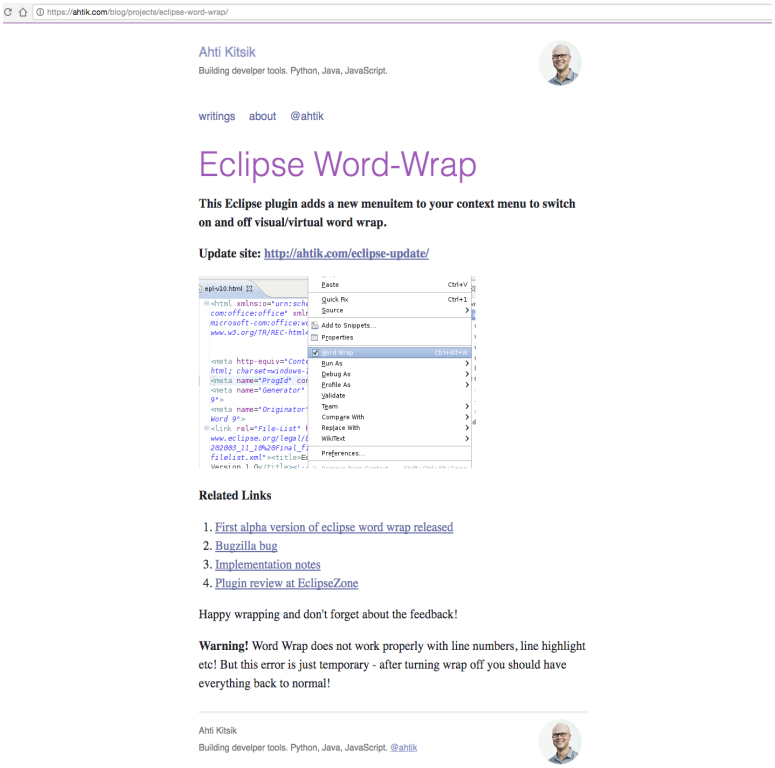


Figure 2.80 The Web page of the word-wrap plug-in developed Ahti Kitsik

Because the lengths of lines, such as Line 41~52 and Line 63~66, are longer than the width of the editor, the code cannot be displayed entirely in it. Programmers have to extend the editor, move the cursor or drag the bottom scroll-bar to see the hidden lines. That is not a convenient experience.

I guess Eclipse expects developers write short lines rather than long ones. That is the reason the function of word-wrap is not included by default in Eclipse. Unfortunately, I like writing long lines with long naming classes, variables and even methods. It helps me recall what I did a long time ago. If you have the similar preferences as me, you can install the word-wrap plug-ins. The one I choose is developed by Ahti Kitsik [36]. You can install it following the link [37], as shown in Figure 2.80.

To install the plug-in, you need to open the menu of Help and then select the option, "Install New Software...", as shown in Figure 2.81. After that, a new dialog, Install, is popped up, as shown in Figure 2.82.

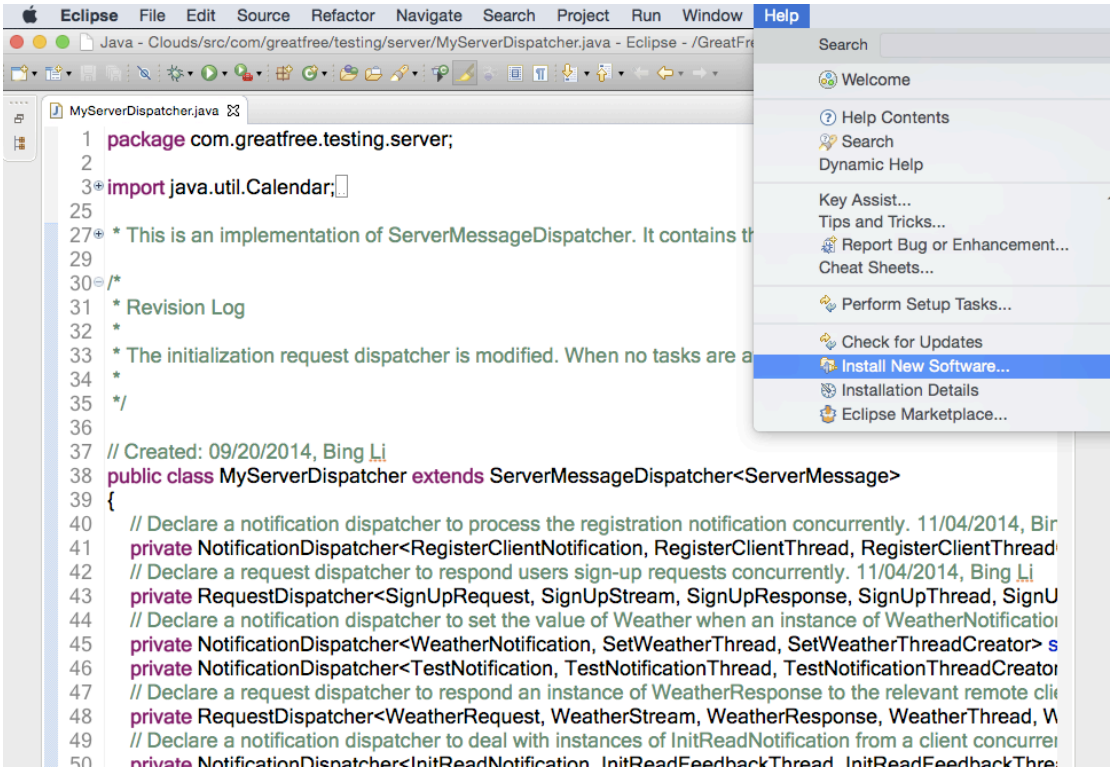


Figure 2.81 Open the menu of Help and select the option, "Install New Software…"

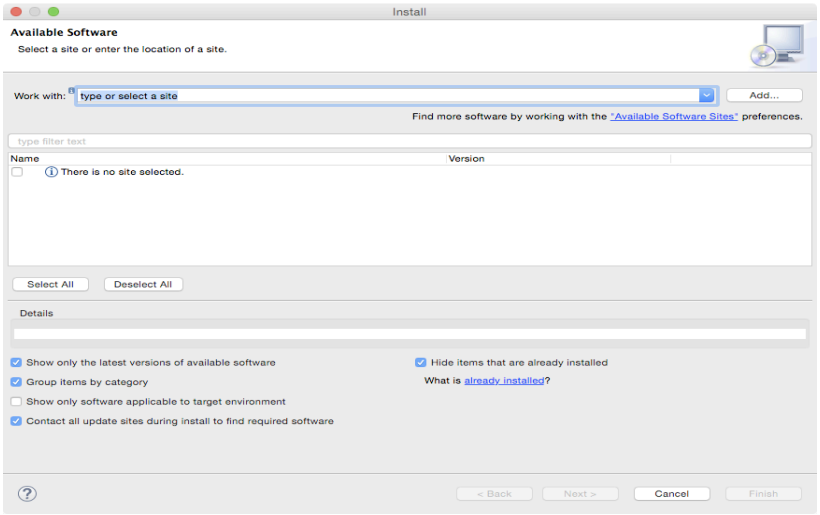


Figure 2.82 The Install dialog for new plug-ins

You need to copy-paste the below link from the Web page to the text field of "Work with" in the Install dialog and type the left-side button, Add. You are required to name the plug-in of word-wrap. For example, you can name it AhtikWordWrap. Then, click the button of OK. That's it.

http://ahtik.com/eclipse-update/

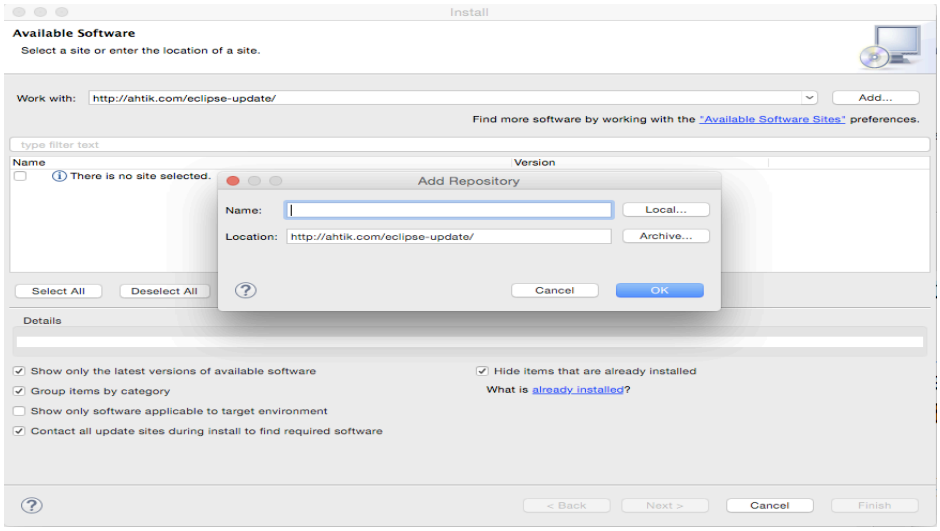


Figure 2.83 Copy-Paste the link of word-wrap and then click the left-side button, Add

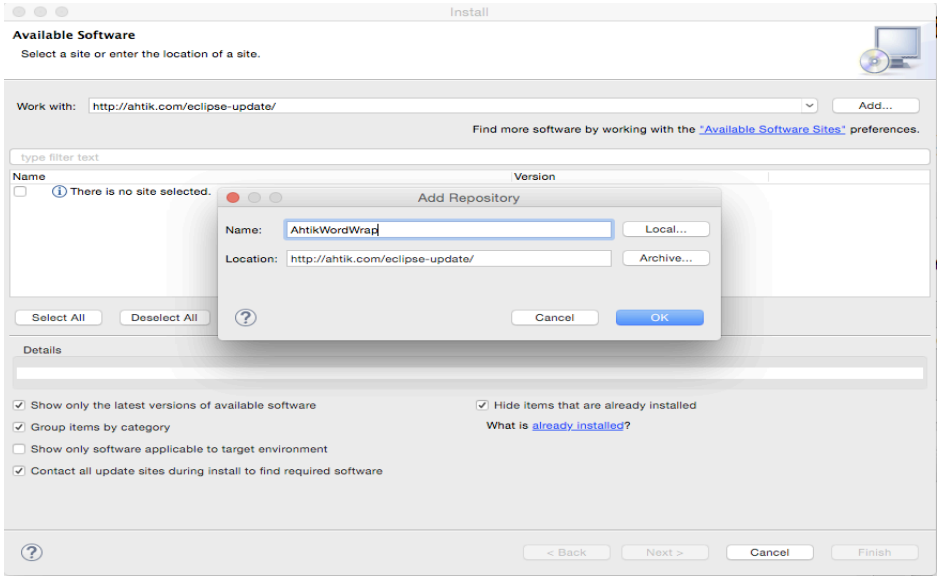


Figure 2.84 Name the plug-in of word-wrap to be installed

After the installation, a new option is shown in the popped menu of the editor when right-clicking, as shown in Figure 2.85. If you select it, the code of MyServerDispatcher should be word-wrapped like the one shown in Figure 2.86.

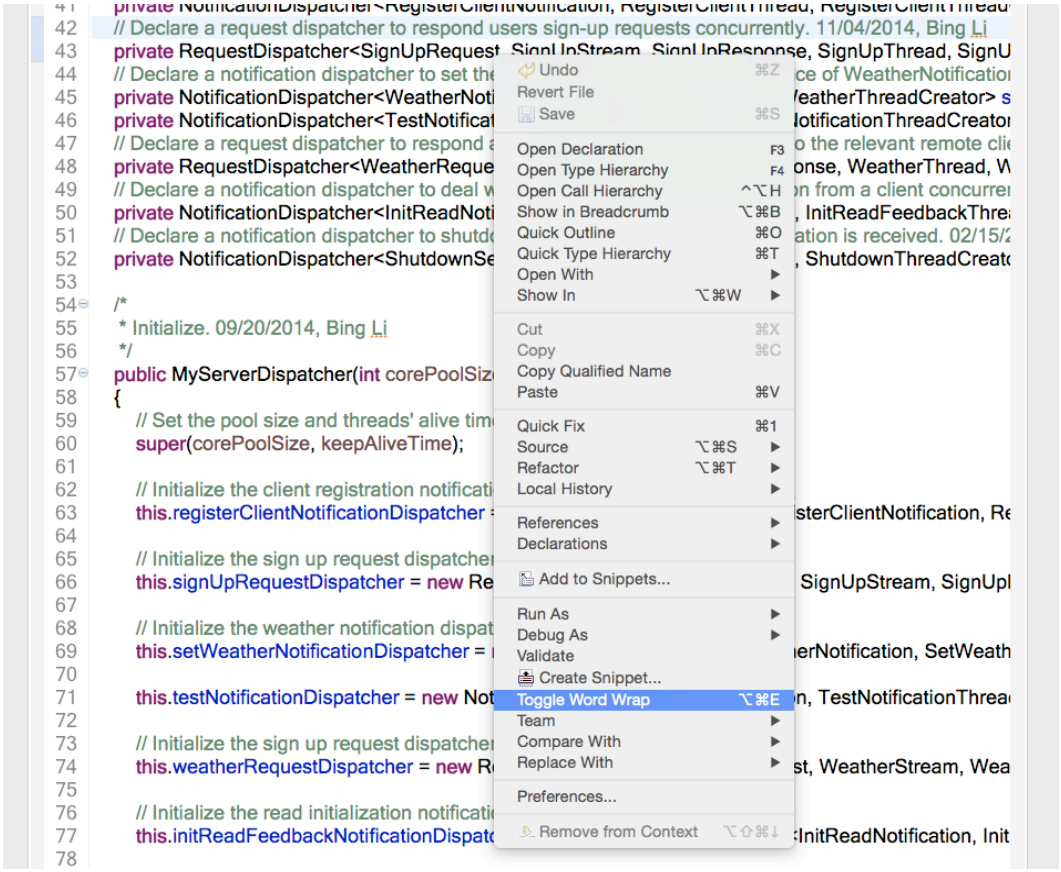


Figure 2.85 The editor popped menu has a new option for word-wrap

Figure 2.86 The code of MyServerDispatcher is word-wrapped

References

[1] Michael Kerrisk. The Linux Programming Interface: A Linux and Unix System Programming Handbook. No Starch Press, 2010, ISBN-13: 978-1-59327-220-2, ISBN-10: 1-59327-220-0.

[2] Neil Matthew, Richar Stones. Beginning Linux Programming, 4th Edition. Wiley Publishing, ISBN: 978-470-14762-7.

[3] Mark Mitchell, Jeffrey Oldham, Alex Samuel. Advanced Linux Programming. New Riders Publishing, 2001, ISBN: 0-7357-1043-0.

[4] Windows OS. DOI: http://www.microsoft.com/en-us/windows.

[5] Solaris OS. DOI: http://www.oracle.com/solaris.

[6] Mac OS X. DOI: http://www.apple.com/macos.

[7] Linux OS. DOI: http://www.linux.org.

[8] Ubuntu OS. DOI: http://www.ubuntu.com.

[9] iOS. DOI: http://www.apple.com/ios.

[10] Android. DOI: http://www.android.com.

[11] Eclipse IDE. DOI: http://eclipse.org/ide.

[12] Google. DOI: http://www.google.com.

[13] Apple. DOI: http://www.apple.com.

[14] iMac. DOI: http://www.apple.com/imac.

[15] Mac Book Pro. DOI: http://www.apple.com/macbook-pro.

[16] Redhat Linus. DOI: http://www.redhat.org.

[17] Debian OS. DOI: http://www.debian.org.

[18] CentOS. DOI: http://www.centos.org.

[19] Evi Nemeth, Garth Snyder, Trent R. Hein, Ben Whaley. UNIX and Linux System Administration Handbook. Prentice Hall, 2011, ISBN-13: 978-0-13-148005-6, ISBN-10: 0-13-148005-7.

[20] Ubuntu SSH. DOI: https://help.ubuntu.com/lts/serverguide/openssh-server.html.

[21] Ubuntu FTP. DOI: https://help.ubuntu.com/lts/serverguide/ftp-server.html.

[22] Java. DOI: http://java.com.

[23] Oracle Java SE. DOI: http://www.oracle.com/technetwork/java/javase.

[24] Apache Ant. DOI: http://ant.apache.org.

[25] Eclipse. DOI: https://www.eclipse.org/downloads.

[26] NetBeans. DOI: http://netbeans.org.

[27] IntelliJ IDEA. DOI: https://www.jetbrains.com/idea.

[28] Gnu Make. Robert Mecklenburg. O'Reilly, 2005, ISBN: 978-0-598-00610-5.

[29] David Haskins. C Programming In Linux. Ventus Publishing, ISBN: 978-7681-472-4.

[30] vi Editor. DOI: https://www.cs.colostate.edu/helpdocs/vi.html.

[31] Heling Wu. ACM Turing Award Winners. University Education Press of China, 2002, ISBN: 978-7040-321-96.

[32] Walter Brainerd. Guide to Fortran 2003 Programming. Springer, 2009, ISBN: 978-1-84882-542-0.

[33] Bing Li. 2015. DOI: https://github.com/greatfree/Programming-Clouds.

[34] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. 2011. Processes, Chapter 3, Distributed Systems: Concepts and Design, the Fifth Edition.

[35] Elliotte Rusty Harold. 2014. Java Network Programming. O'Reilly Media, ISBN: 978-1-449-35767-2.

[36] Ahti Kitsik. DOI: https://ahtik.com/blog/projects/eclipse-word-wrap/.

[37] Ahti Kitsik Eclipse Word Wrap. DOI: http://ahtik.com/eclipse-update/.

Chapter 3 Programming Notifications

Abstract

Programming is a tough job, especially when you program a distributed system. If you want to do that over the Internet, it is really a challenge. We have already discussed about the issue in Chapter 1. So, it is not necessary to repeat that here. What we plan to talk is to teach you how to implement a notification in the complicated computing environment. In a distributed environment, a notification is defined as a message that is sent from a distributed node to a remote one without expecting any feedbacks to the sender. Notifications are common to see in such an environment since that is an indispensable manner for distributed computing devices to interact with each other so as to form a computing system.

In this chapter, you will learn how to program notifications with GreatFree. With the support of GreatFree APIs and design patterns, the procedure becomes convenient even though you know nothing about distributed systems. However, it is not the goal of GreatFree to provide developers with a development environment in which underlying technologies are transparent. In contrast, programmers work in a context with matures lines of code. It is not recommended for them to ignore what the code represents. Rather, they should be aware of the sense of the code since the important goals of GreatFree aim to program not only fast but also visibly since it believes that programming in a transparent environment lowers developers' skills. Knowing distributed code and technologies does not slow down the efficiency of programming with the support of GreatFree's APIs and patterns. Moreover, the visibility of the code notifies developers that they can improve the code if they find a better solution.

You can taste the new style of programming with GreatFree in the chapter starting from notifications. To ease the procedure, it does not explain all of the code you will touch. Instead, you just program the code that can implement your requirements immediately. Of course, you can do that if you do not care deeply about distributed programming. But I highly suggest you to read the book further to dominate the whole environment completely. Then, you will find yourself on the way to become a successful programmer.

1. The Sample Code

You must still remember the client-server application to send and request weather information we run in Chapter 2. However, until now you have not taken a look at the code although you run it successfully in a distributed environment. I need to indicate that the code of that application is established on GreatFree. Therefore, it is a high quality and practical system rather than just a sample code to send one piece of information through TCP/IP [1]. To implement such an application, you must have much richer knowledge about distributed systems and higher programming skills than a regular communication program as discussed in Chapter 1. In this chapter, I have no time to explain that in details since I will show you how to start to program instead of

tell you a lot of theories. The only point I need to indicate is that you must be aware of the fact that you are programming a practical system conveniently because of GreatFree.

Since we will program, we have to return the IDE (Integrated Development Environment), Eclipse. If you select others, that is fine since the UI (User Interface) for the code presentation is similar. By the way, as I said before, I program on Mac OS X. Thus, you will notice that the screenshots shown in the chapter are all in the style of Apple. On other operating systems, the UI of Eclipse is almost identical.

When Eclipse is opened with the installed GreatFree libraries and source code, the interface should be exactly like the one shown in Figure 3.1. You must feel astray at this moment. Do not worry. You will feel comfortable with the code soon. Before exploring the code, I highly recommend you to format your code with the Formatter of Eclipse, as shown in Figure 3.2. From my point of view, it is really important to adjust the format according to your preferences when coding.

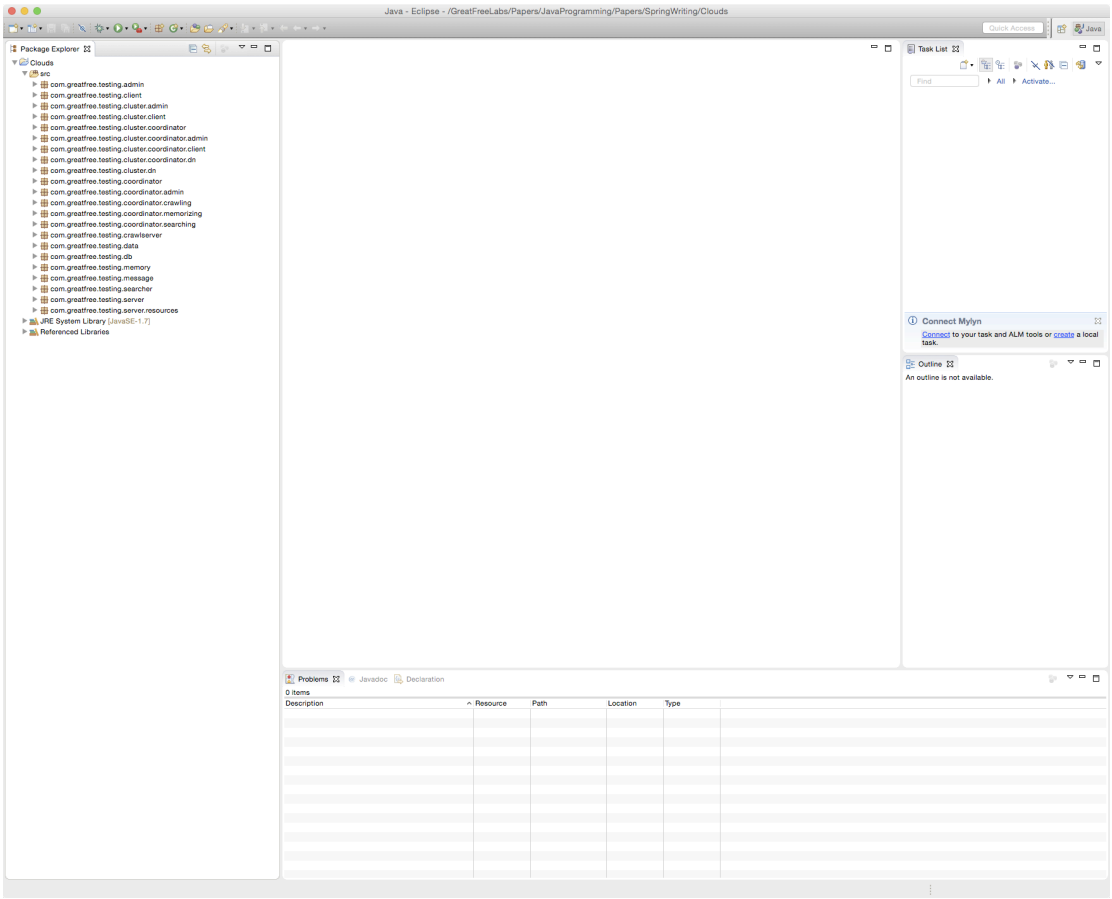


Figure 3.1 Eclipse is opened

2. The Message of Notifications

To program a distributed system, it is necessary to design messages at first since distributed nodes are interconnected with those messages. First, we will implement a very frequently-used type of messages, notifications. According to its name, you

know this type of messages are used as the information that is sent from a source node to a destination without requiring any feedbacks from the destination to the source. To be simplified, such a message seems to be a one-way travel rather than a round-trip one.

The goal of this exercise is to create a new notification such that it can be sent from a client to a server. Once if it is received, the information it contains should be displayed on the server.

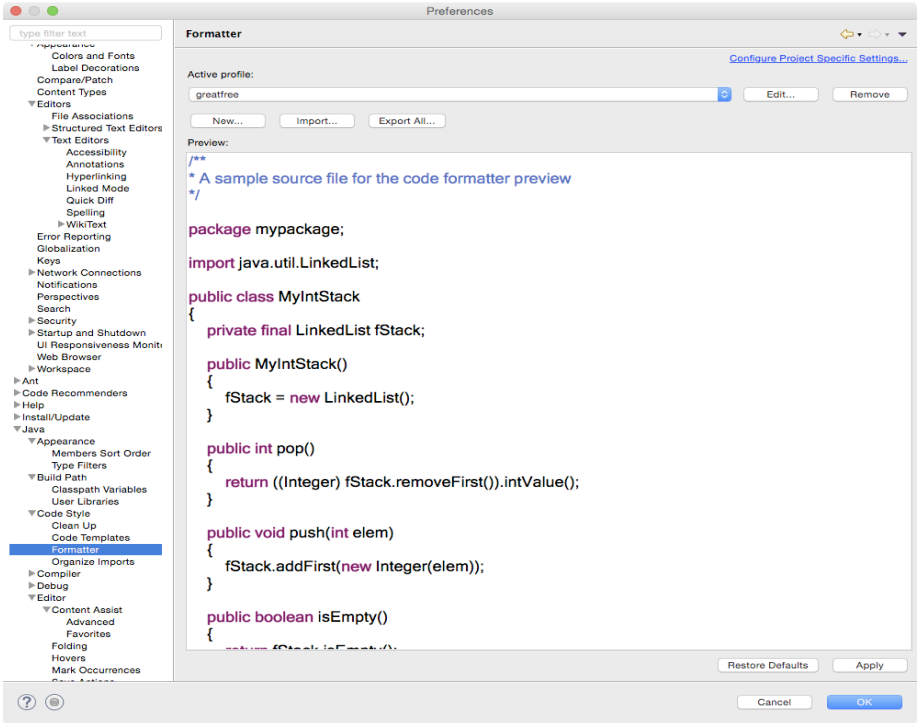


Figure 3.2 Adjust the format of your code

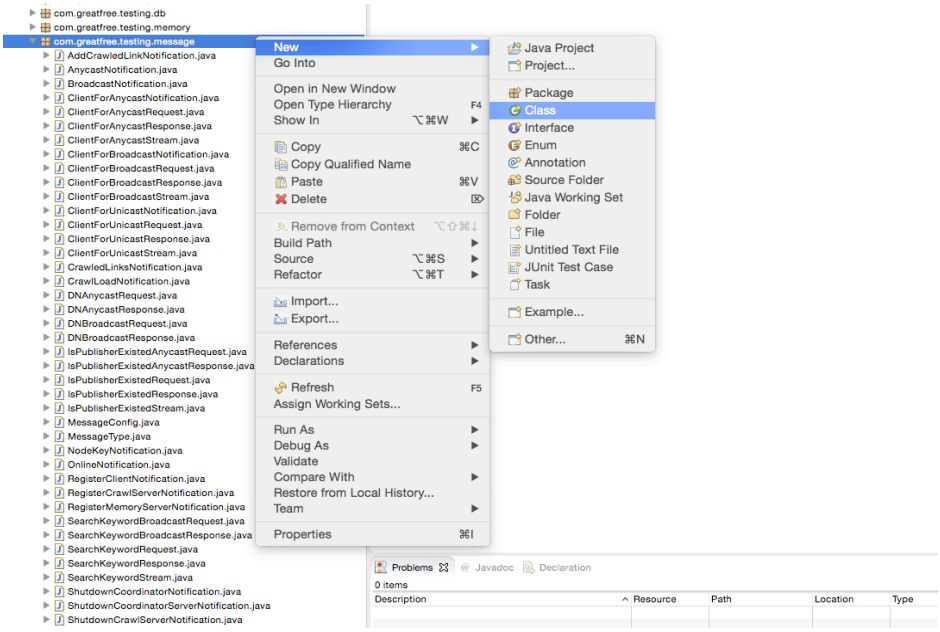


Figure 3.3 Create a notification

2.1 Creating a Notification

First, let us click the Java package, com.greatfree.testing.message, in the perspective of Package Explorer in Eclipse. The package contains all of the messages employed in the project of Clouds you created in Chapter 2. Do not get scared for them although you might be a little bit. Just follow my steps and then you will be able to dominate all of them.

To create a notification, right-click on the package and then select the option of New. When that option is selected, select the option of Class further, as shown in Figure 3.3.

After the option of Class is clicked, a dialog is popped up. Now we are doing an exercise such that we can create an arbitrary message and we do not need to care about whether it makes sense in the practical application. For that, you can just name the notification as TestNotification as shown in Figure 3.4. You do not need to change any other options in the dialog. After the name is typed, just click the button of Finish.

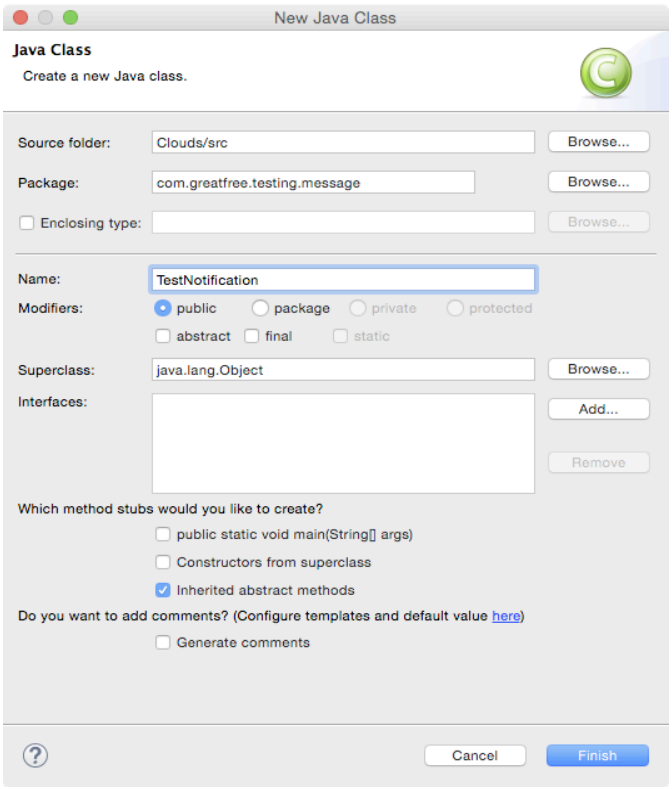


Figure 3.4 Name the notification, TestNotification

Once if the button of Finish is clicked, a new tab window for TestNotification.java is opened as an editor in Eclipse, as shown in Figure 3.5. Actually, you can type your code inside the editor. We will start to program from now on.

2.2 Fixing Errors and Warnings for the Notification After Extending

Before you start to write something into TestNotification.java, you are highly suggested to open a sample code to mimic. If you are an experienced developer, you must learn that each piece of code has a sample to imitate. It is really impossible to type something arbitrarily since we are programming code not writing novels. You must remember we have tried one notification which contains the information about the weather in Chapter 2. Therefore, you can follow that message to write your own notification. The code for the weather notification is WeatherNotification.java as shown in List 3.1. You can find it in the above package, com.greatfree.testing.message.

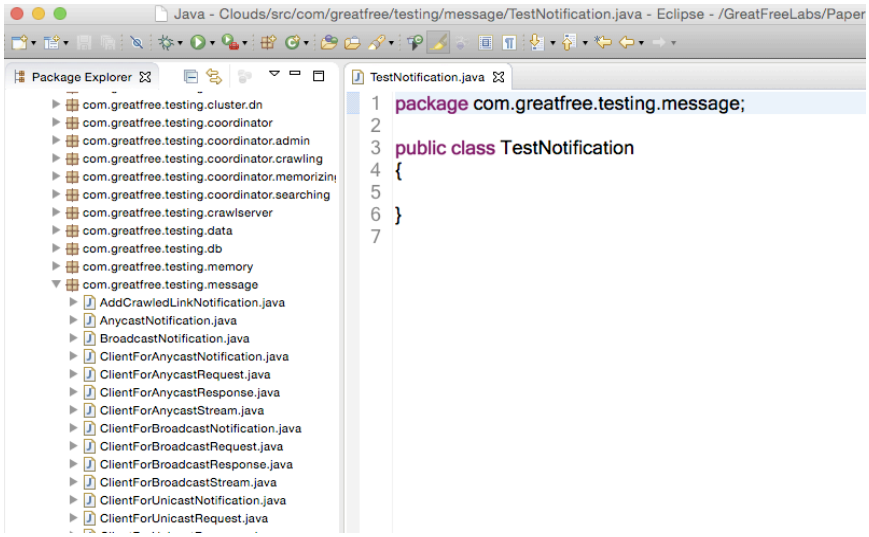


Figure 3.5 TestNotification.java is opened in the editor

```
1   package com.greatfree.testing.message;
2
3   import com.greatfree.multicast.ServerMessage;
4   import com.greatfree.testing.data.Weather;
5
6   /*
7   * The notification contains the data of weather. It is sent to a server such that the data
8   * on the server can be updated. 02/06/2016, Bing Li
9   */
10
11  // Created: 02/06/2016, Bing Li
12  public class WeatherNotification extends ServerMessage
13  {
14      private static final long serialVersionUID = 3555195575233260451L;
15
16      private Weather weather;
17
18      public WeatherNotification(Weather weather)
19      {
20          super(MessageType.WEATHER_NOTIFICATION);
21          this.weather = weather;
22      }
23
24      public Weather getWeather()
25      {
26          return this.weather;
27      }
28  }
```

List 3.1 The code of WeatherNotification.java

According to the code of WeatherNotification.java, you must notice it has a parent class, ServerMessage. So you are required to let your notification, TestNotification.java, to derive it as well. I will explain the class of ServerMessage in later chapters. Now you just extends TestNotification with it.

If you are new to use Eclipse, you might feel weird since it gets errors when TestNotification extends ServerMessage, as shown in Figure 3.6.

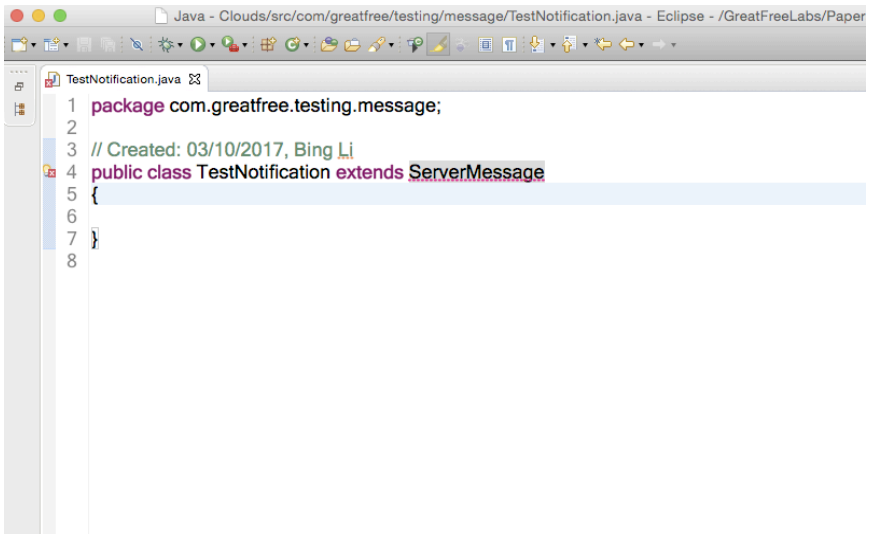


Figure 3.6 Immediately after TestNotification extends ServerMessage, it gets compilation errors

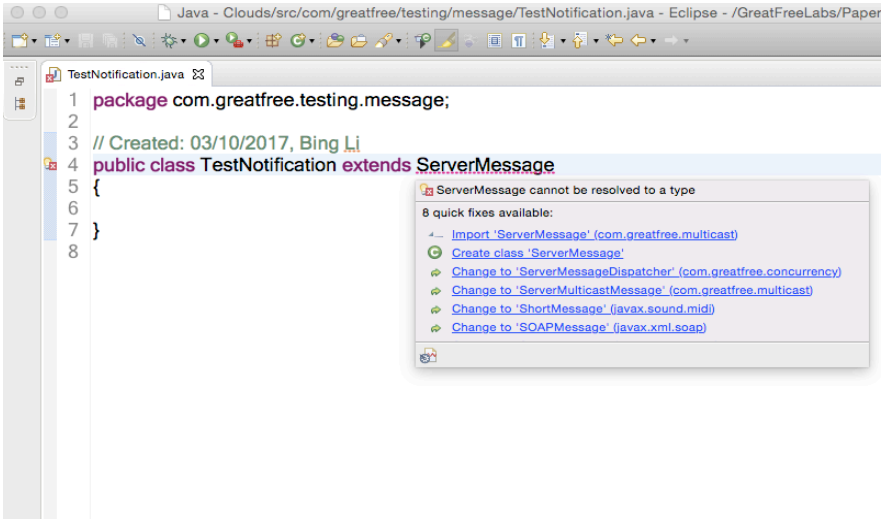


Figure 3.7 A popped up menu provides users with possible solutions to the compilation error

To fix the problem, you just move your mouse over the ServerMessage. Then, a prompt menu is popped up. It offers you possible options how to correct the error, as shown in Figure 3.7. In fact, the error is caused because the extended parent class, ServerMessage, is not imported. So you just select the first solution from the popped

up menu to solve the issue. Then, the editor for TestNotification should be exactly like Figure 3.8.

Unfortunately, another error is still existed even after ServerMessage is imported. To fix the error, you can still move your mouse over the red line. Then, according to the popped up menu, it indicates that constructors are required for the class. In our case, since we start to learn how to program notifications, we just select a simpler one, i.e., the first one, "Add constructor 'TestNotification(int)', as shown in Figure 3.9. After the constructor is added, you still notice that a warning emerges since there is a yellow line under TestNotification, as shown in Figure 3.10.

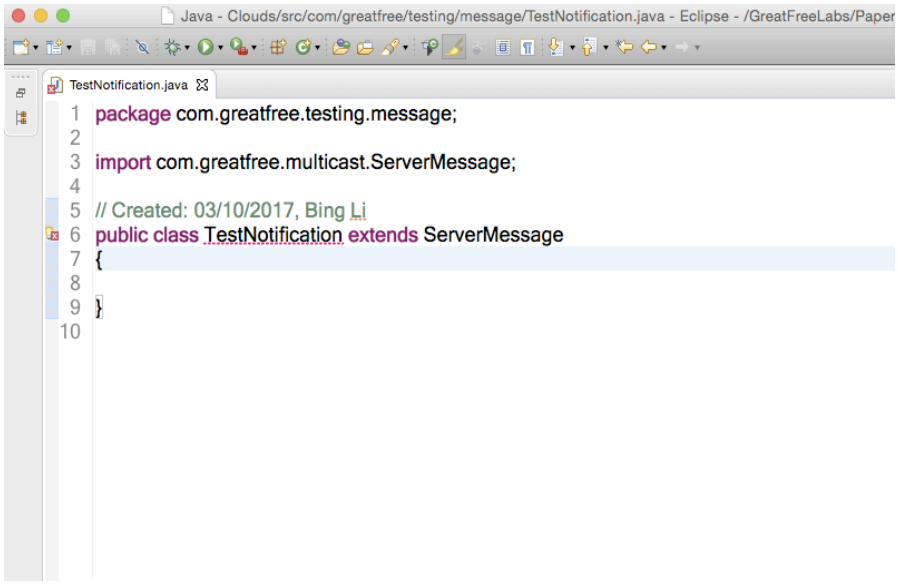


Figure 3.8 After ServerMessage is imported, it is required to fix the error of missing constructors

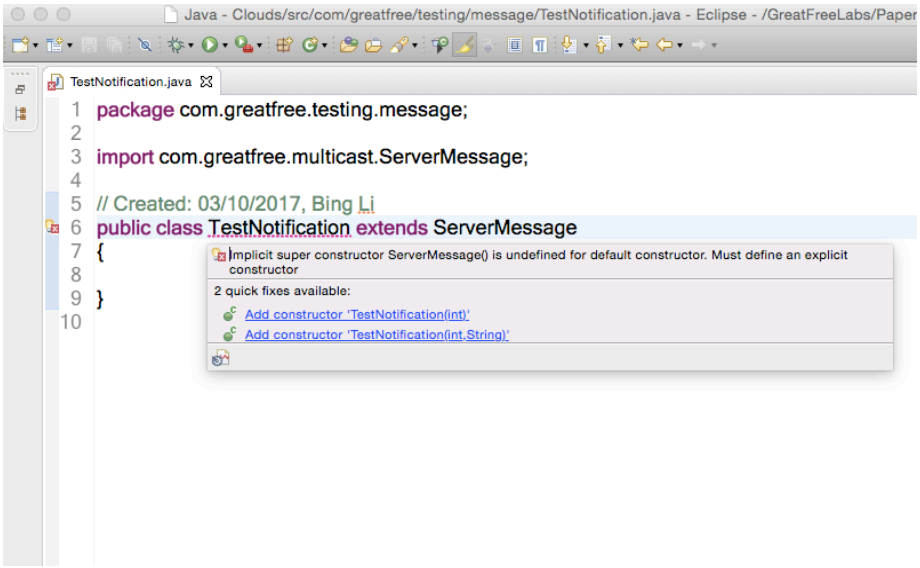


Figure 3.9 Select the first option to add a constructor to TestNotification.java

Similar to the approach to fix the compilation errors, you can also get hints from the popped up menu to remove the warning. You can do that by selecting either the first one or the second one, as shown in Figure 3.11. In my case, the second one is selected. Immediately after either of the options is selected, the editor for TestNotification.java should be exactly like Figure 3.12.

According to Figure 3.12, you find that a so-called serial version UID is generated automatically. For any class to be transmitted over network and persisted on disk, it must implement the interface of Serializable [1]. Then, such an identification code must be generated. It makes your code look messy to some extent. Anyway, just follow the above steps to fix the errors and remove warnings. Just leave the number there and it does not affect our coding procedure.

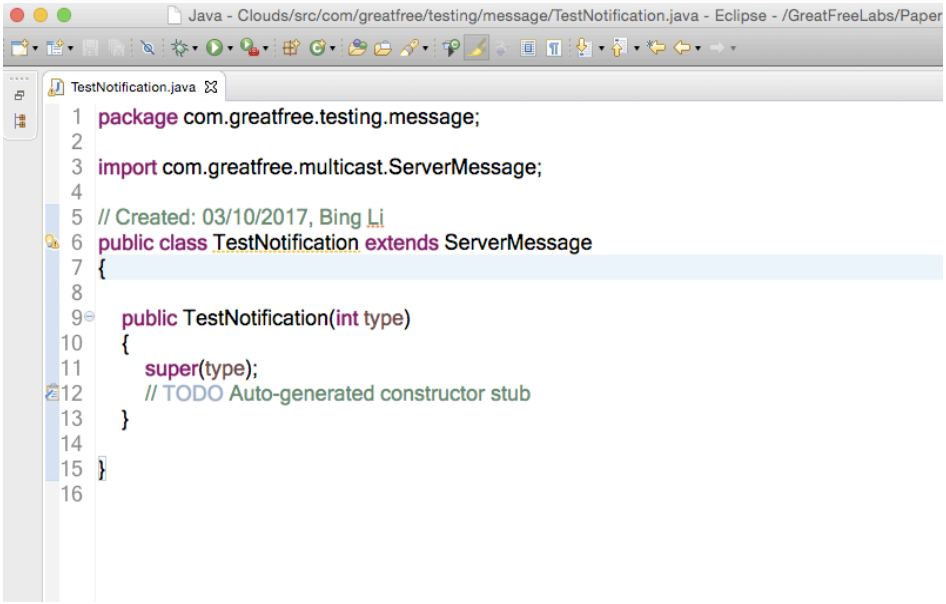


Figure 3.10 TestNotification has a warning after the constructor is added

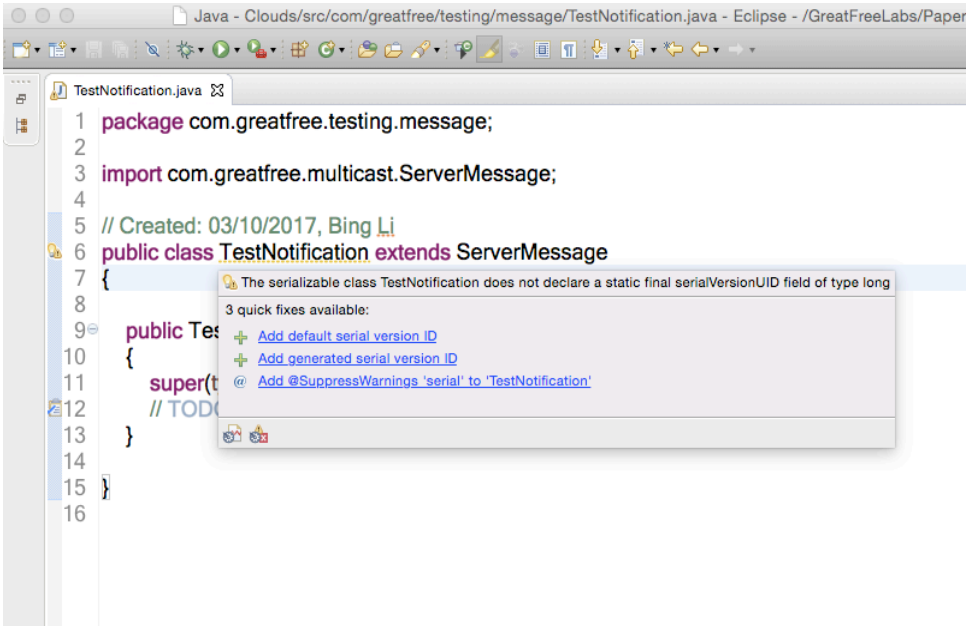


Figure 3.11 A popped up menu helps programmers the remove warning

2.3 Defining an Integer Constant to Represent the Notification

Once if the errors and warnings caused by extending ServerMessage are removed, you can move forward. As mentioned before, when programming, an experienced developer always follow samples. So now it is time for you to return to List 3.1, i.e., the code of WeatherNotification.java. Compared with your current code in Figure 3.12, one obvious difference is that its constructor has an integer constants, MessageType.WEATHER_NOTIFICATION. You must wonder what it is. Actually, in GreatFree, you need to define a unique integer constant to represent each message you create.

It is convenient to do that in GreatFree. First, you just simply put your cursor on MessageType at Line 20 inside the constructor of WeatherNotification. Then, press F3 if you are in Eclipse. You will see the code of MessageType is opened in Eclipse, as shown in Figure 3.13.

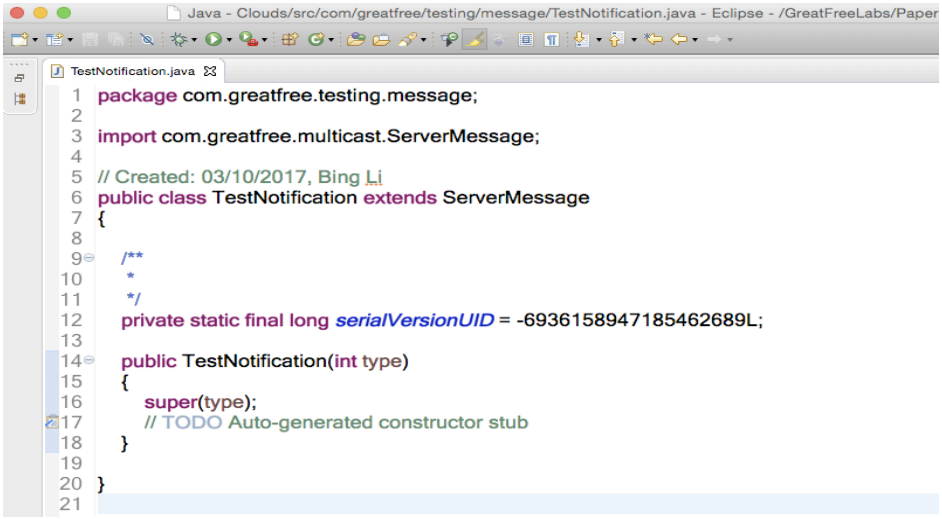


Figure 3.12 TestNotification.java after errors and warnings are removed

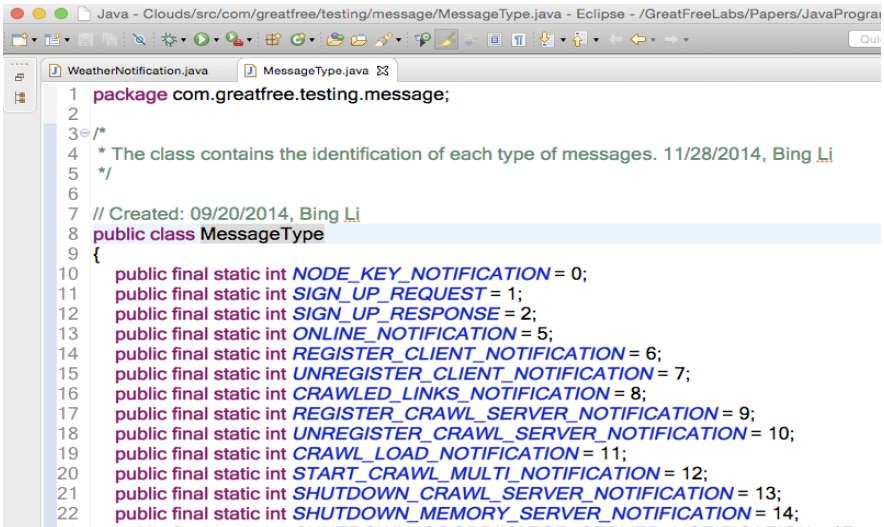


Figure 3.13 MessageType.java is opened

By the way, you can open the class, MessageType, from the package explorer as well. However, the manner is a little bit boring and not straightforward. Since GreatFree puts forward an approach of programming, Copy-Paste-Replace (CPR), you are highly recommended to press F3 on the classes you are interested instead of finding the ones from the package explorer. Even though they are sorted in the alphabet order, it is annoyed, especially when your project becomes large.

The code of MessageType contains all of the integer constants in the current project. But it is not necessary to keep your new constant into the class since it is just an integer. You can put it anywhere inside the project. The reason we would like to keep the value within the class of MessageType is to manage the system conveniently.

In the current version of MessageType, fifty-four constants are contained from 0 to 53. To ease the procedure, you can add a new integer following the same way. Therefore, your notification, TestNotification, can be represented by the number, 54. You can define a new variable, TEST_NOTIFICATION, and assign the value to it. After that, the code of MessageType is updated and a new line, Line 64, in List 3.2.

```
1   package com.greatfree.testing.message;
2
3   /*
4   * The class contains the identification of each type of messages. 11/28/2014, Bing Li
5   */
6
7   // Created: 09/20/2014, Bing Li
8   public class MessageType
9   {
10      public final static int NODE_KEY_NOTIFICATION = 0;
11      public final static int SIGN_UP_REQUEST = 1;
12      public final static int SIGN_UP_RESPONSE = 2;
13      public final static int ONLINE_NOTIFICATION = 5;
14      public final static int REGISTER_CLIENT_NOTIFICATION = 6;
15      public final static int UNREGISTER_CLIENT_NOTIFICATION = 7;
16      public final static int CRAWLED_LINKS_NOTIFICATION = 8;
17      public final static int REGISTER_CRAWL_SERVER_NOTIFICATION = 9;
18      public final static int UNREGISTER_CRAWL_SERVER_NOTIFICATION = 10;
19      public final static int CRAWL_LOAD_NOTIFICATION = 11;
20      public final static int START_CRAWL_MULTI_NOTIFICATION = 12;
21      public final static int SHUTDOWN_CRAWL_SERVER_NOTIFICATION = 13;
22      public final static int SHUTDOWN_MEMORY_SERVER_NOTIFICATION = 14;
23      public final static int SHUTDOWN_COORDINATOR_SERVER_NOTIFICATION = 15;
24      public final static int STOP_CRAWL_MULTI_NOTIFICATION = 16;
25      public final static int STOP_MEMORY_SERVER_NOTIFICATION = 17;
26      public final static int REGISTER_MEMORY_SERVER_NOTIFICATION = 18;
27      public final static int UNREGISTER_MEMORY_SERVER_NOTIFICATION = 19;
28      public final static int ADD_CRAWLED_LINK_NOTIFICATION = 20;
29      public final static int IS_PUBLISHER_EXISTED_REQUEST = 21;
30      public final static int IS_PUBLISHER_EXISTED_RESPONSE = 22;
31      public final static int SEARCH_KEYWORD_REQUEST = 23;
32      public final static int SEARCH_KEYWORD_RESPONSE = 24;
33      public final static int IS_PUBLISHER_EXISTED_ANYCAST_REQUEST = 25;
34      public final static int IS_PUBLISHER_EXISTED_ANYCAST_RESPONSE = 26;
35      public final static int SEARCH_KEYWORD_BROADCAST_REQUEST = 27;
36      public final static int SEARCH_KEYWORD_BROADCAST_RESPONSE = 28;
37      public final static int SHUTDOWN_REGULAR_SERVER_NOTIFICATION = 29;
38      public final static int WEATHER_NOTIFICATION = 30;
39      public final static int WEATHER_REQUEST = 31;
40      public final static int WEATHER_RESPONSE = 32;
41      public final static int CLIENT_FOR_BROADCAST_NOTIFICATION = 33;
42      public final static int CLIENT_FOR_UNICAST_NOTIFICATION = 34;
43      public final static int CLIENT_FOR_ANYCAST_NOTIFICATION = 35;
43      public final static int CLIENT_FOR_BROADCAST_REQUEST = 36;
45      public final static int CLIENT_FOR_BROADCAST_RESPONSE = 37;
46      public final static int CLIENT_FOR_UNICAST_REQUEST = 38;
47      public final static int CLIENT_FOR_UNICAST_RESPONSE = 39;
48      public final static int CLIENT_FOR_ANYCAST_REQUEST = 40;
49      public final static int CLIENT_FOR_ANYCAST_RESPONSE = 41;
```

```
50      public final static int BROADCAST_NOTIFICATION = 42;
51      public final static int UNICAST_NOTIFICATION = 43;
52      public final static int ANYCAST_NOTIFICATION = 44;
53      public final static int BROADCAST_REQUEST = 45;
54      public final static int BROADCAST_RESPONSE = 46;
55      public final static int UNICAST_REQUEST = 47;
56      public final static int UNICAST_RESPONSE = 48;
57      public final static int ANYCAST_REQUEST = 49;
58      public final static int ANYCAST_RESPONSE = 50;
59      public final static int SHUTDOWN_DN_NOTIFICATION = 51;
60      public final static int SHUTDOWN_COORDINATOR_NOTIFICATION = 52;
61      public final static int STOP_DN_NOTIFICATION = 53;
62
63      public final static int TEST_NOTIFICATION = 54;
64  }
```

List 3.2 The code of MessageType.java

2.4 Typing Code Into the Notification

Now it is time for you to type some code into the notification, TestNotification, you just created.

The first task to write any of your messages is to initialize the corresponding constants to them. In this case, the integer constant, TEST_NOTIFICATION, defined in MessageType, should be imported into the code, TestNotification. If you have no idea how to do that, you just follow the sample code, WeatherNotification.java, in List 3.1 again. According to Line 20 of the sample, the constant of the message must be the parameter of the constructor of its parent class, ServerMessage. So, you need to do the same thing exactly. Then, your code should be exactly like the one shown in Figure 3.14.

Figure 3.14 The constant representing the type of TestNotification is imported

Next, intuitively, you are expected to put some data into the notification such that they can be transmitted over the Internet. Since this is the first one you program with GreatFree, I suggest you not to make it too complicated. For me, I just put one attribute of string into it. For example, one string, testMessage, is added into the class.

The attribute of testMessage is usually expected to be initialized in the constructor of TestNotification. So you need to add relevant code to do that.

Additionally, since the parameter, type, in the constructor of TestNotification is replaced by the constant TEST_NOTIFICATION, you just remove it from the parameter list of the constructor.

Finally, you are suggested to add a setter and a getter for the attribute. After all of those are done, the updated TestNotification should be exactly like List 3.3.

```
1   package com.greatfree.testing.message;
2
3   import com.greatfree.multicast.ServerMessage;
4
5   // Created: 03/10/2017, Bing Li
6   public class TestNotification extends ServerMessage
7   {
8       private static final long serialVersionUID = -6936158947185462689L;
9
10      private String testMessage;
11
12      public TestNotification(String testMessage)
13      {
14          super(MessageType.TEST_NOTIFICATION);
15          this.testMessage = testMessage;
16      }
17
18      public String getTestMessage()
19      {
20          return this.testMessage;
21      }
22
23      public void setTestMessage(String testMessage)
24      {
25          this.testMessage = testMessage;
26      }
27  }
```

List 3.3 The complete code of TestNotification.java

3. The Server Side

So far so good. You already has your first message, TestNotification. You must look forward to sending it over the Internet. That is also what GreatFree aims to achieve. Since we do not attempt to explain the details of the techniques in the chapter, I just give you a rough introduction before we start to program continually.

First of all, you should be aware of the fact that it is really complicated to send a message over the Internet if you program with a generic language, like C, C++ or Java SE. Instead of using TCP/IP technology [1] only, you are required to take care about a bunch of issues [2] that definitely force you to surrender. Fortunately, you are not programming with GreatFree such that you are free from those stuffs.

In addition, although now you can follow a sample code to program smoothly, I still indicate that it is anticipated for you to know that your first notification is sent over one particular distributed mode, the Client/Server model [3], which is just one of the distributed ones over the Internet. If in other environments, such as Peer-to-Peer [3] or

clusters [3], you should learn how to handle the notification transmission. At least, there are some differences such that you cannot use the code in this section anywhere.

Moreover, you will start to program over the server side in the Client/Server model. There are many lines of code on the server. Do not worry about that especially when you are interested in upper level applications only. Just concentrate on the skills I discuss in the chapter.

Finally, the model of Client/Server is very popular. So in most simple applications, those code can be employed to deal with problems in practice. The coding experiences you learn in this section must give you a general taste about GreatFree programming although the code is a little bit narrow in terms of its distributed models.

