Master Go Distributed Tasks in 3 Steps! The Asynq Library Makes Async Jobs Simpler Than Ever

Язык оригинала: en

# Оригинал

How Many Pitfalls of Go Background Tasks Have You Encountered?  
In Go application development, there are always tasks that are  
not suitable for immediate execution  
, such as:  
Sending emails/SMS  
: Should users have to wait after clicking a button? That's a terrible experience!  
Computationally intensive tasks  
: Generating reports, data analysis—if the CPU is constantly occupied, will all other requests get stuck?  
Scheduled tasks  
: Running statistics at midnight, synchronizing data every hour—are you really going to write an infinite  
time.Sleep  
loop for this?  
Therefore, smart developers like us throw these tasks into an  
asynchronous task queue  
, letting background "workers" handle them.  
It sounds great in theory, but once you start, you might run into these pitfalls:  
Goroutine Explosion  
: When tasks pile up, you might be tempted to call  
go func()  
wildly, leading to thousands of unmanageable Goroutines and immense scheduling pressure.  
What if a Task Fails?  
: Due to network jitters or a service outage, is the task simply lost? A  
retry mechanism  
is essential!  
Chaotic Task Priorities  
: Should a payment notification be treated the same as a log entry? Of course not!  
Code Becomes a Mess  
: Defining tasks, serialization, registering handlers... when business logic gets entangled with queue management code, maintenance becomes a nightmare.  
If you've experienced  
any of the pain points above  
, congratulations,  
sasynq  
is the "antidote" you've been looking for!  
What is  
sasynq  
? And Why Can It Save You?  
sasynq  
is a  
super easy-to-use wrapper  
built on top of  
asynq  
. Asynq is a stable, efficient, Redis-based distributed task queue, and  
sasynq  
makes it even simpler and smoother to use.  
What are its advantages?  
✅  
Out-of-the-box  
: Supports Redis Cluster and Sentinel, eliminating single points of failure.  
✅  
Comprehensive Features  
: Priority queues, delayed tasks, deduplication, cancellation, and scheduled tasks are all supported.  
✅  
Safe and Reliable  
: With retries, timeouts, and deadlines, you'll never have to worry about losing tasks again.  
✅  
Extremely Simple API  
: Compared to the native  
asynq  
, writing code is more elegant and clear.  
In short,  
it makes complex things incredibly simple  
.  
How Easy is  
sasynq  
to Use? Let's See the Code!  
① Defining a Task  
sasynq  
makes task definition effortless.  
// example/common/task.go  
const  
TypeEmailSend  
=  
"email:send"  
// Task payload  
type  
EmailPayload  
struct  
{  
UserID  
int  
`json:"user\_id"`  
Message  
string  
`json:"message"`  
}  
// Task handler  
func  
HandleEmailTask  
(  
ctx  
context  
.  
Context  
,  
p  
\*  
EmailPayload  
)  
error  
{  
fmt  
.  
Printf  
(  