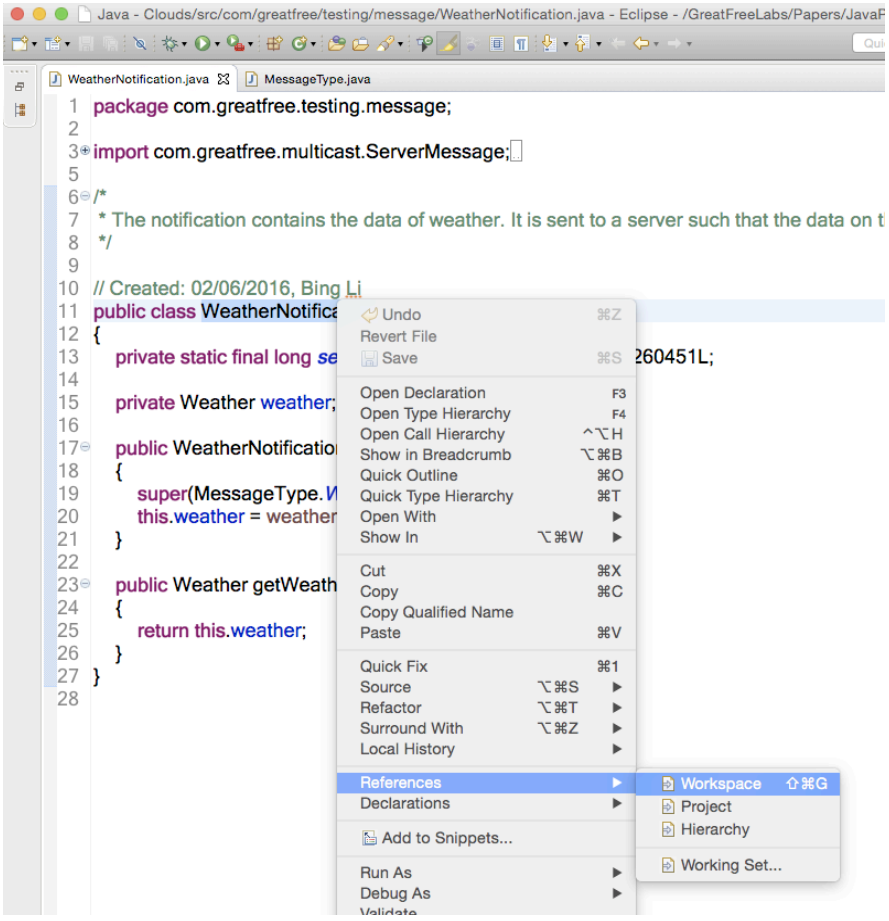


Figure 3.15 Search references of the notification, WeatherNotification, in Eclipse

3.1 The Sample Thread

Similar to the approach of creating the notification in Section 2, you must expect to follow some samples as well when working on the server side. If so, I am happy to say you are becoming familiar with the style of GreatFree gradually.

For that, you can get an assistance from Eclipse since the IDE has a function to find references of one specific class, method or an attribute. You just followed one sample code, WeatherNotification.java, such that you can find other samples you might

follow by searching its references. It is easy to do that in Eclipse. Put the cursor on the name of the notification, WeatherNotification, in the editor and then press the compound button, Command+Shift+G (Mac OS X) or Ctrl+Shift+G (Linux or Windows). Another way to search references is to right-click on WeatherNotification to select the menu option, References, and then click the Workspace in the bottom submenu, as shown in Figure 3.15. Then, the search window in Eclipse is opened and the results of the search are shown in a tree hierarchy like Figure 3.16.

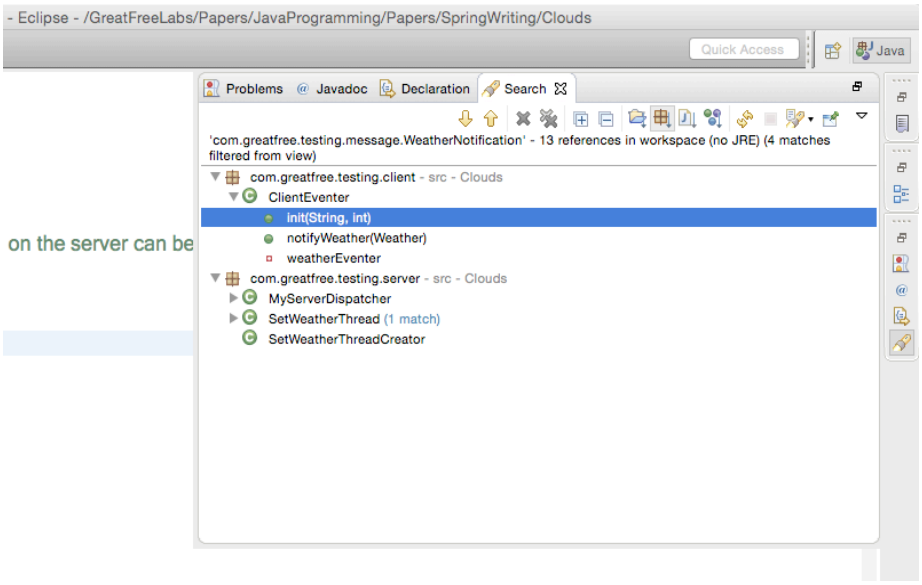


Figure 3.16 The references of WeatherNotification

The references of WeatherNotification are listed in Table 3.1. Actually, all of the references are the one you need to work on for a notification. In this sense, all of them are also the sample code you need to follow for the new created notification, TestNotification.

According to Table 3.1, the references of WeatherNotification exist on both sides, the client and the server. That is reasonable since WeatherNotification is one message that is sent from the client to the server. You can follow either of them to finish the coding.

Class	Explanation	Package	Side
ClientEventer	The client side that sends notifications to the server side	com.greatfree.testing.client	Client
MyServerDispatcher	The dispatcher that is responsible for dispatching received messages to corresponding threads such that those messages are processed concurrently	com.greatfree.testing.server	Server
SetWeatherThread	The thread that deals with the notification of WeatherNotification	com.greatfree.testing.server	Server
SetWeatherThreadCreator	The thread that creates a new instance of SetWeatherThread in case that existing instances are too busy	com.greatfree.testing.server	Server

Table 3.1 The references of WeatherNotification

Another important task to be finished before we move forward is to find new classes you need to create for your notification, TestNotification. According to Table 3.1, there are four classes listed, i.e., ClientEventer, MyServerDispatcher,

SetWeatherThread and SetWeatherThreadCreator. With them, it is easy to know that the later two, SetWeatherThread and SetWeatherThreadCreator, are the ones we need to create since they are designed particularly for WeatherNotification whereas the other two are the ones for any messages according to the explanations to them. For that, we can create a new table, Table 3.2, which lists all of classes for WeatherNotification. Correspondingly, as you are attempting to program a notification, you are supposed to create similar classes, TestNotificationThread and TestNotificationThreadCreator, for TestNotification. More important, as they are counterparts of WeatherNotification, you can CPR them when programming TestNotification.

Type	Classes for WeatherNotification	Classes for TestNotification
ID	MessageType.WEATHER_NOTIFICATION	MessageType.TEST_NOTIFICATION
Notification	WeatherNotification	TestNotification
Thread	SetWeatherThread	TestNotificationThread
Thread Creator	SetWeatherThreadCreator	TestNotificationThreadCreator

Table 3.2 The counterparts of WeatherNotification and TestNotification

In short, according to Table 3.1, the references of the sample, the same type of message, i.e., the Notification of WeatherNotification, you can know which class you need to follow and change. When programming, you can CPR the counterparts based on Table 3.2, the counterparts of the sample notification, WeatherNotification, and the new one, TestNotification.

3.2 Creating the Thread for the Notification

To continue, you are recommended to start from the sample, SetWeatherThread, at the server side according to Table 3.1. Although this is not the unique choice, I believe it is more straightforward. That means you need to create a thread at the server side to deal with your new created notification. Therefore, you need to open the code in the editor of Eclipse by double-clicking on the SetWeatherThread in the window of references of WeatherNotification in Figure 3.16. You can also find it from the package, com.greatfree.testing.server, although this way follows our intuition. The code of SetWeatherThread is listed in List 3.4.

```
1   package com.greatfree.testing.server;
2
3   import com.greatfree.concurrency.NotificationQueue;
4   import com.greatfree.testing.data.ServerConfig;
5   import com.greatfree.testing.message.WeatherNotification;
6   import com.greatfree.testing.server.resources.WeatherDB;
7
8   /*
9   * The thread implements following the pattern of notification queue. It receives a notification that contains the
10  * weather information to set the weather instance on the server. 02/11/2016, Bing Li
11  */
12
13  // Created: 02/10/2016, Bing Li
14  public class SetWeatherThread extends NotificationQueue<WeatherNotification>
15  {
16      /*
17      * Initialize the thread. 02/11/2016, Bing Li
18      */
19      public SetWeatherThread(int taskSize)
20      {
21          super(taskSize);
22      }
23
```

```
24          /*
25          * This is the kernel of the notification pattern that sets the weather instance
26          * concurrently. 02/11/2016, Bing Li
27          */
28          public void run()
29          {
30              // Declare an instance of WeatherNotification. 02/11/2016, Bing Li
31              WeatherNotification notification;
32              // The thread always runs until it is shutdown by the NotificationDispatcher. 02/11/2016, Bing Li
33              while (!this.isShutdown())
34              {
35                  // Check whether the notification queue is empty. 02/11/2016, Bing Li
36                  while (!this.isEmpty())
37                  {
38                      try
39                      {
40                          // Dequeue the notification. 02/11/2016, Bing Li
41                          notification = this.getNotification();
42                          // Set the value of the weather. 02/11/2016, Bing Li
43                          WeatherDB.SERVER().setWeather(notification.getWeather());
44                          // Collect the resource kept by the notification. 02/11/2016, Bing Li
45                          this.disposeMessage(notification);
46                      }
47                      catch (InterruptedException e)
48                      {
49                          e.printStackTrace();
50                      }
51                  }
52                  try
53                  {
54                      // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
55                      this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
56                  }
57                  catch (InterruptedException e)
58                  {
59                      e.printStackTrace();
60                  }
61              }
62          }
63      }
```

List 3.4 The code of SetWeatherThread.java

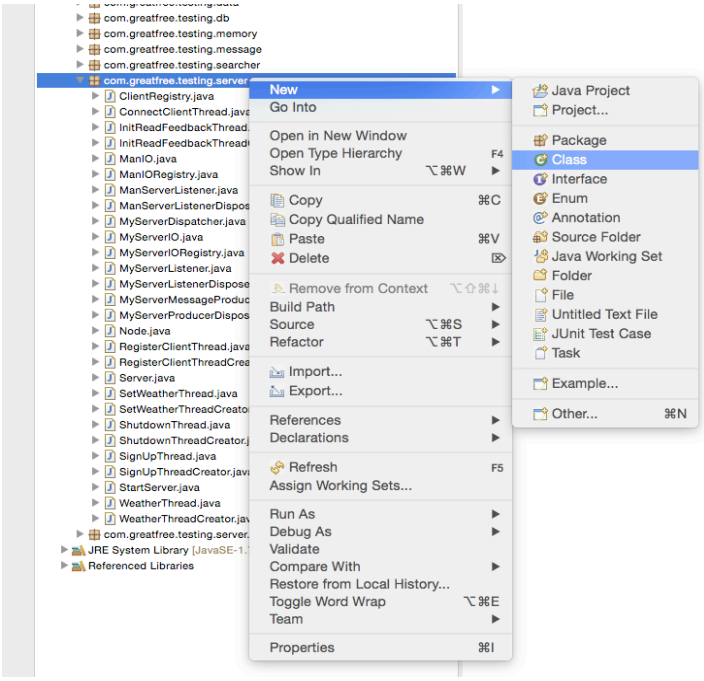


Figure 3.17 Create a thread for the notification on the server side

The code of SetWeatherThread contains one of the most important pattern of GreatFree. It is called the Double While Concurrency (DWC) in GreatFree. Since at this moment we focus on programming your notification by the approach of CPR, we just ignore what it means from the technical point of view.

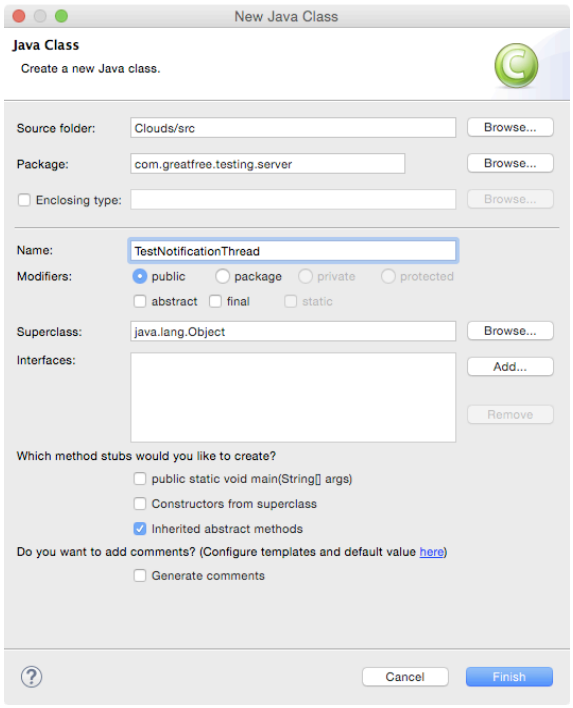


Figure 3.18 Name the new class, TestNotificationThread

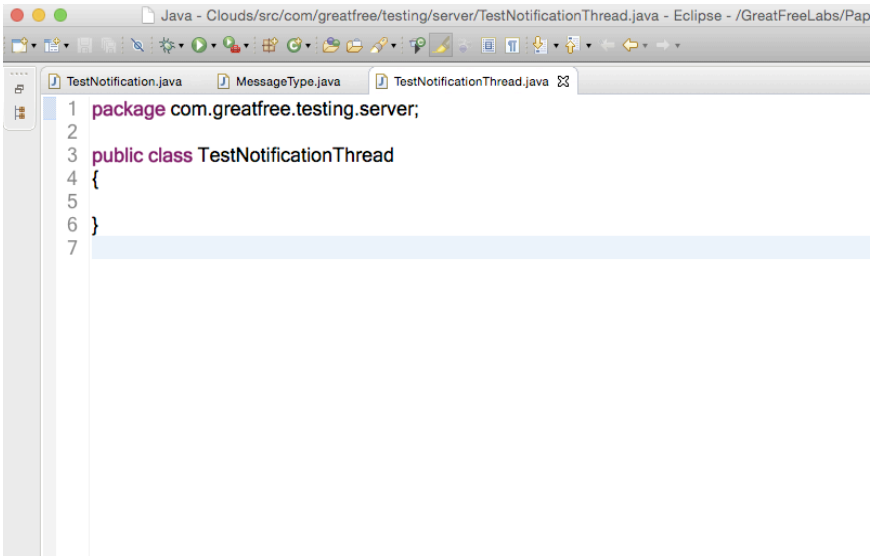


Figure 3.19 TestNotificationThread is opened in the editor of Eclipse

Before moving forward, you need to refer to Table 3.2, in which the counterparts of WeatherNotification and TestNotification are listed. Each for WeatherNotification has a one-to-one mapping component for TestNotification. When creating the stuffs for a new notification, you are just required to follow such a form to name and program

your code. With CPR, the classes of WeatherNotification should be refered by its respective counterpart of TestNotification in the table.

According to Table 3.2, now you need to follow the sample to create your own thread, TestNotificationThread, the counterpart of SetWeatherThread. Since the thread should work on the server side, you just create it by right-clicking on the package, com.greatfree.testing.server, as shown in Figure 3.17 and Figure 3.18. Thereafter, similar to the class, TestNotification, you created in Section 2, a new class, TestNotificationThread, is opened in the editor of Eclipse, as shown in Figure 3.19.

You are suggested to program TestNotificationThread using CPR. The first question is what to be copied into the new thread you just created? It is easy to answer. Except the class name, TestNotificationThread, you can copy all of the rest code into the editor shown in Figure 3.19. After that, your editor should be like the own shown in Figure 3.20.

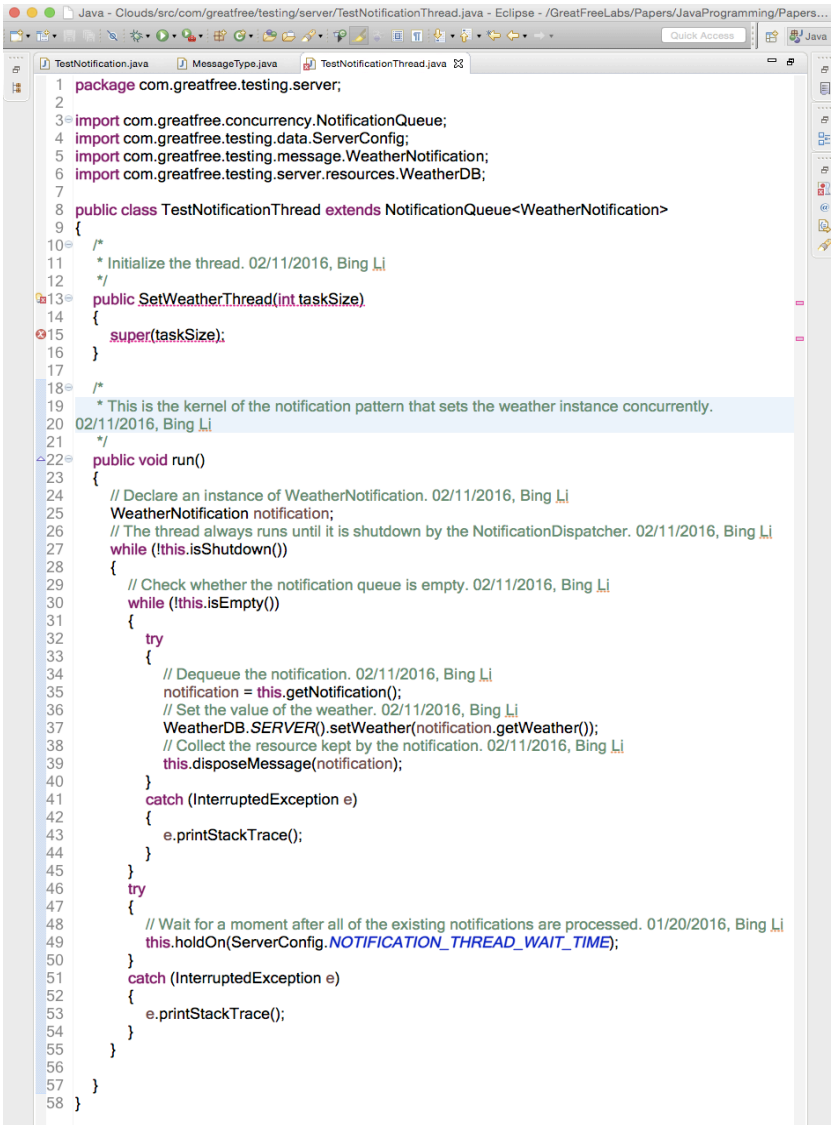


Figure 3.20 The code of TestNotificationThread.java after copying and pasting its counterpart, SetWeatherThread.java

After copying and pasting its counterpart, SetWeatherThread, there are definitely some compilation errors in TestNotificationThread. Do not worry since they can be fixed by the next simple operation, replacing. What you need to do is to replace the classes in TestNotificationThread with its counterparts in Table 3.2. That is, replace WeatherNotification with TestNotification in Line 8 and Line 25 and replace SetWeatherThread with TestNotificationThread in Line 13. After the replacement, your code should be like the one in Figure 3.21.

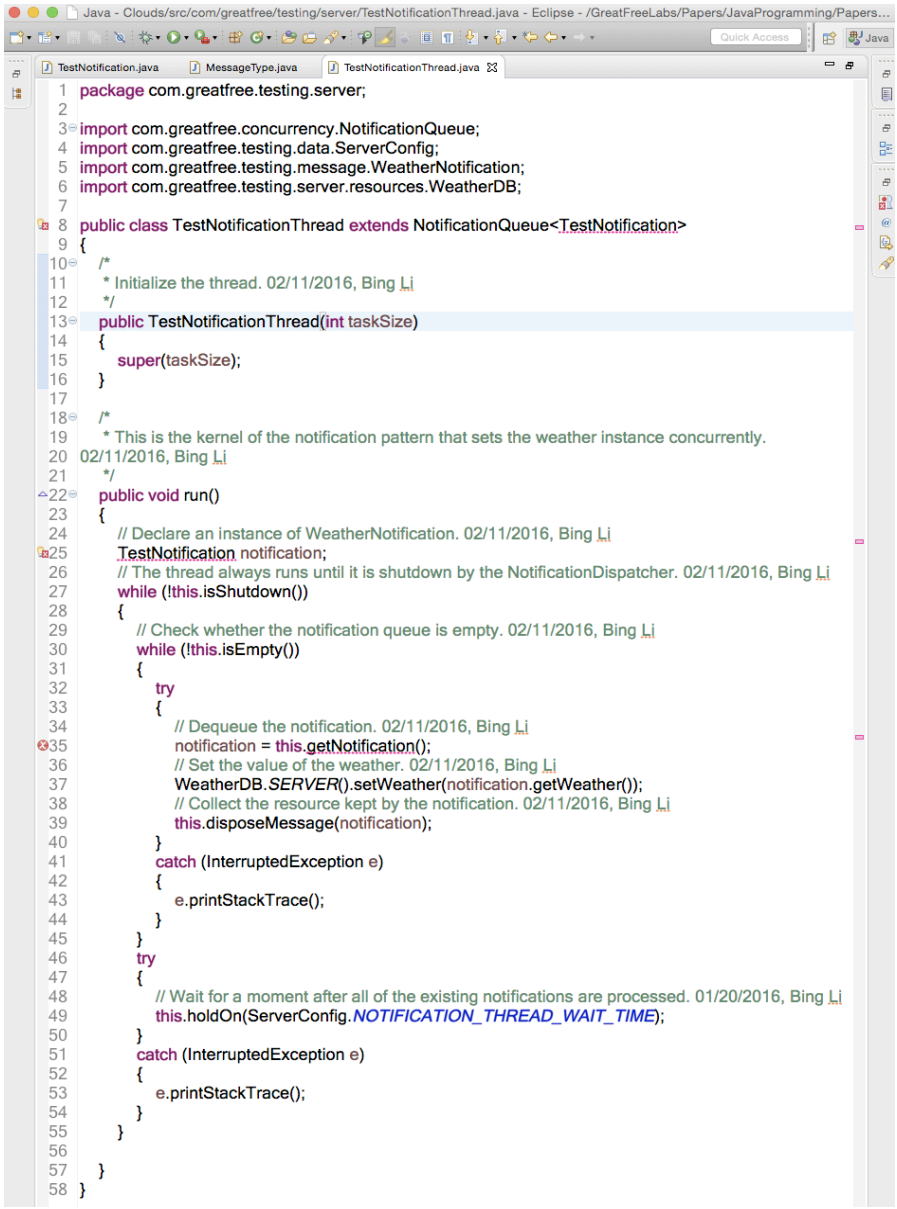


Figure 3.21 The code of TestNotificationThread.java after replacing its counterparts of SetWeatherThread.java

Now there are still some compilation errors in the code. You can just fix them by selecting prompts of Eclipse. After that, your code is almost done, as shown in Figure 3.22.

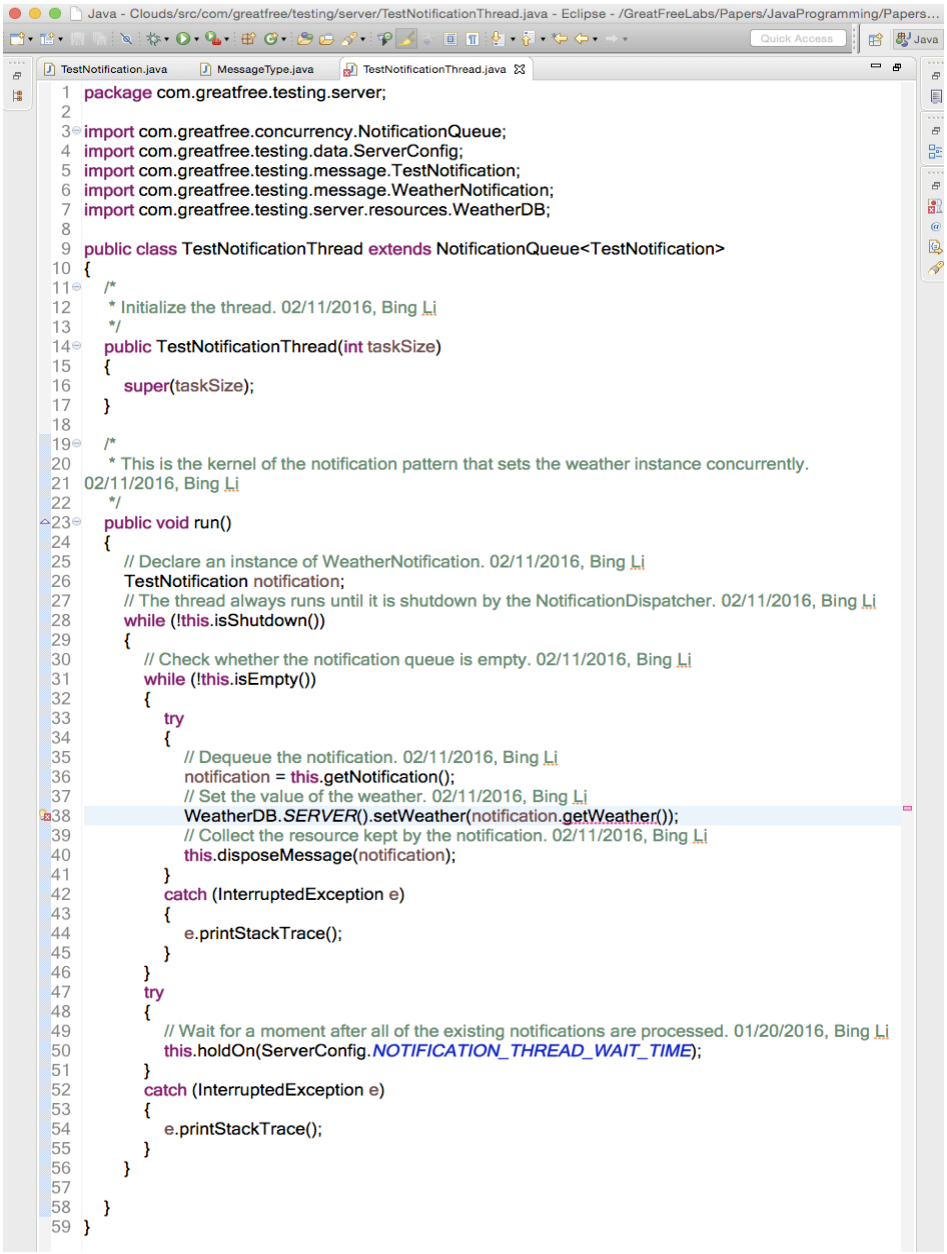


Figure 3.22 The code of TestNotifictionThread.java after fixing the compilation errors by selecting prompts of Eclipse

There is one error in Line 38, which has nothing to do with TestNotification since the class, WeatherDB, is not shown in Table 3.2. Actually, it is the logic to process the message of WeatherNotification. So you can remove it simply, as shown in Figure 3.23. Besides that, Line 6 and Line 7 should be removed as well because of the warnings. They have nothing to do with TestNotification now. Then your code is almost done, as shown in Figure 3.24.

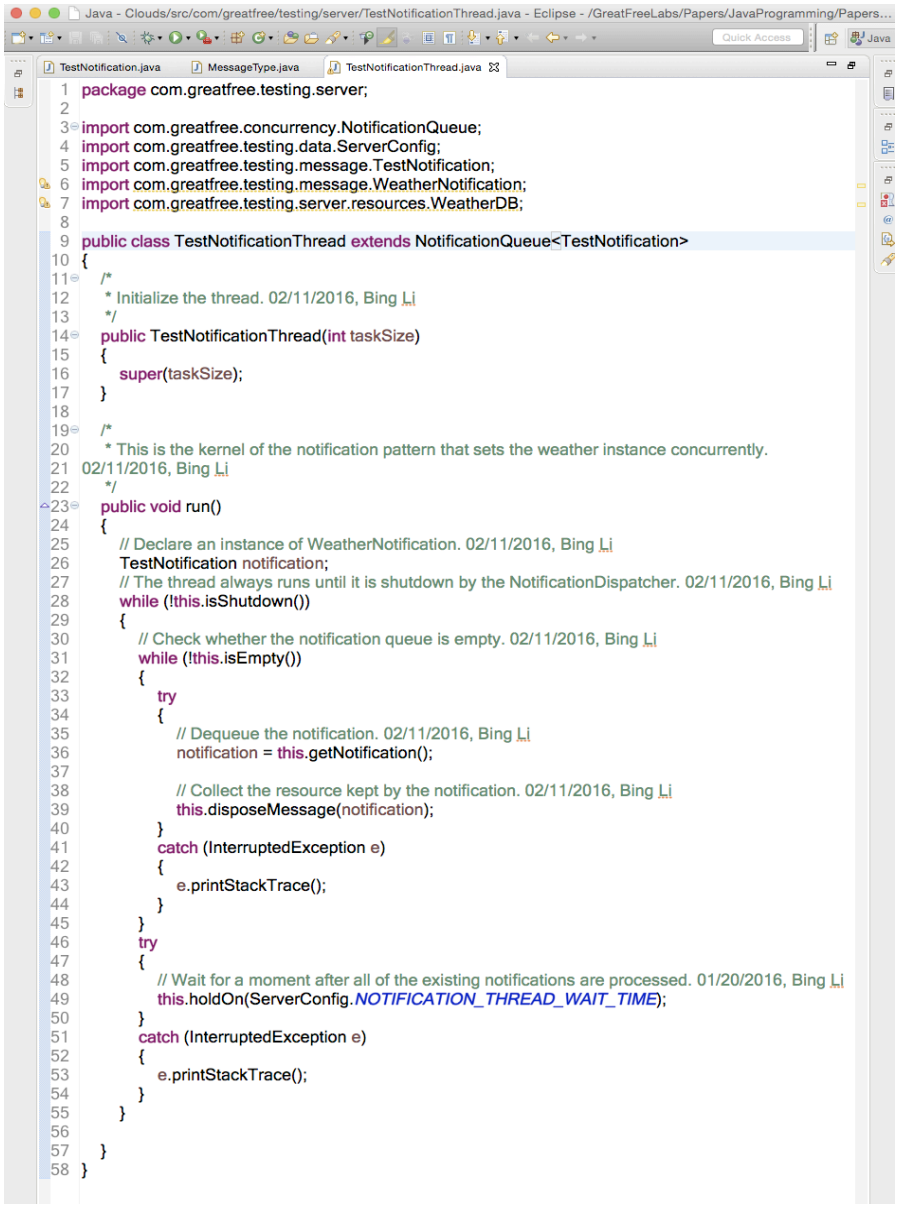


Figure 3.23 The code TestNotificationThread.java after removing the class, WeatherDB

So until now, there should be no any compilation errors and warnings at all. Moreover, it is the code that contains the pattern of DWC for notifications. You can reuse the code in all of the cases that need to process notifications concurrently. One shorting coming of the code is that it just receives the remote notification and does nothing on it. To be clear, you can insert one line to do something immediately after the one you get the notification. In this case, you just put one line immediately after Line 34, which takes one notification out by calling the method, this.getNotification(). You must remember TestNotification has a String attribute, testMessage. So you can just display it on screen in the code. Finally, your code is exact like the one in List 3.5 and Line 35 for a comment and Line 36 for displaying information are added to the list. Your task to add a thread to process the new notification is accomplished.

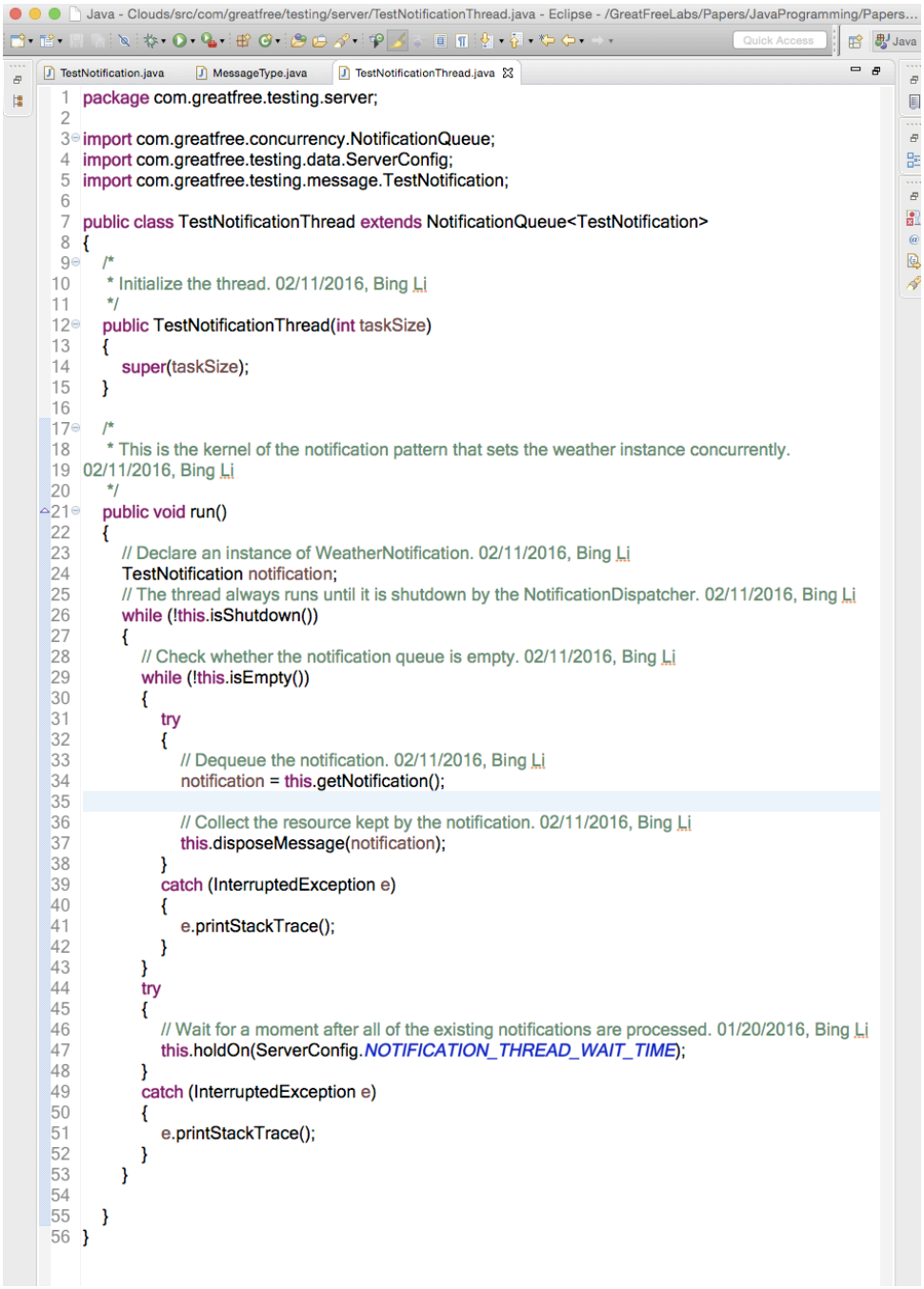


Figure 3.24 The code of TestNotificationThread.java after all errors and warnings caused by CPR are fixed

```
1   package com.greatfree.testing.server;
2
3   import com.greatfree.concurrency.NotificationQueue;
4   import com.greatfree.testing.data.ServerConfig;
5   import com.greatfree.testing.message.TestNotification;
6
7   // Created: 03/15/2017, Bing Li
8   public class TestNotificationThread extends NotificationQueue<TestNotification>
9   {
10      /*
11       * Initialize the thread. 02/11/2016, Bing Li
12       */
13      public TestNotificationThread(int taskSize)
14      {
15          super(taskSize);
```

```
16          }
17
18          /*
19           * This is the kernel of the notification pattern that sets the weather instance
20           * concurrently. 02/11/2016, Bing Li
21           */
22          public void run()
23          {
24              TestNotification notification;
25              // The thread always runs until it is shutdown by the NotificationDispatcher. 02/11/2016, Bing Li
26              while (!this.isShutdown())
27              {
28                  // Check whether the notification queue is empty. 02/11/2016, Bing Li
29                  while (!this.isEmpty())
30                  {
31                      try
32                      {
33                          // Dequeue the notification. 02/11/2016, Bing Li
34                          notification = this.getNotification();
35                          // Do something on your notification. 03/16/2017, Bing Li
36                          System.out.println(notification.getTestMessage());
37                          // Collect the resource kept by the notification. 02/11/2016, Bing Li
38                          this.disposeMessage(notification);
39                      }
40                      catch (InterruptedException e)
41                      {
42                          e.printStackTrace();
43                      }
44                  }
45                  try
46                  {
47                      // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
48                      this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
49                  }
50                  catch (InterruptedException e)
51                  {
52                      e.printStackTrace();
53                  }
54              }
55          }
56      }
```

List 3.5 The code of TestNotificationThread.java after all of classes related to SetWeatherThread are removed

You are suggested to go back and read Section 3.2 again. You must be aware of the easy procedure to program the thread, TestNotificationThread.java. The procedure can be summarized as the following steps.

1) Finding the thread sample according to the counterpart table of the sample notification, such as Table 3.2;

2) Copying-pasting the code of the sample except the class name to your newly created thread class;

3) Replacing the sample notification with its counterpart in the counterpart table of the sample notification, such as Table 3.2;

4) Removing the code that handles the sample notification;

5) Removing warnings caused by the previous removal.

During the entire procedure, the primary operations are just copy-paste-replace. So the programming process is named concisely as CPR.

3.3 Creating the Thread Creator

With respect to Table 3.1, on the server side, you are still required to work on two additional classes, SetWeatherThreadCreator and MyServerDispatcher, which are also references of WeatherNotification. Based on our previous experiences, you can find them by the searching reference function of Eclipse. You can also search the references the sample class, SetWeatherThread by Command+Shift+G (Mac OS X) or Ctrl+Shift+G (Linux or Windows). If doing so, you will get the results like the one shown in Figure 3.25.

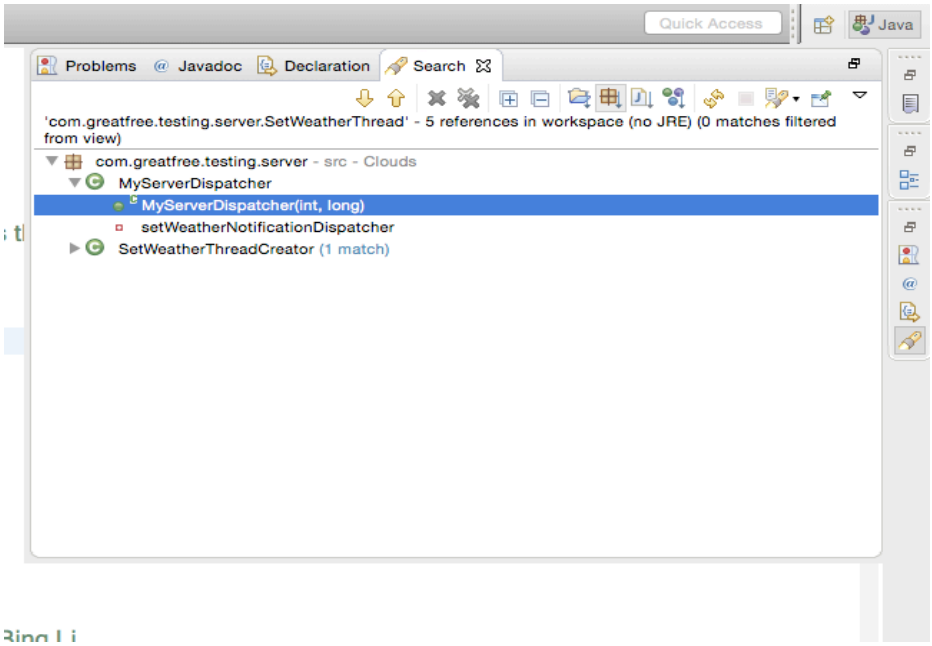


Figure 3.25 The references of SetWeatherThread

What you can do next is to double-click the class, SetWeatherThreadCreator, in the reference search result window in Figure 3.25. Then, you get the code SetWeatherThreadCreator.java in List 3.6.

```
1   package com.greatfree.testing.server;
2
3   import com.greatfree.concurrency.NotificationThreadCreatable;
4   import com.greatfree.testing.message.WeatherNotification;
5
6   /*
7    * The class creates an instance of the thread, SetWeatherThread. It is used by the NotificationDispatcher to
8    * manage the thread count and relevant resources. 02/15/2016, Bing Li
9    */
10
11  // Created: 02/15/2016, Bing Li
12  public class SetWeatherThreadCreator implements NotificationThreadCreatable
13      <WeatherNotification, SetWeatherThread>
14  {
15      @Override
16      public SetWeatherThread createNotificationThreadInstance(int taskSize)
17      {
18          return new SetWeatherThread(taskSize);
19      }
```

```
20  }
```

List 3.6 The code of SetWeatherThreadCreator.java

The code is simple. With respect to Table 3.2, you just need to create a new class, TestNotificationThreadCreator, which is named in a similar way to SetWeatherThreadCreator, as shown in Figure 3.26.

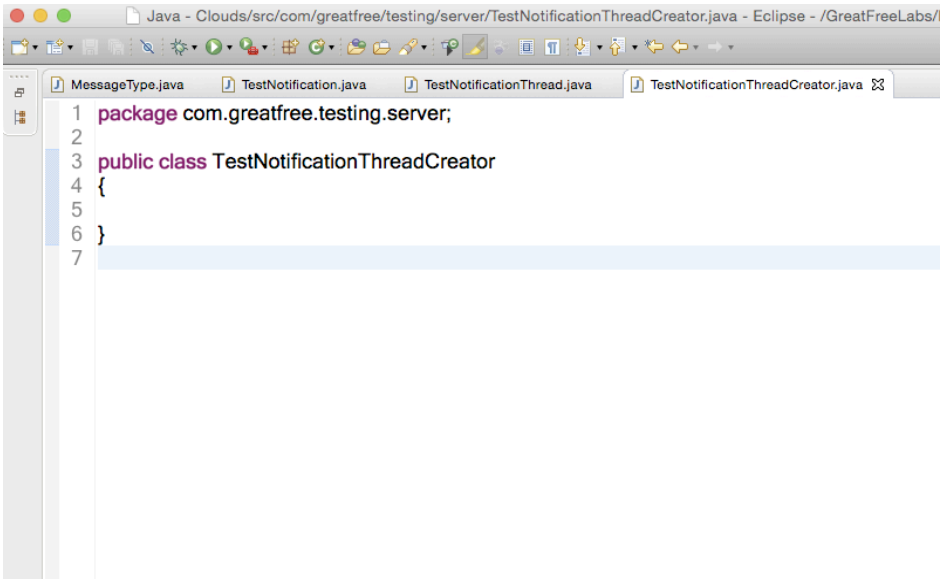


Figure 3.26 The code of TestNotificationThreadCreator.java is created

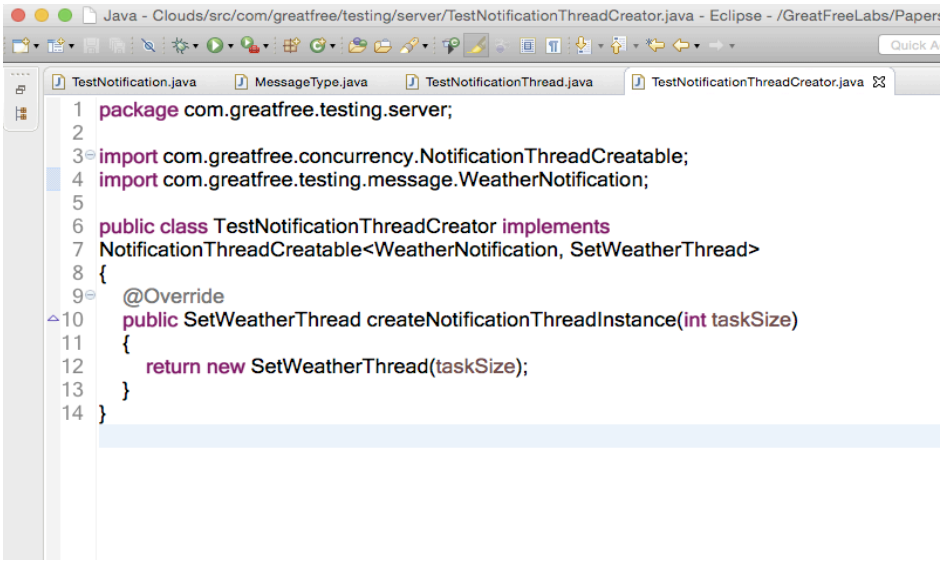


Figure 3.27 The code of TestNotificationThreadCreator.java after copying the code from SetWeatherThreadCreator.java

To continue, you can just follow the approach of CPR we discussed above. You can refer to the below steps to complete your code, which are similar to what we did for TestNotificationThread.java in Section 3.2.

1) Copy the code in List 3.6 except the class name, SetWeatherThreadCreator, and paste it to your code to overwrite the code except the class name, TestNotificationThreadCreator. Then, your code in the editor is updated as shown in Figure 3.27;

2) Replace WeatherNotification with TestNotification and replace SetWeatherThread with TestNotificationThread according to Table 3.2, as shown in Figure 3.28;

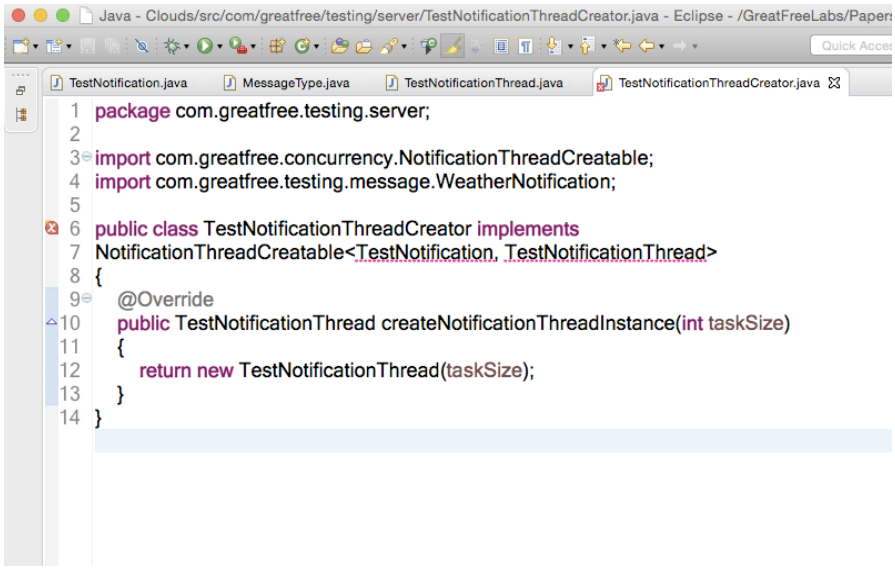


Figure 3.28 Replace the counterparts for TestNotificationThreadCreator.java

3) Fix the compilation errors by selecting the options prompted by Eclipse in Line 7, as shown in Figure 3.29;

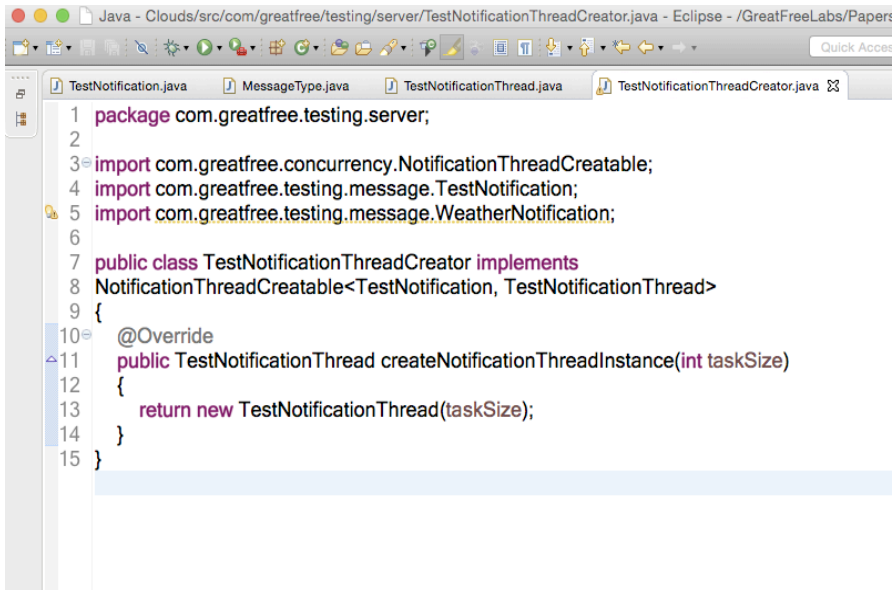


Figure 3.29 The code of TestNotificationThreadCreator.java after fixing errors by selecting options prompted by Eclipse and removing

4) Remove the warning in Line 5 since the class, WeatherNotification, has nothing to do with the current class, TestNotificationThreadCreator, as shown in Figure 3.30.

After that, you get the correct code of TestNotificationThreadCreator as shown in List 3.7.

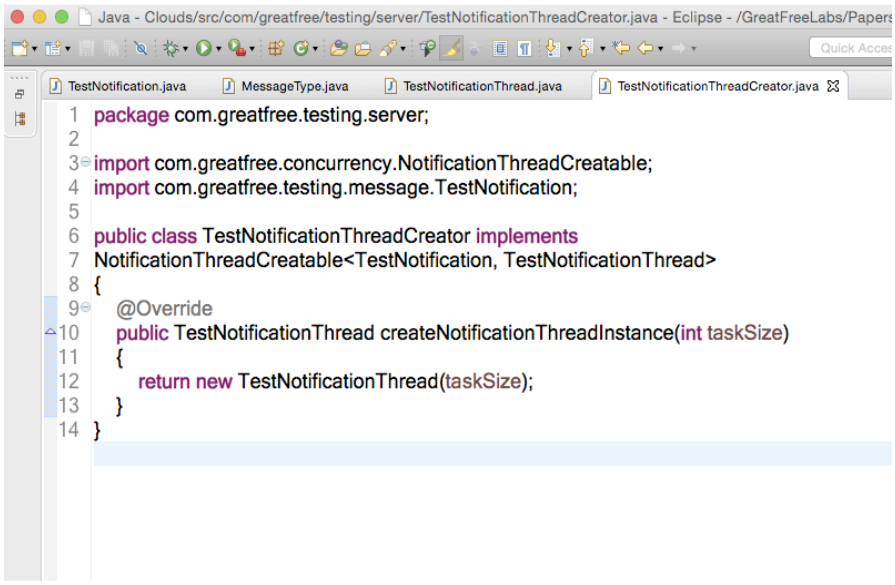


Figure 3.30 Remove the warning from TestNotificationThreadCreator.java

```
1     package com.greatfree.testing.server;
2
3     import com.greatfree.concurrency.NotificationThreadCreatable;
4     import com.greatfree.testing.message.TestNotification;
5
6     // Created: 03/16/2017, Bing Li
7     public class TestNotificationThreadCreator implements NotificationThreadCreatable
8         <TestNotification, TestNotificationThread>
9     {
10        @Override
11        public TestNotificationThread createNotificationThreadInstance(int taskSize)
12        {
13            return new TestNotificationThread(taskSize);
14        }
15    }
```

List 3.7 The code of TestNotificationThreadCreator.java

During the four steps to create the code of TestNotificationThreadCreator.java, the primary operations are copying and pasting code from its counterpart, SetWeatherThreadCreator and then replacing the classes with their counterparts in Table 3.2. The entire procedure is straightforward and rapid.

3.4 Revising the Server Dispatcher

You must remember in Table 3.1, Figure 3.16 and Figure 3.25, there is one reference, MyServerDispatcher, of WeatherNotification or SetWeatherThread. We have never touched it until now. Simply speaking, it is a concurrency mechanism to respond remote messages. We have already created three components to learn how to program a notification. Now it is time to embed them into the concurrency code such that your notification can be handled.

We can achieve the goal with the same approach, CPR. Let us first open the code. You can do that by searching the references of either of WeatherNotification, SetWeatherThread or SetWeatherThreadCreator. The results for the above three searches are shown in Figure 3.16, Figure 3.25 and Figure 3.31, respectively.

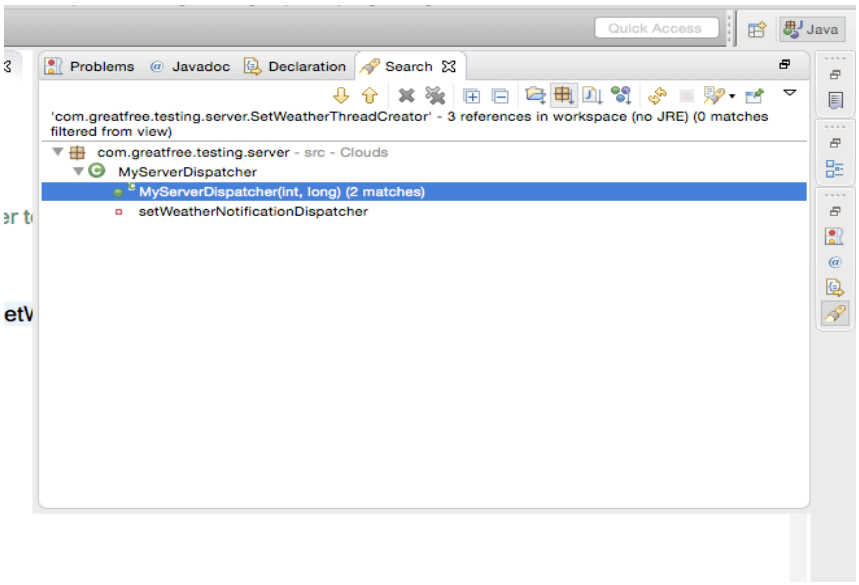


Figure 3.31 The references of SetWeatherThreadCreator

No matter which reference you search, the class, MyServerDispatcher, is always one of the results. You can open it by double-clicking its icon. The complete code is listed in List 3.8.

You can also open it by finding from the package explorer, like what is shown in Figure 3.32. But it is so comfortable and straightforward.

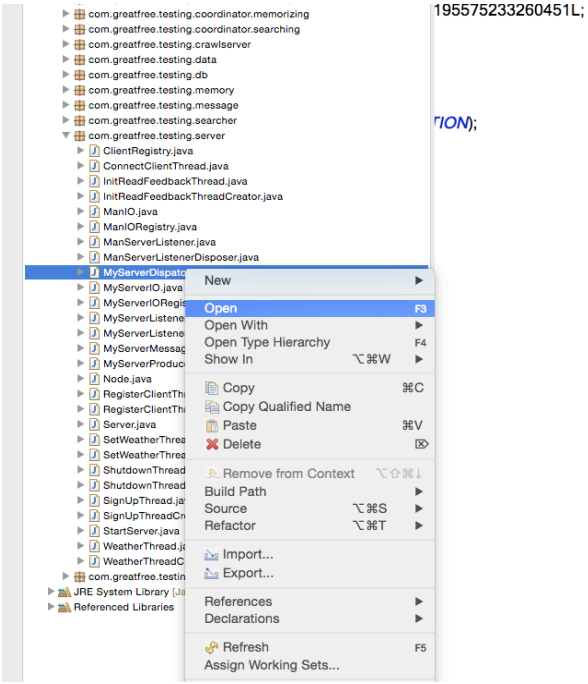


Figure 3.32 Open the code, MyServerDispatcher.java

```
1   package com.greatfree.testing.server;
2
3   import java.util.Calendar;
4
5   import com.greatfree.concurrency.NotificationDispatcher;
6   import com.greatfree.concurrency.RequestDispatcher;
7   import com.greatfree.concurrency.Scheduler;
8   import com.greatfree.concurrency.ServerMessageDispatcher;
9   import com.greatfree.message.InitReadNotification;
10  import com.greatfree.message.SystemMessageType;
11  import com.greatfree.multicast.ServerMessage;
12  import com.greatfree.remote.OutMessageStream;
13  import com.greatfree.testing.data.ServerConfig;
14  import com.greatfree.testing.message.MessageType;
15  import com.greatfree.testing.message.RegisterClientNotification;
16  import com.greatfree.testing.message.ShutdownServerNotification;
17  import com.greatfree.testing.message.SignUpRequest;
18  import com.greatfree.testing.message.SignUpResponse;
19  import com.greatfree.testing.message.SignUpStream;
20  import com.greatfree.testing.message.WeatherNotification;
21  import com.greatfree.testing.message.WeatherRequest;
22  import com.greatfree.testing.message.WeatherResponse;
23  import com.greatfree.testing.message.WeatherStream;
24
25  /*
26  * This is an implementation of ServerMessageDispatcher. It contains the concurrency mechanism to
27  * respond clients' requests and receive clients' notifications for the server. 09/20/2014, Bing Li
28  */
29
30  /*
31  * Revision Log
32  *
33  * The initialization request dispatcher is modified. When no tasks are available for some time, it needs to
34  * be shut down. 01/14/2016, Bing Li
35  *
36  */
37
38  // Created: 09/20/2014, Bing Li
39  public class MyServerDispatcher extends ServerMessageDispatcher<ServerMessage>
40  {
41      // Declare a notification dispatcher to process the registration
42      // notification concurrently. 11/04/2014, Bing Li
43      private NotificationDispatcher<RegisterClientNotification, RegisterClientThread,
44          RegisterClientThreadCreator> registerClientNotificationDispatcher;
45      // Declare a request dispatcher to respond users sign-up requests
46      // concurrently. 11/04/2014, Bing Li
47      private RequestDispatcher<SignUpRequest, SignUpStream, SignUpResponse,
48          SignUpThread, SignUpThreadCreator> signUpRequestDispatcher;
49      // Declare a notification dispatcher to set the value of Weather when an instance of
50      // WeatherNotification is received. 02/15/2016, Bing Li
51      private NotificationDispatcher<WeatherNotification, SetWeatherThread,
52          SetWeatherThreadCreator> setWeatherNotificationDispatcher;
53      // Declare a request dispatcher to respond an instance of WeatherResponse to
54      // the relevant remote client when an instance of WeatherReques is received. 02/15/2016, Bing Li
55      private RequestDispatcher<WeatherRequest, WeatherStream, WeatherResponse,
56          WeatherThread, WeatherThreadCreator> weatherRequestDispatcher;
57      // Declare a notification dispatcher to deal with instances of InitReadNotification from
58      // a client concurrently such that the client can initialize its
59      // ObjectInputStream. 11/09/2014, Bing Li
60      private NotificationDispatcher<InitReadNotification, InitReadFeedbackThread,
61          InitReadFeedbackThreadCreator> initReadFeedbackNotificationDispatcher;
62      // Declare a notification dispatcher to shutdown the server when such a
63      // notification is received. 02/15/2016, Bing Li
64      private NotificationDispatcher<ShutdownServerNotification, ShutdownThread,
65          ShutdownThreadCreator> shutdownNotificationDispatcher;
66
67      /*
68      * Initialize. 09/20/2014, Bing Li
69      */
70      public MyServerDispatcher(int corePoolSize, long keepAliveTime)
71      {
72          // Set the pool size and threads' alive time. 11/04/2014, Bing Li
73          super(corePoolSize, keepAliveTime);
74
75          // Initialize the client registration notification dispatcher. 11/30/2014, Bing Li
```

```
76          this.registerClientNotificationDispatcher = new
77              NotificationDispatcher<RegisterClientNotification, RegisterClientThread,
78              RegisterClientThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
79              ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
80              new RegisterClientThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
81              ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
82              ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
83              ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
84              ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
85              ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
86              Scheduler.GREATFREE().getSchedulerPool());
87
88          // Initialize the sign up request dispatcher. 11/04/2014, Bing Li
89          this.signUpRequestDispatcher = new RequestDispatcher<SignUpRequest, SignUpStream,
90              SignUpResponse, SignUpThread, SignUpThreadCreator>
91              (ServerConfig.REQUEST_DISPATCHER_POOL_SIZE,
92              ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME,
93              new SignUpThreadCreator(), ServerConfig.MAX_REQUEST_TASK_SIZE,
94              ServerConfig.MAX_REQUEST_THREAD_SIZE,
95              ServerConfig.REQUEST_DISPATCHER_WAIT_TIME,
96              ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND,
97              ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY,
98              ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD,
99              Scheduler.GREATFREE().getSchedulerPool());
100
101         // Initialize the weather notification dispatcher. 02/15/2016, Bing Li
102         this.setWeatherNotificationDispatcher = new NotificationDispatcher<WeatherNotification,
103             SetWeatherThread, SetWeatherThreadCreator>
104             (ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
105             ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
106             new SetWeatherThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
107             ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
108             ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
109             ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
110             ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
111             ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
112             Scheduler.GREATFREE().getSchedulerPool());
113
114         // Initialize the sign up request dispatcher. 11/04/2014, Bing Li
115         this.weatherRequestDispatcher = new RequestDispatcher<WeatherRequest, WeatherStream,
116             WeatherResponse, WeatherThread, WeatherThreadCreator>
117             (ServerConfig.REQUEST_DISPATCHER_POOL_SIZE,
118             ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME,
119             new WeatherThreadCreator(), ServerConfig.MAX_REQUEST_TASK_SIZE,
120             ServerConfig.MAX_REQUEST_THREAD_SIZE,
121             ServerConfig.REQUEST_DISPATCHER_WAIT_TIME,
122             ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND,
123             ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY,
124             ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD,
125             Scheduler.GREATFREE().getSchedulerPool());
126
127         // Initialize the read initialization notification dispatcher. 11/30/2014, Bing Li
128         this.initReadFeedbackNotificationDispatcher = new NotificationDispatcher<InitReadNotification,
129             InitReadFeedbackThread, InitReadFeedbackThreadCreator>
130             (ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
131             ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
132             new InitReadFeedbackThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
133             ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
134             ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
135             ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
136             ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
137             ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
138             Scheduler.GREATFREE().getSchedulerPool());
139
140         // Initialize the shutdown notification dispatcher. 11/30/2014, Bing Li
141         this.shutdownNotificationDispatcher = new NotificationDispatcher<ShutdownServerNotification,
142             ShutdownThread, ShutdownThreadCreator>
143             (ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
144             ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
145             new ShutdownThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
146             ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
147             ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
148             ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
149             ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
150             ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
151             Scheduler.GREATFREE().getSchedulerPool());
152     }
```

```
153
154         /*
155         * Shut down the server message dispatcher. 09/20/2014, Bing Li
156         */
157         public void shutdown() throws InterruptedException
158         {
159             // Dispose the register dispatcher. 01/14/2016, Bing Li
160             this.registerClientNotificationDispatcher.dispose();
161             // Dispose the sign-up dispatcher. 11/04/2014, Bing Li
162             this.signUpRequestDispatcher.dispose();
163             // Dispose the weather notification dispatcher. 02/15/2016, Bing Li
164             this.setWeatherNotificationDispatcher.dispose();
165             // Dispose the weather request dispatcher. 02/15/2016, Bing Li
166             this.weatherRequestDispatcher.dispose();
167             // Dispose the dispatcher for initializing reading feedback. 11/09/2014, Bing Li
168             this.initReadFeedbackNotificationDispatcher.dispose();
169             // Dispose the dispatcher for shutdown. 11/09/2014, Bing Li
170             this.shutdownNotificationDispatcher.dispose();
171             // Shutdown the derived server dispatcher. 11/04/2014, Bing Li
172             super.shutdown();
173         }
174
175         /*
176         * Process the available messages in a concurrent way. 09/20/2014, Bing Li
177         */
178         public void consume(OutMessageStream<ServerMessage> message)
179         {
180             // Check the types of received messages. 11/09/2014, Bing Li
181             switch (message.getMessage().getType())
182             {
183                 case MessageType.REGISTER_CLIENT_NOTIFICATION:
184                     System.out.println("REGISTER_CLIENT_NOTIFICATION received @" +
185                         Calendar.getInstance().getTime());
186                     // Check whether the registry notification dispatcher is ready. 01/14/2016, Bing Li
187                     if (!this.registerClientNotificationDispatcher.isReady())
188                     {
189                         // Execute the notification dispatcher as a thread. 01/14/2016, Bing Li
190                         super.execute(this.registerClientNotificationDispatcher);
191                     }
192                     // Enqueue the notification into the dispatcher for concurrent processing. 01/14/2016, Bing Li
193                     this.registerClientNotificationDispatcher.enqueue((RegisterClientNotification)
194                         message.getMessage());
195                     break;
196
197                 // If the message is the one of sign-up requests. 11/09/2014, Bing Li
198                 case MessageType.SIGN_UP_REQUEST:
199                     System.out.println("SIGN_UP_REQUEST received @" + Calendar.getInstance().getTime());
200                     // Check whether the sign-up dispatcher is ready. 01/14/2016, Bing Li
201                     if (!this.signUpRequestDispatcher.isReady())
202                     {
203                         // Execute the sign-up dispatcher as a thread. 01/14/2016, Bing Li
204                         super.execute(this.signUpRequestDispatcher);
205                     }
206                     // Enqueue the request into the dispatcher for concurrent responding. 11/09/2014, Bing Li
207                     this.signUpRequestDispatcher.enqueue(new SignUpStream(message.getOutStream(),
208                         message.getLock(), (SignUpRequest)message.getMessage()));
209                     break;
210
211                 // If the message is the one of WeatherNotification. 02/15/2016, Bing Li
212                 case MessageType.WEATHER_NOTIFICATION:
213                     System.out.println("WEATHER_NOTIFICATION received @" +
214                         Calendar.getInstance().getTime());
215                     // Check whether the weather notification dispatcher is ready or not. 02/15/2016, Bing Li
216                     if (!this.setWeatherNotificationDispatcher.isReady())
217                     {
218                         // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
219                         super.execute(this.setWeatherNotificationDispatcher);
220                     }
221                     // Enqueue the instance of WeatherNotification into the dispatcher for concurrent
222                     // processing. 02/15/2016, Bing Li
223                     this.setWeatherNotificationDispatcher.enqueue((WeatherNotification)message.getMessage());
224                     break;
225
226                 // If the message is the one of weather requests. 11/09/2014, Bing Li
227                 case MessageType.WEATHER_REQUEST:
228                     System.out.println("WEATHER_REQUEST received @" + Calendar.getInstance().getTime());
229                     // Check whether the weather request dispatcher is ready. 02/15/2016, Bing Li
```

```
230                 if (!this.weatherRequestDispatcher.isReady())
231                 {
232                     // Execute the weather request dispatcher concurrently. 02/15/2016, Bing Li
233                     super.execute(this.weatherRequestDispatcher);
234                 }
235                 // Enqueue the instance of WeatherRequest into the dispatcher for concurrent
236                 // responding. 02/15/2016, Bing Li
237                 this.weatherRequestDispatcher.enqueue(new WeatherStream(message.getOutStream(),
238                     message.getLock(), (WeatherRequest)message.getMessage()));
239                 break;
240
241             // If the message is the one of initializing notification. 11/09/2014, Bing Li
242             case SystemMessageType.INIT_READ_NOTIFICATION:
243                 System.out.println("INIT_READ_NOTIFICATION received @" +
244                     Calendar.getInstance().getTime());
245                 // Check whether the reading initialization dispatcher is ready or not. 01/14/2016, Bing Li
246                 if (!this.initReadFeedbackNotificationDispatcher.isReady())
247                 {
248                     // Execute the notification dispatcher as a thread. 01/14/2016, Bing Li
249                     super.execute(this.initReadFeedbackNotificationDispatcher);
250                 }
251                 // Enqueue the notification into the dispatcher for concurrent processing. 11/09/2014, Bing Li
252                 this.initReadFeedbackNotificationDispatcher.enqueue((InitReadNotification)
253                     message.getMessage());
254                 break;
255
256             case MessageType.SHUTDOWN_REGULAR_SERVER_NOTIFICATION:
257                 System.out.println("SHUTDOWN_REGULAR_SERVER_NOTIFICATION received @"
258                     + Calendar.getInstance().getTime());
259                 // Check whether the shutdown dispatcher is ready or not. 01/14/2016, Bing Li
260                 if (!this.shutdownNotificationDispatcher.isReady())
261                 {
262                     // Execute the notification dispatcher as a thread. 01/14/2016, Bing Li
263                     super.execute(this.shutdownNotificationDispatcher);
264                 }
265                 // Enqueue the notification into the dispatcher for concurrent processing. 11/09/2014, Bing Li
266                 this.shutdownNotificationDispatcher.enqueue((ShutdownServerNotification)
267                     message.getMessage());
268                 break;
269         }
270     }
271 }
```

List 3.8 The code of MyServerDispatcher.java

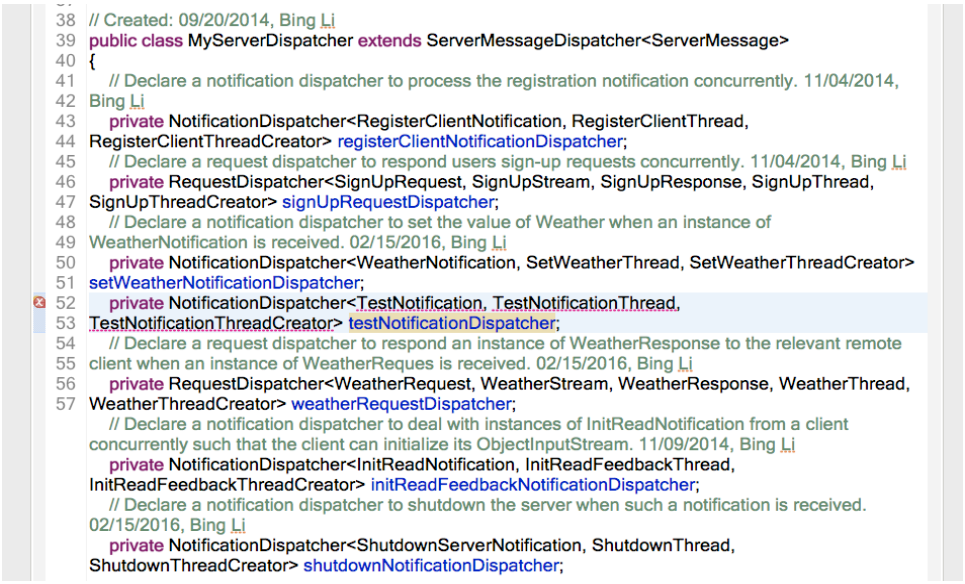
```
38 // Created: 09/20/2014, Bing Li
39 public class MyServerDispatcher extends ServerMessageDispatcher<ServerMessage>
40 {
41     // Declare a notification dispatcher to process the registration notification concurrently. 11/04/2014,
42 Bing Li
43     private NotificationDispatcher<RegisterClientNotification, RegisterClientThread,
44 RegisterClientThreadCreator> registerClientNotificationDispatcher;
45     // Declare a request dispatcher to respond users sign-up requests concurrently. 11/04/2014, Bing Li
46     private RequestDispatcher<SignUpRequest, SignUpStream, SignUpResponse, SignUpThread,
47 SignUpThreadCreator> signUpRequestDispatcher;
48     // Declare a notification dispatcher to set the value of Weather when an instance of
49 WeatherNotification is received. 02/15/2016, Bing Li
50     private NotificationDispatcher<WeatherNotification, SetWeatherThread, SetWeatherThreadCreator>
51 setWeatherNotificationDispatcher;
52     // Declare a request dispatcher to respond an instance of WeatherResponse to the relevant remote
53 client when an instance of WeatherReques is received. 02/15/2016, Bing Li
54     private RequestDispatcher<WeatherRequest, WeatherStream, WeatherResponse, WeatherThread,
55 WeatherThreadCreator> weatherRequestDispatcher;
56     // Declare a notification dispatcher to deal with instances of InitReadNotification from a client
57 concurrently such that the client can initialize its ObjectInputStream. 11/09/2014, Bing Li
58     private NotificationDispatcher<InitReadNotification, InitReadFeedbackThread,
59 InitReadFeedbackThreadCreator> initReadFeedbackNotificationDispatcher;
60     // Declare a notification dispatcher to shutdown the server when such a notification is received.
61 02/15/2016, Bing Li
    private NotificationDispatcher<ShutdownServerNotification, ShutdownThread,
ShutdownThreadCreator> shutdownNotificationDispatcher;

    /*
```

Figure 3.33 One segment of the code of MyServerDispatcher.java

That is a long code, right? Do not worry because you are programming with GreatFree. If you care about implementing your requirements only, you do not need to spend much time on the code. If you are interested in updating the underlying code, you can further open each class to investigate it.

```
38 // Created: 09/20/2014, Bing Li
39 public class MyServerDispatcher extends ServerMessageDispatcher<ServerMessage>
40 {
41    // Declare a notification dispatcher to process the registration notification concurrently. 11/04/2014,
42 Bing Li
43    private NotificationDispatcher<RegisterClientNotification, RegisterClientThread,
44 RegisterClientThreadCreator> registerClientNotificationDispatcher;
45    // Declare a request dispatcher to respond users sign-up requests concurrently. 11/04/2014, Bing Li
46    private RequestDispatcher<SignUpRequest, SignUpStream, SignUpResponse, SignUpThread,
47 SignUpThreadCreator> signUpRequestDispatcher;
48    // Declare a notification dispatcher to set the value of Weather when an instance of
49 WeatherNotification is received. 02/15/2016, Bing Li
50    private NotificationDispatcher<WeatherNotification, SetWeatherThread, SetWeatherThreadCreator>
51 setWeatherNotificationDispatcher;
52    private NotificationDispatcher<TestNotification, TestNotificationThread,
53 TestNotificationThreadCreator> setWeatherNotificationDispatcher;
54    // Declare a request dispatcher to respond an instance of WeatherResponse to the relevant remote
55 client when an instance of WeatherReques is received. 02/15/2016, Bing Li
56    private RequestDispatcher<WeatherRequest, WeatherStream, WeatherResponse, WeatherThread,
57 WeatherThreadCreator> weatherRequestDispatcher;
58    // Declare a notification dispatcher to deal with instances of InitReadNotification from a client
59 concurrently such that the client can initialize its ObjectInputStream. 11/09/2014, Bing Li
60    private NotificationDispatcher<InitReadNotification, InitReadFeedbackThread,
61 InitReadFeedbackThreadCreator> initReadFeedbackNotificationDispatcher;
62    // Declare a notification dispatcher to shutdown the server when such a notification is received.
   02/15/2016, Bing Li
   private NotificationDispatcher<ShutdownServerNotification, ShutdownThread,
   ShutdownThreadCreator> shutdownNotificationDispatcher;
```

Figure 3.34 One segment of the code of MyServerDispatcher.java after copying-pasting-replacing Line 50



```
38 // Created: 09/20/2014, Bing Li
39 public class MyServerDispatcher extends ServerMessageDispatcher<ServerMessage>
40 {
41    // Declare a notification dispatcher to process the registration notification concurrently. 11/04/2014,
42 Bing Li
43    private NotificationDispatcher<RegisterClientNotification, RegisterClientThread,
44 RegisterClientThreadCreator> registerClientNotificationDispatcher;
45    // Declare a request dispatcher to respond users sign-up requests concurrently. 11/04/2014, Bing Li
46    private RequestDispatcher<SignUpRequest, SignUpStream, SignUpResponse, SignUpThread,
47 SignUpThreadCreator> signUpRequestDispatcher;
48    // Declare a notification dispatcher to set the value of Weather when an instance of
49 WeatherNotification is received. 02/15/2016, Bing Li
50    private NotificationDispatcher<WeatherNotification, SetWeatherThread, SetWeatherThreadCreator>
51 setWeatherNotificationDispatcher;
52    private NotificationDispatcher<TestNotification, TestNotificationThread,
53 TestNotificationThreadCreator> testNotificationDispatcher;
54    // Declare a request dispatcher to respond an instance of WeatherResponse to the relevant remote
55 client when an instance of WeatherReques is received. 02/15/2016, Bing Li
56    private RequestDispatcher<WeatherRequest, WeatherStream, WeatherResponse, WeatherThread,
57 WeatherThreadCreator> weatherRequestDispatcher;
   // Declare a notification dispatcher to deal with instances of InitReadNotification from a client
   concurrently such that the client can initialize its ObjectInputStream. 11/09/2014, Bing Li
   private NotificationDispatcher<InitReadNotification, InitReadFeedbackThread,
   InitReadFeedbackThreadCreator> initReadFeedbackNotificationDispatcher;
   // Declare a notification dispatcher to shutdown the server when such a notification is received.
   02/15/2016, Bing Li
   private NotificationDispatcher<ShutdownServerNotification, ShutdownThread,
   ShutdownThreadCreator> shutdownNotificationDispatcher;
```

Figure 3.35 One segment of the code of MyServerDispatcher.java after renaming

You must know what we should do next. Yes, we need to find the samples or the counterparts of TestNotification. Since the lengths of the lines in List 3.8 are too long to fit the width of the book, the lines have to be split into multiple ones. And the word-wrap is enabled in Eclipse such that the line numbers in the book do not match with the ones in my Eclipse. When programming step by step, just follow the figures since they are the screenshots. Thus, according to Figure 3.33, instead of List 3.8, in Line 50, the three classes we are familiar with emerge there initially in the code. To

complete our programming, make a copy of Line 50 and paste it anywhere in the attribute definition area of the class of MyServerDispatcher. To be clear, it is pasted immediately after Line 50. After that, you are required to replace the counterparts following Table 3.2. Then, the code of MyServerDispatcher.java is updated like the one shown in Figure 3.34.

Figure 3.34 shows two compilation errors in Line 50 and Line 51, respectively. It happens because two attributes of the class, MyServerDispatcher, have the same name or their names conflict. To solve the problem, you need to rename the second one, which you just copied. For instance, I rename the second one as testNotificationDispatcher. After renaming, the editor is updated as the one shown in Figure 3.35.

After renaming, although the previous error is fixed, new errors are caused. But this time, you must get used to them. You can select the options prompted by Eclipse to fix them. Now the code of MyServerDispatcher.java should be exactly like the one shown in Figure 3.36. Actually, the errors are fixed by importing the class, TestNotification.

```
39 // Created: 09/20/2014, Bing Li
40 public class MyServerDispatcher extends ServerMessageDispatcher<ServerMessage>
41 {
42    // Declare a notification dispatcher to process the registration notification concurrently. 11/04/2014,
43 Bing Li
44    private NotificationDispatcher<RegisterClientNotification, RegisterClientThread,
45 RegisterClientThreadCreator> registerClientNotificationDispatcher;
46    // Declare a request dispatcher to respond users sign-up requests concurrently. 11/04/2014, Bing Li
47    private RequestDispatcher<SignUpRequest, SignUpStream, SignUpResponse, SignUpThread,
48 SignUpThreadCreator> signUpRequestDispatcher;
49    // Declare a notification dispatcher to set the value of Weather when an instance of
50 WeatherNotification is received. 02/15/2016, Bing Li
51    private NotificationDispatcher<WeatherNotification, SetWeatherThread, SetWeatherThreadCreator>
52 setWeatherNotificationDispatcher;
53    private NotificationDispatcher<TestNotification, TestNotificationThread,
54 TestNotificationThreadCreator> testNotificationDispatcher;
55    // Declare a request dispatcher to respond an instance of WeatherResponse to the relevant remote
56 client when an instance of WeatherReques is received. 02/15/2016, Bing Li
57    private RequestDispatcher<WeatherRequest, WeatherStream, WeatherResponse, WeatherThread,
58 WeatherThreadCreator> weatherRequestDispatcher;
59    // Declare a notification dispatcher to deal with instances of InitReadNotification from a client
60 concurrently such that the client can initialize its ObjectInputStream. 11/09/2014, Bing Li
61    private NotificationDispatcher<InitReadNotification, InitReadFeedbackThread,
62 InitReadFeedbackThreadCreator> initReadFeedbackNotificationDispatcher;
63    // Declare a notification dispatcher to shutdown the server when such a notification is received.
   02/15/2016, Bing Li
   private NotificationDispatcher<ShutdownServerNotification, ShutdownThread,
ShutdownThreadCreator> shutdownNotificationDispatcher;
```

Figure 3.36 One segment of the code of MyServerDispatcher.java after importing TestNotification

3.5 Initializing the Notification Dispatcher

Now the code of MyServerDispatcher looks much better except one warning in Line 53 as shown in Figure 3.36. It cannot be fixed by the prompt of Eclipse since the warning tells you that the attribute, testNotificationDispatcher, i.e., a notification dispatcher, has not been initialized and exploited. We will explain the term, the notification dispatcher, in later chapters.

To initialize the new attribute, testNotificationDispatcher, of MyServerDispatcher, let us move into the constructor of MyServerDispatcher, as shown in Figure 3.37.

According to our experiences, since testNotificationDispatcher is mimicked from setWeatherNotificationDispatcher by CPR, we can also follow the approach to complete the other code. Therefore, although there are many lines of code inside the constructor, the only portions we need to take care of are the lines between Line 69 and Line 79 as shown in Figure 3.38. Actually, that is just one line that is much longer than the width of the current editor of Eclipse. Because of the function of word-wrap, it is turned into multiple ones.

```
57  /*
58   * Initialize. 09/20/2014, Bing Li
59   */
60  public MyServerDispatcher(int corePoolSize, long keepAliveTime)
61  {
62     // Set the pool size and threads' alive time. 11/04/2014, Bing Li
63     super(corePoolSize, keepAliveTime);
64
65     // Initialize the client registration notification dispatcher. 11/30/2014, Bing Li
66     this.registerClientNotificationDispatcher = new NotificationDispatcher<RegisterClientNotification,
67 RegisterClientThread,
68 RegisterClientThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
69 ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME, new
70 RegisterClientThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
71 ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
72 ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
    ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
    ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
    ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
    Scheduler.GREATFREE().getSchedulerPool());

        // Initialize the sign up request dispatcher. 11/04/2014, Bing Li
        this.signUpRequestDispatcher = new RequestDispatcher<SignUpRequest, SignUpStream,
    SignUpResponse, SignUpThread,
    SignUpThreadCreator>(ServerConfig.REQUEST_DISPATCHER_POOL_SIZE,
    ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME, new SignUpThreadCreator(),
```

Figure 3.37 The constructor of MyServerDispatcher

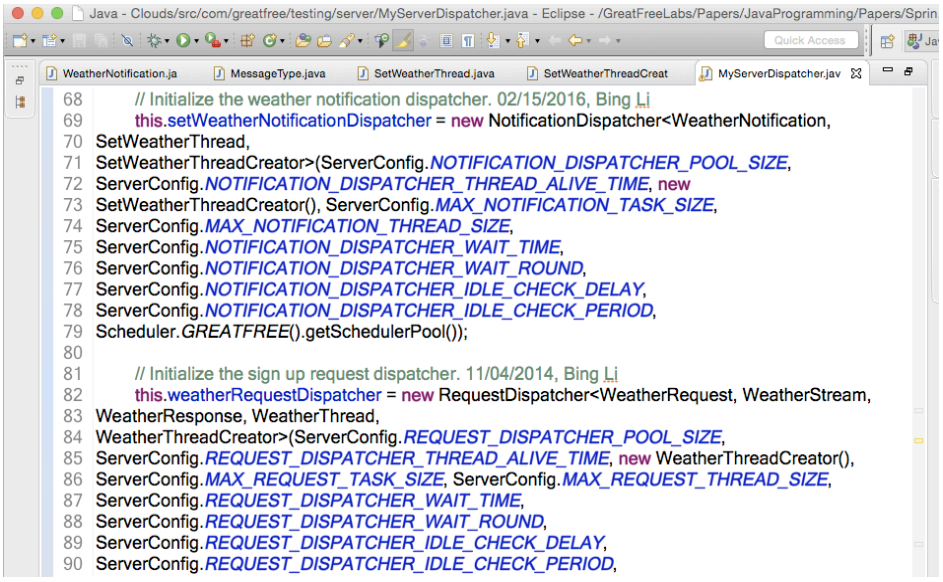


Figure 3.38 The initialization for setWeatherNotificationDispatcher, i.e., the lines between Line 69 and Line 79

Between Line 69 and Line 79, setWeatherNotificationDispatcher is initialized. With its constructor, besides eight constants and one singleton, Scheduler.GREATFREE().getSchedulerPool(), the only one you are familiar with is SetWeatherThreadCreator, which is shown in List 3.6. Its instance is the third parameter of the constructor. Following our previous rules, you do not need to take care about the ones that are related to WeatherNotification as shown in Table 3.2.

Using CPR, copy the lines between Line 69 and Line 79 and paste them immediately after Line 80. Then, the editor is updated as shown in Figure 3.39.

Figure 3.39 Copy and paste the sample notification dispatcher, setWeatherNotificationDispatcher

Next, to continue the initialization, you need to replace setWeatherNotificationDispatcher with testNotificationDispatcher, as shown in Figure 3.40. After the replacement, a compilation error is caused in Line 81. It is easy to know the reason according to the previous experiences. To correct the error, you need to continue to replace according to Table 3.2. In Line 81, each of the classes for WeatherNotification must be replaced by its counterpart for TestNotification. After the step, the initialization for testNotificationDispatcher is done, as shown in Figure 3.41.

Figure 3.40 Replace setWeatherNotificationDispatcher with testNotificationDispatcher

Figure 3.41 Replace each class for WeatherNotification with its counterpart for TestNotification

3.6 Shutting Down the Notification Dispatcher

After the initialization for testNotificationDispatcher is done, the programming for it is still not finished. To know what to do for that, you can search its counterpart, setWeatherNotificationDispatcher with MyServerDispatcher.java. Besides the initialization in Line 69, it emerges in Line 112 in Figure 3.42 and Line 145, Line 149 and Line 154 in Figure 3.43. That means you must CPR all of them before finishing programming a notification.

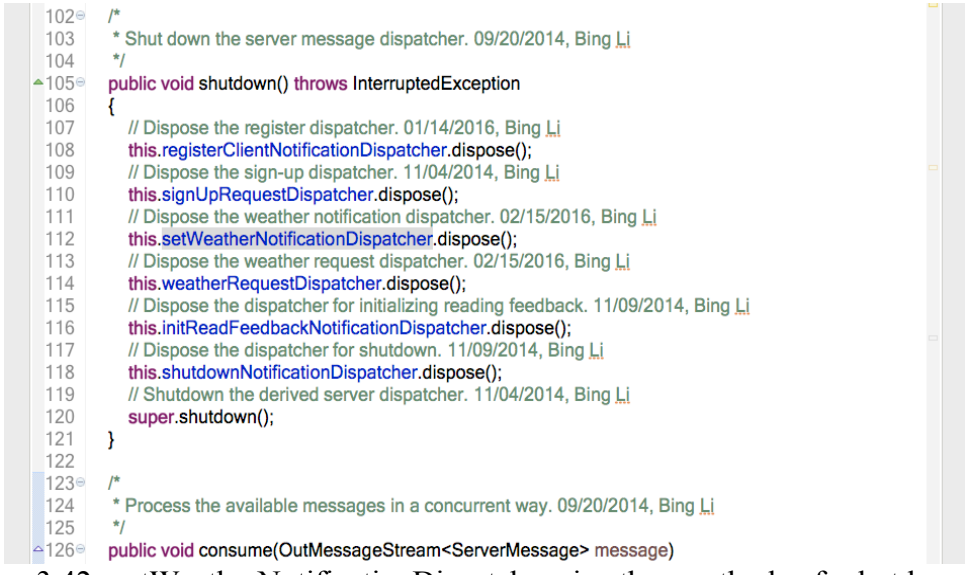


Figure 3.42 setWeatherNotificationDispatcher in the method of shutdown() in MyServerDispatcher

```
139
140         // If the message is the one of WeatherNotification. 02/15/2016, Bing Li
141         case MessageType.WEATHER_NOTIFICATION:
142            System.out.println("WEATHER_NOTIFICATION received @" +
143 Calendar.getInstance().getTime());
144            // Check whether the weather notification dispatcher is ready or not. 02/15/2016, Bing Li
145            if (!this.setWeatherNotificationDispatcher.isReady())
146            {
147 //             System.out.println("Raise setWeatherNotificationDispatcher");
148               // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
149               super.execute(this.setWeatherNotificationDispatcher);
150            }
151            // Enqueue the instance of WeatherNotification into the dispatcher for concurrent processing.
152 02/15/2016, Bing Li
153
154 this.setWeatherNotificationDispatcher.enqueue((WeatherNotification)message.getMessage());
155            break;
156
157            // If the message is the one of weather requests. 11/09/2014, Bing Li
158         case MessageType.WEATHER_REQUEST:
159            System.out.println("WEATHER_REQUEST received @" + Calendar.getInstance().getTime());
160            // Check whether the weather request dispatcher is ready. 02/15/2016, Bing Li
161            if (!this.weatherRequestDispatcher.isReady())
162            {
163 //             System.out.println("Raise weatherRequestDispatcher");
164               // Execute the weather request dispatcher concurrently. 02/15/2016, Bing Li
```

Figure 3.43 setWeatherNotificationDispatcher in the method of consume(OutMessageStream<ServerMessage> message) in MyServerDispatcher

Figure 3.42 illustrates that all of attributes of MyServerDispatcher must be disposed when the server is shut down. So must be testNotificationDispatcher. For that, CPR it inside the method of shutdown(). Then, disposing testNotificationDispatcher is done, as shown in Figure 3.44.

```
92  /*
93   * Shut down the server message dispatcher. 09/20/2014, Bing Li
94   */
95  public void shutdown() throws InterruptedException
96  {
97     // Dispose the register dispatcher. 01/14/2016, Bing Li
98     this.registerClientNotificationDispatcher.dispose();
99     // Dispose the sign-up dispatcher. 11/04/2014, Bing Li
100    this.signUpRequestDispatcher.dispose();
101    // Dispose the weather notification dispatcher. 02/15/2016, Bing Li
102    this.setWeatherNotificationDispatcher.dispose();
103    this.testNotificationDispatcher.dispose();
104    // Dispose the weather request dispatcher. 02/15/2016, Bing Li
105    this.weatherRequestDispatcher.dispose();
106    // Dispose the dispatcher for initializing reading feedback. 11/09/2014, Bing Li
107    this.initReadFeedbackNotificationDispatcher.dispose();
108    // Dispose the dispatcher for shutdown. 11/09/2014, Bing Li
109    this.shutdownNotificationDispatcher.dispose();
110    // Shutdown the derived server dispatcher. 11/04/2014, Bing Li
111    super.shutdown();
112 }
113
114 /*
115  * Process the available messages in a concurrent way. 09/20/2014, Bing Li
116  */
117 public void consume(OutMessageStream<ServerMessage> message)
```

Figure 3.44 CPR testNotificationDispatcher in the method of shutdown() in MyServerDispatcher

3.7 Exploiting the Notification Dispatcher

The final step to program TestNotification is to exploit its references in Table 3.2 in the method of consume(OutMessageStream<ServerMessage> message). At this moment, you must have already grasped the skills how to do that.

First of all, we need to copy something. Which lines should be copied? At least the lines that include setWeatherNotificationDispatcher must be copied. Besides them, we should be aware of a new pattern, the Message Dispatcher (MD), emerges here. In the pattern, there are multiple case-switch statements and each of them is responsible for dispatching one particular type of messages to its relevant dispatcher. For our sample to be copied, the case-switch lines are located between Line 142 and Line 156. They can be identified easily by the message type integer constant, MessageType.WEATHER_NOTIFICATION, which represents the message, WeatherNotification according to Table 3.2. After copying and pasting the lines, the code is updated as shown in Figure 3.45.

```
138
139         // If the message is the one of WeatherNotification. 02/15/2016, Bing Li
140         case MessageType.WEATHER_NOTIFICATION:
141             System.out.println("WEATHER_NOTIFICATION received @" +
142 Calendar.getInstance().getTime());
143             // Check whether the weather notification dispatcher is ready or not. 02/15/2016, Bing Li
144             if (!this.setWeatherNotificationDispatcher.isReady())
145             {
146 //              System.out.println("Raise setWeatherNotificationDispatcher");
147                 // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
148                 super.execute(this.setWeatherNotificationDispatcher);
149             }
150             // Enqueue the instance of WeatherNotification into the dispatcher for concurrent processing.
151 02/15/2016, Bing Li
152
153 this.setWeatherNotificationDispatcher.enqueue((WeatherNotification)message.getMessage());
154             break;
155
156         case MessageType.WEATHER_NOTIFICATION:
157             System.out.println("WEATHER_NOTIFICATION received @" +
158 Calendar.getInstance().getTime());
159             // Check whether the weather notification dispatcher is ready or not. 02/15/2016, Bing Li
160             if (!this.setWeatherNotificationDispatcher.isReady())
161             {
162 //              System.out.println("Raise setWeatherNotificationDispatcher");
163                 // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
164                 super.execute(this.setWeatherNotificationDispatcher);
165             }
166             // Enqueue the instance of WeatherNotification into the dispatcher for concurrent processing.
167 02/15/2016, Bing Li
168
169 this.setWeatherNotificationDispatcher.enqueue((WeatherNotification)message.getMessage());
170             break;
171
```

Figure 3.45 Copy and paste the lines of dispatching the message of WeatherNotification

After copying and pasting the lines, replacing is expected. It is not difficult to accomplish the task if you follow Table 3.2 as well. Each class for WeatherNotification, including the message ID, WEATHER_NOTIFICATION, needs to be replaced by its counterpart respectively between Line 156 and Line 169 in the pattern of MD. One line, Line 157, is a little bit special since it just prints something within a terminal when a notification is received. Since it also includes the string, WEATHER_NOTIFICATION, it is better to replace it with TEST_NOTIFICATION to avoid misleading when the program is executed. There are some comments for the message of WeatherNotification. You can remove them or write something new related to TestNotification. It depends on you. For other added lines, you can decide whether it is necessary to write comments or not.

After CPR, all of the compilation errors and warnings should be gone as shown in Figure 3.46. Your code should be exactly like the one shown in List 3.9.

```
139
140         // If the message is the one of WeatherNotification. 02/15/2016, Bing Li
141         case MessageType.WEATHER_NOTIFICATION:
142             System.out.println("WEATHER_NOTIFICATION received @" +
143 Calendar.getInstance().getTime());
144             // Check whether the weather notification dispatcher is ready or not. 02/15/2016, Bing Li
145             if (!this.setWeatherNotificationDispatcher.isReady())
146             {
147 //              System.out.println("Raise setWeatherNotificationDispatcher");
148                 // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
149                 super.execute(this.setWeatherNotificationDispatcher);
150             }
151             // Enqueue the instance of WeatherNotification into the dispatcher for concurrent processing.
152 02/15/2016, Bing Li
153
154 this.setWeatherNotificationDispatcher.enqueue((WeatherNotification)message.getMessage());
155             break;
156
157         case MessageType.TEST_NOTIFICATION:
158             System.out.println("TEST_NOTIFICATION received @" + Calendar.getInstance().getTime());
159             // Check whether the test notification dispatcher is ready or not. 02/15/2016, Bing Li
160             if (!this.testNotificationDispatcher.isReady())
161             {
162                 // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
163                 super.execute(this.testNotificationDispatcher);
164             }
165             // Enqueue the instance of TestNotification into the dispatcher for concurrent processing.
166 02/15/2016, Bing Li
167             this.testNotificationDispatcher.enqueue((TestNotification)message.getMessage());
168             break;
169
170             // If the message is the one of weather requests. 11/09/2014, Bing Li
```

Figure 3.46 Replacing happens between the lines in the pattern of MD

```
1   package com.greatfree.testing.server;
2
3   import java.util.Calendar;
4
5   import com.greatfree.concurrency.NotificationDispatcher;
6   import com.greatfree.concurrency.RequestDispatcher;
7   import com.greatfree.concurrency.Scheduler;
8   import com.greatfree.concurrency.ServerMessageDispatcher;
9   import com.greatfree.message.InitReadNotification;
10  import com.greatfree.message.SystemMessageType;
11  import com.greatfree.multicast.ServerMessage;
12  import com.greatfree.remote.OutMessageStream;
13  import com.greatfree.testing.data.ServerConfig;
14  import com.greatfree.testing.message.MessageType;
15  import com.greatfree.testing.message.RegisterClientNotification;
16  import com.greatfree.testing.message.ShutdownServerNotification;
17  import com.greatfree.testing.message.SignUpRequest;
18  import com.greatfree.testing.message.SignUpResponse;
19  import com.greatfree.testing.message.SignUpStream;
20  import com.greatfree.testing.message.WeatherNotification;
21  import com.greatfree.testing.message.WeatherRequest;
22  import com.greatfree.testing.message.WeatherResponse;
23  import com.greatfree.testing.message.WeatherStream;
24
25  /*
26  * This is an implementation of ServerMessageDispatcher. It contains the concurrency mechanism to
27  * respond clients' requests and receive clients' notifications for the server. 09/20/2014, Bing Li
28  */
29
30  /*
31  * Revision Log
32  *
33  * The initialization request dispatcher is modified. When no tasks are available for some time, it needs to
34  * be shut down. 01/14/2016, Bing Li
35  *
36  */
37
38  // Created: 09/20/2014, Bing Li
39  public class MyServerDispatcher extends ServerMessageDispatcher<ServerMessage>
40  {
41      // Declare a notification dispatcher to process the registration
42      // notification concurrently. 11/04/2014, Bing Li
43      private NotificationDispatcher<RegisterClientNotification, RegisterClientThread,
44          RegisterClientThreadCreator> registerClientNotificationDispatcher;
45      // Declare a request dispatcher to respond users sign-up requests
```

```
46          // concurrently. 11/04/2014, Bing Li
47          private RequestDispatcher<SignUpRequest, SignUpStream, SignUpResponse,
48              SignUpThread, SignUpThreadCreator> signUpRequestDispatcher;
49          // Declare a notification dispatcher to set the value of Weather when an instance of
50          // WeatherNotification is received. 02/15/2016, Bing Li
51          private NotificationDispatcher<WeatherNotification, SetWeatherThread,
52              SetWeatherThreadCreator> setWeatherNotificationDispatcher;
53          private NotificationDispatcher<TestNotification, TestNotificationThread,
54              TestNotificationThreadCreator> testNotificationDispatcher;
55          // Declare a request dispatcher to respond an instance of WeatherResponse to
56          // the relevant remote client when an instance of WeatherReques is received. 02/15/2016, Bing Li
57          private RequestDispatcher<WeatherRequest, WeatherStream, WeatherResponse,
58              WeatherThread, WeatherThreadCreator> weatherRequestDispatcher;
59          // Declare a notification dispatcher to deal with instances of InitReadNotification from
60          // a client concurrently such that the client can initialize its
61          // ObjectInputStream. 11/09/2014, Bing Li
62          private NotificationDispatcher<InitReadNotification, InitReadFeedbackThread,
63              InitReadFeedbackThreadCreator> initReadFeedbackNotificationDispatcher;
64          // Declare a notification dispatcher to shutdown the server when such a
65          // notification is received. 02/15/2016, Bing Li
66          private NotificationDispatcher<ShutdownServerNotification, ShutdownThread,
67              ShutdownThreadCreator> shutdownNotificationDispatcher;
68
69          /*
70          * Initialize. 09/20/2014, Bing Li
71          */
72          public MyServerDispatcher(int corePoolSize, long keepAliveTime)
73          {
74              // Set the pool size and threads' alive time. 11/04/2014, Bing Li
75              super(corePoolSize, keepAliveTime);
76
77              // Initialize the client registration notification dispatcher. 11/30/2014, Bing Li
78              this.registerClientNotificationDispatcher = new
79                  NotificationDispatcher<RegisterClientNotification, RegisterClientThread,
80                  RegisterClientThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
81                  ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
82                  new RegisterClientThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
83                  ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
84                  ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
85                  ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
86                  ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
87                  ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
88                  Scheduler.GREATFREE().getSchedulerPool());
89
90              // Initialize the sign up request dispatcher. 11/04/2014, Bing Li
91              this.signUpRequestDispatcher = new RequestDispatcher<SignUpRequest, SignUpStream,
92                  SignUpResponse, SignUpThread, SignUpThreadCreator>
93                  (ServerConfig.REQUEST_DISPATCHER_POOL_SIZE,
94                  ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME,
95                  new SignUpThreadCreator(), ServerConfig.MAX_REQUEST_TASK_SIZE,
96                  ServerConfig.MAX_REQUEST_THREAD_SIZE,
97                  ServerConfig.REQUEST_DISPATCHER_WAIT_TIME,
98                  ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND,
99                  ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY,
100                 ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD,
101                 Scheduler.GREATFREE().getSchedulerPool());
102
103             // Initialize the weather notification dispatcher. 02/15/2016, Bing Li
104             this.setWeatherNotificationDispatcher = new NotificationDispatcher<WeatherNotification,
105                 SetWeatherThread, SetWeatherThreadCreator>
106                 (ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
107                 ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
108                 new SetWeatherThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
109                 ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
110                 ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
111                 ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
112                 ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
113                 ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
114                 Scheduler.GREATFREE().getSchedulerPool());
115
116             this.testNotificationDispatcher = new NotificationDispatcher<TestNotification,
117                 TestNotificationThread, TestNotificationThreadCreator>
118                 (ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
119                 ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
120                 new TestNotificationThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
121                 ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
122                 ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
```

```
            ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
            ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
            ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
            Scheduler.GREATFREE().getSchedulerPool());

        // Initialize the sign up request dispatcher. 11/04/2014, Bing Li
        this.weatherRequestDispatcher = new RequestDispatcher<WeatherRequest, WeatherStream,
            WeatherResponse, WeatherThread, WeatherThreadCreator>
            (ServerConfig.REQUEST_DISPATCHER_POOL_SIZE,
            ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME,
            new WeatherThreadCreator(), ServerConfig.MAX_REQUEST_TASK_SIZE,
            ServerConfig.MAX_REQUEST_THREAD_SIZE,
            ServerConfig.REQUEST_DISPATCHER_WAIT_TIME,
            ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND,
            ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY,
            ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD,
            Scheduler.GREATFREE().getSchedulerPool());

        // Initialize the read initialization notification dispatcher. 11/30/2014, Bing Li
        this.initReadFeedbackNotificationDispatcher = new NotificationDispatcher<InitReadNotification,
            InitReadFeedbackThread, InitReadFeedbackThreadCreator>
            (ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
            ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
            new InitReadFeedbackThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
            ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
            ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
            ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
            ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
            ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
            Scheduler.GREATFREE().getSchedulerPool());

        // Initialize the shutdown notification dispatcher. 11/30/2014, Bing Li
        this.shutdownNotificationDispatcher = new NotificationDispatcher<ShutdownServerNotification,
            ShutdownThread, ShutdownThreadCreator>
            (ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
            ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
            new ShutdownThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
            ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
            ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
            ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
            ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
            ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
            Scheduler.GREATFREE().getSchedulerPool());
    }

    /*
    * Shut down the server message dispatcher. 09/20/2014, Bing Li
    */
    public void shutdown() throws InterruptedException
    {
        // Dispose the register dispatcher. 01/14/2016, Bing Li
        this.registerClientNotificationDispatcher.dispose();
        // Dispose the sign-up dispatcher. 11/04/2014, Bing Li
        this.signUpRequestDispatcher.dispose();
        // Dispose the weather notification dispatcher. 02/15/2016, Bing Li
        this.setWeatherNotificationDispatcher.dispose();
        // Dispose the weather request dispatcher. 02/15/2016, Bing Li
        this.weatherRequestDispatcher.dispose();
        this.testNotificationDispatcher.dispose();
        // Dispose the dispatcher for initializing reading feedback. 11/09/2014, Bing Li
        this.initReadFeedbackNotificationDispatcher.dispose();
        // Dispose the dispatcher for shutdown. 11/09/2014, Bing Li
        this.shutdownNotificationDispatcher.dispose();
        // Shutdown the derived server dispatcher. 11/04/2014, Bing Li
        super.shutdown();
    }

    /*
    * Process the available messages in a concurrent way. 09/20/2014, Bing Li
    */
    public void consume(OutMessageStream<ServerMessage> message)
    {
        // Check the types of received messages. 11/09/2014, Bing Li
        switch (message.getMessage().getType())
        {
            case MessageType.REGISTER_CLIENT_NOTIFICATION:
                System.out.println("REGISTER_CLIENT_NOTIFICATION received @" +
```

```
200                     Calendar.getInstance().getTime());
201                 // Check whether the registry notification dispatcher is ready. 01/14/2016, Bing Li
202                 if (!this.registerClientNotificationDispatcher.isReady())
203                 {
204                     // Execute the notification dispatcher as a thread. 01/14/2016, Bing Li
205                     super.execute(this.registerClientNotificationDispatcher);
206                 }
207                 // Enqueue the notification into the dispatcher for concurrent processing. 01/14/2016, Bing Li
208                 this.registerClientNotificationDispatcher.enqueue((RegisterClientNotification)
209                     message.getMessage());
210                 break;
211
212             // If the message is the one of sign-up requests. 11/09/2014, Bing Li
213             case MessageType.SIGN_UP_REQUEST:
214                 System.out.println("SIGN_UP_REQUEST received @" + Calendar.getInstance().getTime());
215                 // Check whether the sign-up dispatcher is ready. 01/14/2016, Bing Li
216                 if (!this.signUpRequestDispatcher.isReady())
217                 {
218                     // Execute the sign-up dispatcher as a thread. 01/14/2016, Bing Li
219                     super.execute(this.signUpRequestDispatcher);
220                 }
221                 // Enqueue the request into the dispatcher for concurrent responding. 11/09/2014, Bing Li
222                 this.signUpRequestDispatcher.enqueue(new SignUpStream(message.getOutStream(),
223                     message.getLock(), (SignUpRequest)message.getMessage()));
224                 break;
225
226             // If the message is the one of WeatherNotification. 02/15/2016, Bing Li
227             case MessageType.WEATHER_NOTIFICATION:
228                 System.out.println("WEATHER_NOTIFICATION received @" +
229                     Calendar.getInstance().getTime());
230                 // Check whether the weather notification dispatcher is ready or not. 02/15/2016, Bing Li
231                 if (!this.setWeatherNotificationDispatcher.isReady())
232                 {
233                     // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
234                     super.execute(this.setWeatherNotificationDispatcher);
235                 }
236                 // Enqueue the instance of WeatherNotification into the dispatcher for concurrent
237                 // processing. 02/15/2016, Bing Li
238                 this.setWeatherNotificationDispatcher.enqueue((WeatherNotification)message.getMessage());
239                 break;
240
241             case MessageType.TEST_NOTIFICATION:
242                 System.out.println("TEST_NOTIFICATION received @" + Calendar.getInstance().getTime());
243                 // Check whether the test notification dispatcher is ready or not. 02/15/2016, Bing Li
244                 if (!this.testNotificationDispatcher.isReady())
245                 {
246                     // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
247                     super.execute(this.testNotificationDispatcher);
248                 }
249                 // Enqueue the instance of TestNotification into the dispatcher for concurrent
250                 // processing. 02/15/2016, Bing Li
251                 this.testNotificationDispatcher.enqueue((TestNotification)message.getMessage());
252                 break;
253
254             // If the message is the one of weather requests. 11/09/2014, Bing Li
255             case MessageType.WEATHER_REQUEST:
256                 System.out.println("WEATHER_REQUEST received @" + Calendar.getInstance().getTime());
257                 // Check whether the weather request dispatcher is ready. 02/15/2016, Bing Li
258                 if (!this.weatherRequestDispatcher.isReady())
259                 {
260                     // Execute the weather request dispatcher concurrently. 02/15/2016, Bing Li
261                     super.execute(this.weatherRequestDispatcher);
262                 }
263                 // Enqueue the instance of WeatherRequest into the dispatcher for concurrent
264                 // responding. 02/15/2016, Bing Li
265                 this.weatherRequestDispatcher.enqueue(new WeatherStream(message.getOutStream(),
266                     message.getLock(), (WeatherRequest)message.getMessage()));
267                 break;
268
269             // If the message is the one of initializing notification. 11/09/2014, Bing Li
270             case SystemMessageType.INIT_READ_NOTIFICATION:
271                 System.out.println("INIT_READ_NOTIFICATION received @" +
272                     Calendar.getInstance().getTime());
273                 // Check whether the reading initialization dispatcher is ready or not. 01/14/2016, Bing Li
274                 if (!this.initReadFeedbackNotificationDispatcher.isReady())
275                 {
276                     // Execute the notification dispatcher as a thread. 01/14/2016, Bing Li
```

```
277                     super.execute(this.initReadFeedbackNotificationDispatcher);
278                 }
279                 // Enqueue the notification into the dispatcher for concurrent processing. 11/09/2014, Bing Li
280                 this.initReadFeedbackNotificationDispatcher.enqueue((InitReadNotification)
281                     message.getMessage());
282                 break;
283
284             case MessageType.SHUTDOWN_REGULAR_SERVER_NOTIFICATION:
285                 System.out.println("SHUTDOWN_REGULAR_SERVER_NOTIFICATION received @"
286                     + Calendar.getInstance().getTime());
287                 // Check whether the shutdown dispatcher is ready or not. 01/14/2016, Bing Li
288                 if (!this.shutdownNotificationDispatcher.isReady())
289                 {
290                     // Execute the notification dispatcher as a thread. 01/14/2016, Bing Li
291                     super.execute(this.shutdownNotificationDispatcher);
292                 }
293                 // Enqueue the notification into the dispatcher for concurrent processing. 11/09/2014, Bing Li
294                 this.shutdownNotificationDispatcher.enqueue((ShutdownServerNotification)
295                     message.getMessage());
296                 break;
297         }
298     }
299 }
```

List 3.9 After CPR, the code of MyServerDispatcher.java

3.8 Summary

Congratulations! The server side is done. Now it is time for us to relax for a little while. You need to recall what you have done in the previous sections. At least, you must agree with me that the overall procedure is smooth or straightforward once if you know the magic of GreatFree.

Originally, programming a distributed server is the most difficult task for any distributed systems because you have to take into account too many issues. If you program the same thing, TestNotification, with a generic programming language like C, C++ or Java SE, it must make you crazy. However, when doing that with GreatFree, your work is lowered to a great extent. Therefore, it is necessary for us to make a summary before moving forward.

The overall procedure can be summarized as follows.

1) Starting with a message and deciding whether it is a notification or a request;

2) Finding a sample from the message package (List 3.1; Section 2.2);

3) Defining the message following the sample message (Figure 3.5~3.14; List 3.2~3.3; Section 2.1~2.2);

4) Finding references for the sample message (Figure 3.15~3.16; Table 3.1; Section 3.1);

5) Listing the counterparts for all of the references of the sample message (Table 3.2; Section 3.1);

6) Creating one of the counterparts, the thread, for your message by CPR;

7) Creating one of the counterparts, the thread creator, for your message by CPR;

8) Updating the server dispatcher by CPR.

4. The Client Side

After the server side is finished, you need to work on the client side. According to Figure 3.16 and Table 3.1, you get the references of WeatherNotification, which is the sample you need to follow.