"[Email] Email sent successfully to user %d!  
\n  
"  
,  
p  
.  
UserID  
)  
return  
nil  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Isn't that clean? Just define the payload and the handler, and  
no more manual  
json.Unmarshal  
.  
② Enqueuing Tasks: The Ultimate Simplicity  
One-off Task Producer  
payload  
:=  
&  
common  
.  
EmailPayload  
{  
UserID  
:  
101  
,  
Message  
:  
"Important task!"  
}  
\_  
,  
\_  
,  
err  
:=  
client  
.  
EnqueueNow  
(  
common  
.  
TypeEmailSend  
,  
payload  
,  
sasynq  
.  
WithQueue  
(  
"critical"  
),  
sasynq  
.  
WithRetry  
(  
3  
),  
)  
Enter fullscreen mode  
Exit fullscreen mode  
EnqueueNow  
,  
EnqueueIn  
,  
EnqueueAt  
—the function names are self-explanatory! Use  
sasynq.WithXXX  
for method chaining to configure queues, retries, and deadlines in an intuitive and elegant way.  
Periodic Task Producer  
payload  
:=  
&  
common  
.  
EmailPayload  
{  
UserID  
:  
102  
,  
Message  
:  
"Periodic task!"  
}  
scheduler  
.  
RegisterTask  
(  
"@every 1m"  
,  
"request:url"  
,  
&  
payload  
)  
Enter fullscreen mode  
Exit fullscreen mode  
One line of code is all it takes to set up a scheduled task.  
③ Consuming Tasks: Register a Handler in One Line, and You're Done!  
srv  
:=  
sasynq  
.  
NewServer  
(  
redisCfg  
,  
sasynq  
.  
DefaultServerConfig  
())  
sasynq  
.  
RegisterTaskHandler  
(  
srv  
.  
Mux  
(),  
common  
.  
TypeEmailSend  
,  
sasynq  
.  
HandleFunc  
(  
common  
.  
HandleEmailTask  
))  
srv  
.  
Run  
()  
Enter fullscreen mode  
Exit fullscreen mode  
No redundant code.  
Register → Run → Done  
!  
Canceling a Task  
For a one-off, pending task:  
inspector  
.  
CancelTask  
(  
queue  
,  
taskID  
)  
Enter fullscreen mode  
Exit fullscreen mode  
For a periodic task:  
scheduler  
.  
Unregister  
(  
entryID  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Cancel a task with a single line of code.  
Summary: Why Choose  
sasynq  
?  
Simpler  
: Minimalist API design and clear code structure.  
More Powerful  
: Fully covers all common scenarios with support for retries, deduplication, delays, scheduling, and priorities.  
Safer  
: Deadlines, timeouts, and retry strategies provide better control over task processing.  
If you're looking for a  
simple, efficient, and feature-rich  
asynchronous task solution for your Go project,  
sasynq  
is the best choice.  
sasynq  
URL: github.com/go-dev-frame/sponge/pkg/sasynq  
sasynq  
is a sub-component of the Sponge framework. Sponge is a powerful and easy-to-use Go development framework based on the core principle of "Definition is Code." It helps developers easily build stable, reliable, and high-performance backend services (including RESTful API, gRPC, HTTP+gRPC, gRPC Gateway, etc.) in a "low-code" manner.  
👉  
Sponge Project URL  
:  
https://github.com/go-dev-frame/sponge