4.1 Updating the Client Eventer

At this moment, the only one sample you do not refer is the one, ClientEventer.java. Now it is time to open the code by right-clicking the icon in Figure 3.16. The code is shown in List 3.10.

```
1   package com.greatfree.testing.client;
2
3   import java.io.IOException;
4
5   import com.greatfree.concurrency.Scheduler;
6   import com.greatfree.concurrency.ThreadPool;
7   import com.greatfree.remote.AsyncRemoteEventer;
8   import com.greatfree.remote.SyncRemoteEventer;
9   import com.greatfree.testing.data.ClientConfig;
10  import com.greatfree.testing.data.Weather;
11  import com.greatfree.testing.message.ClientForAnycastNotification;
12  import com.greatfree.testing.message.ClientForBroadcastNotification;
13  import com.greatfree.testing.message.ClientForUnicastNotification;
14  import com.greatfree.testing.message.OnlineNotification;
15  import com.greatfree.testing.message.RegisterClientNotification;
16  import com.greatfree.testing.message.UnregisterClientNotification;
17  import com.greatfree.testing.message.WeatherNotification;
18  import com.greatfree.util.NodeID;
19
20  /*
21   * The class is an example that applies SynchRemoteEventer and AsyncRemoteEventer. 11/05/2014, Bing Li
22   */
23
24  // Created: 11/05/2014, Bing Li
25  public class ClientEventer
26  {
27      // Declare the ip of the remote server. 11/07/2014, Bing Li
28      private String ip;
29      // Declare the port of the remote server. 11/07/2014, Bing Li
30      private int port;
31      // The synchronous eventer to send the online notification. 11/07/2014, Bing Li
32      private SyncRemoteEventer<OnlineNotification> onlineEventer;
33      // The synchronous eventer to send the registering notification. 11/07/2014, Bing Li
34      private SyncRemoteEventer<RegisterClientNotification> registerClientEventer;
35      // The synchronous eventer to send the unregistering notification. 11/07/2014, Bing Li
36      private SyncRemoteEventer<UnregisterClientNotification> unregisterClientEventer;
37      // The asynchronous eventer to send one instance of WeatherNotification to the remote server
38      // to set the value of the weather. 02/15/2016, Bing Li
39      private AsyncRemoteEventer<WeatherNotification> weatherEventer;
40
41      // The asynchronous eventer to send one instance of ClientForBroadcastNotification to
42      // the remote server. 02/15/2016, Bing Li
43      private AsyncRemoteEventer<ClientForBroadcastNotification> broadcastEventer;
44      // The asynchronous eventer to send one instance of ClientForUnicastNotification to
45      // the remote server. 02/15/2016, Bing Li
46      private AsyncRemoteEventer<ClientForUnicastNotification> unicastEventer;
47      // The asynchronous eventer to send one instance of ClientForAnycastNotification to
48      // the remote server. 02/15/2016, Bing Li
```

```
49      private AsyncRemoteEventer<ClientForAnycastNotification> anycastEventer;
50
51      // A thread pool to assist sending notification asynchronously. 11/07/2014, Bing Li
52      private ThreadPool pool;
53
54      /*
55       * Initialize. 11/07/2014, Bing Li
56       */
57      private ClientEventer()
58      {
59      }
60
61      /*
62       * A singleton implementation. 11/07/2014, Bing Li
63       */
64      private static ClientEventer instance = new ClientEventer();
65
66      public static ClientEventer NOTIFY()
67      {
68          if (instance == null)
69          {
70              instance = new ClientEventer();
71              return instance;
72          }
73          else
74          {
75              return instance;
76          }
77      }
78
79      /*
80      * Dispose the eventers. 11/07/2014, Bing Li
81      */
82      public void dispose() throws InterruptedException
83      {
84          this.onlineEventer.dispose();
85          this.registerClientEventer.dispose();
86          this.unregisterClientEventer.dispose();
87          this.weatherEventer.dispose();
88
89          this.broadcastEventer.dispose();
90          this.unicastEventer.dispose();
91          this.anycastEventer.dispose();
92
93          // Shutdown the thread pool. 11/07/2014, Bing Li
94          this.pool.shutdown();
95      }
96
97      /*
98      * Initialize the eventers. 11/07/2014, Bing Li
99      */
100     public void init(String ip, int port)
101     {
102         this.ip = ip;
103         this.port = port;
104         this.pool = new ThreadPool(ClientConfig.EVENTER_THREAD_POOL_SIZE,
105             ClientConfig.EVENTER_THREAD_POOL_ALIVE_TIME);
106         // Initialize the synchronous eventer. The FreeClient pool is one parameter for the initialization.
107         // The clients needs to be managed by the pool. 02/15/2016, Bing Li
108         this.onlineEventer = new SyncRemoteEventer<OnlineNotification>(ClientPool.LOCAL().getPool());
109         // Initialize the synchronous eventer. The FreeClient pool is one parameter for the initialization.
110         // The clients needs to be managed by the pool. 02/15/2016, Bing Li
111         this.registerClientEventer = new SyncRemoteEventer<RegisterClientNotification>
112             (ClientPool.LOCAL().getPool());
113         // Initialize the synchronous eventer. The FreeClient pool is one parameter for the initialization.
114         // The clients needs to be managed by the pool. 02/15/2016, Bing Li
115         this.unregisterClientEventer = new SyncRemoteEventer<UnregisterClientNotification>
116             (ClientPool.LOCAL().getPool());
117
118         // Initialize the asynchronous eventer. Since the eventer is sent out to the remote server
119         // asynchronously, more parameters are required to set. 02/15/2016, Bing Li
120         this.weatherEventer = new AsyncRemoteEventer<WeatherNotification>
121             (ClientPool.LOCAL().getPool(), this.pool, ClientConfig.EVENT_QUEUE_SIZE,
122             ClientConfig.EVENTER_SIZE, ClientConfig.EVENTING_WAIT_TIME,
123             ClientConfig.EVENTER_WAIT_TIME, ClientConfig.EVENTER_WAIT_ROUND,
124             ClientConfig.EVENT_IDLE_CHECK_DELAY,
125             ClientConfig.EVENT_IDLE_CHECK_PERIOD,
```

```
126             Scheduler.GREATFREE().getSchedulerPool());
127
128         // Initialize the asynchronous eventer. Since the eventer is sent out to the remote server
129         // asynchronously, more parameters are required to set. 02/15/2016, Bing Li
130         this.broadcastEventer = new AsyncRemoteEventer<ClientForBroadcastNotification>
131             (ClientPool.LOCAL().getPool(), this.pool, ClientConfig.EVENT_QUEUE_SIZE,
132             ClientConfig.EVENTER_SIZE, ClientConfig.EVENTING_WAIT_TIME,
133             ClientConfig.EVENTER_WAIT_TIME, ClientConfig.EVENTER_WAIT_ROUND,
134             ClientConfig.EVENT_IDLE_CHECK_DELAY,
135             ClientConfig.EVENT_IDLE_CHECK_PERIOD,
136             Scheduler.GREATFREE().getSchedulerPool());
137
138         // Initialize the asynchronous eventer. Since the eventer is sent out to the remote server
139         // asynchronously, more parameters are required to set. 02/15/2016, Bing Li
140         this.unicastEventer = new AsyncRemoteEventer<ClientForUnicastNotification>
141             (ClientPool.LOCAL().getPool(), this.pool, ClientConfig.EVENT_QUEUE_SIZE,
142             ClientConfig.EVENTER_SIZE, ClientConfig.EVENTING_WAIT_TIME,
143             ClientConfig.EVENTER_WAIT_TIME, ClientConfig.EVENTER_WAIT_ROUND,
144             ClientConfig.EVENT_IDLE_CHECK_DELAY,
145             ClientConfig.EVENT_IDLE_CHECK_PERIOD,
146             Scheduler.GREATFREE().getSchedulerPool());
147
148         // Initialize the asynchronous eventer. Since the eventer is sent out to the remote server
149         // asynchronously, more parameters are required to set. 02/15/2016, Bing Li
150         this.anycastEventer = new AsyncRemoteEventer<ClientForAnycastNotification>
151             (ClientPool.LOCAL().getPool(), this.pool, ClientConfig.EVENT_QUEUE_SIZE,
152             ClientConfig.EVENTER_SIZE, ClientConfig.EVENTING_WAIT_TIME,
153             ClientConfig.EVENTER_WAIT_TIME, ClientConfig.EVENTER_WAIT_ROUND,
154             ClientConfig.EVENT_IDLE_CHECK_DELAY,
155             ClientConfig.EVENT_IDLE_CHECK_PERIOD,
156             Scheduler.GREATFREE().getSchedulerPool());
157     }
158
159     /*
160      * Send the online notification to the remote server in a synchronous manner. 11/07/2014, Bing Li
161      */
162     public void notifyOnline() throws IOException, InterruptedException
163     {
164         this.onlineEventer.notify(this.ip, this.port, new OnlineNotification());
165     }
166
167     /*
168      * Send the registering notification to the remote server in a synchronous manner. 11/07/2014, Bing Li
169      */
170     public void register()
171     {
172         try
173         {
174             this.registerClientEventer.notify(this.ip, this.port, new
175                 RegisterClientNotification(NodeID.DISTRIBUTED().getKey()));
176         }
177         catch (IOException e)
178         {
179             e.printStackTrace();
180         }
181         catch (InterruptedException e)
182         {
183             e.printStackTrace();
184         }
185     }
186
187     /*
188      * Send the unregistering notification to the remote server in a synchronous manner. 11/07/2014, Bing Li
189      */
190     public void unregister()
191     {
192         try
193         {
194             this.unregisterClientEventer.notify(this.ip, this.port, new
195                 UnregisterClientNotification(NodeID.DISTRIBUTED().getKey()));
196         }
197         catch (IOException e)
198         {
199             e.printStackTrace();
200         }
201         catch (InterruptedException e)
202         {
```

```
203                 e.printStackTrace();
204             }
205         }
206 
207         /*
208          * Send the weather notification to the remote server in an asynchronous manner. 11/07/2014, Bing Li
209          */
210         public void notifyWeather(Weather weather)
211         {
212             if (!this.weatherEventer.isReady())
213             {
214                 this.pool.execute(this.weatherEventer);
215             }
216             this.weatherEventer.notify(this.ip, this.port, new WeatherNotification(weather));
217         }
218 
219         /*
220          * Send the broadcast notification to the remote server in an asynchronous manner. 11/07/2014, Bing Li
221          */
222         public void notifyBroadcastly(String message)
223         {
224             if (!this.broadcastEventer.isReady())
225             {
226                 this.pool.execute(this.broadcastEventer);
227             }
228             this.broadcastEventer.notify(this.ip, this.port, new ClientForBroadcastNotification(message));
229         }
230 
231         /*
232          * Send the unicast notification to the remote server in an asynchronous manner. 11/07/2014, Bing Li
233          */
234         public void notifyUnicastly(String message)
235         {
236             if (!this.unicastEventer.isReady())
237             {
238                 this.pool.execute(this.unicastEventer);
239             }
240             this.unicastEventer.notify(this.ip, this.port, new ClientForUnicastNotification(message));
241         }
242 
243         /*
244          * Send the unicast notification to the remote server in an asynchronous manner. 11/07/2014, Bing Li
245          */
246         public void notifyAnycastly(String message)
247         {
248             if (!this.anycastEventer.isReady())
249             {
250                 this.pool.execute(this.anycastEventer);
251             }
252             this.anycastEventer.notify(this.ip, this.port, new ClientForAnycastNotification(message));
253         }
254     }
```

List 3.10 The code of ClientEventer.java

To achieve the goal to send your notification, TestNotification, you are required to update the code of ClientEventer.java. Because of the natures of GreatFree, you can program it with CPR as well. Since you must be more familiar with CPR, let us move forward a little bit faster. You just copy and paste all of the lines, i.e., Line 39, Line 87, Line 120~126, Line 210~217, which contain or be related to WeatherNotification inside ClientEventer.java of List 3.10. After the operation, the code is turned into the one as shown in Figure 3.47, Figure 3.48, Figure 3.49 and Figure 3.50, respectively. Because the width of the book page is shorter than the editor of Eclipse, as mentioned before, the line numbers are different from the ones in List 3.10. Fortunately, it does not affect you to understand the content.

```
31    private int port;
32    // The synchronous eventer to send the online notification. 11/07/2014, Bing Li
33    private SyncRemoteEventer<OnlineNotification> onlineEventer;
34    // The synchronous eventer to send the registering notification. 11/07/2014, Bing Li
35    private SyncRemoteEventer<RegisterClientNotification> registerClientEventer;
36    // The synchronous eventer to send the unregistering notification. 11/07/2014, Bing Li
37    private SyncRemoteEventer<UnregisterClientNotification> unregisterClientEventer;
38    // The asynchronous eventer to send one instance of WeatherNotification to the remote server to set
39 the value of the weather. 02/15/2016, Bing Li
40    private AsyncRemoteEventer<WeatherNotification> weatherEventer;
41
42    private AsyncRemoteEventer<WeatherNotification> weatherEventer;
43
44    // The asynchronous eventer to send one instance of ClientForBroadcastNotification to the remote
45 server. 02/15/2016, Bing Li
46    private AsyncRemoteEventer<ClientForBroadcastNotification> broadcastEventer;
47    // The asynchronous eventer to send one instance of ClientForUnicastNotification to the remote
48 server. 02/15/2016, Bing Li
49    private AsyncRemoteEventer<ClientForUnicastNotification> unicastEventer;
50    // The asynchronous eventer to send one instance of ClientForAnycastNotification to the remote
51 server. 02/15/2016, Bing Li
52    private AsyncRemoteEventer<ClientForAnycastNotification> anycastEventer;
53
```

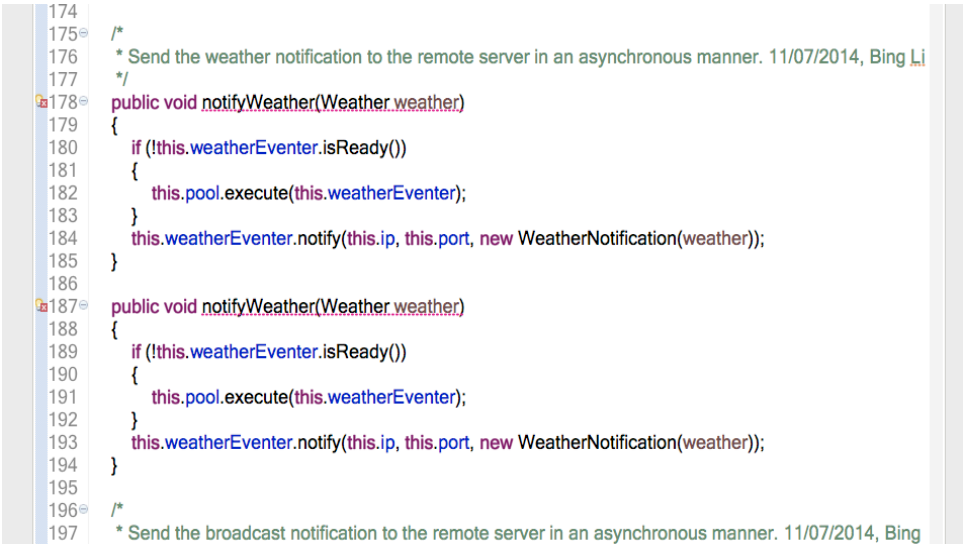
Figure 3.47 The first segment of code of ClientEventer.java after copying and pasting

```
77    /*
78     * Dispose the eventers. 11/07/2014, Bing Li
79     */
80    public void dispose() throws InterruptedException
81    {
82       this.onlineEventer.dispose();
83       this.registerClientEventer.dispose();
84       this.unregisterClientEventer.dispose();
85       this.weatherEventer.dispose();
86
87       this.weatherEventer.dispose();
88
89       this.broadcastEventer.dispose();
90       this.unicastEventer.dispose();
91       this.anycastEventer.dispose();
92
93       // Shutdown the thread pool. 11/07/2014, Bing Li
94       this.pool.shutdown();
95    }
96
97    /*
98     * Initialize the eventers. 11/07/2014, Bing Li
99     */
```

Figure 3.48 The second segment of code of ClientEventer.java after copying and pasting

```
116       // Initialize the asynchronous eventer. Since the eventer is sent out to the remote server
117 asynchronously, more parameters are required to set. 02/15/2016, Bing Li
118       this.weatherEventer = new
119 AsyncRemoteEventer<WeatherNotification>(ClientPool.LOCAL().getPool(), this.pool,
120 ClientConfig.EVENT_QUEUE_SIZE, ClientConfig.EVENTER_SIZE,
121 ClientConfig.EVENTING_WAIT_TIME, ClientConfig.EVENTER_WAIT_TIME,
122 ClientConfig.EVENTER_WAIT_ROUND, ClientConfig.EVENT_IDLE_CHECK_DELAY,
123 ClientConfig.EVENT_IDLE_CHECK_PERIOD, Scheduler.GREATFREE().getSchedulerPool());
124
125       this.weatherEventer = new
126 AsyncRemoteEventer<WeatherNotification>(ClientPool.LOCAL().getPool(), this.pool,
127 ClientConfig.EVENT_QUEUE_SIZE, ClientConfig.EVENTER_SIZE,
128 ClientConfig.EVENTING_WAIT_TIME, ClientConfig.EVENTER_WAIT_TIME,
129 ClientConfig.EVENTER_WAIT_ROUND, ClientConfig.EVENT_IDLE_CHECK_DELAY,
130 ClientConfig.EVENT_IDLE_CHECK_PERIOD, Scheduler.GREATFREE().getSchedulerPool());
131
132       // Initialize the asynchronous eventer. Since the eventer is sent out to the remote server
133 asynchronously, more parameters are required to set. 02/15/2016, Bing Li
134       this.broadcastEventer = new
135 AsyncRemoteEventer<ClientForBroadcastNotification>(ClientPool.LOCAL().getPool(), this.pool,
136 ClientConfig.EVENT_QUEUE_SIZE, ClientConfig.EVENTER_SIZE,
137 ClientConfig.EVENTING_WAIT_TIME, ClientConfig.EVENTER_WAIT_TIME,
    ClientConfig.EVENTER_WAIT_ROUND, ClientConfig.EVENT_IDLE_CHECK_DELAY,
```

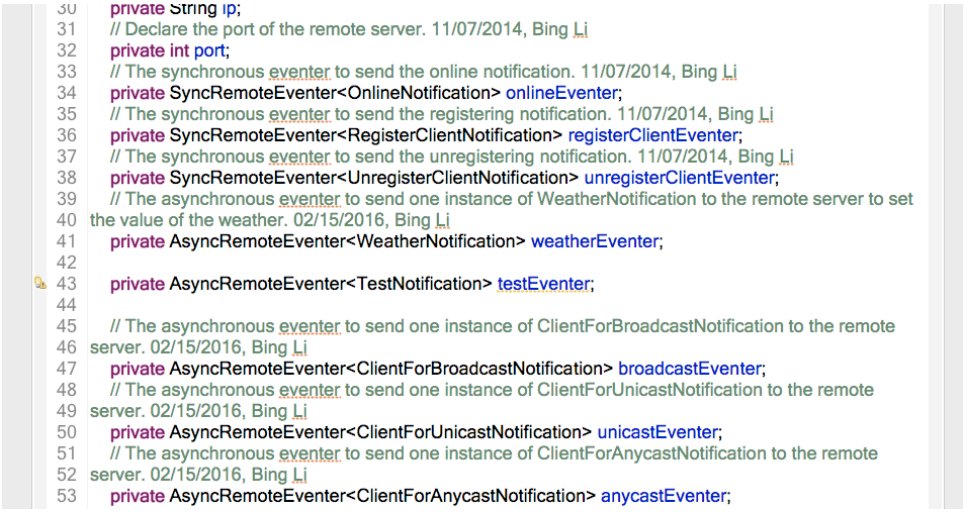
Figure 3.49 The third segment of code of ClientEventer.java after copying and pasting



```
174
175    /*
176     * Send the weather notification to the remote server in an asynchronous manner. 11/07/2014, Bing Li
177     */
178    public void notifyWeather(Weather weather)
179    {
180       if (!this.weatherEventer.isReady())
181       {
182          this.pool.execute(this.weatherEventer);
183       }
184       this.weatherEventer.notify(this.ip, this.port, new WeatherNotification(weather));
185    }
186
187    public void notifyWeather(Weather weather)
188    {
189       if (!this.weatherEventer.isReady())
190       {
191          this.pool.execute(this.weatherEventer);
192       }
193       this.weatherEventer.notify(this.ip, this.port, new WeatherNotification(weather));
194    }
195
196    /*
197     * Send the broadcast notification to the remote server in an asynchronous manner. 11/07/2014, Bing
```

Figure 3.50 The fourth segment of code of ClientEventer.java after copying and pasting

The next step is the operation of replacing and let us start from Figure 3.47. The entire procedure is identical to what did in Section 3.4. First, you need to replace WeatherNotification, with TestNotification in Line 42 following Table 3.2. And then in the same line, rename the attribute, weatherEventer, to a new name depending on your notification. For example, you can rename it as testEventer. A compilation error is caused by the replacement and it can be fixed by selecting the option of the prompts of Eclipse. After that, the first segment is updated as the one shown in Figure 3.51. You must notice that there is a warning in Line 43 you just work on. Do not worry. It will be gone after the following updates.



```
30     private String ip;
31     // Declare the port of the remote server. 11/07/2014, Bing Li
32     private int port;
33     // The synchronous eventer to send the online notification. 11/07/2014, Bing Li
34     private SyncRemoteEventer<OnlineNotification> onlineEventer;
35     // The synchronous eventer to send the registering notification. 11/07/2014, Bing Li
36     private SyncRemoteEventer<RegisterClientNotification> registerClientEventer;
37     // The synchronous eventer to send the unregistering notification. 11/07/2014, Bing Li
38     private SyncRemoteEventer<UnregisterClientNotification> unregisterClientEventer;
39     // The asynchronous eventer to send one instance of WeatherNotification to the remote server to set
40  the value of the weather. 02/15/2016, Bing Li
41     private AsyncRemoteEventer<WeatherNotification> weatherEventer;
42
43     private AsyncRemoteEventer<TestNotification> testEventer;
44
45     // The asynchronous eventer to send one instance of ClientForBroadcastNotification to the remote
46  server. 02/15/2016, Bing Li
47     private AsyncRemoteEventer<ClientForBroadcastNotification> broadcastEventer;
48     // The asynchronous eventer to send one instance of ClientForUnicastNotification to the remote
49  server. 02/15/2016, Bing Li
50     private AsyncRemoteEventer<ClientForUnicastNotification> unicastEventer;
51     // The asynchronous eventer to send one instance of ClientForAnycastNotification to the remote
52  server. 02/15/2016, Bing Li
53     private AsyncRemoteEventer<ClientForAnycastNotification> anycastEventer;
```

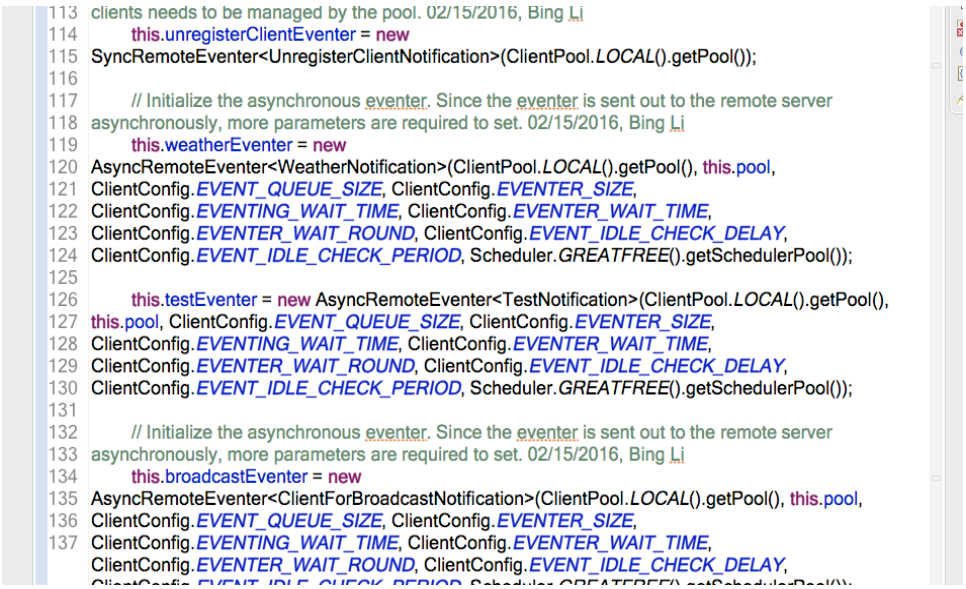
Figure 3.51 The first segment of code of ClientEventer.java after replacing

In Figure 3.48, you just made a copy. So Line 85 and Line 87 become the same. As you just create a new eventer, testEventer, just replace the one, weatherEventer, with it. Then, the segment is changed to the one shown in Figure 3.52. At this moment, the warning in Line 43 also disappears.

```
77
78  /*
79   * Dispose the eventers. 11/07/2014, Bing Li
80   */
81  public void dispose() throws InterruptedException
82  {
83      this.onlineEventer.dispose();
84      this.registerClientEventer.dispose();
85      this.unregisterClientEventer.dispose();
86      this.weatherEventer.dispose();
87
88      this.testEventer.dispose();
89
90      this.broadcastEventer.dispose();
91      this.unicastEventer.dispose();
92      this.anycastEventer.dispose();
93
94      // Shutdown the thread pool. 11/07/2014, Bing Li
95      this.pool.shutdown();
96  }
97
98  /*
99   * Initialize the eventers. 11/07/2014, Bing Li
100  */
```

Figure 3.52 The second segment of code of ClientEventer.java after replacing

Figure 3.49 has the similar situation to Figure 3.47 and Figure 3.48. You just need to make two replacements, i.e., replacing weatherEventer with testEventer and replacing WeatherNotification with TestNotification. Then, the third segment of code is changed to the one shown in Figure 3.53.



```
113 clients needs to be managed by the pool. 02/15/2016, Bing Li
114     this.unregisterClientEventer = new
115 SyncRemoteEventer<UnregisterClientNotification>(ClientPool.LOCAL().getPool());
116
117     // Initialize the asynchronous eventer. Since the eventer is sent out to the remote server
118 asynchronously, more parameters are required to set. 02/15/2016, Bing Li
119     this.weatherEventer = new
120 AsyncRemoteEventer<WeatherNotification>(ClientPool.LOCAL().getPool(), this.pool,
121 ClientConfig.EVENT_QUEUE_SIZE, ClientConfig.EVENTER_SIZE,
122 ClientConfig.EVENTING_WAIT_TIME, ClientConfig.EVENTER_WAIT_TIME,
123 ClientConfig.EVENTER_WAIT_ROUND, ClientConfig.EVENT_IDLE_CHECK_DELAY,
124 ClientConfig.EVENT_IDLE_CHECK_PERIOD, Scheduler.GREATFREE().getSchedulerPool());
125
126     this.testEventer = new AsyncRemoteEventer<TestNotification>(ClientPool.LOCAL().getPool(),
127 this.pool, ClientConfig.EVENT_QUEUE_SIZE, ClientConfig.EVENTER_SIZE,
128 ClientConfig.EVENTING_WAIT_TIME, ClientConfig.EVENTER_WAIT_TIME,
129 ClientConfig.EVENTER_WAIT_ROUND, ClientConfig.EVENT_IDLE_CHECK_DELAY,
130 ClientConfig.EVENT_IDLE_CHECK_PERIOD, Scheduler.GREATFREE().getSchedulerPool());
131
132     // Initialize the asynchronous eventer. Since the eventer is sent out to the remote server
133 asynchronously, more parameters are required to set. 02/15/2016, Bing Li
134     this.broadcastEventer = new
135 AsyncRemoteEventer<ClientForBroadcastNotification>(ClientPool.LOCAL().getPool(), this.pool,
136 ClientConfig.EVENT_QUEUE_SIZE, ClientConfig.EVENTER_SIZE,
137 ClientConfig.EVENTING_WAIT_TIME, ClientConfig.EVENTER_WAIT_TIME,
    ClientConfig.EVENTER_WAIT_ROUND, ClientConfig.EVENT_IDLE_CHECK_DELAY,
```

Figure 3.53 The third segment of code of ClientEventer.java after replacing

For the fourth segment as shown in Figure 3.50, it is a little bit more work to be accomplished. First, there is an error in Line 187. It is caused by your copying and pasting. So Line 178 and Line 187 are identical and it is illegal because two methods should not have the identical signature. I do not want to talk about techniques in the chapter. But it is also impossible for you to know nothing when programming clouds, right? At least, the error is extremely easy to be fixed. Just rename the method, notifyWeather, as notifyTest. Then, the error should be gone, as shown in Figure 3.54.

```
175
176    /*
177     * Send the weather notification to the remote server in an asynchronous manner. 11/07/2014, Bing Li
178     */
179    public void notifyWeather(Weather weather)
180    {
181        if (!this.weatherEventer.isReady())
182        {
183            this.pool.execute(this.weatherEventer);
184        }
185        this.weatherEventer.notify(this.ip, this.port, new WeatherNotification(weather));
186    }
187
188    public void notifyTest(Weather weather)
189    {
190        if (!this.weatherEventer.isReady())
191        {
192            this.pool.execute(this.weatherEventer);
193        }
194        this.weatherEventer.notify(this.ip, this.port, new WeatherNotification(weather));
195    }
196
197    /*
198     * Send the broadcast notification to the remote server in an asynchronous manner. 11/07/2014, Bing
```

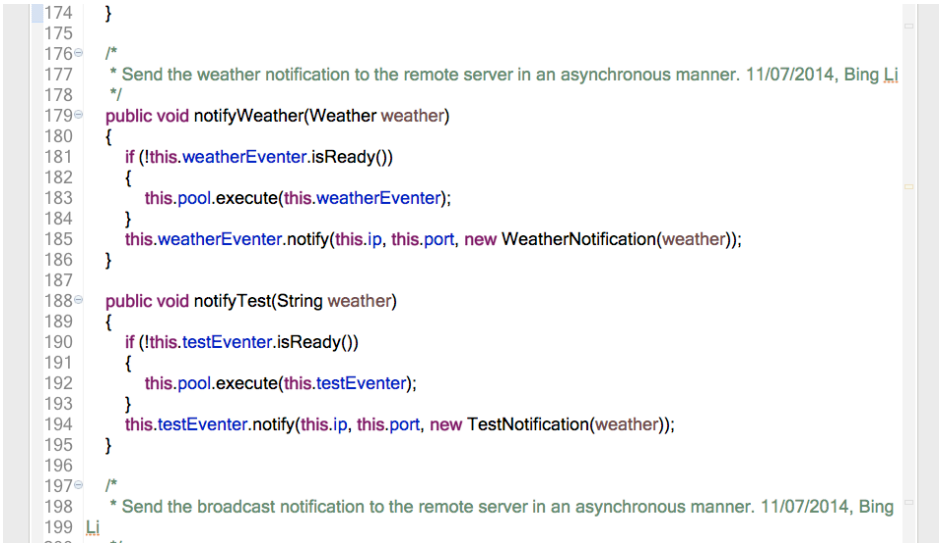
Figure 3.54 The fourth segment of code of ClientEventer.java after renaming the method, notifyWeather, as notifyTest

To continue, you just need to make two replacements, which you have done for the code in Figure 3.53, i.e., replacing weatherEventer with testEventer and replacing WeatherNotification with TestNotification. After that, your code should be exactly like the one shown in Figure 3.55.

```
173        }
174    }
175
176    /*
177     * Send the weather notification to the remote server in an asynchronous manner. 11/07/2014, Bing Li
178     */
179    public void notifyWeather(Weather weather)
180    {
181        if (!this.weatherEventer.isReady())
182        {
183            this.pool.execute(this.weatherEventer);
184        }
185        this.weatherEventer.notify(this.ip, this.port, new WeatherNotification(weather));
186    }
187
188    public void notifyTest(Weather weather)
189    {
190        if (!this.testEventer.isReady())
191        {
192            this.pool.execute(this.testEventer);
193        }
194        this.testEventer.notify(this.ip, this.port, new TestNotification(weather));
195    }
196
197    /*
198     * Send the broadcast notification to the remote server in an asynchronous manner. 11/07/2014, Bing
199 Li
200     */
```

Figure 3.55 The fourth segment of code of ClientEventer.java after replacing weatherEventer with testEventer and replacing WeatherNotification with TestNotification

Unfortunately, a new error in Line 194 is raised after the replacements. Here, I have to mention a little bit technique although it is very simple from any levels of programmers' point of view. The unique parameter of the constructor of TestNotification needs to input data in the type of String. However, the current parameter in Line 194 is not String but the instance of Weather. So to fix it, replace Weather in Line 188 with String. Then, the error is gone, as shown in Figure 3.56.



```
174     }
175
176     /*
177      * Send the weather notification to the remote server in an asynchronous manner. 11/07/2014, Bing Li
178      */
179     public void notifyWeather(Weather weather)
180     {
181         if (!this.weatherEventer.isReady())
182         {
183             this.pool.execute(this.weatherEventer);
184         }
185         this.weatherEventer.notify(this.ip, this.port, new WeatherNotification(weather));
186     }
187
188     public void notifyTest(String weather)
189     {
190         if (!this.testEventer.isReady())
191         {
192             this.pool.execute(this.testEventer);
193         }
194         this.testEventer.notify(this.ip, this.port, new TestNotification(weather));
195     }
196
197     /*
198      * Send the broadcast notification to the remote server in an asynchronous manner. 11/07/2014, Bing
199 Li
```

Figure 3.56 The four segment of code of ClientEventer.java after replacing Weather with String

You are almost done although there is still one tiny problem, which is neither an error nor a warning. In Line 188, the parameter of the method, notifyTest, is named weather in the type of String. It is not a proper name since it is the heritage from the previous copying and pasting. To refine your code, you can name the string of weather as testMessage, which is consistent with the parameter name of your notification constructor. The final state of the four segment should be exactly like the one shown in Figure 3.57.

```
174     }
175
176     /*
177      * Send the weather notification to the remote server in an asynchronous manner. 11/07/2014, Bing Li
178      */
179     public void notifyWeather(Weather weather)
180     {
181         if (!this.weatherEventer.isReady())
182         {
183             this.pool.execute(this.weatherEventer);
184         }
185         this.weatherEventer.notify(this.ip, this.port, new WeatherNotification(weather));
186     }
187
188     public void notifyTest(String testMessage)
189     {
190         if (!this.testEventer.isReady())
191         {
192             this.pool.execute(this.testEventer);
193         }
194         this.testEventer.notify(this.ip, this.port, new TestNotification(testMessage));
195     }
196
```

Figure 3.57 The four segment of code of ClientEventer.java after renaming weather as testMessage

The updated complete code of ClientEventer.java is shown in List 3.11.

```
1   package com.greatfree.testing.client;
2
3   import java.io.IOException;
4
```

```
5   import com.greatfree.concurrency.Scheduler;
6   import com.greatfree.concurrency.ThreadPool;
7   import com.greatfree.remote.AsyncRemoteEventer;
8   import com.greatfree.remote.SyncRemoteEventer;
9   import com.greatfree.testing.data.ClientConfig;
10  import com.greatfree.testing.data.Weather;
11  import com.greatfree.testing.message.ClientForAnycastNotification;
12  import com.greatfree.testing.message.ClientForBroadcastNotification;
13  import com.greatfree.testing.message.ClientForUnicastNotification;
14  import com.greatfree.testing.message.OnlineNotification;
15  import com.greatfree.testing.message.RegisterClientNotification;
16  import com.greatfree.testing.message.TestNotification;
17  import com.greatfree.testing.message.UnregisterClientNotification;
18  import com.greatfree.testing.message.WeatherNotification;
19  import com.greatfree.util.NodeID;
20
21  /*
22   * The class is an example that applies SynchRemoteEventer and AsyncRemoteEventer. 11/05/2014, Bing Li
23   */
24
25  // Created: 11/05/2014, Bing Li
26  public class ClientEventer
27  {
28      // Declare the ip of the remote server. 11/07/2014, Bing Li
29      private String ip;
30      // Declare the port of the remote server. 11/07/2014, Bing Li
31      private int port;
32      // The synchronous eventer to send the online notification. 11/07/2014, Bing Li
33      private SyncRemoteEventer<OnlineNotification> onlineEventer;
34      // The synchronous eventer to send the registering notification. 11/07/2014, Bing Li
35      private SyncRemoteEventer<RegisterClientNotification> registerClientEventer;
36      // The synchronous eventer to send the unregistering notification. 11/07/2014, Bing Li
37      private SyncRemoteEventer<UnregisterClientNotification> unregisterClientEventer;
38      // The asynchronous eventer to send one instance of WeatherNotification to the remote server
39      // to set the value of the weather. 02/15/2016, Bing Li
40      private AsyncRemoteEventer<WeatherNotification> weatherEventer;
41
42      private AsyncRemoteEventer<TestNotification> testEventer;
43
44      // The asynchronous eventer to send one instance of ClientForBroadcastNotification to the
45      // remote server. 02/15/2016, Bing Li
46      private AsyncRemoteEventer<ClientForBroadcastNotification> broadcastEventer;
47      // The asynchronous eventer to send one instance of ClientForUnicastNotification to the remote
48      // server. 02/15/2016, Bing Li
49      private AsyncRemoteEventer<ClientForUnicastNotification> unicastEventer;
50      // The asynchronous eventer to send one instance of ClientForAnycastNotification to the remote
51      // server. 02/15/2016, Bing Li
52      private AsyncRemoteEventer<ClientForAnycastNotification> anycastEventer;
53
54      // A thread pool to assist sending notification asynchronously. 11/07/2014, Bing Li
55      private ThreadPool pool;
56
57      /*
58       * Initialize. 11/07/2014, Bing Li
59       */
60      private ClientEventer()
61      {
62      }
63
64      /*
65       * A singleton implementation. 11/07/2014, Bing Li
66       */
67      private static ClientEventer instance = new ClientEventer();
68
69      public static ClientEventer NOTIFY()
70      {
71          if (instance == null)
72          {
73              instance = new ClientEventer();
74              return instance;
75          }
76          else
77          {
78              return instance;
79          }
80      }
81
```

```
82          /*
82           * Dispose the eventers. 11/07/2014, Bing Li
83           */
85          public void dispose() throws InterruptedException
86          {
87              this.onlineEventer.dispose();
88              this.registerClientEventer.dispose();
89              this.unregisterClientEventer.dispose();
90              this.weatherEventer.dispose();
91
92              this.testEventer.dispose();
93
94              this.broadcastEventer.dispose();
95              this.unicastEventer.dispose();
96              this.anycastEventer.dispose();
97
98              // Shutdown the thread pool. 11/07/2014, Bing Li
99              this.pool.shutdown();
100         }
101
102         /*
103          * Initialize the eventers. 11/07/2014, Bing Li
104          */
105         public void init(String ip, int port)
106         {
107             this.ip = ip;
108             this.port = port;
109             this.pool = new ThreadPool(ClientConfig.EVENTER_THREAD_POOL_SIZE,
110                 ClientConfig.EVENTER_THREAD_POOL_ALIVE_TIME);
111             // Initialize the synchronous eventer. The FreeClient pool is one parameter for the initialization.
112             // The clients needs to be managed by the pool. 02/15/2016, Bing Li
113             this.onlineEventer = new SyncRemoteEventer<OnlineNotification>(ClientPool.LOCAL().getPool());
114             // Initialize the synchronous eventer. The FreeClient pool is one parameter for the initialization.
115             // The clients needs to be managed by the pool. 02/15/2016, Bing Li
116             this.registerClientEventer = new SyncRemoteEventer<RegisterClientNotification>
117                 (ClientPool.LOCAL().getPool());
118             // Initialize the synchronous eventer. The FreeClient pool is one parameter for the initialization.
119             // The clients needs to be managed by the pool. 02/15/2016, Bing Li
120             this.unregisterClientEventer = new SyncRemoteEventer<UnregisterClientNotification>
121                 (ClientPool.LOCAL().getPool());
122
123             // Initialize the asynchronous eventer. Since the eventer is sent out to the remote server
124             // asynchronously, more parameters are required to set. 02/15/2016, Bing Li
125             this.weatherEventer = new AsyncRemoteEventer<WeatherNotification>
126                 (ClientPool.LOCAL().getPool(), this.pool, ClientConfig.EVENT_QUEUE_SIZE,
127                 ClientConfig.EVENTER_SIZE, ClientConfig.EVENTING_WAIT_TIME,
128                 ClientConfig.EVENTER_WAIT_TIME, ClientConfig.EVENTER_WAIT_ROUND,
129                 ClientConfig.EVENT_IDLE_CHECK_DELAY,
130                 ClientConfig.EVENT_IDLE_CHECK_PERIOD, Scheduler.GREATFREE().getSchedulerPool());
131
132             this.testEventer = new AsyncRemoteEventer<TestNotification>(ClientPool.LOCAL().getPool(),
133                 this.pool, ClientConfig.EVENT_QUEUE_SIZE, ClientConfig.EVENTER_SIZE,
134                 ClientConfig.EVENTING_WAIT_TIME, ClientConfig.EVENTER_WAIT_TIME,
135                 ClientConfig.EVENTER_WAIT_ROUND, ClientConfig.EVENT_IDLE_CHECK_DELAY,
136                 ClientConfig.EVENT_IDLE_CHECK_PERIOD, Scheduler.GREATFREE().getSchedulerPool());
137
138             // Initialize the asynchronous eventer. Since the eventer is sent out to the remote server
139             // asynchronously, more parameters are required to set. 02/15/2016, Bing Li
140             this.broadcastEventer = new AsyncRemoteEventer<ClientForBroadcastNotification>
141                 (ClientPool.LOCAL().getPool(), this.pool, ClientConfig.EVENT_QUEUE_SIZE,
142                 ClientConfig.EVENTER_SIZE, ClientConfig.EVENTING_WAIT_TIME,
143                 ClientConfig.EVENTER_WAIT_TIME, ClientConfig.EVENTER_WAIT_ROUND,
144                 ClientConfig.EVENT_IDLE_CHECK_DELAY,
145                 ClientConfig.EVENT_IDLE_CHECK_PERIOD, Scheduler.GREATFREE().getSchedulerPool());
146
147             // Initialize the asynchronous eventer. Since the eventer is sent out to the remote server
148             // asynchronously, more parameters are required to set. 02/15/2016, Bing Li
149             this.unicastEventer = new AsyncRemoteEventer<ClientForUnicastNotification>
150                 (ClientPool.LOCAL().getPool(), this.pool, ClientConfig.EVENT_QUEUE_SIZE,
151                 ClientConfig.EVENTER_SIZE, ClientConfig.EVENTING_WAIT_TIME,
152                 ClientConfig.EVENTER_WAIT_TIME, ClientConfig.EVENTER_WAIT_ROUND,
153                 ClientConfig.EVENT_IDLE_CHECK_DELAY,
154                 ClientConfig.EVENT_IDLE_CHECK_PERIOD, Scheduler.GREATFREE().getSchedulerPool());
155
156             // Initialize the asynchronous eventer. Since the eventer is sent out to the remote server
157             // asynchronously, more parameters are required to set. 02/15/2016, Bing Li
158             this.anycastEventer = new AsyncRemoteEventer<ClientForAnycastNotification>
```

```
159             (ClientPool.LOCAL().getPool(), this.pool, ClientConfig.EVENT_QUEUE_SIZE,
160             ClientConfig.EVENTER_SIZE, ClientConfig.EVENTING_WAIT_TIME,
161             ClientConfig.EVENTER_WAIT_TIME, ClientConfig.EVENTER_WAIT_ROUND,
162             ClientConfig.EVENT_IDLE_CHECK_DELAY,
163             ClientConfig.EVENT_IDLE_CHECK_PERIOD, Scheduler.GREATFREE().getSchedulerPool());
164     }
165
166     /*
167      * Send the online notification to the remote server in a synchronous manner. 11/07/2014, Bing Li
168      */
169     public void notifyOnline() throws IOException, InterruptedException
170     {
171         this.onlineEventer.notify(this.ip, this.port, new OnlineNotification());
172     }
173
174     /*
175     * Send the registering notification to the remote server in a synchronous manner. 11/07/2014, Bing Li
176     */
177     public void register()
178     {
179         try
180         {
181             this.registerClientEventer.notify(this.ip, this.port, new
182                 RegisterClientNotification(NodeID.DISTRIBUTED().getKey()));
183         }
184         catch (IOException e)
185         {
186             e.printStackTrace();
187         }
188         catch (InterruptedException e)
189         {
190             e.printStackTrace();
191         }
192     }
193
194     /*
195      * Send the unregistering notification to the remote server in a synchronous manner. 11/07/2014, Bing Li
196      */
197     public void unregister()
198     {
199         try
200         {
201             this.unregisterClientEventer.notify(this.ip, this.port, new
202                 UnregisterClientNotification(NodeID.DISTRIBUTED().getKey()));
203         }
204         catch (IOException e)
205         {
206             e.printStackTrace();
207         }
208         catch (InterruptedException e)
209         {
210             e.printStackTrace();
211         }
212     }
213
214     /*
215      * Send the weather notification to the remote server in an asynchronous manner. 11/07/2014, Bing Li
216      */
217     public void notifyWeather(Weather weather)
218     {
219         if (!this.weatherEventer.isReady())
220         {
221             this.pool.execute(this.weatherEventer);
222         }
223         this.weatherEventer.notify(this.ip, this.port, new WeatherNotification(weather));
224     }
225
226     public void notifyTest(String testMessage)
227     {
228         if (!this.testEventer.isReady())
229         {
230             this.pool.execute(this.testEventer);
231         }
232         this.testEventer.notify(this.ip, this.port, new TestNotification(testMessage));
233     }
234
235     /*
```

```
236         * Send the broadcast notification to the remote server in an asynchronous manner. 11/07/2014, Bing Li
237         */
238        public void notifyBroadcastly(String message)
239        {
240            if (!this.broadcastEventer.isReady())
241            {
242                this.pool.execute(this.broadcastEventer);
243            }
244            this.broadcastEventer.notify(this.ip, this.port, new ClientForBroadcastNotification(message));
245        }
246
247        /*
248         * Send the unicast notification to the remote server in an asynchronous manner. 11/07/2014, Bing Li
249         */
250        public void notifyUnicastly(String message)
251        {
252            if (!this.unicastEventer.isReady())
253            {
254                this.pool.execute(this.unicastEventer);
255            }
256            this.unicastEventer.notify(this.ip, this.port, new ClientForUnicastNotification(message));
257        }
258
259        /*
260         * Send the unicast notification to the remote server in an asynchronous manner. 11/07/2014, Bing Li
261         */
262        public void notifyAnycastly(String message)
263        {
264            if (!this.anycastEventer.isReady())
265            {
266                this.pool.execute(this.anycastEventer);
267            }
268            this.anycastEventer.notify(this.ip, this.port, new ClientForAnycastNotification(message));
269        }
270    }
```

List 3.11 The code of ClientEventer.java after CPR

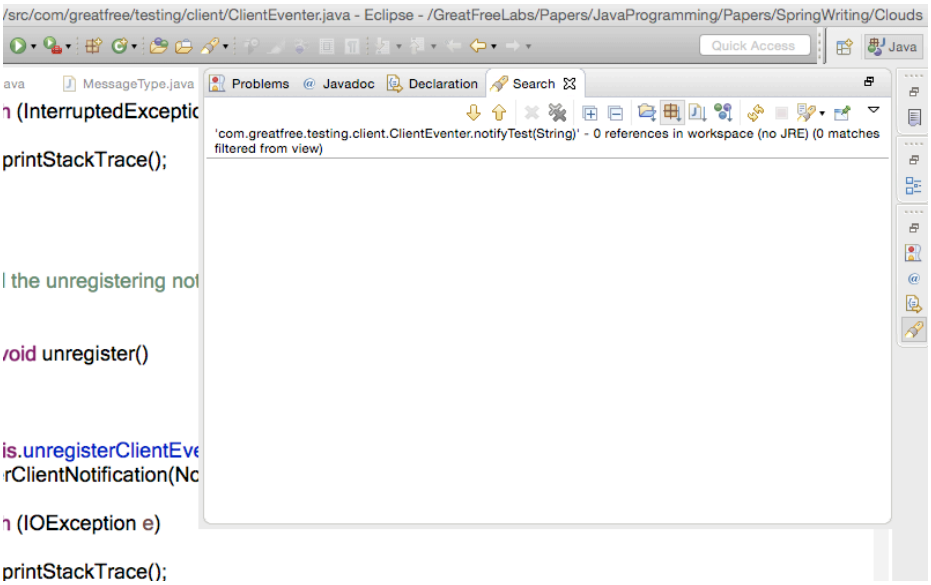


Figure 3.58 No references are available for testNotify()

4.2 Invoking the Client Eventer

I am sorry that I have to talk about techniques for a little bit at this time. Since you have finished updating the code of ClientEventer.java, now you take a test whether

the method, notifyTest(), you just added is invoked or not. To do that, you can search the references of the method, just what you have done for WeatherNotification in Section 3.1, as shown in Figure 3.15 and Figure 3.16.

If you do that, unfortunately, no any references are available for your new method, notifyTest(), ash shown in Figure 3.58. It indicates that your method is not invoked by any classes at the client side. Thus, the notification cannot be sent to the server side. To solve the problem, you can get assistance from the sample method, notifyWeather(). It is a straightforward solution since you just CPR the method. What you can do is to search the references of notifyWeather(). According to Figure 3.59, the sample method has only one reference, ClientUI. So now you just double-click the icon of ClientUI in the reference window to open it and check notifyWeather is invoked there. The code of ClientUI.java is listed in List 3.12.

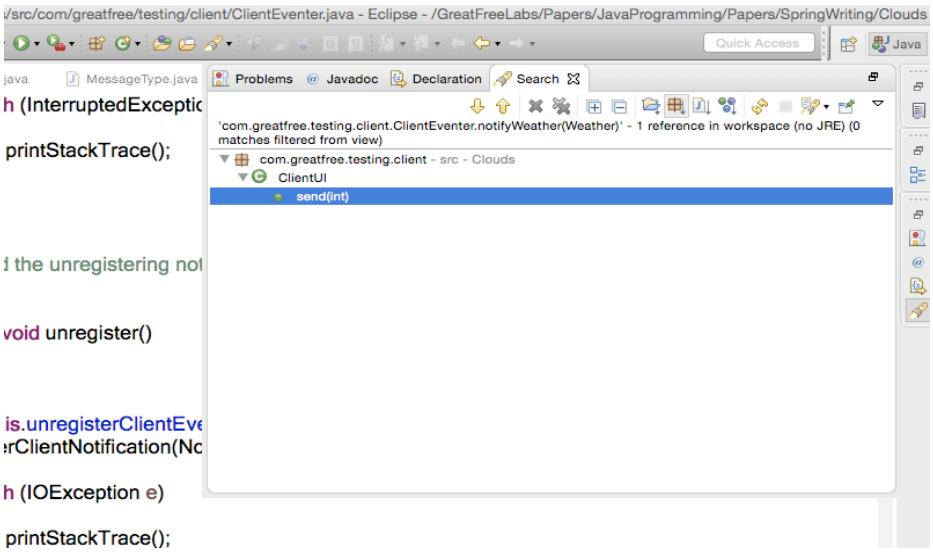


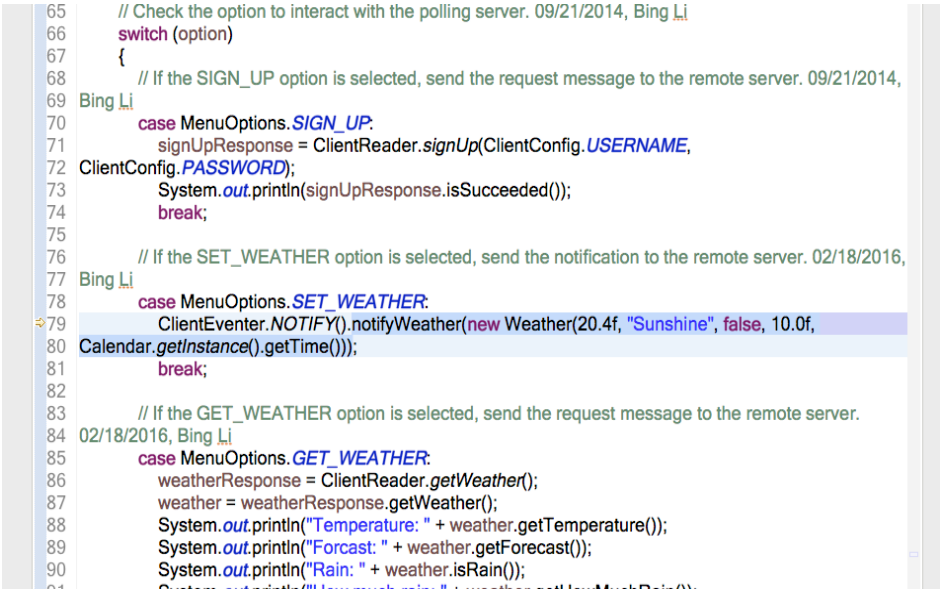
Figure 3.59 The reference of notifyWeather(), ClientUI

```
1   package com.greatfree.testing.client;
2
3   import java.util.Calendar;
4
5   import com.greatfree.testing.data.ClientConfig;
6   import com.greatfree.testing.data.Weather;
7   import com.greatfree.testing.message.SignUpResponse;
8   import com.greatfree.testing.message.WeatherResponse;
9
10  /*
11   * The class aims to print a menu list on the screen for users to interact with the client and communicate with
12   * the polling server. The menu is unique in the client such that it is implemented in the pattern of a singleton.
13   * 09/21/2014, Bing Li
14   */
15
16  // Created: 09/21/2014, Bing Li
17  public class ClientUI
18  {
19      /*
20       * Initialize. 09/21/2014, Bing Li
21       */
22      private ClientUI()
23      {
24      }
25
26      /*
27       * Initialize a singleton. 09/21/2014, Bing Li
28       */
29      private static ClientUI instance = new ClientUI();
```

```
30
31          public static ClientUI FACE()
32          {
33              if (instance == null)
34              {
35                  instance = new ClientUI();
36                  return instance;
37              }
38              else
39              {
40                  return instance;
41              }
45          }
46
47          /*
48           * Print the menu list on the screen. 09/21/2014, Bing Li
49           */
50          public void printMenu()
51          {
52              System.out.println(ClientMenu.MENU_HEAD);
53              System.out.println(ClientMenu.SIGN_UP);
54              System.out.println(ClientMenu.SET_WEATHER);
55              System.out.println(ClientMenu.GET_WEATHER);
56              System.out.println(ClientMenu.QUIT);
57              System.out.println(ClientMenu.MENU_TAIL);
58              System.out.println(ClientMenu.INPUT_PROMPT);
59          }
60
61          /*
62           * Send the users' option to the polling server. 09/21/2014, Bing Li
63           */
64          public void send(int option)
65          {
66              SignUpResponse signUpResponse;
67              WeatherResponse weatherResponse;
68              Weather weather;
69
70              // Check the option to interact with the polling server. 09/21/2014, Bing Li
71              switch (option)
72              {
73                  // If the SIGN_UP option is selected, send the request message to the remote
74                  // server. 09/21/2014, Bing Li
75                  case MenuOptions.SIGN_UP:
76                      signUpResponse = ClientReader.signUp(ClientConfig.USERNAME,
77                          ClientConfig.PASSWORD);
78                      System.out.println(signUpResponse.isSucceeded());
79                      break;
80
81                  // If the SET_WEATHER option is selected, send the notification to the remote
82                  // server. 02/18/2016, Bing Li
83                  case MenuOptions.SET_WEATHER:
84                      ClientEventer.NOTIFY().notifyWeather(new Weather(20.4f, "Sunshine", false,
85                          10.0f, Calendar.getInstance().getTime()));
86                      break;
87
88                  // If the GET_WEATHER option is selected, send the request message to the
89                  // remote server. 02/18/2016, Bing Li
90                  case MenuOptions.GET_WEATHER:
91                      weatherResponse = ClientReader.getWeather();
92                      weather = weatherResponse.getWeather();
93                      System.out.println("Temperature: " + weather.getTemperature());
94                      System.out.println("Forcast: " + weather.getForecast());
95                      System.out.println("Rain: " + weather.isRain());
96                      System.out.println("How much rain: " + weather.getHowMuchRain());
97                      System.out.println("Time: " + weather.getTime());
98                      break;
99
100                 // If the quit option is selected, send the notification message to the remote
101                 // server. 09/21/2014, Bing Li
102                 case MenuOptions.QUIT:
103                     break;
104             }
105         }
106     }
```

List 3.12 The code of ClientUI.java

If you open the code of ClientUI.java by double-clicking its icon in the references of the method, notifyWeather(), you can see the line, Line 79, is in the color of deep blue. That means the method is invoked there, as shown in Figure 3.60.



```
65      // Check the option to interact with the polling server. 09/21/2014, Bing Li
66      switch (option)
67      {
68         // If the SIGN_UP option is selected, send the request message to the remote server. 09/21/2014,
69   Bing Li
70         case MenuOptions.SIGN_UP:
71            signUpResponse = ClientReader.signUp(ClientConfig.USERNAME,
72   ClientConfig.PASSWORD);
73            System.out.println(signUpResponse.isSucceeded());
74            break;
75
76         // If the SET_WEATHER option is selected, send the notification to the remote server. 02/18/2016,
77   Bing Li
78         case MenuOptions.SET_WEATHER:
79            ClientEventer.NOTIFY().notifyWeather(new Weather(20.4f, "Sunshine", false, 10.0f,
80   Calendar.getInstance().getTime()));
81            break;
82
83         // If the GET_WEATHER option is selected, send the request message to the remote server.
84   02/18/2016, Bing Li
85         case MenuOptions.GET_WEATHER:
86            weatherResponse = ClientReader.getWeather();
87            weather = weatherResponse.getWeather();
88            System.out.println("Temperature: " + weather.getTemperature());
89            System.out.println("Forcast: " + weather.getForecast());
90            System.out.println("Rain: " + weather.isRain());
```

Figure 3.60 One code segment of ClientUI.java

I have to say that the following updates are very intuitive if you have the very basic knowledge about computer programming. You can even skip what I discuss below because you can make the simple change in your own way.

According to Figure 3.60, you can also make copy for Line 78~81 and paste them immediately after Line 81, as shown in Figure 3.61.

```
70         case MenuOptions.SIGN_UP:
71            signUpResponse = ClientReader.signUp(ClientConfig.USERNAME,
72   ClientConfig.PASSWORD);
73            System.out.println(signUpResponse.isSucceeded());
74            break;
75
76         // If the SET_WEATHER option is selected, send the notification to the remote server.
77   02/18/2016, Bing Li
78         case MenuOptions.SET_WEATHER:
79            ClientEventer.NOTIFY().notifyWeather(new Weather(20.4f, "Sunshine", false, 10.0f,
80   Calendar.getInstance().getTime()));
81            break;
82
83         case MenuOptions.SET_WEATHER:
84            ClientEventer.NOTIFY().notifyWeather(new Weather(20.4f, "Sunshine", false, 10.0f,
85   Calendar.getInstance().getTime()));
86            break;
87
88         // If the GET_WEATHER option is selected, send the request message to the remote server.
89   02/18/2016, Bing Li
90         case MenuOptions.GET_WEATHER:
91            weatherResponse = ClientReader.getWeather();
92            weather = weatherResponse.getWeather();
93            System.out.println("Temperature: " + weather.getTemperature());
94            System.out.println("Forcast: " + weather.getForecast());
95            System.out.println("Rain: " + weather.isRain());
```

Figure 3.61 The code of ClientUI.java after copying and pasting

The compilation error is caused by the case statements in Line 78 and Line 83 because they are the same. That is the illegal in the syntax of Java. So you need to add

a new constant in the class of MenuOptions to replace the later one in Line 83. The code of MenuOptions.java is listed as follows in List 3.13.

```
1    package com.greatfree.testing.client;
2
3    /*
4     * The class contains all of constants of menu options. 09/22/2014, Bing Li
5     */
6
7    // Created: 09/21/2014, Bing Li
8    public class MenuOptions
9    {
10       public final static int NO_OPTION = -1;
11       public final static int SIGN_UP = 1;
12       public final static int SET_WEATHER = 2;
13       public final static int GET_WEATHER = 3;
14       public final static int QUIT = 0;
15   }
```

List 3.13 The code of MenuOptions.java

It is necessary to offer a meaningful name to the new menu option. In this case, it can be named as NOTIFY_TEST, as shown in the updated code, List 3.14.

```
1    package com.greatfree.testing.client;
2
3    /*
4     * The class contains all of constants of menu options. 09/22/2014, Bing Li
5     */
6
7    // Created: 09/21/2014, Bing Li
8    public class MenuOptions
9    {
10       public final static int NO_OPTION = -1;
11       public final static int SIGN_UP = 1;
12       public final static int SET_WEATHER = 2;
13       public final static int GET_WEATHER = 3;
14       public final static int NOTIFY_TEST = 4;
15       public final static int QUIT = 0;
16   }
```

List 3.14 The code of MenuOptions.java after adding a new option, NOTIFY_TEST

After the new menu option is added, you can do the replacement to overwrite SET_WEATHER with NOTIFY_TEST in Line 83 of Figure 3.61. One more replacement is to invoke the method, notifyTest(), in Line 84. The invocation of notifyWeather() should be removed. You need to assign one string to the method of notifyTest() since it needs a value for the notification to be sent. After that, the code is updated without any errors and warnings. The updated code of ClientUI.java is presented in List 3.15.

```
1    package com.greatfree.testing.client;
2
3    import java.util.Calendar;
4
5    import com.greatfree.testing.data.ClientConfig;
6    import com.greatfree.testing.data.Weather;
7    import com.greatfree.testing.message.SignUpResponse;
8    import com.greatfree.testing.message.WeatherResponse;
9
10   /*
11    * The class aims to print a menu list on the screen for users to interact with the client and communicate with
12    * the polling server. The menu is unique in the client such that it is implemented in the pattern of a singleton.
13    * 09/21/2014, Bing Li
```

```
14      */
15
16     // Created: 09/21/2014, Bing Li
17     public class ClientUI
18     {
19         /*
20          * Initialize. 09/21/2014, Bing Li
21          */
22         private ClientUI()
23         {
24         }
25
26         /*
27          * Initialize a singleton. 09/21/2014, Bing Li
28          */
29         private static ClientUI instance = new ClientUI();
30
31         public static ClientUI FACE()
32         {
33             if (instance == null)
34             {
35                 instance = new ClientUI();
36                 return instance;
37             }
38             else
39             {
40                 return instance;
41             }
45         }
46
47         /*
48          * Print the menu list on the screen. 09/21/2014, Bing Li
49          */
50         public void printMenu()
51         {
52             System.out.println(ClientMenu.MENU_HEAD);
53             System.out.println(ClientMenu.SIGN_UP);
54             System.out.println(ClientMenu.SET_WEATHER);
55             System.out.println(ClientMenu.GET_WEATHER);
56             System.out.println(ClientMenu.QUIT);
57             System.out.println(ClientMenu.MENU_TAIL);
58             System.out.println(ClientMenu.INPUT_PROMPT);
59         }
60
61         /*
62          * Send the users' option to the polling server. 09/21/2014, Bing Li
63          */
64         public void send(int option)
65         {
66             SignUpResponse signUpResponse;
67             WeatherResponse weatherResponse;
68             Weather weather;
69
70             // Check the option to interact with the polling server. 09/21/2014, Bing Li
71             switch (option)
72             {
73                 // If the SIGN_UP option is selected, send the request message to the remote
74                 // server. 09/21/2014, Bing Li
75                 case MenuOptions.SIGN_UP:
76                     signUpResponse = ClientReader.signUp(ClientConfig.USERNAME,
77                         ClientConfig.PASSWORD);
78                     System.out.println(signUpResponse.isSucceeded());
79                     break;
80
81                 // If the SET_WEATHER option is selected, send the notification to the remote
82                 // server. 02/18/2016, Bing Li
83                 case MenuOptions.SET_WEATHER:
84                     ClientEventer.NOTIFY().notifyWeather(new Weather(20.4f, "Sunshine", false,
85                         10.0f, Calendar.getInstance().getTime()));
86                     break;
87
88                 case MenuOptions.NOTIFY_TEST:
89                     ClientEventer.NOTIFY().notifyTest("Hello World!");
90                     break;
91
92                 // If the GET_WEATHER option is selected, send the request message to the
93                 // remote server. 02/18/2016, Bing Li
```

```
94                case MenuOptions.GET_WEATHER:
95                    weatherResponse = ClientReader.getWeather();
96                    weather = weatherResponse.getWeather();
97                    System.out.println("Temperature: " + weather.getTemperature());
98                    System.out.println("Forcast: " + weather.getForecast());
99                    System.out.println("Rain: " + weather.isRain());
100                   System.out.println("How much rain: " + weather.getHowMuchRain());
101                   System.out.println("Time: " + weather.getTime());
102                   break;
103
104               // If the quit option is selected, send the notification message to the remote
105               // server. 09/21/2014, Bing Li
106               case MenuOptions.QUIT:
107                   break;
108           }
109       }
110   }
```

List 3.15 The updated code of ClientUI.java

4.3 Updating the Menu

The last step at the client side is very easy even if you are a junior programmer. Actually, the rest steps have nothing to do with the mechanism to send a notification to the server. Instead, just some trivial code needs to be added to execute the distributed application.

You must notice the method, printMenu(), between Line 50~59 in List 3.15. It prints a simple menu on the screen to prompt users to interact with the server through the client. However, no prompts are displayed for your notification, TestNotification. It is easy to fix the problem. You can open the code, ClientMenu.java, as shown in List 3.16, and add one option, NOTITY_TEST, in Line 15 for your notification. After that, the code of ClientMenu.java should be updated like the one in List 3.17.

```
1     package com.greatfree.testing.client;
2
3     /*
4      * This is a simple menu for the client which is operated by a human being. 11/30/2014, Bing Li
5      */
6
7     // Created: 09/21/2014, Bing Li
8     public class ClientMenu
9     {
10        public final static String TAB = "   ";
11        public final static String MENU_HEAD = "\n========== Menu Head ==========";
12        public final static String SIGN_UP = ClientMenu.TAB + "1) Sign Up";
13        public final static String SET_WEATHER = ClientMenu.TAB + "2) Set Weather";
14        public final static String GET_WEATHER = ClientMenu.TAB + "3) Get Weather";
15        public final static String QUIT = ClientMenu.TAB + "0) Quit";
16        public final static String MENU_TAIL = "========== Menu Tail ==========\n";
17        public final static String INPUT_PROMPT = "Input an option:";
18
19        public final static String WRONG_OPTION = "Wrong option!";
20    }
```

List 3.16 The code of ClientMenu.java

```
1     package com.greatfree.testing.client;
2
3     /*
4      * This is a simple menu for the client which is operated by a human being. 11/30/2014, Bing Li
5      */
6
```

```
7   // Created: 09/21/2014, Bing Li
8   public class ClientMenu
9   {
10      public final static String TAB = "   ";
11      public final static String MENU_HEAD = "\n========== Menu Head ==========";
12      public final static String SIGN_UP = ClientMenu.TAB + "1) Sign Up";
13      public final static String SET_WEATHER = ClientMenu.TAB + "2) Set Weather";
14      public final static String GET_WEATHER = ClientMenu.TAB + "3) Get Weather";
15      public final static String NOTIFY_TEST = ClientMenu.TAB + "4) Notify Test";
16      public final static String QUIT = ClientMenu.TAB + "0) Quit";
17      public final static String MENU_TAIL = "========== Menu Tail ==========\n";
18      public final static String INPUT_PROMPT = "Input an option:";
19
20      public final static String WRONG_OPTION = "Wrong option!";
21  }
```

List 3.17 The updated code of ClientMenu.java

Although the client menu is updated, there is one trivial step to be done. Since the menu is displayed by the method, printMenu(), of the code of ClientUI.java, you ned to add the below line immediately after Line 55 of the code, ClientUI.java. This is also the final step to program a notification. Congratulations! The updated code of ClientUI.java should be exactly like the one in List 3.16.

System.out.println(ClientMenu.NOTIFY_TEST);

```
1   package com.greatfree.testing.client;
2
3   import java.util.Calendar;
4
5   import com.greatfree.testing.data.ClientConfig;
6   import com.greatfree.testing.data.Weather;
7   import com.greatfree.testing.message.SignUpResponse;
8   import com.greatfree.testing.message.WeatherResponse;
9
10  /*
11   * The class aims to print a menu list on the screen for users to interact with the client and communicate with
12   * the polling server. The menu is unique in the client such that it is implemented in the pattern of a singleton.
13   * 09/21/2014, Bing Li
14   */
15
16  // Created: 09/21/2014, Bing Li
17  public class ClientUI
18  {
19      /*
20       * Initialize. 09/21/2014, Bing Li
21       */
22      private ClientUI()
23      {
24      }
25
26      /*
27       * Initialize a singleton. 09/21/2014, Bing Li
28       */
29      private static ClientUI instance = new ClientUI();
30
31      public static ClientUI FACE()
32      {
33          if (instance == null)
34          {
35              instance = new ClientUI();
36              return instance;
37          }
38          else
39          {
40              return instance;
41          }
45      }
46
47      /*
```

```
48          * Print the menu list on the screen. 09/21/2014, Bing Li
49          */
50         public void printMenu()
51         {
52             System.out.println(ClientMenu.MENU_HEAD);
53             System.out.println(ClientMenu.SIGN_UP);
54             System.out.println(ClientMenu.SET_WEATHER);
55             System.out.println(ClientMenu.GET_WEATHER);
56             System.out.println(ClientMenu.NOTIFY_TEST);
57             System.out.println(ClientMenu.QUIT);
58             System.out.println(ClientMenu.MENU_TAIL);
59             System.out.println(ClientMenu.INPUT_PROMPT);
60         }
61 
62         /*
63          * Send the users' option to the polling server. 09/21/2014, Bing Li
64          */
65         public void send(int option)
66         {
67             SignUpResponse signUpResponse;
68             WeatherResponse weatherResponse;
69             Weather weather;
70 
71             // Check the option to interact with the polling server. 09/21/2014, Bing Li
72             switch (option)
73             {
74                 // If the SIGN_UP option is selected, send the request message to the remote
75                 // server. 09/21/2014, Bing Li
76                 case MenuOptions.SIGN_UP:
77                     signUpResponse = ClientReader.signUp(ClientConfig.USERNAME,
78                         ClientConfig.PASSWORD);
79                     System.out.println(signUpResponse.isSucceeded());
80                     break;
81 
82                 // If the SET_WEATHER option is selected, send the notification to the remote
83                 // server. 02/18/2016, Bing Li
84                 case MenuOptions.SET_WEATHER:
85                     ClientEventer.NOTIFY().notifyWeather(new Weather(20.4f, "Sunshine", false,
86                         10.0f, Calendar.getInstance().getTime()));
87                     break;
88 
89                 case MenuOptions.NOTIFY_TEST:
90                     ClientEventer.NOTIFY().notifyTest("Hello World!");
91                     break;
92 
93                 // If the GET_WEATHER option is selected, send the request message to the
94                 // remote server. 02/18/2016, Bing Li
95                 case MenuOptions.GET_WEATHER:
96                     weatherResponse = ClientReader.getWeather();
97                     weather = weatherResponse.getWeather();
98                     System.out.println("Temperature: " + weather.getTemperature());
99                     System.out.println("Forcast: " + weather.getForecast());
100                        System.out.println("Rain: " + weather.isRain());
101                    System.out.println("How much rain: " + weather.getHowMuchRain());
102                    System.out.println("Time: " + weather.getTime());
103                    break;
104 
105                // If the quit option is selected, send the notification message to the remote
106                // server. 09/21/2014, Bing Li
107                case MenuOptions.QUIT:
108                    break;
109            }
110        }
111    }
```

List 3.18 The final code of ClientUI.java

5. Testing

Until now, all of the code is updated. It is time for you to run the code to test whether the notification, TestNotification, can be sent from the client to the server or not.

5.1 Deploying Code

Before running the code, you need to deploy the code to two computers which are connected by the Internet. In my lab, I deploy them to the computers, Mum and freescape, respectively, which you have seen before. If you have only one computer, you can copy the code to separated directories and run the client and the server as two processes on the computer. Since my environment is good, I will present what I do in the case. I need to indicate that my approach must not be the most convenient one. Anyway, I get familiar with it such that I do not feel uncomfortable. If you do, you do not need to follow my solutions.

For me, the coding environment is separated with the running environment. So I need to move the source code from the coding machine to the running ones. Since in this case we need to send a notification from a client to a server, two machines must be employed. First of all, let us go to the directory that saves the source code by Eclipse, which is ./Clouds/src/com/greatfree/testing/ in my case and select the source code in the bottommost level, as shown in Figure 3.62.

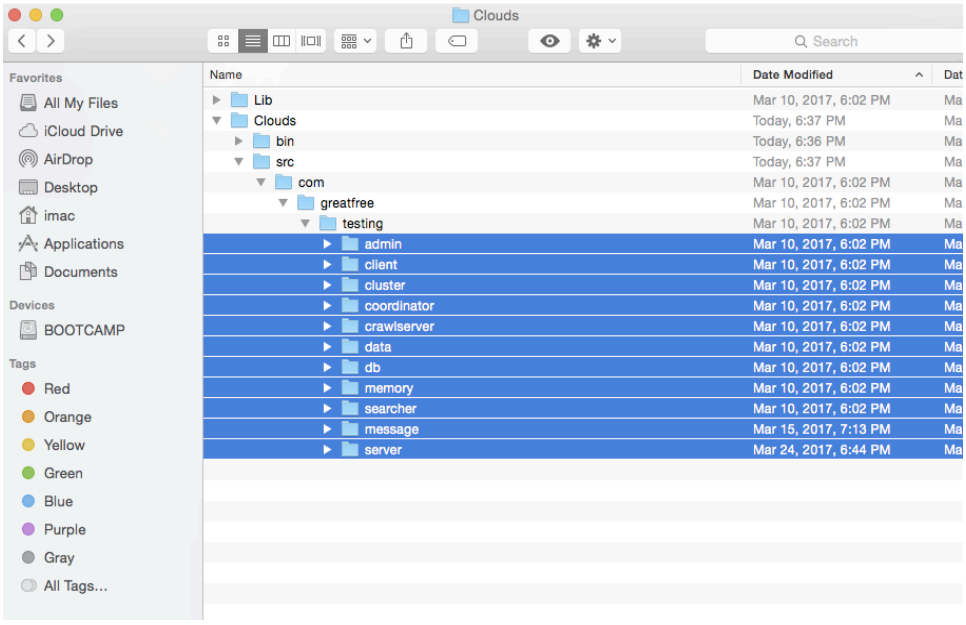


Figure 3.62 The directory that saves the source code by Eclipse

Then, you are required to prepare for an empty directory that is used to take the source code to be deployed. The reason to do that is due to the consideration that the directory of Eclipse needs to be protected with the highest priority because that is all of your effort. Any wrong operations, especially the one of deleting, should be avoided. So what you can do in the directory is only the operation of reading, i.e., copying. For that, you should copy the source code from the source directory of Eclipse to the empty one you just prepared. In my case, the empty directory is /Temp. After copying, it must contain the code, as shown in Figure 3.63.

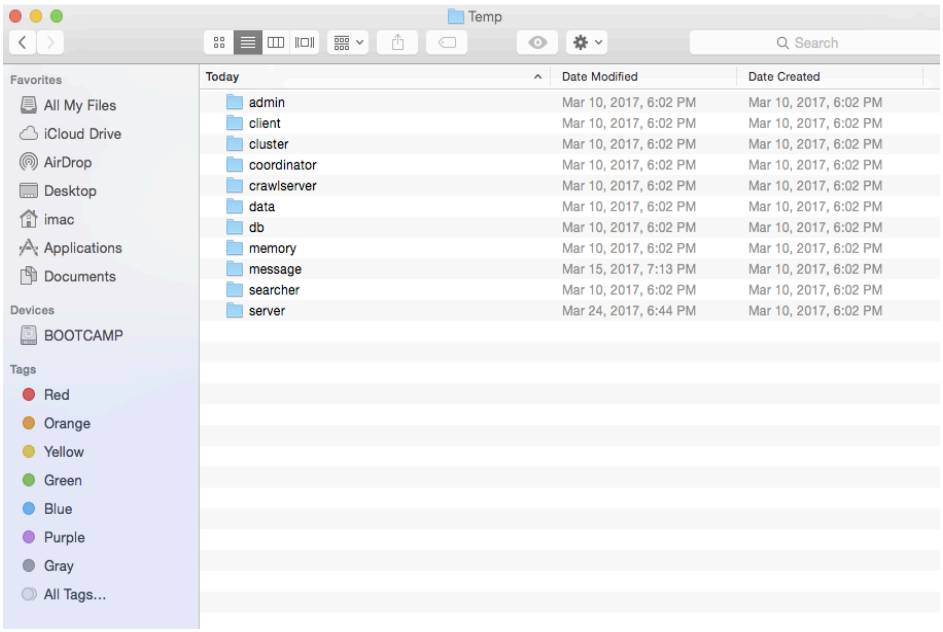


Figure 3.63 The source is copied into an empty directory, /Temp

Now you need to open a terminal and enter the directory, /Temp, that contains the copied code. You should make a package to enclose all of the code and move it to the running environment. In my case, I type the command as follows to package the code. For the package name, you can select any one you like. For me, I just simply name it g.tar, as shown in Figure 3.64.

$ tar cvf g.tar ./*

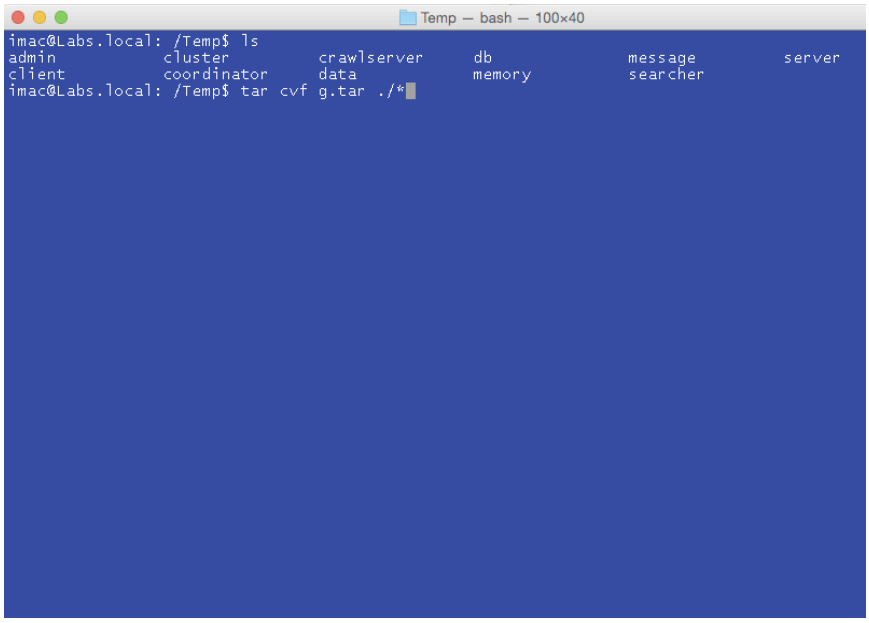


Figure 3.64 Package the source code as g.tar

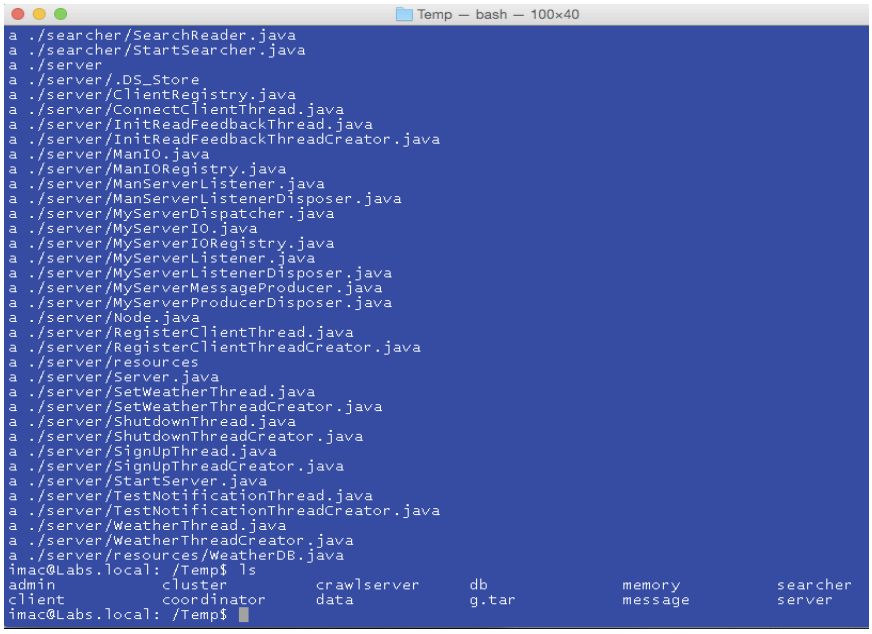


Figure 3.65 A tar file, g.tar, is created after packaging the source code

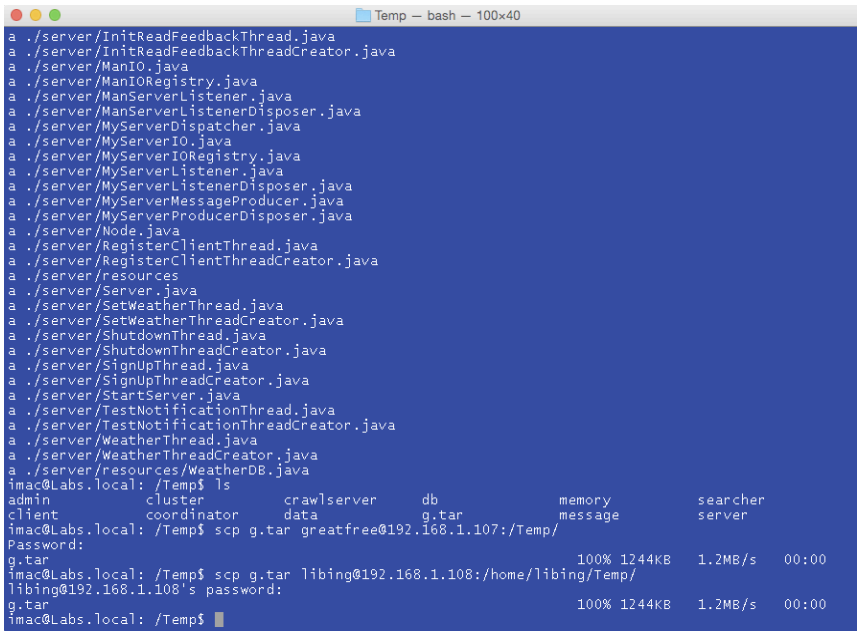


Figure 3.66 Deploy the code to the server and the client, respectively

The final step is to deploy the package, g.tar, to the running environment, the server and the client. In my case, the server's IP is 192.168.1.107 and the client's IP is 192.168.1.108, respectively. To do that, the tool, SSH, which is the one you are asked to install in Chapter 2, should be invoked.

To deploy it to the server, the following command is typed in my environment. After that, you need to enter the password to complete the deployment.

$ scp g.tar greatfree@192.168.1.107:/Temp/

To deploy it to the client, the similar command is as follows, but the IP address and the account should be different. Moreover, the destination directory is also different in my case. Actually, in my laboratory, the server is a Mac OS X machine and the client is a Ubuntu. You can also refer to Figure 3.66 for details of the deployment.

$ scp g.tar libing@192.168.1.108:/home/libing/Temp/

5.2 Working on the Server

Since the code is deployed to the server, which is named as Mum, you need to get it ready to start the server. In my case, I operate the server remotely with SSH. To do that, I need to sign in the server with the below command, which is also in Figure 3.67.

$ ssh greatfree@192.168.1.107

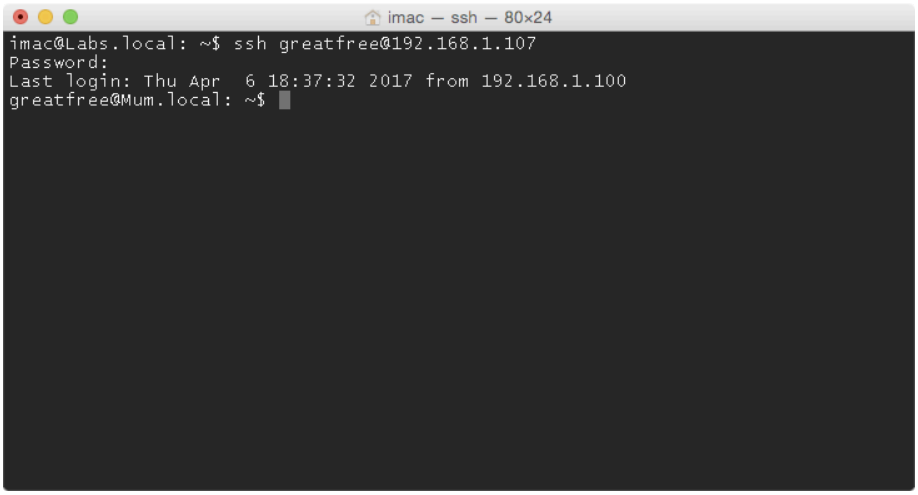


Figure 3.67 Sign in the remote server

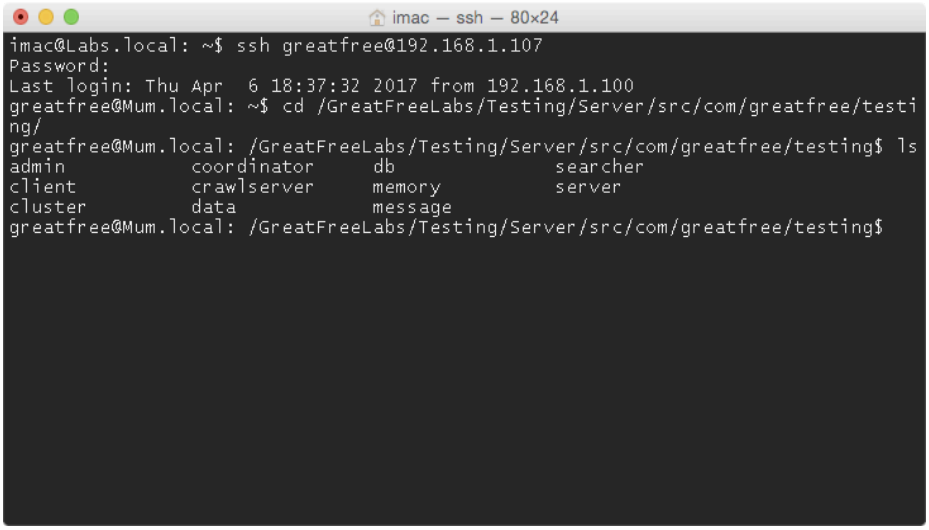


Figure 3.68 Enter into the bottommost source directory of the Ant project for the server

Next, you need to enter into the directory of the Ant project for the server, which you tested in Chapter 2. If you forget, you need to review. In my case, I type the below command to enter the bottommost directory that keeps the source code, which is shown in Figure 3.69 as well.

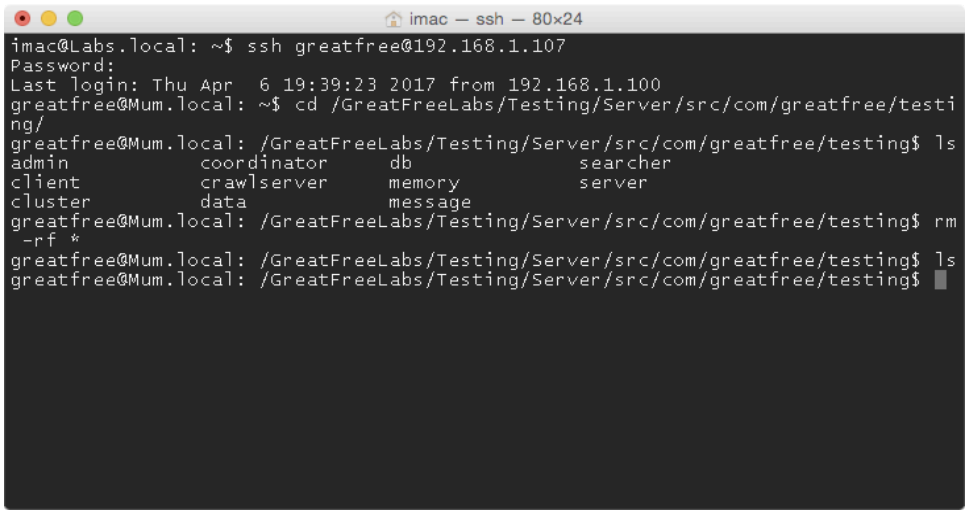


Figure 3.69 Remove the old code from the Ant project for the server

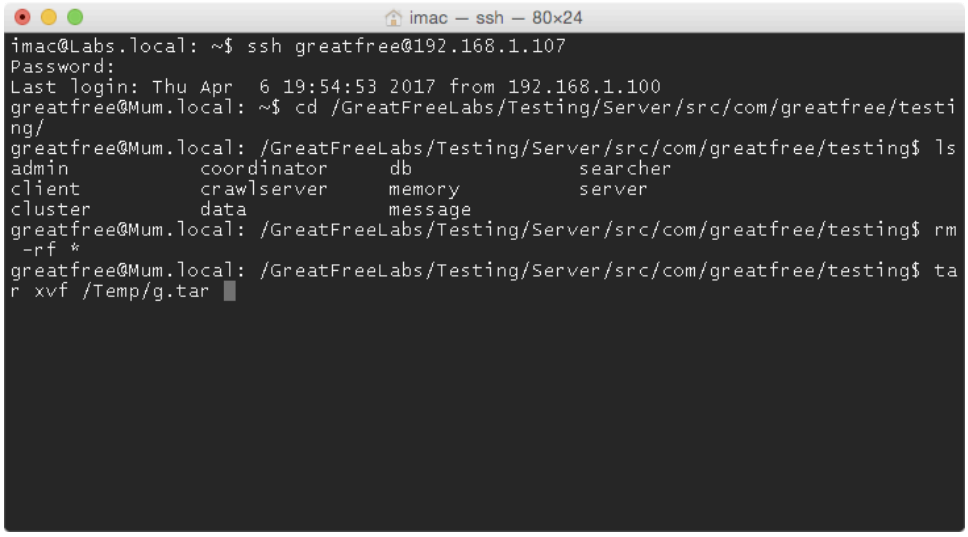


Figure 3.70 Extract the deployed code to the Ant project for the server

$ cd /GreatFreeLabs/Testing/Server/src/com/greatfree/testing/

Since the code is changed and deployed to the directory /Temp/, which is done in Section 5.1, you need to update the code in the Ant project. To do that, you need to remove the code in the project at first by typing the below command, as shown in Figure 3.69.

$ rm -rf *

Now there is no code in the Ant project for the server. You ought to extract the latest code from the directory, /Temp, which contains the package, g.tar, deployed from the coding machine. To do that, just type the below command in the current directory, as shown in Figure 3.70 and Figure 3.71. After the step, your server is ready to be started.

$ tar xvf /Temp/g.tar

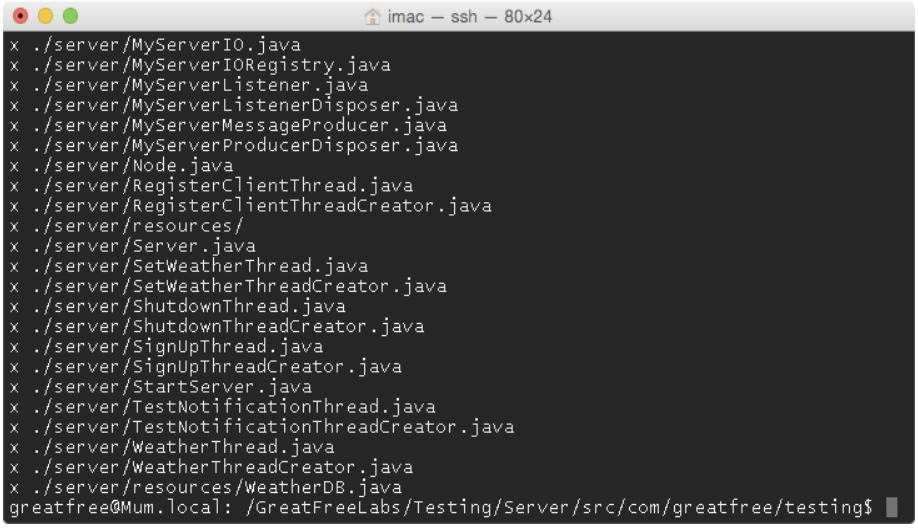


Figure 3.71 The deployed code is extracted to the directory of the Ant project for the server

5.3 Working on the Client

On the client side, you need to do almost the same thing as what we have done on the server. In my case, the difference is that the client is a Ubuntu machine and the client is located in different directory.

First, you need to sign in the remote client with SSH by typing the below command.

$ ssh libing@192.168.1.108

After that, you need to enter the bottommost source directory of the Ant project for the server by typing the following command.

$ cd ./GreatFreeLabs/Testing/Client/src/com/greatfree/testing/

Since the code is old, you are also required to remove the current one with the command as follows.

$ rm -rf *

The last step is to extract the deployed source code from the package, g.tar, which is located at the directory /home/libing/Temp/. Now the client is also ready for execution. You can refer to Figure 3.72 and Figure 3.73 for the above operations.

$ tar xvf /home/libing/Temp/g.tar

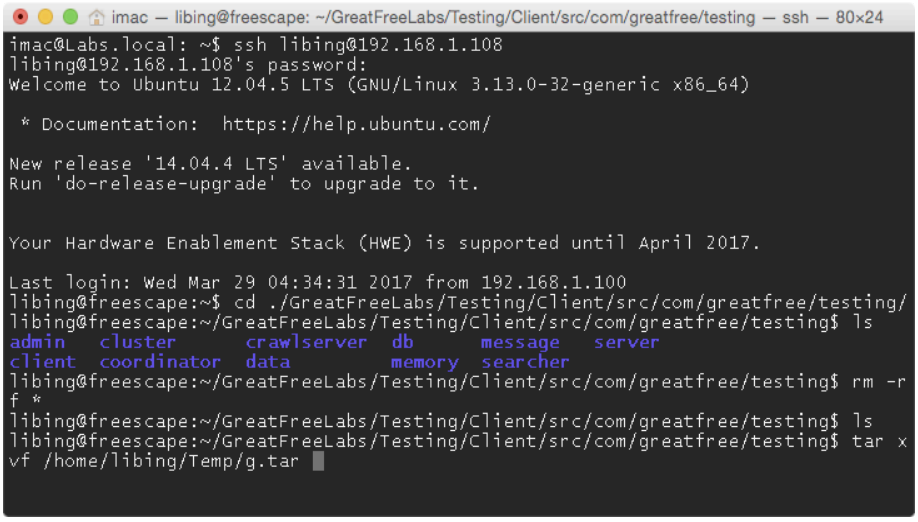


Figure 3.72 The operations on the client to get it ready

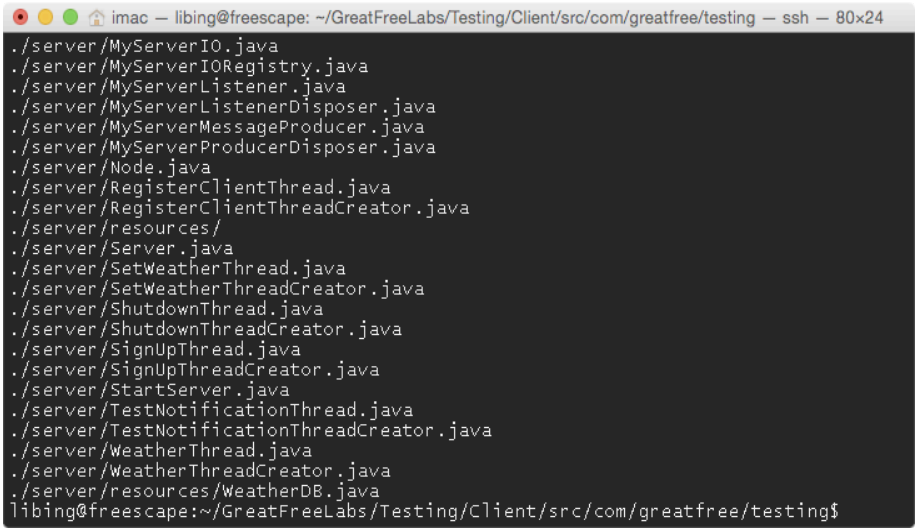


Figure 3.73 The deployed code is extracted on the Ant project for the client

5.4 Testing

The testing procedure is identical to the one we tried in Chapter 2 since the distributed environment is the same. However, since we update some code, you need to pay attention to whether your effort takes effect or not.

First, you start the server by typing the below command, ant, in the topmost directory, /GreatFreeLabs/Testing/Server/, of the Ant project for the server. If your server is started without problems, it should be exactly like the one shown in Figure 3.74.

$ ant

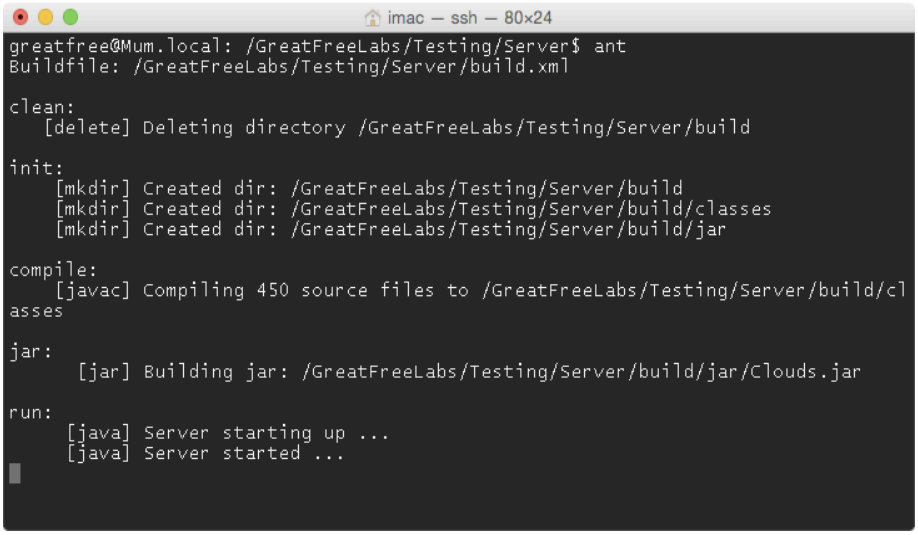


Figure 3.74 The updated server is started for testing

After the server is started, you can start the client remotely by the same command, ant. The started client should be exactly like the one shown in Figure 3.75.

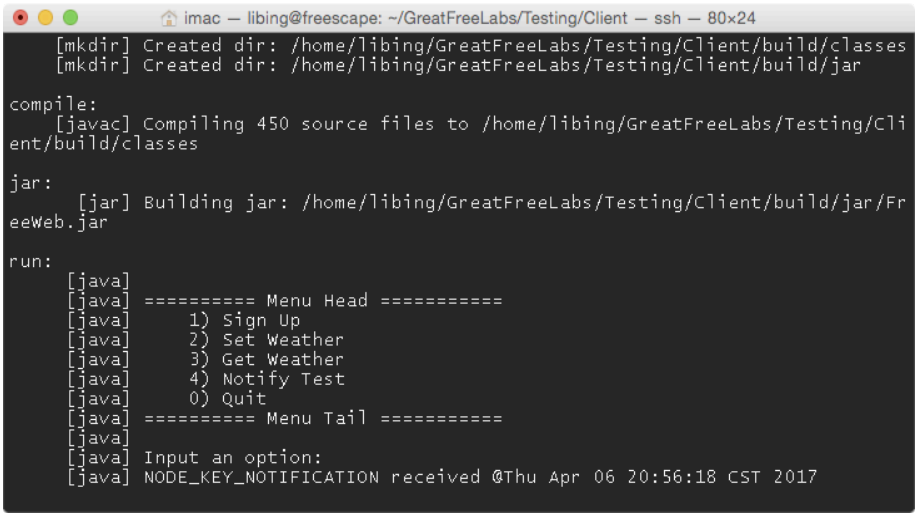


Figure 3.75 The started client with the added menu option

You should notice that a new option, Notify Test, is displayed in the menu. Just select the option, the No. 4. Then, you can find that a message, “Hello World!”, is displayed on the server side. It represents that you program a new notification successfully. You can refer the Figure 3.76 and Figure 3.77 for the above operations.

6. Summary

This chapter introduces the detailed procedure to program a notification between two distributed nodes that form a client/server distributed model. During the entire process, the programming behaviors are simplified as the simple behaviors of copy-paste-replace (CPR). Even though they do not have any knowledge about distributed systems, they can handle that since only some trivial programming skills are needed. However, it does not conflict with the methodology of GreatFree that encourages developers retain knowledge about specific computing environments they work on

because they are aware of the distributed model, the eventer at the client, the dispatcher at the server, the notification between them and even the thread that deals with the message.

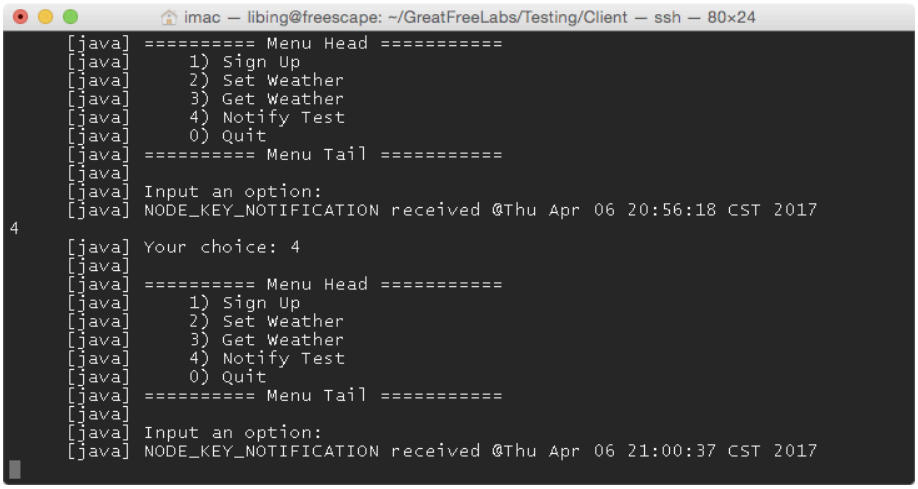


Figure 3.76 Select the new option to test your notification at the client

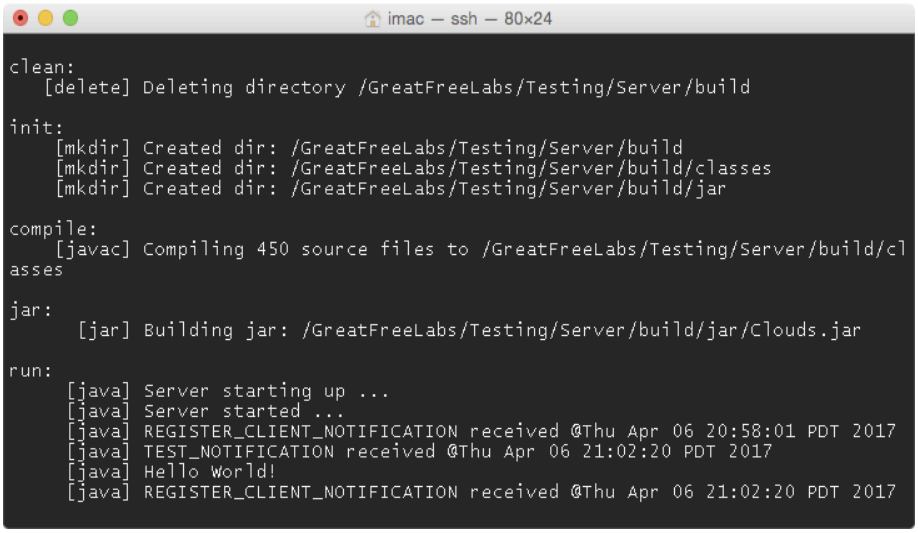


Figure 3.77 The message, “Hello World!”, of the new notification, TestNotification, is displayed on the server

References

[1] Elliotte Rusty Harold. 2014. Java Network Programming. O'Reilly Media, ISBN: 978-1-449-35767-2.

[2] Bing Li. GreatFree: The Java APIs and Idioms to Program Large-Scale Distributed Systems. International Journal of Advanced Information Technology, Volume 6, No. 1, Pages 1-22, February 2016.

[3] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. 2011. Processes, Chapter 3, Distributed Systems: Concepts and Design, the Fifth Edition.

Chapter 4 Programming Requests

Abstract

Programming requests aims to implement a solution to the requirement that a query or input message is sent from a client to a server and the client needs to wait until an answer or an output is sent from the server and received by it. The scenario is common in a distributed environment. The same as the one of notifications, it is also the fundamental mechanism to form any distributed applications. Many issues should be involved into it if it is programmed with a generic language. Thanks to the techniques of GreatFree. Similar to that of programming notifications, the procedure of programming requests are also simplified as the simple and repeatable behaviors, i.e., copy-paste-replace (CPR), which looks like editing a document in a word processor.

1. The Sample Code

To program with GreatFree using the simplified behaviors of CPR, you must find a sample code to start. You can almost follow the similar steps in Chapter 3 to program a request. Since a client/server code is provided in Chapter 2, a request, WeatherRequest, is implemented within it. It is sent from a client to the server to query the information about the weather. Correspondently, an answer message, WeatherResponse, is replied to the client by the server. That is different from the mechanism of notifications, which we implement in Chapter 3. Now you need to program a new request by yourself. The existing one, WeatherRequest, is your perfect sample to CPR for sure according to your experiences with GreatFree so far, right?

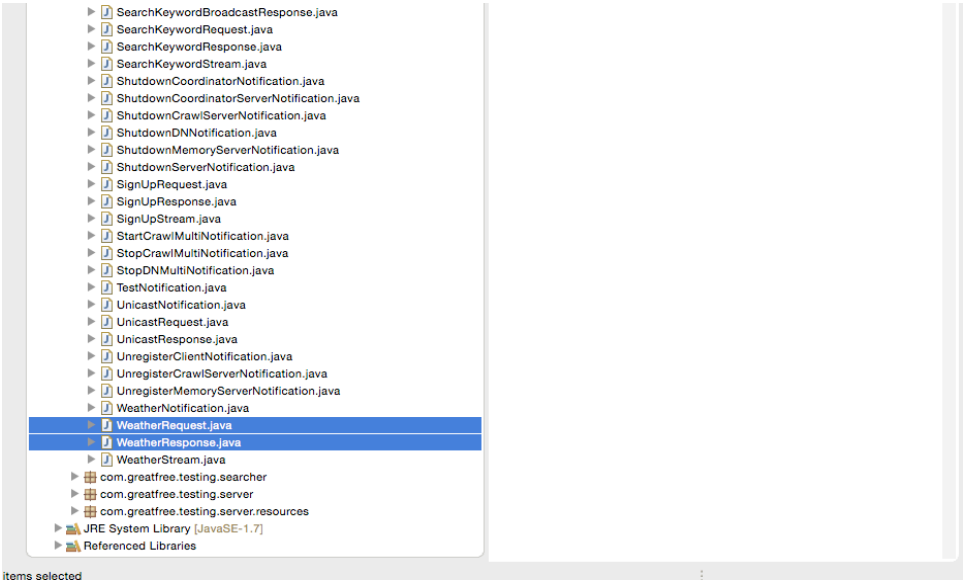


Figure 4.1 Find the sample request/response from the package com.greatfree.testing.message

Let us go back to the code in the package of com.greatfree.testing of the project of Clouds we created in Chapter 2. The same as that of notifications, the message of WeatherRequest is also kept in the package of com.greatfree.testing.message. You can find it there, as shown in Figure 4.1. If you have no idea where it is, you can try to search it with Eclipse. Anyway, that should be easy to open the sample, which is shown in List 4.1.

```
1   package com.greatfree.testing.message;
2
3   import com.greatfree.multicast.ServerMessage;
4
5   // Created: 02/15/2016, Bing Li
6   public class WeatherRequest extends ServerMessage
7   {
8       private static final long serialVersionUID = 2535049530104518280L;
9
10      /*
11       * Initialize the request. 02/15/2016, Bing Li
12       */
13      public WeatherRequest()
14      {
15          super(MessageType.WEATHER_REQUEST);
16      }
17  }
```

List 4.1 The code of WeatherRequest.java

Different from programming notifications, each request has a counterpart, a response. For the request of WeatherRequest, you can also find its counterpart, WeatherResponse, from the same package, com.greatfree.testing.message, as shown in Figure 4.1. Its code is listed in List 4.2.

```
1   package com.greatfree.testing.message;
2
3   import com.greatfree.multicast.ServerMessage;
4   import com.greatfree.testing.data.Weather;
5
6   /*
7    * The response contains an instance of Weather. It is returned to a client after it sends out a request,
8    * WeatherRequest. 02/15/2016, Bing Li
9    */
10
11  // Created: 02/15/2016, Bing Li
12  public class WeatherResponse extends ServerMessage
13  {
14      private static final long serialVersionUID = -5680951714528104272L;
15
16      // Declare the instance of Weather. 02/15/2016, Bing Li
17      private Weather weather;
18
19      /*
20       * Initialize the response. 02/15/2016, Bing Li
21       */
22      public WeatherResponse(Weather weather)
23      {
24          super(MessageType.WEATHER_RESPONSE);
25          this.weather = weather;
26      }
27
28      /*
29       * Expose the instance of Weather. 02/15/2016, Bing Li
30       */
31      public Weather getWeather()
32      {
33          return this.weather;
34      }
```

```
35  }
```

List 4.2 The code of WeatherResponse.java

Since GreatFree encourages developers to program with the simplified behaviors, copy-paste-replace (CPR), the sample code is the most important source for you to start.

2. The References and Counterparts of the Sample Request/Response

To start to program requests with CPR, you need to find the references of the sample of request, WeatherRequest, at first. We did the similar job in Section 3.1 of Chapter 3. It is easy to do that with Eclipse, as shown in Figure 4.2. You need to do that for the response, WeatherResponse, as shown in Figure 4.3.

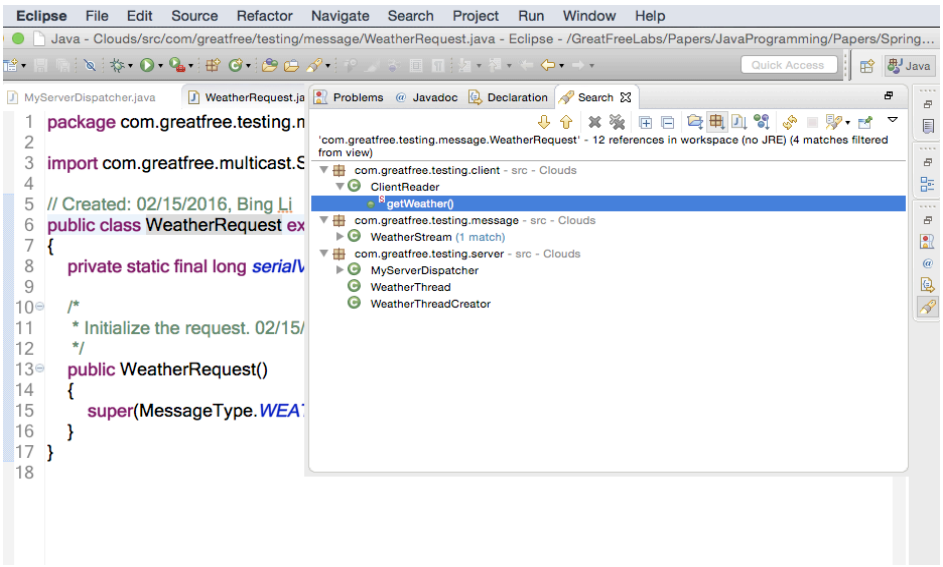


Figure 4.2 The references of the sample request, WeatherRequest

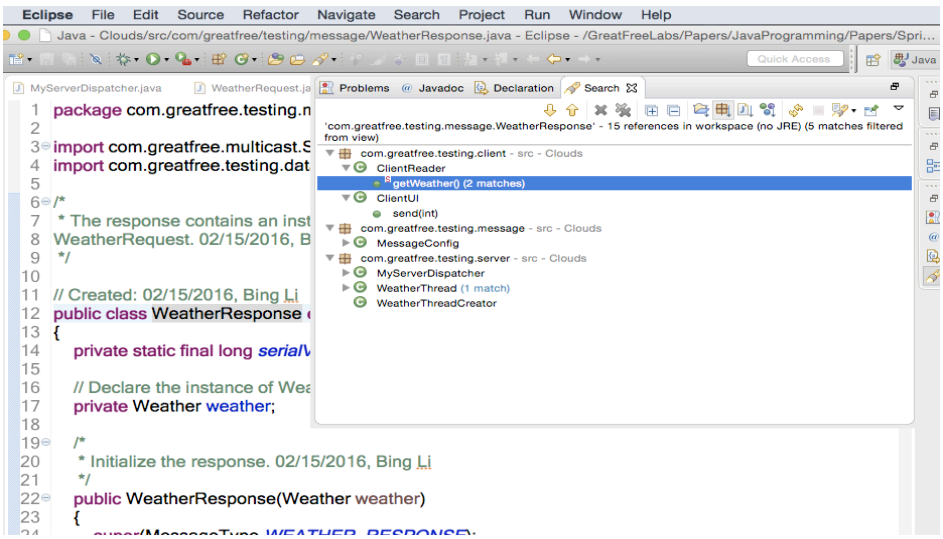


Figure 4.3 The references of the sample response, WeatherResponse

Table 4.1 lists all references of the request/response. Actually, those stuffs are the samples you need to CPR. To continue, similar to what you did in Section 3.1 of Chapter 3, you can create the table of counterparts of the samples you will attempt to CPR since it is an important guide for your further programming your requests. In short, according to the references of the request/response table, Table 4.1, you can learn which classes you need to CPR. Well, the request/response counterparts table, Table 4.2, reminds you which classes to be created and their corresponding ones to CPR.

Class	Explanation	Package	Side
ClientReader	The client side that sends requests to the server side and waits until a response is received	com.greatfree.testing.client	Client
ClientUI	The UI presenter at the client side	com.greatfree.testing.client	Client
MessageConfig	The configurations of messages	com.greatfree.testing.message	N/A
WeatherStream	The output stream that sends the response to the client	com.greatfree.testing.message	N/A
MyServerDispacher	The dispatcher that is responsible for dispatching received messages to corresponding threads such that those messages are processed concurrently	com.greatfree.testing.server	Server
WeatherThread	The thread that deals with the request of WeatherRequest	com.greatfree.testing.server	Server
WeatherThreadCreator	The thread that creates a new instance of WeatherThread in case that existing instances are too busy	com.greatfree.testing.server	Server

Table 4.1 The references of the sample request/response, WeatherRequest/WeatherResponse

Type	Classes for WeatherRequest/WeatherResponse	Classes for TestRequest/TestResponse
Request ID	MessageType.WEATHER_REQUEST	MessageType.TEST_REQUEST
Response ID	MessageType.WEATHER_RESPONSE	MessageType.TEST_RESPONSE
Request	WeatherRequest	TestRequest
Response	WeatherResponse	TestResponse
Stream	WeatherStream	TestStream
Thread	WeatherThread	TestRequestThread
Thread Creator	WeatherThreadCreator	TestRequestThreadCreator

Table 4.2 The counterparts of WeatherRequest/WeatherResponse and TestRequest/TestResponse

3. The Messages of Request/Response

Now you must have got some experiences to program with GreatFree. It is time to create the request, TestRequest, prompted by Table 4.2. Since each message in GreatFree is required to have a ID, the first step is to initialize two integral constants, TEST_REQUEST and TEST_RESPONSE, in MessageType. It is easy to do that. If you forget, you can refer to Figure 3.13 and List 3.2 in Chapter 3. The constants to be defined usually follow the immediately previous value. After the two constants are added, the code of MessageType.java is shown in List 4.3.

```
1    package com.greatfree.testing.message;
2
3    /*
4    * The class contains the identification of each type of messages. 11/28/2014, Bing Li
5    */
6
7    // Created: 09/20/2014, Bing Li
```

```
8   public class MessageType
9   {
10      public final static int NODE_KEY_NOTIFICATION = 0;
11      public final static int SIGN_UP_REQUEST = 1;
12      public final static int SIGN_UP_RESPONSE = 2;
13      public final static int ONLINE_NOTIFICATION = 5;
14      public final static int REGISTER_CLIENT_NOTIFICATION = 6;
15      public final static int UNREGISTER_CLIENT_NOTIFICATION = 7;
16      public final static int CRAWLED_LINKS_NOTIFICATION = 8;
17      public final static int REGISTER_CRAWL_SERVER_NOTIFICATION = 9;
18      public final static int UNREGISTER_CRAWL_SERVER_NOTIFICATION = 10;
19      public final static int CRAWL_LOAD_NOTIFICATION = 11;
20      public final static int START_CRAWL_MULTI_NOTIFICATION = 12;
21      public final static int SHUTDOWN_CRAWL_SERVER_NOTIFICATION = 13;
22      public final static int SHUTDOWN_MEMORY_SERVER_NOTIFICATION = 14;
23      public final static int SHUTDOWN_COORDINATOR_SERVER_NOTIFICATION = 15;
24      public final static int STOP_CRAWL_MULTI_NOTIFICATION = 16;
25      public final static int STOP_MEMORY_SERVER_NOTIFICATION = 17;
26      public final static int REGISTER_MEMORY_SERVER_NOTIFICATION = 18;
27      public final static int UNREGISTER_MEMORY_SERVER_NOTIFICATION = 19;
28      public final static int ADD_CRAWLED_LINK_NOTIFICATION = 20;
29      public final static int IS_PUBLISHER_EXISTED_REQUEST = 21;
30      public final static int IS_PUBLISHER_EXISTED_RESPONSE = 22;
31      public final static int SEARCH_KEYWORD_REQUEST = 23;
32      public final static int SEARCH_KEYWORD_RESPONSE = 24;
33      public final static int IS_PUBLISHER_EXISTED_ANYCAST_REQUEST = 25;
34      public final static int IS_PUBLISHER_EXISTED_ANYCAST_RESPONSE = 26;
35      public final static int SEARCH_KEYWORD_BROADCAST_REQUEST = 27;
36      public final static int SEARCH_KEYWORD_BROADCAST_RESPONSE = 28;
37      public final static int SHUTDOWN_REGULAR_SERVER_NOTIFICATION = 29;
38      public final static int WEATHER_NOTIFICATION = 30;
39      public final static int WEATHER_REQUEST = 31;
40      public final static int WEATHER_RESPONSE = 32;
41      public final static int CLIENT_FOR_BROADCAST_NOTIFICATION = 33;
42      public final static int CLIENT_FOR_UNICAST_NOTIFICATION = 34;
43      public final static int CLIENT_FOR_ANYCAST_NOTIFICATION = 35;
43      public final static int CLIENT_FOR_BROADCAST_REQUEST = 36;
45      public final static int CLIENT_FOR_BROADCAST_RESPONSE = 37;
46      public final static int CLIENT_FOR_UNICAST_REQUEST = 38;
47      public final static int CLIENT_FOR_UNICAST_RESPONSE = 39;
48      public final static int CLIENT_FOR_ANYCAST_REQUEST = 40;
49      public final static int CLIENT_FOR_ANYCAST_RESPONSE = 41;
50      public final static int BROADCAST_NOTIFICATION = 42;
51      public final static int UNICAST_NOTIFICATION = 43;
52      public final static int ANYCAST_NOTIFICATION = 44;
53      public final static int BROADCAST_REQUEST = 45;
54      public final static int BROADCAST_RESPONSE = 46;
55      public final static int UNICAST_REQUEST = 47;
56      public final static int UNICAST_RESPONSE = 48;
57      public final static int ANYCAST_REQUEST = 49;
58      public final static int ANYCAST_RESPONSE = 50;
59      public final static int SHUTDOWN_DN_NOTIFICATION = 51;
60      public final static int SHUTDOWN_COORDINATOR_NOTIFICATION = 52;
61      public final static int STOP_DN_NOTIFICATION = 53;
62
63      public final static int TEST_NOTIFICATION = 54;
64      public final static int TEST_REQUEST = 55;
65      public final static int TEST_RESPONSE = 56;
64  }
```

List 4.3 The code of MessageType.java after the new request/response IDs are added

After the message IDs are added as constants in MessageType, you are required to create the request, TestRequest, and the response, TestResponse. You must have already been familiar with how to do that in Eclipse. Since both of the two messages have their counterparts, i.e., WeatherRequest and WeatherResponse, respectively, you can complete the task by CPR conveniently. Moreover, because you need to test them in the C/S distributed environment, it is suggested to add some data into the messages for displaying on screen. You can recall Section 2 of Chapter 3 for the above

programming details. Anyway, the procedure is very straightforward. After that, you should have the request/response in List 4.4 and List 4.5, respectively.

```
1    package com.greatfree.testing.message;
2
3    import com.greatfree.multicast.ServerMessage;
4
5    // Created: 04/08/2017, Bing Li
6    public class TestRequest extends ServerMessage
7    {
8        private static final long serialVersionUID = -8727719427732353149L;
9
10       private String request;
11
12       public TestRequest(String request)
13       {
14           super(MessageType.TEST_REQUEST);
15           this.request = request;
16       }
17
18       public String getRequest()
19       {
20           return this.request;
21       }
22   }
```

List 4.4 The code of TestRequest.java

```
1    package com.greatfree.testing.message;
2
3    import com.greatfree.multicast.ServerMessage;
4
5    // Created: 04/08/2017, Bing Li
6    public class TestResponse extends ServerMessage
7    {
8        private static final long serialVersionUID = -4141489337639907434L;
9
10       private String response;
11
12       public TestResponse(String response)
13       {
14           super(MessageType.TEST_RESPONSE);
15           this.response = response;
16       }
17
18       public String getResponse()
19       {
20           return this.response;
21       }
22   }
```

List 4.5 The code of TestResponse.java

Another apparent difference between programming notifications and programming requests is that a new class, which derives OutMessageStream<Request>, should be created. At this moment, it is not necessary to explain it much. You just know that the response is sent by the stream. For our sample, WeatherRequest/WeatherResponse, the corresponding OutMessageStream is named WeatherStream, which is listed in List 4.6.

```
1    package com.greatfree.testing.message;
2
3    import java.io.ObjectOutputStream;
4    import java.util.concurrent.locks.Lock;
5
6    import com.greatfree.remote.OutMessageStream;
```

```
7
8       // Created: 02/15/2016, Bing Li
9       public class WeatherStream extends OutMessageStream<WeatherRequest>
10      {
11          // Initialize the instance of the request stream. 02/15/2016, Bing Li
12          public WeatherStream(ObjectOutputStream out, Lock lock, WeatherRequest message)
13          {
14              super(out, lock, message);
15          }
16      }
```

List 4.6 The code of WeatherStream.java

According to Table 4.2, As a counterpart of WeatherStream, you need to create a new class, TestStream. After CPR, the code of TestStream.java should be exactly like the one shown in List 4.7.

```
1       package com.greatfree.testing.message;
2
3       import java.io.ObjectOutputStream;
4       import java.util.concurrent.locks.Lock;
5
6       import com.greatfree.remote.OutMessageStream;
7
8       // Created: 04/08/2017, Bing Li
9       public class TestStream extends OutMessageStream<TestRequest>
10      {
11          public TestStream(ObjectOutputStream out, Lock lock, TestRequest message)
12          {
13              super(out, lock, message);
14          }
15      }
```

List 4.7 The code of TestStream.java

4. The Server Side

Now the messages, TestRequest/TestResponse and the stream, TestStream, are programmed. After that, you need to concentrate on the server side. Sorry, you should be aware of some techniques since I have no time to explain what a server is.