# Перевод на русский

How Many Pitfalls of Go Background Tasks Have You Encountered?  
In Go application development, there are always tasks that are  
not suitable for immediate execution  
, such as:  
Sending emails/SMS  
: Should users have to wait after clicking a button? That's a terrible experience!  
Computationally intensive tasks  
: Generating reports, data analysis—if the CPU is constantly occupied, will all other requests get stuck?  
Scheduled tasks  
: Running statistics at midnight, synchronizing data every hour—are you really going to write an infinite  
time.Sleep  
loop for this?  
Therefore, smart developers like us throw these tasks into an  
asynchronous task queue  
, letting background "workers" handle them.  
It sounds great in theory, but once you start, you might run into these pitfalls:  
Goroutine Explosion  
: When tasks pile up, you might be tempted to call  
go func()  
wildly, leading to thousands of unmanageable Goroutines and immense scheduling pressure.  
What if a Task Fails?  
: Due to network jitters or a service outage, is the task simply lost? A  
retry mechanism  
is essential!  
Chaotic Task Priorities  
: Should a payment notification be treated the same as a log entry? Of course not!  
Code Becomes a Mess  
: Defining tasks, serialization, registering handlers... when business logic gets entangled with queue management code, maintenance becomes a nightmare.  
If you've experienced  
any of the pain points above  
, congratulations,  
sasynq  
is the "antidote" you've been looking for!  
What is  
sasynq  
? And Why Can It Save You?  
sasynq  
is a  
super easy-to-use wrapper  
built on top of  
asynq  
. Asynq is a stable, efficient, Redis-based distributed task queue, and  
sasynq  
makes it even simpler and smoother to use.  
What are its advantages?  
✅  
Out-of-the-box  
: Supports Redis Cluster and Sentinel, eliminating single points of failure.  
✅  
Comprehensive Features  
: Priority queues, delayed tasks, deduplication, cancellation, and scheduled tasks are all supported.  
✅  
Safe and Reliable  
: With retries, timeouts, and deadlines, you'll never have to worry about losing tasks again.  
✅  
Extremely Simple API  
: Compared to the native  
asynq  
, writing code is more elegant and clear.  
In short,  
it makes complex things incredibly simple  
.  
How Easy is  
sasynq  
to Use? Let's See the Code!  
① Defining a Task  
sasynq  
makes task definition effortless.  
// example/common/task.go  
const  
TypeEmailSend  
=  
"email:send"  
// Task payload  
type  
EmailPayload  
struct  
{  
UserID  
int  
`json:"user\_id"`  
Message  
string  
`json:"message"`  
}  
// Task handler  
func  
HandleEmailTask  
(  
ctx  
context  
.  
Context  
,  
p  
\*  
EmailPayload  
)  
error  
{  
fmt  
.  
Printf  
(  
"[Email] Email sent successfully to user %d!  
\n  
"  
,  
p  
.  
UserID  
)  
return  
nil  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Isn't that clean? Just define the payload and the handler, and  
no more manual  
json.Unmarshal  
.  
② Enqueuing Tasks: The Ultimate Simplicity  
One-off Task Producer  
payload  
:=  
&  
common  
.  
EmailPayload  
{  
UserID  
:  
101  
,  
Message  
:  
"Important task!"  
}  
\_  
,  
\_  
,  
err  
:=  
client  
.  
EnqueueNow  
(  
common  
.  
TypeEmailSend  
,  
payload  
,  
sasynq  
.  
WithQueue  
(  
"critical"  
),  
sasynq  
.  
WithRetry  
(  
3  
),  
)  
Enter fullscreen mode  
Exit fullscreen mode  
EnqueueNow  
,  
EnqueueIn  
,  
EnqueueAt  
—the function names are self-explanatory! Use  
sasynq.WithXXX  
for method chaining to configure queues, retries, and deadlines in an intuitive and elegant way.  
Periodic Task Producer  
payload  
:=  
&  
common  
.  
EmailPayload  
{  
UserID  
:  
102  
,  
Message  
:  
"Periodic task!"  
}  
scheduler  
.  
RegisterTask  
(  
"@every 1m"  
,  
"request:url"  
,  
&  
payload  
)  
Enter fullscreen mode  
Exit fullscreen mode  
One line of code is all it takes to set up a scheduled task.  
③ Consuming Tasks: Register a Handler in One Line, and You're Done!  
srv  
:=  
sasynq  
.  
NewServer  
(  
redisCfg  
,  
sasynq  
.  
DefaultServerConfig  
())  
sasynq  
.  
RegisterTaskHandler  
(  
srv  
.  
Mux  
(),  
common  
.  
TypeEmailSend  
,  
sasynq  
.  
HandleFunc  
(  
common  
.  
HandleEmailTask  
))  
srv  
.  
Run  
()  
Enter fullscreen mode  
Exit fullscreen mode  
No redundant code.  
Register → Run → Done  
!  
Canceling a Task  
For a one-off, pending task:  
inspector  
.  
CancelTask  
(  
queue  
,  
taskID  
)  
Enter fullscreen mode  
Exit fullscreen mode  
For a periodic task:  
scheduler  
.  
Unregister  
(  
entryID  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Cancel a task with a single line of code.  
Summary: Why Choose  
sasynq  
?  
Simpler  
: Minimalist API design and clear code structure.  
More Powerful  
: Fully covers all common scenarios with support for retries, deduplication, delays, scheduling, and priorities.  
Safer  
: Deadlines, timeouts, and retry strategies provide better control over task processing.  
If you're looking for a  
simple, efficient, and feature-rich  
asynchronous task solution for your Go project,  
sasynq  
is the best choice.  
sasynq  
URL: github.com/go-dev-frame/sponge/pkg/sasynq  
sasynq  
is a sub-component of the Sponge framework. Sponge is a powerful and easy-to-use Go development framework based on the core principle of "Definition is Code." It helps developers easily build stable, reliable, and high-performance backend services (including RESTful API, gRPC, HTTP+gRPC, gRPC Gateway, etc.) in a "low-code" manner.  
👉  
Sponge Project URL  
:  
https://github.com/go-dev-frame/sponge