4.1 Creating the Thread for the Request

Following Table 4.2, the next task is to created the thread, TestRequestThread, which should be located at the server side because it is a counterpart of WeatherThread, which processes the request of WeatherRequest concurrently at the server.

The code of WeatherThread.java is shown in List 4.8 as follows. It can be found from the references of WeatherRequest. To open it, just double-click on the icon.

```
1       package com.greatfree.testing.server;
2
3       import java.io.IOException;
4
5       import com.greatfree.concurrency.RequestQueue;
6       import com.greatfree.testing.data.ServerConfig;
7       import com.greatfree.testing.message.WeatherRequest;
8       import com.greatfree.testing.message.WeatherResponse;
9       import com.greatfree.testing.message.WeatherStream;
```

```
10  import com.greatfree.testing.server.resources.WeatherDB;
11
12  /*
13   * The thread derives the RequestQueue. It receives the request of WeatherRequest and responds to
14   * users with the response of WeatherResponse. 02/15/2016, Bing Li
15   */
16
17  // Created: 02/15/2016, Bing Li
18  public class WeatherThread extends RequestQueue<WeatherRequest, WeatherStream,
19      WeatherResponse>
20  {
21      /*
22       * Initialize the thread of request queue. The value of maxTaskSize is the length of the queue to
23       * take the count of requests. 02/15/2016, Bing Li
24       */
25      public WeatherThread(int maxTaskSize)
26      {
27          super(maxTaskSize);
28      }
29
30      /*
31       * Respond users' requests concurrently. 02/15/2015, Bing Li
32       */
33      public void run()
34      {
35          // Declare the request stream. 02/15/2015, Bing Li
36          WeatherStream request;
37          // Declare the response. 02/15/2014, Bing Li
38          WeatherResponse response;
39          // The thread is shutdown when it is idle long enough or when the server is shut down.
40          // Before that, the thread keeps alive. It is necessary to detect whether it is time to end
41          // the task. 02/15/2014, Bing Li
42          while (!this.isShutdown())
43          {
44              // The loop detects whether the queue is empty or not. 02/15/2016, Bing Li
45              while (!this.isEmpty())
46              {
47                  // Dequeue a request. 02/15/2016, Bing Li
48                  request = this.getRequest();
49                  // Initialize an instance of WeatherResponse. 02/15/2016, Bing Li
50                  response = new WeatherResponse(WeatherDB.SERVER().getWeather());
51                  try
52                  {
53                      // Respond the response to the remote client. 02/15/2016, Bing Li
54                      this.respond(request.getOutStream(), request.getLock(), response);
55                  }
56                  catch (IOException e)
57                  {
58                      e.printStackTrace();
59                  }
60                  // Dispose the messages after the responding is performed. 02/15/2016, Bing Li
61                  this.disposeMessage(request, response);
62              }
63              try
64              {
65                  // Wait for some time when the queue is empty. During the period and before the thread
66                  // is killed, some new requests might be received. If so, the thread can
67                  // keep working. 02/15/2016, Bing Li
68                  this.holdOn(ServerConfig.REQUEST_THREAD_WAIT_TIME);
69              }
70              catch (InterruptedException e)
71              {
72                  e.printStackTrace();
73              }
74          }
75      }
76  }
```

List 4.8 The code of WeatherThread.java

TestRequestThread.java can be created with the behaviors of CPR conveniently. For details, you can refer to Section 3.2 of Chapter 3. In that section, a thread,

TestNotificationThread, is created following its counterpart, SetWeatherThread.java, in Table 3.2. Although the message is turned from a notification to a request, both of them are threads that handle messages concurrently at the server side. In the sense, they looks similar to each other. At least, both of them employ the pattern of DWC (Double While Concurrency). The biggest difference between them is that they have different parent classes. The one for notifications is NotificationQueue<WeatherNotification> whereas the one for requests is RequestQueue<WeatherRequest, WeatherStream, WeatherResponse>. After CPR, the code of TestRequestThread.java must be like the one in List 4.9.

```
1   package com.greatfree.testing.server;
2
3   import java.io.IOException;
4
5   import com.greatfree.concurrency.RequestQueue;
6   import com.greatfree.testing.data.ServerConfig;
7   import com.greatfree.testing.message.TestRequest;
8   import com.greatfree.testing.message.TestResponse;
9   import com.greatfree.testing.message.TestStream;
10
11  // Created: 04/09/2017, Bing Li
12  public class TestRequestThread extends RequestQueue<TestRequest, TestStream, TestResponse>
13  {
14      public TestRequestThread(int maxTaskSize)
15      {
16          super(maxTaskSize);
17      }
18
19      @Override
20      public void run()
21      {
22          // Declare the request stream. 02/15/2015, Bing Li
23          TestStream request;
24          // Declare the response. 02/15/2014, Bing Li
25          TestResponse response;
26          // The thread is shutdown when it is idle long enough or when the server is shut down.
27          // Before that, the thread keeps alive. It is necessary to detect whether it is time to end
28          // the task. 02/15/2014, Bing Li
29          while (!this.isShutdown())
30          {
31              // The loop detects whether the queue is empty or not. 02/15/2016, Bing Li
32              while (!this.isEmpty())
33              {
34                  // Dequeue a request. 02/15/2016, Bing Li
35                  request = this.getRequest();
36                  // Initialize an instance of WeatherResponse. 02/15/2016, Bing Li
37                  response = new TestResponse("response");
38                  try
39                  {
40                      // Respond the response to the remote client. 02/15/2016, Bing Li
41                      this.respond(request.getOutStream(), request.getLock(), response);
42                  }
43                  catch (IOException e)
44                  {
45                      e.printStackTrace();
46                  }
47                  // Dispose the messages after the responding is performed. 02/15/2016, Bing Li
48                  this.disposeMessage(request, response);
49              }
50              try
51              {
52                  // Wait for some time when the queue is empty. During the period and before the thread is killed,
53                  // some new requests might be received. If so, the thread can keep working. 02/15/2016, Bing Li
54                  this.holdOn(ServerConfig.REQUEST_THREAD_WAIT_TIME);
55              }
56              catch (InterruptedException e)
57              {
58                  e.printStackTrace();
60              }
61          }
62      }
```

```
63    }
```

List 4.9 The code of TestRequestThread.java

4.2 Creating the Thread Creator

The thread of TestRequestThread has a creator as well. According to Table 4.2, you need to create such a creator named, TestRequestThreadCreator. Its counterpart is WeatherThreadCreator, which is shown in List 4.10.

```
1     package com.greatfree.testing.server;
2
3     import com.greatfree.concurrency.RequestThreadCreatable;
4     import com.greatfree.testing.message.WeatherRequest;
5     import com.greatfree.testing.message.WeatherResponse;
6     import com.greatfree.testing.message.WeatherStream;
7
8     /*
9      * This is a class that creates the instance of WeatherThread. It is used by the RequestDispatcher to
10     * create the instances in a high-performance and low-cost manner. 02/15/2016, Bing Li
11     */
12
13    // Created: 02/15/2016, Bing Li
14    public class WeatherThreadCreator implements RequestThreadCreatable<WeatherRequest,
15        WeatherStream, WeatherResponse, WeatherThread>
16    {
17        @Override
18        public WeatherThread createRequestThreadInstance(int taskSize)
19        {
20            return new WeatherThread(taskSize);
21        }
22    }
```

List 4.10 The code of WeatherThreadCreator.java

Using the approach of CPR, TestRequestThreadCreator can be created conveniently as shown in List 4.11

```
1     package com.greatfree.testing.server;
2
3     import com.greatfree.concurrency.RequestThreadCreatable;
4     import com.greatfree.testing.message.TestRequest;
5     import com.greatfree.testing.message.TestResponse;
6     import com.greatfree.testing.message.TestStream;
7
8     // Created: 04/09/2017, Bing Li
9     public class TestRequestThreadCreator implements RequestThreadCreatable<TestRequest,
10        TestStream, TestResponse, TestRequestThread>
11    {
12        @Override
13        public TestRequestThread createRequestThreadInstance(int taskSize)
14        {
15            return new TestRequestThread(taskSize);
16        }
17    }
```

List 4.11 The code of TestRequestThreadCreator.java

4.3 Revising the Server Dispatcher

Till now, all of the classes in the counterpart table, Table 4.2, are created for your new request/response, i.e., TestRequest/TestResponse. What you have to do next is to revise the code of MyServerDispatcher since it is the last reference of the sample, WeatherRequest/WeatherResponse, at the server side according to Table 4.1.

Before the revision, the code of MyServerDispatcher is exactly like the one shown in List 3.9. Within it, the code you need to CPR first is Line 57 since it declares an attribute, weatherRequestDispatcher, which contains all of counterparts you need to follow in Table 4.2. Similar to what you did in Section 3.4~3.7 of Chapter 3, you can locate all of the lines you need to CPR by checking whether it contains any classes listed in Table 4.2 or defined by them. Using the approach, the lines to follow are listed in Table 4.3.

Line Number	Why to CPR
57~58	Containing counterparts
129~139	Containing counterparts
180	Defined by counterparts
255~267	Containing counterparts

Table 4.3 The lines to follow in MyServerDispatcher as shown in List 3.9

Furthermore, you can simply come up with the lines by the approach of CPR. Before moving forward to revise the real code, a one-to-one mapping table, Table 4.4, can be created to help you complete the task. According to it, you can learn how straightforward it is to program distributed applications with GreatFree.

Line to CPR	Line to Be Added
private RequestDispatcher<WeatherRequest, WeatherStream, WeatherResponse, WeatherThread, WeatherThreadCreator> weatherRequestDispatcher;	private RequestDispatcher<TestRequest, TestStream, TestResponse, TestRequestThread, TestRequestThreadCreator> testRequestDispatcher;
this.weatherRequestDispatcher = new RequestDispatcher<WeatherRequest, WeatherStream, WeatherResponse, WeatherThread, WeatherThreadCreator> (ServerConfig.REQUEST_DISPATCHER_POOL_SIZE, ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME, new WeatherThreadCreator(), ServerConfig.MAX_REQUEST_TASK_SIZE, ServerConfig.MAX_REQUEST_THREAD_SIZE, ServerConfig.REQUEST_DISPATCHER_WAIT_TIME, ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND, ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY, ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD, Scheduler.GREATFREE().getSchedulerPool());	this.testRequestDispatcher = new RequestDispatcher<TestRequest, TestStream, TestResponse, TestRequestThread, TestRequestThreadCreator> (ServerConfig.REQUEST_DISPATCHER_POOL_SIZE, ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME, new TestRequestThreadCreator(), ServerConfig.MAX_REQUEST_TASK_SIZE, ServerConfig.MAX_REQUEST_THREAD_SIZE, ServerConfig.REQUEST_DISPATCHER_WAIT_TIME, ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND, ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY, ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD, Scheduler.GREATFREE().getSchedulerPool());
this.weatherRequestDispatcher.dispose();	this.testRequestDispatcher.dispose();

case MessageType.WEATHER_REQUEST:
if (!this.weatherRequestDispatcher.isReady())
{
super.execute(this.weatherRequestDispatcher);
}
this.weatherRequestDispatcher.enqueue(new WeatherStream(message.getOutStream(), message.getLock(), (WeatherRequest)message.getMessage()));
break;
	case MessageType.TEST_REQUEST:
	if (!this.testRequestDispatcher.isReady())
	{
	super.execute(this.testRequestDispatcher);
	}
	this.testRequestDispatcher.enqueue(new TestStream(message.getOutStream(), message.getLock(), (TestRequest)message.getMessage()));
	break;

Table 4.4 The lines to CPR and the lines to be added in MyServerDispatcher

After the revision of MyServerDispatcher, the updated code is listed in List 4.12.

```
1   package com.greatfree.testing.server;
2
3   import java.util.Calendar;
4
5   import com.greatfree.concurrency.NotificationDispatcher;
6   import com.greatfree.concurrency.RequestDispatcher;
7   import com.greatfree.concurrency.Scheduler;
8   import com.greatfree.concurrency.ServerMessageDispatcher;
9   import com.greatfree.message.InitReadNotification;
```

```
10  import com.greatfree.message.SystemMessageType;
11  import com.greatfree.multicast.ServerMessage;
12  import com.greatfree.remote.OutMessageStream;
13  import com.greatfree.testing.data.ServerConfig;
14  import com.greatfree.testing.message.MessageType;
15  import com.greatfree.testing.message.RegisterClientNotification;
16  import com.greatfree.testing.message.ShutdownServerNotification;
17  import com.greatfree.testing.message.SignUpRequest;
18  import com.greatfree.testing.message.SignUpResponse;
19  import com.greatfree.testing.message.SignUpStream;
20  import com.greatfree.testing.message.TestNotification;
21  import com.greatfree.testing.message.TestRequest;
22  import com.greatfree.testing.message.TestResponse;
23  import com.greatfree.testing.message.TestStream;
24  import com.greatfree.testing.message.WeatherNotification;
25  import com.greatfree.testing.message.WeatherRequest;
26  import com.greatfree.testing.message.WeatherResponse;
27  import com.greatfree.testing.message.WeatherStream;
28
29  /*
30   * This is an implementation of ServerMessageDispatcher. It contains the concurrency mechanism
31   * to respond clients' requests and receive clients' notifications for the server. 09/20/2014, Bing Li
32   */
33
34  /*
35   * Revision Log
36   *
37   * The initialization request dispatcher is modified. When no tasks are available for some time, it needs
38   * to be shut down. 01/14/2016, Bing Li
39   *
40   */
41
42  // Created: 09/20/2014, Bing Li
43  public class MyServerDispatcher extends ServerMessageDispatcher<ServerMessage>
44  {
45      // Declare a notification dispatcher to process the registration notification concurrently. 11/04/2014, Bing Li
46      private NotificationDispatcher<RegisterClientNotification, RegisterClientThread,
47          RegisterClientThreadCreator> registerClientNotificationDispatcher;
48      // Declare a request dispatcher to respond users sign-up requests concurrently. 11/04/2014, Bing Li
49      private RequestDispatcher<SignUpRequest, SignUpStream, SignUpResponse, SignUpThread,
50          SignUpThreadCreator> signUpRequestDispatcher;
51      // Declare a notification dispatcher to set the value of Weather when an instance of
52      // WeatherNotification is received. 02/15/2016, Bing Li
53      private NotificationDispatcher<WeatherNotification, SetWeatherThread, SetWeatherThreadCreator>
54          setWeatherNotificationDispatcher;
55      private NotificationDispatcher<TestNotification, TestNotificationThread, TestNotificationThreadCreator>
56          testNotificationDispatcher;
57      // Declare a request dispatcher to respond an instance of WeatherResponse to the relevant remote client
58      // when an instance of WeatherReques is received. 02/15/2016, Bing Li
59      private RequestDispatcher<WeatherRequest, WeatherStream, WeatherResponse, WeatherThread,
60          WeatherThreadCreator> weatherRequestDispatcher;
61      private RequestDispatcher<TestRequest, TestStream, TestResponse, TestRequestThread,
62          TestRequestThreadCreator> testRequestDispatcher;
63      // Declare a notification dispatcher to deal with instances of InitReadNotification from a client concurrently
64      // such that the client can initialize its ObjectInputStream. 11/09/2014, Bing Li
65      private NotificationDispatcher<InitReadNotification, InitReadFeedbackThread,
66          InitReadFeedbackThreadCreator> initReadFeedbackNotificationDispatcher;
67      // Declare a notification dispatcher to shutdown the server when such a notification
68      // is received. 02/15/2016, Bing Li
69      private NotificationDispatcher<ShutdownServerNotification, ShutdownThread, ShutdownThreadCreator>
70      shutdownNotificationDispatcher;
71
72      /*
73       * Initialize. 09/20/2014, Bing Li
74       */
75      public MyServerDispatcher(int corePoolSize, long keepAliveTime)
76      {
77          // Set the pool size and threads' alive time. 11/04/2014, Bing Li
78          super(corePoolSize, keepAliveTime);
79
80          // Initialize the client registration notification dispatcher. 11/30/2014, Bing Li
81          this.registerClientNotificationDispatcher = new NotificationDispatcher<RegisterClientNotification,
82              RegisterClientThread, RegisterClientThreadCreator>
83              (ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
84              ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
85              new RegisterClientThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
86              ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
```

```
87              ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
88              ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
89              ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
90              ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
91              Scheduler.GREATFREE().getSchedulerPool());
92
93          // Initialize the sign up request dispatcher. 11/04/2014, Bing Li
94          this.signUpRequestDispatcher = new RequestDispatcher<SignUpRequest, SignUpStream,
95              SignUpResponse, SignUpThread, SignUpThreadCreator>
96              (ServerConfig.REQUEST_DISPATCHER_POOL_SIZE,
97              ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME,
98              new SignUpThreadCreator(), ServerConfig.MAX_REQUEST_TASK_SIZE,
99              ServerConfig.MAX_REQUEST_THREAD_SIZE,
100             ServerConfig.REQUEST_DISPATCHER_WAIT_TIME,
101             ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND,
102             ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY,
103             ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD,
104             Scheduler.GREATFREE().getSchedulerPool());
105
106         // Initialize the weather notification dispatcher. 02/15/2016, Bing Li
107         this.setWeatherNotificationDispatcher = new NotificationDispatcher<WeatherNotification,
108             SetWeatherThread, SetWeatherThreadCreator>
109             (ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
110             ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
111             new SetWeatherThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
112             ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
113             ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
114             ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
115             ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
116             ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
117             Scheduler.GREATFREE().getSchedulerPool());
118
119         this.testNotificationDispatcher = new NotificationDispatcher<TestNotification, TestNotificationThread,
120             TestNotificationThreadCreator>(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
121             ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
122             new TestNotificationThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
123             ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
124             ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
125             ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
126             ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
127             ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
128             Scheduler.GREATFREE().getSchedulerPool());
129
130         // Initialize the sign up request dispatcher. 11/04/2014, Bing Li
131         this.weatherRequestDispatcher = new RequestDispatcher<WeatherRequest, WeatherStream,
132             WeatherResponse, WeatherThread, WeatherThreadCreator>
133             (ServerConfig.REQUEST_DISPATCHER_POOL_SIZE,
134             ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME,
135             new WeatherThreadCreator(), ServerConfig.MAX_REQUEST_TASK_SIZE,
136             ServerConfig.MAX_REQUEST_THREAD_SIZE,
137             ServerConfig.REQUEST_DISPATCHER_WAIT_TIME,
138             ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND,
139             ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY,
140             ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD,
141             Scheduler.GREATFREE().getSchedulerPool());
142
143         this.testRequestDispatcher = new RequestDispatcher<TestRequest, TestStream, TestResponse,
144             TestRequestThread, TestRequestThreadCreator>
145             (ServerConfig.REQUEST_DISPATCHER_POOL_SIZE,
146             ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME,
147             new TestRequestThreadCreator(), ServerConfig.MAX_REQUEST_TASK_SIZE,
148             ServerConfig.MAX_REQUEST_THREAD_SIZE,
149             ServerConfig.REQUEST_DISPATCHER_WAIT_TIME,
150             ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND,
151             ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY,
152             ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD,
153             Scheduler.GREATFREE().getSchedulerPool());
154
155         // Initialize the read initialization notification dispatcher. 11/30/2014, Bing Li
156         this.initReadFeedbackNotificationDispatcher = new NotificationDispatcher<InitReadNotification,
157             InitReadFeedbackThread, InitReadFeedbackThreadCreator>
158             (ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
159             ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
160             new InitReadFeedbackThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
161             ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
162             ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
163             ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
```

```
164                 ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
165                 ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
166                 Scheduler.GREATFREE().getSchedulerPool());
167
168         // Initialize the shutdown notification dispatcher. 11/30/2014, Bing Li
169         this.shutdownNotificationDispatcher = new NotificationDispatcher<ShutdownServerNotification,
170             ShutdownThread, ShutdownThreadCreator>
171             (ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE,
172             ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME,
173             new ShutdownThreadCreator(), ServerConfig.MAX_NOTIFICATION_TASK_SIZE,
174             ServerConfig.MAX_NOTIFICATION_THREAD_SIZE,
175             ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME,
176             ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND,
177             ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY,
178             ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD,
179             Scheduler.GREATFREE().getSchedulerPool());
180     }
181
182     /*
183      * Shut down the server message dispatcher. 09/20/2014, Bing Li
184      */
185     public void shutdown() throws InterruptedException
186     {
187         // Dispose the register dispatcher. 01/14/2016, Bing Li
188         this.registerClientNotificationDispatcher.dispose();
189         // Dispose the sign-up dispatcher. 11/04/2014, Bing Li
190         this.signUpRequestDispatcher.dispose();
191         // Dispose the weather notification dispatcher. 02/15/2016, Bing Li
192         this.setWeatherNotificationDispatcher.dispose();
193         this.testNotificationDispatcher.dispose();
194         // Dispose the weather request dispatcher. 02/15/2016, Bing Li
195         this.weatherRequestDispatcher.dispose();
196         this.testRequestDispatcher.dispose();
197         // Dispose the dispatcher for initializing reading feedback. 11/09/2014, Bing Li
198         this.initReadFeedbackNotificationDispatcher.dispose();
199         // Dispose the dispatcher for shutdown. 11/09/2014, Bing Li
200         this.shutdownNotificationDispatcher.dispose();
201         // Shutdown the derived server dispatcher. 11/04/2014, Bing Li
202         super.shutdown();
203     }
204
205     /*
206      * Process the available messages in a concurrent way. 09/20/2014, Bing Li
207      */
208     public void consume(OutMessageStream<ServerMessage> message)
209     {
210         // Check the types of received messages. 11/09/2014, Bing Li
211         switch (message.getMessage().getType())
212         {
213             case MessageType.REGISTER_CLIENT_NOTIFICATION:
214                 System.out.println("REGISTER_CLIENT_NOTIFICATION received @" +
215                     Calendar.getInstance().getTime());
216                 // Check whether the registry notification dispatcher is ready. 01/14/2016, Bing Li
217                 if (!this.registerClientNotificationDispatcher.isReady())
218                 {
219                     // Execute the notification dispatcher as a thread. 01/14/2016, Bing Li
220                     super.execute(this.registerClientNotificationDispatcher);
221                 }
222                 // Enqueue the notification into the dispatcher for concurrent processing. 01/14/2016, Bing Li
223                 this.registerClientNotificationDispatcher.enqueue((RegisterClientNotification)
224                     message.getMessage());
225                 break;
226
227             // If the message is the one of sign-up requests. 11/09/2014, Bing Li
228             case MessageType.SIGN_UP_REQUEST:
229                 System.out.println("SIGN_UP_REQUEST received @" + Calendar.getInstance().getTime());
230                 // Check whether the sign-up dispatcher is ready. 01/14/2016, Bing Li
231                 if (!this.signUpRequestDispatcher.isReady())
232                 {
233                     // Execute the sign-up dispatcher as a thread. 01/14/2016, Bing Li
234                     super.execute(this.signUpRequestDispatcher);
235                 }
236                 // Enqueue the request into the dispatcher for concurrent responding. 11/09/2014, Bing Li
237                 this.signUpRequestDispatcher.enqueue(new SignUpStream(message.getOutStream(),
238                     message.getLock(), (SignUpRequest)message.getMessage()));
239                 break;
240
```

```
241             // If the message is the one of WeatherNotification. 02/15/2016, Bing Li
242             case MessageType.WEATHER_NOTIFICATION:
243                 System.out.println("WEATHER_NOTIFICATION received @" +
244                     Calendar.getInstance().getTime());
245                 // Check whether the weather notification dispatcher is ready or not. 02/15/2016, Bing Li
246                 if (!this.setWeatherNotificationDispatcher.isReady())
247                 {
248                     // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
249                     super.execute(this.setWeatherNotificationDispatcher);
250                 }
251                 // Enqueue the instance of WeatherNotification into the dispatcher for
252                 // concurrent processing. 02/15/2016, Bing Li
253                 this.setWeatherNotificationDispatcher.enqueue((WeatherNotification)message.getMessage());
254                 break;
255
256             case MessageType.TEST_NOTIFICATION:
257                 System.out.println("TEST_NOTIFICATION received @" + Calendar.getInstance().getTime());
258                 // Check whether the test notification dispatcher is ready or not. 02/15/2016, Bing Li
259                 if (!this.testNotificationDispatcher.isReady())
260                 {
261                     // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
262                     super.execute(this.testNotificationDispatcher);
263                 }
264                 // Enqueue the instance of TestNotification into the dispatcher for concurrent
265                 // processing. 02/15/2016, Bing Li
266                 this.testNotificationDispatcher.enqueue((TestNotification)message.getMessage());
267                 break;
268
269             // If the message is the one of weather requests. 11/09/2014, Bing Li
270             case MessageType.WEATHER_REQUEST:
271                 System.out.println("WEATHER_REQUEST received @" + Calendar.getInstance().getTime());
272                 // Check whether the weather request dispatcher is ready. 02/15/2016, Bing Li
273                 if (!this.weatherRequestDispatcher.isReady())
274                 {
275                     // Execute the weather request dispatcher concurrently. 02/15/2016, Bing Li
276                     super.execute(this.weatherRequestDispatcher);
277                 }
278                 // Enqueue the instance of WeatherRequest into the dispatcher for
279                 // concurrent responding. 02/15/2016, Bing Li
280                 this.weatherRequestDispatcher.enqueue(new WeatherStream(message.getOutStream(),
281                     message.getLock(), (WeatherRequest)message.getMessage()));
282                 break;
283
284             case MessageType.TEST_REQUEST:
285                 System.out.println("TEST_REQUEST received @" + Calendar.getInstance().getTime());
286                 // Check whether the test request dispatcher is ready. 02/15/2016, Bing Li
287                 if (!this.testRequestDispatcher.isReady())
288                 {
289                     // Execute the test request dispatcher concurrently. 02/15/2016, Bing Li
290                     super.execute(this.testRequestDispatcher);
291                 }
292                 // Enqueue the instance of WeatherRequest into the dispatcher for
293                 // concurrent responding. 02/15/2016, Bing Li
294                 this.testRequestDispatcher.enqueue(new TestStream(message.getOutStream(),
295                     message.getLock(), (TestRequest)message.getMessage()));
296                 break;
297
298             // If the message is the one of initializing notification. 11/09/2014, Bing Li
299             case SystemMessageType.INIT_READ_NOTIFICATION:
300                 System.out.println("INIT_READ_NOTIFICATION received @" +
301                     Calendar.getInstance().getTime());
302                 // Check whether the reading initialization dispatcher is ready or not. 01/14/2016, Bing Li
303                 if (!this.initReadFeedbackNotificationDispatcher.isReady())
304                 {
305                     // Execute the notification dispatcher as a thread. 01/14/2016, Bing Li
306                     super.execute(this.initReadFeedbackNotificationDispatcher);
307                 }
308                 // Enqueue the notification into the dispatcher for concurrent processing. 11/09/2014, Bing Li
309                 this.initReadFeedbackNotificationDispatcher.enqueue((InitReadNotification)
310                     message.getMessage());
311                 break;
312
313             case MessageType.SHUTDOWN_REGULAR_SERVER_NOTIFICATION:
314                 System.out.println("SHUTDOWN_REGULAR_SERVER_NOTIFICATION received @" +
315                     Calendar.getInstance().getTime());
316                 // Check whether the shutdown dispatcher is ready or not. 01/14/2016, Bing Li
317                 if (!this.shutdownNotificationDispatcher.isReady())
```

```
318                 {
319                     // Execute the notification dispatcher as a thread. 01/14/2016, Bing Li
320                     super.execute(this.shutdownNotificationDispatcher);
321                 }
322                 // Enqueue the notification into the dispatcher for concurrent processing. 11/09/2014, Bing Li
323                 this.shutdownNotificationDispatcher.enqueue((ShutdownServerNotification)
324                     message.getMessage());
325                 break;
326             }
327         }
328     }
```

List 4.12 The code of MyServerDispatcher.java after it is revised for the new request/response

5. The Client Side

After the server is accomplished, you need to concentrate on the client side. According to Table 4.1, two references, i.e., ClientReader and ClientUI, on the client are related to your new request and response.

The code of ClientReader.java is listed in List 4.13.

```
1   package com.greatfree.testing.client;
2
3   import java.io.IOException;
4
5   import com.greatfree.exceptions.RemoteReadException;
6   import com.greatfree.remote.RemoteReader;
7   import com.greatfree.testing.data.ServerConfig;
8   import com.greatfree.testing.message.ClientForAnycastRequest;
9   import com.greatfree.testing.message.ClientForAnycastResponse;
10  import com.greatfree.testing.message.ClientForBroadcastRequest;
11  import com.greatfree.testing.message.ClientForBroadcastResponse;
12  import com.greatfree.testing.message.ClientForUnicastRequest;
13  import com.greatfree.testing.message.ClientForUnicastResponse;
14  import com.greatfree.testing.message.MessageConfig;
15  import com.greatfree.testing.message.SignUpRequest;
16  import com.greatfree.testing.message.SignUpResponse;
17  import com.greatfree.testing.message.WeatherRequest;
18  import com.greatfree.testing.message.WeatherResponse;
19  import com.greatfree.util.NodeID;
20
21  /*
22   * The class wraps the class, RemoteReader, to send requests to the remote server and wait until relevant
23   * responses are received. 09/22/2014, Bing Li
24   */
25
26  // Created: 09/21/2014, Bing Li
27  public class ClientReader
28  {
29      /*
30       * Send the request of SignUpRequest to the remote server and wait for the response,
31       * SignUpResponse. 09/22/2014, Bing Li
32       */
33      public static SignUpResponse signUp(String userName, String password)
34      {
35          try
36          {
37              return (SignUpResponse)(RemoteReader.REMOTE().read(NodeID.DISTRIBUTED().getKey(),
38                  ServerConfig.SERVER_IP, ServerConfig.SERVER_PORT,
39                  new SignUpRequest(userName, password)));
40          }
41          catch (ClassNotFoundException | RemoteReadException | IOException e)
42          {
43              // When the connection gets something wrong, a RemoteReadException and other
44              // exceptions must be raised. 09/22/2014, Bing Li
45              e.printStackTrace();
```

```
46              }
47              // When reading gets something wrong, a null response is returned. 09/22/2014, Bing Li
48              return MessageConfig.NO_SIGN_UP_RESPONSE;
49          }
50
51          /*
52           * Send the request of WeatherRequest to the remote server and wait for the response,
53           * WeatherResponse. 02/18/2016, Bing Li
54           */
55          public static WeatherResponse getWeather()
56          {
57              try
58              {
59                  return (WeatherResponse)(RemoteReader.REMOTE().read(NodeID.DISTRIBUTED().getKey(),
60                      ServerConfig.SERVER_IP, ServerConfig.SERVER_PORT, new WeatherRequest()));
61              }
62              catch (ClassNotFoundException | RemoteReadException | IOException e)
63              {
64                  // When the connection gets something wrong, a RemoteReadException and other exceptions
65                  // must be raised. 09/22/2014, Bing Li
66                  e.printStackTrace();
67              }
68              return MessageConfig.NO_WEATHER_RESPONSE;
69          }
70
71          /*
72           * Send the request of ClientForBroadcastRequest to the remote server and wait for the response,
73           * ClientForBroadcastResponse. 09/22/2014, Bing Li
74           */
75          public static ClientForBroadcastResponse requestBroadcastly(String message)
76          {
77              try
78              {
79                  return (ClientForBroadcastResponse)(RemoteReader.REMOTE().read
80                      (NodeID.DISTRIBUTED().getKey(), ServerConfig.SERVER_IP,
81                      ServerConfig.SERVER_PORT, new ClientForBroadcastRequest(message)));
82              }
83              catch (ClassNotFoundException | RemoteReadException | IOException e)
84              {
85                  // When the connection gets something wrong, a RemoteReadException and other exceptions
86                  // must be raised. 09/22/2014, Bing Li
87                  e.printStackTrace();
88              }
89              // When reading gets something wrong, a null response is returned. 09/22/2014, Bing Li
90              return MessageConfig.NO_CLIENT_FOR_BROADCAST_RESPONSE;
91          }
92
93          /*
94           * Send the request of ClientForUnicastRequest to the remote server and wait for the response,
95           * ClientForUnicastResponse. 09/22/2014, Bing Li
96           */
97          public static ClientForUnicastResponse requestUnicastly(String message)
98          {
99              try
100             {
101                 return (ClientForUnicastResponse)(RemoteReader.REMOTE().read
102                     (NodeID.DISTRIBUTED().getKey(), ServerConfig.SERVER_IP,
103                     ServerConfig.SERVER_PORT, new ClientForUnicastRequest(message)));
104             }
105             catch (ClassNotFoundException | RemoteReadException | IOException e)
106             {
107                 // When the connection gets something wrong, a RemoteReadException and other exceptions
108                 // must be raised. 09/22/2014, Bing Li
109                 e.printStackTrace();
110             }
111             // When reading gets something wrong, a null response is returned. 09/22/2014, Bing Li
112             return MessageConfig.NO_CLIENT_FOR_UNICAST_RESPONSE;
113         }
114
115         /*
116          * Send the request of ClientForUnicastRequest to the remote server and wait for the response,
117          * ClientForUnicastResponse. 09/22/2014, Bing Li
118          */
119         public static ClientForAnycastResponse requestAnycastly(String message)
120         {
121             try
122             {
```

```
123             return (ClientForAnycastResponse)(RemoteReader.REMOTE().read
124                 (NodeID.DISTRIBUTED().getKey(), ServerConfig.SERVER_IP,
125                 ServerConfig.SERVER_PORT, new ClientForAnycastRequest(message)));
126         }
127         catch (ClassNotFoundException | RemoteReadException | IOException e)
128         {
129             // When the connection gets something wrong, a RemoteReadException and other exceptions
130             // must be raised. 09/22/2014, Bing Li
131             e.printStackTrace();
132         }
133         // When reading gets something wrong, a null response is returned. 09/22/2014, Bing Li
134         return MessageConfig.NO_CLIENT_FOR_ANYCAST_RESPONSE;
135     }
136 }
```

List 4.13 The code of ClientReader.java

You must notice that Line 55~69 contains the counterparts in Table 4.2. That indicates that you need to CPR the lines. After that, the code of ClientReader.java is updated as shown in Figure 4.4.

```
51    /*
52     * Send the request of WeatherRequest to the remote server and wait for the response,
53  WeatherResponse. 02/18/2016, Bing Li
54     */
55    public static WeatherResponse getWeather()
56    {
57        try
58        {
59            return (WeatherResponse)(RemoteReader.REMOTE().read(NodeID.DISTRIBUTED().getKey(),
60  ServerConfig.SERVER_IP, ServerConfig.SERVER_PORT, new WeatherRequest()));
61        }
62        catch (ClassNotFoundException | RemoteReadException | IOException e)
63        {
64            // When the connection gets something wrong, a RemoteReadException and other exceptions
65  must be raised. 09/22/2014, Bing Li
66            e.printStackTrace();
67        }
68        return MessageConfig.NO_WEATHER_RESPONSE;
69    }
70
71    public static TestResponse getWeather(String request)
72    {
73        try
74        {
75            return (TestResponse)(RemoteReader.REMOTE().read(NodeID.DISTRIBUTED().getKey(),
76  ServerConfig.SERVER_IP, ServerConfig.SERVER_PORT, new TestRequest(request)));
77        }
78        catch (ClassNotFoundException | RemoteReadException | IOException e)
79        {
80            // When the connection gets something wrong, a RemoteReadException and other exceptions
81  must be raised. 09/22/2014, Bing Li
82            e.printStackTrace();
83        }
84        return MessageConfig.NO_WEATHER_RESPONSE;
85    }
86
```

Figure 4.4 The code of ClientReader.java after CPR

However, one compilation error exists in Line 84 of the code in Figure 4.4 after the behaviors of CPR. You can open it by pressing F3 when the cursor is on the class of MessageConfig, which is also one reference listed in Table 4.1. The code is shown in List 4.14. The code contains the null value constants for existing responses. The error is caused because your new response is not defined in it. To correct it, just add one line for your response, TestResponse. The updated code of MessageConfig.java is listed in List 4.15.

```
1   package com.greatfree.testing.message;
2
3   import com.greatfree.multicast.ServerMessage;
4
5   /*
```

```
6       * The class contains all of the constants related to messages transmitted between remote
7       * nodes. 09/21/2014, Bing Li
8       */
9
10      // Created: 09/21/2014, Bing Li
11      public class MessageConfig
12      {
13          public final static ServerMessage NO_MESSAGE = null;
14          public final static SignUpResponse NO_SIGN_UP_RESPONSE = null;
15          public final static WeatherResponse NO_WEATHER_RESPONSE = null;
16          public final static ClientForBroadcastResponse
17              NO_CLIENT_FOR_BROADCAST_RESPONSE = null;
18          public final static ClientForUnicastResponse NO_CLIENT_FOR_UNICAST_RESPONSE = null;
19          public final static ClientForAnycastResponse NO_CLIENT_FOR_ANYCAST_RESPONSE = null;
20      }
```

List 4.14 The code of MessageConfig.java

```
1       package com.greatfree.testing.message;
2
3       import com.greatfree.multicast.ServerMessage;
4
5       /*
6        * The class contains all of the constants related to messages transmitted between remote
7        * nodes. 09/21/2014, Bing Li
8        */
9
10      // Created: 09/21/2014, Bing Li
11      public class MessageConfig
12      {
13          public final static ServerMessage NO_MESSAGE = null;
14          public final static SignUpResponse NO_SIGN_UP_RESPONSE = null;
15          public final static WeatherResponse NO_WEATHER_RESPONSE = null;
16          public final static ClientForBroadcastResponse
17              NO_CLIENT_FOR_BROADCAST_RESPONSE = null;
18          public final static ClientForUnicastResponse NO_CLIENT_FOR_UNICAST_RESPONSE = null;
19          public final static ClientForAnycastResponse NO_CLIENT_FOR_ANYCAST_RESPONSE = null;
20          public final static TestResponse NO_TEST_RESPONSE = null;
21      }
```

List 4.15 The updated code of MessageConfig.java

After the new constant, NO_TEST_RESPONSE, is added, you need to replace NO_WEATHER_RESPONSE in Line 84 with it. Then the code of ClentReader.java should not have any errors and warnings. The updated code is shown in List 4.16.

```
1       package com.greatfree.testing.client;
2
3       import java.io.IOException;
4
5       import com.greatfree.exceptions.RemoteReadException;
6       import com.greatfree.remote.RemoteReader;
7       import com.greatfree.testing.data.ServerConfig;
8       import com.greatfree.testing.message.ClientForAnycastRequest;
9       import com.greatfree.testing.message.ClientForAnycastResponse;
10      import com.greatfree.testing.message.ClientForBroadcastRequest;
11      import com.greatfree.testing.message.ClientForBroadcastResponse;
12      import com.greatfree.testing.message.ClientForUnicastRequest;
13      import com.greatfree.testing.message.ClientForUnicastResponse;
14      import com.greatfree.testing.message.MessageConfig;
15      import com.greatfree.testing.message.SignUpRequest;
16      import com.greatfree.testing.message.SignUpResponse;
17      import com.greatfree.testing.message.WeatherRequest;
18      import com.greatfree.testing.message.WeatherResponse;
19      import com.greatfree.util.NodeID;
20
21      /*
22       * The class wraps the class, RemoteReader, to send requests to the remote server and wait until relevant
23       * responses are received. 09/22/2014, Bing Li
```

```
24      */
25
26      // Created: 09/21/2014, Bing Li
27      public class ClientReader
28      {
29          /*
30           * Send the request of SignUpRequest to the remote server and wait for the response,
31           * SignUpResponse. 09/22/2014, Bing Li
32           */
33          public static SignUpResponse signUp(String userName, String password)
34          {
35              try
36              {
37                  return (SignUpResponse)(RemoteReader.REMOTE().read(NodeID.DISTRIBUTED().getKey(),
38                      ServerConfig.SERVER_IP, ServerConfig.SERVER_PORT,
39                      new SignUpRequest(userName, password)));
40              }
41              catch (ClassNotFoundException | RemoteReadException | IOException e)
42              {
43                  // When the connection gets something wrong, a RemoteReadException and other
44                  // exceptions must be raised. 09/22/2014, Bing Li
45                  e.printStackTrace();
46              }
47              // When reading gets something wrong, a null response is returned. 09/22/2014, Bing Li
48              return MessageConfig.NO_SIGN_UP_RESPONSE;
49          }
50
51          /*
52           * Send the request of WeatherRequest to the remote server and wait for the response,
53           * WeatherResponse. 02/18/2016, Bing Li
54           */
55          public static WeatherResponse getWeather()
56          {
57              try
58              {
59                  return (WeatherResponse)(RemoteReader.REMOTE().read(NodeID.DISTRIBUTED().getKey(),
60                      ServerConfig.SERVER_IP, ServerConfig.SERVER_PORT, new WeatherRequest()));
61              }
62              catch (ClassNotFoundException | RemoteReadException | IOException e)
63              {
64                  // When the connection gets something wrong, a RemoteReadException and other exceptions
65                  // must be raised. 09/22/2014, Bing Li
66                  e.printStackTrace();
67              }
68              return MessageConfig.NO_WEATHER_RESPONSE;
69          }
70
71          public static TestResponse getResponse(String request)
72          {
73              try
74              {
75                  return (TestResponse)(RemoteReader.REMOTE().read(NodeID.DISTRIBUTED().getKey(),
76                      ServerConfig.SERVER_IP, ServerConfig.SERVER_PORT, new TestRequest(request)));
77              }
78              catch (ClassNotFoundException | RemoteReadException | IOException e)
79              {
80                  // When the connection gets something wrong, a RemoteReadException and other exceptions
81                  // must be raised. 09/22/2014, Bing Li
82                  e.printStackTrace();
83              }
84              return MessageConfig.NO_TEST_RESPONSE;
85          }
86
87          /*
88           * Send the request of ClientForBroadcastRequest to the remote server and wait for the response,
89           * ClientForBroadcastResponse. 09/22/2014, Bing Li
90           */
91          public static ClientForBroadcastResponse requestBroadcastly(String message)
92          {
93              try
94              {
95                  return (ClientForBroadcastResponse)(RemoteReader.REMOTE().read
96                      (NodeID.DISTRIBUTED().getKey(), ServerConfig.SERVER_IP,
97                      ServerConfig.SERVER_PORT, new ClientForBroadcastRequest(message)));
98              }
99              catch (ClassNotFoundException | RemoteReadException | IOException e)
100             {
```

```
101                 // When the connection gets something wrong, a RemoteReadException and other exceptions
102                 // must be raised. 09/22/2014, Bing Li
103                 e.printStackTrace();
104             }
105             // When reading gets something wrong, a null response is returned. 09/22/2014, Bing Li
106             return MessageConfig.NO_CLIENT_FOR_BROADCAST_RESPONSE;
107         }
108
109         /*
110          * Send the request of ClientForUnicastRequest to the remote server and wait for the response,
111          * ClientForUnicastResponse. 09/22/2014, Bing Li
112          */
113         public static ClientForUnicastResponse requestUnicastly(String message)
114         {
115             try
116             {
117                 return (ClientForUnicastResponse)(RemoteReader.REMOTE().read
118                     (NodeID.DISTRIBUTED().getKey(), ServerConfig.SERVER_IP,
119                     ServerConfig.SERVER_PORT, new ClientForUnicastRequest(message)));
120             }
121             catch (ClassNotFoundException | RemoteReadException | IOException e)
122             {
123                 // When the connection gets something wrong, a RemoteReadException and other exceptions
124                 // must be raised. 09/22/2014, Bing Li
125                 e.printStackTrace();
126             }
127             // When reading gets something wrong, a null response is returned. 09/22/2014, Bing Li
128             return MessageConfig.NO_CLIENT_FOR_UNICAST_RESPONSE;
129         }
130
131         /*
132          * Send the request of ClientForUnicastRequest to the remote server and wait for the response,
133          * ClientForUnicastResponse. 09/22/2014, Bing Li
134          */
135         public static ClientForAnycastResponse requestAnycastly(String message)
136         {
137             try
138             {
139                 return (ClientForAnycastResponse)(RemoteReader.REMOTE().read
140                     (NodeID.DISTRIBUTED().getKey(), ServerConfig.SERVER_IP,
141                     ServerConfig.SERVER_PORT, new ClientForAnycastRequest(message)));
142             }
143             catch (ClassNotFoundException | RemoteReadException | IOException e)
144             {
145                 // When the connection gets something wrong, a RemoteReadException and other exceptions
146                 // must be raised. 09/22/2014, Bing Li
147                 e.printStackTrace();
148             }
149             // When reading gets something wrong, a null response is returned. 09/22/2014, Bing Li
150             return MessageConfig.NO_CLIENT_FOR_ANYCAST_RESPONSE;
151         }
152     }
```

List 4.16 The updated code of ClientReader.java

Until now, the kernel code to handle your new request/response is done. After that, what you need to do is similar to what you have done in Section 4.2 of Chapter 3. That is, you should update the code that can invoke ClientReader since you just add one new method between Line 71 and Line 85 in List 4.16. Since the procedure is trivial, the detailed steps are omitted. The code to be updated includes ClientUI.java, MenuOptions and ClientMenu, which are identical to the ones updated in Section 4.2 of Chapter 3. The final updated code is as shown in List 4.17, List 4.18 and List 4.19, respectively.

```
1   package com.greatfree.testing.client;
2
3   import java.util.Calendar;
4
5   import com.greatfree.testing.data.ClientConfig;
```

```
6   import com.greatfree.testing.data.Weather;
7   import com.greatfree.testing.message.SignUpResponse;
8   import com.greatfree.testing.message.WeatherResponse;
9
10  /*
11   * The class aims to print a menu list on the screen for users to interact with the client and communicate with
12   * the polling server. The menu is unique in the client such that it is implemented in the pattern of a singleton.
13   * 09/21/2014, Bing Li
14   */
15
16  // Created: 09/21/2014, Bing Li
17  public class ClientUI
18  {
19      /*
20       * Initialize. 09/21/2014, Bing Li
21       */
22      private ClientUI()
23      {
24      }
25
26      /*
27       * Initialize a singleton. 09/21/2014, Bing Li
28       */
29      private static ClientUI instance = new ClientUI();
30
31      public static ClientUI FACE()
32      {
33          if (instance == null)
34          {
35              instance = new ClientUI();
36              return instance;
37          }
38          else
39          {
40              return instance;
41          }
45      }
46
47      /*
48       * Print the menu list on the screen. 09/21/2014, Bing Li
49       */
50      public void printMenu()
51      {
52          System.out.println(ClientMenu.MENU_HEAD);
53          System.out.println(ClientMenu.SIGN_UP);
54          System.out.println(ClientMenu.SET_WEATHER);
55          System.out.println(ClientMenu.GET_WEATHER);
56          System.out.println(ClientMenu.NOTIFY_TEST);
57          System.out.println(ClientMenu.REQUEST_TEST);
58          System.out.println(ClientMenu.QUIT);
59          System.out.println(ClientMenu.MENU_TAIL);
60          System.out.println(ClientMenu.INPUT_PROMPT);
61      }
62
63      /*
64       * Send the users' option to the polling server. 09/21/2014, Bing Li
65       */
66      public void send(int option)
67      {
68          SignUpResponse signUpResponse;
69          WeatherResponse weatherResponse;
70          Weather weather;
71          TestResponse testResponse;
72
73          // Check the option to interact with the polling server. 09/21/2014, Bing Li
74          switch (option)
75          {
76              // If the SIGN_UP option is selected, send the request message to the remote
77              // server. 09/21/2014, Bing Li
78              case MenuOptions.SIGN_UP:
79                  signUpResponse = ClientReader.signUp(ClientConfig.USERNAME,
80                      ClientConfig.PASSWORD);
81                  System.out.println(signUpResponse.isSucceeded());
82                  break;
83
84              // If the SET_WEATHER option is selected, send the notification to the remote
85              // server. 02/18/2016, Bing Li
```

```
86                case MenuOptions.SET_WEATHER:
87                    ClientEventer.NOTIFY().notifyWeather(new Weather(20.4f, "Sunshine", false,
88                        10.0f, Calendar.getInstance().getTime()));
89                    break;
90
91                case MenuOptions.NOTIFY_TEST:
92                    ClientEventer.NOTIFY().notifyTest("Hello World!");
93                    break;
94
95                // If the GET_WEATHER option is selected, send the request message to the
96                // remote server. 02/18/2016, Bing Li
97                case MenuOptions.GET_WEATHER:
98                    weatherResponse = ClientReader.getWeather();
99                    weather = weatherResponse.getWeather();
100                   System.out.println("Temperature: " + weather.getTemperature());
101                   System.out.println("Forcast: " + weather.getForecast());
102                   System.out.println("Rain: " + weather.isRain());
103                   System.out.println("How much rain: " + weather.getHowMuchRain());
104                   System.out.println("Time: " + weather.getTime());
105                   break;
106
107               case MenuOptions.REQUEST_TEST:
108                   testResponse = (TestResponse)ClientReader.getResponse("request");
109                   System.out.println(testResponse.getResponse());
110                   break;
111
112               // If the quit option is selected, send the notification message to the remote
113               // server. 09/21/2014, Bing Li
114               case MenuOptions.QUIT:
115                   break;
116           }
117       }
118   }
```

List 4.17 The updated code of ClientUI.java

```
1   package com.greatfree.testing.client;
2
3   /*
4    * The class contains all of constants of menu options. 09/22/2014, Bing Li
5    */
6
7   // Created: 09/21/2014, Bing Li
8   public class MenuOptions
9   {
10      public final static int NO_OPTION = -1;
11      public final static int SIGN_UP = 1;
12      public final static int SET_WEATHER = 2;
13      public final static int GET_WEATHER = 3;
14      public final static int NOTIFY_TEST = 4;
15      public final static int REQUEST_TEST = 5;
16      public final static int QUIT = 0;
17  }
```

List 4.18 The updated code of MenuOptions.java

```
1   package com.greatfree.testing.client;
2
3   /*
4    * This is a simple menu for the client which is operated by a human being. 11/30/2014, Bing Li
5    */
6
7   // Created: 09/21/2014, Bing Li
8   public class ClientMenu
9   {
10      public final static String TAB = "   ";
11      public final static String MENU_HEAD = "\n========== Menu Head ==========";
12      public final static String SIGN_UP = ClientMenu.TAB + "1) Sign Up";
13      public final static String SET_WEATHER = ClientMenu.TAB + "2) Set Weather";
14      public final static String GET_WEATHER = ClientMenu.TAB + "3) Get Weather";
15      public final static String NOTIFY_TEST = ClientMenu.TAB + "4) Notify Test";
```

```
16        public final static String REQUEST_TEST = ClientMenu.TAB + "5) Request Test";
17        public final static String QUIT = ClientMenu.TAB + "0) Quit";
18        public final static String MENU_TAIL = "========== Menu Tail ==========\n";
19        public final static String INPUT_PROMPT = "Input an option:";
20
21        public final static String WRONG_OPTION = "Wrong option!";
22    }
```

List 4.19 The updated code of ClientMenu.java

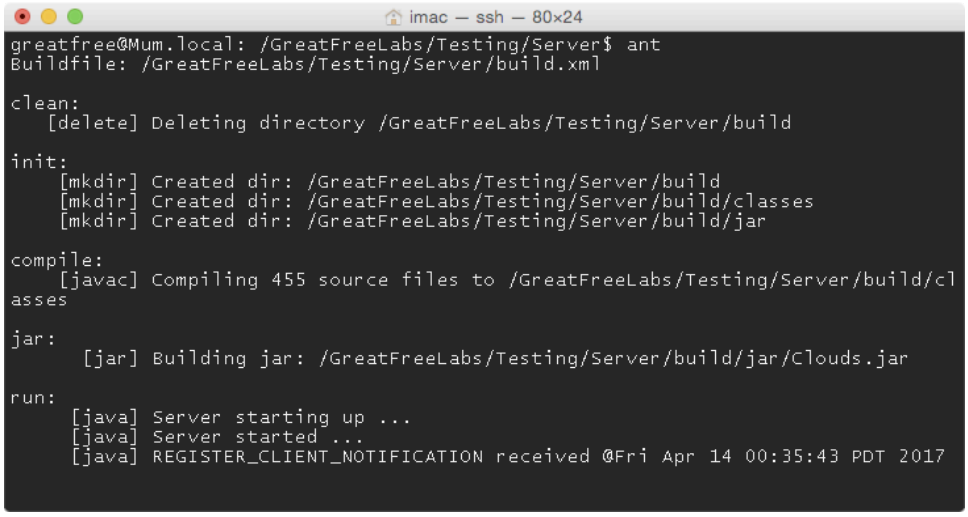


Figure 4.5 The terminal of the server side after running initially

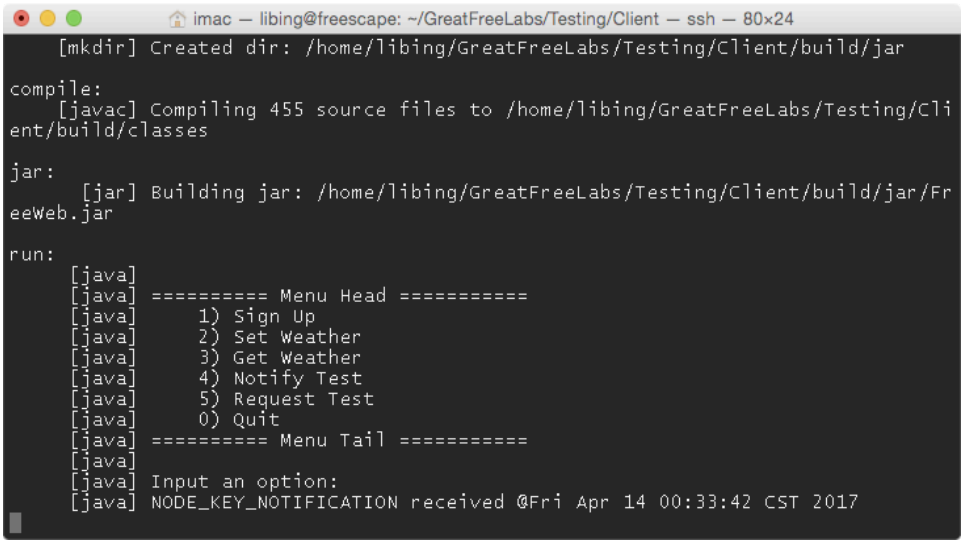


Figure 4.6 The terminal of the client side after running initially

6. Testing

Now testing can be performed since the code for your new request/response, i.e., TestRequest/TestResponse, is added and updated. To save time, the steps to deploy the code is omitted. You can refer to Section 5.1~5.3 of Chapter 3.

After running both of the server and the client, the terminals are like the ones of Figure 4.5 and Figure 4.6, respectively. One new option, “Request Test”, is presented in the menu of the client.

When you select the new option, “Request Test”, at the client side, the updated terminals for both of the server and the client are shown in Figure 4.7 and Figure 4.8, respectively. If your testing results are identical to what are shown in the figures of the section, it means your new request/response, TestRequest/TestResponse, is succeeded to be added.

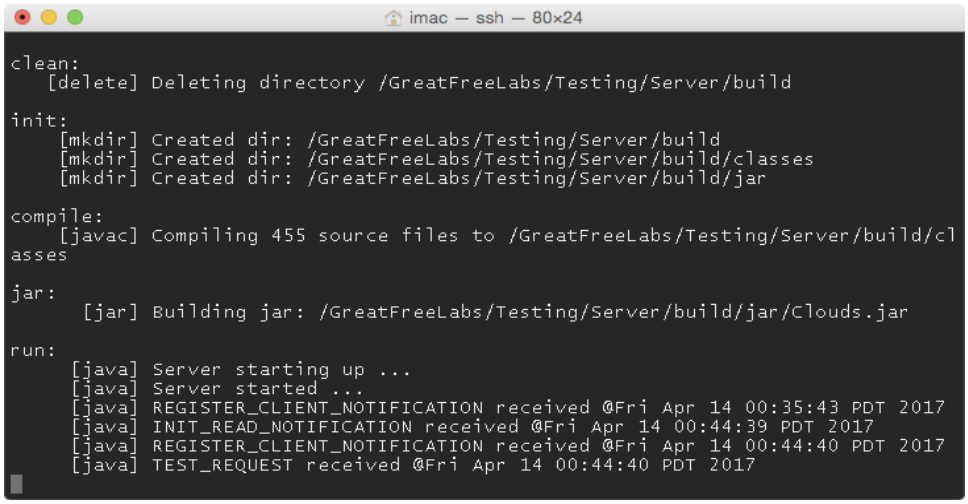


Figure 4.7 The server side after the new request is received

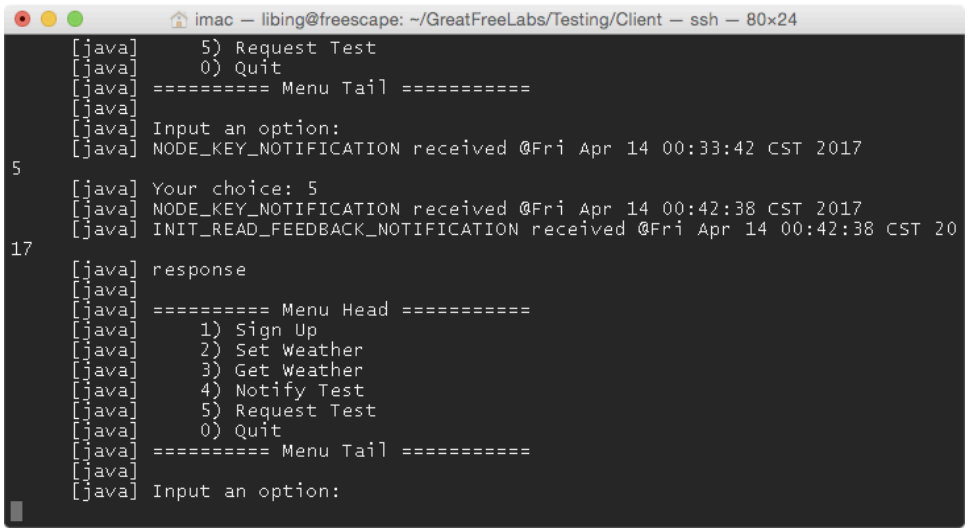


Figure 4.8 The client side after the new request is sent and the response is received

7. Summary

In the chapter, you learn how to program requests/responses with GreatFree. As one of two important interaction approaches between two distributed nodes, requesting/responding is critical for any distributed applications. If you do that with a generic programming language, like Java SE, the effort is too huge to tolerate. If you

do the same thing with a framework, you are hidden from the distributed computing environment as well as the underlying code. Additionally, you have to rely on the framework for one particular instance of distributed environments only. With GreatFree, you are free from tedious effort to achieve the goal rapidly with simplified behaviors, CPR. Moreover, you must notice that you have to work on the open source itself directly to add and update your code. That is completely different from the traditional open source environment, in which you have to work within an environment that is independent of the open source. That is why GreatFree is called the unwrapped open source. Finally, GreatFree provides you with a general programming tool that adapts to any distributed environments. In the case of requesting/responding programming, the skills are needed for any applications rather than any particular one.

Chapter 5 Programming with CSServer

Abstract

This chapter starts to discuss the issue of programming clouds with some important GreatFree APIs and idioms. Although you have learned some new ideas and skills to program distributed systems with GreatFree by programming notifications and requests, you must feel weird why you have to start up with some existing code, like the one that processing weather information when you CPR (copy-paste-replace). You must have a question whether you can program a distributed system from scratch. This is the generic approach for almost any programming tools or languages. The answer is yes. Similar to others, GreatFree provides developers with some convenient APIs and idioms for developers to implement their applications without any other strange code, like what you see in the previous chapters. I need to emphasize again that it does not mean what you have learned is insignificant. On the contrary, those experiences are unique for GreatFree and you will use the skills repeatedly when programming with GreatFree in the future chapters. One of the most important goals of GreatFree is to reach the balance between lowering the development effort and unwrapping distributed techniques. At least, you must agree with me that it is easy to program with GreatFree. In the previous two chapters, a programmer, even though he knows nothing about distributed techniques, can come up with a distributed application using notifications and requests/responses. What they need to know is only a little bit object oriented knowledge. The effort is lowered sufficiently, right?

What about another side, i.e., unwrapping the distributed techniques, of the balance? From this chapter, you will learn something about the goal. At least, you have to start programming based on thinking in the distributed way. That is different from what you have learned from the previous chapters. Why does GreatFree intend to do that? Simply speaking, GreatFree attempts to protect developers from losing their skills or knowledge. A programmer always works with a tool that hides lower techniques form him must become degenerated on his proficiency. In addition, GreatFree aims to assign the full control of the system to developers rather than presenting only high level applications. With their skills and such a flexibility, developers can not only implement applications on well-known distributed environments but also design systems that are brand new. Moreover, GreatFree is a generic tool in terms of developing any distributed systems instead of the one focusing on a specific domain. Without the support of developers' skills and knowledge, it is tough to achieve the goal. In contrast, if the effort is lowered, it does not matter that developers know more about a system. That is why it is critical to reach the balance between the development effort and the developers' skills. Finally, the degree of those skills and knowledge is not high when programming GreatFree. It is impossible for a person to keep their proficiency for ever. Humans always forget something. Fortunately, the rough knowledge is sufficient to program with GreatFree. So, do not scare. Let us start the new topic, CSServer, and enjoy the programming procedure.

1. CSServer

CSServer is the implementation of the server side in the distributed model of the client/server (C/S) [1]. Developers can employ it in their applications if they decide to choose their distributed architecture as the C/S one. The C/S must be the most frequently adopted model in practice. It works roughly in this way. What the client can do only is to send a notification or a request to the server actively without the allowance from the server. In contrast, the server can only receive notifications or respond requests from clients. When programming notifications and requests, we just work on the model although we do not specific it explicitly. This chapter will present a new API, CSServer, to you rather than working with existing code. Then, you can learn the model deeply and exclude others from your system.

1.1 The Chatting System

Before starting to talk about CSServer, let us learn something about the chatting system. If you are required to program an application over the Internet for users to chat, do you know how to do that? From now on, we will discuss something about techniques. But I believe, you love that.

The first question to implement a chatting system is how to transmit data from one computer that a user holds to another one where the chatting partner is located. It seems to be a stupid question since it is always accomplished over the Internet by the technology of TCP/IP (Transmission Control Protocol/Internet Protocol) [2]. However, in practice, it is not a question that can be ignored. Although each computer over the Internet can be connected by TCP/IP, some non-technical issues affect the final situation. For example, it is possible that two computers have no real IPs [2]. It always happens when a computer is connected to the Internet through LAN (Local Area Network) [2] rather than an ISP (Internet Server Provider) [2]. If no real IPs are available, it is difficult to transmit data between them directly although it is not impossible. Another more frequently factor is that it is not feasible to let any computers be accessed without the allowance of their users, especially over the Internet, because of the consideration of malicious behaviors. Hence, you can see that smart phones over 4G cannot be connected through TCP/IP. Although PCs are different since it is possible for them to have real IPs over the Internet, it is seldom for most PCs to hold a static IP. Usually, only dynamic IPs are available to them. That means that they cannot find each other next time even though their IPs are real. For the above problems, the only convenient situation in which two computers can transmit data to one another is when both of them is located in the same LAN. It does not always happen in practice, unfortunately.

Thus, in the real world, to simplify the problem that seems to be trivial, the chatting service provider has to set up a server, called the chatting server, to assist the interaction among computers, including wired devices and wireless ones. The server is erected with a static IP address over the Internet such that it can be accessed by any computers from any corners of the globe wherever they are located within a LAN or supported by ISP. Then, what happen next if chatting needs to be performed between them? Rather than the implementation to transmit data from one computer to one another, each user sends his chatting messages to the centralized server the chatting

service provider sets up. It can be done conveniently since they can communicate with each other without any barriers. Thereafter, the chatting server keeps all of the received messages and manages those messages in accordance with their respective partners. Until now, their partners have not received those messages even though they are the potential receivers. To achieve the goal, those computers which the partners hold must send polling requests to the chatting server periodically such that those chatting messages can be sent to the partners as polling responses. Unless that happens, the chatting procedure is done.

According to the above explanation, you learn one solution to the chatting system in practice. Actually, each current popular chatting applications are implemented in that way. In such a system, you must be aware of the importance of the chatting server. It helps those users find each other as well as behaves as post offices to distribute messages for them.

Constructors	Explanation
CSServer(CSServer.CSServerBuilder<Dispatcher> builder)	This is the constructor implemented by the builder pattern
CSServer(int port, int listenerCount, int listenerThreadPoolSize, long listenerThreadKeepAliveTime, Dispatcher dispatcher)	The standard constructor for an instance of CSServer
CSServer(int port, int listenerCount, int listenerThreadPoolSize, long listenerThreadKeepAliveTime, Dispatcher dispatcher, boolean isPeer)	The CSServer is also the parent class of Peer. If so, its socket should not be initialized when the instance of Peer is initialized. It should be delayed until an idle port is assigned to it by the registry server

Table 5.1 The constructors of CSServer

Methods	Explanation
String getID()	Get the ID of CSServer
int getPort()	Get the port of CSServer
void setID(String key)	Set the ID of CSServer
void setPort()	Set the port of CSServer: when no port conflicts, the default port is used to initialize the socket of CSServer
void setPort(int port)	Set the port of CSServer: when port conflicts exist, the newly assigned port is set to initialize the socket of CSServer
void start()	The method that starts the CSServer
void stop()	The method that stops the CSServer

Table 5.2 The methods of CSServer

1.2 CSServer

Now it is time for us to present the concept of CSServer to you. In fact, the chatting server in the above system works exactly in the same way as CSServer. In other words, the chatting server is a classic CSServer and that solution to chatting is a typical application scenario of CSServer. Additionally, as we have to face with some technologies from now on, I need to indicate that the chatting system based on CSServer contains some important components of a distributed system although it is simple. For that, it is necessary for us to program it with our existing APIs and idioms. Through that, you learn you can program distributed systems or even clouds from scratch using GreatFree. Moreover, your background of distributed technologies must be improved when implementing the application. That helps you program more difficult applications as well. Finally, you will learn GreatFree more deeply. As a generic programming for any distributed environments, it provides developers with a

bunch of APIs and idioms such that you can implement your systems rapidly with visible distributed techniques instead of the traditional technique-transparent development environment.

Table 5.1 and Table 5.2 list the APIs of CSServer. According to it, you find that the APIs of CSServer is very limited. That is true since CSServer is an underlying distributed component to receive remote messages. It seems that you cannot operate it directly to do something. Actually, the trick happens in the particular structure. You must notice that CSServer is implemented in the form of generics and its placeholder is a subclass of ServerDispatcher. In other words, developers do not need to interact with CSServer directly. Instead, most effort is spent on the subclass of ServerDispatcher when programming with CSServer.

2. Programming the Chatting Server with CSServer

Different from the previous chapters, I am sorry to tell you we have to talk about technical issues. In addition, this section contains many list of code that you start from scratch and you did not see the code previously either.

2.1 SP – The Pattern of Starting Point: Starting the Chatting Server

Since you need to start to program from scratch, let us see the code of List 5.1, which presents the starting point of the chatting server.

```
1   package com.greatfree.chat.server;
2
3   import java.io.IOException;
4
5   import com.greatfree.exceptions.RemoteReadException;
6   import com.greatfree.testing.data.ServerConfig;
7   import com.greatfree.util.TerminateSignal;
8
9   /*
10   * This is the starting point of the chatting server. 05/20/2017, Bing Li
11   */
12
13  // Created: 04/22/2017, Bing Li
14  public class StartChatServer
15  {
16      public static void main(String[] args)
17      {
18          System.out.println("Chatting server starting up ...");
19
20          try
21          {
22              ChatServer.CS().start();
23          }
24          catch (IOException | ClassNotFoundException | RemoteReadException e)
25          {
26              e.printStackTrace();
27          }
28
29          System.out.println("Chatting server started ...");
30
31          // After the server is started, the loop check whether the flag of terminating is set. If the terminating
32          // flag is true, the process is ended. Otherwise, the process keeps running. 08/22/2014, Bing Li
33          while (!TerminateSignal.SIGNAL().isTerminated())
34          {
35              try
36              {
```

```
37                  // If the terminating flag is false, it is required to sleep for some time. Otherwise, it might cause
38                  // the high CPU usage. 08/22/2014, Bing Li
39                  Thread.sleep(ServerConfig.TERMINATE_SLEEP);
40              }
41              catch (InterruptedException e)
42              {
43                  e.printStackTrace();
44              }
45          }
46      }
47  }
```

List 5.1 The code of StartChatServer.java

When you attempt to program with GreatFree, you should start like the code in List 5.1. I strive to reduce your workload to develop distributed systems or the cloud, which is a popular term. However, Coding is coding. You need to know the technique and you need to be a proficient engineer. And, you ought to be proud of the title, right? If so, you should not scare about List 5.1 although you never see it in the previous chapters.

The primary component in List 5.1 is the singleton, ChatServer.CS.start(), as shown in Line 22. Before opening it, you should learn why it is required to design a singleton in the case. You can refer to the books of design patterns [3]. Briefly speaking, it is unreasonable that there are many instances of chatting servers in a process. So a single one is enough.

Pattern: SP (Starting Point)		
Class	StartChatServer	
Singleton	Yes	
Method	void main(String[] args)	
Enclosed Pattern	MS (Main Server)	ChatServer
Employed API	TerminalSignal	

Table 5.3 The primary components of the pattern of SP

Line 33~45 presents one while-loop, which is terminated only if the signal, TerminalSignal.SIGNAL().isTerminate() is set to true. As you see, the signal is a singleton as well. It is not set to true unless the chatting server is stopped by the administrator. Before it is stopped, the loop is always executed. Inside the loop, one sleeping statement, Line 39, is enclosed. It denotes that the process checks whether the chatting server is shut down or not periodically.

In addition, one of the most important usages of the signal is to terminate long term running threads. In a distributed system, it is possible some tasks are executed for a long time. It is necessary to terminate them when the process is shut down. To do that, the signal, TerminalSignal, is inserted into those code such that it can be checked periodically during the long-running procedure. As a singleton implementation, the goal can be achieved easily. It solves the problem that a process cannot be shut down even though its administrator decides to do that.

Almost each server you program with GreatFree is started like List 5.1 Therefore, it is reasonable to name it although I dislike that. For example, the pattern can be called,

the Starting Pointing (SP) of a server. Table 5.3 summarizes the primary components of the pattern of SP.

2.2 MS – The Pattern of Main Server: The Chatting Server

Let us move forward by opening the singleton, ChatServer, in Line 22. The code is also highly patterned. To be convenient, it is named as the Main Server (MS). The pattern is usually implemented as a singleton. One instance of CSServer at least should be defined and invoked. Moreover, two methods are required, i.e., stop() and start(). Inside the two methods, besides initializing and disposing the instances of CSServer, some application level classes should be initialized and disposed. Table 5.4 presents the major components of the pattern of MS.

Pattern: MS (Main Server)		
Class	ChatServer	
Singleton	Yes	
Method	void start()	
	void stop()	
Enclosed Pattern	SD (Server Dispatcher)	CSServer<ChatServerDispatcher>
		CSServer<ChatManServerDispatcher>
Employed API	CSServer<Dispatcher extends ServerDispatcher>	

Table 5.4 The primary components of the pattern of MS for the chatting server

List 5.2 shows the complete code. If you learn it well, you must agree with my opinions of programming distributed systems and enjoy the entire procedure.

```
1   package com.greatfree.chat.server;
2
3   import java.io.IOException;
4
5   import com.greatfree.chat.ChatConfig;
6   import com.greatfree.chat.PrivateChatSessions;
7   import com.greatfree.exceptions.RemoteReadException;
8   import com.greatfree.server.CSServer;
9   import com.greatfree.testing.data.ServerConfig;
10  import com.greatfree.util.TerminateSignal;
11
12  /*
13   * This is the chatting server that is located at the center of the chatting system. Clients need to poll
14   * it periodically to interact with each other instantly. 04/30/2017, Bing Li
15   */
16
17  // Created: 04/21/2017, Bing Li
18  public class ChatServer
19  {
20      private CSServer<ChatServerDispatcher> chatServer;
21      private CSServer<ChatManServerDispatcher> manServer;
22
23      private ChatServer()
24      {
25      }
26
27      private static ChatServer instance = new ChatServer();
28
29      public static ChatServer CS()
30      {
31          if (instance == null)
32          {
33              instance = new ChatServer();
34              return instance;
35          }
36          else
37          {
38              return instance;
```

```
39              }
40          }
41
42          public void stop() throws IOException, InterruptedException
43          {
44              // Set the terminating signal. 11/25/2014, Bing Li
45              TerminateSignal.SIGNAL().setTerminated();
46
47              // Dispose the account registry. 04/30/2017, Bing Li
48              AccountRegistry.CS().dispose();
49
50              // Dispose the chatting session resources. 04/30/2017, Bing Li
51              PrivateChatSessions.HUNGARY().dispose();
52
53              // Stop the two servers. 04/30/2017, Bing Li
54              this.chatServer.stop();
55              this.manServer.stop();
56          }
57
58          public void start() throws IOException, ClassNotFoundException, RemoteReadException
59          {
60              // Initialize the private chatting sessions. 04/23/2017, Bing Li
61              PrivateChatSessions.HUNGARY().init();
62
63              // Initialize the chat server. 04/30/2017, Bing Li
64              this.chatServer = new CSServer.CSServerBuilder<ChatServerDispatcher>()
65                  .port(ChatConfig.CHAT_SERVER_PORT)
66                  .listenerCount(ServerConfig.LISTENING_THREAD_COUNT)
67                  .listenerThreadPoolSize(ServerConfig.SHARED_THREAD_POOL_SIZE)
68                  .listenerThreadKeepAliveTime(
69                      ServerConfig.SHARED_THREAD_POOL_KEEP_ALIVE_TIME)
70                  .dispatcher(new ChatServerDispatcher(ChatConfig.DISPATCHER_THREAD_POOL_SIZE,
71                      ChatConfig.DISPATCHER_THREAD_POOL_KEEP_ALIVE_TIME,
72                      ChatConfig.SCHEDULER_THREAD_POOL_SIZE,
73                      ChatConfig.SCHEDULER_THREAD_POOL_KEEP_ALIVE_TIME))
74                  .build();
75
76              // Initialize the chat management server. 04/30/2017, Bing Li
77              this.manServer = new CSServer.CSServerBuilder<ChatManServerDispatcher>()
78                  .port(ChatConfig.CHAT_ADMIN_PORT)
79                  .listenerCount(ServerConfig.SINGLE_THREAD_COUNT)
80                  .listenerThreadPoolSize(ServerConfig.SHARED_THREAD_POOL_SIZE)
81                  .listenerThreadKeepAliveTime(
82                      ServerConfig.SHARED_THREAD_POOL_KEEP_ALIVE_TIME)
83                  .dispatcher(new ChatManServerDispatcher(
84                      ChatConfig.DISPATCHER_THREAD_POOL_SIZE,
85                      ChatConfig.DISPATCHER_THREAD_POOL_KEEP_ALIVE_TIME,
86                      ChatConfig.SCHEDULER_THREAD_POOL_SIZE,
87                      ChatConfig.SCHEDULER_THREAD_POOL_KEEP_ALIVE_TIME))
88                  .build();
89
90              // Start up the two servers. 04/30/2017, Bing Li
91              this.chatServer.start();
92              this.manServer.start();
93          }
94      }
```

List 5.2 The code of ChatServer.java

Coding is not always tough, specially when you do that with GreatFree, as its name contains the meanings of happiness and relax. You can taste it from the code in List 5.2. Line 20 and Line 21 declare two instances of CSServer, as we discuss above. CSServer is a server that is able to receive requests from clients and then make responses to them immediately after they are generated. I need to indicate that this is the unique function of the server. In other words, the CSServer never intends to interact with other distributed nodes unless it receives requests from them. In short, CSServer behaves over the Internet following the distributed model of C/S restrictedly.

According to my experiences, which is rich indeed, I have no idea which languages ever provide such an API. You must have used some frameworks called servers, such as Tomcat [4], JBoss [5] or Apache Web Server [6], and so forth, you scarcely creates an instance in your code, right?

Why are two instances of CSServer declared for the chatting server? It is not necessary although it is more practical. What cases should you do that? Usually, it should be taken into account when multiple TCP/IP ports [2] should be employed in your applications. Each port is prepared for one particular service. In this case of the chatting server, two ports are set up for the chatting service (Line 20) and the management of the server (Line 21), respectively. Thus, two instances of CSServer are declared.

You must notice that the two instances of CSServer enclose one component, which is named as the dispatcher, such as ChatServerDispatcher and ChatManServerDispatcher, as shown in Line 20 and Line 21, respectively. Do not worry about that since they are programmed exactly in the same way as we discuss previously. Before opening them, let us go though the chatting server.

Line 23~40 defines the chatting server as a singleton [3]. If you have no idea how to do that, you can copy the lines when you need to implement a singleton.

Line 42~65 defines the method of stop(), which is used to terminate the chatting procedure. First of all, Line 45 sets the terminating signal, TerminalSignal, by calling its method, setTerminated(). Since the signal is the unique flag to represent the process is shut down, it impacts the entire process to terminate their respective tasks once if the flag is detected.

Line 48 disposes one class called AccountRegistry. As a demonstration code of CSServer, it is not important. Since we need to program chatting with CSServer, some code for the chatting scenarios should be implemented to some extent. It has nothing to do with the techniques of GreatFree. In the case, AccountRegistry just keeps all of the registered accounts in the chatting. It is opened later as shown in List 5.3.

Line 51 disposes another class called PrivateChatSessions, which is also associated with the chatting application. According to its name, it retains chatting sessions and messages. Its code is shown in List 5.5.

Line 54~55 stops the two instances of CSServer.

Line 58~92 starts the two instances of CSServer. As the counterpart of the method of stop(), it defines the new method of start().

Line 61 initializes the chatting sessions and messages, which are implemented with the traditional approaches of Object-Oriented methodology such that we do not spend time on that.

Line 64~74 initializes one instance of CSServer, chatServer. A bunch of parameters are required to do that. Since they are important to CSServer, they are explained in Table 5.5.

Line 70~73 encloses four additional parameters which are used to define the dispatcher. They are explained in Table 5.6.

Parameter	Explanation
int port	The TCP port of the chatting server
int listenerCount	The listening thread count for the TCP port
int listenerThreadPoolSize	The listening thread is managed by a shared thread pool, which might be employed by more than one scenarios. Its size is set by the argument
long listenerThreadKeepAliveTime	The listening thread should be killed when it is believed to be idle long enough. The time is called keeping alive time
ChatServerDispatcher dispatcher	The dispatcher that is responsible for dispatching chatting messages to corresponding threads such that those messages are processed concurrently. Its instance is set as an argument of the chatting server

Table 5.5 The parameters of the constructor of CSServer

Parameter	Explanation
int threadPoolSize	ChatServerDispatcher has a built-in thread pool that achieves the goal of efficient concurrency. Its pool size is set by the argument
long threadKeepAliveTime	The keeping alive time for the thread pool of ChatServerDispatcher
int schedulerPoolSize	The thread pool size for the scheduler of ChatServerDispatcher. The scheduler is employed by the dispatcher to check the states of threads periodically
long schedulerKeepAliveTime	The keeping alive time for the scheduler of ChatServerDispatcher

Table 5.6 The parameters of the constructor of ChatServerDispatcher

Similar to Line 64~74, Line 77~88 initializes another instance of CSServer, manServer. Both of them have the same parameters and even arguments. In this demonstration code, it is fine. In practice, developers are required to adjust those parameters according to the requirements of particular computing environments.

Line 91~92 starts the two instances of CSServer.

Until now, the code of ChatServer is explained completely. How do you feel about programming with GreatFree so far? From my point of view, it is really straightforward if you already have some rough knowledge about distributed systems as discussed in Section 1 of the chapter. If you decide the most important distributed component is engaged, everything is convenient. The additional effort for you is to understand some parameters like the ones in Table 5.3 and Table 5.4. I do not think you can get stuck by them.

Before we move forward to the next topic, two additional code need to be discussed. They are not related to GreatFree since they are totally Java object-oriented programming. The code can be regarded as the application-level code. Compared with them, GreatFree APIs and patterns can be understood as the system-level code. To run the chatting system, you need to have such code. Because the book is used to teach you how to program with GreatFree, it assumes that you have some fundamental knowledge about programming. So the code is not discussed in details. You can learn it according to the brief explanation and the comments in the code.

List 5.3 presents the code of AccountRegistry.java. The class is disposed (Line 48) in the code of ChatServer.java in List 5.2. This is a very simple implementation to keep

users' accounts just in memory, which is far from a practical account management system. It contains some basic methods for that, such as adding users (Line 60~66) and retrieving a user (Line 79~86).

```
1   package com.greatfree.chat.server;
2
3   import java.util.Collection;
4   import java.util.Map;
5   import java.util.concurrent.ConcurrentHashMap;
6
7   /*
8    * This is an account registry. All of the accounts of the chatting system are retained in it. 04/16/2017, Bing Li
9    */
10
11  // Created: 04/16/2017, Bing Li
12  public class AccountRegistry
13  {
14      private Map<String, CSAccount> accounts;
15
16      private AccountRegistry()
17      {
18          // Define a concurrent map for the consideration of consistency. 11/09/2014, Bing Li
19          this.accounts = new ConcurrentHashMap<String, CSAccount>();
20      }
21
22      /*
23       * A singleton implementation. 11/09/2014, Bing Li
24       */
25      private static AccountRegistry instance = new AccountRegistry();
26
27      public static AccountRegistry CS()
28      {
29          if (instance == null)
30          {
31              instance = new AccountRegistry();
32              return instance;
33          }
34          else
35          {
36              return instance;
37          }
38      }
39
40      /*
41       * Dispose the accounts. 05/21/2017, Bing Li
42       */
43      public void dispose()
44      {
45          this.accounts.clear();
46          this.accounts = null;
47      }
48
49      /*
50       * Check whether one particular account is existed. 05/21/2017, Bing Li
51       */
52      public boolean isAccountExisted(String userKey)
53      {
54          return this.accounts.containsKey(userKey);
55      }
56
57      /*
58       * Add one account. 05/21/2017, Bing Li
59       */
60      public void add(CSAccount account)
61      {
62          if (!this.accounts.containsKey(account.getUserKey()))
63          {
64              this.accounts.put(account.getUserKey(), account);
65          }
66      }
67
68      /*
69       * Expose all of registered accounts. 05/21/2017, Bing Li
70       */
```

```
71          public Collection<CSAccount> getAllAccounts()
72          {
73              return this.accounts.values();
74          }
75
76          /*
77           * Retrieve one account by its key. 05/21/2017, Bing Li
78           */
79          public CSAccount getAccount(String key)
80          {
81              if (this.accounts.containsKey(key))
82              {
83                  return this.accounts.get(key);
84              }
85              return null;
86          }
87      }
```

List 5.3 The code of AccountRegistry.java

The account registry contains a new class, CSAccount, which is named like this because the chatting is based on the C/S model. The code of CSAccount.java is shown in List 5.4. The account contains the user key, the user name and the description about the user. It is extremely simple, right?

```
1       package com.greatfree.chat.server;
2
3       /*
4        * The account of the chatting system for the C/S based chatting. 04/16/2017, Bing Li
5        */
6
7       // Created: 04/16/2017, Bing Li
8       public class CSAccount
9       {
10          private String userKey;
11          private String userName;
12          private String description;
13
14          public CSAccount(String userKey, String userName, String description)
15          {
16              this.userKey = userKey;
17              this.userName = userName;
18              this.description = description;
19          }
20
21          public String getUserKey()
22          {
23              return this.userKey;
24          }
25
26          public String getUserName()
27          {
28              return this.userName;
29          }
30
31          public String getDescription()
32          {
33              return this.description;
34          }
35      }
```

List 5.4 The code of CSAccount.java

```
1       package com.greatfree.chat.server;
2
3       import java.util.ArrayList;
4       import java.util.List;
5       import java.util.Map;
6       import java.util.Set;
```

```
7   import java.util.concurrent.ConcurrentHashMap;
8
9   import com.google.common.collect.Sets;
10
11  /*
12   * The class keeps the chatting sessions for private users. In the case, the private chatting is defined
13   * as the one that happens between two users only. 04/23/2017, Bing Li
14   */
15
16  // Created: 04/23/2017, Bing Li
17  public class PrivateChatSessions
18  {
19      // The map contains the new chatting messages. The key is the chatting session key and the value of
20      // the session that contains the new chatting messages. 04/24/2017, Bing Li
21      private Map<String, ChatSession> chats;
22      // The map contains the new sessions. The key is the receiver key and the value is the new session
23      // key just created. 04/24/2017, Bing Li
24      private Map<String, Set<String>> newSessions;
25
26      private PrivateChatSessions()
27      {
28      }
29
30      /*
31       * A singleton definition. 04/17/2017, Bing Li
32       */
33      private static PrivateChatSessions instance = new PrivateChatSessions();
34
35      public static PrivateChatSessions HUNGARY()
36      {
37          if (instance == null)
38          {
39              instance = new PrivateChatSessions();
40              return instance;
41          }
42          else
43          {
44              return instance;
45          }
46      }
47
48      /*
49       * Dispose the free client pool. 04/17/2017, Bing Li
50       */
51      public void dispose()
52      {
53          this.chats.clear();
54          this.chats = null;
55      }
56
57      /*
58       * Initialize explicitly. 04/23/2017, Bing Li
59       */
60      public void init()
61      {
62          this.chats = new ConcurrentHashMap<String, ChatSession>();
63          this.newSessions = new ConcurrentHashMap<String, Set<String>>();
64      }
65
66      /*
67       * Add a new message. 04/23/2017, Bing Li
68       */
69      public void addMessage(String sessionKey, String senderKey, String receiverKey, String message)
70      {
71          if (!this.chats.containsKey(sessionKey))
72          {
73              this.chats.put(sessionKey, new ChatSession(sessionKey));
74              this.chats.get(sessionKey).addMessage(senderKey, receiverKey, message);
75          }
76          else
77          {
78              this.chats.get(sessionKey).addMessage(senderKey, receiverKey, message);
79          }
80      }
81
82      /*
83       * Get new messages for the receiver. 04/23/2017, Bing Li
```

```
84          */
85         public List<ChatMessage> getNewMessages(String sessionKey, String receiverKey)
86         {
87             if (this.chats.containsKey(sessionKey))
88             {
89                 if (this.chats.get(sessionKey).isPartnerExisted(receiverKey))
90                 {
91                     List<ChatMessage> newMessages = new ArrayList<ChatMessage>();
92                     List<Integer> removedIndexes = new ArrayList<Integer>();
93                     int index = 0;
94                     for (ChatMessage message : this.chats.get(sessionKey).getMessages())
95                     {
96                         if (message.getReceiverKey().equals(receiverKey))
97                         {
98                             newMessages.add(message);
99                             removedIndexes.add(index);
100                        }
101                        index++;
102                    }
103                    for (Integer entry : removedIndexes)
104                    {
105                        this.chats.get(sessionKey).removeMessage(entry);
106                    }
107                    return newMessages;
108                }
109            }
110            return null;
111        }
112
113        public void removeMessage(String sessionKey)
114        {
115            this.chats.get(sessionKey).clearMessages();
116        }
117
118        /*
119         * Add new sessions. 04/24/2017, Bing Li
120         */
121        public void addSession(String receiverKey, String senderKey)
122        {
123            if (!this.newSessions.containsKey(receiverKey))
124            {
125                Set<String> sessionKeys = Sets.newHashSet();
126                this.newSessions.put(receiverKey, sessionKeys);
127            }
128            String sessionKey = ChatTools.getChatSessionKey(senderKey, receiverKey);
129            if (!this.chats.containsKey(sessionKey))
130            {
131                this.chats.put(sessionKey, new ChatSession(sessionKey));
132                this.chats.get(sessionKey).addPartner(senderKey);
133                this.chats.get(sessionKey).addPartner(receiverKey);
134            }
135            this.newSessions.get(receiverKey).add(sessionKey);
136        }
137
138        /*
139         * Is one session available? 04/24/2017, Bing Li
140         */
141        public boolean isSessionExisted(String receiverKey)
142        {
143            return this.newSessions.containsKey(receiverKey);
144        }
145
146        /*
147         * Get the existing session keys. 04/24/2017, Bing Li
148         */
149        public Set<String> getSessionKeys(String receiverKey)
150        {
151            return this.newSessions.get(receiverKey);
152        }
153
154        /*
155         * Remove the session. 04/24/2017, Bing Li
156         */
157        public void removeSession(String receiverKey)
158        {
159            this.newSessions.remove(receiverKey);
160        }
```

```
161 }
```

List 5.5 The code of PrivateChatSessions.java

The code of PrivateChatSessions.java keeps all of the chat sessions and the chat messages within each sessions. Additionally, two new classes, ChatSession (List 5.6) and ChatMessage (List 5.7) emerge in the code. I do not think I need to explain the code too much since it is implemented in a contrived way. It is far from the code in the real world since we need to focus on the distributed issues at this moment. In the later chapters, you will see how to keep huge data persistently in a distributed environment. So now we just leave the code here and move forward.

List 5.6 shows the code of ChatSession, which keeps the information about a chatting session, including the session key, its partners and new chatting messages of the session.

```
1   package com.greatfree.chat.server;
2
3   import java.util.ArrayList;
4   import java.util.Calendar;
5   import java.util.List;
6   import java.util.Set;
7   import java.util.concurrent.locks.ReentrantLock;
8
9   import com.google.common.collect.Sets;
10  import com.greatfree.chat.server.AccountRegistry;
11
12  /*
13  * It keeps all of the messages in a session. As a simulation system, it is too simple. Actually, the ranking
14  * by time and a persistent mechanism are required for a practical system. 04/23/2017, Bing Li
15  */
16
17  // Created: 04/23/2017, Bing Li
18  public class ChatSession
19  {
20      // The unique key of the session. 04/23/2017, Bing Li
21      private final String sessionKey;
22
23      // The chatting partners of the session. 04/23/2017, Bing Li
24      private Set<String> partners;
25
26      // The messages in the session. 04/23/2017, Bing Li
27      private List<ChatMessage> messages;
28      private ReentrantLock lock;
29
30      /*
31       * The constructor. 04/23/2017, Bing Li
32       */
33      public ChatSession(String sessionKey)
34      {
35          // Generate the unique key. 04/23/2017, Bing Li
36          this.sessionKey = sessionKey;
37          this.partners = Sets.newHashSet();
38
39          this.messages = new ArrayList<ChatMessage>();
40          this.lock = new ReentrantLock();
41      }
42
43      public String getSessionKey()
44      {
45          return this.sessionKey;
46      }
47
48      public Set<String> getPartners()
49      {
50          this.lock.lock();
51          try
52          {
```

```
53                  return this.partners;
54              }
55              finally
56              {
57                  this.lock.unlock();
58              }
59          }
60
61          /*
62           * Add a new partner to the chatting session. 04/23/2017, Bing Li
63           */
64          public void addPartner(String userKey)
65          {
66              this.lock.lock();
67              this.partners.add(userKey);
68              this.lock.unlock();
69          }
70
71          /*
72           * Check whether one user participates the chatting. 04/23/2017, Bing Li
73           */
74          public boolean isPartnerExisted(String userKey)
75          {
76              this.lock.lock();
77              try
78              {
79                  return this.partners.contains(userKey);
80              }
81              finally
82              {
83                  this.lock.unlock();
84              }
85          }
86
87          /*
88           * Add a new message. 04/23/2017, Bing Li
89           */
90          public void addMessage(String senderKey, String receiverKey, String message)
91          {
92              this.lock.lock();
93              this.messages.add(new ChatMessage(senderKey, AccountRegistry.CS().getAccount
94                  (senderKey).getUserName(), receiverKey, message, Calendar.getInstance().getTime()));
95              this.lock.unlock();
96          }
97
98          /*
99           * Expose all messages. 04/23/2017, Bing Li
100          */
101         public List<ChatMessage> getMessages()
102         {
103             this.lock.lock();
104             try
105             {
106                 return this.messages;
107             }
108             finally
109             {
110                 this.lock.unlock();
111             }
112         }
113
114         public void removeMessage(int index)
115         {
116             this.lock.lock();
117             try
118             {
119                 this.messages.remove(index);
120             }
121             finally
122             {
123                 this.lock.unlock();
124             }
125         }
126
127         /*
128          * Clear old messages. 04/24/2017, Bing Li
129          */
```

```
130         public void clearMessages()
131         {
132             this.lock.lock();
133             this.messages.clear();
134             this.lock.unlock();
135         }
136     }
```

List 5.6 The code of ChatSession.java

List 5.7 shows the code of ChatMessage.java. It keeps one piece of chatting message. It is created when a polling request for polling new chatting messages is received. It is sent back to the polling requestor such that two clients can interact with one another.

```
1       package com.greatfree.chat;
2
3       import java.io.Serializable;
4       import java.util.Date;
5
6       /*
7        * The class keeps one piece of message for a chat. 04/23/2017, Bing Li
8        */
9
10      // Created: 04/23/2017, Bing Li
11      public class ChatMessage implements Serializable
12      {
13          private static final long serialVersionUID = 8656444434139713955L;
14
15          // The sender of the message. 04/23/2017, Bing Li
16          private String senderKey;
17          // The sender's name. 04/24/2017, Bing Li
18          private String senderName;
19          // The receiver of the message. 04/23/2017, Bing Li
20          private String receiverKey;
21          // The message itself. 04/23/2017, Bing Li
22          private String message;
23          // The time of the message being created. 04/23/2017, Bing Li
24          private Date time;
25
26          /*
27           * The constructor. 04/23/2017, Bing Li
28           */
29          public ChatMessage(String senderKey, String senderName, String receiverKey, String message,
30              Date time)
31          {
32              this.senderKey = senderKey;
33              this.senderName = senderName;
34              this.receiverKey = receiverKey;
35              this.message = message;
36              this.time = time;
37          }
38
39          public String getSenderKey()
40          {
41              return this.senderKey;
42          }
43
44          public String getSenderName()
45          {
46              return this.senderName;
47          }
48
49          public String getReceiverKey()
50          {
51              return this.receiverKey;
52          }
53
54          public String getMessage()
55          {
```

```
56              return this.message;
57          }
58
59          public Date getTime()
60          {
61              return this.time;
62          }
63      }
```

List 5.7 The code of ChatMessage.java

Pattern: SD (Server Dispatcher)		
Class	CSServer<ChatServerDispatcher>	
Singleton	No	
Method	void consume(OutMessageStream<ServerMessage> message)	
	void shutdown()	
Enclosed Pattern	RD (Request Dispatcher)	RequestDispatcher<ChatRegistryRequest, ChatRegistryStream, ChatRegistryResponse, ChatRegistryThread, ChatRegistryThreadCreator>
		RequestDispatcher<ChatPartnerRequest, ChatPartnerStream, ChatPartnerResponse, ChatPartnerRequestThread, ChatPartnerRequestThreadCreator>
		RequestDispatcher<PollNewSessionsRequest, PollNewSessionsStream, PollNewSessionsResponse, PollNewSessionsThread, PollNewSessionsThreadCreator>
		RequestDispatcher<PollNewChatsRequest, PollNewChatsStream, PollNewChatsResponse, PollNewChatsThread, PollNewChatsThreadCreator>
	ND (Notification Dispatcher)	NotificationDispatcher<AddPartnerNotification, AddPartnerThread, AddPartnerThreadCreator>
		NotificationDispatcher<ChatNotification, ChatThread, ChatThreadCreator>
Employed API	ServerDispatcher	
	RequestDispatcher<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends RequestQueue<Request, Stream, Response>, ThreadCreator extends RequestThreadCreatable<Request, Stream, Response, RequestThread>>	
	NotificationDispatcher<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>, ThreadCreator extends NotificationThreadCreatable<Notification, NotificationThread>>	

Table 5.7 The primary components of the pattern of SD for the chatting server dispatcher

Besides ChatSession and ChatMessage, PrivateChatSessions still contains a new class we never see, ChatTools, as shown in List 5.8. It is used to generate hash keys for chatting sessions and users. Inside it, it is actually supported by the API, Tools, from GreatFree.

```
1   package com.greatfree.chat.server;
2
3   import com.greatfree.util.Tools;
4
5   /*
6    * The tool is shared by the client, the server and even the administrator for chatting. 04/27/2017, Bing Li
7    */
8
9   // Created: 04/27/2017, Bing Li
10  public class ChatTools
11  {
12      public static String getChatSessionKey(String senderKey, String receiverKey)
13      {
14          if (senderKey.compareTo(receiverKey) > 0)
15          {
16              return Tools.getHash(senderKey + receiverKey);
17          }
18          else
19          {
```

```
            return Tools.getHash(receiverKey + senderKey);
        }
    }

    public static String getUserKey(String username)
    {
        return Tools.getHash(username);
    }
}
```

List 5.8 The code of ChatTools.java

2.3 SD – The Pattern of Server Dispatcher: Dispatching Messages

To continue the implementation of the chatting server, you have to program ChatServerDispatcher, which is the concurrency mechanism for the chatting server. This is a specific case of the most pattern, Server Dispatcher (SD), in GreatFree. When programming a server, the pattern is the critical component for you to design. With the support of SD, the issue of distributed concurrency becomes convenient to be implemented by coding. Table 5.7 illustrates the primary components of the pattern of SD for the chatting server. Furthermore, it can be generalized as the pattern for all of the cases on the server side when the distributed model of C/S is adopted.

I believe you must understand what the concurrency means. This is the most important topic for any distributed systems. Fortunately, only such a rough idea is sufficient for you to program with GreatFree. Let us open the code of ChatServerDispatcher and you must be surprised to the code inside it. List 5.9 shows the complete code of the dispatcher.

```
package com.greatfree.chat.server;

import java.util.Calendar;

import com.greatfree.chat.message.ChatMessageType;
import com.greatfree.chat.message.cs.AddPartnerNotification;
import com.greatfree.chat.message.cs.ChatNotification;
import com.greatfree.chat.message.cs.ChatPartnerRequest;
import com.greatfree.chat.message.cs.ChatPartnerResponse;
import com.greatfree.chat.message.cs.ChatPartnerStream;
import com.greatfree.chat.message.cs.ChatRegistryRequest;
import com.greatfree.chat.message.cs.ChatRegistryResponse;
import com.greatfree.chat.message.cs.ChatRegistryStream;
import com.greatfree.chat.message.cs.PollNewChatsRequest;
import com.greatfree.chat.message.cs.PollNewChatsResponse;
import com.greatfree.chat.message.cs.PollNewChatsStream;
import com.greatfree.chat.message.cs.PollNewSessionsRequest;
import com.greatfree.chat.message.cs.PollNewSessionsResponse;
import com.greatfree.chat.message.cs.PollNewSessionsStream;
import com.greatfree.concurrency.NotificationDispatcher;
import com.greatfree.concurrency.RequestDispatcher;
import com.greatfree.message.ServerMessage;
import com.greatfree.remote.OutMessageStream;
import com.greatfree.server.ServerDispatcher;
import com.greatfree.testing.data.ServerConfig;

/*
 * The server dispatcher extends com.greatfree.server.ServerDispatcher to implement a sub server
 * dispatcher. 04/15/2017, Bing Li
 */

// Created: 04/15/2017, Bing Li
public class ChatServerDispatcher extends ServerDispatcher
{
    // Declare a request dispatcher to respond users' registry requests concurrently. 04/17/2017, Bing Li
```

```
36      private RequestDispatcher<ChatRegistryRequest, ChatRegistryStream, ChatRegistryResponse,
37          ChatRegistryThread, ChatRegistryThreadCreator> registryRequestDispatcher;
38      // Declare a request dispatcher to respond users' chat partner requests concurrently. 04/17/2017, Bing Li
39      private RequestDispatcher<ChatPartnerRequest, ChatPartnerStream, ChatPartnerResponse,
40          ChatPartnerRequestThread, ChatPartnerRequestThreadCreator> chatPartnerRequestDispatcher;
41      // Declare a notification dispatcher to add friends for the chatting when such a notification is
42      // received. 04/18/2016, Bing Li
43      private NotificationDispatcher<AddPartnerNotification, AddPartnerThread, AddPartnerThreadCreator>
44          addPartnerNotificationDispatcher;
45      // Declare a request dispatcher to respond users' new sessions requests concurrently. 04/17/2017, Bing Li
46      private RequestDispatcher<PollNewSessionsRequest, PollNewSessionsStream,
47          PollNewSessionsResponse, PollNewSessionsThread, PollNewSessionsThreadCreator>
48          pollNewSessionsRequestDispatcher;
49      // Declare a request dispatcher to respond users' new chat requests concurrently. 04/17/2017, Bing Li
50      private RequestDispatcher<PollNewChatsRequest, PollNewChatsStream, PollNewChatsResponse,
51           PollNewChatsThread, PollNewChatsThreadCreator> pollNewChatsRequestDispatcher;
52      // Declare a notification dispatcher to add chatting messages to the server when such a
53      // notification is received. 04/18/2016, Bing Li
54      private NotificationDispatcher<ChatNotification, ChatThread, ChatThreadCreator>
55          chatNotificationDispatcher;
56
57      /*
58       * The constructor of ChatServerDispatcher. 04/15/2017, Bing Li
59       */
60      public ChatServerDispatcher(int threadPoolSize, long threadKeepAliveTime, int schedulerPoolSize,
61          long schedulerKeepAliveTime)
62      {
63          super(threadPoolSize, threadKeepAliveTime, schedulerPoolSize, schedulerKeepAliveTime);
64
65          this.registryRequestDispatcher = new RequestDispatcher.RequestDispatcherBuilder
66              <ChatRegistryRequest, ChatRegistryStream, ChatRegistryResponse, ChatRegistryThread,
67              ChatRegistryThreadCreator>()
68              .poolSize(ServerConfig.REQUEST_DISPATCHER_POOL_SIZE)
69              .keepAliveTime(ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME)
70              .threadCreator(new ChatRegistryThreadCreator())
71              .maxTaskSize(ServerConfig.MAX_REQUEST_TASK_SIZE)
72              .dispatcherWaitTime(ServerConfig.REQUEST_DISPATCHER_WAIT_TIME)
73              .waitRound(ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND)
74              .idleCheckDelay(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY)
75              .idleCheckPeriod(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD)
76              .scheduler(super.getSchedulerPool())
77              .build();
78
79          this.chatPartnerRequestDispatcher = new RequestDispatcher.RequestDispatcherBuilder
80              <ChatPartnerRequest, ChatPartnerStream, ChatPartnerResponse, ChatPartnerRequestThread,
81              ChatPartnerRequestThreadCreator>()
82              .poolSize(ServerConfig.REQUEST_DISPATCHER_POOL_SIZE)
83              .keepAliveTime(ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME)
84              .threadCreator(new ChatPartnerRequestThreadCreator())
85              .maxTaskSize(ServerConfig.MAX_REQUEST_TASK_SIZE)
86              .dispatcherWaitTime(ServerConfig.REQUEST_DISPATCHER_WAIT_TIME)
87              .waitRound(ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND)
88              .idleCheckDelay(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY)
89              .idleCheckPeriod(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD)
90              .scheduler(super.getSchedulerPool())
91              .build();
92
93          this.addPartnerNotificationDispatcher = new NotificationDispatcher.NotificationDispatcherBuilder
94              <AddPartnerNotification, AddPartnerThread, AddPartnerThreadCreator>()
95              .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
96              .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
97              .threadCreator(new AddPartnerThreadCreator())
98              .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
99              .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
100             .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
101             .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
102             .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
103             .scheduler(super.getSchedulerPool())
104             .build();
105
106         this.pollNewSessionsRequestDispatcher = new RequestDispatcher.RequestDispatcherBuilder
107             <PollNewSessionsRequest, PollNewSessionsStream, PollNewSessionsResponse,
108             PollNewSessionsThread, PollNewSessionsThreadCreator>()
109             .poolSize(ServerConfig.REQUEST_DISPATCHER_POOL_SIZE)
110             .keepAliveTime(ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME)
111             .threadCreator(new PollNewSessionsThreadCreator())
112             .maxTaskSize(ServerConfig.MAX_REQUEST_TASK_SIZE)
```

```
113                 .dispatcherWaitTime(ServerConfig.REQUEST_DISPATCHER_WAIT_TIME)
114                 .waitRound(ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND)
115                 .idleCheckDelay(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY)
116                 .idleCheckPeriod(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD)
117                 .scheduler(super.getSchedulerPool())
118                 .build();
119
120             this.pollNewChatsRequestDispatcher = new RequestDispatcher.RequestDispatcherBuilder
121                 <PollNewChatsRequest, PollNewChatsStream, PollNewChatsResponse, PollNewChatsThread,
122                 PollNewChatsThreadCreator>()
123                 .poolSize(ServerConfig.REQUEST_DISPATCHER_POOL_SIZE)
124                 .keepAliveTime(ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME)
125                 .threadCreator(new PollNewChatsThreadCreator())
126                 .maxTaskSize(ServerConfig.MAX_REQUEST_TASK_SIZE)
127                 .dispatcherWaitTime(ServerConfig.REQUEST_DISPATCHER_WAIT_TIME)
128                 .waitRound(ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND)
129                 .idleCheckDelay(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY)
130                 .idleCheckPeriod(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD)
131                 .scheduler(super.getSchedulerPool())
132                 .build();
133
134             this.chatNotificationDispatcher = new NotificationDispatcher.NotificationDispatcherBuilder
135                 <ChatNotification, ChatThread, ChatThreadCreator>()
136                 .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
137                 .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
138                 .threadCreator(new ChatThreadCreator())
139                 .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
140                 .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
141                 .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
142                 .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
143                 .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
144                 .scheduler(super.getSchedulerPool())
145                 .build();
146         }
147
148         /*
149          * Shut down the server message dispatcher. 04/15/2017, Bing Li
150          */
151         public void shutdown() throws InterruptedException
152         {
153             this.registryRequestDispatcher.dispose();
154             this.chatPartnerRequestDispatcher.dispose();
155             this.addPartnerNotificationDispatcher.dispose();
156             this.pollNewSessionsRequestDispatcher.dispose();
157             this.pollNewChatsRequestDispatcher.dispose();
158             this.chatNotificationDispatcher.dispose();
159             super.shutdown();
160         }
161
162         /*
163          * Process the available messages in a concurrent way. 04/17/2017, Bing Li
164          */
165         public void consume(OutMessageStream<ServerMessage> message)
166         {
167             // Check the types of received messages. 04/17/2017, Bing Li
168             switch (message.getMessage().getType())
169             {
170                 case ChatMessageType.CS_CHAT_REGISTRY_REQUEST:
171                     System.out.println("CS_CHAT_REGISTRY_REQUEST received @" +
172                         Calendar.getInstance().getTime());
173                     // Check whether the registry dispatcher is ready. 04/17/2017, Bing Li
174                     if (!this.registryRequestDispatcher.isReady())
175                     {
176                         // Execute the registry dispatcher as a thread. 04/17/2017, Bing Li
177                         super.execute(this.registryRequestDispatcher);
178                     }
179                     // Enqueue the request into the dispatcher for concurrent responding. 04/17/2017, Bing Li
180                     this.registryRequestDispatcher.enqueue(new ChatRegistryStream(message.getOutStream(),
181                         message.getLock(), (ChatRegistryRequest)message.getMessage()));
182                     break;
183
184                 case ChatMessageType.CS_CHAT_PARTNER_REQUEST:
185                     System.out.println("CS_CHAT_PARTNER_REQUEST received @" +
186                         Calendar.getInstance().getTime());
187                     // Check whether the partner request dispatcher is ready. 04/17/2017, Bing Li
188                     if (!this.chatPartnerRequestDispatcher.isReady())
189                     {
```

```
190                     // Execute the partner request dispatcher as a thread. 04/17/2017, Bing Li
191                     super.execute(this.chatPartnerRequestDispatcher);
192                 }
193                 // Enqueue the request into the dispatcher for concurrent responding. 04/17/2017, Bing Li
194                 this.chatPartnerRequestDispatcher.enqueue(new ChatPartnerStream
195                     (message.getOutStream(), message.getLock(),
196                     (ChatPartnerRequest)message.getMessage()));
197                 break;
198
199             case ChatMessageType.CS_ADD_PARTNER_NOTIFICATION:
200                 System.out.println("CS_ADD_PARTNER_NOTIFICATION received @" +
201                     Calendar.getInstance().getTime());
202                 // Check whether the adding friends notification dispatcher is ready or not. 02/15/2016, Bing Li
203                 if (!this.addPartnerNotificationDispatcher.isReady())
204                 {
205                     // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
206                     super.execute(this.addPartnerNotificationDispatcher);
207                 }
208                 // Enqueue the instance of AddFriendNotification into the dispatcher for concurrent
209                 // processing. 02/15/2016, Bing Li
210                 this.addPartnerNotificationDispatcher.enqueue((AddPartnerNotification)
211                     message.getMessage());
212                 break;
213
214             case ChatMessageType.POLL_NEW_SESSIONS_REQUEST:
215                 System.out.println("POLL_NEW_SESSIONS_REQUEST received @" +
216                     Calendar.getInstance().getTime());
217                 // Check whether the partner request dispatcher is ready. 04/17/2017, Bing Li
218                 if (!this.pollNewSessionsRequestDispatcher.isReady())
219                 {
220                     // Execute the partner request dispatcher as a thread. 04/17/2017, Bing Li
221                     super.execute(this.pollNewSessionsRequestDispatcher);
222                 }
223                 // Enqueue the request into the dispatcher for concurrent responding. 04/17/2017, Bing Li
224                 this.pollNewSessionsRequestDispatcher.enqueue(new PollNewSessionsStream
225                     (message.getOutStream(), message.getLock(), (PollNewSessionsRequest)
226                     message.getMessage()));
227                 break;
228
229             case ChatMessageType.POLL_NEW_CHATS_REQUEST:
230                 System.out.println("POLL_NEW_CHATS_REQUEST received @" +
231                     Calendar.getInstance().getTime());
232                 // Check whether the partner request dispatcher is ready. 04/17/2017, Bing Li
233                 if (!this.pollNewChatsRequestDispatcher.isReady())
234                 {
235                     // Execute the partner request dispatcher as a thread. 04/17/2017, Bing Li
236                     super.execute(this.pollNewChatsRequestDispatcher);
237                 }
238                 // Enqueue the request into the dispatcher for concurrent responding. 04/17/2017, Bing Li
239                 this.pollNewChatsRequestDispatcher.enqueue(new PollNewChatsStream
240                     (message.getOutStream(), message.getLock(), (PollNewChatsRequest)
241                     message.getMessage()));
242                 break;
243
244             case ChatMessageType.CS_CHAT_NOTIFICATION:
245                 System.out.println("CHAT_NOTIFICATION received @" + Calendar.getInstance().getTime());
246                     // Check whether the adding friends notification dispatcher is ready or not. 02/15/2016, Bing Li
247                 if (!this.chatNotificationDispatcher.isReady())
248                 {
249                     // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
250                     super.execute(this.chatNotificationDispatcher);
251                 }
252                 // Enqueue the instance of ChatNotification into the dispatcher for concurrent
253                 // processing. 02/15/2016, Bing Li
254                 this.chatNotificationDispatcher.enqueue((ChatNotification)message.getMessage());
255                 break;
256         }
257     }
258 }
```

List 5.9 The code of ChatServerDispatcher.java

In this case, as the implementation of SD, ChatServerDispatcher is the most important code you need to program. In general, when programming with GreatFree, what developers take care of is not so much. Until now, when you program the chatting application using GreatFree, the code you need to program is the distributed component, CSServer with the pattern of SP and MS, and the messaging dispatching mechanism, ChatServerDispatcher with SD. From my point of view, the load is lower since the knowledge and relevant techniques are very fundamental. You can do that intuitively indeed based on your basic skills. Besides those APIs and patterns, you need to know the concept of threading and messages roughly. In short, to program a distributed application, you should answer the four aspects of rudimentary questions.

1) Which distributed component should be selected?

2) What is the messaging concurrency mechanism, i.e., the server dispatcher?

3) What is the threading?

4) How to design messages?

If all of the above problems should be resolved by developers per se, it is still a tough job. Fortunately, GreatFree has already patterned those solutions. If you are just interested in programming applications, the load is low enough even though you are required to take care about the technical issues.

Now let us take a look at the code of ChatServerDispatcher.java as shown in List 5.3. The code must look familiar to you, right? Recall the code of List 3.8 in the Chapter 3, which also presents a messaging concurrency mechanism, MyServerDispatcher. The dispatcher contains a couple of instances of NotificationDispatcher and RequestDispatcher, which is similar to those enclosed in ChatServerDispatcher. Although it seems that they have different parent classes, they are actually the same since ServerDispatcher derived by ChatServerDispatcher inherits ServerMessageDispatcher, which is extended by MyServerDispatcher directly. To make the interface more concise, some system level messaging mechanisms are hidden by ServerDispatcher. So in the future programming, you seldom need to take care about ServerMessageDispatcher. In brief, you just focus on the pattern of SD.

Anyway, when you are aware of the similarity, you should feel comfortable because the programming procedure is turned into the simple behaviors, CPR (copy-paste-replace). So everything becomes straightforward and repeatable. The convenience is due to the highly patterned code of GreatFree.

After discussing about the character of GreatFree, you must get relaxed. So, it is time for us to explain the code of ChatServerDispatcher.java in List 5.3. This is the code you need to program line by line from scratch with CPR.

Line 1~25 defines the package and imports all of APIs employed in the code.

Line 33 illustrates that your dispatcher should extend ServerDispatcher in any cases.

Line 36~37 declares one instance of RequestDispatcher. The request of the instance is ChatRegistryRequest whereas its counterpart response is ChatRegistryResponse. It is used to register a user when it enters the chatting system for the first time. You have ever programmed a request/response in Chapter 4. So you do not need to scare the code.

Line 39~40 declares one instance of RequestDispatcher again. It is sent from a user to make a query about one potential chatting partner. It ensures whether the potential partner exists and obtains some necessary descriptions about him.

Parameter	Explanation
int poolSize	An instance of RequestDispatcher has a built-in thread pool that achieves the goal of efficient concurrency. Its pool size is set by the argument
long keepAliveTime	The keeping alive time for the thread pool of RequestDispatcher
RequestThreadCreatable threadCreator	RequestDispatcher is also a thread management mechanism. When all of alive threads are busy and the upper limit is not reached, those creators are used to create new threads for the requests currently waiting to be processed
int maxTaskSize	Each thread managed by RequestDispatcher has a queue to take incoming requests. The size of the queue is specified here. The larger the size, the lower concurrency the system can achieve. In contrast, the smaller the size, the higher concurrency the system can gain.
long dispatcherWaitTime	When no requests to be processed, the instance of RequestDispatcher need to wait for some time. Without the waiting, the CPU usage turns out to be high
int waitRound	When no notifications should be processed, to ensure whether it is time to dispose itself, it is necessary to wait for a couple of times before doing that. One time is identical to the round. It indicates that the instance of NotificationDispatcher has waited for a couple of periods, each of which is equal to eventingWaitTime. If the value is higher, it represents that it should wait for more rounds, and vice versa. The parameter of waitRound ensures whether one instance of RequestDispatcher should be disposed. The larger the value, the longer the life cycle of the instance
long idleCheckDelay	The instance of RequestDispatcher is actually a thread pool. The parameter is used to check how long to wait before starting check whether one thread is idle or not.
long idleCheckPeriod	The parameter defines the period to check whether one thread inside the instance of RequestDispatcher is idle or not
ScheduledThreadPoolExecutor scheduler	Since it is necessary to check idleness of threads periodically, the periodically running task is executed by the scheduler pool

Table 5.8 The parameters of the constructor of RequestDispatcher

Line 43~44 declares one instance of NotificationDispatcher. It notifies the server from a user who needs to add another one as his chatting partner.

Line 46~48 declares a session polling request dispatcher for chatting sessions. At discussed previously, in practice, most chatting systems are implemented in the distributed model of Client/Server. Thus, a polling request is mandatory for users to interact. In this case, it checks whether any chatting sessions exist in the CSServer. If it really exists, it is possible for the requesting user to accept the chatting and start to chat. The session polling request is sent to the CSServer periodically.

Line 50~51 declares a chat polling request dispatcher to check whether new chatting messages exist or not. If they do, the new messages are retrieved from the server and respond to the polling user.

Line 54~55 declares an instance of NotificationDispatcher which notifies the CSServer that one new chatting message is generated and sent from one user. The CSServer must retain the message and keep it into PrivateChatSessions, which is

presented in List 5.5. Before a message polling request is received, the chatting message has to be retained there.

Line 60~146 is the constructor of the class, ChatServerDisaptcher. Table 5.4 explains the parameters of the constructor. Besides those parameters, all of instances of RequestDispatcher and NotificationDispatcher are initialized in the constructor. No matter whether you like or not, some parameters should be specified. You must be aware that the format of those message dispatchers' initializations are different from those in List 3.8. Actually, here one design pattern is utilized, which is called the Builder Pattern [7]. It is needed when the number of the parameters of a constructor is large. In this case, both the instances of RequestDispatcher and those of NotificationDispatcher have nine parameters for their constructors. They are explained in Table 5.8 and Table 5.9, respectively.

Parameter	Explanation
int poolSize	An instance of NotificationDispatcher has a built-in thread pool that achieves the goal of efficient concurrency. Its pool size is set by the argument
long keepAliveTime	The keeping alive time for the thread pool of NotificationDispatcher
NotificationThreadCreatable threadCreator	NotificationDispatcher is also a thread management mechanism. When all of alive threads are busy and the upper limit is not reached, those creators are used to create new threads for the requests currently waiting to be processed
int maxTaskSize	Each thread managed by NotificationDispatcher has a queue to take incoming requests. The size of the queue is specified here. The larger the size, the lower concurrency the system can achieve. In contrast, the smaller the size, the higher concurrency the system can gain.
long dispatcherWaitTime	When no requests to be processed, the instance of NotificationDispatcher need to wait for some time. Without the waiting, the CPU usage turns out to be high
int waitRound	When no notifications should be processed, to ensure whether it is time to dispose itself, it is necessary to wait for a couple of times before doing that. One time is identical to the round. It indicates that the instance of NotificationDispatcher has waited for a couple of periods, each of which is equal to eventingWaitTime. If the value is higher, it represents that it should wait for more rounds, and vice versa. The parameter of waitRound ensures whether one instance of NotificationDispatcher should be disposed. The larger the value, the longer the life cycle of the instance
long idleCheckDelay	The instance of NotificationDispatcher is actually a thread pool. The parameter is used to check how long to wait before starting check whether one thread is idle or not.
long idleCheckPeriod	The parameter defines the period to check whether one thread inside the instance of NotificationDispatcher is idle or not
ScheduledThreadPoolExecutor scheduler	Since it is necessary to check idleness of threads periodically, the periodically running task is executed by the scheduler pool

Table 5.9 The parameters of the constructor of NotificationDispatcher

With respect to Table 5.8 and Table 5.9, you learn that the constructors of RequestDispatcher and NotificationDispatcher are identical. In fact, the internal management algorithm is the same as well. The only difference is that the threads they maintain are different.

Line 65~77 initializes the RequestDispatcher instance, registryRequestDispatcher.

Line 79~91 initializes the RequestDispatcher instance, chatPartnerRequestDispatcher.

Line 93~104 initializes the NotificationDispatcher instance, addPartnerNotificationDispatcher.

Line 106~119 initializes the RequestDispatcher instance, pollNewSessionsRequestDispatcher.

Line 120~132 initializes the RequestDispatcher instance, pollNewChatsRequestDispatcher.

Line 134~145 initializes the NotificationDispatcher instance, chatNotificationDispatcher.

Line 151~160 defines the method, shutdown(), to dispose all of instances of dispatchers and shut down the ServerDispatcher. It is executed when the process is stopped.

Line 153~158 disposes all of the instances of RequestDispatcher and NotificationDispatcher, respectively.

Line 159 shuts down the ServerDispatcher.

Line 165~258 defines the method, consume(), which receives incoming messages, including requests and notifications, and dispatches each of them to their respective dispatchers to process them concurrently. The method is come up with a switch-case statement only.

Line 170~182 dispatches the request, ChatRegistryRequest. First, Line 174 checks whether the registryRequestDispatcher is ready or not. If it is ready, it means that the internal management resources of registryRequestDispatcher are not only initialized but also active. If so, the new incoming request can be enqueued into the dispatcher. If not, those required resources should be activated. It is necessary to mention that all of those resources can be collected if no requests are received for a certain time. Thus, the activation can probably be done repeatedly according to the situations in the real world. Doing that in this way aims to save resources. Line 177 executes the dispatcher such that all of required resources are initialized and ready for processing new coming requests. Line 180~181 enqueues the new request into the dispatcher for scheduling and processing. In that line, the instance of ChatRegistryStream, which is a subclass of OutMessageStream, is initialized. After all of the tasks are done, the new incoming request is dispatched and ready to be processed.

Line 184~197, Line 214~227 and Line 229~242 enqueues the requests, such as ChatPartneRequest, PollNewSessionsRequest and PollNewChatsRequest, respectively. All of them utilize the same pattern. So developers can program them by CPR.

A little bit different from that of RequestDispatcher, Line 199~212 dispatches notifications rather than requests. The type of notification to be processed is that of AddPartnerNotification. Line 210 illustrates the difference. Instead of creating the instance of OutMessageStream, the notification is enqueued into the dispatcher directly since no response is needed in this case.

Line 244~255 is also one notification dispatcher to process notifications. It aims to dispatch the notification of ChatNotification. The pattern is actually identical to that for AddPartnerNotification between Line 199 and Line 212.

In fact, the code of ChatServerDispatcher.java demonstrates one important pattern in GreatFree, Server Dispatcher (SD). When programming with GreatFree, most developers' effort is spent on the pattern since all of the messages, threads and dispatchers must be programmed by hand.

2.4 SD – The Pattern of Server Dispatcher Again: Managing the Server Remotely

This is another case that employs the pattern of Server Dispatcher as done in List 5.9. As discussed previously, it is necessary to program such a server dispatcher when multiple ports are utilized in one process. The dispatcher is used to receive remote management instructions from an administrator. For simplicity, only one command for such an administration is designed in the sample. It is programmed in the form of a notification message, i.e., ShutdownChatServerNotification. Programming notification are explained in Chapter 3 in detail. The code is shown in List 5.10. As the pattern of SD, the structure of the code is identical to the code of ChatServerDispatcher.java. So we do not need to spent much time on that. Table 5.10 illustrates the primary components of the pattern of SD for the chatting management server. According to Table 5.10 and Table 5.7, you find that they looks very similar. Although the current SD for the chatting management is more simple, the fundamental structure for both of them is identical. Thus, when programming a server for the distributed model of C/S, the pattern is certainly a proper choice.

Pattern: SD (Server Dispatcher)		
Class	CSServer<ChatManServerDispatcher>	
Singleton	No	
Method	void consume(OutMessageStream<ServerMessage> message)	
	void shutdown()	
Enclosed Pattern	ND (Notification Dispatcher)	NotificationDispatcher<ShutdownChatServerNotification, ShutdownChattingServerThread, ShutdownChattingServerThreadCreator>
Employed API	ServerDispatcher	
	NotificationDispatcher<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>, ThreadCreator extends NotificationThreadCreatable<Notification, NotificationThread>>	

Table 5.10 The primary components of the pattern of SD for the chatting management server

```
1   package com.greatfree.chat.server;
2
3   import java.util.Calendar;
4
5   import com.greatfree.chat.message.ChatMessageType;
6   import com.greatfree.chat.message.ShutdownChatServerNotification;
7   import com.greatfree.concurrency.NotificationDispatcher;
8   import com.greatfree.message.ServerMessage;
9   import com.greatfree.remote.OutMessageStream;
10  import com.greatfree.server.ServerDispatcher;
11  import com.greatfree.testing.data.ServerConfig;
12
13  /*
14   * The server dispatcher extends com.greatfree.server.ServerDispatcher to implement a sub server
15   * dispatcher for administration. 04/15/2017, Bing Li
16   */
17
```

```
18  // Created: 04/20/2017, Bing Li
19  public class ChatManServerDispatcher extends ServerDispatcher
20  {
21      // Declare a notification dispatcher to shutdown the server when such a notification is
22      // received. 04/18/2016, Bing Li
23      private NotificationDispatcher<ShutdownChatServerNotification, ShutdownChattingServerThread,
24          ShutdownChattingServerThreadCreator> shutdownNotificationDispatcher;
25
26      public ChatManServerDispatcher(int threadPoolSize, long threadKeepAliveTime,
27          int schedulerPoolSize, long schedulerKeepAliveTime)
28      {
29          super(threadPoolSize, threadKeepAliveTime, schedulerPoolSize, schedulerKeepAliveTime);
30
31          this.shutdownNotificationDispatcher = new NotificationDispatcher.NotificationDispatcherBuilder
32              <ShutdownChatServerNotification, ShutdownChattingServerThread,
33              ShutdownChattingServerThreadCreator>()
34          .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
35          .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
36          .threadCreator(new ShutdownChattingServerThreadCreator())
37          .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
38          .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
39          .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
40          .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
41          .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
42          .scheduler(super.getSchedulerPool())
43          .build();
44      }
45
46      /*
47       * Shut down the server message dispatcher. 04/15/2017, Bing Li
48       */
49      public void shutdown() throws InterruptedException
50      {
51          this.shutdownNotificationDispatcher.dispose();
52          super.shutdown();
53      }
54
55      /*
56      * Process the available messages in a concurrent way. 04/17/2017, Bing Li
57      */
58      public void consume(OutMessageStream<ServerMessage> message)
59      {
60          // Check the types of received messages. 04/17/2017, Bing Li
61          switch (message.getMessage().getType())
62          {
63              case ChatMessageType.SHUTDOWN_CHAT_SERVER_NOTIFICATION:
64                  System.out.println("SHUTDOWN_CHAT_SERVER_NOTIFICATION received @"
65                      + Calendar.getInstance().getTime());
66                  // Check whether the shutdown notification dispatcher is ready or not. 02/15/2016, Bing Li
67                  if (!this.shutdownNotificationDispatcher.isReady())
68                  {
69                      // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
70                      super.execute(this.shutdownNotificationDispatcher);
71                  }
72                  // Enqueue the instance of ShutdownChatServerNotification into the dispatcher
73                  // for concurrent processing. 02/15/2016, Bing Li
74
75                  this.shutdownNotificationDispatcher.enqueue((ShutdownChatServerNotification)
76                      message.getMessage());
77              break;
78          }
79      }
80  }
```

List 5.10 The code of ChatManServerDispatcher.java

Till now, all of the code emerged in List 5.2 is explained. It is essential to emphasize that all of code should be programmed from scratch although you can follow existing APIs and idioms. When programming a distributed system in the model of C/S, you can borrow code from the sample in the chapter.

Before moving forward, let us take a look at the code of ChatMessageType.java shown in List 5.11. As we program notifications and requests, each message needs to have an integer code to represent it, as we do that for the code MessageType.java in List 3.2 in Chapter 3 and List 3.2 in Chapter 4. In our current case, we have seen the class of ChatMessageType is used in the switch-case statements. In fact, it contains all of the integer values that represent the chatting-related messages. In other words, the class is referenced in each of messages to represent those messages in the chatting system.

```
1   package com.greatfree.chat.message;
2
3   /*
4    * The class defines all of the integer constants that identify the messages in the chatting
5    * system. 04/15/2017, Bing Li
6    */
7
8   // Created: 04/15/2017, Bing Li
9   public class ChatMessageType
10  {
11      public final static int CS_CHAT_REGISTRY_REQUEST = 101;
12      public final static int CS_CHAT_REGISTRY_RESPONSE = 102;
13      public final static int CS_CHAT_PARTNER_REQUEST = 103;
14      public final static int CS_CHAT_PARTNER_RESPONSE = 104;
15
16      public final static int SHUTDOWN_CHAT_SERVER_NOTIFICATION = 105;
17
18      public final static int CS_ADD_PARTNER_NOTIFICATION = 106;
19
20      public final static int POLL_NEW_SESSIONS_REQUEST = 107;
21      public final static int POLL_NEW_SESSIONS_RESPONSE = 108;
22
23      public final static int POLL_NEW_CHATS_REQUEST = 109;
24      public final static int POLL_NEW_CHATS_RESPONSE = 110;
25
26      public final static int CS_CHAT_NOTIFICATION = 111;
27
28      public final static int PEER_CHAT_REGISTRY_REQUEST = 112;
29      public final static int PEER_CHAT_REGISTRY_RESPONSE = 113;
30
31      public final static int PEER_CHAT_PARTNER_REQUEST = 114;
32      public final static int PEER_CHAT_PARTNER_RESPONSE = 115;
33
34      public final static int PEER_ADD_PARTNER_NOTIFICATION = 116;
35
36      public final static int PEER_CHAT_NOTIFICATION = 117;
37
38      public final static int POLL_SERVER_STATUS_REQUEST = 118;
39      public final static int POLL_SERVER_STATUS_RESPONSE = 119;
40  }
```

List 5.11 The code of ChatMessageType.java

2.5 RD – The Pattern of Request Dispatcher: Processing Requests Concurrently

Anyway, this is not a new topic since you have programmed with the pattern in Chapter 4 although I do not indicate that explicitly in the chapter. Since the pattern is utilized a couple of times in the chatting server, I have to list all of the code here for your reference. You must understand the notion of GreatFree more deeply when you look at those similar code again.

2.5.1 The Pattern of Request Dispatcher

The pattern of Request Dispatcher (RD) is made up with five components, including the Request, the Output Message Stream (OMS), the Response, the Request Queue (RQ) and the Request Queue Creator (RQC), which can also be regarded as patterns. Each of them should be programmed when you are required to implement a new pair of request/response.

The request: The first component is the request, which extends the parent class, ServerMessage. It is the query message sent from a client to a server. List 5.12 shows the request of ChatRegistryRequest. Its structure is identical to List 4.2 in Chapter 4. It needs to derive the parent class, ServerMessage and its overall structure is a classic object enclosing data.

```
1   package com.greatfree.chat.message.cs;
2
3   import com.greatfree.chat.message.ChatMessageType;
4   import com.greatfree.message.ServerMessage;
5
6   /*
7    * The message encloses the data to register one account in the chatting system. 04/15/2017, Bing Li
8    */
9
10  // Created: 04/15/2017, Bing Li
11  public class ChatRegistryRequest extends ServerMessage
12  {
13      private static final long serialVersionUID = -4329938193364238991L;
14
15      // The user key of the account. 04/15/2017, Bing Li
16      private String userKey;
17      // The user name of the account. 04/15/2017, Bing Li
18      private String userName;
19      // The description about the user. 04/16/2017, Bing Li
20      private String description;
21
22      public ChatRegistryRequest(String userKey, String userName, String description)
23      {
24          super(ChatMessageType.CS_CHAT_REGISTRY_REQUEST);
25          this.userKey = userKey;
26          this.userName = userName;
27          this.description = description;
28      }
29
30      public String getUserKey()
31      {
32          return this.userKey;
33      }
34
35      public String getUserName()
36      {
37          return this.userName;
38      }
39
40      public String getDescription()
41      {
42          return this.description;
43      }
44  }
```

List 5.12 The code of ChatRegistryRequest.java

The output message stream: The second one is the output message stream, which is responsible for sending the generated response from the server to the client using Java

supported API, ObjectOutputStream [2]. List 5.13 show the output message stream sample of ChatRegistryStream. It should inherit the parent class, OutMessageStream.

```
1   package com.greatfree.chat.message.cs;
2
3   import java.io.ObjectOutputStream;
4   import java.util.concurrent.locks.Lock;
5
6   import com.greatfree.remote.OutMessageStream;
7
8   /*
9    * The class encloses the output stream to send the response of ChatRegistryResponse
10   * to the client. 04/15/2017, Bing Li
11   */
12
13  // Created: 04/15/2017, Bing Li
14  public class ChatRegistryStream extends OutMessageStream<ChatRegistryRequest>
15  {
16      public ChatRegistryStream(ObjectOutputStream out, Lock lock, ChatRegistryRequest message)
17      {
18          super(out, lock, message);
19      }
20  }
```

List 5.13 The code of ChatRegistryStream.java

The response: The third component of RD extends ServerMessage as well. It is the message generated by the server to answer the request correspondingly. List 5.14 illustrates the code of ChatRegistryResponse.java, which is a response for the request of ChatRegistryRequest.

```
1   package com.greatfree.chat.message.cs;
2
3   import com.greatfree.chat.message.ChatMessageType;
4   import com.greatfree.message.ServerMessage;
5
6   /*
7    * The response is used to respond to the client node about the result of the account
8    * registry. 04/15/2017, Bing Li
9    */
10
11  // Created: 04/15/2017, Bing Li
12  public class ChatRegistryResponse extends ServerMessage
13  {
14      private static final long serialVersionUID = 6301739846673781822L;
15
16      // The result of one account's registry. 04/15/2017, Bing Li
17      private boolean isSucceeded;
18
19      public ChatRegistryResponse(boolean isSucceeded)
20      {
21          super(ChatMessageType.CS_CHAT_REGISTRY_RESPONSE);
22          this.isSucceeded = isSucceeded;
23      }
24
25      public boolean isSucceeded()
26      {
27          return this.isSucceeded;
28      }
29  }
```

List 5.14 The code of ChatRegistryResponse.java

The request queue: The four component of RD is the request queue which is a thread to process the queued request sequentially. You need to set up the length of the queue

carefully. If it is too long, the concurrency of processing the request is closed to the sequential manner. It is too short, although the degree of the concurrency is high, more threads have to be consumed and it might result in the high cost of computing resources. Developers should determine a proper value according to the practical situation in your computing environment. List 5.15 shows one example of the request queue that serves for the request, ChatRegistryRequest, and the response, ChatRegistryResponse. Repeatedly, it employs another important pattern DWC (Double While Concurrency), as we discuss on Chapter 3 and Chapter 4. Table 5.11 gives a summary about the pattern of DWC for responding chatting registry requests. In short, the pattern of DWC is an effective one when processing remote incoming messages concurrently on the server side.

```
1   package com.greatfree.chat.server;
2
3   import java.io.IOException;
4
5   import com.greatfree.chat.message.cs.ChatRegistryRequest;
6   import com.greatfree.chat.message.cs.ChatRegistryResponse;
7   import com.greatfree.chat.message.cs.ChatRegistryStream;
8   import com.greatfree.concurrency.RequestQueue;
9   import com.greatfree.testing.data.ServerConfig;
10
11  /*
12   * The thread registers the chatting account concurrently. 04/15/2017, Bing Li
13   */
14
15  // Created: 04/15/2017, Bing Li
16  public class ChatRegistryThread extends RequestQueue<ChatRegistryRequest,
17      ChatRegistryStream, ChatRegistryResponse>
18  {
19
20      public ChatRegistryThread(int maxTaskSize)
21      {
22          super(maxTaskSize);
23      }
24
25      @Override
26      public void run()
27      {
28          ChatRegistryStream request;
29          ChatRegistryResponse response;
30          while (!this.isShutdown())
31          {
32              while (!this.isEmpty())
33              {
34                  request = this.getRequest();
35
36                  AccountRegistry.CS().add(new CSAccount(request.getMessage().getUserKey(),
37                      request.getMessage().getUserName(), request.getMessage().getDescription()));
38
39                  response = new ChatRegistryResponse(true);
40                  try
41                  {
42                      this.respond(request.getOutStream(), request.getLock(), response);
43                  }
44                  catch (IOException e)
45                  {
46                      e.printStackTrace();
47                  }
48                  this.disposeMessage(request, response);
49              }
50              try
51              {
52                  // Wait for some time when the queue is empty. During the period and before the thread is killed,
53                  // some new requests might be received. If so, the thread can keep working. 02/15/2016, Bing Li
54                  this.holdOn(ServerConfig.REQUEST_THREAD_WAIT_TIME);
55              }
56              catch (InterruptedException e)
57              {
58                  e.printStackTrace();
```

```
59                }
60            }
61        }
62    }
```

List 5.15 The code of ChatRegistryThread.java

Pattern: DWC (Double While Concurrency)		
Class	ChatRegistryThread	
Singleton	No	
Method	void dispose()	
	void enqueue(ChatRegistryStream request)	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
	ChatRegistryStream getRequest()	
	void respond(OutputObjectStream out, Lock outLock, ChatRegistryResponse response)	
Enclosed Pattern	Request	ChatRegistryRequest
	OMS (Output Message Stream)	ChatRegistryStream
	Response	ChatRegistryResponse
Employed API	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	OutMessageStream<Message extends ServerMessage>	
	ServerMessage	

Table 5.11 The pattern of DWC for processing and responding chatting registry requests

Pattern: RD (Request Dispatcher)		
Class	RequestDispatcher<ChatRegistryRequest, ChatRegistryStream, ChatRegistryResponse, ChatRegistryThread, ChatRegistryThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(ChatRegistryStream request)	
Enclosed Pattern	Request	ChatRegistryRequest
	OMS (Output Message Stream)	ChatRegistryStream
	Response	ChatRegistryResponse
	RQ (Request Queue)	ChatRegistryThread
	RQC (Request Queue Creator)	ChatRegistryThreadCreator
Employed API	ServerMessage	
	OutMessageStream<Message extends ServerMessage>	
	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	RequestThreadCreatable<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends RequestQueue<Request, Stream, Response>>	

Table 5.12 The pattern of RD for dispatching chatting registry requests

The request queue creator: The last component of RD is the request queue creator, which generates instances of the request queue when no sufficient threads are available. It is necessary to do so when a large number of requests should be processed and no idle request queues exist. List 5.16 shows the code of ChatRegistryThreadCreator.java, which is the request queue creator for RD to process the request, ChatRegistryRequest, and the response, ChatRegistryResponse. Table 5.12 summarizes the pattern of RD for processing chatting requests. Similarly, this is a generic pattern that can be employed in any cases when programming the code to dispatch remote requests concurrently.

```
1    package com.greatfree.chat.server;
2
```

```
3   import com.greatfree.chat.message.cs.ChatRegistryRequest;
4   import com.greatfree.chat.message.cs.ChatRegistryResponse;
5   import com.greatfree.chat.message.cs.ChatRegistryStream;
6   import com.greatfree.concurrency.RequestThreadCreatable;
7
8   /*
9    * The creator generates one instance of ChatRegistryThread. It is invoked by the thread management
10   * mechanism, RequestDispatcher. 04/15/2017, Bing Li
11   */
12
13  // Created: 04/15/2017, Bing Li
14  public class ChatRegistryThreadCreator implements RequestThreadCreatable<ChatRegistryRequest,
15      ChatRegistryStream, ChatRegistryResponse, ChatRegistryThread>
16  {
17
18      @Override
19      public ChatRegistryThread createRequestThreadInstance(int taskSize)
20      {
21          return new ChatRegistryThread(taskSize);
22      }
23  }
```

List 5.16 The code of ChatRegistryThreadCreator.java

2.5.2 The RD for Chatting Partner Request

The RD is made use of processing requests in a distributed environment. It is a very common pattern or idiom. In the chatting system, four instances of RD exist. Their code is listed as follows for your reference. For the reason of their highly patterned structure, it is not necessary to explain them line by line. Table 5.13 concludes the pattern for chatting partner request. According to Table 5.12 and Table 5.13, although they serve for different requests and responses, the code structures are identical. When programming the code to dispatch incoming requests on the server side, the pattern of RD is highly recommended to raise your productivity.

Pattern: RD (Request Dispatcher)		
Class	RequestDispatcher<ChatPartnerRequest, ChatPartnerStream, ChatPartnerResponse, ChatPartnerRequestThread, ChatPartnerRequestThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(ChatPartnerStream request)	
Enclosed Pattern	Request	ChatPartnerRequest
	OMS (Output Message Stream)	ChatPartnerStream
	Response	ChatPartnerResponse
	RQ (Request Queue)	ChatPartnerRequestThread
	RQC (Request Queue Creator)	ChatPartnerRequestThreadCreator
Employed API	ServerMessage	
	OutMessageStream<Message extends ServerMessage>	
	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	RequestThreadCreatable<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends RequestQueue<Request, Stream, Response>>	

Table 5.13 The RD pattern for dispatching chatting partner requests

List 5.17 shows the request of ChatPartnerRequest, which contains one key to search whether such a user exists in the chatting system.

```
1   package com.greatfree.chat.message.cs;
2
```

```
3   import com.greatfree.chat.message.ChatMessageType;
4   import com.greatfree.message.ServerMessage;
5
6   /*
7    * The request asks the chatting registry server to obtain the potential chatting partners. 04/16/2017, Bing Li
8    */
9
10  // Created: 04/16/2017, Bing Li
11  public class ChatPartnerRequest extends ServerMessage
12  {
13      private static final long serialVersionUID = -1587573982719535239L;
14
15      private String userKey;
16
17      public ChatPartnerRequest(String userKey)
18      {
19          super(ChatMessageType.CS_CHAT_PARTNER_REQUEST);
20          this.userKey = userKey;
21      }
22
23      public String getUserKey()
24      {
25          return this.userKey;
26      }
27  }
```

List 5.17 The code of ChatPartnerRequest.java

List 5.18 presents the code of ChatParnterStream, which the output message stream of the RD for the request, ChatPartnerRequest, and the response, ChatPartnerResponse.

```
1   package com.greatfree.chat.message.cs;
2
3   import java.io.ObjectOutputStream;
4   import java.util.concurrent.locks.Lock;
5
6   import com.greatfree.remote.OutMessageStream;
7
8   /*
9    * The class encloses the output stream to send the response of ChatPartnerResponse
10   * to the client. 04/16/2017, Bing Li
11   */
12
13  // Created: 04/16/2017, Bing Li
14  public class ChatPartnerStream extends OutMessageStream<ChatPartnerRequest>
15  {
16      public ChatPartnerStream(ObjectOutputStream out, Lock lock, ChatPartnerRequest message)
17      {
18          super(out, lock, message);
19      }
20  }
```

List 5.18 The code of ChatPartnerStream.java

List 5.19 shows the code of ChatPartnerResponse.java, which is the response for the request of ChatPartnerRequest. If the searched user exists, the description about the user is contained in the response and replied to the querying client.

```
1   package com.greatfree.chat.message.cs;
2
3   import com.greatfree.chat.message.ChatMessageType;
4   import com.greatfree.message.ServerMessage;
5
6   /*
7    * The response contains the account description. 04/16/2017, Bing Li
8    */
9
10  // Created: 04/16/2017, Bing Li
```

```
11  public class ChatPartnerResponse extends ServerMessage
12  {
13      private static final long serialVersionUID = 2055070734600953034L;
14
15      private String userKey;
16      private String userName;
17      private String description;
18
19      public ChatPartnerResponse(String userKey, String userName, String description)
20      {
21          super(ChatMessageType.CS_CHAT_PARTNER_RESPONSE);
22          this.userKey = userKey;
23          this.userName = userName;
24          this.description = description;
25      }
26
27      public String getUserKey()
28      {
29          return this.userKey;
30      }
31
32      public String getUserName()
33      {
34          return this.userName;
35      }
36
37      public String getDescription()
38      {
39          return this.description;
40      }
41  }
```

List 5.19 The code of ChatPartnerResponse.java

List 5.20 presents the code of the request queue, the most important component of the RD. It should derive the parent class, RequestQueue, as well as utilize the DWC. You have seen such a code many times in List 4.8 and List 4.9 of Chapter 4 as well as List 5.15 of this chapter. Table 5.14 gives a summary of the DWC. Compared with Table 5.11, their structures are identical although they respond to clients with different messages. So, the pattern of DWC is great for you to programming the code of processing and responding requests on the server side.

Pattern: DWC (Double While Concurrency)		
Class	ChatPartnerRequestThread	
Singleton	No	
Method	void dispose()	
	void enqueue(ChatPartnerStream request)	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
	ChatPartnerStream getRequest()	
	void respond(OutputObjectStream out, Lock outLock, ChatPartnerResponse response)	
Enclosed Pattern	Request	ChatPartnerRequest
	OMS (Output Message Stream)	ChatPartnerStream
	Response	ChatPartnerResponse
Employed API	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	OutMessageStream<Message extends ServerMessage>	
	ServerMessage	

Table 5.14 The DWC pattern for processing and responding chatting partner requests

```
1  package com.greatfree.chat.server;
2
3  import java.io.IOException;
4
5  import com.greatfree.chat.message.cs.ChatPartnerRequest;
```

```
6   import com.greatfree.chat.message.cs.ChatPartnerResponse;
7   import com.greatfree.chat.message.cs.ChatPartnerStream;
8   import com.greatfree.concurrency.RequestQueue;
9   import com.greatfree.testing.data.ServerConfig;
10  import com.greatfree.util.UtilConfig;
11
12  /*
13   * The thread checks the retrieved user of chatting. 04/16/2017, Bing Li
14   */
15
16  // Created: 04/16/2017, Bing Li
17  public class ChatPartnerRequestThread extends RequestQueue<ChatPartnerRequest,
18      ChatPartnerStream, ChatPartnerResponse>
19  {
20
21      public ChatPartnerRequestThread(int maxTaskSize)
22      {
23          super(maxTaskSize);
24      }
25
26      @Override
27      public void run()
28      {
29          ChatPartnerStream request;
30          ChatPartnerResponse response;
31          CSAccount account;
32          while (!this.isShutdown())
33          {
34              while (!this.isEmpty())
35              {
36                  request = this.getRequest();
37                  // Check whether the account is existed or not? Bing Li
38                  if (AccountRegistry.CS().isAccountExisted(request.getMessage().getUserKey()))
39                  {
40                      // Get the account. 04/16/2017, Bing Li
41                      account = AccountRegistry.CS().getAccount(request.getMessage().getUserKey());
42                      // Generate the response. 04/16/2017, Bing Li
43                      response = new ChatPartnerResponse(account.getUserKey(), account.getUserName(),
44                          account.getDescription());
45                  }
46                  else
47                  {
48                      // Generate the response. 04/16/2017, Bing Li
49                      response = new ChatPartnerResponse(UtilConfig.EMPTY_STRING,
50                          UtilConfig.EMPTY_STRING, UtilConfig.EMPTY_STRING);
51                  }
52
53                  try
54                  {
55                      // Respond to the client. 04/16/2017, Bing Li
56                      this.respond(request.getOutStream(), request.getLock(), response);
57                  }
58                  catch (IOException e)
59                  {
60                      e.printStackTrace();
61                  }
62                  this.disposeMessage(request, response);
63              }
64              try
65              {
66                  // Wait for some time when the queue is empty. During the period and before the thread is killed,
67                  // some new requests might be received. If so, the thread can keep working. 02/15/2016, Bing Li
68                  this.holdOn(ServerConfig.REQUEST_THREAD_WAIT_TIME);
69              }
70              catch (InterruptedException e)
71              {
72                  e.printStackTrace();
73              }
74          }
75      }
76  }
```

List 5.20 The code of ChatPartnerRequestThread.java

List 5.21 illustrates the last component, the request queue creator, of the RD for the case of ChatPartnerRequest. Your must not feel weird for the code right now, right?

```
1   package com.greatfree.chat.server;
2
3   import com.greatfree.chat.message.cs.ChatPartnerRequest;
4   import com.greatfree.chat.message.cs.ChatPartnerResponse;
5   import com.greatfree.chat.message.cs.ChatPartnerStream;
6   import com.greatfree.concurrency.RequestThreadCreatable;
7
8   /*
9    * The creator generates one instance of ChatPartnerRequestThread. It is invoked by the thread management
10   * mechanism, RequestDispatcher. 04/16/2017, Bing Li
11   */
12
13  // Created: 04/16/2017, Bing Li
14  public class ChatPartnerRequestThreadCreator implements RequestThreadCreatable
15      <ChatPartnerRequest, ChatPartnerStream, ChatPartnerResponse, ChatPartnerRequestThread>
16  {
17      @Override
18      public ChatPartnerRequestThread createRequestThreadInstance(int taskSize)
19      {
20          return new ChatPartnerRequestThread(taskSize);
21      }
22  }
```

List 5.21 The code of ChatPartnerRequestThreadCreator.java

2.5.3 The RD for Polling New Sessions

For a chatting system based on the distributed model of C/S, it is necessary to perform the interactions between clients with polling operations. In the sample chatting implementation, one of the polling aims to make sure whether a chatting session exists or not. A chatting session is set up once if one user adds another one as his chatting partner. Only if the session is created, it is possible for them to chat with one another. Table 5.15 illustrates the pattern for polling new chatting sessions. Again, the RD pattern is great to be always employed for the server side to dispatch requests concurrently.

Pattern: RD (Request Dispatcher)		
Class	RequestDispatcher<PollNewSessionsRequest, PollNewSessionsStream, PollNewSessionsResponse, PollNewSessionsThread, PollNewSessionsThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(PollNewSessionsStream request)	
Enclosed Pattern	Request	PollNewSessionsRequest
	OMS (Output Message Stream)	PollNewSessionsStream
	Response	PollNewSessionsResponse
	RQ (Request Queue)	PollNewSessionsThread
	RQC (Request Queue Creator)	PollNewSessionsThreadCreator
Employed API	ServerMessage	
	OutMessageStream<Message extends ServerMessage>	
	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	RequestThreadCreatable<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends RequestQueue<Request, Stream, Response>>	

Table 5.15 The pattern of RD for dispatching the requests of polling new chatting sessions

List 5.22 shows the code of PollNewSessionsRequest.java. It sends from a client to the chatting server periodically to ensure whether a new chatting session is created or not. The request contains the potential chatting receiver key, receiverKey, i.e., the user who sends the polling request, and the user who raises the session, username.

```
1    package com.greatfree.chat.message.cs;
2
3    import com.greatfree.chat.message.ChatMessageType;
4    import com.greatfree.message.ServerMessage;
5
6    /*
7     * The request is sent to the chatting server to check whether new sessions are available to the request
8     * sender periodically. 04/24/2017, Bing Li
9     */
10
11   // Created: 04/24/2017, Bing Li
12   public class PollNewSessionsRequest extends ServerMessage
13   {
14       private static final long serialVersionUID = 6979466184161327628L;
15
16       private String receiverKey;
17       private String username;
18
19       public PollNewSessionsRequest(String receiverKey, String username)
20       {
21           super(ChatMessageType.POLL_NEW_SESSIONS_REQUEST);
22           this.receiverKey = receiverKey;
23           this.username = username;
24       }
25
26       public String getReceiverKey()
27       {
28           return this.receiverKey;
29       }
30
31       public String getUsername()
32       {
33           return this.username;
34       }
35   }
```

List 5.22 The code of PollNewSessionsRequest.java

List 5.23 demonstrates the code of the output message stream for the pattern of RD.

```
1    package com.greatfree.chat.message.cs;
2
3    import java.io.ObjectOutputStream;
4    import java.util.concurrent.locks.Lock;
5
6    import com.greatfree.remote.OutMessageStream;
7
8    // Created: 04/24/2017, Bing Li
9    public class PollNewSessionsStream extends OutMessageStream<PollNewSessionsRequest>
10   {
11       public PollNewSessionsStream(ObjectOutputStream out, Lock lock, PollNewSessionsRequest message)
12       {
13           super(out, lock, message);
14       }
15   }
```

List 5.23 The code of PollNewSessionsStream.java

List 5.24 presents the code of the response, PollNewSessionsResponse. If sessions exists, their keys are enclosed in the response and replied to the client.

```
1   package com.greatfree.chat.message.cs;
2
3   import java.util.Set;
4
5   import com.greatfree.chat.message.ChatMessageType;
6   import com.greatfree.message.ServerMessage;
7
8   /*
9    * The response tells the request sender whether a new session is available. If so, the session keys are
10   * sent to the request sender. If not, a null message is responded. 04/24/2017, Bing Li
11   */
12
13  // Created: 04/24/2017, Bing Li
14  public class PollNewSessionsResponse extends ServerMessage
15  {
16      private static final long serialVersionUID = -4911768419106384745L;
17
18      private Set<String> newSessionKeys;
19
20      public PollNewSessionsResponse(Set<String> newSessionKeys)
21      {
22          super(ChatMessageType.POLL_NEW_SESSIONS_RESPONSE);
23          this.newSessionKeys = newSessionKeys;
24      }
25
26      public Set<String> getNewSessionKeys()
27      {
28          return this.newSessionKeys;
29      }
30  }
```

List 5.24 The code of PollNewSessionsResponse.java

List 5.25 illustrates the most important component of the RD for PollNewSessionsRequest, PollNewSessionsThread.java. It retrieves its local in-memory storage to check whether such sessions exist or not. If it really exists, the response is created and responded to the user. If not, a blank response is formed as well. Although I feel tired to draw a table like the ones of Table 5.11 and Table 5.14 again, I repeat the job, Table 5.16, again to show the effectiveness of the pattern of DWC in terms of accelerating your programming.

Pattern: DWC (Double While Concurrency)		
Class	PollNewSessionsThread	
Singleton	No	
Method	void dispose()	
	void enqueue(PollNewSessionsStream request)	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
	PollNewSessionsStream getRequest()	
	void respond(OutputObjectStream out, Lock outLock, PollNewSessionsResponse response)	
Enclosed Pattern	Request	PollNewSessionsRequest
	OMS (Output Message Stream)	PollNewSessionsStream
	Response	PollNewSessionsResponse
Employed API	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	OutMessageStream<Message extends ServerMessage>	
	ServerMessage	

Table 5.16 The pattern of DWC for processing and responding the request, PollNewSessionsRequest

```
1   package com.greatfree.chat.server;
2
3   import java.io.IOException;
4
```

```
5   import com.greatfree.chat.PrivateChatSessions;
6   import com.greatfree.chat.message.cs.PollNewSessionsRequest;
7   import com.greatfree.chat.message.cs.PollNewSessionsResponse;
8   import com.greatfree.chat.message.cs.PollNewSessionsStream;
9   import com.greatfree.concurrency.RequestQueue;
10  import com.greatfree.testing.data.ServerConfig;
11
12  /*
13   * The thread processes PollNewSessionsRequest to ensure whether new sessions are
14   * available or not. 04/24/2017, Bing Li
15   */
16
17  // Created: 04/24/2017, Bing Li
18  public class PollNewSessionsThread extends RequestQueue<PollNewSessionsRequest,
19      PollNewSessionsStream, PollNewSessionsResponse>
20  {
21      public PollNewSessionsThread(int maxTaskSize)
22      {
23          super(maxTaskSize);
24      }
25
26      @Override
27      public void run()
28      {
29          PollNewSessionsStream request;
30          PollNewSessionsResponse response;
31          while (!this.isShutdown())
32          {
33              while (!this.isEmpty())
34              {
35                  request = this.getRequest();
36                  if (PrivateChatSessions.HUNGARY().isSessionExisted
37                      (request.getMessage().getReceiverKey()))
38                  {
39                      // Check whether a new exists. 05/24/2017, Bing Li
40                      response = new PollNewSessionsResponse(PrivateChatSessions.HUNGARY()
41                          .getSessionKeys(request.getMessage().getReceiverKey()));
42                      // The polled session keys should be removed. It is not necessary. It depends on how to
43                      // design your application. 05/24/2017, Bing Li
44                      PrivateChatSessions.HUNGARY().removeSession(request.getMessage().getReceiverKey());
45                  }
46                  else
47                  {
48                      response = new PollNewSessionsResponse(null);
49                  }
50                  try
51                  {
52                      this.respond(request.getOutStream(), request.getLock(), response);
53                  }
54                  catch (IOException e)
55                  {
56                      e.printStackTrace();
57                  }
58                  this.disposeMessage(request, response);
59              }
60              try
61              {
62                  // Wait for some time when the queue is empty. During the period and before the thread is killed,
63                  // some new requests might be received. If so, the thread can keep working. 02/15/2016, Bing Li
64                  this.holdOn(ServerConfig.REQUEST_THREAD_WAIT_TIME);
65              }
66              catch (InterruptedException e)
67              {
68                  e.printStackTrace();
69              }
70          }
71      }
72  }
```

List 5.25 The code of PollNewSessionsThread.java

List 5.26 shows the creator that generates the instance of PollNewSessionsThreadCreator to support concurrency.

```
1    package com.greatfree.chat.server;
2
3    import com.greatfree.chat.message.cs.PollNewSessionsRequest;
4    import com.greatfree.chat.message.cs.PollNewSessionsResponse;
5    import com.greatfree.chat.message.cs.PollNewSessionsStream;
6    import com.greatfree.concurrency.RequestThreadCreatable;
7
8    // Created: 04/24/2017, Bing Li
9    public class PollNewSessionsThreadCreator implements RequestThreadCreatable
10       <PollNewSessionsRequest, PollNewSessionsStream, PollNewSessionsResponse,
11       PollNewSessionsThread>
12   {
13       @Override
14       public PollNewSessionsThread createRequestThreadInstance(int taskSize)
15       {
16           return new PollNewSessionsThread(taskSize);
17       }
18   }
```

List 5.26 The code of PollNewSessionsThreadCreator.java

2.5.4 The RD for Polling New Chatting Messages

Another most often used RD is the one for polling new chatting messages. As discussed previously, this is the unique way for two clients to interact with each other since they cannot send messages to one another directly. They have to get assistance from the chatting server through sending the polling request to detect whether their respective partner ever keeps chatting messages in one chatting session on the chatting server. Table 5.17 summarizes the RD pattern to dispatch polling new chatting messages.

Pattern: RD (Request Dispatcher)		
Class	RequestDispatcher<PollNewChatsRequest, PollNewChatsStream, PollNewChatsResponse, PollNewChatsThread, PollNewChatsThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(PollNewChatsStream request)	
Enclosed Pattern	Request	PollNewChatsRequest
	OMS (Output Message Stream)	PollNewChatsStream
	Response	PollNewChatsResponse
	RQ (Request Queue)	PollNewChatsThread
	RQC	PollNewChatsThreadCreator
Employed API	ServerMessage	
	OutMessageStream<Message extends ServerMessage>	
	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	RequestThreadCreatable<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends RequestQueue<Request, Stream, Response>>	

Table 5.17 The RD pattern for dispatching polling new chatting messages

List 5.27 illustrates the code for the request, PollNewChatsRequest.java. It checks the new chatting messages with the detected chatting session key obtained by another polling request, PollNewSessionsRequest discussed in Section 2.5.3. In addition, the chatting partner's key, receiverKey, and the name, username, are also enclosed in the request for the polling.

```
1    package com.greatfree.chat.message.cs;
```

```
2
3   import com.greatfree.chat.message.ChatMessageType;
4   import com.greatfree.message.ServerMessage;
5
6   /*
7    * The request is used to check the chatting server whether new chatting messages are available. It is
8    * sent to the chatting server periodically. If so, the new messages must be returned to the local
9    * user. 4/23/2017, Bing Li
10   */
11
12  // Created: 04/23/2017, Bing Li
13  public class PollNewChatsRequest extends ServerMessage
14  {
15      private static final long serialVersionUID = -6869452383641427790L;
16
17      // The chat session to be checked. 04/24/2017, Bing Li
18      private String chatSessionKey;
19      // The chat messages' receiver. 04/24/2017, Bing Li
20      private String receiverKey;
21      // The username of the receiver. 04/27/2017, Bing Li
22      private String username;
23
24      public PollNewChatsRequest(String chatSessionKey, String receiverKey, String username)
25      {
26          super(ChatMessageType.POLL_NEW_CHATS_REQUEST);
27          this.chatSessionKey = chatSessionKey;
28          this.receiverKey = receiverKey;
29          this.username = username;
30      }
31
32      public String getChatSessionKey()
33      {
34          return this.chatSessionKey;
35      }
36
37      public String getReceiverKey()
38      {
39          return this.receiverKey;
40      }
41
42      public String getUsername()
43      {
44          return this.username;
45      }
46  }
```

List 5.27 The code of PollNewChatsRequest.java

List 5.28 presents another component, the output message stream, of the RD, PollNewChatsStream. It is responsible for sending the generated response to the polling requestor.

```
1   package com.greatfree.chat.message.cs;
2
3   import java.io.ObjectOutputStream;
4   import java.util.concurrent.locks.Lock;
5
6   import com.greatfree.remote.OutMessageStream;
7
8   // Created: 04/23/2017, Bing Li
9   public class PollNewChatsStream extends OutMessageStream<PollNewChatsRequest>
10  {
11      public PollNewChatsStream(ObjectOutputStream out, Lock lock, PollNewChatsRequest message)
12      {
13          super(out, lock, message);
14      }
15  }
```

List 5.28 The code of PollNewChatsStream.java

List 5.29 demonstrates the response for the RD, PollNewChatsResponse. All of newly posted chatting messages are contained in the response.

```
1   package com.greatfree.chat.message.cs;
2
3   import java.util.List;
4
5   import com.greatfree.chat.ChatMessage;
6   import com.greatfree.chat.message.ChatMessageType;
7   import com.greatfree.message.ServerMessage;
8
9   /*
10   * If new chatting messages are available, they should be returned to the request sender. 04/23/2017, Bing Li
11   */
12
13  // Created: 04/23/2017, Bing Li
14  public class PollNewChatsResponse extends ServerMessage
15  {
16      private static final long serialVersionUID = -4869442313781358479L;
17
18      private List<ChatMessage> messages;
19
20      public PollNewChatsResponse(List<ChatMessage> messages)
21      {
22          super(ChatMessageType.POLL_NEW_CHATS_RESPONSE);
23          this.messages = messages;
24      }
25
26      public List<ChatMessage> getMessages()
27      {
28          return this.messages;
29      }
30  }
```

List 5.29 The code of PollNewChatsResponse.java

List 5.30 presents the code of PollNewChatsThread.java, which is the most important component, the request queue, for the RD. It retrieves the local storage, PrivateChatSessions, to detect whether new chatting messages are available. If new chatting messages are retrieved, they are responded to the polling requester after the response, which contains the messages, is created. If not, an empty response is created and sent back to the requester as well. Table 5.18 summarizes the DWC. You can follow Table 5.11, Table 5.14 and Table 5.16 as well such that you can memorize them deeply.

```
1   package com.greatfree.chat.server;
2
3   import java.io.IOException;
4   import java.util.ArrayList;
5   import java.util.List;
6
7   import com.greatfree.chat.ChatMessage;
8   import com.greatfree.chat.PrivateChatSessions;
9   import com.greatfree.chat.message.cs.PollNewChatsRequest;
10  import com.greatfree.chat.message.cs.PollNewChatsResponse;
11  import com.greatfree.chat.message.cs.PollNewChatsStream;
12  import com.greatfree.concurrency.RequestQueue;
13  import com.greatfree.testing.data.ServerConfig;
14
15  /*
16   * The thread checks whether new chats are available. If so, the chats should be responded to the sender of
17   * the request. If not, a null message is responded as well. 04/24/2017, Bing Li
18   */
19
20  // Created: 04/24/2017, Bing Li
21  public class PollNewChatsThread extends RequestQueue<PollNewChatsRequest, PollNewChatsStream,
```

```
22          PollNewChatsResponse>
23      {
24          public PollNewChatsThread(int maxTaskSize)
25          {
26              super(maxTaskSize);
27          }
28
29          @Override
30          public void run()
31          {
32              PollNewChatsStream request;
33              PollNewChatsResponse response;
34              List<ChatMessage> chatMessages;
35              while (!this.isShutdown())
36              {
37                  while (!this.isEmpty())
38                  {
39                      request = this.getRequest();
40                      // Check whether new chatting messages are available or not. 05/25/2017, Bing Li
41                      chatMessages = PrivateChatSessions.HUNGARY().getNewMessages
42                          (request.getMessage().getChatSessionKey(), request.getMessage().getReceiverKey());
43                      if (chatMessages != null)
44                      {
45                          response = new PollNewChatsResponse(chatMessages);
46                      }
47                      else
48                      {
49                          response = new PollNewChatsResponse(new ArrayList<ChatMessage>());
50                      }
51                      try
52                      {
53                          this.respond(request.getOutStream(), request.getLock(), response);
54                      }
55                      catch (IOException e)
56                      {
57                          e.printStackTrace();
58                      }
59                      this.disposeMessage(request, response);
60                  }
61                  try
62                  {
63                      // Wait for some time when the queue is empty. During the period and before the thread is killed,
64                      // some new requests might be received. If so, the thread can keep working. 02/15/2016, Bing Li
65                      this.holdOn(ServerConfig.REQUEST_THREAD_WAIT_TIME);
66                  }
67                  catch (InterruptedException e)
68                  {
69                      e.printStackTrace();
70                  }
71              }
72          }
73      }
```

List 5.30 The code of PollNewChatsThread.java

List 5.31 illustrates the creator to generate the instance of PollNewChatsThread, which is also one component of the RD.

```
1       package com.greatfree.chat.server;
2
3       import com.greatfree.chat.message.cs.PollNewChatsRequest;
4       import com.greatfree.chat.message.cs.PollNewChatsResponse;
5       import com.greatfree.chat.message.cs.PollNewChatsStream;
6       import com.greatfree.concurrency.RequestThreadCreatable;
7
8       // Created: 04/24/2017, Bing Li
9       public class PollNewChatsThreadCreator implements RequestThreadCreatable<PollNewChatsRequest,
10          PollNewChatsStream, PollNewChatsResponse, PollNewChatsThread>
11      {
12
13          @Override
14          public PollNewChatsThread createRequestThreadInstance(int taskSize)
```

```
15        {
16            return new PollNewChatsThread(taskSize);
17        }
18    }
```

List 5.31 The code of PollNewChatsThreadCreator.java

Pattern: DWC (Double While Concurrency)		
Class	PollNewChatsThread	
Singleton	No	
Method	void dispose()	
	void enqueue(PollNewChatsStream request)	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
	PollNewChatsStream getRequest()	
	void respond(OutputObjectStream out, Lock outLock, PollNewChatsResponse response)	
Enclosed Pattern	Request	PollNewChatsRequest
	OMS (Output Message Stream)	PollNewChatsStream
	Response	PollNewChatsResponse
Employed API	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	OutMessageStream<Message extends ServerMessage>	
	ServerMessage	

Table 5.18 The DWC pattern for processing and responding polling new chatting messages

2.6 ND – The Pattern of Notification Dispatcher: Processing Notifications Concurrently

Pattern: ND (Notification Dispatcher)		
Class	NotificationDispatcher<AddPartnerNotification, AddPartnerThread, AddPartnerThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(AddPartnerNotification notification)	
Enclosed Pattern	Notification	AddPartnerNotification
	NQ (Notification Queue)	AddPartnerThread
	NQC (Notification Queue Creator)	AddPartnerThreadCreator
Employed API	ServerMessage	
	NotificationQueue<Notification extends ServerMessage>	
	NotificationThreadCreatable<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>>	

Table 5.19 The pattern of ND for dispatching adding partner notifications

The message of notification is another type of data that is exchanged between two distributed nodes over the Internet. Different from the request/response, when a server receives such a message, it just processes it immediately without sending any responses to the sender of the client. From the client's point of view, after sending the notification, it keeps on working its own tasks without waiting for any responses from the server. For details of programming notifications, you can refer to Chapter 3. When doing that, you actually follow one important pattern, i.e., the Notification Dispatcher (ND).

The ND is made up with three components, including the Notification, the Notification Queue (NQ) and the Notification Queue Creator (NQC). The notification

is the message sent from one client to the server. The notification queue, which is a thread with a built-in task queue, is responsible for processing notifications concurrently. The third component, the notification queue creator is responsible for generating instances of the notification queue. Once if existing threads, i.e., the notification queues, are full and busy, a new notification queue is created by the component to take more notifications.

You must be familiar with them if you program with the technique discussed in Chapter 3. In the case of the chatting system, two instances of notifications exists. Let us take a look at them just briefly. Table 5.19 summarizes the pattern of ND according to the one for adding partner notifications. In general, when dispatching incoming notifications on the server side, you need to choose the one.

2.6.1 The ND for Adding Chatting Partner

List 5.32 shows the notification of the ND, AddPartnerNotification. The notification contains the sender key, localUserKey, the partner key, partnerKey, and the invitation message, invitation.

```
1   package com.greatfree.chat.message.cs;
2
3   import com.greatfree.chat.message.ChatMessageType;
4   import com.greatfree.message.ServerMessage;
5
6   /*
7    * The message encloses the chatting partner to be added. 04/15/2017, Bing Li
8    */
9
10  // Created: 04/23/2017, Bing Li
11  public class AddPartnerNotification extends ServerMessage
12  {
13      private static final long serialVersionUID = 2290167983632240615L;
14
15      private String localUserKey;
16      private String partnerKey;
17      private String invitation;
18
19      public AddPartnerNotification(String localUserKey, String partnerKey, String invitation)
20      {
21          super(ChatMessageType.CS_ADD_PARTNER_NOTIFICATION);
22          this.localUserKey = localUserKey;
23          this.partnerKey = partnerKey;
24          this.invitation = invitation;
25      }
26
27      public String getLocalUserKey()
28      {
29          return this.localUserKey;
30      }
31
32      public String getPartnerKey()
33      {
34          return this.partnerKey;
35      }
36
37      public String getInvitation()
38      {
39          return this.invitation;
40      }
41  }
```

List 5.32 The code of AddPartnerNotification.java

List 5.33 presents the notification queue, i.e., the thread with a built-in queue, which processes the incoming notification, AddPartnerNotification, concurrently. When such a notification is received, a new chatting session is created. It is necessary that this is a transformation of the pattern of DWC. Updated a little bit, the one is specially designed for dealing with incoming notifications. Compared with DWC for requests, it is more simple since it is not necessary to respond to clients. Table 5.20 summarizes the pattern. When processing remote notifications on the server side, just choose it.

```
1   package com.greatfree.chat.server;
2
3   import com.greatfree.chat.PrivateChatSessions;
4   import com.greatfree.chat.message.cs.AddPartnerNotification;
5   import com.greatfree.concurrency.NotificationQueue;
6   import com.greatfree.testing.data.ServerConfig;
7
8   /*
9    * The thread deals with adding friends notification from the client. 04/23/2017, Bing Li
10   */
11
12  // Created: 04/23/2017, Bing Li
13  public class AddPartnerThread extends NotificationQueue<AddPartnerNotification>
14  {
15      public AddPartnerThread(int taskSize)
16      {
17          super(taskSize);
18      }
19
20      @Override
21      public void run()
22      {
23          AddPartnerNotification notification;
24          while (!this.isShutdown())
25          {
26              while (!this.isEmpty())
27              {
28                  try
29                  {
30                      notification = this.getNotification();
31
32                      // When the notification is received, a new chatting session is created. 05/25/2017, Bing Li
33                      PrivateChatSessions.HUNGARY().addSession(notification.getPartnerKey(),
34                          notification.getLocalUserKey());
35
36                      this.disposeMessage(notification);
37                  }
38                  catch (InterruptedException e)
39                  {
40                      e.printStackTrace();
41                  }
42              }
43              try
44              {
45                  // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
46                  this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
47              }
48              catch (InterruptedException e)
49              {
50                  e.printStackTrace();
51              }
52          }
53      }
54  }
```

List 5.33 The code of AddPartnerThread.java

List 5.34 shows the thread creator for the ND. It creates instances of AddPartnerThread when no sufficient threads exist in the pool to raise the performance of the system.

```
1    package com.greatfree.chat.server;
2
3    import com.greatfree.chat.message.cs.AddPartnerNotification;
4    import com.greatfree.concurrency.NotificationThreadCreatable;
5
6    // Created: 04/23/2017, Bing Li
7    public class AddPartnerThreadCreator implements NotificationThreadCreatable
8        <AddPartnerNotification, AddPartnerThread>
9    {
10       @Override
11       public AddPartnerThread createNotificationThreadInstance(int taskSize)
12       {
13           return new AddPartnerThread(taskSize);
14       }
15   }
```

List 5.34 The code of AddPartnerThread.java

Pattern: DWC (Double While Concurrency)		
Class	AddPartnerThread	
Singleton	No	
Method	void dispose()	
	void enqueue(AddPartnerNotification notification)	
	AddPartnerNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	AddPartnerNotification
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 5.20 The DWC pattern for processing adding partner notifications

2.6.2 The ND for Chatting Notifications

When a client needs to chat with his partner, the chatting messages are sent to the chatting server and retain there until they are polled by the partner. Table 5.21 summarizes the pattern of ND to process incoming chatting notifications. Compared with Table 5.19, they are identical in terms of their code structures. So when dispatching incoming notifications on the server side, you need to choose the pattern of ND without doubt.

Pattern: ND (Notification Dispatcher)		
Class	NotificationDispatcher<ChatNotification, ChatThread, ChatThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(ChatNotification notification)	
Enclosed Pattern	Notification	ChatNotification
	NQ (Notification Queue)	ChatThread
	NQC (Notification Queue Creator)	ChatThreadCreator
Employed API	ServerMessage	
	NotificationQueue<Notification extends ServerMessage>	
	NotificationThreadCreatable<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>>	

Table 5.21 The pattern of ND to dispatch incoming chatting notifications

List 5.35 illustrates the notification of ChatNotification, which includes the new chatting message and other relevant information, such as the session key and the chatting users.

```
1   package com.greatfree.chat.message.cs;
2
3   import com.greatfree.chat.message.ChatMessageType;
4   import com.greatfree.message.ServerMessage;
5
6   // Created: 04/27/2017, Bing Li
7   public class ChatNotification extends ServerMessage
8   {
9       private static final long serialVersionUID = -2557671230941837747L;
10
11      // The chatting session key. 04/27/2017, Bing Li
12      private String sessionKey;
13      // The chatting message. 04/27/2017, Bing Li
14      private String message;
15      // The sender name. 04/27/2017, Bing Li
16      private String senderName;
17      // The sender key. 04/27/2017, Bing Li
18      private String senderKey;
19      // The receiver name. 04/27/2017, Bing Li
20      private String receiverName;
21      // The receiver key. 04/27/2017, Bing Li
22      private String receiverKey;
23
24      public ChatNotification(String sessionKey, String message, String senderKey, String senderName,
25          String receiverKey, String receiverName)
26      {
27          super(ChatMessageType.CS_CHAT_NOTIFICATION);
28          this.sessionKey = sessionKey;
29          this.message = message;
30          this.senderKey = senderKey;
31          this.senderName = senderName;
32          this.receiverKey = receiverKey;
33          this.receiverName = receiverName;
34      }
35
36      public String getSessionKey()
37      {
38          return this.sessionKey;
39      }
40
41      public String getMessage()
42      {
43          return this.message;
44      }
45
46      public String getSenderKey()
47      {
48          return this.senderKey;
49      }
50
51      public String getSenderName()
52      {
53          return this.senderName;
54      }
55
56      public String getReceiverKey()
57      {
58          return this.receiverKey;
59      }
60
61      public String getReceiverName()
62      {
63          return this.receiverName;
64      }
65  }
```

List 5.35 The code of ChatNotification.java